Azure IN ACTION

Chris Hay
Brian H. Prince

MANNING

www.allitebooks.com

Azure in Action

www.allitebooks.com

www.allitebooks.com

Azure in Action

CHRIS HAY
BRIAN H. PRINCE

MANNING
Greenwich
(74° w. long.)

www.allitebooks.com

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901
Email: orders@manning.com

©2011 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.

Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901

Development editor: Lianna Wlasiuk
Copyeditor: Joan Celmer
Proofreader: Katie Tennant
Illustrator: Martin Murtonen
Designer: Marija Tudor

ISBN: 9781935182481

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 16 15 14 13 12 11 10

www.allitebooks.com

brief contents

PART 1 WELCOME TO THE CLOUD 1

1 ■ Getting to know Windows Azure 3
2 ■ Your first steps with a web role 27

PART 2 UNDERSTANDING THE AZURE SERVICE MODEL............. 49

3 ■ How Windows Azure works 51
4 ■ It's time to run with the service 78
5 ■ Configuring your service 94

PART 3 RUNNING YOUR SITE WITH WEB ROLES...................... 111

6 ■ Scaling web roles 113
7 ■ Running full-trust, native, and other code 139

PART 4 WORKING WITH BLOB STORAGE...............................153

8 ■ The basics of BLOBs 155
9 ■ Uploading and downloading BLOBs 181
10 ■ When the BLOB stands alone 209

PART 5 WORKING WITH STRUCTURED DATA 237

11 ■ The Table service, a whole different entity 239
12 ■ Working with the Table service REST API 265
13 ■ SQL Azure and relational data 296
14 ■ Working with different types of data 315

PART 6 DOING WORK WITH MESSAGES................................ 333

15 ■ Processing with worker roles 335
16 ■ Messaging with the queue 357
17 ■ Connecting in the cloud with AppFabric 379
18 ■ Running a healthy service in the cloud 404

contents

preface xix
acknowledgments xxi
about this book xxiv
author online xxvi
about the authors xxvii
about the cover illustration xxix

PART 1 WELCOME TO THE CLOUD 1

1 *Getting to know Windows Azure* 3

1.1 What's the Windows Azure platform? 4

Windows is in the title, so it must be an operating system 5
Hosting and running applications the Azure way 6

1.2 Building your first Windows Azure web application 9

Setting up your environment 9 ▪ *Creating a new project* 10
Modifying the web page 12 ▪ *Running the web page* 12

1.3 Putting all the Azure pieces together 13

How the load balancer works 13 ▪ *Creating worker roles* 14
How the fabric and the Fabric Controller work 14

1.4 Storing data in the cloud with Azure 15

Understanding Azure's shared storage mechanism 16 ▪ Storing and accessing BLOB data 16 ▪ Messaging via queues 17 Storing data in tables 18

1.5 Why run in the cloud? 18

Treating computing power as a utility service 19 ▪ Simplified data-center management 21

1.6 Inside the Windows Azure platform 23

SQL Server capability in the cloud 23 ▪ Enterprise services in the cloud 25

1.7 Summary 26

2 *Your first steps with a web role 27*

2.1 Getting around the Azure SDK 28

Exploring the SDK folders 29 ▪ Using the Cloud Service project templates 29 ▪ Running the cloud locally 31 ▪ How the local and cloud environments differ 32

2.2 Taking Hello World to the next level 33

Creating the project 33 ▪ Laying down some markup with XHTML and a CSS 34 ▪ Binding your data in the code-behind 36 ▪ Just another place to run your code 37 Configuring the Azure service model 37 ▪ Running the website in the local development fabric 38

2.3 Deploying with the Azure portal 39

Signing up for Azure 39 ▪ The Azure portal 40 ▪ Setting up your service online 41 ▪ Putting on your logging boots 43 Setting up your storage environment 43 ▪ Packaging and deploying your application 45 ▪ Moving to production 47

2.4 Summary 47

PART 2 UNDERSTANDING THE AZURE SERVICE MODEL 49

3 *How Windows Azure works 51*

3.1 The big shift 51

The data centers of yore 52 ▪ The latest Azure data centers 53 How many administrators do you need? 54 ▪ Data center: the next generation 55

3.2 Windows Azure, an operating system for the cloud 56

3.3 The Fabric Controller 57

How the FC works: the driver model 58 ▪ Resource allocation 58 Instance management 59

3.4 The service model and you 59

Defining configuration 59 ▪ Adding a custom configuration element 60 ▪ Centralizing file-reading code 61 ▪ The many sizes of roles 62

3.5 It's not my fault 63

Fault domains 63 ▪ Update domains 63 ▪ A service model example 64

3.6 Rolling out new code 64

Static upgrades 65 ▪ Rolling upgrades 66

3.7 The bare metal 68

Free parking 68 ▪ A special blend of spices 69 ▪ Creating instances on the fly 69 ▪ Image is everything 71

3.8 The innards of the web role VM 72

Exploring the VM details 72 ▪ The process list 74 ▪ The hosting process of your website (WaWebHost) 75 ▪ The health of your web role (RDAgent) 76

3.9 Summary 76

4 *It's time to run with the service 78*

4.1 Using the Windows Azure Service Management API 78

Adding the ServiceRuntime assembly to your application 79 Is your application running in Windows Azure? 80

4.2 Defining your service 81

The format of the service definition file 81 ▪ Configuring the endpoint of your web role 82 ▪ Configuring trust level, instances, and startup action 86 ▪ Configuring local storage 87

4.3 Setting up certificates in Windows Azure 90

Generating a certificate 90 ▪ Adding certificates 91 Configuring your HTTPS endpoint to use the certificate 92

4.4 Summary 93

5 *Configuring your service 94*

5.1 Working with the service configuration file 94

The format of the service configuration file 95 ▪ Configuring standard settings 96 ▪ Configuring runtime settings 98

5.2 Handling configuration at runtime 102

Modifying configuration settings in the Azure portal 102
Tracking service configuration changes 103

5.3 Configuring non-application settings 104

Database connection strings 104 ▪ Application build configuration 104 ▪ Tweakable configuration 104
Endpoint configuration 105

5.4 Developing a common code base 105

Using the RoleEnvironment.IsAvailable property 106
Pluggable configuration settings using inversion of control 107

5.5 The RoleEnvironment class and callbacks 109

5.6 Summary 110

PART 3 RUNNING YOUR SITE WITH WEB ROLES 111

6 *Scaling web roles 113*

6.1 What happens to your web server under extreme load? 113

Web server under normal load 114 ▪ Simulating extreme load 115 ▪ How the web server responds under extreme load 116
Handling increased requests by scaling up or out 117

6.2 How the load balancer distributes requests 118

Multi-instance sample application 119 ▪ The development fabric load balancer 120 ▪ Load balancing in the live environment 124

6.3 Session management 127

How do sessions work? 127 ▪ Sample session application 128
In-process session management 130 ▪ Table-storage session state sample provider 132

6.4 Cache management 135

In-process caching with the ASP.NET cache 135 ▪ Distributed caching with Memcached 136 ▪ Cache extensibility in ASP.NET 4.0 137

6.5 Summary 138

7 *Running full-trust, native, and other code 139*

7.1 Enabling full-trust support 140

7.2 FastCGI in Windows Azure 141

Enabling FastCGI in your local cloud environment 142
Configuring Azure for FastCGI and PHP 142 ▪ Setting up HelloAzureWorld.php 143

7.3 External processes in Windows Azure 146

Spawning a sample process 147 ▪ Using BLOB storage 148

7.4 Calling native libraries with P/Invoke 149

Getting started 150 ▪ Calling into the method 151

7.5 Summary 152

PART 4 WORKING WITH BLOB STORAGE 153

8 ***The basics of BLOBs 155***

8.1 Storing files in a scaled-out fashion is a pain in the NAS 156

Traditional approaches to BLOB management 157 ▪ The BLOB service approach to file management 160

8.2 A closer look at the BLOB storage service 163

Accessing the BLOB (file) 163 ▪ Setting up a storage account 164 ▪ Registering custom domain names 164
Using containers to store BLOBs 166

8.3 Getting started with development storage 167

SQL Server backing store 168 ▪ Getting around in the development storage UI 168

8.4 Developing against containers 169

Accessing the StorageClient library 170 ▪ Accessing development storage 171 ▪ Creating a container 173 ▪ Listing containers 175 ▪ Deleting a container 177

8.5 Configuring your application to work against the live service 178

Switching to the live storage account 178 ▪ Configuring the access key 179

8.6 Summary 180

9 ***Uploading and downloading BLOBs 181***

9.1 Using the REST API 181

Listing BLOBs in a public container using REST 182
Authenticating private requests 185

9.2 Managing BLOBs using the StorageClient library 188

Listing BLOBs using the storage client 189 ▪ Uploading BLOBs 191 ▪ Deleting BLOBs 192

9.3 Downloading BLOBs 193

Downloading BLOBs from a public container 193
Downloading BLOBs from a private container using the storage client 193

9.4 Integrating BLOBs with your ASP.NET websites 195

Integrating ASP.NET websites with table-driven BLOB content 195 ▪ Integrating protected, private content 196

9.5 Using local storage with BLOB storage 199

Using a local cache 199 ▪ Defining and accessing local storage 199
Updating your HTTP handler to use local storage 200
Checking properties of a BLOB without downloading it 201
Improving your handler to check the last modified time 202
Adding and returning custom metadata 203

9.6 Copying BLOBs 204

Copying files via the StorageClient library 206

9.7 Setting shared access permissions 206

Setting shared access permissions on a container 207

9.8 Summary 208

10 *When the BLOB stands alone 209*

10.1 Hosting static HTML websites 209

Creating a static HTML website 210 ▪ Publishing your website to BLOB services 212

10.2 Hosting Silverlight applications in BLOB storage 215

Hosting the Silverlight Spectrum emulator 215 ▪ Communicating with third-party sites 217

10.3 Using BLOB storage as a media server 223

Building a Silverlight or WPF video player 224 ▪ A WPF-based adaptive-streaming video player 225 ▪ A Silverlight-based chunking media player 228

10.4 Content delivery networks 232

What's a CDN? 232 ▪ CDN performance advantages 233
Using the Windows Azure CDN 234

10.5 Summary 236

PART 5 WORKING WITH STRUCTURED DATA 237

11 *The Table service, a whole different entity* 239

11.1 A brief overview of the Table service 240

11.2 How we'd normally represent entities outside of Azure 241

How we'd normally represent an entity in C# 241 ▪ *How we'd normally store an entity in SQL Server* 242 ▪ *Mapping an entity to a SQL Server database* 243

11.3 Modifying an entity to work with the Table service 244

Modifying an entity definition 244 ▪ *Table service representation of products* 245 ▪ *Storing completely different entities* 247

11.4 Partitioning data across lots of servers 249

Partitioning the storage account 249 ▪ *Partitioning tables* 250

11.5 Developing with the Table service 252

Creating a project 252 ▪ *Defining an entity* 253 ▪ *Creating a table* 253

11.6 Doing CRUDy stuff with the Table service 256

Creating a context class 257 ▪ *Adding entities* 258 *Listing entities* 260 ▪ *Deleting entities* 261 ▪ *Updating entities* 263

11.7 Summary 264

12 *Working with the Table service REST API* 265

12.1 Performing storage account operations using REST 266

Listing tables in the development storage account using the REST API 266 ▪ *Deleting tables using the REST API* 269 *WCF Data Services and AtomPub* 270 ▪ *Creating a table using the REST API* 271

12.2 Authenticating requests against the Table service 273

Shared Key authentication 273 ▪ *Shared Key Lite authentication* 274

12.3 Modifying entities with the REST API is CRUD 275

Inserting entities 275 ▪ *Deleting entities* 277 ▪ *Updating entities* 279

12.4 Batching data 281

Entity group transactions 282 ▪ *Retries* 284

12.5 Querying data 284

Retrieving all entities in a table using the REST API 285 ▪ Querying with LINQ 288 ▪ Filtering data with the REST API 288 ▪ Filtering data with LINQ 290 ▪ Selecting data using the LINQ syntax 292 ▪ Paging data 294

12.6 Summary 295

13 *SQL Azure and relational data 296*

13.1 The march of SQL Server to the cloud 297

13.2 Setting up SQL Azure 297

Creating your database 298 ▪ Connecting to your database 299

13.3 Size matters 300

Partitioning your data 301 ▪ Sharding your data for easier scale 302

13.4 How SQL Azure works 303

SQL Azure from a logical viewpoint 304 ▪ SQL Azure from a physical viewpoint 304

13.5 Managing your database 305

Moving your data 305 ▪ Controlling access to your data with the firewall 307 ▪ Creating user accounts 308

13.6 Migrating an application to SQL Azure 309

Migrating the traditional way 309 ▪ Migrating with the wizard 310

13.7 Limitations of SQL Azure 311

13.8 Common SQL Azure scenarios 312

Far-data scenarios 312 ▪ Near-data scenarios 313 SQL Azure versus Azure Tables 313

13.9 Summary 314

14 *Working with different types of data 315*

14.1 Static reference data 316

Representing simple static data in SQL Azure 316 ▪ Representing simple static data in the Table service 318 ▪ Performance disadvantages of a chatty interface 320 ▪ Caching static data 321

14.2 Storing static reference data with dynamic data 323

Representing the shopping cart in SQL Azure 323 ▪ Partitioning the SQL Azure shopping cart 324 ▪ Representing the shopping cart's static data in the Table service 326

14.3 Joining dynamic and infrequently changing data together 329

Duplicating data instead of joining 329 ▪ Client-side joining of uncached data 330

14.4 Summary 331

PART 6 DOING WORK WITH MESSAGES 333

15 *Processing with worker roles 335*

15.1 A simple worker role service 336

No more Hello World 337

15.2 Communicating with a worker role 338

Consuming messages from a queue 339 ▪ Exposing a service to the outside world 340 ▪ Inter-role communication 344

15.3 Common uses for worker roles 345

Offloading work from the frontend 345 ▪ Using threads in a worker role 347 ▪ Simulating worker roles in a web role 347 State-directed workers 349

15.4 Working with local storage 353

Setting up local storage 353 ▪ Working with local storage 354

15.5 Summary 355

16 *Messaging with the queue 357*

16.1 Decoupling your system with messaging 358

How messaging works 358 ▪ What is a message? 360 What is a queue? 361 ▪ StorageClient and the REST API 362

16.2 Working with basic queue operations 363

Get a list of queues 364 ▪ Creating a queue 365 ▪ Attaching metadata 365 ▪ Deleting a queue 366

16.3 Working with messages 366

Putting a message on the queue 367 ▪ Peeking at messages 367 Getting messages 368 ▪ Deleting messages 368

16.4 Understanding message visibility 369

About message visibility and invisibility 369 ▪ Setting visibility timeout 370 ▪ Planning on failure 370 ▪ Use idempotent processing code 370

16.5 Patterns for message processing 371

Shared counters 371 ▪ Work complete receipt 373 ▪ Asymmetric queues versus symmetric queues 373 ▪ Truncated exponential backoff 374 ▪ Queue creation on startup 376 ▪ Dynamic queues versus static queues 376 ▪ Ordered delivery 376 ▪ Long queues 377 ▪ Dynamically scaling to meet queue demand 377

16.6 Summary 378

17 *Connecting in the cloud with AppFabric 379*

17.1 The road AppFabric has traveled 380

The two AppFabrics 380 ▪ Two key AppFabric services 380

17.2 Controlling access with ACS 381

Identity in the cloud 381 ▪ Working with actors 382 Tokens communicate authorization 383 ▪ Making claims about who you are 384

17.3 Example: A return to our string-reversing service 385

Putting ACS in place 385 ▪ Reviewing the string-reversal service 387 ▪ Accepting tokens from ACS 388 ▪ Checking the token 389 ▪ Sending a token as a client 390 ▪ Attaching the token 392 ▪ Configuring the ACS namespace 392 ▪ Putting it all together 396

17.4 Connecting with the Service Bus 397

What is a Service Bus? 397 ▪ Why an ESB is a good idea in the cloud 398

17.5 Example: Listening for messages on the bus 400

Connecting the service to the bus 400 ▪ Connecting to the service 401

17.6 The future of AppFabric 402

17.7 Summary 402

18 *Running a healthy service in the cloud 404*

18.1 Diagnostics in the cloud 405

Using Azure Diagnostics to find what's wrong 405 Challenges with troubleshooting in the cloud 406

18.2 Diagnostics in the cloud is just like normal (almost) 406

Managing event sources 407 ▪ It's not just for diagnostics 408

18.3 Configuring the diagnostic agent 409

Default configuration 411 ▪ Diagnostic host configuration 411 The other data sources 416 ▪ Arbitrary diagnostic sources 418

18.4 Transferring diagnostic data 419

Scheduled transfer 419 ▪ On-demand transfer 420

18.5 Using the service management API 421

What the API doesn't do 422 ▪ Setting up the management credentials 422 ▪ Listing your services and containers 424 Automating a deployment 426 ▪ Changing configuration and dynamically scaling your application 430

18.6 Better together for scaling 432

The thermostat 433 ▪ The control system 434 ▪ Risks and managing them 434 ▪ Managing service health 435

18.7 Summary 436

index 437

preface

Both of us have a passion for cloud computing and Windows Azure, and in this book we'd like to share with you what we've learned from working with the technology. We want to show you how to get the most out of Azure and how to best use the cloud.

Writing a book is a far more complex project than either of us expected, involving a lot of people, a lot of collaboration, and plenty of late nights hunched over a keyboard. We did it because we wanted to help you understand what happens inside Azure, how it works, and how you can leverage it as you work with your applications. We wanted to show you not only how to run your complete system in the cloud, but all the other ways you can leverage the cloud, specifically by using hybrid applications and distributed applications.

As we worked with all sorts of developers in our day jobs, we knew they could easily learn how to use the cloud, but they were all scared. We hadn't seen people so afraid of a new technology that could help so much since web services came onto the scene years ago. We knew if developers would take a minute to play with Azure just a little bit, it would become less scary and more approachable. Ultimately, we wanted this book to answer the question, "What can Azure do, and why do I care?" We hope we've succeeded.

We've leveraged a lot of resources to write this book, and you might have been one of them. We worked in forums, we worked with other cloud techs, we crawled through every scrap of public Azure information we could find (even obscure blog posts in the dark corners of the internet), and we had personal conversations with Azure team members and anyone else we could get to take our calls. We leveraged our own experience and insight. Sometimes we guessed at how things work based on how we would

have built Azure, and then pushed Microsoft to give us more details to see if we were right. We wrote a lot of code, and tried out ideas that we would get asked about at conferences, in forums, over email, and as responses to our articles.

The rest is history, with about a year of writing, rewriting, reviews, intense discussion, and coding. We faced two big challenges as a writing team. The first was PDC 2009. We knew that would be the coming-out party for Windows Azure with its official 1.0 release, and that a lot of what we had written up to that point would change. This involved rewriting most of our code, retaking all our screen shots, and changing a lot of our text. The second challenge was the time zone differences between us. With up to fourteen time zones separating us at times, our combined travel schedules exacerbated the time zone challenge. Much of this book was written in airports, hotels, at conferences, during late weekend hours, and at every other conceivable time and place.

Windows Azure was released for commercial availability on February 1, 2010, and by all accounts has been a huge success. Microsoft won't publicly state how many applications have been deployed to Azure, but you can infer some trends from the case studies and press releases they make available. It looks like tens of thousands of applications (from small test apps to major internet-scale applications) have been deployed to Azure globally. The Azure teams ship new features about every 2–3 months. As a developer, it's exciting to see so much innovation coming out of Microsoft on a platform you use. It's gratifying to see the features that customers have asked for being deployed.

For book authors, the pace can be a little grueling, with things changing in the technology all the time, but maybe that just sets us up for a second edition. We hope you enjoy the book.

acknowledgments

We would like to thank all the people who helped us during the writing process; their input made this a much better book. First on the list is our amazing editor at Manning, Lianna Wlasiuk. She showed an endless amount of patience and had a seemingly inexhaustible supply of the proverbial red ink. Her feedback and guidance turned these cloud geeks into writers.

Secondly, a big thanks to Mike Stephens. He's a great guy who did an amazing job in shaping this project. We'd also like to thank our publisher Marjan Bace for his insight and vision. Those early conversations with him helped us go in the right direction. And thanks to Christina Rudhoff for kicking off the book in the first place, and to Mary Piergies for her management of the production process. You guys are awesome.

We would also like to thank the other staff at Manning. While any author can ship a book, Manning knows that shipping a great book is a team sport, and they have an excellent team in place. Their constant support and guidance—and the challenge to push the book further—are greatly appreciated.

There's another group of people who were key to making this book successful, the group of reviewers that read the manuscript four or five times over the past year, pointing out weak parts of the story, plot holes, and places where better code samples could be provided. We'd like to thank James Hatheway, Alex Thissen, Scott Turner, Darren Neimke, Christian Siegers, Margriet Bruggeman, Nikander Bruggeman, Eric Nelson, Ray Booysen, Jonas Bandi, Frank Wang, Wade Wegner, Mark Monster, Lester Lobo, Shreekanth Joshi, Berndt Hamboeck, Jason Jung, and Kunal Mittal.

Special thanks to Michael Wood who served as the technical proofreader of the book, reviewing it again shortly before it went to press and testing the code. We couldn't have done it without you.

Our early readers, people who bought the book through the Early Access program, before it was even done, were a big help too. They suffered through drafts, impartial chapters, and early cuts of code. Their feedback in the forums was critical to where we went with the book.

Chris Hay

I don't want this to sound like an Oscar acceptance speech (boo hoo, I want to thank my goldfish, blah blah blah), but it's gonna be a little like that as I really do want to call out a few folks. I guess I lose my right to laugh at those blubbering celebrities in the future.

The biggest thanks of all go to my wonderful wife, who woke up one morning to discover that due to the UK/US time zone difference, I had negotiated a book deal whilst she was sleeping. In spite of this, she gave me her full support, without which this book would never have happened. She is totally awesome and I love her very much. Thank you, Katy, for being so cool and supportive.

I want to apologize to my dogs (Sascha and Tufty) for the impact on their walking time and thank them for distracting me when I got bogged down with too much work. They brought me their bouncy balls and even figured out how to shut down my computer.

Big thanks to my parents and my brother (please don't read anything into the order of thanks; you really don't come after the dogs). Thanks for the great start in life, especially buying me that ZX81 when I was 4 years old.

Thanks to Nathan for being my sounding board; truly appreciated it, dude.

Thanks to Brian and Michael for doing the production work on the book while I was working 18-hour days in India. You guys are awesome, thank you.

Santa Claus, thank you for bringing me presents every year, and Tooth Fairy, thank you for making tooth loss more bearable.

I'd like to thank all the guys at NxtGenUG (especially Rich, Dave, John, and Allister) for their support. P.S. If you have never gone to a .NET User Group then be sure to do so—it's a lot of fun. Big thanks to the UK/US community in general (you guys know who you are, thank you).

Also thanks to Girls Aloud, the Pussycat Dolls, and Alesha Dixon for making cool music and helping me keep my sanity throughout the writing process. And if you are reading this book, then something has gone wrong with the universe which will require The Doctor to fix.

Finally, thanks to you, dear reader, for buying the book. I love you, kiss, kiss, kiss, boo hoo, wah wah ;)

Brian H. Prince

I started learning how to write code when I was ten. My parents were supportive and understanding when they figured out that their middle son wasn't normal, that he was a geek. Back then, geeks hadn't risen to their current social prominence. They picked me up after work from UMF, and they didn't kick me out of the house after I caused a small electrical fire while trying to control the box fan in my room with my CoCo 3. Thanks, Mom and Dad. A few years later, one of my aunts suggested I stick with computers as I grew up. She expected they would be important in the future. That sounds like a trivial prediction today, but back then, it seemed like something out of Nostradamus's writings.

I also want to thank everyone at Microsoft for their encouragement, including my manager, Brian, who supported me in the extra work that writing a book takes.

Above all, I owe a tremendous debt to my family. My kids, Miranda and Elliot, kept me from totally disappearing into my office for 10 months with regular forced breaks. Elliot would come in and declare a 15-minute recess to go and play Xbox with him. Miranda would come in and write cute notes of support on my whiteboard or tell me about that latest book she was reading. Thanks kids, you're the best!

But the one person I owe the most to is my beautiful wife. She kept me motivated; she gave me the time and quiet to write when I needed to write and the push to take a break when I needed to release pressure. I'd heard rumors about how hard it is to live with an author in the house from friends who gave me advice along the way (thanks Bill, Jim, and Jason). Without her I wouldn't have been able to complete this huge project. She spent hours helping me simplify the story, revise the approaches, and dream up segues. Joanne, I would not be without you, and without you I would not be.

about this book

This book will teach you about Windows Azure, Microsoft's cloud computing platform. We'll cover all aspects and components of Windows Azure from a developer's point of view.

The book is written from the perspective of a .NET developer who's using C#. We feel that most developers using Azure will be using .NET. Everything in this book applies to any platform that uses Azure. You'll need to use the appropriate SDK for your development tools and platform of choice.

You should be fairly familiar with .NET, but you don't have to be an expert. We expect a developer with a few years of experience to be able to get the most out of this book. Someone new to development, or perhaps even a manager, can still read the book to get a grasp of the broad concepts of Azure. If that's your situation, skip over the code samples and try to understand what the moving parts are.

Roadmap

This book is broken into six parts, each with its own focus.

Part 1 is titled "Welcome to the cloud" and that's exactly what it is: a welcome to the world of cloud computing. Chapters 1 and 2 explain what cloud computing is, and what the big moving parts of Windows Azure are. You'll build and deploy some simple applications in this part, just to whet your appetite.

Part 2 is called "Understanding the Azure service model." Chapter 3 gives you a peek behind the curtain and shows you how Azure works. Chapters 4 and 5 cover how to run and configure your applications in Azure.

Part 3, "Running your site with web roles," covers running web applications in Azure. This part includes chapter 6, which describes scaling your application, and chapter 7, which covers using native code in Azure.

Part 4 is called "Working with BLOB storage," and covers the first part of Windows Azure storage, BLOBs. Chapter 8 discusses the conceptual basics of BLOBs, chapter 9 covers how to work with them in your code, and chapter 10 tells you when to use BLOBs outside Azure.

Part 5, "Working with structured data," tells you all about Windows Azure tables and SQL Azure. Chapters 11 and 12 focus on tables, chapter 13 dives into SQL Azure, and chapter 14 takes a broader look at how to work with data in the cloud and how to make decisions on what strategies to use.

Part 6, titled "Doing work with messages," covers the last several parts of Azure, including specialized aspects of using worker roles, which is detailed in chapter 15. We discuss working with queues in chapter 16. Connecting your applications together and securing your services are delved into in chapter 17. Finally, chapter 18 describes how to work with diagnostics and how to manage your infrastructure in the cloud.

About the source code

All source code in listings or in text is in a `fixed-width font like this` to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

Source code for all working examples in this book is available for download from the publisher's website at www.manning.com/AzureinAction.

To work with the sample code in this book, you'll need Windows Vista, Windows 7, or Windows Server 2008. You'll also need either Visual Studio 2008 or 2010. We used VS2010 in this book for samples and screen shots. Additionally, you need to install the Azure SDK and the AppFabric SDK. Both of these can be found at Azure.com.

Author Online

The purchase of *Azure in Action* includes free access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. You can access and subscribe to the forum at www.manning.com/AzureinAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and authors can take place. It isn't a commitment to any specific amount of participation on the part of the authors, whose contributions to the book's forum remain voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Chris Hay is a Microsoft MVP in Client App Dev, an international conference speaker, and cofounder of a .NET usergroup in Cambridge, UK (http://nxtgenug.net/). He has spent part of the past year working and living in India. Brian H. Prince is an Architect Evangelist for Microsoft, cofounder of the nonprofit organization CodeMash (www.codemash.org), and a speaker at various regional and national technology events. He lives in Westerville, Ohio. In their own words, here's what they say about how they came to Azure.

CHRIS HAY

My day job involves building some of the largest m-commerce systems in the world. When Microsoft announced Windows Azure to the world at the Professional Developers Conference in Los Angeles in 2008, I immediately thought of how I could use the cloud as part of the systems I was actively building.

Of all of the key scenarios for using the cloud, dynamic scaling is one of the most well-known. I was hoping that the promise of massive numbers of servers and a simplified platform would be able to meet my enormous scale needs, while making it easier to build large-scale systems. Azure offered the promise of being able to deploy an application into the cloud and have an automated deployment and provisioning system, with a complete abstraction of the underlying physical infrastructure. This book is focused on exploring those promises, and seeing how they worked out.

Coupling this newfound passion with my long-held desire to someday write a book, I settled down to write the proposal that I would send to Manning, pitching my idea for a book titled *Azure in Action*. And a year later, here it is!

BRIAN H. PRINCE

While working for Microsoft in recent years, I found myself spending more and more of my time focusing on Windows Azure (or Red Dog, as it was called internally at Microsoft at the time) and cloud computing. I was already at work on another *In Action* book when I made a comment in one of my many meetings with Manning that I was surprised they weren't planning a book for each piece of the upcoming Microsoft cloud platform. Thinking that writing my first book ever wasn't enough work, I further commented that I would love to get involved and help with the Azure book.

This simple comment initiated a lot of work for the editors at Manning as they started looking for experienced authors who could write a series of books on Microsoft's cloud platform. They approached me to see if I would pitch in and help write *Azure in Action* with Chris. I agreed, and after a few chats with Chris over Skype, we finalized the draft table of contents and submitted it to Manning. The rest is history and you are now holding that book in your hands.

about the cover illustration

The figure on the cover of *Azure in Action* is captioned "Woman with child from Durdevac." The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

The village of Durdevac is near the town of Osijek in Slavonia, a geographical and historical region in eastern Croatia. Women in Slavonia were known for their intricate embroidery and sewing skills, and everything they wore was made by hand requiring the weaving of textiles and dyeing of wool. Slavonian women typically wore long white skirts and white linen shirts with a collar, topped with long black and brown vests embroidered along the edges in wool of different colors, with white headscarves and necklaces made of red coral beads. The long aprons that completed the traditional costume were elaborately embroidered with colorful patterns of flowers or geometric designs.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few

miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Welcome to the cloud

Part 1 is all about dipping your toes into the water and getting ready to dive in headfirst.

We cover what Azure is in chapter 1—what the moving parts are, and why people are so excited about cloud computing.

We throw you in the deep end of the pool in chapter 2, building and deploying—step-by-step—your first cloud application. We've all written Hello, World apps; after you've read part 1, you'll begin to see how you can easily scale them to hundreds of servers.

1 Getting to know Windows Azure

This chapter covers

- Overview of Windows Azure
- Building your first Windows Azure web role
- Windows Azure infrastructure
- How Windows Azure implements core cloud concepts
- Flagship Windows Azure platform services

Imagine a world where your applications were no longer constrained by hardware and you could consume whatever computing power you needed, when you needed it. More importantly, imagine a world where you paid only for the computing power that you used.

Now that your imagination is running wild, imagine you don't need to care about managing hardware infrastructure and you can focus on the software that you develop. In this world, you can shift your focus from managing servers to managing applications.

If this is the sort of thing you daydream about, then you should burn your server farm and watch the smoke form into a cloud in the perfect azure sky. Welcome to the cloud, and welcome to Windows Azure. We also suggest that if this is the sort of thing you daydream about, you might want to lie to your non-IT friends.

We'll slowly introduce lots of new concepts to you throughout this book, eventually giving you the complete picture about cloud computing. In this chapter, we'll keep things relatively simple. As you get more comfortable with this new paradigm, and as the book progresses, we'll introduce more of Azure's complexities. To get the ball rolling, we'll start by looking at the big Azure picture: the entire platform.

1.1 What's the Windows Azure platform?

As you might have already gathered, the Windows Azure platform encompasses Microsoft's complete cloud offering. Every service that Microsoft considers to be part of the cloud will be included under this banner. If the whole cloud thing passed you by, there isn't really anything magical about it. The *cloud* refers to a bunch of servers that host and run your applications, or to an offering of services that are consumed (think web service).

The main difference between a cloud offering and a noncloud offering is that the infrastructure is abstracted away—in the cloud, you don't care about the physical hardware that hosts your service. Another difference is that most public cloud solutions are offered as a metered service, meaning you pay for the resources that you use (compute time, disk space, bandwidth, and so on) as and when you use them.

Based on the Azure release announced in November 2009 at the Professional Developers Conference (PDC) held in Los Angeles, the Windows Azure platform splits into the three parts shown in figure 1.1: Windows Azure, SQL Azure, and the Windows Azure platform AppFabric. You can expect the parts included in the platform to

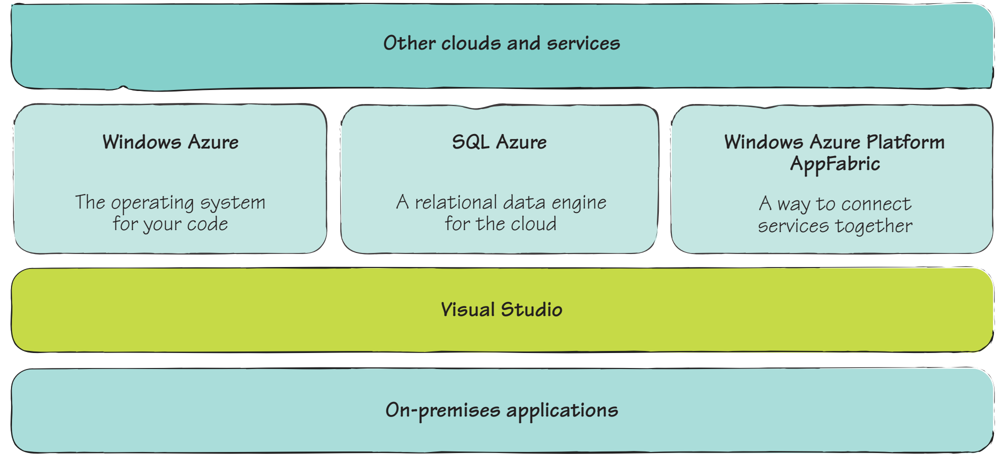


Figure 1.1 The parts that make up the Windows Azure platform include the Windows Azure operating system, SQL Azure, and AppFabric.

increase over time; in fact, we wouldn't be surprised to see Microsoft Flight Simulator in the cloud.

As cool as AppFabric and SQL Azure are, for now we're going to stay focused on the Windows Azure part of the Windows Azure platform and ignore all the other platform-specific stuff until the end of the chapter. Talking about Windows Azure immediately gets a little confusing. Unfortunately, when most folks refer to Windows Azure, it's not clear whether they're referring to the Windows Azure platform, the complete cloud offering, or to Windows Azure, which is a part of the platform.

It's kind of like the ESPN naming convention. The ESPN Network has multiple channels (ESPN, ESPN2, ESPN News, and so on), yet we tend to refer to these channels collectively as ESPN rather than as the ESPN Network. To confuse matters further, we also refer to the individual ESPN channel as ESPN, also. If you ask someone what game is on ESPN tonight, it's not clear if you mean all the channels on ESPN (including ESPN News and ESPN2) or if you mean just the channel named ESPN (not including ESPN2 and the others).To keep things consistent, whenever we talk about the platform as a whole, we'll refer to the *Windows Azure platform* or *the platform*; but if we're talking about the core Windows Azure product, then we'll use the term *Windows Azure*, or just *Azure*.

So, what exactly is Windows Azure? Microsoft calls Azure its core operating system for the cloud. OK, so now you know what Windows Azure is, and we can skip on, right? Not so fast! Let's break it down, strip away all the hype, and find out what Azure is all about.

1.1.1 Windows is in the title, so it must be an operating system

Windows Azure is an operating system that provides the ability to run applications in a highly scalable manner on Microsoft servers, in Microsoft's data centers, in a manageable way. You can host either your web applications, such as a website that sells Hawaiian shirts, or backend processing services, such as an MP3-to-WMA file converter, in Microsoft's data centers.

If you need more computing power (more instances of your website or more instances of your backend service) to run your application, you can allocate more resources to the application, which are then spread across many servers. By increasing the number of resources to your application, you'll ultimately be able to process more data or handle more incoming traffic.

Hmmm…how exactly is that an operating system? To answer that question, we have to define what it means to be a *cloud operating system*.

When Microsoft refers to Windows Azure as an operating system for the cloud, it doesn't literally mean an operating system as you might know it (Windows 7, Windows Vista, Leopard, Snow Leopard, and so on). What Microsoft means is that Windows Azure performs jobs that are similar to those that a traditional operating system might perform. What does an operating system do? Well, it has four tasks in life:

- Host and run applications
- Remove the complexities of hardware from applications

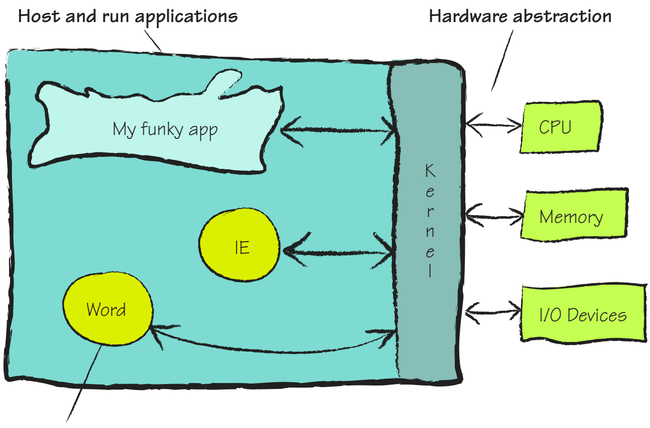


Figure 1.2 A typical representation of an operating system interacting with applications and resources. Notice that applications don't directly interact with CPU, memory, or I/O devices.

- Provide an interface between users and applications
- Provide a mechanism that manages what's running where and enforces permissions in the system

Figure 1.2 shows how a traditional operating system achieves these tasks in a typical PC environment.

The applications shown in figure 1.2 are running within an operating system. The applications don't have direct access to the hardware; all interactions must come through the kernel, the low-level operating system component that performs all the tasks we're discussing: processing, memory management, and device management. We'll look at how some components of Windows Azure fill the role of the kernel in the cloud later in this chapter.

The analogy of Windows Azure being an operating system looks like it could work out after all. Over the next few sections, we'll use this analogy to see how Windows Azure fares as an operating system, which will give you a good overview of how Windows Azure works and what services it provides.

1.1.2 Hosting and running applications the Azure way

Hosting and running applications might be the most important task of an operating system. Without applications, we're just moving a mouse around with no purpose. Let's look at the types of applications that can be run in both traditional operating systems and in Windows Azure.

TYPES OF APPLICATIONS: WHAT'S IN A NAME?

In a traditional operating system, such as Windows 7, we can consider most of the following to be applications:

- Microsoft Word (yep, it's an app)
- Internet Explorer or Firefox (still an app)
- Killer Mutant Donkey Zombie Blaster game (even that's an app)

Remember those applications running in the context of a typical PC operating system in figure 1.1? Instead of hosting client applications (games, Word, Excel, and so on), the types of applications that you host in Windows Azure are server applications, such as web applications (for example, a Hawaiian Shirt Shop website) or background computational applications (for example, an MP3 file converter).

Figure 1.3 shows these server applications running in a traditional operating system.

Turns out (see figures 1.2 and 1.3) that there's no real difference between Microsoft Excel and a Hawaiian Shirt Shop website. As far as a traditional operating system is concerned, they're both applications.

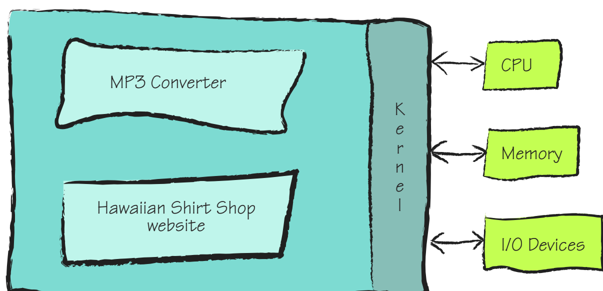


Figure 1.3 Windows Azure–type applications running in a traditional OS. Azure applications function in an OS the same way that traditional applications do.

RUNNING APPLICATIONS ACROSS THOUSANDS OF SERVERS

The traditional operating system is responsible for allocating CPU time and memory space that allows your application to run (as seen in both figures 1.1 and 1.2). Not only is the operating system responsible for allocating these resources, but it's also responsible for managing these resources. For example, if an application fails, then it's the operating system's job to clean up the application's resource usage and restart the application, if necessary. This level of abstraction is perfect for an operating system that manages a single server, but it isn't scalable when it comes to a cloud operating system. With Windows Azure, your application doesn't necessarily run on a single server; it can potentially run in parallel on thousands of servers.

A cloud operating system can't be responsible for allocating CPU time and memory on thousands of physically separate servers. This responsibility has to be abstracted away from the OS. In Windows Azure, that responsibility is given to *virtual machines* (VMs). Figure 1.4 shows how your applications might be distributed among the VMs in a Windows Azure data center.

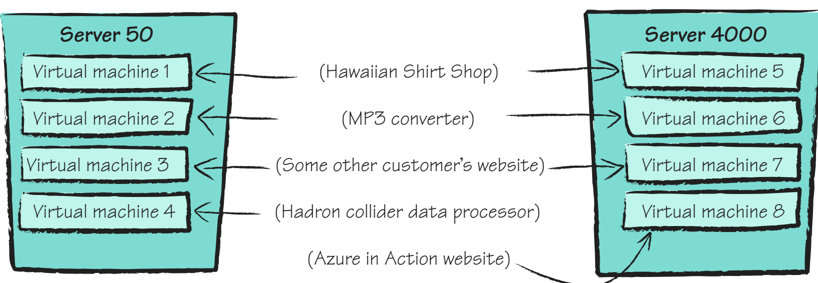


Figure 1.4 Applications split across many VMs in a Windows Azure data center

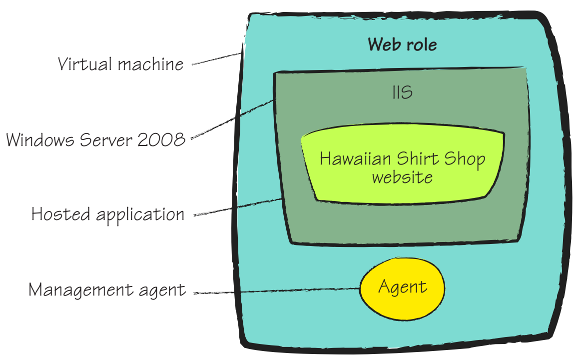
Your cloud operating system is no longer responsible for assigning your applications' resources by CPU and memory, but is instead responsible for allocating resources using VMs. Windows Azure uses VMs to achieve separation of services across physical servers. Each physical server is divided into multiple VMs. An application from another customer on the same physical hardware as yours won't interfere with your application.

In figure 1.4, the Hawaiian Shirt Shop website is allocated across two VMs (VM1 and VM5), which are hosted on two different physical servers (server 50 and server 4000), whereas the *Azure in Action* website is allocated only a single VM (VM8) on server 4000 (shirt shops make more money, so they get more resources).

Let's drill down and take a closer look at what constitutes a VM.

ANATOMY OF A VIRTUAL MACHINE

Figure 1.5 shows what the VM hosting a web application looks like.



The physical server is split up into one or more VMs. Every instance of your service (web role or worker role) is installed onto its own VM, which is a base installation of Windows Server 2008 (with some extra Azure bits). The VM hosts the web application within Internet Information Services (IIS) 7.0.

Although your application runs on a VM, the VM is abstracted away from you, and you only have a view of the role instance, never of the VM. A single instance of your web application is assigned to a single VM, and no other applications will be assigned to that VM. In this way, every instance of your web application is isolated from other applications running on the same physical server. The VM image also runs an agent process. We'll explain what this agent does in chapter 3 when we discuss the Red Dog Agent.

Web role and worker role

A role is another name for your application. The role refers to the base VM image that hosts your application. A web role is a VM that hosts your application within IIS. A worker role is the same as a web role, but without IIS. It's intended for typical backend processing workloads.

To be honest, we're now itching for some code. Let's look at how you can build a simple ASP.NET website that you can run in one of those Windows Azure VMs. Don't worry; we'll continue dissecting Windows Azure after you get your hands dirty with a little code.

1.2 Building your first Windows Azure web application

Although you're going to build an ASP.NET website in this example, the good news is that almost any website that can currently be hosted in IIS on Windows Server 2008 can be hosted in Windows Azure.

The following are examples of the types of web applications Azure supports out of the box:

- ASP.NET 3.5 web applications
- ASP.NET MVC 1.0, 2.0 web applications
- Web services (WCF, ASMX)
- Any FastCGI-based website such as PHP or Python
- Java and Ruby applications

Although Windows Azure supports the ability to host different types of websites, for now you'll create a simple Hello World web application using ASP.NET 3.5 SP1. In chapters 7 and 15, we'll look at how you can create PHP websites, WCF Web Services, and ASP.NET MVC websites.

To get started developing an ASP.NET 3.5 SP1 website, you'll need to download the Windows Azure software development kit (SDK).

1.2.1 Setting up your environment

The SDK contains a whole bunch of things that'll make your life easier when developing for Windows Azure, including the following:

- Windows Azure development fabric (a simulation of the live fabric)
- Visual Studio templates for creating web applications
- Windows Azure storage environment
- Deployment tools
- A glimpse of a bright new world

In chapter 2, we'll take a deeper look at some of the items in the SDK. For now, you'll just install it. If you're an experienced ASP.NET developer, you should be able to install the SDK by clicking the Next button a few times. You can grab the SDK from www.Azure.com.

Before installing the SDK, check your version of Windows and Visual Studio. A local instance of some flavor of SQL Server (either Express, which is installed with Visual Studio, or full-blown SQL Server) is required to use the SDK. We'll explain this in more depth in chapter 9.

SUPPORTED OPERATING SYSTEMS

Before you attempt to install the SDK, make sure that you have a suitable version of Windows. Supported versions of Windows currently include the following:

- Windows 7 (which you should be running because it's lovely)
- Windows Vista
- Windows Server 2008 (and beyond)

> **NOTE** Windows XP isn't supported by Windows Azure. Before you jump up and down about Windows XP, there isn't some conspiracy against it. XP isn't supported because Windows Azure web roles are heavily built on IIS 7.0. Windows XP and Windows 2003 use earlier versions of IIS that won't work with Windows Azure.

SUPPORTED VERSIONS OF VISUAL STUDIO

To develop Windows Azure applications in Visual Studio, you'll need either Visual Studio 2008 or Visual Studio 2010. If you're still running Visual Studio 2005, you now have the excuse you need to upgrade. If for some reason you can't get Visual Studio or your company won't upgrade you, then you can use the Web Express versions of either Visual Studio 2008 or 2010 for free, or you can use Visual Studio 2008. We'll be using Visual Studio 2010 throughout this book. The windows and dialog boxes shown in the figures might differ slightly from those in Visual Studio 2008 or the Express Edition, but, all in all, it works in the same way.

STARTING VISUAL STUDIO

To launch your Windows Azure application in the development fabric from Visual Studio, you need Administrator privileges. Get into the habit (for Azure development) of right-clicking your Visual Studio icon and selecting Run as Administrator.

Now we'll help you create your first Azure web application.

1.2.2 *Creating a new project*

Your first step is to create a new project. Open Visual Studio and select File > New > Project. Select the Cloud Service project type, which gives you the option to select the Cloud Service template, as shown in figure 1.6.

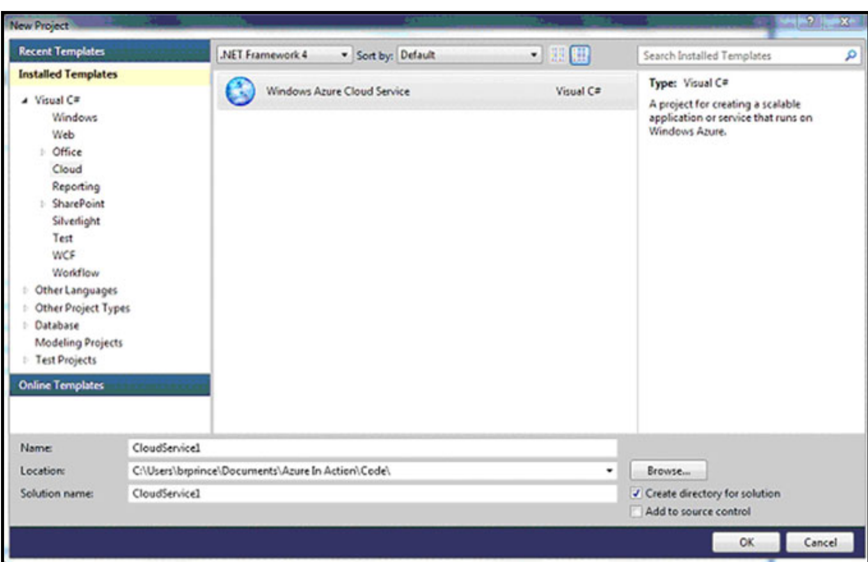


Figure 1.6 The Cloud Service template in the New Project dialog box of Visual Studio 2010

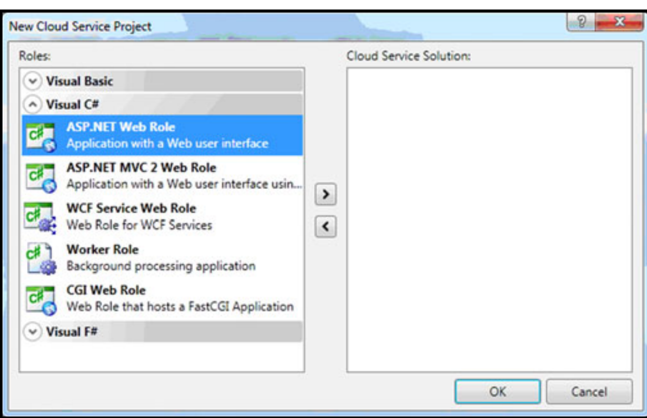


Figure 1.7 New Cloud Service Project dialog box. From here, you can add several Azure projects to your solution.

After you select the Cloud Service template, enter a name for your project and solution, and then click OK. The dialog box shown in figure 1.7 opens, in which you select the type of Windows Azure project that you want to create.

You can create the following types of roles:

- ASP.NET web roles
- ASP.NET MVC 2 web roles
- WCF service web roles
- Worker roles
- CGI-based web roles

You can create your projects in either Visual Basic or C#. In this book, we use C# rather than Visual Basic. This is no disrespect to Visual Basic; we've found over time that although C# developers typically aren't comfortable with Visual Basic, Visual Basic developers are comfortable with both languages (you have to be though, because most samples are in C#).

Select the ASP.NET Web Role project, and then click the right arrow button to add the project to the Cloud Service Solution panel, as shown in figure 1.8.

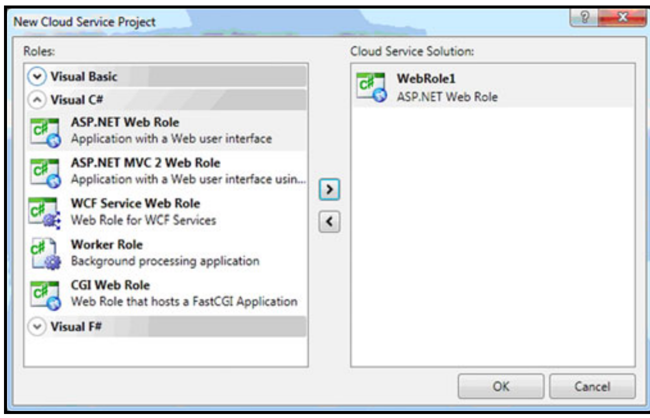
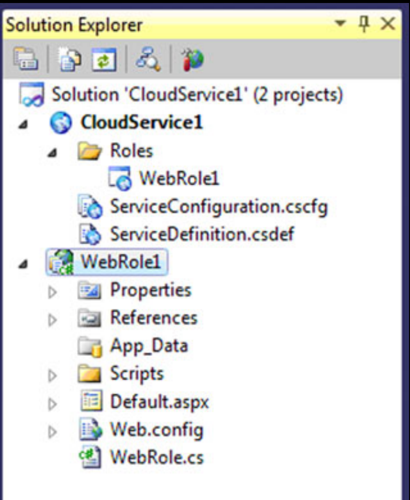


Figure 1.8 Selecting a web role project from the New Cloud Service Project dialog box. Click the default name WebRole1 to change it to something more to your liking.



Now that you've selected your web project, click OK and wait for Visual Studio to generate your solution. After Visual Studio has taken some time to set up your solution and project, it'll have created a new solution for you with two new projects, as shown in figure 1.9.

The first project (CloudService1) contains configuration that's specific to your Windows Azure web role. For now, we won't look at the contents of this project and instead save that for chapter 2. Next, you'll create a simple web page.

Figure 1.9 Solution Explorer for your newly created web role project. The top project (CloudService1) defines your application to Azure. The bottom one (WebRole1) is a regular ASP.NET project with a starter template.

1.2.3 Modifying the web page

The second project (WebRole1) in figure 1.9 is a regular old ASP.NET web application. You can modify the default.aspx file as you would normally. In this case, modify the file to display Hello World, as shown in the following listing.

Listing 1.1 Modifying the default.aspx file to display Hello World

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="WebRole1._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Hello World</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
       Hello World
    </div>
    </form>
</body>
</html>
```

Now that you've created your web page, you can run it in your development fabric.

1.2.4 Running the web page

Before you run your new web role, you must ensure that the cloud service, rather than the ASP.NET project, is your startup project. By default, Visual Studio does this for you

when you create your new project. If the ASP.NET project is the startup project, Visual Studio will run it with the built-in development web server and not the Azure SDK.

Now for the exciting part: you're about to run your first web application in the Windows Azure development fabric. Press F5 as you would for any other Visual Studio application. Visual Studio fires up the development fabric and launches your web page in your browser just like any other web page. Unlike regular ASP.NET web applications, the development fabric hosts your web page rather than the Visual Studio Web Development Server (Cassini). Figure 1.10 shows your web page running in the development fabric.

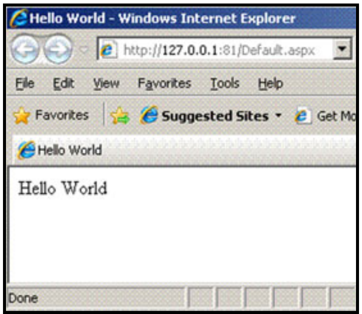


Figure 1.10 ASP.NET 3.5 Hello World running in the development fabric

Congratulations! You've developed your first cloud application. In chapter 2, we'll look in more detail at the development SDK, the development fabric, and how to deploy your service to the live production servers.

Let's return now to our big-picture discussion of Azure.

1.3 Putting all the Azure pieces together

Even Hello World web applications often require multiple instances as the result of the levels of traffic they receive. To understand all that's involved in multiple instances, you first need to understand the Windows Azure logical infrastructure and how it makes it so easy to deploy and run applications in the cloud. As you can see in figure 1.11, the web role is just one piece of the overall infrastructure.

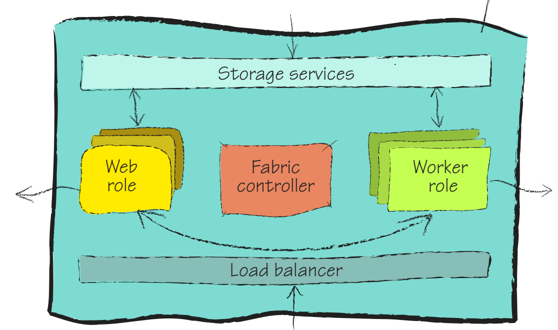


Figure 1.11 The Windows Azure compute infrastructure involves several components. They all work together to run your application.

Over the next couple of sections, we'll look at how the other components—worker roles, the fabric and the Fabric Controller, and the storage services—fit together. First, let's take a closer look at the *load balancer* (the component at the bottom of figure 1.11).

1.3.1 How the load balancer works

Note in figure 1.11 that neither your web roles (web applications) nor your worker roles (background services) have direct incoming traffic from the internet. For both

worker and web roles, all incoming traffic must be forwarded via one or more load balancers. The load balancer provides four important functions, as listed in table 1.1.

Table 1.1 Primary load balancer functions

Function	Purpose
Minimize attack surface area	Improves security
Load distribution	Enables incoming requests to be forwarded to multiple instances
Fault tolerance	Routes traffic to another instance during a fault
Maintenance	Routes traffic to another instance during an upgrade

Not only can a role receive incoming traffic, but roles can also initiate communication with services hosted outside the Windows Azure data centers, with roles inside the data center, and with storage services.

Now that you understand the load balancer's job of distributing requests across multiple instances of web roles, we'll take a brief look at Azure's other type of supported role, the worker role.

1.3.2 Creating worker roles

Worker roles are a lot like web roles and will be covered in depth in chapter 15. The biggest difference is that they lack IIS, which means they can't host a web application, at least not in the traditional sense. Worker roles are best suited for hosting backend processing and a wide variety of web services. These types of servers are often referred to as *application servers* in many IT departments.

At this point, we've explored a few tasks an operating system performs (hosting and running applications). What we haven't explained is how the kernel fits into this analogy of Azure as a cloud operating system. You need something that will manage your applications and all your VMs running in the Windows Azure data center. It's one thing to host an application; it's another to manage what's running and enforce permissions and resource allocation. In a normal operating system, the kernel performs these tasks. In Windows Azure, the kernel is the Fabric Controller (it sits right in the center of figure 1.11).

1.3.3 How the fabric and the Fabric Controller work

Azure contains a massive number of servers, and there isn't any way they can possibly be managed on an individual basis. This is where the Azure operating system concept comes into play. By abstracting away all of those individual servers into a swarm or cloud, you only have to manage the cloud as a whole. This swarm of servers is called the *fabric*, and your applications run in the fabric when you deploy them to the cloud.

The fabric is managed by a software overlord known as the *Fabric Controller*. The Fabric Controller plays the role of the kernel and is aware of every hardware and software

asset in the fabric. It's responsible for installing your web and worker roles onto the physical or virtual servers living in the fabric (this process is similar to how the kernel assigns memory or CPU to an application in a traditional operating system). The Fabric Controller is responsible for maintaining its inventory by monitoring the health of all its assets. If any of the assets are unhealthy, it's responsible for taking steps to resolve the fault, which might include the following:

- Restarting your role
- Restarting a server
- Reprogramming a load balancer to remove the server from the active pool
- Managing upgrades
- Moving instances of your role in fault situations

Windows Azure follows a cloud computing paradigm known as the fabric, which is another way of describing the data center. Like in the movie *The Matrix*, the fabric is everywhere. Every single piece of hardware (server, router, switch, network cable, and so on) and every VM is connected together to form the fabric. Each resource in the fabric is designed and monitored for fault tolerance. The fabric forms an abstract representation of the physical data center, allowing your applications to run in the fabric without knowledge of the underlying infrastructure.

Figure 1.11 shows how the Fabric Controller monitors and interacts with the servers. It's the central traffic cop, managing the servers and the code that's running on those servers. The Fabric Controller performs the job of the kernel (except across multiple servers at a server level rather than at CPU and memory level) in terms of allocating resources and monitoring resources.

One of the jobs that the Fabric Controller doesn't do (but that a kernel does) is the abstraction of the I/O devices. In Azure, this job is performed by storage services, which we'll discuss next (the storage services component sits near the top of figure 1.11).

1.4 Storing data in the cloud with Azure

Suppose you're developing a new podcasting application for Windows 7. For this application, you want to convert MP3 files to WMA. To convert an MP3 file, you first need to read the file from a hard disk (and eventually write the result). Even though there are thousands of different disk drives, you don't need to concern yourself with the implementation of these drives because the operating system provides you with an abstracted view of the disk drive. To save the converted file to the disk, you can write the file to the filesystem; the operating system manages how it's written to the physical device. The same piece of code that you would use to save your podcast will work, regardless of the physical disk drive.

In the same way that Windows 7 abstracts the complexities of the physical hardware of a desktop PC away from your application, Windows Azure abstracts the physical cloud infrastructure away from your applications using configuration and managed APIs.

Applications can't subsist on code alone; they usually need to store and retrieve data to provide any real value. In the next section, we'll discuss how Azure provides you with shared storage, and then we'll take a quick tour of the BLOB storage service, messaging, and the Table storage service. Each of these is covered in detail in their related sections later in this book.

1.4.1 Understanding Azure's shared storage mechanism

If we consider the MP3 example in the context of Windows Azure, rather than abstracting your application away from a single disk, Windows Azure needs to abstract your application away from the physical server (not just the disk). Your application doesn't have to be directly tied to the storage infrastructure of Azure. You're abstracted away from it so that changes in the infrastructure don't impact your code or application. Also, the data needs to be stored in shared space, which isn't tied to a physical server and can be accessed by multiple physical servers. Figure 1.12 shows this logical abstraction.

You can see how storage is logically represented in figure 1.12, but how does this translate into the world of Windows Azure? Your services won't always be continually running on the same physical machine. Your roles (web or worker) could be shut down and moved to another machine at any time to handle faults or upgrades. In the case of web roles, the load balancer could be distributing requests to a pool of web servers, meaning that an incoming request could be performed on any machine.

To run services in such an environment, all instances of your roles (web and worker) need access to a consistent, durable, and scalable storage service. Windows Azure provides scalable storage service, which can be accessed both inside and outside the Microsoft data centers. When you register for Windows Azure, you'll be able to create your own storage accounts with a set of endpoint URIs that you can use to access access the storage services for your account. The storage services are accessed via a set of REST APIs that's secured by an authentication token. We'll take a more detailed look at these APIs in parts 4 and 5 of this book.

Windows Azure storage services are hosted in the fabric in the same way as your own roles are hosted. Windows Azure is a scalable solution; you never need to worry about running out of capacity.

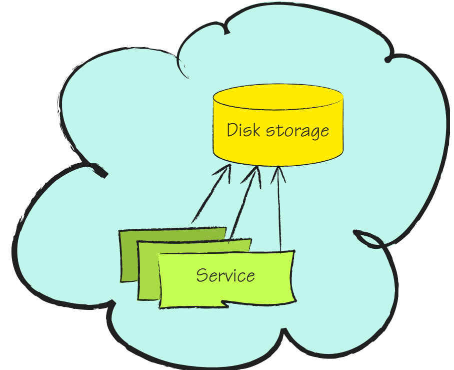


Figure 1.12 Multiple instances of your service (that don't care what physical server they live on) talking to an abstracted logical filesystem, rather than to a physical drive

1.4.2 Storing and accessing BLOB data

Windows Azure provides the ability to store binary files (BLOBs) in a storage area known as *BLOB storage*.

In your storage account, you create a set of containers (similar to folders) that you can store your binary files in. In the initial version of the BLOB storage service, containers can either be restricted to private access (you must use an authentication key to access the files held in this container) or to public access (anyone on the internet can access the file, without using an authentication key).

In figure 1.13, we return to the audio file conversion (MP3 to WMA) scenario. In this example, you're converting a source recording of your podcast (Podcast01.mp3) to Windows Media Audio (Podcast01.wma). The source files are held in BLOB storage in a private container called *Source Files*, and the destination files are held in BLOB storage in a public container called *Converted Files*. Anyone in the world can access the converted files because they're held in a public container, but only you can access the files in the private container because it's secured by your authentication token. Both the private and public containers are held in the storage account called *MyStorage*.

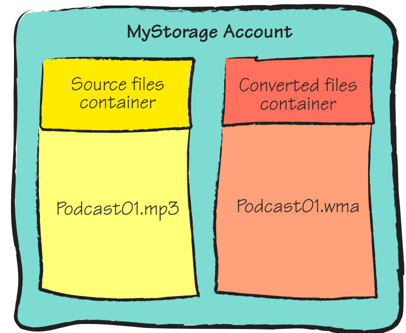


Figure 1.13 Audio files held in BLOB storage

BLOBs can be split up into more manageable chunks known as *blocks* for more efficient uploading of files. This is only the tip of the iceberg in terms of what you can do with BLOB storage in Azure. In part 4, we'll explore BLOB storage and usage in much more detail.

BLOBs play the role of a filesystem in the cloud, but there are other important aspects of the storage subsystem. One of those is the ability to store and forward messages to other services through a message queue.

1.4.3 Messaging via queues

Message queues are the primary mechanism for communicating with worker roles. Typically, a web role or an external service places a message in the queue for processing. Instances of the worker role poll the queue for any new messages and then process the retrieved message. After a message is read from the queue, it's not available to any other instances of the worker role. Queues are considered part of the Azure storage system because the messages are stored in a durable manner while they wait to be picked up in the queue.

In the audio file conversion example, after the source podcast BLOB (Podcast01.mp3) is placed in the Source Files container, a web role or external service places a message (containing the location of the BLOB) in the queue. A worker role retrieves the message and performs the conversion. After the worker role converts the file from MP3 to WMA, it places the converted file (Podcast01.wma) in the Converted Files container.

If you're experiencing information overload at this point, don't worry! In part 6, we'll look at message queues in much greater detail and give you some concrete examples to chew on. Windows Azure also provides you with the ability to store data in a highly scalable, simple Table storage service.

1.4.4 Storing data in tables

The Table storage service provides the ability to store serialized entities in a big table; entities can then be partitioned across multiple servers.

Using tables is a simple storage mechanism that's particularly suitable for session management or user authentication. Tables don't provide a relational database in the cloud, and if you need the power of a database (such as when using server-side joins), then SQL Azure, discussed in chapter 13, is a more appropriate technology.

In chapters 11 and 12, you'll learn how to use Table storage and in what scenarios it can be useful. Let's turn now to the question of why you might want to run your applications in the cloud. You'll want to read the next section, if for no other reason than to convince your boss to let you use it. But you should probably have a better argument prepared than "it's real cool, man" or "this book told me to."

1.5 Why run in the cloud?

So far in this chapter, we've said, "Isn't Azure shiny and cool?" We've also said, "Wow, it's so great I can take my existing IT app and put it in the cloud." But what we haven't asked is, "Why would I want to stick it in the cloud? Why would I want to host my applications with Microsoft rather than host them myself? What advantages do I get using this new platform?" The answers to these questions include the following:

- You can save lots of money.
- You won't need to buy any infrastructure to run your application.
- You don't need to manage the infrastructure to run your application.
- Your application runs on the same infrastructure that Microsoft uses to host its services, not some box under a desk.
- You can scale out your application on demand to use whatever resources it needs to meet its demands.
- You pay only for the resources that you use, when you use them.
- You're provided with a framework that allows you to develop scalable software that runs in the Windows Azure platform so your applications can run at internet scale.
- You can focus on what you're good at: developing software.
- You can watch football and drink milkshakes without being disturbed because someone pulled out the server power cable so they could do the vacuuming.
- You can save lots of money.

In case you think we're repeating ourselves by saying "You can save lots of money" twice, well, it's the key point: you can save a lot. We're often involved in large-scale systems for which the infrastructure costs millions (and most of the time, the servers sit idle). That's not including the cost of running these systems. The equivalent systems in Azure are about 10 percent of the cost.

With that in mind, this section will show you a few of the ways the Windows Azure platform can help you out and save lots of money.

1.5.1 Treating computing power as a utility service

In traditional on-premises or managed-hosting solutions, you either rent or own the infrastructure that your service is hosted on. You're paying for future capacity that you're currently not using. The Windows Azure platform, like other cloud platforms, follows a model of utility computing.

Utility computing treats computing power or storage in the same way you treat a utility service (such as gas or electricity). Your usage of the Windows Azure platform is metered, and you pay only for what you consume.

PAY AS YOU GROW

If you have to pay only for the resources you use, you can launch a scalable service without making a huge investment up front in hardware. In the early days of a new venture, a start-up company survives from investment funding and generates very little income. The less money the company spends, the more chance it has of surviving long enough to generate sufficient income to sustain itself. If the service is successful, then the generated income will pay for the use of the resources.

It's not unusual for technology start-ups to purchase large and expensive hardware solutions for new ventures to cope with predicted future demand. If the service is successful, then it'll require the extra capacity; in the meantime, the start-up is paying for resources that it's not using. Utility computing offers the best of both worlds, giving you the ability to use extra capacity as the service grows without making up-front investments in hardware, and to pay only for the resources that that you use.

SCALE ON DEMAND

Some situations involve large, unpredictable growth; you want to handle the load, but not pay for the unused capacity. This situation might appear in the following scenarios:

- Viral marketing campaigns
- Referrals by a popular website
- Concert ticket sales

Let's say you run a Hawaiian Shirt Shop, and you typically have a predictable pattern of usage. If, for example, Ashton Kutcher (who has 2,000,000 Twitter followers) tweets that he buys his shirts from your website, and he posts a link to your site to all his followers, it's likely that your website will experience a surge in traffic.

Look at the graph in figure 1.14. It shows that your website normally receives around 1,000 hits per day. After Ashton Kutcher tweeted about your website, that increased to

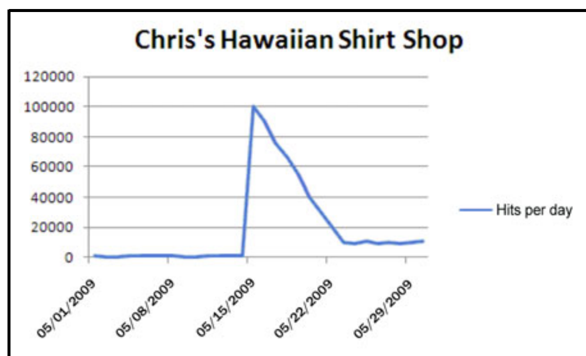


Figure 1.14 Traffic before, during, and after Ashton Kutcher plugged your site

100,000 hits per day. The traffic dropped off after about a week, and then the website had a new baseline of around 10,000 hits per day.

With Windows Azure, you can dynamically scale up to handle the increased traffic load for that week (and get all the extra shirt sales); then, as the traffic decreases, you can scale back down again, paying only for the resources you use. Scaling up and down in Azure is quite simple, and we'll discuss how to do it in detail in chapter 6. You have several options at your disposal. It's important to remember that Azure doesn't scale your service for you. Because it costs money, you have to tell Azure how many servers you want. Azure gives you tools to do this. You can simply log into the portal and make a small change to a configuration file, which is the easy, manual way. You can also use the Service Management API (covered in chapter 18). This API lets you change the resources you have allocated to your application in an automated way.

It's not only possible to scale up and down for predictable (or unpredictable) bursts of growth; you can also dynamically scale your service based on normal, varied usage patterns.

VARIED USAGE PATTERNS

Returning to the Hawaiian Shirt Shop example: after the Ashton Kutcher hype died down a little, your website leveled off at around 10,000 hits per day. Figure 1.15 shows how this traffic varies over the course of a day. Most of the time there's little traffic on the site, apart from during lunch and in the evening. Evidently, most people don't buy Hawaiian shirts when they're at work.

Because it takes only a few minutes to provision a new web server in Windows Azure, you can dynamically scale your website as your usage patterns dictate. For the Hawaiian Shirt Shop, you might decide to run one instance of the website during the day, but in the evening to run three instances to deal with the increased traffic.

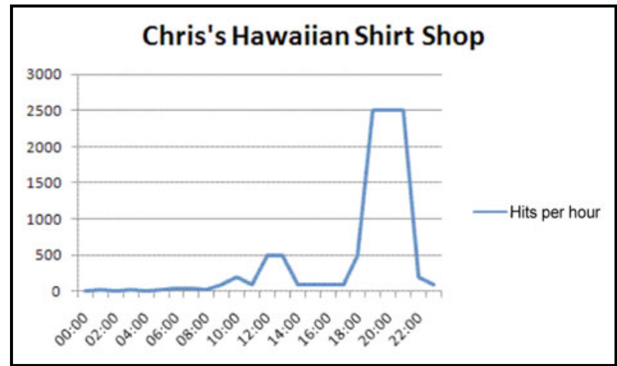


Figure 1.15 Distribution of website traffic over a single day

This sort of scenario is a perfect example of when cloud computing, specifically using Windows Azure, is a perfect fit for your business. If you need to scale beyond a single web server for certain periods of the day when traffic is high, Windows Azure is a cost-effective choice because it allows you to scale back when traffic dies down. Cloud computing solutions are the only offerings that give you this elastic scalability. Other solutions typically require you to over-provision your hardware to cope with the peak periods, but that hardware is underused at off-peak times.

Enough capacity

This example of utility computing can be extended further in regard to available capacity. It's fair to say that most people don't have any idea of the available spare electricity capacity of the network supplying their home, but most of us are confident that if we plug in an extra television, the required electricity will be supplied. The same holds true in Windows Azure. We have no idea how much spare capacity the Microsoft data centers have, but we do know that there's enough. If you require an extra instance of your website, web service, or backend service to be hosted, this will be provisioned for you, within minutes. If the data that your service stores is much larger than you originally anticipated due to the level of growth, more disk space will be allocated to you. You never have to worry about running out of disk space or running out of computing power. You only have to worry about running out of money to pay for the services.

Growth is difficult to model effectively, so knowing there's always enough capacity to grow allows you to concentrate on providing your service rather than worrying about capacity planning, provisioning of new servers, and all their associated tasks.

So far, we've discussed the cost savings you can achieve by scaling your application up and down. Let's now look at how you can save money in maintenance costs.

1.5.2 Simplified data-center management

In this section, we'll look at how the effort involved in operationally maintaining your application is reduced in the Windows Azure environment.

BUILT-IN FAULT TOLERANCE

In Windows Azure, if the physical hardware that your service instance resides on fails, that failed instance is redeployed to another machine. Hardware maintenance resides solely in Microsoft's domain, and you don't have to worry about it.

When you have more than two instances of your application running in Windows Azure, each instance of your web role doesn't live on the same physical server as another instance. This arrangement ensures that if the hardware for one instance dies, the other instance can continue to perform incoming requests. Not only does the second instance live on a different physical server, but it also lives on a different rack (in case the server rack fails). The server rack where the second instance resides is connected to a different network and power grid from the first rack. This level of

fault tolerance ensures that if there's a failure on the physical server, the server rack, the network, or in the electricity supply, your service continues to run and is able to service requests.

When you install your application, Windows Azure decides what servers to place your instances on, with the intention of providing the maximum levels of fault tolerance. Because all data-center assets are tracked and mapped, the placement of applications on each physical server is determined by an algorithm designed to match the fault-tolerance criteria. Even with only two instances of an application, these considerations are pretty complex, but Windows Azure maximizes fault tolerance even when there are hundreds of instances.

Although fault tolerance is maintained within a physical data center, Azure doesn't currently run across the data centers, but runs only within a single data center. You still need to perform offsite backups (if you need them). You can replicate your data to a second data center and run your applications in more than one data center if geo-redundancy is required.

One of the key differences between Windows Azure–hosted applications and regular on-premises solutions or other cloud solutions is that Windows Azure abstracts away everything about the infrastructure, including the underlying operating system, leaving you to focus on your application. Let's see how Azure's ability to maintain the servers your applications run on reduces cost.

SERVER SOFTWARE MAINTENANCE BEGONE!

Whether you're running an entire data center or hosting a website on a dedicated server at a hosting company, maintaining the operating system is usually your responsibility. Maintenance tasks can include managing antivirus protection, installing Windows updates, applying service packs, and providing security. If you're running your own dedicated machine on your own premises rather than it being hosted for you, then you're even responsible for performing backups.

In Windows Azure, because the tasks associated with maintaining the server are the responsibility of Microsoft, you can focus completely on your application. This situation greatly simplifies and reduces the cost of running a service.

A final cost consideration is that if you have a service hosted in Windows Azure, you don't have to worry about the licensing costs for the underlying operating system. You gain all the benefits of running under the latest version of Windows without paying for the costs of maintaining that software. The underlying software is abstracted away from your service, but the base underlying operating system of your service is Windows Server 2008. If you're running multiple servers, the cost of licensing usually runs into thousands of dollars.

Although you don't have to worry about hardware or software maintenance from an operational or cost perspective, you do have to worry about it from a software design perspective.

DESIGNING FOR DISTRIBUTION

Your services won't always be running on the same machine, and they might be failed over to another machine at any time. Failover might be caused by hardware failures, software maintenance cycles, or load distribution. You must design your software so that it can handle these failures. This might mean automatically retrying a failed operation when an exception occurs or reloading any local caches when a service restarts. We'll delve further into these issues in chapter 18.

Let's switch gears now and look at what the Windows Azure *platform* is all about.

1.6 Inside the Windows Azure platform

To reiterate, the major difference between Windows Azure and the Windows Azure platform is the first one is the operating system, and the latter is the broader ecosystem of related services and components. In this section, we'll briefly overview the flagship cloud services provided by the Windows Azure platform (beyond Windows Azure itself): namely, a relational database using SQL Azure, and a set of enterprise services that use the Windows Azure platform AppFabric.

The Windows Azure platform provides many services. In this book, we don't cover every last aspect of every service that's offered across the platform because each component could probably justify its own dedicated book. We'll give you an overview of what's offered, when it's useful, and how you can use the more common scenarios.

Let's get started with the service that you're most likely to use: SQL Azure.

1.6.1 SQL Server capability in the cloud

Although Windows Azure does offer support for storing data in tables, this is a basic storage capability that's suited only for certain core scenarios, which we discuss in chapters 11 and 12.

If you need to create more advanced databases, need to migrate existing SQL databases to Azure, or can't cope with learning another data storage technology, then SQL Azure is the best solution for you. SQL Azure is a relational database (very similar to SQL Server Express Edition) that's hosted within the Windows Azure platform.

A history lesson

When SQL Azure was first announced and made available as a Community Technology Preview (CTP), it was architected differently from the way it currently is. The initial previews of what was known as SQL Server Data Services (SSDS) were of a nonrelational model that was similar to Windows Azure storage services. The feedback given to the product teams made it clear that customers wanted a relational database in the cloud, and SSDS was later retired. Prior to being renamed as SQL Azure, SSDS was renamed SQL Data Services, but there was never a public CTP under this name.

WHAT IS SQL AZURE?

Version 1.0 of SQL Azure, which was released at PDC 2009, provides the core capabilities of SQL Server in the cloud. The first release can be likened to running an instance of SQL Server Express Edition on a shared host, with some changes to security so that you can't mess with other databases on the same server.

Communication with SQL Azure is via the Tabular Data Stream (TDS) protocol, which is the same protocol that's used for the on-premises editions of SQL Server. You can connect SQL Management Studio directly to your database hosted in the cloud, as if it were hosted locally.

> **NOTE** In the first release of SQL Azure, security is limited to SQL Server user accounts. Windows Integrated Security isn't yet supported. Expect some sort of support beyond SQL Security at a later date.

Because you can connect to SQL Azure with a regular connection string, any existing data access layers continue to work normally. Figure 1.16 shows communication between SQL Azure and applications that are hosted both inside and outside Windows Azure.

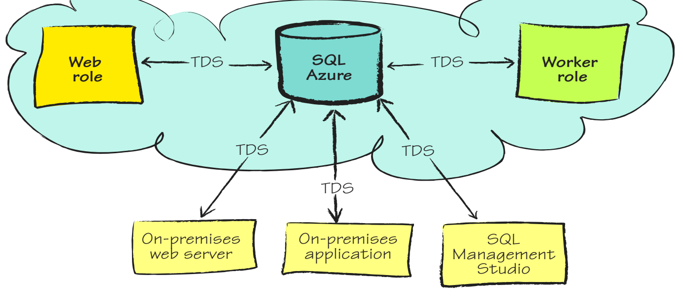


Figure 1.16 On-premises, Azure-hosted applications and SQL Management Studio communicating with SQL Azure via TDS

If your application works today using SQL Server Express Edition and doesn't use some of the more advanced features of SQL Server (see chapter 13), then your application should work in the cloud with little or no modification.

Although on-premises applications can talk to SQL Azure, latency might make this a less attractive option. The closer your application is to the database, the faster it'll go. You can reduce the impact of latency by making your application less chatty.

HOW SCALABLE IS SQL AZURE?

In version 1.0 of SQL Azure, there's no built-in support for data partitioning (the ability to split your data across multiple servers). The initial release is targeted for databases that are sized up to 10 GB; larger databases aren't suitable for SQL Azure in this initial release, but support for larger databases will be available in future service updates. If you need to perform partitioning, you need to implement it in the application layer.

Let's turn now to Azure platform's enterprise services. Known as AppFabric, these services include the Access Control Service (ACS) and the Service Bus.

1.6.2 *Enterprise services in the cloud*

The Windows Azure platform AppFabric (formerly .NET Services) is a set of services that's more oriented toward enterprise applications and is comprised of the following components:

- AppFabric ACS
- AppFabric Service Bus

AppFabric is a large set of technologies that would require its own book to cover in any depth. In this section, we'll give you an overview of the technologies. In chapter 17, we'll show you how to get started using them and discuss a couple of key scenarios in which to use the technology.

Velocity and Dublin: where are they now?

In addition to the Windows Azure platform AppFabric product, there's also an on-premises product known as Windows Server AppFabric. This is a completely different product set, which currently includes AppFabric Caching (formerly Velocity) and AppFabric Service Workflow and Management (formerly Dublin).

Although these services aren't currently part of the Windows Azure platform (PDC 2009), you can expect them to appear at some point. Depending on when you're reading this book, they could be available right now. Alternatively, you could be reading this in the far future (relative to PDC 2009) and there's no need for this technology because we're all plugged in to the Matrix.

ACCESS CONTROL SERVICE

User authentication security and management is a fairly standard requirement for any service used in an enterprise organization. Enterprises require that their employees, customers, and vendors be able to access all services in the organization with a single login; typically, the authentication process occurs using the Windows login. After you're authenticated, you're typically issued a token, which is automatically passed to other services in the enterprise as you access them. The automatic passing and authentication of this token means you don't have to continually log in each time you access a service.

Services in the enterprise don't implement their own individual authentication and user-management systems but hook directly into the organization's identity-management service (such as Active Directory). This single sign-on process provides many benefits to the company (centralized and simplified user management and security) and is a much more integrated and less frustrating user experience.

Traditionally, identity management and access control have been restricted to the enterprise space. With the advent of Web 2.0 social platforms, such as Live Services and Facebook, this level of integration is now creeping into everyday websites. Web users are now more concerned about data privacy and are reluctant to cheaply give away personal information to third-party websites. They don't want a long list of user names and passwords for various sites and want a much richer social experience on

the web. For example, it's increasingly common for people to want to be able to tell their friends on Facebook about their latest purchase. Between Facebook, Live Services, OpenID (and its differing implementations), and all the various enterprise identity-management systems, access control has now became a complex task.

AppFabric ACS abstracts away the nuances of the various third-party providers by using a simple rules-based authentication service that manages authentication across multiple providers for users with multiple credentials. We'll look more closely at ACS in chapter 17 and show you how to use it in a couple of key scenarios.

APPFABRIC SERVICE BUS

The Service Bus is a cool piece of technology that allows you to message with applications that aren't necessarily running in the Azure data centers. If you have a custom, proprietary service that you need to continue to host, but you want to use Azure for all other aspects of your service offering, then the Service Bus is a good way to integrate with those services.

The Service Bus is effectively an Enterprise Message Bus that's hosted in the cloud. We'll explore in more detail what this means and look at a couple of key scenarios where you could use this technology in chapter 17.

1.7 Summary

In one chapter, you've learned about cloud computing, Windows Azure, and the Windows Azure platform. Although both Windows and Windows Azure are operating systems, providing all the needed functions, they differ greatly in terms of scale. Windows is an operating system for a single machine, whereas Windows Azure is an operating system for a whole fabric of machines, devices, networks, and other related items.

You learned that you can easily scale applications that run in Windows Azure to support the future needs of your application, but you pay only for your current needs.

We also briefly discussed why you might want to use the cloud. The cloud can give you new capabilities, such as dynamic scaling, disposable resources, and the freedom from manning any of it. But the real reason anyone uses the cloud always boils down to money. The rationale is simple: functionality you can't afford to provide in a normal data center is affordable in the cloud.

You developed your first Windows Azure web application and saw that developing in Windows Azure builds on your existing skills. You can now easily write code that runs locally or runs in the cloud—depending on your needs, not your skills or tools.

Like a desktop operating system, the Windows Azure platform consists of a lot of different parts. The platform includes SQL Azure, which provides a traditional relational database that gives you a familiar setting and makes it easier to migrate applications. Azure can also run your applications and manage storage.

We looked at how the Windows Azure platform AppFabric fits into the equation. AppFabric provides both a Service Bus you use to connect your applications together and a simple, standards-based way to secure your services called Access Control Service (ACS).

In chapter 2, we'll continue our discussion of Azure by showing you how to take your first steps with a web role and how to work with code that goes beyond Hello World.

2 Your first steps with a web role

This chapter covers

- Building a basic website
- Signing up for Azure
- Deploying your first cloud application

With the first chapter out of the way, you should have gotten a feeling for the lay of the land, installed the Windows Azure SDK, and run a Hello World application locally. Let's dive right in to building a website to run on Azure. In this chapter, we'll cover all the steps involved:

- Starting a new Visual Studio project
- Building the XHTML and code for the website
- Running and debugging the site locally
- Deploying the site to the Azure staging environment
- Moving the site to the Azure production environment

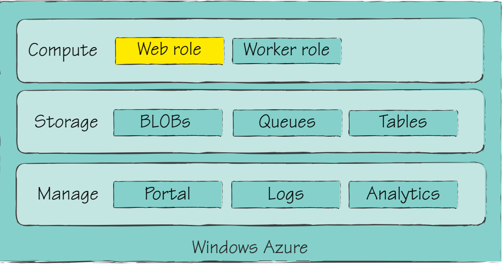


Figure 2.1 The basic components of Windows Azure. In this chapter, we'll focus on how you use the web role in Azure.

Don't let these steps daunt you. If you've ever developed a website with ASP.NET, you're already ahead of the game. You'll have to complete far fewer steps to deploy an application with Azure than you would with a traditional server.

In a large enterprise project one of us worked on, 15 percent of the work hours was spent planning the development, quality assurance, and production environments. Most of this time was used to define hardware requirements, acquire capital expenditure approval, and deal with vendor management. We could've shipped much sooner if we'd been able to focus on the application and not the underlying infrastructure and platform. Many organizations take three to six months just to deploy a server! You won't require this much time to complete the entire process using Windows Azure.

In this chapter, we'll focus on the basic process of deploying a simple website using the web role in Azure, which is highlighted in figure 2.1.

Before we start discussing the web role in detail, let's take a closer look at the Azure SDK.

2.1 Getting around the Azure SDK

After you install the SDK, a folder structure is created on your computer that's filled with tools, goodies, and documentation. The default path is C:\Program Files\Windows

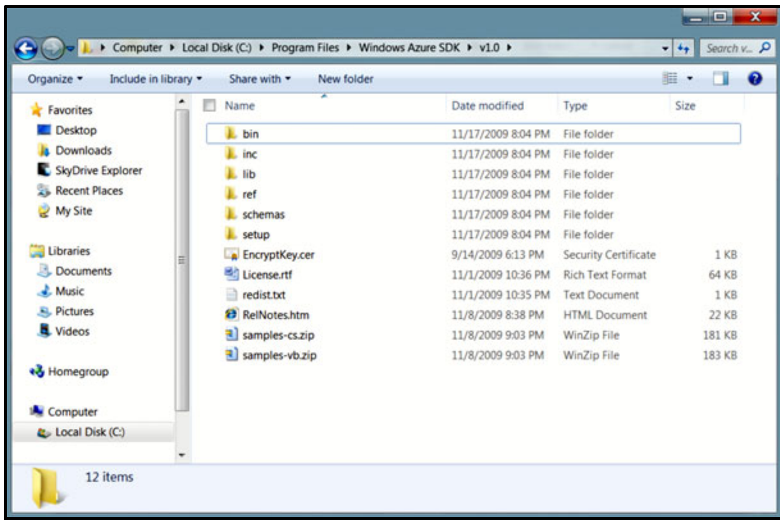


Figure 2.2 The directories that the Azure SDK installs for you. You should examine these because they contain lots of handy tools.

Azure SDK\v1.1. (Pay attention to the path name. It'll vary, depending on the version of the SDK you've installed. In our case, we have version 1.1 installed.) The contents of this directory are shown in figure 2.2.

You should explore these folders briefly, to be familiar with what's available. There are several tools included in the SDK that are quite powerful. Some tools can help you automate build environments and the management of your live Azure applications.

2.1.1 Exploring the SDK folders

Two of the more important SDK folders are bin and inc. The bin folder contains the assemblies and tools that you need to work with Azure packages and to run the development fabric. The inc folder holds a header file that you use if you're working with C++. Table 2.1 lists the tools included in the bin directory.

Table 2.1 The most important tools included in the SDK are found in the bin directory.

Visual Studio tool	Purpose
`CSmonitor`	This tool loads the local development fabric. You'll see the icon appear in your systray.
`CSpack`	Use this tool to manually create deployment packages to upload to the Azure portal when you're ready to run your application in the cloud.
`CSrun`	This tool lets you deploy a package generated by `CSpack` to run on the local development fabric.
`DFUI`	This script starts the management UI for the local development fabric.
`DSinit`	Use this tool to run the initialization needed for the Development storage service on a new SQL instance. If you're using SQL Server Express, the initialization is usually run once, on first startup. You'll need to run this yourself if you're running a local instance of SQL Server. This tool is located in the devstore subdirectory, under the bin folder.

The SDK provides tools you can use when you aren't using Visual Studio or when you're automating some of your processes. For example, if you're using Eclipse to write Java code for Windows Azure, you'll need to run these programs yourself, whereas Visual Studio will run them for you automatically.

2.1.2 Using the Cloud Service project templates

When you installed the SDK, a new project template group called Cloud Service was created in Visual Studio. This group includes several templates for you to start working with. You can use them when you're adding a new project to your solution. Never fear; you can also add an existing project, which is handy when you're migrating an application to the cloud.

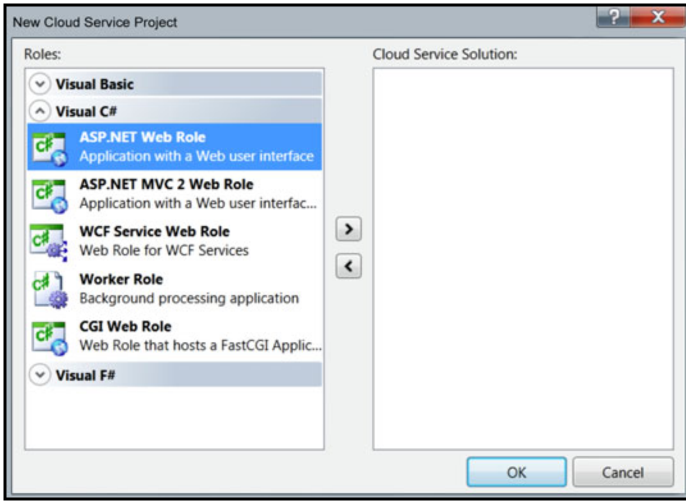


Figure 2.3 You have several options when it comes to adding an Azure project to your solution. These project templates are provided with the SDK.

Let's look at the Cloud Service project options that are available to you. The New Cloud Service Project window is shown in figure 2.3.

The SDK provides the following Visual Basic templates that you can use in your solutions:

- *ASP.NET Web Role*—This template creates an ASP.NET project, preconfigured with an accompanying Azure project.
- *ASP.NET MVC2 Web Role*—This template creates a project similar to the ASP.NET Web Role template, but is prewired to support the MVC2 framework.
- *WCF Service Web Role*—Planning on hosting a Windows Communication Foundation (WCF) service instead of a normal web application? Then this is the project for you. You set this up like a normal WCF project, using sample files for your first service.
- *Worker Role*—This template creates a class library project, preconfigured with a related Azure project. You should use this project if you're building a background processing service.
- *CGI Web Role*—This project template creates the required files needed to host a FastCGI project, which we'll cover in chapter 6.
- *Blank Cloud Service*—This isn't really a template, but if you click OK without adding any projects, you'll have a solution that contains an Azure project without any supporting application projects.

NOTE We're using Visual Studio 2010 in this book. You can use Visual Studio 2008. If you do, your experience will be similar to that depicted in this book, but some screenshots will look different, and the installation process is slightly different.

2.1.3 Running the cloud locally

We talked in chapter 1 about the internal web server that Visual Studio has included for the past few versions, called *Cassini*. This server makes it easy for a web developer to develop locally, without having to install full-blown server software on their desktop. Cassini was also designed to be lightweight and to respond only to local web requests, which enhances performance and security. The Azure SDK includes similar services that help the developer develop cloud websites and services locally.

The SDK installs the development fabric service and the development storage service. Both these services are started automatically by Visual Studio when you run an Azure project. (If you aren't using Visual Studio, or if the services don't start for you, you can start them from the Windows Start menu or from the Azure command prompt using `CSMonitor.exe`.) Although these services are separate processes, they appear as a single icon in the notification area of your task bar, shown in figure 2.4. Each service has several different processes related to it, which handle load balancing, node management, and a few other tasks. These processes allow you to run and debug your cloud services locally, before you deploy them to the cloud. They effectively work together, with a local instance of SQL Server Express, to simulate the real Azure storage and runtime environment. Because the environment is simulated, there are some limitations, which we'll discuss later in this chapter.

Figure 2.4 You can find the local cloud service icons in your notification area. The blue Windows flag tells you that the development fabric and the development storage services are running. Right-click the icon to show the UI you use to manage the services.

When these services start, both services continue to run until you stop them with the taskbar UI. We recommend that you leave them running while you're developing, and then shut them down when you're done for the day.

You can use an existing local instance of SQL Server instead of SQL Server Express to support the storage service. The SDK includes a tool called `DSinit` that you can use to initialize the storage databases. To configure the storage service to use your local SQL instance, use the `/sqlInstance` parameter with the name of the SQL instance you want to initialize. You can use a single period to represent the default instance for the server running, so you don't have to look up the name you gave the instance when you installed SQL Server: `DSinit.exe /sqlInstance`.

The SDK also includes a variety of samples that you can use to get a good idea of how to approach supported scenarios. One of the best ways to learn Azure is to walk through the code in the samples and understand how they work. The samples are provided in a zip file in the root of the SDK folder. Extract them to a folder to work with the code.

2.1.4 How the local and cloud environments differ

Although the SDK tools try to provide a complete simulation of the production cloud environment, there are some limitations developers should be aware of. Most of these limitations exist because the local environment is running on one machine, with limited resources. Others exist because the local storage service is based on a local SQL Server Express instance. We think that over time, the gaps between the local environment and the cloud environment will shrink. Table 2.2 summarizes the differences between the environments. The SDK documentation has a complete and up-to-date list of all the differences.

Table 2.2 Differences between the local and Azure cloud environments

Feature	Local environment	Cloud environment
Storage environment access	Uses a special account key for access.	Storage environment access is different from your cloud key.
HTTPS support	Doesn't support the use of HTTPS.	Cloud storage supports both HTTP and HTTPS.
Storage performance	Local storage is intended only for a few local connections, nothing more.	Expect the performance of the cloud-based storage to be much faster.
URI management	Because the Azure Domain Name System (DNS) is not part of the local environment, storage URIs are different in the local environment.	URIs are based on the Azure DNS system.
Storage management	The local storage subsystem doesn't provide extended error information.	Cloud storage provides extended error information.
BLOB storage	BLOBs in the local store are limited to 2 GB in size.	BLOBs in the cloud can be as big as 1 TB.

One way to minimize the impact of the local storage limitations is to shift to a blended model during your development. Early on in your project, you can configure your application to use local storage. As you start to bump up against the limits, adjust the configuration of the application to use the cloud-based storage. In this blended mode, you continue to run your application locally. Running your code locally with your data in the cloud during development will incur some slight charges for the bandwidth and storage of your data. We believe the cost is worth it because you'll be developing against the real storage infrastructure and not the local simulation provided by the SDK. Eventually, you'll deploy your application to the cloud staging environment, with the application configured to use the cloud storage.

To get a feel for how the SDK and tools work, you're now going to create a simple website.

2.2 Taking Hello World to the next level

You're going to build a second web application now, and it'll be a bit more complicated than the Hello World sample in chapter 1. This new website isn't going to have a lot of functionality, just a bit of XHTML and a cascading style sheet (CSS), with some simple code. You aren't going to be using any databases or advanced topics. Your goal is to make sure that the SDK and tools are installed correctly, and to learn the general workflow of working with a web application in Azure.

The one-page website that you're going to create lists the different shirts available at Chris's Hawaiian Shirt Shop. In addition to walking through the basic workflow, we'll also show you that in many scenarios, running a website in Azure is like running a website on your own server, and that it's as easy, if not easier, to deploy it. We're going to walk you through only the pertinent code; you can review the complete code listing in the sample code provided with this book.

2.2.1 Creating the project

Your first step is to create the project. Open Visual Studio and select File > New > Project. Visual Studio displays a list of available templates; select the Cloud Service template group, and then select Windows Azure Cloud Service in the menu. Give the project a name, such as HawaiianShirtShop, and click OK. A pop-up menu opens in which you pick the project types you want to add to your Azure solution. Because you're building a simple website, without any backend processing, choose the ASP.NET Web Role template. After you add the web role to the project, you can rename it; then click OK. You'll see the New Cloud Service Project window with one solution, as shown in figure 2.5.

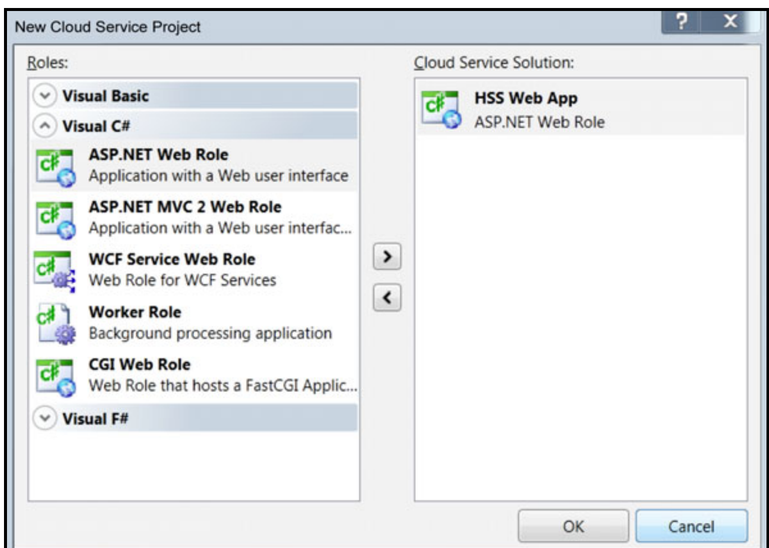


Figure 2.5 Creating a new Azure Application project is easy. Just add the different roles you'll need in your solution.

Visual Studio starts to build a solution file for you that contains a few projects. The first project is given the name that you entered on the New Project window. This new project will be the Azure project, and it doesn't contain a lot. It merely contains the model and configuration Azure needs to run your website. You can think of this information as the metadata for your whole Azure application. The project also contains links to other projects in your solution, and the role type those projects play in your application. These links are stored in the Roles folder. The solution and project structure is shown in figure 2.6.

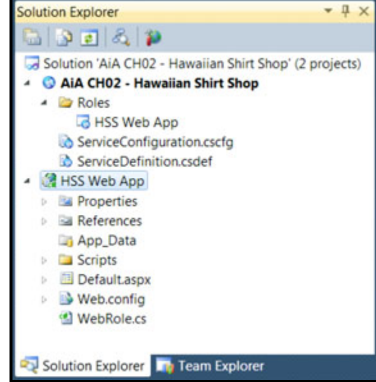


Figure 2.6 The projects that Visual Studio creates for you. The bolded project is the Azure project, which holds the cloud configuration data for your application. Also shown is the HSS Web App project, which is linked to the Azure project as a web role.

Visual Studio also created an ASP.NET web application project for you, and linked it to the Azure project as a web role. Remember that you can name the subprojects when you select them from the list of roles (figure 2.5) by clicking the pencil icon. This icon shows up when you hover over the projects in the Cloud Service Solution list.

This ASP.NET project is typical in almost every way. Because Visual Studio knew the project was going to be part of an Azure application, it created assembly references to three Azure-related assemblies for you. The three assemblies are part of the `Microsoft.WindowsAzure` namespace. The `Diagnostics` assembly covers logging needs, the `ServiceRuntime` assembly provides methods for interacting with the Azure fabric, and the `StorageClient` assembly makes it easy to work with the Azure storage services. Visual Studio also added a file called webrole.cs or webrole.vb. This file is similar to the global.asax file for your website, but is for all of your instances. We'll cover this file in a later chapter.

You could press F5 right now and the project would compile and run locally. Visual Studio would start the local development Fabric and storage services, and then package and deploy your application. Of course, without any code or markup, you wouldn't see too much, except for an empty web browser. To enhance your app, you need to give it some content.

2.2.2 Laying down some markup with XHTML and a CSS

Now that you have the empty shell in place, you need to put in the content for the website. This new page is going to announce the name of your business and list a few of the shirts you have for sale. When you've completed this task, you'll have a simple web page that looks like figure 2.7.

Open the default.aspx file and paste in the markup shown in listing 2.1. With Azure, you don't usually need any special tags or changes to your website to do what

Figure 2.7 After you add some simple XML and a CSS, your web page will look like this.

you've always done. This is one thing that makes it easy for a web developer to learn how to use Azure and quickly move to the cloud.

Listing 2.1 The simple ASP.NET XHTML markup for your web page content

```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeBehind="Default.aspx.cs" Inherits="HSS_Web_App._Default" %>

<!doctype html public "-//w3c//dtd xhtml 1.0 transitional//en"
➥   "http://www.w3.org/tr/xhtml1/dtd/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Chris's Hawaiian Shirt Shop</title>
    <link rel="stylesheet" href="Main.css" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div id="header">
        <h1>
            Chris's Hawaiian Shirt Shop</h1>
    </div>
    <div id="mainContent">
        <h3>
            Many Styles Available!</h3>
        <p>
```

```
        We have many styles available, to suit any special need or
    occasion. Please check out our amazing classic 50's styles. Call now to
    place your order or to check on what we have in stock.</p>
        <asp:Repeater ID="rptProductList" runat="server">      <-- 1
            <ItemTemplate>
                <div class="item">
                    <h3>
                        <%# Eval("productName") %>
                    </h3>
                    <div>
                        <b>
                            $<%# Eval("unitPrice")%>
                            each.</b>
                        <p><%# Eval("description") %></p></div>
                </div>
            </ItemTemplate>
        </asp:Repeater>
    </div>
    </form>
</body>
</html>
```

1 (annotation)
2 Renders only shirt data

The markup shown in listing 2.1 is going to produce a pretty boring and run-of-the-mill ASP.NET web page. The `asp:Repeater` (1) is bound to a collection of shirt data so that it renders a list of products available. The repeater renders your business data with the `eval` statements (2).

2.2.3 Binding your data in the code-behind

The next step in your simple sample is to get the product data from a stub that will provide sample data, and then bind it to the repeater control in the markup. Let's call the stub `ProductInfo`. The complete code for the `ProductInfo` class can be found in the sample code provided with this book. A lot of developers would call this class `ProductService`, but we think the *service* word is overloaded as it is and makes things confusing. (You should feel free to roll the way you want; it's your code.) You'll also have a data transfer object (a simple class that has no methods, only properties), called `Product`, to hold your product data. You'll use a generic list collection to hold the multiple product classes as you bind them to the repeater control. You can place these classes in a new C# class file in your project:

```
public class Product
{
     public Product()
     {
     }

    public int sku { get; set; }
    public string productName { get; set; }
    public decimal unitPrice { get; set; }
    public string description { get; set; }
}
```

1 Accessors for product properties

Because we're trying to keep things simple, we're suggesting you use simple property accessors (1) around the business data. Next, you'll add the `Page_Load` code that'll gather the product data and bind it to the fields on the form, as shown in the following code:

```
   ProductInfo productInfoSource = new ProductInfo();
IList<Product> allProducts = productInfoSource.GetAllProducts();

rptProductList.DataSource = allProducts;
rptProductList.DataBind();
```

(1) **Connects to data source and calls DataBind()**

When the page is loaded, it creates a `ProductInfo` object. The page uses that object to get a collection of `Product` objects, which is then bound (1) to the `ProductList` repeater on the web form.

When you've entered the code correctly and run the application, Visual Studio starts the local Azure services (you'll see their icon in your task base like in figure 2.3), and launches the website in your browser. When you're running Azure applications locally, you must run Visual Studio as an administrator. If you forget to do this, Visual Studio will kindly remind you when you press F5.

If you look closely at the code, your web project doesn't have or do anything a normal website wouldn't have or do. The only differences are the Azure project, which tells either the local services or the Azure cloud how to run your application, and the webrole.cs file, which acts as a global event handler for your web role instances.

The Azure Fabric Controller knows how to deploy and manage your application, based on the settings included in the Azure project in your solution. There are several parts to this configuration that we need to cover now.

2.2.4 Just another place to run your code

We're going to spend a lot of time in this book discussing the unique capabilities the Azure platform has. You've probably heard a lot about cloud computing, and might be a little confused or intimidated. Don't worry! Until you fully grok the powers of the cloud, the easiest way to wrap your head around it is to think of it as just another place to run your code. If you boil it down to this simple concept, then its most powerful aspect is the simplicity of the deployment and management of applications.

Each cloud project defines a service model for the application. This model defines how the app should be deployed and run in the cloud. With this model, there isn't a 30-page deployment document with obscure and arcane instructions for deploying and upgrading. You can simply deploy your application with a few clicks and a few minutes of time. Azure knows what to do with your code because of the configuration and service model in the Azure project in your solution.

2.2.5 Configuring the Azure service model

The most important aspects of the Azure project are the links, which tell Visual Studio which projects in the solution are parts of the application, and what role they play. These links appear in the portal as elements that can be configured and scaled separately. A class library in your solution should be referenced with a project assembly

reference as usual, and not linked to from the Azure project file. Besides these links, the Azure project also includes two configuration files:

- *ServiceDefinition.csdef*—This file defines which services are part of the application, and whether the services have any endpoints or connections that need to be provisioned. The information includes the public port that a website might need in order to receive traffic from the internet. This file mainly concerns itself with the configuration of the infrastructure.
- *ServiceConfiguration.cscfg*—This file determines the configuration of the services and how they should be provisioned by Azure. The configuration includes how many instances of each role should be deployed, and other operational characteristics.

We'll investigate each of these files in more depth in chapter 3.

2.2.6 Running the website in the local development fabric

When you run an Azure application locally, Visual Studio starts the development fabric and storage services, and then launches the application. Part of this process includes removing old temporary files and packaging the project to be deployed to the local services.

After the services are loaded, you can run their UIs to see what's happening in the local environment. Right-click the icon in the system tray and select Show Development Fabric UI. Each time you run your application, a new deployment is created, each with a successive number. Other deployments from your applications might show up in the UI, in addition to the one you're currently running.

The UI lists the deployments in the left panel. Each deployment displays the details of that service, as well as a list of the different roles and their instances that are

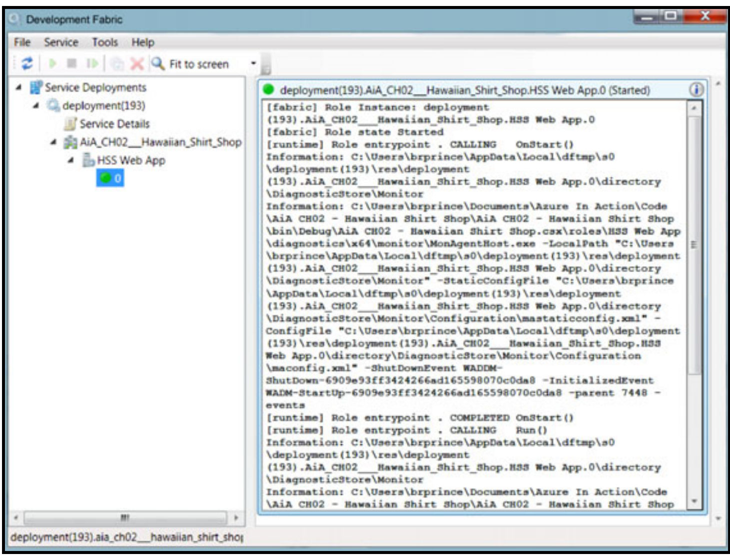


Figure 2.8 The UI of the development fabric. This UI shows that there is one web role running; it also shows the log history for that instance.

running. In figure 2.8, you can see one role, a web role called HSS Web App, which was the ASP.NET project in your solution that was linked to from the Azure project. That role has one instance running. In the UI the instances are numbered, starting with 0, because we're all geeks. In the right panel you see the log history for that instance.

With this UI you can pause or stop the different services, which is helpful during debugging. You can also change the logging levels of each service, which better refines what the log is capturing, and which is also helpful during debugging.

Now that you know how to install and use the Azure SDK and how to run your applications locally using the development fabric and storage services, we're going to show you how to deploy your little website to the cloud, and then become rich selling Hawaiian shirts.

2.3 Deploying with the Azure portal

The Azure portal is one of the three major components you use to manage your Azure applications. The other two components are your system logs (often referred to as diagnostics), and the analytical and billing tools. These components are highlighted in figure 2.9.

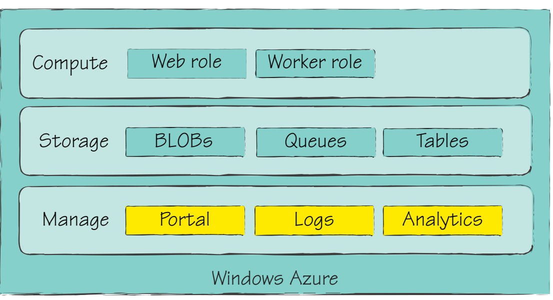


Figure 2.9 The management components of Azure include the portal, system logs, and the analytical tools.

Before you can log in to the portal to manage your applications, you need to create an Azure account.

2.3.1 Signing up for Azure

You need to sign up for an account and provide a live ID and billing details. You'll need to visit www.Azure.com to sign up. Go to the account section and click Get Your Account. You should open two accounts and provide two live IDs: one for business purposes such as billing and contract details, and another that's limited to the technical aspects of the system. Having two accounts cleanly separates the management of the business aspects from the management of the technical aspects. The business account manager is prevented from accidentally shutting down the applications or reconfiguring them. The cloud doesn't remove the risk of human mistakes; Azure just tries to automate these mistakes so that they cause trouble faster and more efficiently.

You'll need to give your credit card information so that they know who to bill for the time you use. Even if you have access to free time in the cloud through MSDN or some other channel, you'll still need to provide a credit card number in case you go over your free allocation.

Now that you have your account provisioned, you're ready to create a project on Azure to host your account. To do this, you use the Azure portal.

2.3.2 The Azure portal

The Azure portal is your central management tool for all aspects of running your application in the Azure environment. As shown in figure 2.10, across the top you'll be able to view the technical details (using the Summary tab) or the account details (using the Account tab) of your services. When you click one of these tabs, any services that you've created (websites, services, storage, and so on) are listed in the Windows Azure section on the left.

The following tabs are available on the portal:

- *Summary*—Lists your projects, and gives you access to managing them. From here you can upload new versions, flip staging and production, and manage log files.
- *Account*—Helps you manage your affinity groups, certificates, and any other broad configuration that affects all services under your account.
- *Help and Resources*—Brings together all the help options available to you, including technical support and public forums.

Azure will show you the different types of services available to you on your Azure account. You'll see icons for Storage Account and Hosted Services. You might also see icons for the other services that are part of the greater Azure Services platform, such as the AppFabric and SQL Azure.

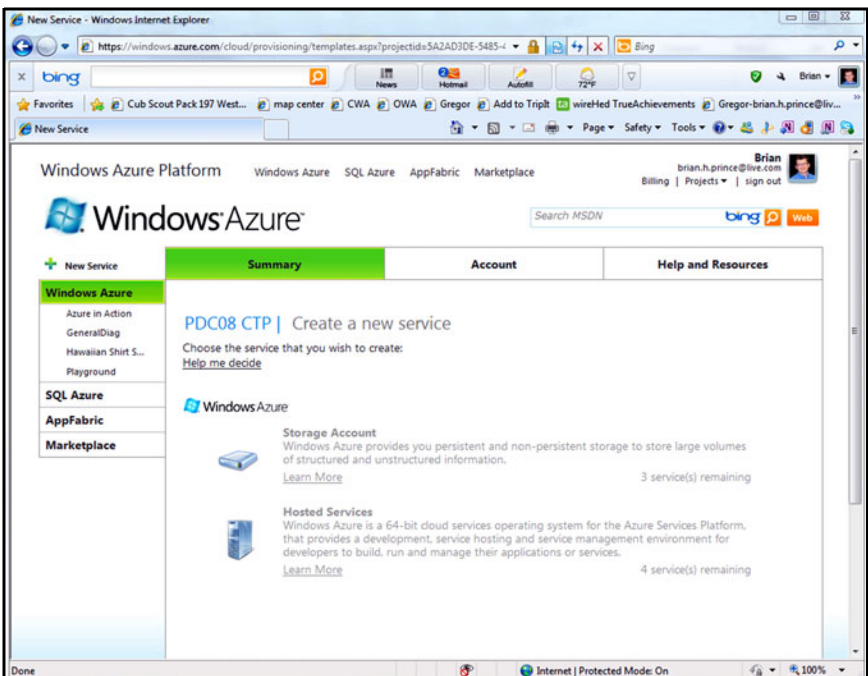


Figure 2.10 The Windows Azure portal. After you've created an application to run in the cloud, choose the Hosted Services project type. This window also shows how many more storage services you can create; the current user is allowed to create three more storage services.

Now that you have your account provisioned, you need to create a project on Azure to host your account.

2.3.3 Setting up your service online

Log in to your Azure portal with the live ID you created your Azure account with. On the home page, click the New Service link at the top right of the window. The process of creating a new service begins; you'll push your code to this new service. The term *service* is used as a general term on the Azure portal to represent some usage of the system, which might be a storage account or one of several websites that you've deployed.

Click the Hosted Services icon shown in figure 2.10 to start the new service wizard. Oh, how we love wizards.

The first step is to provide some basic properties for your project. You need to provide a project name and a description. These are for your own use, but by being careful in how you name and describe your project, you can make it easier to manage your projects over the long haul. After you provide the requested information, click Next to proceed. You're ready for the second step, which is to provide a name for the service and information about affinity groups. You enter this information in the window shown in figure 2.11.

The most important part of this window is the service name. (Remember that pretty much any code running in Azure is called a service, even if it's a simple website like our sample.) This service name will carry over into almost everything you do with the service you're deploying, so choose wisely. Selecting an embarrassing childhood nickname is tempting (such as Doogie Howser or Brainy Smurf), but this should be a name that supports the way you're going to manage your Azure environment.

The name you provide will be the first part of the URL for your service. For example, if your service name is `HawaiianShirtShop`, the Azure DNS server would refer to this site as `HawaiianShirtShop.cloudapp.net`. Because this isn't the name you embroidered on thousands of T-shirts (preferring instead the URL www.HawaiianShirtShop.com), you can use your own DNS tricks to refer people to the correct URL. A simple CNAME record in your DNS server can direct people to your service with any domain name you want to use.

The service name has to be unique across all of Azure. Click the Availability button to find out if the name you want is available. We're going to enter `AzureInAction`, and see if it's available. Because we're doing this, that name won't be available to you; go ahead and check.

The lower half of this window has to do with what Azure calls *affinity groups*. Affinity groups are used to make it easy to deploy services that are related to each other to the same regional data center for performance purposes, or to different data centers for disaster recovery or geo-distribution purposes. As you create more services, instead of having to remember which geographic area to set for your new service, you can select a customer affinity group. Later, when you want to move your services, you just update the affinity group as a whole.

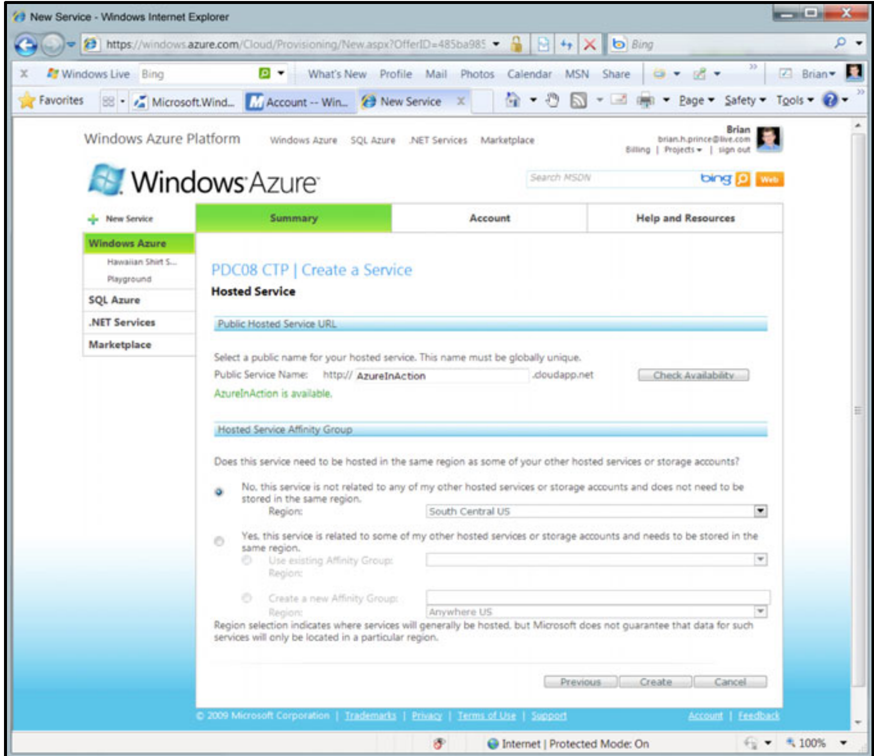


Figure 2.11 In this window you complete the second step in creating an Azure hosted services project. You can provide the name that will be in the URL for your site, as well as the affinity group for geo-distribution. Assign a name carefully; it will permeate everything you do with you service.

Click Create to finish the process. At this point, a project has been provisioned, and is ready for you to deploy an application to. The portal displays the project page for the project you just created.

Each project contains two zones, staging and production. The typical process of deploying your application involves uploading it to the staging environment, testing it, and *flipping* it to the production environment. This flipping is called a *VIP swap*. *VIP* stands for virtual IP.

When you flip your application to a new environment, nothing is actually moved. The Azure Fabric Controller labels the current running instances `production`, and the other instances are labeled `staging`. The Azure Fabric Controller adjusts the assignment of DNS and IP addresses in the Azure data center. The fact that nothing moves makes it fast to do a cutover and reduces the chance of deployment issues. It also makes it easy to flip back to the old setup if the new configuration doesn't work out as planned (not that that has ever happened to us).

Before you set up your storage environment, we should say a few words about logging and the diagnostic agent.

2.3.4 Putting on your logging boots

Logging is important in Azure. Like most production environments, you can't attach your debugger to the code running in the cloud. If there's a problem, you need to have good logging in place to help in diagnosing that problem.

Learn to instrument your code with logging commands so that you have the right information at your fingertips when you need it. Every time you fix a bug, you should consider adding some information to the log as well.

Azure runs a diagnostic agent on your virtual server while the server is running. This agent gathers logs and diagnostic information. You can configure it in code, and manage it remotely with the Azure management APIs. The default project templates provided by the SDK include code that'll start the diagnostic agent when your role instance starts.

The diagnostic agent uses the custom error log extensions to gather logs from several standard sources and from any source you can think of. The standard sources include Azure logs, Windows logs, IIS logs, the Windows event log, and performance counters. The data is transferred to an Azure storage account of your choosing. The diagnostic data is collected and then transferred, either on demand or according to a schedule you set. When the data is there, you can do anything you want to with it. Some data sources are copied to an Azure table, while others are copied to an Azure blog account. The type of destination storage depends on the type of the source.

We'll cover logging in greater depth in chapter 18. For now, you need to set up the storage environment you'll need for your basic application.

2.3.5 Setting up your storage environment

Open your application in Visual Studio and make one more run locally to make sure that it's running fine. There's one default setting you need to change when you're ready to deploy to the cloud. The default configuration specifies that the diagnostic monitor uses development storage. Before you upload your code, you need to reconfigure it to use the cloud for storage; cloud apps can't reach your development storage on your local desktop.

Return to the portal and create a new service. But instead of choosing Hosted Service, choose Storage Account for the type of service that you create. Figure 2.12 shows that the process is similar to creating a hosted service. As before, you need to provide a service name and a description.

After the storage account is created, you'll be given an account key that you can use to securely access your online data, as shown in figure 2.13. We'll go much deeper into storage accounts in chapter 8, but for now just roll with it.

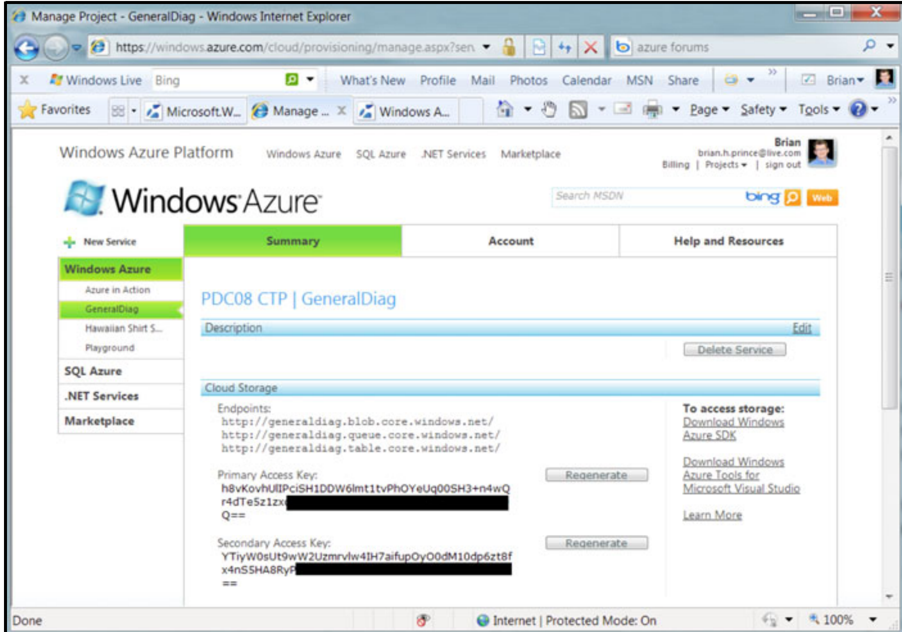


Figure 2.12 You need to create a storage account to store your logs and diagnostic data in. Creating a storage account is similar to creating a hosted service.

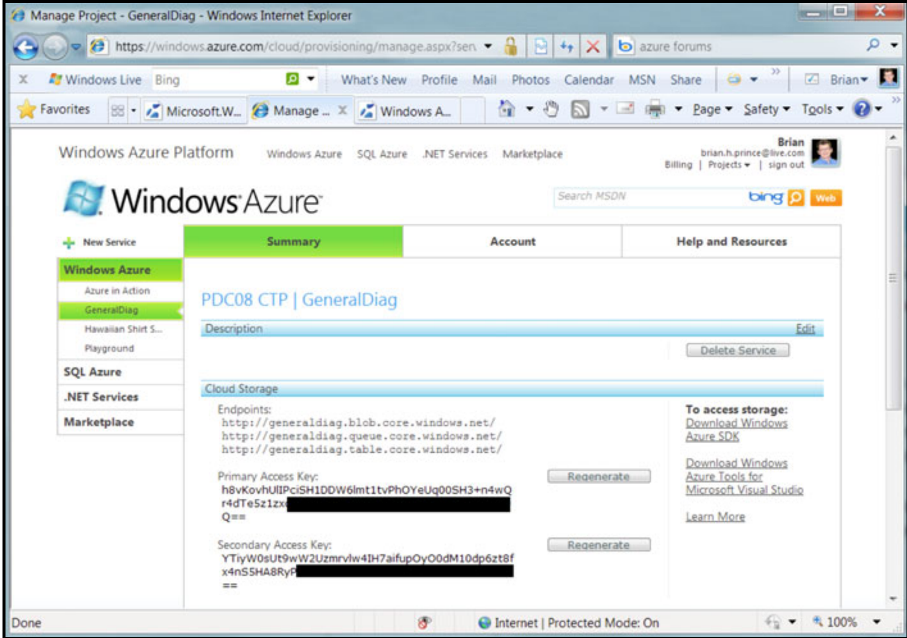


Figure 2.13 You can access your access keys after you've set up the storage account. These keys give anyone access to your data, so keep them safe. They're redacted here for that reason.

Open the ServiceConfiguration.cscfg file and replace the following line to include your storage account settings. This change reconfigures the diagnostic system to store log data into your storage account.

The old setting:

```
<Setting name="DiagnosticsConnectionString"
      value="UseDevelopmentStorage=true" />
```

The new setting:

```
<Setting name="DiagnosticsConnectionString"
     value="DefaultEndpointsProtocol=https;AccountName=YourAccountName;
     AccountKey=YourAccountKey" />
```

The `AccountName` parameter needs to be lowercase and URL friendly; we'll cover why in the chapters about Azure storage. The access key and the account name will be important later on when you want to connect to your storage in the cloud. Your application will still run locally; you've merely pointed the diagnostic system to store its logs in your cloud-based storage system. After you've made this change, you're ready to package your fabulous shirt website and move it to the cloud.

2.3.6 Packaging and deploying your application

You have an Azure account, you've created a hosted services project, and you have some parts for an application sitting on your local disk. Now you're going to take your application, and its configuration, and create a CS package for deployment.

> **What does CS stand for?**
>
> You'll see the acronym CS throughout the SDK and the tools for Azure. VB.NETers don't need to get upset; CS doesn't stand for C#, but for Cloud Service.

One way to create the package is to use the `CSpack` utility in the SDK. Although this is great for scripting, it can be tedious for those who live inside Visual Studio all day. A quicker method is to right-click the Azure project in your solution (named `AiA CH02 - Hawaiian Shirt Shop` in our sample), and choose Publish.

The `publish` command calls `CSpack` for you, which creates the package, opens a file explorer window that points to where the package files are placed, and then opens a browser window to your Azure project portal page. Sign in to your portal, and then select the project you want to deploy to in the Windows Azure section on the left. By default, only the production environment is displayed. Click the small arrow on the right side of the screen to show the staging environment. Click the Deploy button under the gelatinous staging cube, and use the screen to upload the package and the configuration file from the file explorer window that was opened for you. After you select these files, we suggest that you use the text box on the bottom of the form to give the deployment a label for history purposes. We usually use either a build number or

version number, but the label can be anything you want. In this example, we're going to enter `version 1, build 14, ch2 sample preview, sp1 spring refresh`.

After you click the Deploy button, the files are uploaded, and you're redirected to the service page on the Azure portal. You can choose to deploy directly to the production environment, but we recommend that you always deploy to the staging environment first. When the files are uploaded, the state of the service is set to Stopped.

Deployment is as easy as uploading a few files and clicking the Run button, at least for you. For Azure, the Fabric Controller kicks in, identifies some unused CPU cores, deploys a virtual server image, copies over your bits, wires up a VLAN for your instances only, reconfigures the load balancers, and then updates the service directories. While all this is happening, which can take from seconds to minutes, depending on what you're deploying, you'll see the status of your environment flip to Initializing; when Azure is finished, the status changes to Started. When things are deployed, the cube turns a nice shade of blue, and the status flips to Staging, as shown in figure 2.14. Note

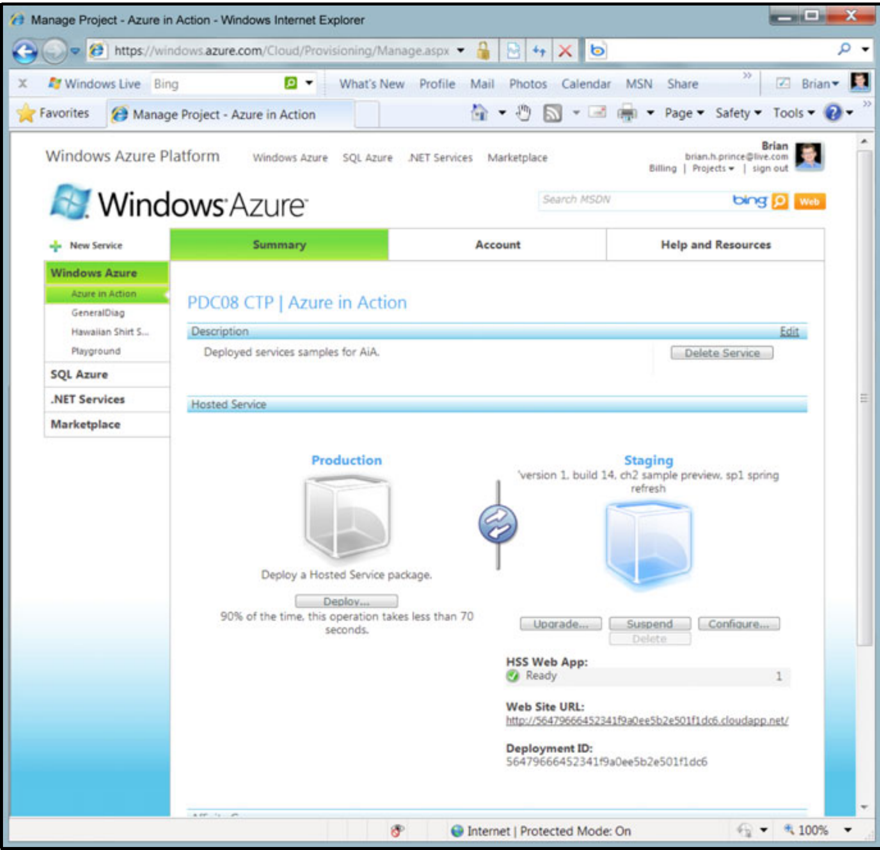


Figure 2.14 Your sample application has been deployed to the staging environment and is fully initialized. It's assigned an easy-to-remember GUID-based host name so that you can test your deployment.

that servers are reserved for you when you do the initial upload of your code with the Deploy button, and before you click Run. At this point, the cube turns blue, and you'll start being charged, even before you click Run. A little saying to remind you when you'll be charged is, "If the cube is grey, you're O.K. If the cube is blue, a bill is due."

While your application is in the staging environment, it has a temporary URL with a GUID as the host name of the service. The friendly name you picked when you created the Azure project won't be used until you migrate your application to production. Before you do that, test your application while it's in staging by using the temporary URL, which is located toward the bottom of the window.

When your application is running, the Run button changes to a Suspend button. If you click this button, you'll take your application offline. Azure is stopping the IIS application in the background when you do this, and even though it's suspended, you're still using a VM; the Azure hours for your billing continue to accrue.

Now that you've tested your application in the staging environment, you can move it to production.

2.3.7 Moving to production

Remember, when you move your application to production, you aren't moving the application or the settings. When you flip the button, Azure reconfigures the fabric to route all new traffic from the outside world to this particular instance. The old production environment becomes the new staging environment. Because nothing actually moves, you can easily roll back to the old version if things go wrong.

To move your application to production, click the circular arrows in the center of the screen. When you click them, you're prompted to make sure that you want to do that, and then the magic happens. Because Azure is making only a small configuration change, the cutover takes only a moment.

Congratulations! In only a few pages you've created a website where you can sell awesome shirts, tested it in a local simulation of the cloud, published it to staging, and promoted it to production. Now that business is rolling in, you'll want to monitor how much your application is costing you. Microsoft provides detailed information about your usage of the Azure platform on a regular basis. You can look at these reports at any time on the Azure portal and see where you are.

2.4 Summary

In this chapter we took you on a broad tour of how to start working with Azure. The SDK is important to be able to build Azure applications, including all of the different tools, APIs, and documentation needed to get started.

In a few pages, if you were following along, you developed and deployed a web application. Before either of us started using Windows Azure, we had never been able to deploy a web application with several load-balanced nodes without weeks or months of meetings, arguments, and planning the deployment and the production configuration.

The one thing you should've noticed is how little difference there is, code-wise, between what you would normally write for a website and what you write for Windows Azure. The web application you built has nothing special about it, and it'll run on a normal on-premises server. Azure is just another place to run your code, much more dynamic than any data center you've ever worked with, but the same. Our sample was simple and didn't delve into the more advanced scenarios, but the point is still valid. The strength of Azure, for a developer, is how *not* different it is from what you do every day.

Deployment is almost a trivial matter. It doesn't take a lot of effort to create a service, upload the package, and promote it to production. The simple staging and production environments make it easy to roll back to the old code in case the new deployment goes horribly wrong.

We'll look into what's under the covers in the next chapter, and see what Microsoft had to do to build such a powerful platform, yet keep it simple.

Part 2

Understanding the Azure service model

With the cloud basics and Windows Azure concepts under your belt, we dial it up a notch. In part 2, we look at all the parts of the service model.

Chapter 3 explains what the service model is, how Azure uses it, and how Azure works behind the scenes. A brilliant chapter if there ever was one.

The quality only gets better as we move into chapter 4, which discusses how to reference the Azure APIs in your code and how to exploit the service runtime.

In chapter 5, we trot out how to configure your service model using the configuration files and the portal. An exciting chapter, especially if you like XML and angle braces.

3

How Windows Azure works

This chapter covers

- How Microsoft built Azure
- What a cloud operating system is
- How your application is provisioned and managed in the cloud

Now that you have a basic understanding of what you can do with Azure, let's drill deeper into the pieces of Azure and how to best work with them. In this chapter, we'll discuss how Windows Azure is architected and how it does the cloud magic that it does. Understanding this background will help you develop better services, be a better person, and get the most out of your Azure infrastructure.

3.1 The big shift

When Azure was first announced at the PDC in 2008, Microsoft wasn't a recognized player in the cloud industry. It was the underdog to the giants Google and Amazon, which had been offering cloud services for years by that time. Building and

deploying Azure was a big bet for Microsoft. It was a major change in the company's direction, from where Microsoft had been and where it needed to go in the future. Up until that time, Microsoft had been a product company. It designed and built a product, burnt it to CD, and sold it to customers. Over time, the product was enhanced, but the product was installed and operated in the client's environment. The trick was to build the right product at the right time, for the right market.

With the addition of Ray Ozzie to the Microsoft culture, there was a giant shift toward services. Microsoft wasn't abandoning the selling of products, but it was expanding its expertise and portfolio to offer its products as services. Every product team at Microsoft was asked if what they were doing could be enhanced and extended with services. They wanted to do much more than just put Exchange in a data center and rent it to customers. This became a fundamental shift in how Microsoft developed code, how the code was shipped, and how it was marketed and sold to customers.

This shift toward services wasn't an executive whim, thought up during an exclusive executive retreat at a resort we'll never be able to afford to even drive by. It was based on the trends and patterns the leaders saw in the market, in the needs of their customers, and on the continuing impact of the internet on our world. Those in charge saw that people needed to use their resources in a more flexible way, more flexible than even the advances in virtualization were providing. Companies needed to easily respond to a product's sudden popularity as social networking spread the word. Modern businesses were screaming that six months was too long to wait for an upgrade to their infrastructure; they needed it now.

Customers were also becoming more sensitive to the massive power consumption and heat that was generated by their data centers. Power and cooling bills were often the largest component of their total data-center cost. Coupling this with a concern over global warming, customers were starting to talk about the greening of IT. They wanted to reduce the carbon footprint that these beasts produced. Not only did they want to reduce the power and cooling waste, but also the waste of lead, packing materials, and the massive piles of soda cans produced by the huge number of server administrators that they had to employ.

3.1.1 The data centers of yore

Microsoft is continually improving all the important aspects of its data centers. It closely manages all the costs of a data center, including power, cooling, staff, local laws, risk of disaster, availability of natural resources, and many other factors. While managing all this, it has designed its fourth generation of data centers. Microsoft didn't just show up at this party; it planned it by building on a deep expertise in building and running global data centers over the past few decades.

The first generation of data centers is still the most common in the world. Think of the special room with servers in it. It has racks, cable ladders, raised floors, cooling, uninterruptable power supplies (UPSs), maybe a backup generator, and it's cooled to a temperature that could safely house raw beef. The focus is placed on making sure

the servers are running; no thought or concern is given to the operating costs of the data center. These data centers are built to optimize the capital cost of building them, with little thought given to costs accrued beyond the day the center opens. (By the way, the collection of servers under your desk doesn't qualify as a Generation 1 data center. Please be careful not to kick a cord loose while you do your work.)

Generation 2 data centers take all the knowledge learned by running Generation 1 data centers and apply a healthy dose of thinking about what happens on the second day of operation. Ongoing operational costs are reduced by optimizing for sustainability and energy efficiency. To meet these goals, Microsoft powers its Quincy, Washington, data center with clean hydroelectric power. Its data center in San Antonio, Texas, uses recycled civic gray water to cool the data center, reducing the stress on the water sources and infrastructure in the area.

3.1.2 The latest Azure data centers

Even with the advances found in Generation 2 data centers, companies couldn't find the efficiencies and scale needed to combat rising facility costs, let alone meet the demands that the cloud would generate. The density of the data center needed to go up dramatically, and the costs of operations had to plummet. The first Generation 3 data center, located in Chicago, Illinois, went online on June 20, 2009. Microsoft considers it to be a mega data center, which is a class designation that defines how large the data center is. The Chicago data center looks like a large parking deck, with parking spaces and ramps for tractor trailers. Servers are placed into containers, called *CBlox*, which are parked in this structure. A smaller building that looks more like a traditional data center is also part of the complex. This area is for high-maintenance workloads that can't run in Azure.

CBlox are made out of the shipping containers that you see on ocean-going vessels and on eighteen wheelers on the highways. They're sturdily built and follow a standard size and shape that are easy to move around. One CBlox can hold anywhere from 1,800 to 2,500 servers. This is a massive increase in data-center density, 10 times more dense than a traditional data center. The Chicago mega data center holds about 360,000 servers and is the only primary consumer of a dedicated nuclear power plant core run by Chicago Power & Light. How many of your data centers are nuclear powered?

Each parking spot in the data center is anchored by a refrigerator-size device that acts as the primary interconnect to the rest of the data center. Microsoft developed a standard coupler that provides power, cooling, and network access to the container. Using this interconnect and the super-dense containers, massive amounts of capacity can be added in a matter of hours. Compare how long it would take your company to plan, order, deploy, and configure 2,500 servers. It would take at least a year, and a lot of people, not to mention how long it would take to recycle all the cardboard and extra parts you always seem to have after racking a server. Microsoft's goal with this strategy is to make it as cheap and easy as possible to expand capacity as demand increases.

The containers are built to Microsoft's specifications by a vendor and delivered on site, ready for burn-in tests and allocation into the fabric. Each container includes networking gear, cooling infrastructure, servers, and racks, and is sealed against the weather.

Not only are the servers now packaged and deployed in containers, but the necessary generators and cooling machinery are designed to be modular as well. To set up an edge data center, one that's located close to a large-demand population, all that's needed is the power and network connections, and a level paved surface. The trucks with the power and cooling equipment show up first, and the equipment is deployed. Then the trucks with the computing containers back in and drop their trailers, leaving the containers on the wheels that were used to deliver them. The facility is protected by a secure wall and doorway with monitoring equipment. The use of laser fences is pure speculation and just a rumor, as far as we know. The perimeter security is important, because the edge data center doesn't have a roof! Yes, no roof! Not using a roof reduces the construction time and the cooling costs. A roof isn't needed because the containers are completely sealed.

Microsoft opened a second mega data center, the first outside the United States, in Dublin, Ireland, on July 1, 2009. When Azure became commercially available in January 2010, the following locations were known to have an Azure data center: Texas, Chicago, Ireland, Amsterdam, Singapore, and Hong Kong. Although Microsoft won't tell where all its data centers are for security reasons, it purports to have more than 10 and fewer than 100 data centers. Microsoft already has data centers all over the world to support its existing services, such as Virtual Earth, Bing Search, Xbox Live, and others. If we assume there are only 10, and each one is as big as Chicago, then Microsoft needs to manage 3.5 million servers as part of Azure. That's a lot of work.

3.1.3 How many administrators do you need?

Data centers are staffed with IT pros to care and feed the servers. Data centers need a lot of attention, ranging from hardware maintenance to backup, disaster recovery, and monitoring. Think of your company. How many people are allocated to manage your servers? Depending on how optimized your IT center is, the ratio of person-to-servers can be anywhere from 1:10 to 1:100. With that ratio, Microsoft would need 35,000 server managers. Hiring that many server administrators would be hard, considering that Microsoft employs roughly 95,000 people already.

To address this demand, Azure was designed to use as much automation as possible, using a strategy called *lights-out operations*. This strategy seeks to centralize and automate as much of the work as possible by reducing complexity and variability. The result is a person-to-servers ratio closer to 1:30,000 or higher.

Microsoft is achieving this level of automation mostly by using its own off-the-shelf software. Microsoft is literally eating its own dog food. It's using System Center Operations Manager and all the related products to oversee and automate the management of the underlying machines. It's built custom automation scripts and profiles, much like any customer would do.

One key strategy in effectively managing a massive number of servers is to provision them with identical hardware. In traditional data centers where we've worked, each year brought the latest and greatest of server technology, resulting in a wide variety of technology and hardware diversity. We even gave each server a distinct name, such as Protoss, Patty, and Zelda. With this many servers, you can't name them; you have to number them. Not just by server, but by rack, room, and facility. Diversity is usually a great thing, but not when you're managing millions of boxes.

The hardware in each Azure server is optimized for power, cost, density, and management. The optimization process drives exactly which motherboard, chipset, and every other component needs to be in the server; this is truly bang for your buck in action. Then that server recipe is kept for a specific lifecycle, only moving to a new bill of materials when there are significant advantages to doing so.

3.1.4 Data center: the next generation

Microsoft isn't done. It's already spent years planning the fourth generation of data centers. Much like the edge data center we described previously, the whole data center is located outside. The containers make it easy to scale out the computing resources as demand increases; prior generations of data centers had to have the complete data center shell built and provisioned, which meant provisioning the cooling and power systems as if the data center were at maximum capacity from day one. The older systems were too expensive to expand dynamically. The fourth generation data centers are using an extendable spine of infrastructure that the computing containers need, so that both the infrastructure and the computing resources are easily scaled out (see figure 3.1). All of this is outside, in a field of grass, without a roof. They'll be the only data centers in the world that need a grounds crew.

Figure 3.1 Generation 4 data centers are built on extensible spines. This configuration makes it easy to add not only computational capacity, but the required infrastructure as well, including power and cooling.

OK, you're impressed. Microsoft has a lot of servers, some of them are even outside, and all the servers are managed in an effective way. But how does the cloud really work?

3.2 Windows Azure, an operating system for the cloud

Think of the computer on your desk today. When you write code for that computer, you don't have to worry about which sound card it uses, which type of printer it's connected to, or which or how many monitors are used for the display. You don't worry, to a degree, about the CPU, about memory, or even about how storage is provided (solid-state drive [SSD], carrier pigeon, or hard disk drive). The operating system on that computer provides a layer of abstraction away from all of those gritty details, frees you up to focus on the application you need to write, and makes it easy to consume the resources you need. The desktop operating system protects you from the details of the hardware, allocates time on the CPU to the code that's running, makes sure that code is allowed to run, plays traffic cop by controlling shared access to resources, and generally holds everything together.

Now think of that enterprise application you want to deploy. You need a DNS, networking, shared storage, load balancers, plenty of servers to handle load, a way to control access and permissions in the system, and plenty of other moving parts. Modern systems can get complicated. Dealing with all of that complexity by hand is like compiling your own video driver; it doesn't provide any value to the business. Windows Azure does all this work, but on a much grander scale and for distributed applications (see figure 3.2) by using something called the *fabric*. Let's look into this fabric and see how it works.

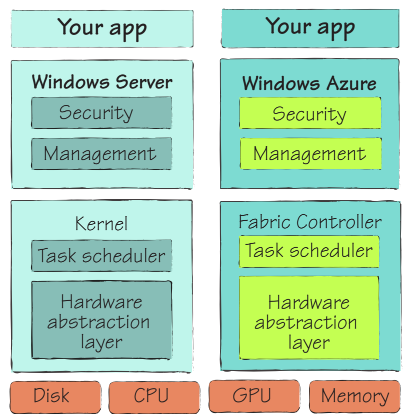


Figure 3.2 The Fabric Controller is like the kernel of your desktop operating system. It's responsible for many of the same tasks, including resource sharing, code security, and management.

Windows Azure takes care of the whole platform so you can focus on your application. The term *fabric* is used because of the similarity of the Azure fabric to a woven blanket. Each thread on its own is weak and can't do a lot. When they're woven together into a fabric, the whole blanket becomes strong and warm. The Azure fabric consists of thousands of servers, woven together and working as a cohesive unit. In Azure, you don't need to worry about which hardware, which node, what underlying operating system, or even how the nodes are load balanced or clustered. Those are just gritty details best left to someone else. You just need to worry about your application and whether it's operating effectively. How much time do you spend wrangling with these details for your on-premises projects? It's probably at least 10–20 percent of the total project cost in meetings alone. There are savings to be gained by abstracting away these issues.

In fact, Azure manages much more than just servers. There are plenty of other assets that are managed. Azure manages routers, switches, IP addresses, DNS servers, load balancers, and dynamic virtual local area networks (VLANs). In a static data center, managing all these assets is a complex undertaking. It's even more complex when you're managing multiple data centers that need to operate as one cohesive pool of resources, in a dynamic and real-time way.

If the fabric is the operating system, then the *Fabric Controller* is the kernel.

3.3 The Fabric Controller

Operating systems have at their core a kernel. This kernel is responsible for being the traffic cop in the system. It manages the sharing of resources, schedules the use of precious assets (CPU time), allocates work streams as appropriate, and keeps an eye on security. The fabric has a kernel called the Fabric Controller (FC). Figure 3.3 shows the relationship between Azure, the fabric, and the FC. Understanding these relationships will help you get the most out of the platform.

The FC handles all of the jobs a normal operating system's kernel would handle. It manages the running servers, deploys code, and makes sure that everyone is happy and has a seat at the table.

The FC is an Azure application in and of itself, running multiple copies of itself for redundancy's sake. It's largely written in managed code. The FC contains the complete state of the fabric internally, which is replicated in real time to all the nodes that are part of the FC. If one of the primary nodes goes offline, the latest state information is available to the remaining nodes, which then elect a new primary node.

The FC manages a state machine for each service deployed, setting a goal state that's based on what the service model for the service requires. Everything the FC does is in an effort to reach this state and then to maintain that state when it's reached. We'll go into the details of what the service model is in the next few pages, but for now, just think of it as a model that defines the needs and expectations that your service has.

The FC is obviously very busy. Let's look at how it manages to seamlessly perform all these tasks.

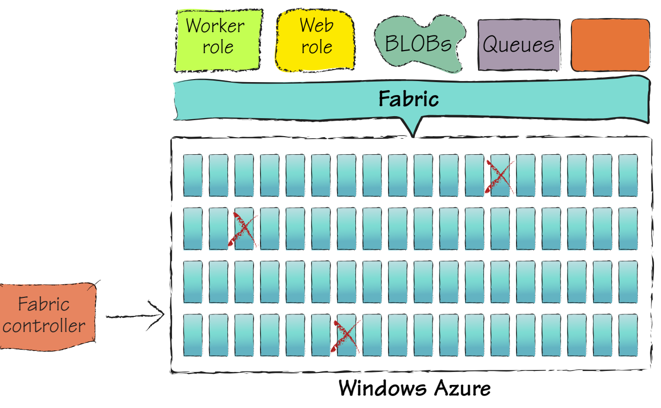


Figure 3.3 The relationship between Azure, the fabric, and the Fabric Controller (FC). The fabric is an abstract model of the massive number of servers in the Azure data center. The FC manages everything. For example, it recovers failed servers and moves your application to a healthy server.

3.3.1 How the FC works: the driver model

The FC follows a driver model, just like a conventional OS. Windows has no idea how to specifically work with your video card. What it does know is how to speak to a video driver, which in turn knows how to work with a specific video card. The FC works with a series of drivers for each type of asset in the fabric. These assets include the machines, as well as the routers, switches, and load balancers.

Although the variability of the environment is low today, over time new types of each asset are likely to be introduced. The goal is to reduce unnecessary diversity, but you'll have business needs that require breadth in the platform. Perhaps you'll get a software load balancer for free, but you'll have to pay a little bit more per month to use a hardware load balancer. A customer might choose a certain option, such as a hardware load balancer, to meet a specific need. The FC would have a different driver for each piece of infrastructure it controls, allowing it to control and communicate with that infrastructure.

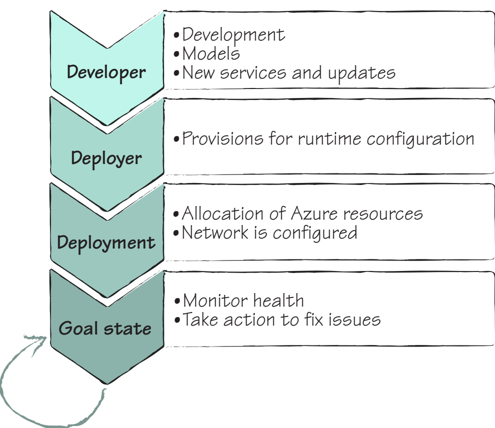


Figure 3.4 How the lifecycle of an Azure service progresses towards a running state. Each role on your team has a different set of responsibilities. From here the FC does what it needs to make sure your servers are always running.

The FC uses these drivers to send commands to each device that help these devices reach the desired running state. The commands might create a new VLAN to a switch or allocate a pool of virtual IP addresses. These commands help the FC move the state of the service towards the goal state. Figure 3.4 shows how a service progresses to the goal state, from the developer writing the code and defining the service model to the FC allocating and managing the resources the service requires.

While the FC is moving all your services toward the running state, it's also allocating resources and managing the health of the nodes in the fabric and of your services.

3.3.2 Resource allocation

One of the key jobs of the FC is to allocate resources to services. It analyzes the service model of the service, including the fault and update domains, and the availability of resources in the fabric. Using a greedy resource allocation algorithm, it finds which nodes can support the needs of each instance in the model. When it has reserved the capacity, the FC updates its data structures in one transaction. After the update, the goal state of each node is changed, and the FC starts moving each node towards its goal state by deploying the proper images and bits, starting up services, and issuing other commands through the driver model to all the resources needed for the change.

3.3.3 Instance management

The FC is also responsible for managing the health of all of the nodes in the fabric, as well as the health of the services that are running. If it detects a fault in a service, it tries to remediate that fault, perhaps by restarting the node or taking it offline and replacing it with a different node in the fabric.

When a new container is added to the data center, the FC performs a series of burn-in tests to ensure that the hardware delivered is working correctly. Part of this process results in the new resource being added into the inventory for the data center, making it available to be allocated by the FC.

If hardware is determined to be faulty, either during installation or during a fault, the hardware is flagged in the inventory as being unusable and is left alone until later. When a container has enough failures, the remaining workloads are moved to different containers and then the whole container is taken offline for repair. After the problems have been fixed, the whole container is retested and returned into service.

3.4 The service model and you

The driving force behind what the FC does is the service model that you define for your service (see figure 3.5). You define the service model indirectly by defining the following things when you're developing a service:

- Some configuration about what the pieces to your service are
- How the pieces communicate
- Expectations you have about the availability of the service

The service model is broken into two pieces of configuration and is deployed with your service. Each piece focuses on a different aspect of the model. In the following sections, you're going to learn about these configuration pieces and how to customize them. We'll also show you how best to manage all the pieces of your configuration.

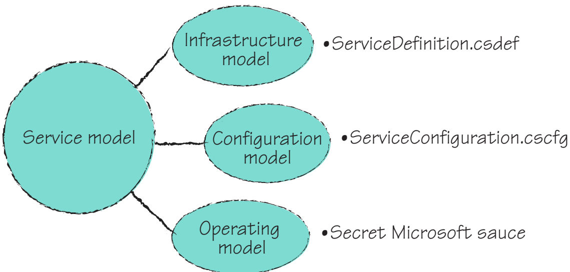


Figure 3.5 The service model consists of several different pieces of information. This model helps Azure run your application correctly.

3.4.1 Defining configuration

Your solution in Visual Studio contains these two pieces of configuration in different files, both of which are found in the Azure Service project in your solution:

- Service definition file (ServiceDefinition.csdef)
- Service configuration file (ServiceConfiguration.cscfg)

The service definition file defines what the roles and their communication endpoints are in your service. This includes public HTTP traffic for a website, or the endpoint

details for a web service. You can also configure your service to use local storage (which is different from Azure storage) and any custom configuration elements of the service configuration file. The service definition can't be changed at runtime; any change requires a new deployment of your service. Your service is restricted to using only the network endpoints and resources that are defined in this model. We're going to look at the service definition file in depth in chapter 4; for now you can think of this piece of the configuration as defining what the infrastructure of your service is, and how the parts fit together.

The service configuration file, which we'll discuss in detail in chapter 5, includes the entire configuration needed for the role instances in your service. Each role has its own dedicated part of the configuration. The contents of the configuration file can be changed at runtime, which removes the need to redeploy your application when some part of the role configuration changes. You can also access the configuration in code, similar to how you might read a web.config file in an ASP.NET application.

3.4.2 Adding a custom configuration element

In many applications, you store connection strings, default settings, and secret passwords (please don't!) in the app.config or web.config file. You'll often do the same with an Azure application. First, you need to declare the format of the new configuration setting in the .csdef file by adding a `ConfigurationSettings` node inside the role you want the configuration to belong to:

```
<ConfigurationSettings>
  <Setting name="BannerText"/>
</ConfigurationSettings>
```

Adding this node defines the schema of the .cscfg file for that role, which strongly types the configuration file itself. If there's an error in the configuration file during a build, you'll receive a compiler warning. This is a great feature because there's nothing worse than deploying code when there's a simple little problem in a configuration file.

Now that you've told Azure the new format of your configuration files, namely, that you want a new setting called `BannerText`, you can add that node to the service configuration file. Add the following XML into the appropriate role node in the .cscfg file:

```
<ConfigurationSettings>
  <Setting name="BannerText" value="KlatuBaradaNikto"/>
</ConfigurationSettings>
```

During runtime, you want to read in this configuration data and use it for some purpose. Remember that all configuration settings are stored as strings and must be cast to the appropriate type as needed. In this case, you want a string to assign to your label control text, so that you can use it as is.

```
txtPassword.Text = RoleEnvironment.GetConfigurationSettingValue("BannerText");
```

Having lines of code like this all over your application can get messy and hard to manage. Sometimes developers consolidate their configuration access code into one class. This class's only job is to be a façade into the configuration system.

3.4.3 Centralizing file-reading code

It's a best practice to move your entire configuration file-reading code from wherever it's sprinkled into a `ConfigurationManager` class of your own design. Many people use the term *service* instead of *manager*, but we think that the term *service* is too overloaded and that *manager* is just as clear. Moving your code centralizes all the code that knows how to read the configuration in one place, making it easier to maintain. More importantly, it removes the complexity of reading the configuration from the relying code, which illustrates the principle of *separation of concerns*. Moving the code to a centralized location also makes it easier to mock out the implementation of the `ConfigurationManager` class for easier testing purposes (see figure 3.6). Over time, when the APIs for accessing configuration change or if the location of your configuration changes, you'll have only one place to go to make the changes you need.

Reading configuration data in this manner might look familiar to you. You've probably done this for your current applications, reading in the settings stored in a web.config or an app.config file. When migrating an existing application to Azure, you might be tempted to keep the configuration settings where they are. Although keeping them in place reduces the amount of change to your code as you migrate it to Azure, it does come at a cost. Unfortunately, the configuration files that are part of your roles are frozen and are read-only at runtime; you can't make changes to them after your package is deployed. If you want to change settings at runtime, you'll need to store those settings in the .cscfg file. Then, when you want to make a change, you only have to upload a new .cscfg file or click Configure on the service management page in the portal.

The FC takes these configuration files and builds a sophisticated service model that it uses to manage your service. At this time, there are about three different core model templates that all other service models inherit from. Over time, Azure will expose more of the service model to the developer, so that you can have more fine-grained control over the platform your service is running on.

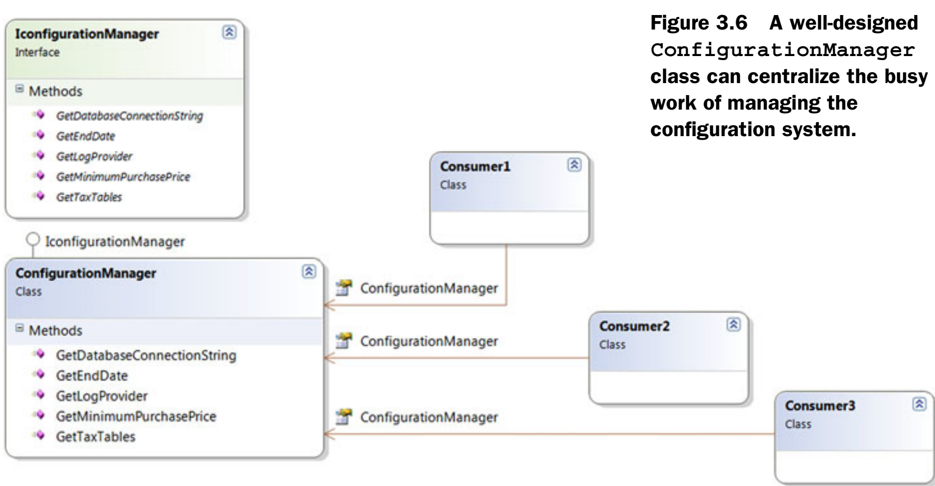


Figure 3.6 A well-designed `ConfigurationManager` class can centralize the busy work of managing the configuration system.

3.4.4 The many sizes of roles

Each role defined in your service model is basically a template for a server you want to be deployed in the fabric. Each role can have a different job and a different configuration. Part of that configuration includes local storage and the number of instances of that role that should be deployed. How these roles connect and work together is part of why the service model exists.

Because each role might have different needs, there are a variety of VM sizes that you can request in your model. Table 3.1 lists each VM size. Each step up in size doubles the resources of the size below it.

Table 3.1 The available sizes of the Azure VMs

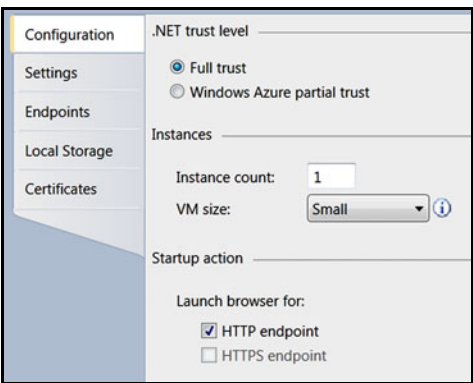
VM size	Dedicated CPU cores	Available memory	Local disk space
Small	1	1.7 GB	250 GB
Medium	2	3.5 GB	500 GB
Large	4	7 GB	1,000 GB
Extra large	8	15 GB	2,000 GB

Each size is basically a slice of how big a physical server is, which makes it easy to allocate resources and keeps the numbers round. Because each physical server has eight CPU cores, allocating an extra-large VM to a role is like dedicating a whole physical machine to that instance. You'll have all the CPU, RAM, and disk available on that machine. Which size you want is defined in the ServiceDefinition.csdef file on a role-by-role basis. The default size, if you don't declare one, is small. To change the default size, add the following code, substituting `ExtraLarge` with the size that you want:

```
<WorkerRole name="ImageCompresser" vmsize="ExtraLarge">
```

If you're using Visual Studio 2010, you can define the role configuration by double-clicking the name of your web role in the Roles folder of your Cloud Service project. Choose Properties and click the Configuration tab, as shown in figure 3.7.

The service model is also used to define fault domains and update domains, which we'll look at next.



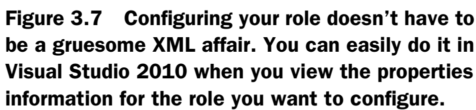


Figure 3.7 Configuring your role doesn't have to be a gruesome XML affair. You can easily do it in Visual Studio 2010 when you view the properties information for the role you want to configure.

3.5 It's not my fault

Fault domains and update domains determine what portions of your service can be offline at the same time, but for different reasons. They're the way that you define your uptime requirements to the FC and how you describe how your service updates will happen when you have new code to deploy.

Let's examine each type of domain in detail. Then we'll present a service model scenario that shows you how fault and update domains help increase fault tolerance in your cloud service.

3.5.1 Fault domains

Fault domains are used to make sure that a set of elements in your service isn't tied to a single point of failure. Fault domains are based more on the physical structure of the data center than on your architecture. Your service should typically have three or more fault domains. If you have only one fault domain, all the parts of your service could potentially be running on one rack, in the same container, connected to the same switch. If there's any failure in that chain, there's a high likelihood of catastrophic failure for your service. If that rack fails, or the switch in use fails, then your service is completely offline. By breaking your service into several fault domains, the FC ensures that those fault domains don't share any dependent infrastructure, which protects your service against single points of failure.

In general, the FC will define three fault domains, meaning that only about a third of them can become unavailable because of a single fault. In a failure scenario, the FC immediately tries to deploy your roles to new nodes in the fabric to make up for the failed nodes. Currently, the Azure SDK and service model don't let you define your own number of fault domains; the default number is thought to be three domains.

3.5.2 Update domains

The second type of domain defined in the service model is the *update domain*. The concept of an update domain is similar to a fault domain. An update domain is the unit of update you've declared for your service. When performing a rolling update, code changes are rolled out across your service one update domain at a time. Cloud services tend to be big and tend to always need to be available. The update domain allows a rolling update to be used to upgrade your service, without having to bring the entire service down. These domains are usually defined to be orthogonal to your fault domains. In this manner, if an update is being pushed out while there's a massive fault, you won't lose all of your resources, just a piece of them.

You can define the number of update domains for your service in your ServiceDefinition.csdef file as part of the `ServiceDefinition` tag at the top of the file.

```
<ServiceDefinition xmlns="http://schemas.microsoft.com/ServiceHosting/
2008/10/ServiceDefinition"
name="HawaiianShirtShop"
upgradeDomainCount="3">
```

If you don't define your own update domain setting, the service model will default to five update domains. Your role instances are assigned to update domains as they're started up, and the FC tries to keep the domains balanced with regard to how many instances are in each domain.

3.5.3 A service model example

If you had a service running on Azure, you might need six role instances to handle the demand on your service, but you should request nine instances instead. You request more than you need because you want a high degree of tolerance in your architecture. As shown in figure 3.8, you would have three fault domains and three update domains defined. If there's a fault, only a third of your nodes are affected. Also, only a third of the nodes will ever be updated at one time, controlling the number of nodes taken out of service for updates, as well as reducing the risk of any update taking down the whole service.

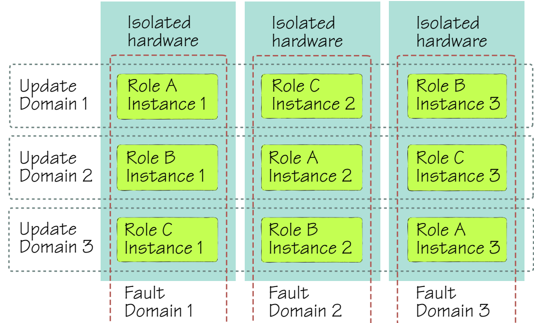


Figure 3.8 Fault and update domains help increase fault tolerance in your cloud service. This figure shows three instances of each of three roles.

In this scenario, a broken switch might take down the first fault domain, but the other two fault domains would not be affected and would keep operating. The FC can manage these fault domains because of the detailed models it has for the Azure data center assets.

The cloud is not about perfect computing, it's about deploying services and managing systems that are fault tolerant. You need to plan for the faults that are inevitable. The magic of cloud computing makes it easy to scale big enough so that a few node failures don't really impact your service.

All this talk about service models and an overlord FC is nice, but at the end of the day, the cloud is built from individual pieces of hardware. There's a lot of hardware, and it all needs to be managed in a hands-off way. There are several approaches to applying updates to a service that's running. You'll see in the next section that you can perform either manual or automated rolling upgrades, or you can perform a full static upgrade (also called a VIP swap).

3.6 Rolling out new code

No matter how great your code is, you'll have to perform an upgrade at some point if for no other reason than to deploy a new feature a user has requested. It's important that you have a plan for updating the application and have a full understanding of the moving parts. There are two major ways to roll out an upgrade: a *static upgrade* or a *rolling upgrade*.

When you perform a static upgrade, you do everything at once and you have to take down your system, at least for a while. You should carefully plan your application architecture to avoid a static upgrade because it impacts the uptime of your service and can be more complicated to roll out. A rolling upgrade keeps your service up and running the whole time. You should always consider performing the upgrade in the staging environment first to make sure the deployment goes well. After a full battery of end-to-end and integration tests are passed, you can proceed with your plans for the production environment.

If the number of endpoints for a role has changed, or if the port numbers have changed, you won't be able to do either a static or a rolling upgrade. You'll be forced to tear down the deployment and redeploy.

3.6.1 Static upgrades

A static upgrade is sometimes referred to as a *forklift upgrade* because you're touching everything all at once. You usually need to do a static upgrade when there's a significant change in the architecture and plumbing of your application. Perhaps there's a whole new architecture of how the services are structured and the database has been completely redesigned. In this case, it can be hard to upgrade just one piece at a time because of interdependencies in the system. This type of upgrade is required if you're changing the service model in any way.

This approach is also called a *VIP swap* because the FC is swapping the virtual IP addresses that are assigned to your resources. When a swap is done, the old staging environment becomes your new production environment and your old production environment becomes your new staging environment (see figure 3.9). This can happen pretty fast, but your service will be down while it's happening and you need to

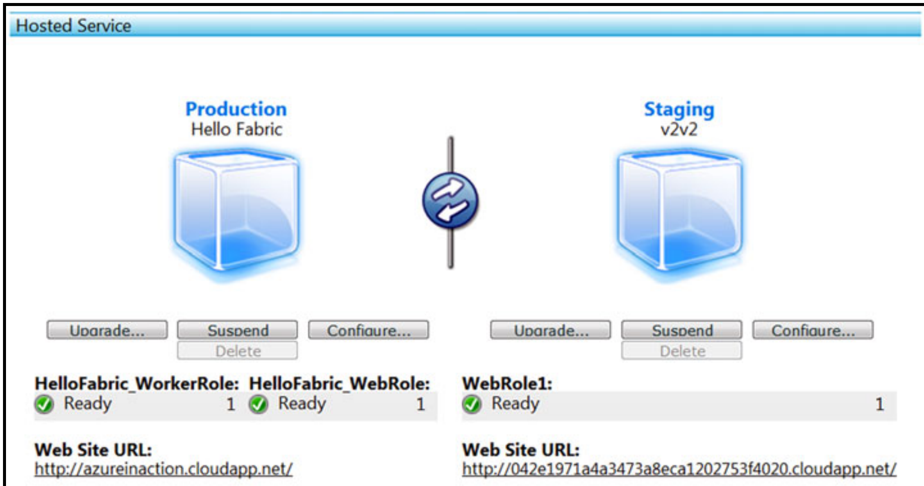


Figure 3.9 Performing a VIP swap, or static upgrade, is as easy as clicking the arrows. If things go horribly awry, you can always swap back to the way things were. It's like rewind for your environment.

plan for that. The one great advantage to this approach is that you can easily swap things back to the way they were if things don't work out.

Your upgrade plan should consider how long the new staging (aka old production) environment should stay around. You might want to keep it for a few days until you know the upgrade has been successful. At that point, you can completely tear down the environment to save resources and money.

To perform a VIP swap, log in to the Azure portal, choose the service that you want to upgrade in the Windows Azure section, and then click the Summary tab. Next, deploy your new application version to the staging environment. After everything is all set up and you're happy with it, click the circular button in the middle. The change-over takes only a few minutes. If the new version isn't working as expected, you can easily click the button again and swap the two environments back where they came from. Voila! The old version is back online.

You can also use the service management API to perform the swap operation. This is one reason why you want to make sure that you've named your deployments clearly, at least more clearly than we did in this example.

VIP swaps are nice, but some customers need more flexibility in the way they perform their rollouts. For them, there's the rolling upgrade.

3.6.2 Rolling upgrades

If your roles are carrying state and you don't want to lose that state as you completely change to a new set of servers, then rolling upgrades are for you. Or maybe you want to upgrade the instances of a specific role instead of all of the roles. For example, you might want to deploy an updated version of the website, without impacting the processing of the shopping carts that's being performed by the backend worker roles. Remember that when doing a rolling upgrade, you can't change the service model of the service that you're upgrading. If you've changed the structure of the service configuration, the number of endpoints, or the number of roles, you'll have to do a VIP swap instead.

There are two types of rolling upgrades: the automatic and the manual. When you perform an automatic rolling upgrade, the FC drains the traffic to the set of instances that's in the first update domain (they're numbered, starting with 0) by removing them from the load balancer's configuration. After the traffic is drained, the instances are stopped, the new code is deployed, and then the instances are restarted. After they're back up and running, they're added back into the load balancer's list of machines to route traffic to. At this point, the FC moves on to the next update domain in the list. It'll proceed in this fashion until all the update domains have been serviced. Each domain should take only a few minutes.

If your situation requires that you control how the progression moves from one domain to the next, you can choose to do a manual rolling upgrade. When you choose this option, the FC stops after updating a domain and waits for your permission to move on to the next one. This gives you a chance to check the status of the machines and the environment before moving forward with the rollout.

Figure 3.10 Performing a rolling upgrade is easy. Click Upgrade on the Summary page for the service to see this page and choose your options. You can upgrade all of the roles or just one role during an upgrade.

To perform a rolling upgrade, log in to the Azure portal, choose the service that you want to upgrade in the Windows Azure section, and then click the Summary tab. Click the Upgrade button for the deployment you want to upgrade. You're presented with some options, as shown in figure 3.10.

You can choose to perform an automatic or a manual upgrade. You can upgrade all the roles in the package or just one of them. As in a normal deployment, you also need to provide a service package, configuration, and a deployment name.

If you choose to upgrade a single role, then only the instances for that role in each domain are taken offline for upgrading. The other role instances are left untouched.

You can also perform a rolling upgrade by using the service management API. When you use the management API, you have to store the package in BLOB storage before starting the process. As with a VIP swap, you need to post a command to a specific URL (all these commands are covered in detail in chapter 18). Customize the URL to match the settings for the deployment you want to upgrade:

```
https://management.core.windows.net/<subscription-id>/services/
     hostedservices/<service-name>
/deployments/<deployment-name>/?comp=upgrade
```

The body of the command needs to contain the elements shown in the following code. You need to change the code to supply the parameters that match your situation. The following sample performs a fully automatic upgrade on all the roles.

```
<?xml version="1.0" encoding="utf-8"?>
<UpgradeDeployment xmlns="http://schemas.microsoft.com/windowsazure">
  <Mode>auto</Mode>
  <PackageUrl>http://azureinaction.blob.core.windows.net/
deployment_container/new_code.cspkg </PackageUrl>
  <Configuration>***the contents of the config file***</Configuration>
  <Label>v3.2</Label>
</UpgradeDeployment>
```

Performing a manual rolling update with the service management API is a little trickier, and requires several calls to the `WalkUpgradeDomain` method. The upgrades are performed in an asynchronous manner; the first command starts the process. As the upgrade is being performed, you can check on the status by using `Get Operation Status` with the operation ID that was supplied to you when you started the operation.

We've covered how to upgrade running instances and talked about what the fabric is. Now we'll go one level deeper and explore the underlying environment.

3.7 The bare metal

No one outside of the Azure team truly knows the nature of the underlying servers and other hardware, and that's OK because it's all abstracted away by the cloud OS. But you can still look at how your instances are provisioned and how automation is used to do this without hiring the entire population of southern Maine to manage it.

Each instance is really a VM running Windows Server 2008 Enterprise Edition x64 bit, on top of *Hyper-V*. Hyper-V is Microsoft's enterprise virtualization solution, and it's available to anyone. Hyper-V is based on a hypervisor, which manages and controls the virtual servers running on the physical server. One of the virtual servers is chosen to be the host OS. The host OS is a virtual server as well, but it has the additional responsibilities of managing the hypervisor and the underlying hardware.

Hyper-V has two features that help in maximizing the performance of the virtual servers, while reducing the overall cost of running those servers. One of these features is core-and-socket parking; the other is the reduced footprint of Hyper-V itself. Core-and-socket parking needs to be supported by the physical CPU.

Let's drill way down into the workings of Hyper-V, how the virtual servers connect to it, and the processes of booting up these servers and getting your instances up and running.

3.7.1 Free parking

The first feature of Hyper-V is core-and-socket parking. Hyper-V can monitor the use of each core and CPU (which is in a socket on the motherboard) as a whole. Hyper-V moves the processes around on the cores to consolidate the work to as few cores as possible. Any cores not needed at that time are put into a low energy state. They can come back online quickly if needed but consume much less power while they wait.

Hyper-V can do this at the socket level as well. If it notices that each CPU socket is being used at only 10 percent of capacity, for example, it can condense the workload to one socket and park the unused sockets, placing them in a low energy state. This helps data centers use less power and require less cooling. In Azure, you have exclusive access to your assigned CPU core. Hyper-V won't condense your core with someone else's. It will, however, turn off cores that aren't in use.

3.7.2 A special blend of spices

The version of Hyper-V used by Azure is a special version that the team created by removing anything they didn't need. Their goal was to reduce the footprint of Hyper-V as much as possible to make it faster and easier to manage. Because they knew exactly the type of hardware and guest operating systems that will run on it, they could rip out a lot of code. For example, they removed support for 32-bit hosts and guest machines, support needed for other types of operating systems, and support for hardware they weren't supporting at all.

Not stopping there, they further tuned the hypervisor scheduler for better performance while working with cloud data-center workloads. They wanted the scheduler to be more predictable in its use of resources and fairer to the different workloads that were running, because each would be running at the same priority level. They also enhanced Hyper-V to support a heavier I/O load on the VM bus.

3.7.3 Creating instances on the fly

When a new server is ready to be used, it's booted. At this point, it's a naked server with a bare hard drive. You can see the steps involved in starting the server, adding an instance to your service, and adding an additional server in figure 3.11.

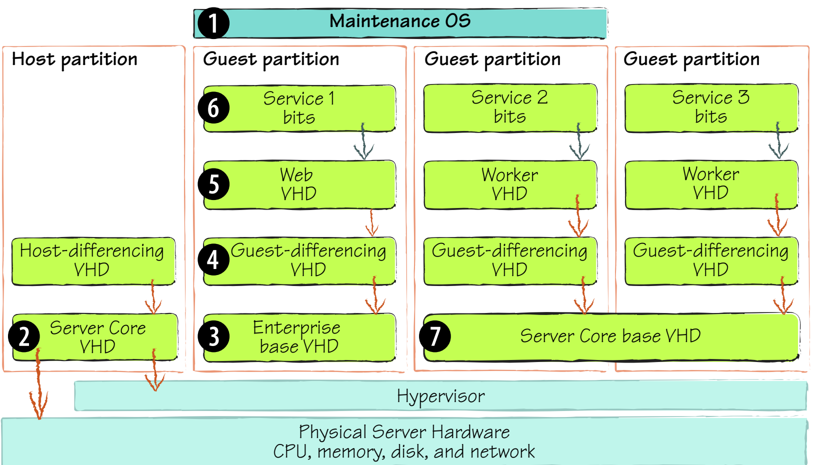


Figure 3.11 The structure of a physical server and virtual instance servers in Azure. This figure illustrates the process involved in starting the server (1 and 2). It also shows the process of starting an instance and adding it to your service (3 through 6), and adding another virtual server (7). All these steps are coordinated by the FC and take only a few minutes.

During boot up, the server locates a maintenance OS on the network, using standard Preboot Execution Environment (PXE) protocols. (PXE is a process for booting to an operating system image that can be found on the network.) The server downloads the image and boots to it (1 in figure 3.11).

The maintenance OS

The maintenance OS is based on Windows Preinstallation Environment (Windows PE). It's a thin OS that's used by many IT organizations for low-level troubleshooting and maintenance. The tools and protocols for Windows PE are available on any Windows server and are used by a lot of companies to easily distribute machine images and automate deployment.

The maintenance OS connects with the FC and acts as an agent for the FC to execute any local commands needed to prepare the disk and machine. The agent prepares the local disk and streams down a Windows Server 2008 Server Core image to the local disk 2. This image is a virtual hard drive (VHD) and is a common file format used to store the contents of hard drives for VMs. (VHDs are large files representing the complete or partial hard drive for a VM.) The machine is then reconfigured to boot from this core VHD. This image becomes the host OS that manages the machine and interacts with the hypervisor. The host OS is Windows Server 2008 Core because almost all but the most necessary modules have been removed from the operating system. You might be running this in your own data center.

The Azure team worked with the Windows Server team to develop the technology needed to boot a machine natively from a VHD that's stored on the local hard drive. The Windows 7 team liked the feature so much that they added it to their product as well. Being able to boot from a VHD is a key component of the Azure automation.

After the machine has rebooted using the host OS image, the maintenance OS is removed and the FC can start allocating resources from the machine to services that need to be deployed. A base OS image is selected from the prepared image library that'll meet the needs of the service that's being deployed 3. This image (a VHD file) is streamed down to the physical disk. The core OS VHDs are marked as read-only, allowing multiple service instances to share a single image. A differencing VHD is stacked on top of the read-only base OS VHD to allow for changes specific to that virtual server 4. Different services can have different base OS images, based on the service model applied to that service.

On top of the base OS image and attached to it is an application VHD that contains additional requirements for your service 5. The bits for your service are downloaded to the application VHD 6, and then the stack is booted. As it starts, the stack reports its health to the FC. The FC then enrolls the stack into the service group, configuring the VLAN assigned to your service and updating the load balancer, IP allocation, and DNS

configuration. When this process is completed, the new node is ready to service requests to your application.

Much of the image deployment can be completed before the node is needed, cutting down on the time it can take to start a new instance and add it to your service.

Each server can contain several VMs. This allows for the optimal use of computing resources and the flexibility to move instances around as needed. As a second or third virtual server is added, it might use the base OS VHD that has already been downloaded (7) or it can download a different base OS VHD based on its needs. This second machine then follows the same process of downloading the application VHD, booting up, and enrolling into the cloud.

All these steps are coordinated by the FC and are usually accomplished in a few minutes.

3.7.4 Image is everything

If the key to the automation of Azure is Hyper-V, then the base VM images and their management are the cornerstone. Images are centrally created, also in an automated fashion, and stored in a library, ready to be deployed by the FC as needed.

A variety of images are managed, allowing for the smallest footprint each role might need. If a role doesn't need IIS, then there's an image that doesn't have IIS installed. This is an effort to shrink the size and runtime footprint of the image, but also to reduce any possible attack surfaces or patching opportunities.

All images are deployed using an Xcopy deployment model. This model keeps deployment simple. If the FC relied on complex scripts and tools, then it would never truly know what the state of each server would be and it would take a lot longer to deploy an instance. Again, diversity is the devil in this environment.

This same approach is used when deploying patches. When the OS needs to be patched, Microsoft doesn't download and execute the patch locally as you might on your workstation at home. Doing so would lead to too much risk, having irregular results on some of the machines. Instead, the patch is applied to an image and the new image is stored in the library. The FC then plans a rollout of the upgrade, taking into account the layout of the cloud and the update domains defined by the various service models that are being managed.

The updated image is copied in the background to all of the servers used by the service. After the files have been staged to the local disk, which can take some time, each update domain group is restarted in turn. In this way, the FC knows exactly what's on the server. The new image is merely wired up to the existing service bits that have already been copied locally. The old image is kept locally for a period of time as an escape hatch in case something goes wrong with the new image. If that happens, the server is reconfigured to use the old image and rebooted, according to the update domains in the service model. This process dramatically reduces the service window of the servers, increasing uptime and reducing the cost of maintenance on the cloud.

We've covered how images are used to manage the environment. Now we're going to explain what you can see when you look inside a running role instance.

3.8 The innards of the web role VM

Your first experience with roles in Azure is likely to be with the web role. To help you develop your web applications more effectively, it's worth looking in more detail at the VM that your web role is hosted in. In this section, we'll look at the following items:

- The details of the VM
- The hosting process of your web role (`WaWebHost`)
- The `RDAgent` process

3.8.1 Exploring the VM details

You can use the power of native code execution to see some of the juicy details about the VM that your web role runs on. Figure 3.12 shows an ASP.NET web page that shows some of the internal details, including the machine name, domain name, and the user name that the code is running under.

Figure 3.12 Using native code, you can see some of the machine details of a web role in Windows Azure. In this example, Microsoft is using Windows Server 2008 Enterprise x64. Notice that the user name that the process is running as is a GUID.

If you want, you can easily generate the web page shown in figure 3.12 by creating a simple ASPX page with some labels that represent the text, as follows:

```
<div>
   ProcessorCount:
            <asp:Label ID="lblProcessorCount" runat="server" />
 </div>
```

Finally, you can display the internal details of the VM using the code-behind in the following listing.

Listing 3.1 Using code behind to display machine details

```
using System.Management;

public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

            // Initialize
            var computer = new Microsoft.VisualBasic.Devices.Computer();
            lblMachineName.Text = computer.Name;
```

Class fetches information about server

```
        // OS Details
        lblOSName.Text = computer.Info.OSFullName;
        lblOSVersion.Text = computer.Info.OSVersion;
        lblMachineName.Text = computer.Name;

        // Computer System Details
        lblProcessorCount.Text =
        ➥   System.Environment.ProcessorCount.ToString();
        lblCLRVersion.Text = System.Environment.Version.ToString();
        lblCurrentDirectory.Text = GetCurrentDirectory();
        lblTimeSinceLastRestart.Text = GetTimeSinceLastRestart();
        lblDomainName.Text = System.Environment.UserDomainName;
        lblUserName.Text = System.Environment.UserName;
        lblCPUName.Text = GetCPUName();
        lblArchitecture.Text = GetArchitecture();
    }

    private string GetCurrentDirectory()
    {
        try
        {
            return System.Environment.CurrentDirectory;
        }
        catch
        {
            return "unavailable";
        }
    }

    private string GetTimeSinceLastRestart()
    {
        try
        {
            TimeSpan time = new TimeSpan(0, 0, 0, 0,
            ➥   System.Environment.TickCount);
            return time.ToString();
        }
        catch
        {
            return "unavailable";
        }
    }

    private string GetCPUName()
    {
        try
        {
            using (ManagementObject Mo = new
            ➥   ManagementObject("Win32_Processor.DeviceID='CPU0'"))
            {
                return (string)(Mo["Name"]);
            }
        }
        catch
        {
            return "unavailable";
```

Gets length of time server has been running (lblTimeSinceLastRestart)

Gets domain name server is running on (lblDomainName)

Gets user name service is running as (lblUserName)

```
            }
        }

        private string GetArchitecture()
        {
            try
            {
                using (ManagementObject Mo = new
                ➥ ManagementObject("Win32_Processor.DeviceID='CPU0'"))
                {
                    ushort result = (ushort)(Mo["Architecture"]);
                    switch (result)
                    {
                        case 0:
                            return "x86";
                        case 9:
                            return "x64";
                        default:
                            return "other";
                    }
                }
            }
            catch
            {
                return "unavailable";
            }

        }
    }
```

You can now, of course, deploy your web page to Windows Azure and see the inner details of your web role, which were shown in figure 3.12. These machine details provide you with some interesting facts:

- Web roles run on Windows 2008 Enterprise Edition x64
- They run quad core AMD processors and one core is assigned
- The domain name of the web role is CIS
- This VM has been running for an hour
- The Windows directory lives on the D:\ drive
- The web application lives on the E:\ drive

This is just the beginning; feel free to experiment and discover whatever information you need to satisfy your curiosity about the internals of Windows Azure by using calls similar to those shown in listing 3.1.

3.8.2 The process list

Now that we're rummaging around the VM, it might be worth having a look at what processes are actually running on the VM. To do that, you'll build an ASP.NET web page that'll return all the processes in a pretty little grid, as shown in figure 3.13.

To generate the list shown in figure 3.13, create a new web page in your web role with a `GridView` component called `processGridView`:

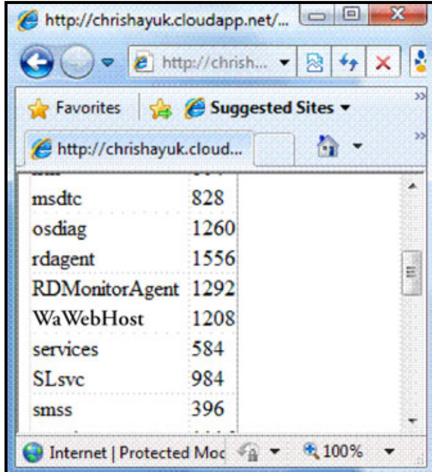


Figure 3.13 The process list of a Windows Azure VM. The RDAgent process is related to Red Dog, which was the code name for Azure while it was being developed.

```
<asp:GridView ID="processGridView" runat="server"/>
```

Next, add a using `System.Diagnostics` statement at the top of the code-behind and then add the following code to the `Page_Load` event:

```
var processes = Process.GetProcesses();

processGridView.DataSource = from process in processes
                             orderby process.ProcessName
                             select new
                             {
                                Name = process.ProcessName,
                                Id = process.Id.ToString() };
                             }
processGridView.DataBind();
```

Gets list of running processes

Uses LINQ query to streamline data returned

Binds query result to grid for screen output

This code will list all the processes on a server and bind the returned list to a `GridView` on a web page, as displayed in figure 3.13. If you look at the process list displayed in figure 3.13, you'll see the two Windows Azure–specific services that we're interested in: `WaWebHost` and `RDAgent`.

We'll now spend the next couple of subtopics looking at these processes in more detail.

3.8.3 *The hosting process of your website (WaWebHost)*

If you were to look at the process list for a live web role (shown in figure 3.13), or if you were to fire up your web application in Windows Azure and click the Process tab, you would notice that the typical IIS worker process (w3wp.exe) isn't present when your web server is running.

You would also notice that if you stop your IIS server by issuing `IISReset - stop`, your web server continues to run. You know from installing the Windows Azure SDK

that web roles are run under IIS 7.0. So, why can't you see your roles in IIS, or restart the server using `IISReset?`

HOSTABLE WEB CORE

Although Windows Azure uses IIS 7.0, it makes use of a new feature, called *hostable web core,* which allows you to host the IIS runtime in-process. In the case of Windows Azure, the `WaWebHost` process hosts the IIS 7.0 runtime. If you were to look at the process list on the live server or on the development fabric, you would see that as you interact with the web server, the utilization of this process changes.

WHY IS AZURE RUN IN-PROCESS RATHER THAN USING PLAIN OLD IIS?

The implementation of the web role is quite different from that of a normal web server. Rather than using a system administrator to manage the running of the web servers, the data center overlord—the FC—performs that task. The FC needs the ability to interact and report on the web roles in a consistent manner. Instead of attempting to use the Windows Management Instrumentation (WMI) routines of IIS, the Windows Azure team opted for a custom Windows Communication Foundation (WCF) approach.

This custom in-process approach also allows your application instances to interact with the `WaWebHost` processing using a custom API via the `RoleEnvironment` class. You can read more about the `RoleEnvironment` class in chapter 4.

3.8.4 The health of your web role (RDAgent)

The `RDAgent` process collects the health status of the role and the following management information on the VM:

- Server time
- Machine name
- Disk capacity
- OS version
- Memory
- Performance statistics (CPU usage, disk usage)

The role instance and the `RDAgent` process use named pipes to communicate with each other. If the instance needs to notify the FC of its current state of health, notification is communicated from the web role to the `RDAgent` process using the named pipe.

All the information collected by the `RDAgent` process is ultimately made available to the FC; it determines how to best run the data center. The FC uses the `RDAgent` process as a proxy between itself, the VM, and the instance. If the FC decides to shut down an instance, it instructs the `RDAgent` process to perform this task.

3.9 Summary

Hopefully, you've learned a little bit about how Azure is architected and how Microsoft runs the cloud OS. You also know how data centers have changed over the generations

of their development. Microsoft has spent billions of dollars and millions of work hours building these data centers and the OS that runs them.

Windows Azure truly is an operating system for the cloud, abstracting away the details of the massive data centers, servers, networks, and other gear so you can simply focus on your application. The FC controls what's happening in the cloud and acts as the kernel in the operating system. With the power of the FC and the massive data centers, you can define the structure of your system and dynamically scale it up or down as needed. The infrastructure makes it easy to do rolling upgrades across your infrastructure, leading to minimal downtime of your service.

The service model that you define consists of the service definition and service configuration files and describes, to the FC, how your application should be deployed and managed. This model is the magic behind the data center automation. New configuration settings are held in the ServiceDefinition.csdef and ServiceConfiguration.cscfg files. Centralizing all your configuration file-reading code into one neat, handy `ConfigurationManager` class is a real time saver.

Fault and update domains describe how the group of servers running your application should be separated to protect against failures and outages. Fault domains ensure that your service is not tied to a single point of failure, which could be catastrophic to your service. An update domain provides the ability to perform a rolling upgrade, keeping you from having to take down your whole service to do an upgrade.

When you need to upgrade your application, you can perform either a static upgrade or a rolling upgrade, which you can do via the Azure portal. All you do is choose a few options and click a button, or you can use the service management API.

The automated nature of Azure is thanks to Hyper-V, Microsoft's enterprise virtualization solution. Hyper-V consolidates work to as few cores as possible by monitoring the use of each core and CPU, all while maintaining a small footprint.

In the next few chapters, we'll work much more closely with the service runtime. We'll look at how you know when you're running in the fabric, and the configuration magic of the service model.

4 It's time to run with the service

This chapter covers

- Interacting with Windows Azure via the `ServiceRuntime` assembly
- Defining your Windows Azure role
- Configuring your Windows Azure role

In the last chapter we got into the guts of the infrastructure and architecture of Windows Azure. During that chapter we introduced the concept of the service model and how it's used by the FC to manage your role.

In this chapter we'll take some time out to look at the parts of the service model that we didn't get to mess around with much (specifically the service definition and service configuration files). But first, let's spend a little time with the Service Management API.

4.1 Using the Windows Azure Service Management API

In both chapter 1 and chapter 2, you created a brand new Windows Azure web role from scratch. As we pointed out then, a web role hosted by Windows Azure is

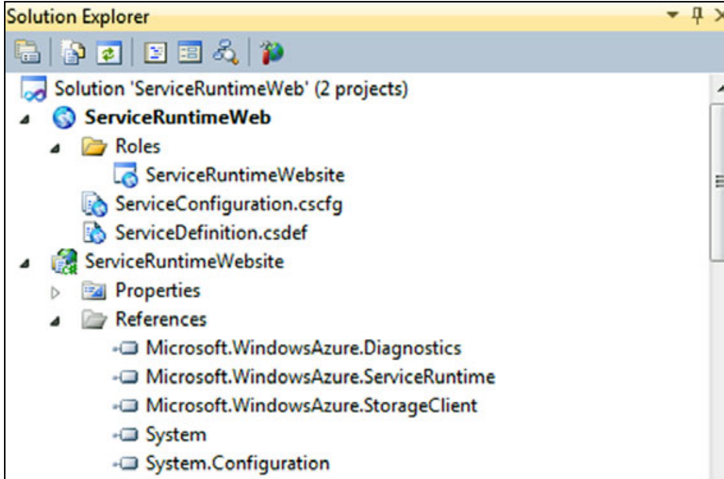


Figure 4.1 Three assemblies to play with in your new web role

a regular old ASP.NET web application with a little bit of extra magic that allows you to interact with Windows Azure. That extra magic is three new assemblies that are automatically added to your new web application when you create a new Windows Azure Cloud Service project in Visual Studio. Figure 4.1 shows these assemblies listed in the Visual Studio 2010 Solution Explorer.

In this chapter we'll focus only on the `Microsoft.WindowsAzure.ServiceRuntime` assembly. We'll look at `Diagnostics` in chapter 18, and the `StorageClient` in chapters 9 through 12 and 16.

The `ServiceRuntime` assembly acts as a bridge between the Windows Azure runtime and your application. Although you don't need to include the `ServiceRuntime` assembly in your web role, you should. Without this assembly, your applications have no way to interact with Windows Azure and make use of the APIs that it exposes to you. This assembly provides the following helper routines that we'll explore throughout this chapter:

- Checking whether your application is running in the cloud
- Retrieving configuration settings
- Getting a reference to a file held in the local cache

This assembly provides some valuable methods that you'll need to fully use the power of Windows Azure. We'll take a look at how you can include this assembly in your projects.

4.1.1 Adding the ServiceRuntime assembly to your application

As you saw earlier, when you create a new ASP.NET web role in Visual Studio, the new assemblies from the SDK are automatically referenced within your project. That's great if you're creating a new role, but what if you're migrating an existing ASP.NET application to the cloud, or if you want to access the API from another assembly?

The ASP.NET web role project created in Visual Studio is a normal ASP.NET web application with three extra assembly references. If you need to migrate your existing application (or use it in another project), you can always add those extra assemblies via the Add Reference dialog box. You can find these assemblies in the c:\Program Files\Windows Azure SDK\v1.1\ref\ directory.

Now that you know how to reference the `ServiceRuntime` assembly, let's take a look at some of the API calls and how you can use them.

4.1.2 Is your application running in Windows Azure?

One of the static properties that the `ServiceRuntime` exposes via the `RoleEnvironment` static class is the ability to check whether your application is running in Windows Azure. You can perform this check using the `RoleEnvironment.IsAvailable` property.

To see this check in action, you're going to quickly create a web page that displays a label that states whether your application is running in Windows Azure. Figure 4.2 shows that web page running in the development fabric.

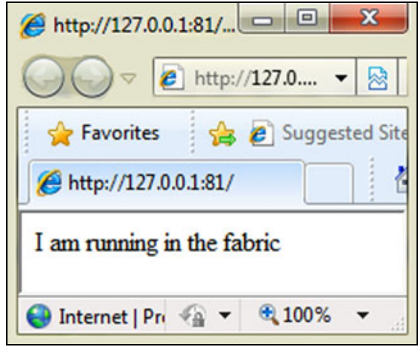


Figure 4.2 A funky little web page that tells you that you're running in the fabric

To create the web page shown in figure 4.2, create a new cloud service solution called `ServiceRuntimeWeb` with a new ASP.NET web role called `ServiceRuntimeWebsite`. In the web project, modify the Default.aspx page to include the following label:

```
<asp:label id="runningInTheFabricLabel" runat="server"/>
```

After you've added this label to your web page and a `using` statement for the `ServiceRuntime` namespace, you can then display the result of the `RoleEnvironment.IsAvailable` call in the contents of the label by adding the following code to the `Page_Load` event:

```
runningInTheFabricLabel.Text = RoleEnvironment.IsAvailable ?
                 "I am running in the fabric" : "Not in the fabric";
```

As you know, web roles are standard ASP.NET web applications; they can still be run on a standard IIS web server (if you like retro computing). If you launched this page in IIS (or the ASP.NET Web Development Server by selecting the ASP.NET project and pressing F5), then it would display the message `Not in the fabric`. If you were to now fire up your web page in the Windows Azure development fabric, it would display `I am running in the fabric` (as shown in figure 4.2).

The `RoleEnvironment.IsAvailable` call is not only useful for announcing to the world that your web application is in heaven (I mean in the cloud), but it's also useful for building applications (or libraries) that will be hosted both inside and outside Windows Azure. Because most of Windows Azure's APIs aren't available outside Windows

Azure, you might want to check that you're running inside Windows Azure before using one of these APIs. Later in this section we'll look at some of these situations and possible solutions that you can use when working with shared code.

We've introduced you to the `ServiceRuntime` assembly. Now let's take a look at some of the other differences between a standard ASP.NET web application and a Windows Azure ASP.NET web role.

4.2 Defining your service

In chapter 3, we spent quite a bit of time discussing the infrastructure and architecture of Windows Azure. In that chapter, we introduced the concept of the service model and how it comprises three elements: the service definition, the service configuration, and the operating model.

In this section, we'll look at the first piece of the service model puzzle, which is the *service definition file* (we'll look at some of the other stuff later on). In chapter 3, we described how the FC uses the service definition file (ServiceDefinition.csdef) to manage your service; in this chapter we'll show you how you can effectively define your service.

The following information is held in the service definition file:

- The number of required upgrade domains (see chapter 3)
- The endpoint of your service (port and protocol)
- Whether the role runs in partial or full trust
- Whether the role has any configuration settings
- The amount of local disk space that the role requires for local file storage
- The required size of the VM

In the following subsections, we'll take a look at how you can define some of that information. Before we do that, let's return to the service definition file itself.

4.2.1 The format of the service definition file

When you created your Cloud Service project in chapter 1, the service definition file was automatically added to your project. The following listing shows the service definition file for the `ServiceRuntimeWeb` project that you created in chapter 1.

Listing 4.1 Service definition file of the `ServiceRuntimeWeb` project

```
<ServiceDefinition name="ServiceRuntimeWeb"                      <-- Name of your
xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/          cloud project
    ServiceDefinition">
  <WebRole name="ServiceRuntimeWebsite">
    <InputEndpoints>
      <InputEndpoint name="HttpIn" protocol="http" port="80" />   <-- 1 HTTP port and protocol
    </InputEndpoints>                                                 your application runs on
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString" />   <-- 2 Any special configuration
    </ConfigurationSettings>                               elements you want
  </WebRole>
</ServiceDefinition>
```

As shown in listing 4.1, the service definition file adheres to the following format:

- Cloud project (ServiceDefinition element)
  - Role definition (web role)
    - Input endpoints (1)
    - Internal endpoints (not shown in listing 4.1)
    - Configuration settings (2)
    - Certificates (not shown in listing 4.1)
    - Local storage (not shown in listing 4.1)

Throughout the course of the next few sections we'll explore the items that define your role in more detail.

Because your Cloud Service project contains only a single web role, you'll see only a single role definition in your service definition file. If your project contained multiple roles, these roles would also be included in the file. The XML in the following listing shows how this would be structured in the service definition file.

Listing 4.2 Configuring multiple roles in the service definition file

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="ServiceRuntimeWeb" xmlns="http://
     schemas.microsoft.com/ServiceHosting/
2008/10/ServiceDefinition">
  <WebRole name="ServiceRuntimeWebsite">

  </WebRole>
  <WorkerRole name="WorkerRole1">

  </WorkerRole>
</ServiceDefinition>
```

In this example, there are two roles in the project: a web role called ServiceRuntime-Website and a worker role called WorkerRole1 (note that the definition of a worker role is exactly the same as the configuration of a web role, except that the definition element is called WorkerRole instead of WebRole).

You have an idea of the structure of the service definition file, but what sort of information describing your service would you put in that file?

4.2.2 Configuring the endpoint of your web role

If you look at the service definition file for your web role in listing 4.1, you'll see that the HTTP port and protocol that your application runs on is defined at (1).

Windows Azure allows web roles to receive incoming HTTP or HTTPS messages (usually via port 80 and port 443 respectively) via your virtual IP address only (see chapter 3). Any other traffic that's sent to your virtual IP address is either filtered out by the firewalls or is not forwarded to your web role from the load balancer. Figure 4.3 shows the interaction between a client browser, the load balancer, and your web role. Worker roles are not held to this protocol restriction. (We'll cover worker roles in chapter 15.)

By locking down the available protocols, Windows Azure reduces the surface area of attack for your web role. Any incoming requests to your web role (outside of the port and protocol combinations defined in the service definition) are filtered out by the firewalls and load balancers; the request never reaches the servers that your web role is hosted on. This level of protection protects your web role from port attacks, as well as from distributed denial-of-service (DDoS) attacks.

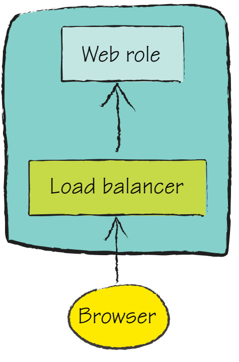


Figure 4.3 The load balancer protects the galaxy (or at least your web role) from the threat of invasion.

By default, Visual Studio correctly configures your web role to use HTTP and port 80 in your service definition file. (This configuration is shown at ❶ in listing 4.1.)

```
<InputEndpoint name="HttpIn" protocol="http" port="80" />
```

If you wanted to expose your service via HTTPS, you would change the `InputEndpoint` in the service definition file to the following:

```
<InputEndpoint name="HttpsIn" protocol="https" port="443" />
```

If you need to run your application in both HTTP and HTTPS, define two `InputEndpoint` tags:

```
<InputEndpoints>
   <InputEndpoint name="HttpIn" protocol="http" port="80"/>
   <InputEndpoint name="HttpsIn" protocol="https" port="443"/>
</InputEndpoints>
```

You can configure other ports for the protocols as well (for example, you can define an endpoint with the port 8080). You would usually configure other ports when there are multiple web roles in the same solution. Such a configuration would allow each web role to be accessed on a different port.

WHERE'S THE GUI?

If you're currently thinking to yourself "I'll never remember all that XML syntax," then good news: the service definition file has an XSD (XML Schema Definition language) associated with it. You get full IntelliSense support when you edit this file in Visual Studio. If you edit in Notepad, you don't get the benefit of this support.

Alternatively, if you feel that we've moved beyond text files and are in a Windows Presentation Foundation (WPF) Minority Report-style GUI interface era, then you'll be pleased to hear that you can edit the service definition file by using a dialog box in Visual Studio 2010. To open the dialog box, double-click the name of your web role in the Roles folder of your Cloud Service project (for example, double-click `ServiceRuntimeWebsite` in the Roles folder in the `ServiceRuntimeWeb` cloud project, as shown in figure 4.1).

To modify the endpoints in the editor, select the Endpoints tab, shown in figure 4.4.

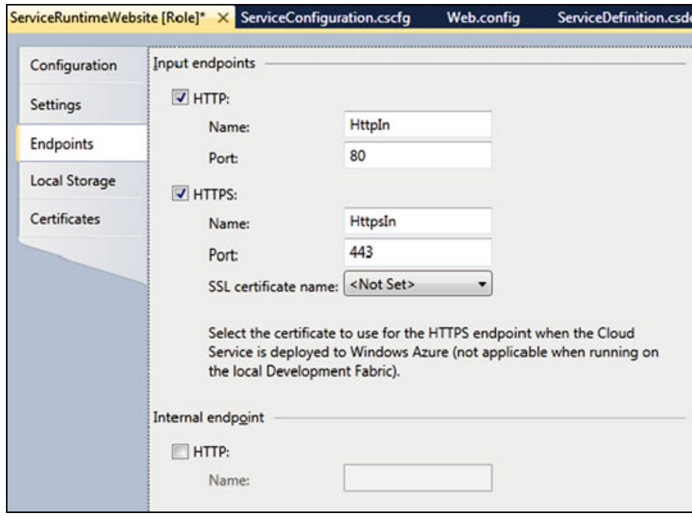


Figure 4.4 If you don't want to use pure XML to configure your endpoints, you can use this GUI by double-clicking the name of your web role in the Roles folder of your Cloud Service project.

There are several reasons why you might want to use a port other than port 80 or 443, but for the most part, these are the traditional ports used with HTTP and HTTPS and are considered best practices.

IGNORING BEST PRACTICES WHEN DEVELOPING

Although security best practices are great for production servers, they can be a real pain to follow during development. We can all thank Ray Ozzie at Microsoft for making this a little easier in the development fabric by allowing us to run our web roles on any port.

Figure 4.4 shows how you can easily set this value via the editor. Because we have an XML fetish, let's look at how it's done in the service definition file. The following bit of configuration shows how you can run your web role on port 87 over HTTP in the development fabric:

```
<InputEndpoint name="HttpIn" protocol="http" port="87" />
```

The development fabric is pretty lenient when it comes to configuring available ports. If a port is already taken (if IIS is hogging port 80), it gives you the next available port. For example, if you asked for port 80, it would fire up your application on port 81 instead.

INTRAROLE COMMUNICATION

You might have noticed in figure 4.4 that there's a little section called Internal Endpoint (go on, look, if you didn't see it already). If you need to host a web role internally (you don't want that web role to be available outside the Windows Azure fabric) but you want to make that web role available to another role, this check box is for you. A typical reason for wanting this functionality is that a web role or worker role is reliant on an internal web service (as is the case with service-oriented-architecture-type applications).

You can configure the internal endpoint of this role by setting the name, port, and protocol of the internal endpoint in this way:

```
<InternalEndpoint name="MyInternalRole" protocol="http"/>
```

Alternatively, you can use the editor, as shown in figure 4.4.

It's worth pointing out that web roles support only HTTP internal endpoints and can't be secured with certificates. You should use the HTTP protocol only for internal endpoints that are legacy web services (old ASP.NET Web Services [ASMX] and the like). If you're considering using this approach for WCF services, you should host your service with a worker role and expose it via TCP instead. TCP generally provides better performance for internal services than does HTTP.

WORKER ROLE ENDPOINTS

Although the service definition file is a common file that's used by both web roles and worker roles, we're going to look at only web roles in this chapter. We'll leave the examples on how to host worker roles that accept incoming requests across various protocols and ports to later in the book.

The configuration of a worker role uses the same `InputEndpoint` tag as the web role. The following XML shows the endpoint for a worker role hosted on TCP port 10000:

```
<InputEndpoint name="MyEndpoint" protocol="tcp" port="10000" />
```

Figure 4.5 shows the GUI for configuring a worker role endpoint.

You can use the editor shown in figure 4.5 to set internal endpoints. Remember, internal endpoints are dynamically assigned ports and you can't manually set the port number. The following XML defines an internal endpoint called `MyEndpoint` that uses TCP (rather than HTTP or HTTPS) as a protocol:

```
<InternalEndpoint name="MyEndpoint" protocol="tcp" />
```

As we said earlier, we intended to give only a brief description about how this configuration applies to worker roles. We'll return to this later on, but now we're going to take a little detour and examine other things you can do in the editor.

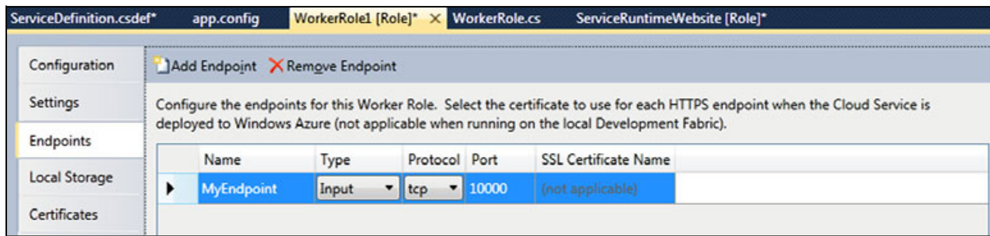


Figure 4.5 Setting a worker role to be hosted via TCP on port 10000 to the external world using the Visual Studio GUI

4.2.3 Configuring trust level, instances, and startup action

When you click the Configuration tab, you see the options shown in figure 4.6.

There are three sections on this page: .NET Trust Level, Instances, and Startup Action. We'll be looking at these sections in more detail later in the book, but for the moment let's have a quick introduction to them.

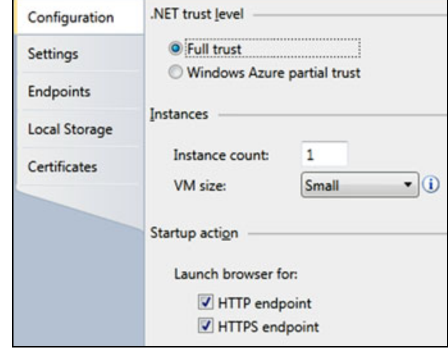


Figure 4.6 The Configuration tab for your role in Visual Studio

.NET TRUST LEVEL

Windows Azure supports two levels of trust: full trust and partial trust. Partial trust is similar to ASP.NET's medium trust level. It restricts the operations that your application can perform to only those that it trusts. Whenever possible, you should run your application in partial-trust mode because it provides a greater level of security (I sleep like a baby at night whenever my application is running under partial trust). If, however, you need to perform big scary actions that would make a security freak's skin crawl (such as C++, Reflection, or P/Invoke) then you'll need to set your application to full trust.

Using the dialog box shown in figure 4.6 is probably the easiest way to set your trust level, but because we all love messing with configuration files, let's modify the service definition file directly.

To configure partial trust, set the `enableNativeExecution` attribute on your role to `false`. For full trust, you can either set the attribute to `true` or not configure it at all (full trust is the default level). The following XML shows how to set your earlier web role to partial trust:

```
<WebRole name="ServiceRuntimeWebsite"
           enableNativeExecution="false">
```

In chapter 6, we'll look in more detail at the supported trust levels. Now let's move on to the Instances section.

INSTANCES

The Instances section allows you to set the number of instances of your role and the size of your VM (small, medium, large, or extra large). The number of instances is an important setting, but because this setting is held in the service configuration file, we're going to save our discussion of it for chapter 6. Now it's on to the Startup Action section.

STARTUP ACTION

The final section on the Configuration page is Startup Action. This section isn't really part of the service definition, but is instead a wee bit of Visual Studio configuration. The following two check boxes are in this section:

- Launch browser for HTTP endpoint
- Launch browser for HTTPS endpoint

These check boxes tell Visual Studio which endpoints you want to launch in the development environment when you press F5.

That's about it for what you can do on the Configuration page in the Visual Studio editor. Let's take a look now at the Local Storage page and find out how to configure your local storage.

4.2.4 Configuring local storage

Local storage is a temporary filesystem storage area that's made available to a role instance to store and retrieve data locally. The local storage area is available only to a single role instance and can't be shared across multiple role instances.

If the VM for your role instance dies and never recovers, you'll lose the data stored in this area forever. Only volatile data should be stored in this storage area; never store any data that you might need to rely on later in a court of law. Any data that you store in local storage should also be stored in a nonvolatile storage area such as a BLOB, Table storage, or SQL Azure.

IF IT'S SO VOLATILE, WHY DO I NEED IT?

Although BLOBs, Table storage, and SQL Azure can be accessed by all instances of a role, the convenience of centralized storage mechanisms comes at a cost: latency. Those other, nonvolatile storage areas are all hosted on separate servers in another part of the data center, whereas local storage is part of your VM. Because the disks are hosted on the same server as your VM, local storage is much faster than the other storage mechanisms.

BLOBs, Table storage, and SQL Azure are suitable for most scenarios, but if you're processing high volumes of data, you might want to use local storage to temporarily cache that data. Let's look at how to set that up.

SETTING UP LOCAL STORAGE

As always, there are two ways to configure local storage. You can either do it manually via the service definition file, or you can use the role editor in Visual Studio 2010 and let it modify the service definition file for you. Figure 4.7 shows the GUI for configuring local storage in Visual Studio.

When you define a local storage resource, you can define the following three items:

- The name of the resource
- The amount of space required, in megabytes
- Whether you want the temporary data deleted when the role is recycled

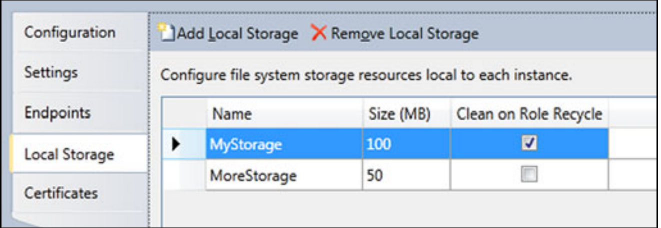


Figure 4.7 Configuring some temporary local storage space in your role using Visual Studio 2010

The maximum amount of local storage that you can use for a single role instance is 20 GB. If you need more than 20 GB of temporary storage space, you might want to rethink the architecture of your application.

The information that you supply in the editor is reflected in the service definition file, as shown in the following listing.

Listing 4.3 Configuring local storage in the service definition file

```
<WebRole name="ServiceRuntimeWebsite">
   <LocalResources>
      <LocalStorage name="MyStorage"
                    cleanOnRoleRecycle="true"
                    sizeInMB="100" />
      <LocalStorage name="MoreStorage"
                    cleanOnRoleRecycle="false"
                    sizeInMB="50" />
   </LocalResources>
</WebRole>
```

In the current version of Windows Azure, you can't dynamically change this setting at runtime, which is why this code lives in the service definition file and not in the service configuration file. If you incorrectly size your temporary storage area and need to request a larger size, you have to redeploy your application. Why is that?

The FC uses the requested size of local storage as part of its algorithm to decide which physical servers will host your VM. If the FC allowed you to modify the local storage dynamically without a redeployment, it might not be able to satisfy your request with the servers that are currently hosting your role instances (there might be too many other roles hosted on that server that have high local storage requirements). By forcing a redeployment of the application, the FC can safely redeploy your role instances to the servers that most appropriately satisfy your request.

Now that you know how to set up local storage, let's take a quick look at how to use it in your application.

USING LOCAL STORAGE

If you need to be able to use local storage in your role (web or worker), then you can make a request to retrieve information about your role using the `RoleEnvironment` class in the `ServiceRuntime`. Use the following call to request information about your local storage resource.

```
LocalResource myStorage =
                   RoleEnvironment.GetLocalResource("myStorage");
```

You need to call `RoleEnvironment.GetLocalResource`, passing in the name of the local storage resource that you defined earlier (`myStorage`). The object returned by `GetLocalResource` exposes three properties (`Name`, `MaximumSizeInMegabytes`, and `RootPath`). The `Name` and `MaximumSizeInMegabytes` properties return the information that you set in the service definition file:

```
string name = myStorage.Name;
int maxSizeInMB = myStorage.MaximumSizeInMegabytes;
```

The RootPath property returns the physical path of the folder where your temporary storage area has been assigned. Using the RootPath property, you can use standard .NET methods to store and retrieve data in this folder. The following code creates a text file called HelloWorld.txt that contains the text "Goodbye World".

```
System.IO.File.WriteAllText(myStorage.RootPath + "HelloWorld.txt",
                            "Goodbye World");
```

It's pretty simple to use local storage. It's built on all the existing system.I/O classes that we all know and love.

Before we leave the subject of local storage, we want to cover one final thing: the Clean on Role Recycle setting.

RECYCLING A ROLE

Use the Clean on Role Recycle setting to indicate whether you want to lose or keep your local storage data if one of the following things occurs:

- An upgrade (you deploy a new version or an OS patch is applied)
- A fault (the server dies)
- You request that the role be recycled

It's pretty hard to test how your application responds to losing your temporary data as part of an upgrade or a fault, but you can manually request that your roles be recycled. All you need to do is call the RequestRecycle method in the RoleEnvironment class:

```
RoleEnvironment.RequestRecycle();
```

This call not only allows you to test that your application handles local storage correctly when your role instance is recycled; it also allows you to test whether the rest of your application behaves gracefully.

If your application needs to know when a role instance is stopping (because it's cleaning up resources, notifying a monitor, or performing some other such task), you can always use the RoleEnvironment.Stopping event:

```
public class WebRole : RoleEntryPoint
{
   public override bool OnStart()
  {
            DiagnosticMonitor.Start("DiagnosticsConnectionString");

            RoleEnvironment.Stopping += new
               EventHandler<RoleEnvironmentStoppingEventArgs>
(RoleEnvironment_Stopping);

            return base.OnStart();
   }

   void RoleEnvironment_Stopping(object sender,
     RoleEnvironmentStoppingEventArgs e)
   {
      Trace.WriteLine("Stopping");
   }
}
```

You can easily stick any cleanup code that you need in the event handler for this event. When this handler is called, your code has only 30 seconds to respond. This time limit protects Windows Azure from sloppy tear-down code or freezes in the cleanup process. This limit is similar to the limit local Windows services face when they're told to stop by the user or the OS.

You now know almost everything you need to know about local storage. We'll revisit this topic in part 4, when we show how you can use this in combination with BLOBs and how local storage can help you when you're building massively scalable worker processes built on MapReduce.

Now let's turn our attention to the tabs in the role editor that we haven't covered yet. So far we've looked at the Configuration, Endpoints, and Local Storage tabs. That leaves the Certificates and Settings tabs. When we looked at configuring endpoints, we discussed HTTPS but we didn't mention how to configure the SSL certificate for your site. Let's return to that subject and look at certificate management in Windows Azure.

4.3 Setting up certificates in Windows Azure

We want to look at how to generate, add, and configure certificates in Azure. Let's look at how to generate one first; then we'll cover how to add and configure them. Certificates are widely used to encrypt, and thereby protect, data as it travels over the network. In this section, when we refer to certificates, we mean the type you'll use for HTTPS/SSL or for your own encryption needs. We're not referring to the management certificates we'll cover in chapter 18.

4.3.1 Generating a certificate

For live production applications you should use a purchased certificate from a trusted authority. If you're just experimenting or testing an application, you can use a test certificate. Because you've already bought this lovely book, we'll save your wallet from more troubles and show you how to generate a test certificate that you can use on the production or development fabric.

To generate a test X.509 certificate, you can use a tool called `makecert`, which is included with both Visual Studio and the .NET Framework. To start using the tool, fire up an instance of the Visual Studio command prompt as an administrator. Using the command prompt, you can generate a test certificate with the following command:

```
makecert -r -pe -a sha1
-n "CN=Windows Azure Authentication Certificate"
-ss My -len 2048
-sp "Microsoft Enhanced RSA and AES Cryptographic Provider"
-sy 24 MyCertificate.cer
```

This command generates a test X.509 certificate called MyCertficate.cer, and stores it in the CurrentUser/Personal store. You'll need to use the certificate management tool in Windows to export it as a PFX-formatted certificate, which is suitable for use in Windows Azure. For more details about the `makecert` tool, you can always visit the following URL: http://msdn.microsoft.com/en-us/library/bfsktky3(VS.80).aspx.

Figure 4.8 To install a certificate using the Windows Azure portal, select the certificate and click Upload. It's that easy.

With your brand new certificate in hand, you can install the certificate in both the production fabric and the development fabric.

4.3.2 Adding certificates

The production fabric and the development fabric both have different methods of managing certificates. Let's look at how to add certificates on the live system first.

ADDING CERTIFICATES TO THE PRODUCTION FABRIC

As you might expect, you manage certificates via the Azure portal. Select your hosted service in the portal, and then click Manage in the Certificates section. You'll see the window shown in figure 4.8.

If you need to install your certificate using your own code, you can use the management APIs. We won't cover how to do this in this book because it's not a typical scenario for most folks. If you're automatically installing lots of websites, then using your own code could be useful; the rest of you should use the portal.

ADDING CERTIFICATES TO THE DEVELOPMENT FABRIC

If you need to test HTTPS in your development fabric, you'll need the appropriate certificate on your development machine. To upload your certificate into the development fabric, click the Certificates tab in the role editor, as shown in figure 4.9.

You can set the name, location (`LocalMachine`, `CurrentUser`), store name (`My`, `Root`, `CA`, `Trust`), and the certificate.

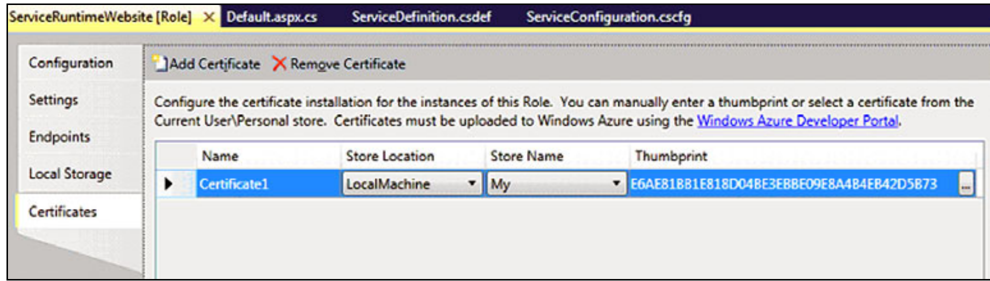


Figure 4.9 Adding certificates to the development fabric

To set the certificate, you can either enter the thumbprint manually or select a certificate from your personal store. If you click the button in the Thumbprint cell, the Select a Certificate dialog box opens (already armed with the test certificate that you generated earlier), as shown in figure 4.10.

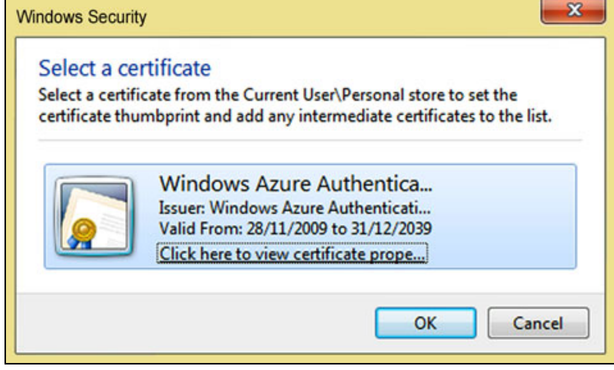


Figure 4.10 The Select a Certificate dialog box shows the certificates you've already generated. This example shows the certificate you generated earlier with `makecert`.

Now that you've selected your certificate, you might be wondering how this is reflected in the service definition file. Don't worry. Your XML-obsessed authors are here to help you out. The following listing shows how the certificate is represented.

Listing 4.4 Adding a certificate to the service definition file

```
<WebRole name="ServiceRuntimeWebsite">
    <InputEndpoints>
      <InputEndpoint name="HttpsIn" protocol="https" port="443" />
    </InputEndpoints>
    <Certificates>
      <Certificate name="MyCertificate"
         storeLocation="LocalMachine"
         storeName="My" />
    </Certificates>
</WebRole>
```

Same information shown in figure 4.9

Everything that you set in figure 4.9 is present in the service definition file, except for the thumbprint, which is stored in the service configuration file. The thumbprint is a configurable setting, not a definable attribute of the service; that's why it's in the configuration file. We're going to talk more about the thumbprint in chapter 5; now let's use that new certificate.

4.3.3 Configuring your HTTPS endpoint to use the certificate

The last thing you need to do is to configure your endpoint to use the new certificate. With your certificate installed, you can either use the SSL Certificate drop-down menu shown in figure 4.4 (which is now populated with your test certificate) to configure the endpoint, or you can manually configure it in the service definition file. To configure the endpoint manually, set the certificate attribute of the `InputEndpoint` element to the name of your certificate:

```
<InputEndpoint name="HttpsIn" protocol="https" port="443"
               certificate="MyCertificate"/>
```

If you expose a worker role externally using an `InputEndpoint` element, you can also secure that service with a certificate. You manage and configure certificates for a worker role in the same way as you do a web role.

4.4 Summary

It's probably a good time to stop and review where we've been. In this chapter, you've taken everything that you've learned about how Windows Azure works (from chapter 3) and started to define how you want your application to run in the environment.

We talked about the `ServiceRuntime` assembly and how you can use that to interact with Windows Azure. This assembly provides several APIs that can help you to get the most out of Windows Azure. You can use the `ServiceRuntime` assembly to determine whether your application is up and running in Azure.

Next, we examined the service definition file, which defines your service. The information in this file instructs the FC about how to manage your application. In this file, you configure the endpoint of your web role, the trust level you want to use, your instances, your local storage, and the certificates you'll use. We showed how you can do all this using either the Visual Studio editor or by putting the information directly into the file in XML.

In the next chapter you'll take what you've learned about defining your service and configure it to work in Windows Azure.

5 Configuring your service

This chapter covers

- Understanding the service configuration file
- Handling configuration at runtime
- Handling non-application settings, based on configuration
- Sharing configuration between Azure and non-Azure applications

In the previous chapter, we concentrated on how you define your role using the service definition file. We'll now look at the second part of the service model picture: the service configuration file.

5.1 Working with the service configuration file

In chapter 4, we described how the service definition file (ServiceDefinition.csdef) describes your role and how it's used by the Fabric Controller (FC) to effectively manage your role. You learned that if you need to change any of the settings in your service definition file, you also need to redeploy your role.

You can change some other settings without redeploying your role. You can even change some of your settings dynamically without restarting the role (surely

not; because we develop on Microsoft products, we love a good restart). These dynamic settings are typically stored in the *service configuration file* (ServiceConfiguration.cscfg).

In the service configuration file, you can dynamically configure standard Windows Azure runtime settings (the number of role instances and the certificate thumbprint) and your own custom settings. Let's see how you configure this information and how you can dynamically modify these settings at runtime.

5.1.1 The format of the service configuration file

Let's now take a look at the standard service configuration file that Visual Studio creates when you create a web role. The following listing shows the service configuration file that was generated when you created your `ServiceRuntimeWeb` project in chapter 1.

Listing 5.1 Service configuration file of the `ServiceRuntimeWeb` project

```
<?xml version="1.0"?>
<ServiceConfiguration serviceName="ServiceRuntimeWeb"       <- Same name as
xmlns="http://schemas.microsoft.com/ServiceHosting/            cloud project
➥ 2008/10/ServiceConfiguration">
  <Role name="ServiceRuntimeWebsite">             <- Same name
    <Instances count="1"/>                           as web role
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString"
      ➥ value="UseDevelopmentStorage=true"/>
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

Before you start thinking, "Oh no, not another XML file," don't worry; you can configure everything you see in this file in the Visual Studio editor. It's useful to understand the structure of this file, because it'll help you understand which settings are dynamic. You might find yourself editing this file in the Azure portal where there isn't a nice GUI, just a plain old text editor. Also, if things go wrong, you might need to recover the file with Notepad.

As shown in listing 5.1, the service configuration file adheres to the following format:

- Cloud project (`ServiceConfiguration` element)
  - Role definition
    - Instances
    - Configuration settings
    - Certificates

Because your Cloud Service project contains only a single web role, only one role definition appears in your service configuration file. The same was true for the service definition file. If your project contained multiple roles, then these would also appear

in the file. The following XML shows how multiple roles would be structured in the service configuration file:

```
<ServiceConfiguration serviceName="ServiceRuntimeWeb" xmlns="http://
     schemas.microsoft.com/ServiceHosting/
     ➥ 2008/10/ServiceConfiguration">
  <Role name="ServiceRuntimeWebsite">
      …
  </Role>
  <Role name="WorkerRole1">
      …
  </Role>
</ServiceConfiguration>
```

The XML shows two roles in this project: a web role called `ServiceRuntimeWebsite` and a worker role called `WorkerRole1` (remember, the configuration of a worker role is exactly the same as the configuration of a web role).

You have a pretty good idea of how the service configuration file is structured; let's look at how you can configure some of the standard settings.

5.1.2 Configuring standard settings

As of the PDC 2009 release, there are only two types of standard settings that you can configure in Windows Azure:

- The number of instances of your role
- The certificate thumbprint

All other settings are custom settings (which we'll look at in the next section).

NUMBER OF INSTANCES

We're now going to briefly look at the most important piece of configuration in Windows Azure: the number of instances that your role has. The following XML defines five instances of your role:

```
<Role name="ServiceRuntimeWebsite">
    <Instances count="5"/>
</Role>
```

If you need to increase the number of instances required for your web role, you can modify this value in your service configuration file. During development, you can also modify this value using the Configuration tab in the Visual Studio GUI.

If you're currently wondering why this setting is in the service configuration file and not in the service definition file, the answer is quite simple. This setting doesn't define the service (the service hasn't changed); you just want more instances of it. By storing this setting in the service configuration file, you can scale the number of services up and down dynamically without having to redeploy your application (after all, no code has changed). Later on in this section we'll look at how you can modify the service configuration file at runtime via the Azure portal.

Now let's return to a topic that we had parked earlier: certificates.

CERTIFICATES

We talked about certificates in chapter 4. To recap, in that section, we showed you how to do the following:

- Generate a test certificate
- Install a certificate into the development fabric and production fabric
- Associate a certificate with your HTTPS endpoint

When you installed your certificate in the development fabric, you saw that it stored the following XML in your service definition:

```
<WebRole name="ServiceRuntimeWebsite">
    <InputEndpoints>
      <InputEndpoint name="HttpsIn" protocol="https" port="443" />
    </InputEndpoints>
    <Certificates>
      <Certificate name="MyCertificate"
         storeLocation="LocalMachine"
         storeName="My" />
    </Certificates>
</WebRole>
```

If we return to the Certificates page in Visual Studio (as shown in figure 4.9), the thumbprint shown there isn't present in the service definition file. That's because this value is stored in the service configuration file, as shown below:

```
<?xml version="1.0"?>
<ServiceConfiguration serviceName="ServiceRuntimeWeb" xmlns="http://
     schemas.microsoft.com/ServiceHosting/
     ➥ 2008/10/ServiceConfiguration">
  <Role name="ServiceRuntimeWebsite">
    <Instances count="1"/>
    <Certificates>
      <Certificate name="Certificate1"
     thumbprint="E6AE81BB1E818D04BE3EBBE09E8A4B4EB42D5B73"
     ➥ thumbprintAlgorithm="sha1" />
    </Certificates>
  </Role>
</ServiceConfiguration>
```

Why is certificate information split across two configuration files (service definition and service configuration)? Think about the intention of this functionality. The definition of your role is as follows: the web role `ServiceRunTimeWebsite` is hosted on port 443 using the protocol HTTPS and it requires a certificate called `MyCertificate`, which you've uploaded.

In reality, certificates aren't a fixed entity. You might want to use a test certificate in the staging environment but use your production certificate only in the production environment. Certificates expire, and you might need to change your certificate over time. For these reasons, the certificate associated with your service is dynamic and configurable, which is why the thumbprint lives in the service configuration file. Do you really want to redeploy your application to change certificates when you switch your application from staging to production?

What's the thumbprint used for? The name of the certificate in the service definition file (`MyCertificate`, the one you typed in Visual Studio) is internal to Windows Azure. This name isn't tied back to the name of the certificate that you generated earlier. You need to be able to retrieve the correct certificate from the store, and the thumbprint is that search parameter. Windows Azure uses the `FindByThumbprint` functionality built into Windows to retrieve the actual certificate.

If you have a production certificate from a trusted authority that you can't install in your development environment and you need to configure that in Windows Azure via the portal, then you can always manually configure the certificate in the service configuration file using the thumbprint.

We've covered the standard configuration settings. Let's look at how you configure your own custom settings.

5.1.3 Configuring runtime settings

As with any other application, when you're working with web or worker roles you need to be able to dynamically configure runtime settings without rebuilding the application from its source. In conventional web applications, the following settings (among others) are typically stored in configuration files:

- Database connection strings
- Service endpoints
- Filesystem paths
- Timeout settings

Although these configuration settings are considered dynamic, in most applications they're rarely changed. The main reason for storing these runtime configuration settings in a configuration file is so that you can easily deploy your application between different environments (development, test, staging, and production). Typically, your production environment talks to different web services, a different database, or a different storage account from your development or staging environment. Using configuration files (rather than rebuilding your source code) to modify these endpoints greatly reduces the complexity, increases the maintainability, and simplifies the deployment process of your application.

In this section we'll look at the following aspects of configuring your runtime settings:

- Configuring runtime settings in conventional web apps
- Defining your runtime configuration settings in Windows Azure
- Initially configuring your runtime configuration settings
- Reading your configuration settings
- Modifying your configuration settings dynamically at runtime

Before we look at how to define your runtime configuration settings in Windows Azure, let's look at how you would do this in conventional ASP.NET web applications.

CONFIGURING RUNTIME SETTINGS IN CONVENTIONAL ASP.NET WEB APPLICATIONS

In conventional ASP.NET web applications, you typically store any configurable runtime settings in the web.config file. Windows Azure provides the ability to read application settings from web.config (as we'll now demonstrate), but this isn't the method that you should use to read the configuration settings.

You're going to build a small web page that'll read and display a setting from the `appSettings` section of web.config. Then you're going to modify this page in future sections of this chapter to use the Windows Azure configuration settings functionality. Figure 5.1 shows the output of the web page that you're going to create. The text "Hello Birds Hello Trees" is read from web.config.

Figure 5.1 Displaying configuration settings in a web page. This text is being read from the `appSettings` section of the web.config file. Because you can't modify this text without redeploying your application, a better place to store it is in the ServiceDefinition.csdef file.

In your ASP.NET web project, add a new ASP.NET web page called ConfigurationSettings.aspx that you'll use to develop the page shown in figure 5.1. Next, add the following label to the page:

```
<asp:label id="mySettingLbl" runat="server"/>
```

The text "Hello Birds Hello Trees" that you'll display in the label `mySettingLbl` is stored in the `appSettings` section of web.config. In the web.config file for your web application, replace the `appSettings` tag with the following:

```
<appSettings>
   <add key="mySetting" value="Hello Birds Hello Trees"/>
</appSettings>
```

Add a `using System.Configuration` line to the top of your code. Then, on the `Page_Load` event of the ConfigurationSettings.aspx page, add the following code to display the contents of the `mySetting` application setting:

```
mySettingLbl.Text = ConfigurationManager.AppSettings["mySetting"];
```

If you were to now run this application in the Web Development Server, IIS, the development fabric, or on the live Windows Azure production fabric, the application would run correctly and display the page shown in figure 5.1.

ISSUES WITH USING WEB.CONFIG

Although Windows Azure can read anything stored in the `appSettings` section of your web.config file, it doesn't provide you with the ability to modify these settings at runtime. If you need to modify a value in your web.config file, then Windows Azure requires you to redeploy your entire application.

In conventional web applications, you would need to modify web.config for each instance if you wanted to change runtime settings. Unfortunately, this approach isn't scalable beyond a single server. If you need to modify 100 instances of a web application, dealing with each individual web.config file is likely to be slow and cause synchronization issues. In such a scenario, you'll need to store the configuration settings centrally and then distribute the changes to each instance. It makes sense to remove the configuration settings from web.config (after all, there's more than just application settings in that file) and provide a new mechanism to feed runtime settings. Let's see how this is done.

SETTING CONFIGURATION IN THE SERVICE DEFINITION

Any configuration settings that your application uses must be defined first in the service definition file (ServiceDefinition.csdef). To display the message "Hello Birds Hello Trees" in your web page, you need to define a new setting in the service definition file. Add the name of the new setting (`mySetting`) to the `ConfigurationSettings` section of the file. The following configuration shows how the `ConfigurationSettings` section should look:

```
<ConfigurationSettings>
   <Setting name="mySetting"/>
   <Setting name="DiagnosticsConnectionString"/>
</ConfigurationSettings>
```

Now the service definition file looks like what's shown in the following listing.

Listing 5.2 Adding a new setting to the service definition file for your project

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="ServiceRuntimeWeb"
     xmlns="http://schemas.microsoft.com/ServiceHosting/
     ➥ 2008/10/ServiceDefinition">
  <WebRole name="ServiceRuntimeWebsite">
    <InputEndpoints>
      <InputEndpoint name="HttpIn" protocol="http" port="80" />
    </InputEndpoints>
    <ConfigurationSettings>
      <Setting name="mySetting"/>
         <Setting name="DiagnosticsConnectionString"
                  value="UseDevelopmentStorage=true" />

    </ConfigurationSettings>
  </WebRole>
</ServiceDefinition>
```

Notice that in the service definition file, you specify only the name of the setting; you don't set the configured value at this point.

SETTING YOUR CONFIGURATION VALUE IN THE SERVICE CONFIGURATION FILE

After you define the configuration settings in the service definition file that'll be used by your application, you need to set the actual value of the setting that'll be used by

the website. The configuration settings are stored in the ServiceConfiguration.cscfg file in your Cloud Service project.

To set your runtime configuration setting, make a copy of the configuration setting in the service definition file and paste it into the service configuration file. Then add a new attribute called `value` that you can set to your default runtime value. The following code shows how your new setting should look:

```
<Setting name="mySetting" value="Hello Birds Hello Trees"/>
```

Now your ServiceConfiguration.cscfg file should look like what's shown in the following listing.

Listing 5.3 Adding a new setting to the service configuration file for your project

```
<?xml version="1.0"?>
<ServiceConfiguration serviceName="ServiceRuntimeWeb"
➥ xmlns="http://schemas.microsoft.com/ServiceHosting/
➥ 2008/10/ServiceConfiguration">
  <Role name="ServiceRuntimeWebsite">
    <Instances count="1"/>
    <ConfigurationSettings>
      <Setting name="mySetting" value="Hello Birds Hello Trees"/>
         <Setting name="DiagnosticsConnectionString"
                  value="UseDevelopmentStorage=true" />

    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

You've set up your ServiceConfiguration.cscfg file and the `ConfigurationSettings` section in the ServiceDefinition.csd file to contain your runtime configuration settings. Now you need to modify your application to use the Windows Azure Service Runtime to read the configuration data.

READING THE CONFIGURATION SETTING

You can access the runtime values of the configuration setting by using the `RoleEnvironment` static class that you used earlier. To retrieve the value of a configuration setting, you can use the following method:

```
RoleEnvironment.GetConfigurationSettingValue(SettingName);
```

To modify your existing web page to read the configuration setting from the Windows Azure configuration settings file rather than from the web.config file, add a using statement for `Microsoft.WindowsAzure.ServiceRuntime` at the top of your code and change the code-behind of your website to the following:

```
    mySettingLbl.Text =
    RoleEnvironment.GetConfigurationSettingValue("mySetting");
```

Now you can run the CloudService project to see the value displayed in your web page, as shown in figure 5.1.

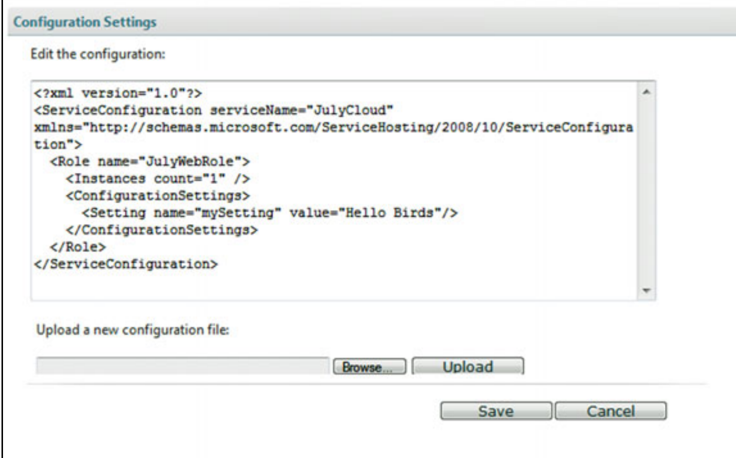


Figure 5.2 Modifying configuration settings in the Azure portal

5.2 Handling configuration at runtime

Congratulations! You understand the service configuration file and how to read those configuration settings. Now let's modify these configuration settings at runtime and find out how to track changes when they occur.

5.2.1 Modifying configuration settings in the Azure portal

You can modify the values that you set in the service configuration file dynamically at runtime by using the Azure portal. First, though, you have to redeploy your application. If you need a little help, skip back to chapter 2 and review that information.

After your application is redeployed and running in the fabric, you can easily modify any of your configuration settings. Select either the production or staging version of the role, and then click Configure. You're redirected to the page shown in figure 5.2, in which you can modify the runtime configuration settings for the role that you selected.

The Azure portal doesn't have any fancy editors; all you see is a big text box with your service configuration file in it. Bet you're glad you paid attention during XML school now.

The Azure portal lets you directly edit the contents of the configuration settings or upload a new version of the service configuration file. After you apply your changes, they're instantly replicated to all instances of your web role; in some cases you don't have to restart any of the roles.

Figure 5.3 shows that you've modified the configuration setting `mySetting` from "Hello Birds Hello Trees" to "Hello Birds".

Figure 5.3 Your web page showing a configuration setting that you modified via the Azure portal

Wow, modifying your configuration settings is easily achieved through the Azure portal. What's even cooler is that your role can pick up and run with the changes without needing a restart. The only issue is that you might need to store a local copy of the previous setting whenever that setting changes. Let's see how to do that.

5.2.2 Tracking service configuration changes

When you change a configuration setting, Windows Azure raises two events that your application can handle:

- `RoleEnvironment.Changing`—Fires before the changes to the setting are applied. You can still read the existing setting before the change is applied.
- `RoleEnvironment.Changed`—Fires after the changes to the setting are applied. You might use this event to set up a new shared resource after a cleanup of a shared resource.

To hook up and define one of these events, you can use the code shown in the following listing.

Listing 5.4 Tracking changes to the service configuration

```
public class WebRole : RoleEntryPoint
 {
       public override bool OnStart()
       {
           DiagnosticMonitor.Start("DiagnosticsConnectionString");

           RoleEnvironment.Changing += RoleEnvironmentChanging;

           return base.OnStart();
       }

       private void RoleEnvironmentChanging
       ➥ (object sender, RoleEnvironmentChangingEventArgs e)
       {
           if (e.Changes.Any(change => change is
           ➥ RoleEnvironmentConfigurationSettingChange))
           {
               Trace.WriteLine("Configuration Setting Changed");

               // Set e.Cancel to true to restart this role instance
               e.Cancel = true;
           }
       }
   }
```

So far we've had a good look at how to use the `ConfigurationSettings` functionality in Windows Azure. The problem is that the `ConfigurationSettings` section of the service configuration file is only a replacement for the data that you would normally store in the `appSettings` section of the web.config file. How do you handle other types of configuration data?

5.3 Configuring non-application settings

In this section we'll take a look at when you would use the service configuration file and when you would use regular old application settings to set up configuration data. We're going to look specifically at where you should store the following kinds of information:

- Database connection strings
- Application build configuration
- Tweakable configuration
- Endpoint configuration

5.3.1 Database connection strings

For database connection strings (such as SQL Azure DB), you might have different development, staging, and production connection strings. In standard ASP.NET web applications, you would store the connection string in the `connectionStrings` element of the web.config file. Like `appSettings`, Windows Azure doesn't provide you with the ability to change data stored in the `connectionStrings` element at runtime. You can, however, treat a connection string like an application setting and store it in the service configuration. Using configuration settings allows you to modify the connection string at runtime via the Azure portal.

5.3.2 Application build configuration

Some configuration data can be tweaked on an individual server via web.config but is never different between various deployments. This configuration should remain in web.config and be treated as part of the application build rather than as a runtime setting. Typical examples include the `httpModules` and `httpHandlers` sections of web.config.

5.3.3 Tweakable configuration

It can be useful to tweak some configuration settings when an unexpected issue comes up, without redeploying the application. For example, a user might upload a file that's larger than you originally configured the web server to handle. In this case, you might decide to modify the maximum HTTP request size in web.config to allow larger files to upload. The following configuration shows this web.config setting:

```
<httpRuntime maxRequestLength="1048576"/>
```

In a conventional production environment, you would probably test this change on a staging environment and then roll out the change to the live environment without redeploying the entire application. Typically, this change would be reflected in source control so that all future builds would have the new settings.

The web.config file you're used to editing at runtime to change behavior is read-only in the cloud. Because you can't change the web.config file dynamically in Windows Azure, you would have to redeploy the full application in the Azure portal to

deploy a changed web.config file. Instead, you can move these configuration elements to the cloud service definition. Then you can update the settings during runtime through the portal. This might cause a restart of your instances (in a controlled upgrade-style manner), but you aren't deploying new code.

5.3.4 Endpoint configuration

Let's say you have an ASP.NET web page that communicates via WCF to an external WCF service. You might have a different endpoint that you communicate with between the development, production, and staging systems. How can you configure your applications hosted by Windows Azure to use external endpoints, such as WCF services?

The following extract from web.config represents the WCF configuration for a typical WCF client proxy (in this example, we've omitted the binding configuration).

```
<client>
   <endpoint address="http://myservice.com/services/timeservice.svc"
             binding="basicHttpBinding"
             bindingConfiguration="BasicHttpBinding_ITimeService"
             contract="TimeService.ITimeService"
             name="BasicHttpBinding_ITimeService" />
</client>
```

Because you can't modify this information in the Windows Azure portal, you can't change the endpoint between staging and production without redeploying the application. What you can do though is extract the information that can change dynamically (in this case, `http://myservice.com/services/timeservice.svc`) and store it in the service configuration file, leaving all the other configuration as it is (accepting that if that needs to change it will require a redeploy). You can then modify the endpoint in a WCF proxy by modifying the endpoint address on the proxy:

```
var timeServiceProxy = new TimeServiceClient();

Uri timeServiceEndpointAddress =
   new
     Uri(RoleEnvironment.GetConfigurationSettingValue("timeServiceEndpoint"))
     ;

timeServiceProxy.Endpoint.Address =
   new EndpointAddress(timeServiceEndpointAddress);
```

You're doing well so far. You know all about how to configure your own custom runtime settings and how to modify them at runtime. You can configure an ASP.NET web application to work in Windows Azure and you're familiar with the sorts of things you can and can't do in Windows Azure with regard to configuration. Now let's revisit how you can develop an application that'll work in both environments.

5.4 Developing a common code base

Because you've just seen how configuration settings are dramatically different between Windows Azure and standard ASP.NET, let's talk about how to develop a unified system

that'll work in both environments. Typically, there are two situations in which you might want to use a common code base for configuration settings:

- You have a common library shared across multiple projects.
- There are two versions of your web application (a cloud version and an on-premises version).

To successfully share configuration settings across Windows Azure applications and applications not running in the cloud, abstract your configuration settings so they can be read either directly from the web.config or app.config file, or via the `RoleEnvironment` class. You can implement this in one of two ways: use the `RoleEnvironment.IsAvailable` property or the configuration settings inversion of control container. Let's examine each of these implementations.

5.4.1 Using the RoleEnvironment.IsAvailable property

Unfortunately, if you attempt to access configuration settings outside the Windows Azure fabric using the `RoleEnvironment.GetConfigurationSettingValue` method, a very large, nasty exception will be thrown. The reason is that the `GetConfigurationSettingValue` method is a Windows Azure–only method that isn't supported in standard .NET applications.

You can get around this issue by performing a check to see whether you're running in Azure using the `RoleEnvironment.IsAvailable` property, as shown in the following code:

```
public static string GetSetting(string settingName)
{
   if (RoleEnvironment.IsAvailable)
   {
      return RoleEnvironment.GetConfigurationSettingValue(settingName);
   }
   else
   {
      return ConfigurationManager.AppSettings[settingName];
   }
}
```

This is probably the simplest method of sharing configuration settings that span across applications that are hosted by Windows Azure and applications that aren't.

In the example code, the `GetSetting` method checks whether the ASP.NET web application is running in the fabric using the `RoleEnvironment.IsAvailable` property. If the application is hosted in Windows Azure (development fabric or live system), then it uses the `RoleEnvironment` class to retrieve the configuration setting; otherwise, it uses the standard method of retrieving a setting from `appSettings`.

Although the previous example is a simple one, using this method will ensure that your non-Windows Azure applications will reference Windows Azure assemblies even though your applications aren't running in the Windows Azure environment. One final thing to remember: your common libraries and any versions of your applications

that aren't running in the cloud need to reference the `Microsoft.WindowsAzure.ServiceRuntime` assembly.

5.4.2 Pluggable configuration settings using inversion of control

If you want to keep your application layers clean, consider using the *inversion of control* (IoC) pattern. The IoC pattern is pluggable architecture that allows you to decouple the execution of an operation from its implementation. Call `GetSetting` in a common layer and let the application figure out the implementation to use (`appSettings` or `RoleEnvironment`), as appropriate. In the following four sections we'll use the *Unity Application Block*, or *Unity* (which is part of Microsoft Enterprise Library 4.1) to implement this pattern, but you can use any IoC framework.

DEFINING YOUR CONFIGURATION SETTINGS INTERFACE

To support a pluggable architecture you need an interface that both the `RoleEnvironment` and `appSettings` versions of your `ConfigurationSettings` classes can access, such as the following:

```
public interface IConfigurationSettings
{
   string GetSetting(string settingName);
}
```

This interface exposes a single method that will accept the name of the setting to retrieve and return the value of the setting.

IMPLEMENTING YOUR INTERFACE IN STANDARD WEB APPLICATIONS

Now that you've got your interface, you can implement that interface to return the setting value from the `appSettings` section of the web.config or app.config file using the following code:

```
public class AppConfigConfiguratonSettings : IConfigurationSettings
{
   public string GetSetting(string settingName)
   {
      return ConfigurationManager.AppSettings[settingName];
   }
}
```

The class above implements `IConfigurationSettings` and uses `System.Configuration.ConfigurationManager` to return the setting value. The `AppConfigConfigurationSettings` class should ideally be located in the same assembly as the interface and both `IConfigurationSettings` and `AppConfigConfigurationSettings` should be located in a common assembly that all your applications can access.

IMPLEMENTING YOUR INTERFACE IN WINDOWS AZURE WEB ROLES

Finally, you need to provide an implementation of `IConfigurationSettings` that can be used to retrieve the configuration settings from the service configuration file via `RoleEnvironment`:

```
public class RoleEnvironmentConfigurationSettings : IConfigurationSettings
{
```

```
    public string GetSetting(string settingName)
    {
        return RoleEnvironment.GetConfigurationSettingValue(settingName);
    }
}
```

This class can be provided in a separate assembly that's not included with non-Windows Azure instances of your web application and that's not referenced by any of the common layers. As you can see from this implementation, the common layers have no references to any of the Windows Azure SDK assemblies.

CALLING THE CORRECT IMPLEMENTATION

Now that you've defined your pluggable classes, how do you call the correct implementation of the interface? Using the Unity implementation of IoC, reference the Unity assembly (Microsoft.Practices.Unity.dll) from your common assembly and add the following namespaces:

```
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.Configuration;
using Microsoft.Practices.Unity.StaticFactory;
```

> **Using Unity**
>
> Download Unity from Microsoft's Patterns and Practices group as part of Enterprise Library 4, Enterprise Library 5, or stand-alone. See http://unity.codeplex.com/ for details.
>
> Unity isn't the only implementation of IoC available; it just happens to be the one used in this example. For more information about Unity or IoC, go to http://msdn.microsoft.com/en-us/library/ff647202(v=pandp.10.aspx.

Then, when you need to retrieve a configuration setting in your application, you call the following code:

```
IUnityContainer myContainer = new UnityContainer();

IConfigurationSettings configurationSettings =
    myContainer.Resolve<IConfigurationSettings>();

configurationSettings.GetSetting("mySetting");
```

When you call the `Resolve` method, that method determines which configuration settings provider is registered, and returns an instance of that class. Then you can retrieve the configuration setting from the registered provider (`appSettings` version or `RoleEnvironment` version) using the interface method `GetSetting`.

Finally, if you want a class to use the `appSettings` version in your applications that aren't hosted by Windows Azure, register it using the following code:

```
IUnityContainer myContainer = new UnityContainer();

myContainer.RegisterType<IConfigurationSettings,
                           AppConfigConfiguratonSettings>();
```

To register the Windows Azure version, you could call the following code instead from your application:

```
IUnityContainer myContainer = new UnityContainer();

myContainer.RegisterType<IConfigurationSettings,
                         RoleEnvironmentConfigurationSettings>();
```

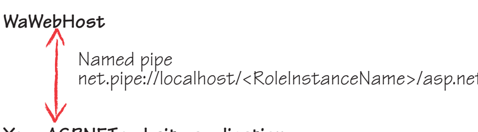
Unity supports both configuration via code, which is what you see above, and via configuration files. You could easily determine which `IConfigurationSettings` to load using the `RoleEnvironment.IsAvailable` method to tell you whether the code was running in the fabric, or you could extract which version to load into a configuration file. Using the configuration file allows you to be a little more flexible by having one configuration file for on-premises and one for the cloud.

Now you know how to share your data across Windows Azure and other applications by using the `RoleEnvironment` class to interact with the Windows Azure runtime. We think it might be interesting to rip away some of the covers and look at how this interaction happens.

5.5 The RoleEnvironment class and callbacks

The `RoleEnvironment` class is hosted by the `Microsoft.WindowsAzure.ServiceRuntime` assembly, which is provided as part of the Windows Azure SDK. On startup of your application, the `WaWebHost` service attempts to call `RoleEnvironment.Initialize` on any hosted web role application. This call populates a singleton instance of the `RoleEnvironment` class (private static variable called at runtime), which is used by static wrapper methods such as `GetConfigurationSettingValue` and `IsAvailable`.

When the `RoleEnvironment` class initializes, a callback is created between the `RoleHost` and the `RoleEnvironment` runtime instance. If any configuration changes are made on the Azure portal and propagated to the VM, the `RoleHost` invokes the callback and notifies the web role that some configuration has changed. This callback mechanism is created across a named pipe. Figure 5.4 shows the communication between your application and the role host.



If the application isn't running in the fabric, then the `Initialize` method is never called; the singleton instance is null. `RoleEnvironment.IsAvailable` checks whether that singleton object is null.

So what happens under the hood when `RoleEnvironment.GetConfigurationSettingValue` is called? The `GetConfigurationSettingValue` static method is calling `GetConfigurationSettings` on an instance of `InteropRoleManager` to retrieve the value. Any changes to the configuration are propagated via a callback over the named pipe. In this way, any configuration settings can be retrieved quickly from an

in-memory copy of the configuration settings that were populated as part of the `AcceptConfigurationChanges` callback.

5.6 Summary

In this chapter, we looked at the final piece of the service model puzzle, configuration. You should now fully understand the service model of Windows Azure and be able to effectively define and configure your service inside and outside Windows Azure. With this knowledge, you should understand the effect of the service model and how it applies to your application.

The service configuration file (ServiceConfiguration.cscfg) stores your dynamic configuration settings. You can change the settings in this file without redeploying your role. You configure the most important piece of information in Windows Azure, the number of instances for your role, in this file. We also looked at configuring certificate information and how to dynamically change these settings at runtime.

Comparing conventional ASP.NET applications with Windows Azure applications is a good way to think about how to define, configure, and dynamically modify your runtime configuration settings. Using its `ConfigurationSettings` functionality, Windows Azure can read application settings from the web.config file. A better way to read those settings is to use the `RoleEnvironment` class.

Sharing code between applications running in Windows Azure and other applications not running in the cloud is just a matter of using the `RoleEnvironment.GetConfigurationSettingValue` method to share your configuration settings. Alternatively, you can use the IoC pattern to achieve the same outcome.

We explained the internals of the `RoleEnvironment` class as exposed via the `ServiceRuntime` class. The `RoleEnvironment.IsAvailable` class works in both Azure and non-Azure environments, but you can't use the class `RoleEnvironment.GetConfigurationSettingValue` outside Azure (because there's no named-pipe interface in a non-Azure environment). There's no performance hit involved when you use the `RoleEnvironment` class because all data is cached in memory.

With all your newly gained knowledge, it's time to move on. We're going to enlighten you as to the mysteries and wonders of web roles.

Part 3

Running your site with web roles

By now, you should know that Azure supports two types of server templates, called *roles*. This part dives super deep into web roles.

Chapter 6 discusses how to scale your web roles for performance reasons. Chapter 7 shows you how to run non-.NET code, like PHP, Ruby, Java, and so on. We also uncover all sorts of code dark magic, like spawning processes and threads, and calling native code.

6 Scaling web roles

This chapter covers

- What happens to your web server under extreme load
- Scaling your web role
- The load balancer
- Session management
- Caching

One of the coolest things about Windows Azure is that you can dynamically scale your application. Whenever you need more computing power, you can just ask for it and get it (as long as you can afford it). The downside is that in order to harness such power, you need to design your application correctly. In this chapter, we'll look at what happens when your application is under pressure and how you can use Windows Azure to effectively scale your web application.

6.1 What happens to your web server under extreme load?

Back in chapter 1, we talked about the challenges of handling and predicting growth for typical websites. In this section, we want to show you what happens to a web server when it's under extreme load and how it handles itself. Using the Ashton

Kutcher example from chapter 1, what would happen to your web server if Ashton Kutcher twittered about your little Hawaiian Shirt Shop and you suddenly found thousands of users trying to access your website at the same time?

In an ideal world, if your website (or service) reached its maximum operating capacity, all other requests would be queued and the application could handle the load at a graceful, yet throttled, rate. In the real world, your website is likely to explode into a ball of flames because the web server will continue to attempt to process all requests (regardless of the rate at which they occur). The processing time of the requests will increase, which results in a longer response time to the client. Eventually, the server will become flooded with requests and it won't be able to service those requests anymore. The server is effectively rendered useless until the requests reduce in volume.

In this section, we'll look at how a web server performs both under normal and extreme load by doing the following:

- Building a sample application that can run under extreme load
- Building an application that can simulate extreme load on your web server
- Looking at how your sample application responds when the server is under load
- Increasing the ability to process requests by scaling up or out

To do all this you need to build a small ASP.NET web page sample application that you can use in all these scenarios.

6.1.1 Web server under normal load

The web page that you're about to build will perform an AJAX poll that returns the time a request was made. Under normal operation, the page should return the current server time every 5 seconds. Figure 6.1 shows this AJAX web page adequately handling the load during normal operation.

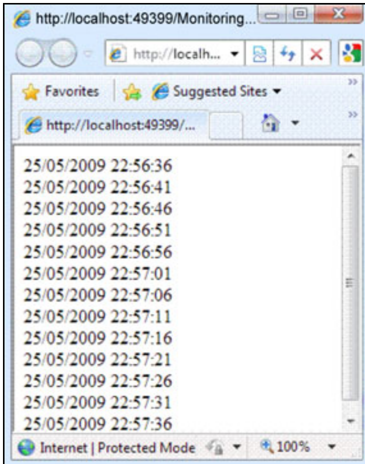


Figure 6.1 An ASP.NET web page making an AJAX request and returning the server time every 5 seconds

Let's take a look at how you can build this web page. The following listing gives the code for the ASP.NET AJAX web poll shown in figure 6.1.

Listing 6.1 AJAX poll web form

```
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<asp:UpdatePanel   ID="RequestsPanel" runat="server"
                   UpdateMode="Always">                       ❶
   <ContentTemplate>
      <asp:Timer ID="Timer1" runat="server"
                 Interval="5000" OnTick="Timer1_Tick" />      ❷
      <asp:Label ID="resultLabel" runat="server" />           ❸
   </ContentTemplate>
</asp:UpdatePanel>
```

❶ Normal AJAX panel makes easy screen updates
❷ Timer refreshes the panel
❸ Label contains date output

At ❶ is a standard ASP.NET AJAX update panel that contains a timer ❷, which is set to poll the web server every 5 seconds. When the timer expires, the `resultLabel` ❸ is concatenated with the current date and time using the following code:

```
protected void Timer1_Tick(object sender, EventArgs e)
{
   resultLabel.Text += DateTime.Now.ToString() + " <br/>";
}
```

So far, everything is fine and dandy. Let's now try to simulate what would happen if your server came under extreme load.

6.1.2 Simulating extreme load

To simulate extreme load, you're going to build a web page that will put your server on an extreme diet (you're going to starve it). You'll build an ASP.NET web form that will send the current thread to sleep for 10 seconds. If you then make enough requests to your web server, you should starve the thread pool, making your website behave like a server under extreme load.

THE LONG-RUNNING WEB PAGE

The following code shows the markup for your empty ASP.NET page:

```
<form id="form1" runat="server">
  <div>There is no need for any code in this sample</div>
</form>
```

This code is for a simple web page; you don't need your web page to display anything exciting. You just need the code-behind for your page to simulate a long-running request by sending the current thread to sleep for 10 seconds, as shown below:

```
protected void Page_Load(object sender, EventArgs e)
{
   Thread.Sleep(10000);
}
```

Now that you have a web page that simulates long-running requests, you need to put your web server under extreme load by hammering it with requests.

HAMMERING YOUR LONG-RUNNING WEB PAGE WITH LOTS OF REQUESTS

To simulate lots of users accessing your website, create a new console application that will spawn 100 threads. Each thread will make 30 asynchronous calls to your new web page.

> **NOTE** Your mileage might vary! You might need to increase the number of threads or calls to effectively hammer the web server.

The following listing shows the code for the console application.

Listing 6.2 Console application code for simulating extreme load

```
static void Main(string[] args)
{
```

```
    Console.WriteLine("Creating 100 threads");
    var webAddress =
       new Uri("http://localhost:49399/Monitoring/CPUThrash.aspx");
for (int i = 0; i < 100; i++)
{
    var t1 = new System.Threading.Thread
    ( () =>
      {
         for (int j = 0; j < 30; j++)
         {
           WebClient client = new WebClient();
           client.DownloadStringAsync(webAddress);
         }
      }
    );

    t1.Start();
}

Console.ReadKey();
}
```

1. Defines the URI
2. Defines the thread
3. Calls the page
4. Spawns the thread

In listing 6.2, you define the URI of your long-running web page at 1. Then you iterate through a loop 100 times, creating a new thread at 2, which you'll spawn at 4. You'll spawn 100 threads.

Each thread will execute the code defined at 2 within the lambda expression. This thread will loop 30 times, making an asynchronous request to the web page at 3. In the end, you should be making around 300 simultaneous requests to your website. How is the web server going to perform?

6.1.3 How the web server responds under extreme load

In figure 6.1, you saw the response time of a simple AJAX website being polled every 5 seconds under normal load. In that example, there were no real issues and the page was served up with ease.

Figure 6.2 shows that same web application, this time coughing and spluttering as it struggles to cope with the simulated extreme load. The extreme load that this page is suffering from is the result of running your console application (listing 6.2) at the same time as your polling application.

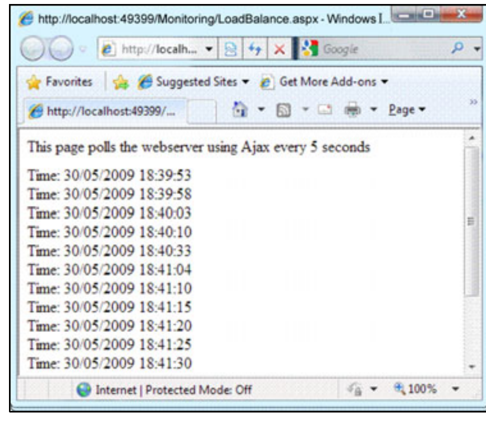


Figure 6.2 AJAX poll under extreme load (running at the same time as your console application)

Figure 6.2 shows that your polling web page is no longer consistently taking 5 seconds to return a result. At one point (between 18:40:33 and 18:41:04), it took more than 20 seconds to return the result. This type of response is typical of a web

server under extreme load. Because the web server is under extreme load, it attempts to service all requests at once until it's so loaded that it can't effectively service any requests. What you need to do now is scale out your application.

6.1.4 Handling increased requests by scaling up or out

If you're hitting the limits of a single instance, then you should consider hosting your website across multiple instances for those busy periods (you can always scale back down when you're not busy). Let's take a quick look at how you can do that manually in Windows Azure.

SCALING OUT

By default, a Windows Azure web role is configured to run in a single instance. If you want to manually set the number of instances that your web role is configured to run on, set the `instances count` value in your service configuration file in the following way:

```
<WebRole name="MyWebRole"
  <Instances count="1" />
  <ConfigurationSettings />
</WebRole>
```

This configuration shows that your web role is configured to run in a single instance. If you need to increase the number of active web roles to two, you could just modify that value from `1` to `2`:

```
<Instances count="2" />
```

Because the number of instances that a web role should run on is stored in the service configuration file, you can modify the configured value via the Azure Service portal at runtime. For more information about how to modify the service configuration file at runtime, see chapter 5.

> **Scaling out automatically**
>
> If you don't fancy increasing your web role instances manually, but you want to take advantage of the ability to increase and decrease the number of instances depending on the load, you can do this automatically by using Windows Azure APIs.
>
> You can use the diagnostics API to monitor the number of requests, CPU usage, and so on. Then, if you hit a threshold, use the service management APIs to increase the number of instances of your web role.
>
> Alternatively, you could increase or decrease the number of instances, based on the time of day, using a model derived from your web logs. For example, you could run one instance between 3:00 a.m. and 4:00 a.m., but run four instances between 7:00 p.m. and 9:00 p.m. To use this kind of schedule, you create a Windows Scheduler job to call the service management API at those times.
>
> In chapter 15, we'll look at how you can automatically scale worker roles. The techniques used in that chapter also apply to scaling web roles.

You've increased the number of instances that host your web role. Now, if you were to rerun your AJAX polling sample and your console application, you would see your polling application responding every 5 seconds as if it were under normal load (rather than taking 30 seconds or so to respond).

Scaling out your web role is great if you designed your application to use this method, but what if you didn't think that far ahead? Well, then, you can scale it up.

SCALING UP

Uh-oh. You didn't design your application to scale out. Maybe your application has an affinity to a particular instance of a web role (it uses in-process sessions, in-memory caches, or the like); in that case, you might not be able to scale out your web role instance. What you can do is run your web application on a bigger box by modifying the `vmsize` value in the service definition file:

```
<WebRole name="MyWebRole" vmsize="Medium">
```

By default, the web role is hosted on a small VM that has 1 GB of memory and one CPU core. In the example above, you've upgraded your web role to run under a medium VM, which means that your web role has an extra CPU core and more memory. For full details about the available VM sizes, see chapter 3, section 3.4.4.

By increasing the size of the VM, your ASP.NET AJAX web polling application should be able to handle the extreme load being placed on the server (or at least process requests a little quicker).

> **Try to avoid scaling up and scale out instead**
>
> Although scaling up will get you out a hole, it's not an effective long-term strategy. Wherever possible, you should scale out rather than up. At the end of the day, when you scale up, you can only scale up to the largest VM size (and no further). Even that might not be enough for your most extreme needs.
>
> Also, it's not easy to dynamically scale your application up and down, based on load. Scaling out requires only a change to the service configuration file. Scaling up requires you to upgrade your application (because it requires a change to the service definition file).

Now you understand what happens when your server is placed under extreme load and how to effectively handle that situation by scaling your application. What you need to know now is how requests are distributed across multiple web role instances via the load balancer.

6.2 How the load balancer distributes requests

In this section, we'll look at how the load balancer distributes requests across multiple servers and how it reacts under failover conditions. In the end, you'll understand how your application will react and behave when you use multiple instances of your web role.

We're going to look at two load balancers: the development fabric load balancer and the production load balancer. Before we do that, you're going to build a sample application to demonstrate the effects of the load balancer coordinating requests between multiple servers.

6.2.1 Multi-instance sample application

The sample application that you're about to build is a web page that consists of a label that displays the name of the web server that processed the request, and a button that posts back to the server when it's clicked. Every time the page is loaded (either on first load or when the button is clicked), the page writes a message to the diagnostic log. Figure 6.3 shows the web page and its log output in the development fabric.

Now we'll walk you through the steps of creating this simple web role so that you'll be able to be able to see the kind of output shown in figure 6.3.

CREATE THE WEB PAGE

To build the sample application shown in figure 6.3, create a new ASP.NET web role (if you're unsure how to do this, refer to chapter 1). In the web role project that you just created, add a new ASP.NET web page called MultipleInstances.aspx.

Before you can run this web page, you must enable native execution in your service definition file in your Cloud Service project:

```
<WebRole name="MyWebRole"
         enableNativeCodeExecution="true">
```

THE ASPX MARKUP

Now that you've added the page, add the following markup in MultipleInstances.aspx:

```
<div>
   Machine Name: <asp:Label ID="lblMachineName" runat="server" />
</div>

<asp:Button ID="btnClickMe" runat="server" Text="Click Me"/>
```

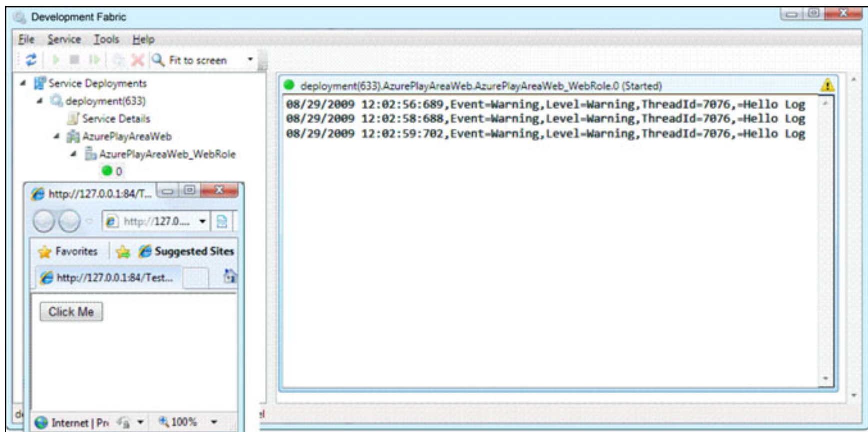


Figure 6.3 A single instance of your web role running in the development fabric, writing out to the diagnostic log

This code represents the page displayed in figure 6.3. There's a label that displays the name of the machine and a button that will post back to the web server when it's clicked.

THE CODE-BEHIND

If you look at figure 6.3, you can see that each time the page is loaded, the name of the web server that processed the request is displayed. To display the web server name that's in the `lblMachineName` label, add the following code to the `Page_Load` event of the ASP.NET page:

```
var computer = new Microsoft.VisualBasic.Devices.Computer();
lblMachineName.Text = Environment.MachineName;
```

Finally, in order to write some data to the log as shown in figure 6.3, you need to write a message to the log on page load. Add the following code to the `Page_Load` event of the ASP.NET page:

```
System.Diagnostics.Trace.WriteLine("Hello Log");
```

Now, fire up the application in the development fabric. You'll see a message written to the log displayed in the development fabric UI every time the page is loaded.

Great! You've got your application working. Now let's go back and take a look at how the load balancers route requests.

6.2.2 The development fabric load balancer

In typical ASP.NET web farms, it's not easy to simulate the load balancing of requests. With Windows Azure, a development version of the load balancer is provided so you can simulate the effects of the real load balancer. The development fabric load balancer helps you find and debug any potential issues that you might have in your development environment (yep, there's a debugger, too). So let's take a look at how the development fabric load balancer behaves.

If you were to fire up your web application in the development fabric with two web roles configured, you would notice two instances of your web role displayed in the UI. Similar to the live production system, the development fabric distributes requests between the instances of your web role. Each time someone clicks the button on the web page, the request is distributed to one of the web role instances. Figure 6.4 shows the two instances of the web role in the development fabric UI.

You can see two instances of your web role in the development fabric, but how is this represented on your development machine?

MULTIPLE INSTANCES OF WAWEBHOST.EXE

In chapter 4, you discovered that Windows Azure hosts the IIS 7.0 runtime in-process in the WaWebHost.exe process using the Hostable Web Core feature of IIS 7.0 rather than using the default w3wp.exe process. Because your web application is hosted by the WaWebHost.exe process, multiple instances of WaWebHost.exe are instantiated if you increase the number of instances of your web role that need to be hosted (one process

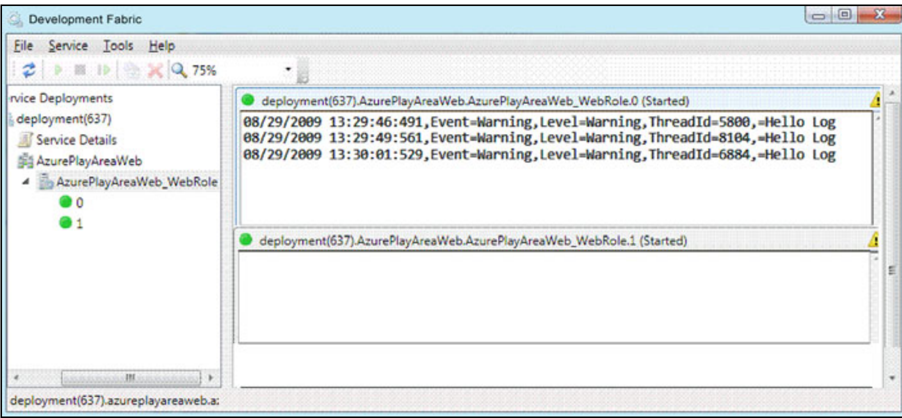


Figure 6.4 Multiple instances of the web role in the development fabric

per instance is instantiated). Figure 6.5 shows the process list of a development machine when it's running multiple instances of the web role.

In figure 6.5 there are two instances of `WaWebHost` in the Processes list, one per web role instance.

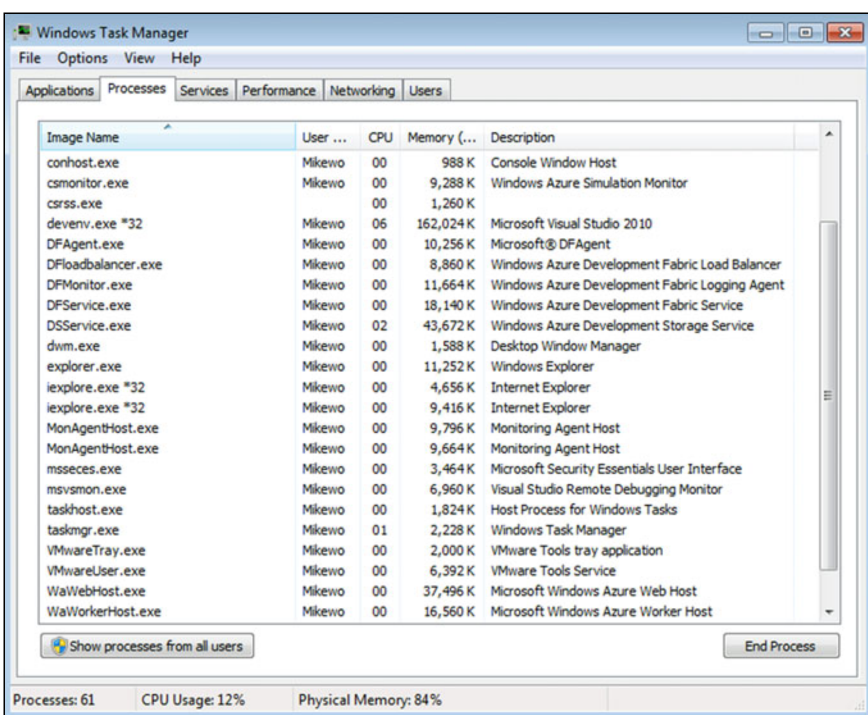


Figure 6.5 Multiple instances of `WaWebHost` shown in the Processes list

Development fabric load balancer process

To help simulate the live production environment, the Windows Azure SDK includes a development load balancer that's used to simulate the hardware load balancers that run in the Windows Azure data centers. Without the development fabric load balancer, it would be difficult to simulate issues that occur across multiple requests (you can't attach a debugger in the live production environment, but you can in the development environment).

The process that simulates the load balancer in the development fabric is called DFLoadBalancer.exe, which is shown in figure 6.5. All HTTP requests that you make in the development fabric are sent to this process first and then distributed to the appropriate `WaWebHost` instance of the web role.

If you were to kill one of the `WaWebHost` instances, then all requests made to the development fabric load balancer would be redistributed to the other instance of `WaWebHost`; the other `WaWebHost` process would be automatically restarted. By performing this test, you can simulate what will happen to your application if there's a hardware or software failure in the live environment.

If you were to kill your DFLoadBalancer.exe process, the entire development fabric on your machine would be shut down and would require restarting.

TESTING WITH MULTIPLE BROWSER INSTANCES

When you're testing your application in the development fabric to see how it responds when requests occur across multiple instances, you should check that your requests haven't just been processed by a single instance of the process.

In the development fabric, the load balancer tends to favor a particular instance (per browser request) unless that role is under load. If you look at figure 6.4, you'll see that although there are two instances of your web role running, the development load balancer seems to be routing all traffic to a single instance.

Because each browser window in the development fabric tends to have affinity with a particular web role instance, you should test your application using multiple browser instances. Figure 6.6 shows the outcome of a test in which both instances of the web role are being used by alternately clicking the button in two different browser instances.

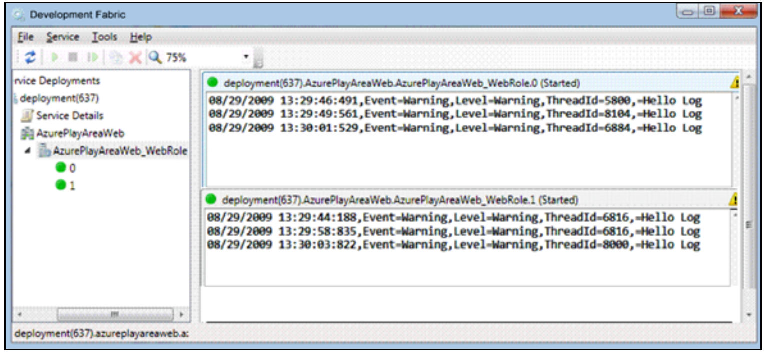


Figure 6.6 Multiple instances of the web role with requests distributed

Now that two different web pages are running, the requests are distributed across both instances of the development fabric.

Although spinning up multiple instances of your web application with multiple browser instances will allow you to test, in your development fabric, that your application will work with multiple roles, it won't test the effects of requests being redistributed to another server. To test that your application behaves as expected in your development fabric when running multiple instances, there are two things that you can do: you can restart one of the `WaWebHost` instances, or you can test your application under load.

RESTARTING THE WAWEBHOST INSTANCE

Although the development fabric load balancer creates affinity between your process and an instance of `WaWebHost`, if you kill that `WaWebHost` instance, the affinity is broken. When the affinity is broken, the request is redistributed to another instance of `WaWebHost`, which creates a new affinity between the web browser and that instance of the web role. Figure 6.7 shows the changing of affinity between a browser and a web role instance when a WaWebHost.exe process is killed.

In the example shown in figure 6.7, two instances of the web role are running (instance 0 and instance 1), and there's a single web browser. Instance 1 didn't process any requests prior to the restarting of the WaWebHost.exe process of instance 0; the browser had affinity with instance 0. When instance 0 was restarted, instance 1 then processed all incoming requests and instance 0 no longer processed any requests from the browser because a new affinity was created.

This test is fairly important for you to perform in the development fabric. Restarting one of your `WaWebHost` instances helps to ensure that your application can truly run against multiple instances of your web role and can recover from a disaster. We'll look at these situations in more detail when we look at session, cache, and local storage later in this chapter.

TESTING UNDER LOAD

The second way to test that your application behaves correctly when it's redistributed to multiple instances is to test your application under load. The best method of testing

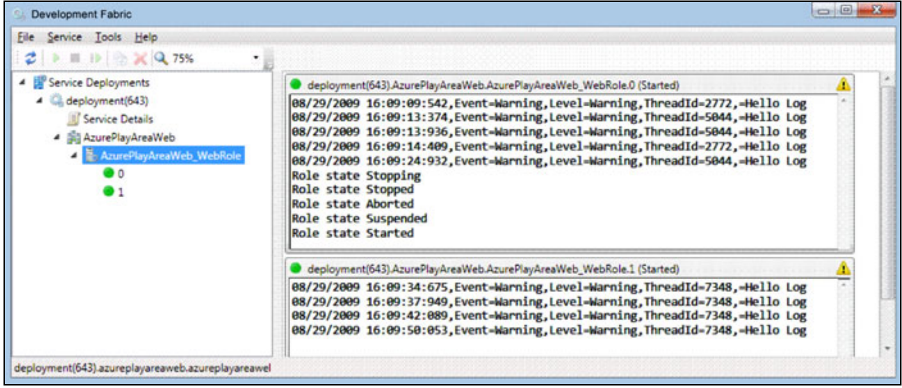


Figure 6.7 Changing of affinity between browser and web role host in the development fabric

this scenario is to use a load testing tool such as Visual Studio Team System Web Load Tester. If you want to ensure that the development fabric load balancer redistributes requests under load, you can simulate this by modifying the sample application that you built earlier. Now, you're going to simulate load by sending the thread to sleep for 10 seconds on page load. You can do this by modifying your `Page_Load` event to include the following code:

```
Thread.Sleep(10000);
```

If you were to now spin up multiple instances of your web page, you would see that as the result of the increase in load, the development fabric load balancer doesn't maintain affinity and starts redistributing the requests more evenly.

Asynchronous AJAX requests

Although the development fabric load balancer keeps affinity between your browser instance and a web role (unless under load), it tends to redistribute requests evenly when performing AJAX requests.

In section 6.1, you created an ASP.NET AJAX web page that asynchronously calls the backend web page every 5 seconds and displays the name of the server that processed the request. If you modify that sample to write to the log, the development fabric load balancer evenly distributes the request, rather than maintaining affinity with a particular web role instance.

Now that you have an understanding of how the development fabric load balancer behaves and how you can effectively test how your application will behave in failover situations, let's look at how things happen in the live environment.

6.2.3 *Load balancing in the live environment*

We've spent quite a bit of time looking at the development fabric load balancer. Hopefully you have a good understanding of how the load balancer works and how it interacts with multiple instances of your web role. Although the development fabric load balancer doesn't behave exactly like the load balancer in the real environment, there are some tricks that you can do to ensure that your application will behave correctly in the live environment. That said, there are some cases in which your application might not behave as expected when it's distributed across physical servers, which isn't something you can easily test for in the development fabric. To ensure that your application will behave correctly prior to making your application live, you'll need to perform some testing in the staging environment.

TESTING IN THE STAGING ENVIRONMENT

In chapter 2, we showed you how to deploy your application to the staging environment and how to move your staging web application to the production environment via the Windows Azure portal. In that chapter, you also learned that when you switch

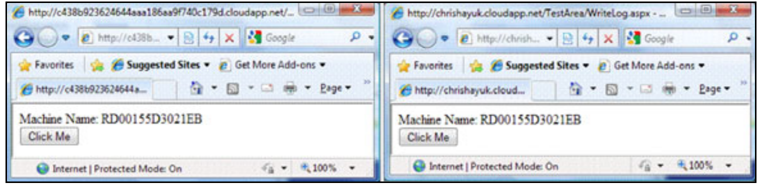


Figure 6.8 Your web application running in the staging environment (left) and in the production environment (right)

your application from staging to production, the application continues to run on the same server as before and the load balancers simply redirect traffic to the correct servers. If you want to, you can prove this by deploying the sample that you built earlier in this chapter to the staging environment (noting the machine name) and then switching over to the production environment (noting the machine name once again). Figure 6.8 shows your application running in the staging environment and in the live environment.

In figure 6.8, the browser on the left is pointed to the staging environment and is running on machine RD00155D3021EB. The browser on the right is your web application running in production after the switchover. Notice that your application is still running on the same server even though you're now running in production (rather than in staging).

Because the staging servers will eventually become the production servers, you should be able to iron out, during your staging testing phase, any errors that might occur when you're running multiple instances of your web role.

> **Nothing beats production environments**
>
> You can't always be sure that an application that works in your development environment will work in your staging environment. On your development machine, `WaWebHost` runs under your user account; anything you can do, it can do. That's not necessarily true with the production servers.

STAGING AND PRODUCTION LOAD BALANCERS

If you modify your ASP.NET web polling application from section 6.1 to display the machine name, and then run it in the staging environment, you get a result that's similar to what's shown in figure 6.9.

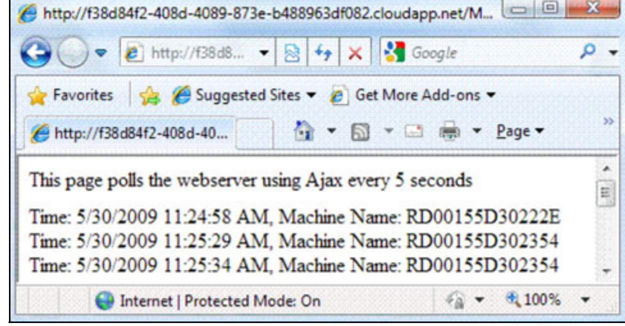


Figure 6.9 Web roles running in the staging environment

The requests are distributed between two different servers, which is great news.

You can be sure that the staging environment will host each instance of your web role on a separate physical machine (a test that you can't perform in your development environment). This behavior could change over time, as Windows Azure matures and is expanded. The point is that you shouldn't rely on any apparent behavior when designing your application. You should design it to be as stateless as possible, with the understanding that successive trips to the server won't necessarily always go to the same server.

Quirky affinitization

Both the staging and the production environments can be a little quirky in how they distribute requests between servers. In some instances, requests will be evenly distributed between all servers (for example, AJAX requests are distributed this way), but generally both environments will maintain affinity between a connection and a web role. If you're testing whether your application works with multiple instances of your web role, you should capture the machine name in your request to ensure that your server can handle requests distributed across instances. Then run your sample applications in both the staging and production environments and monitor how the machine name changes.

INDUCING FAILOVER

In the development fabric, you tested how your application handles failover by killing the WaWebHost.exe process and then monitoring the application's behavior. If you need to, you can perform the same test in the staging and production environments. In the live environment, the web role is also hosted in a process called WaWebHost.exe, so you can kill the process on the live environment using the following command (remember that native execution must be enabled for this to work):

```
System.Diagnostics.Process.GetProcessesByName("WaWebHost").First().Kill();
```

Create a new web page with a button that will execute the above command when it's clicked. Then you can run the AJAX polling application in one browser, kill the `WaWebHost` process in another browser, and watch how the load balancer handles the redirection of traffic. Typically, on the live system (at the time of writing), all traffic is redirected to the single node. When the FC is convinced that the failed role is behaving again, the load balancer starts to direct requests to that server.

If you want to test what happens when all roles are killed, you can execute the command on each role until they're all dead. If no web roles are running in the live environment, your web application won't process requests anymore and your end user will be faced with an error. Typically, your service will be automatically restarted and will be able to service requests again within about a minute.

That's how requests are load balanced across multiple web roles. Now let's take a look at those aspects of a website that are generally affected when running with multiple roles, namely:

- Session management
- Caching
- Local storage

Let's start with session management.

6.3 Session management

HTTP is a stateless protocol. Each HTTP request is an independent call that has no knowledge of state from any previous requests. Using sessions is one method of persisting data so that it can be accessed across multiple requests. In ASP.NET, you can use the following methods of persisting data across requests:

- Sessions
- `ViewState`
- Cookies
- Application state
- Profile
- Database

Throughout the course of this chapter (and future chapters), we'll be looking at the methods of persisting data that you'll use that are affected when you scale to multiple web roles. We won't look at `ViewState` or cookies in this book; these methods aren't used differently in a Windows Azure environment.

In this section, we'll look at how running Windows Azure across multiple roles affects your ASP.NET session and the different types of session providers that you can use. Specifically, we're going to talk about how a session works, and you're going to build a sample session application. We'll also discuss in-process sessions and Table storage sessions.

Although the concept of sessions is probably familiar to most of you, we want to recap the purpose of the session and the `Session` object.

6.3.1 How do sessions work?

A session is effectively a temporary store that's created server-side for a limited window of time for a particular browser instance. Your ASP.NET web application can use this temporary store to store and retrieve data throughout the course of that session.

If we go back to the Hawaiian Shirt Shop example, you can store the shopping cart in a session. Using a session as a storage area lets you store items in the cart, but still have access to the cart across multiple requests. When the session is terminated (the

user closes his browser), the session is destroyed and the data stored in the session is no longer accessible. Similarly, if the user opens a new browser instance, a new independent session is assigned to this new browser instance and it has no access to session data associated with the other browser instances (and vice versa). Figure 6.10 shows how a session is treated with respect to the browser instance and the web server.

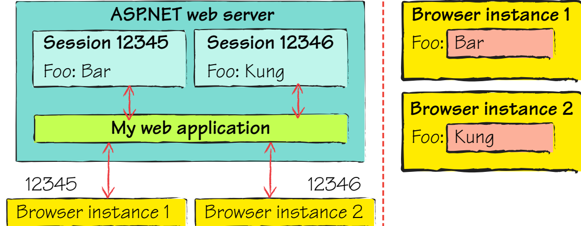


Figure 6.10 Session independence and the temporary store

In figure 6.10, browser instance 1 is associated with session ID 12345; the session key `Foo` has an associated value of `Bar`; browser instance 2 is associated with session ID 12346; and the session key `Foo` has an associated value of `Kung`. If you were to look at the output of browser instance 1 and browser instance 2, they would display the correct values from their associated temporary store.

When a browser makes a request to an ASP.NET website, it passes a session ID in a cookie as part of the request. This session ID is used to marry the request to a session store. For example, in figure 6.10, browser instance 1 has a session ID of 12345 and browser instance 2 has a session ID of 12346. If no session ID is passed in the request, then a new session is created and that session ID is passed back to the browser in the response, to be used by future requests.

If you need to be able to access data beyond a browser session, then you should consider a more permanent storage mechanism such as Table storage or the SQL Azure Database.

Now that we've reminded you how sessions work in ASP.NET, you're going to build a small sample application that you can use to demonstrate the effects of using sessions on your web applications in Windows Azure.

6.3.2 Sample session application

To get started, you need a web page where you can store a value in the session for later retrieval. Add this new ASP.NET web page, called SessionAdd.aspx, to an ASP.NET web role project. Add the following markup to the page:

```
<asp:TextBox ID="txtSessionText" Text="" runat="server"/>
<asp:Button ID="btnAdd" Text="Add"
            runat="server" onclick="btnAdd_Click"/>
```

The markup shows that the page consists of a text box and a button. Use the following code to set the value of the session key `Foo` to what the text box contains when the button is clicked:

```
protected void btnAdd_Click(object sender, EventArgs e)
{
   Session["Foo"] = txtSessionText.Text;
}
```

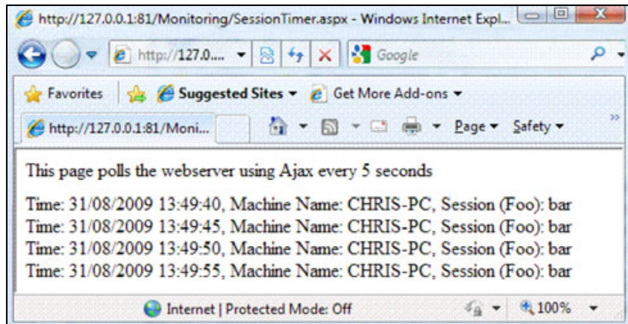


Figure 6.11 Session timer page. Notice that your session is maintained for each request.

Now that you can store some session data in your page, you need a page that you can use to display whatever is stored in `Foo`. For this example, you'll use an ASP.NET AJAX polling timer (similar to the one that you used earlier) that will display whatever is stored in `Foo` every 5 seconds. Figure 6.11 shows how this web page looks (prior to running this page, we used the SessionAdd.aspx page to set the value of `Foo` to `bar)`.

To create the page displayed in figure 6.11, add a new ASP.NET page to the project called SessionTimer.aspx that contains the following markup:

```
<div>
   <asp:ScriptManager ID="ScriptManager1" runat="server" />

   <div style="padding-bottom: 10px;">
      This page polls the webserver using Ajax every 5 seconds
   </div>

   <div>
      <asp:UpdatePanel ID="RequestsPanel"
                       runat="server"
                       UpdateMode="Always">
            <ContentTemplate>
               <asp:Label ID="lblResult" runat="server" />
                  <asp:Timer ID="Timer1"
                             runat="server"
                             Interval="5000"
                             OnTick="Timer1_Tick" />
         </ContentTemplate>
      </asp:UpdatePanel>
   </div>
</div>
```

To display the result of `Foo` in the web page, add the following code-behind to the SessionTimer.aspx page:

```
protected void Timer1_Tick(object sender, EventArgs e)
{
   lblResult.Text +=
      string.Format("Time: {0}, Machine Name: {1}, Session (Foo):{2}<br/>",
      ➥ DateTime.Now.ToString(), Environment.MachineName,
      ➥ Session["Foo"] as string);
}
```

Every 5 seconds the SessionTimer.aspx page makes an AJAX request back to the web server where the request is logged. Then, the name of the computer, the time of the request, and the value stored in the session for `Foo` is returned, all of which is then displayed in the SessionAdd.aspx page.

Using this sample, you can see in both the development and live environments which machine processed the request and what the value of `Foo` is at any particular time.

6.3.3 *In-process session management*

By default, ASP.NET uses an in-process session state provider to store session data. The in-process session state provider stores all session data in memory that's scoped to the web worker process (`w3wp` in standard web servers, or `WaWebHost` in Windows Azure). Let's see how this session provider works.

KILLING YOUR SESSION BY KILLING WAWEBHOST

If the worker process were to be restarted, you would lose any session data because that data is stored in memory. You can simulate this situation in your development environment using your SessionTimer page.

> **TIP** Before you attempt to lose your session, ensure that your ASP.NET web role is running with a single instance.

Go ahead and fire up the SessionAdd.aspx page that you created earlier and set the value of `Foo` to `bar`. After you set this value, open SessionTimer.aspx in the same browser instance. Let the `session` value display a few times and then kill the `WaWebHost` process. As you discovered earlier, if you kill the `WaWebHost` process, the development fabric automatically restarts the process, but all session data is lost. Figure 6.12 shows the result of killing the process.

In figure 6.12, `bar` was displayed up until 13:49:55; just after that point, you killed the `WaWebHost` process. From that point on, the session was lost and no data was returned for all other requests.

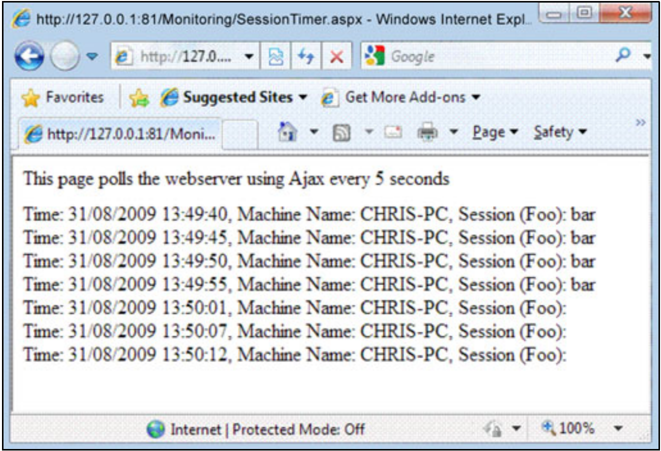


Figure 6.12 Killing the web role that's using in-process sessions

IN-PROCESS SESSION STATE WITH MULTIPLE INSTANCES OF THE WEB ROLE

There are some issues with using the in-process session provider in Windows Azure, but this one is the real killer: if you're using multiple web role instances, Windows Azure doesn't consistently implement sticky sessions in the production environment. Any requests made to a web role might not be routed to the same web role.

As we noted earlier, the production systems generally maintain affinity with a web role but will sometimes evenly distribute requests among roles. In the case of AJAX applications, because requests are likely to be distributed across multiple roles, an in-process session state provider can't be used; the other role won't have access to session data stored in a previous request. Figure 6.13 shows your AJAX polling application running across multiple web roles.

In figure 6.13, you can see that any request made to the first role returns the session data, but any time the request is distributed to another web role, the session data stored in the first web role is no longer accessible. When you're testing your applications in the development environment, you need to keep in mind that sticky sessions aren't always implemented by Windows Azure.

MEMORY CONSUMPTION

If at first you need to run your web role on only a single instance, then you can get better performance by running your application with in-process session management. You should consider this option if you're unconcerned that a user's session might be trashed if the web role is moved to another server (if, for example, the role instance was moved because of a hardware failure). If you need to scale out to multiple servers at a later date, you can always move to a session state provider that'll work cross multiple web roles (such as Table storage) when required.

Before you decide to run with the in-process session state provider, there's one other issue that you should be aware of. If you have a large number of users on your website and they're storing a large amount of session data, you might quickly run into `Out of Memory` exceptions. The web role host doesn't automatically free up any active

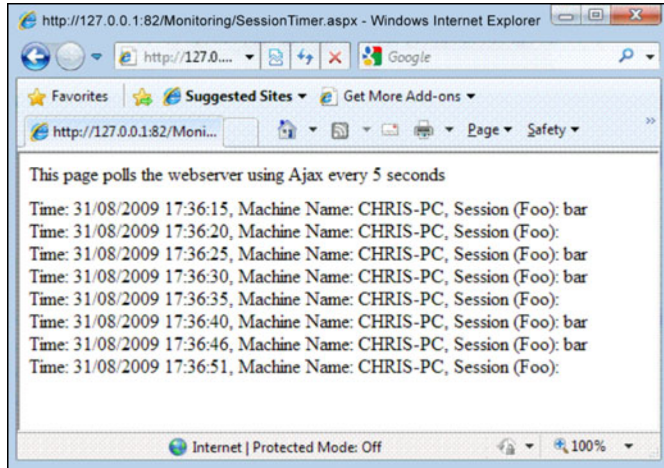


Figure 6.13 Loss of session data across multiple web role instances

session data until sessions start to expire. Remember that your VM only has 2 GB of memory allocated to it if you're running on the smallest (default) size, so you'll run out of memory quite quickly.

If you want to test how your application responds to adding a large amount of session data, you can modify the SessionAdd.aspx page to include a button that will add a large amount of data to the session when clicked. Add the following markup to your SessionAdd.aspx page:

```
<asp:Button ID="btnLarge" Text="Large"
            runat="server" onclick="btnLarge_Click"/>
```

The following code will add a lot of data to the session when the button is clicked: protected void btnLarge_Click(object sender, EventArgs e)

```
{
   StringBuilder sb = new StringBuilder();

   for (int i = 0; i < 100000; i++)
   {
      sb.Append("Hello World");
   }

   for (int i = 0;  i < 10; i++)
      Session[Guid.NewGuid().ToString()] = sb.ToString();
}
```

By repeatedly clicking this new button on your website, you'll find that the memory usage of your WaWebHost.exe process increases until you start getting `Out of Memory` exceptions.

> **NOTE** In Windows Azure, the state server, or out-of-process session state provider, isn't supported.

6.3.4 Table-storage session state sample provider

To maintain a session state that can be accessed by multiple web roles that can have requests evenly distributed between them, you need to use a persistence mechanism that can be accessed by all web roles. In typical ASP.NET web farms, SQL Server is typically used, mainly because ASP.NET has a built-in provider that supports it.

There's a sample online for a Table-storage session state provider that you can use in your ASP.NET web applications.

GETTING STARTED WITH THE TABLE-STORAGE SESSION STATE PROVIDER

To start using the Table-storage session state provider, you need to build the sample provider and then reference that provider in your project. You can get the sample provider at http://code.msdn.microsoft.com/windowsazuresamples.

To build the project, double-click the buildme.cmd file in the directory. After you've built the sample project, add a reference to the assembly in your web role project. Because the Table-storage session state provider is implemented as a custom

provider, you'll need to modify your web.config file to include the provider in the system.web settings:

```
<sessionState mode="Custom"
              customProvider="TableStorageSessionStateProvider">
   <providers>
      <clear/>
         <add name="TableStorageSessionStateProvider"
type="Microsoft.Samples.ServiceHosting.AspProviders
➥ .TableStorageSessionStateProvider"
             allowInsecureRemoteEndpoints="false"
             accountName="devstoreaccount1"
             sharedKey="Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1
             ➥ OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw=="
             containerName="sessionstate"
             applicationName="ProviderTest"
             blobServiceBaseUri="http://127.0.0.1:10000/devstoreaccount1"
             tableServiceBaseUri="http://127.0.0.1:10002/devstoreaccount2"
             sessionTableName="Sessions" />

   </providers>
</sessionState>
```

The above configuration is for using the Table-storage and BLOB-storage providers in the development fabric. The creation of the appropriate tables in the Table-storage account is automatically taken care of for you by the provider. On deployment of your application, you'll need to modify web.config to use your live Table-storage account.

If you now run your application in either the development fabric or the live environment, you'll find that you can store and retrieve session data across multiple instances of your web role.

Ever-growing tables

One word of warning about the Table-storage provider: it doesn't clean up after itself with respect to expired sessions. Because Table storage is a paid, metered service, we advise you to have either a worker role, a simulated worker role (discussed later in this chapter), or a background thread that cleans up any expired sessions from the table. If you don't clean up this data, you'll be paying storage costs for data that is no longer used.

PERFORMANCE CONSIDERATIONS

Although Table storage gives you a session state that's accessible across multiple server instances in a load-balanced environment, it does incur a performance hit. To test the performance of the live system, we modified the Table-storage provider to record the time that lapsed between requesting an item from the session and retrieving a response. Because session state is reloaded from the session provider on every page load, this test will allow you to see the impact of the Table-storage session state provider

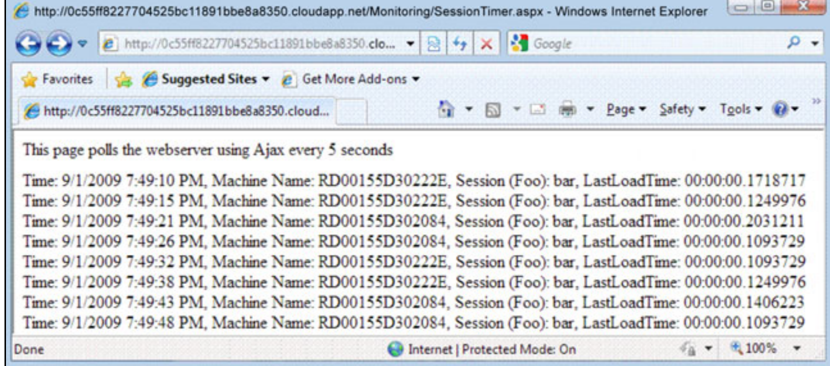


Figure 6.14 Response times of the Table-storage session state provider for a single small item

on your website. Figure 6.14 is a modified version of the SessionTimer.aspx page that you built earlier that also displays the time recorded to load the session during the page load.

You can see in figure 6.14 that although you're storing only one item in the session (`bar` in session key `foo`), it still takes somewhere between 0.1 to 0.2 seconds to retrieve the session state. This load time is probably acceptable for most applications. Table storage is a good solution for the session in Windows Azure until a more performant solution, such as a cache-based session provider, is available.

If you store a large amount of data in the session, you might find the performance of the Table-storage provider a little too slow at the moment. Figure 6.15 shows the SessionTimer.aspx page after we added a large amount of data to the session by clicking the large session button that we built earlier twice.

In figure 6.15, you can see that the performance of the session provider seriously degrades when a large amount of data is stored in the session. In this example, it took 1 to 2 seconds just to load the session. In cases when you need minimal session load times or when you're storing large amounts of data, you should consider another session provider solution (for example, SQL Azure Database or a cache-based session provider).

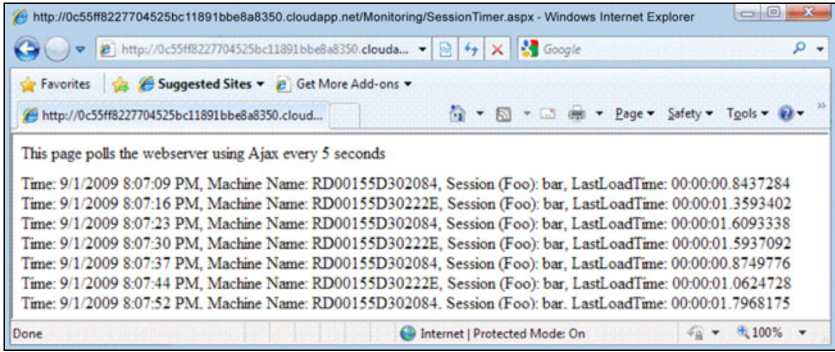


Figure 6.15 Large session state response load times using the Table-storage provider

SQL Azure session state provider

In typical ASP.NET web farms, SQL session state providers are generally used as the session provider. Although this works, it's not the best use of a SQL database; it's not querying across sessions, but rather it's acting as a central storage area.

To date, there isn't a SQL Azure session state provider available (although this could change). Rather than trying to mess with SQL Azure to make it work with sessions, it's probably best to either stick to Table storage, use a cache-based session provider, use an in-process provider, or architect your application so it's not so reliant on sessions.

6.4 Cache management

In any typical website, there's usually some element of static reference data in the system. This static reference data might never change or might change infrequently. Rather than continually querying for the same data from the database, storing the data in a cache can provide great performance benefits.

A *cache* is a temporary, in-memory store that contains duplicated data populated from a persisted backing store, such as a database. Because the cache is an in-memory data store, retrieving data from the cache is fast (compared to database retrieval). Because a cache is an in-memory temporary store, if the host process or underlying hardware dies, the cached data is lost and the cache needs to be rebuilt from its persistent store.

Never rely on data stored in a cache. You should always populate cache data from a persisted storage medium, such as Table storage, which allows you to persist back to that medium if the data isn't present in the cache.

> **NOTE** For small sets of static reference data, a copy of the cached data resides on each server instance. Because the data resides on the actual server, there's no latency with cross-server roundtrips, resulting in the fastest possible response time.

In most systems, there are typically two layers of cache that are used: the in-process cache and the distributed cache. Let's take a look at the first and most simple type of cache you can have in Windows Azure, which is the ASP.NET in-process cache.

6.4.1 In-process caching with the ASP.NET cache

As of the PDC 2009 release, the only caching technology available to Windows Azure web roles is the built-in ASP.NET cache, which is an in-process individual-server cache. Figure 6.16 shows how the cache is related to your web role instances within Windows Azure.

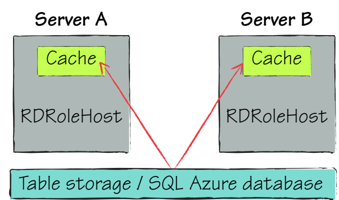


Figure 6.16 shows that both server A and server B maintain their cache of the data that's been retrieved from the data store (either from Table storage or from

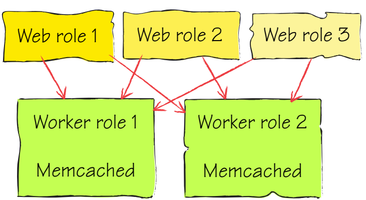
SQL Azure database). Although there's some duplication of data, the performance gains make using this cache worthwhile.

> **In-process memory cache**
>
> You should also notice in figure 6.16 that the default ASP.NET cache is an individual-server cache that's tied to the web server worker process (`WaWebHost`). In Windows Azure, any data you cache is held in the `WaWebHost` process memory space. If you were to kill the `WaWebHost` process, the cache would be destroyed and would need to be repopulated as part of the process restart.
>
> Because the VM has a maximum of 1 GB of memory available to the server, you should keep your server cache as lean as possible.

Although in-memory caching is suitable for static data, it's not so useful when you need to cache data across multiple load-balanced web servers. To make that scenario possible, we need to turn to a distributed cache such as Memcached.

6.4.2 Distributed caching with Memcached

Memcached is an open source caching provider that was originally developed for the blogging site *Live Journal*. Essentially it's a big hash table that you can distribute across multiple servers. Figure 6.17 shows three Windows Azure web roles accessing data from a cache hosted in two Windows Azure worker roles.



> **TIP** In figure 6.17, you can see that your web roles can communicate directly with worker roles. In chapter 15, we'll look at how you can do this in Windows Azure.

Microsoft has developed a solution accelerator that you can use as an example of how to use Memcached in Windows Azure. This accelerator contains a sample website and the worker roles that host Memcached. You can download this accelerator from http://code.msdn.microsoft.com/winazurememcached. Be aware that memcached.exe isn't included in the download. Use version 1.2.1 from http://jehiah.cz/projects/memcached-win32/.

> **Hosting Memcached**
>
> In chapter 7, we'll look at how you can launch executables (such as Memcached) from a Windows Azure role.
>
> Although we're using a worker role to host your cache, you could also host Memcached in your web role (saves a bit of cash).

To get started with the solution accelerator, you just need to download the code and follow the instructions to build the solution. Although we won't go through the downloaded sample, let's take a peek at how you store and retrieve data using the accelerator.

SETTING DATA IN THE CACHE

If you want to store some data in Memcached, you can use the following code:

```
AzureMemcached.Client.Store(Enyim.Caching.
➥ Memcached.StoreMode.Set, "myKey", "Hello World");
```

In this example, the value `"Hello World"` is stored against the key `"myKey"`. Now that you have data stored, let's take a look at how you can get it back (regardless of which web role you're load balanced to).

RETRIEVING DATA FROM THE CACHE

Retrieving data from the cache is pretty simple. The following code will retrieve the contents of the cache using the AzureMemcached library:

```
var myData = AzureMemcached.Client.Get<string>("myKey"));
```

In this example, the value `"Hello World"` that you set earlier for the key `"myKey"` would be returned.

Although our Memcached example is cool, you'll notice that we're not using the ASP.NET `Cache` object to access and store the data. The reason for this is that unlike the `Session` object, the `Cache` object (in .NET Framework 3.5SP1, 3.5, or 2.0) doesn't use the provider factory model; the `Cache` object can be used only in conjunction with the ASP.NET cache provider.

6.4.3 Cache extensibility in ASP.NET 4.0

Using ASP.NET 4.0, you'll be able to specify a cache provider other than the standard ASP.NET in-memory cache. Although this feature was introduced to support Microsoft's new distributed cache product, Windows Server AppFabric Caching (which was code-named Velocity), it can be used to support other cache providers, such as Memcached. The configuration of a cache provider is similar to the configuration of a session provider. You could use the following configuration to configure your cache to use AppFabric caching:

```
<system.caching>
   <cache defaultProvider="FrameworkCacheProvider">
      <providers>
       <add name="myVelocityInstances"

     type="System.Data.Caching.VelocityCacheProvider,System.Data.Caching"
                remoteServerName="myServer"
                remoteServerPort="4435"
                namedCache="myCache"
                securityToken="DEC3D34CA29112" />
      </providers>
   </cache>
</system.caching>
```

Although the Windows Azure team will allow you to hook into any cache provider that you like, the real intention is for you to use a Windows Azure-hosted shared-cache role (that's probably based on AppFabric caching).

> **Cache-based session provider**
>
> Now that you can see the benefits of a distributed cache, it's worth revisiting session providers. As stated earlier, most ASP.NET web farms tend to use SQL Server as the session provider. With the increasing popularity of distributed caches, it's now becoming more common for web farms to use a cache-based session provider rather than a SQL Server–based provider.
>
> When a distributed cache is used in a web farm, it makes sense to leverage it whenever possible to maximize its value. Typically, distributed caches perform better than do databases such as SQL Server; you can improve the performance of your web application by using a faster session provider. Because session state is ultimately volatile data (not unlike cached data), a transactional data storage mechanism such as SQL Server is typically overkill for the job required.
>
> If you want to use a Memcached-based session provider (your session data is stored in your Memcached instance), you can download a ready-made provider from http://www.codeplex.com/memcachedproviders.

6.5 Summary

OK, so you've probably learned everything that's relevant about scaling your web applications from this chapter (and even from some of the earlier chapters). You should've come to realize that websites can't cope with being under pressure and the best thing that you can do is design your web application to scale out. If, for whatever reason (and there isn't a good one that we can think of), you can't scale out, you can always host your website on a bigger box until you can.

In this chapter, we also looked at how Windows Azure distributes requests and how you can test them in your own environment using the development fabric load balancer. Although you can't test every scenario, you can get a sense as to how your application will behave when you run under multiple instances. Finally, you learned how to handle sessions and caching across multiple servers (if you want to do that).

Now that we're starting to look at some of the more advanced web scenarios, in the next chapter we're going to take a peek at how you can use Windows Azure support for full-trust applications, how to build non-ASP.NET–based websites, and how to execute non-.NET Framework applications.

7 Running full-trust, native, and other code

This chapter covers

- Running any Common Gateway Interface (CGI) interpreter you want
- Spawning processes and calling local executables
- Calling native libraries with P/Invoke

Microsoft is committed to making Windows the best place to run any type of application. To that end, it's making Azure an open system, where you can run anything you want. Microsoft could have easily made Azure .NET-only. Azure would've been easier for Microsoft to manage, and easier to design the infrastructure for. But Microsoft didn't do that. It opened Azure up, as wide as the on-premises version of Windows is, so that its customers can run almost anything on Azure that can be run on Windows today. Azure can run unmanaged code (C++, for example), any code that needs full trust on the local machine, and code from any other platform that runs on Windows. There's support for PHP, Python, Ruby, and Java. But Microsoft didn't even stop there.

Microsoft worked with a series of open source teams to provide useful and valuable SDKs for each platform, so that they're equal citizens in the cloud. A plug-in for Eclipse was developed in a partnership between Microsoft and an open source team so that Eclipse developers can have an integrated experience.

The openness of Azure is its power. The core of this openness is Azure's support for running in a full-trust environment. After this environment was enabled, you could run anything on Azure, including FastCGI. Because you can run FastCGI, you can run most other web platforms, including PHP. For some, the challenge isn't about just running a different web platform; it's about running legacy code in a better way. Azure also supports spawning processes and calling into native libraries with P/Invoke so that developers can squeeze all the power out of those massive eight-way servers out there.

Enough background information. Let's talk about how you can harness all the power that Azure has to offer.

7.1 Enabling full-trust support

When any code is run in Windows, it's run in a particular trust level. This trust level defines what the code is allowed to do. For example, code in a partial-trust level can't access local hardware and system resources, whereas code running in a full-trust environment has access to just about anything.

Trust is only one piece of the equation. The user account permissions must allow an operation, in addition to the trust level. Trust level is enforced by *code access security* (CAS). CAS is a way to define what an application is allowed to do and is applied at the Common Language Runtime (CLR) level. A CAS policy can determine what methods and libraries you use and what level of local-system access your code has access to. ASP.NET comes with several standard trust policies, expressed in CAS. One of them is the ASP.NET medium-trust policy, which restricts the application to just being able to run itself, without any access to the broader system at play. The Azure team took this policy, tweaked it to fit its needs, and published a modified medium-trust policy.

You should try to run in the lowest trust level possible. Doing so minimizes the damage that can be done if your application is hijacked, or if your code runs amok. This concept is called *least privileged* and refers to always running your code with the least amount of privileges needed to get the job done. If your code doesn't need access to the registry, it shouldn't have access. Running in this way restricts what the bad guys can do if your system is compromised, or the damage you might cause if some code in your application becomes self-aware and starts to run amok. Bad outcomes might include files being placed in the system folders, your desktop wallpaper being changed to lolcats, or naughty things being written to the registry.

You should run your code in full trust only when you absolutely have to. Unfortunately, full trust is required for any unmanaged code you might want to run and for accessing the Azure diagnostics systems.

There are times when your application legitimately needs advanced permissions. It could be because you're referencing a library that requires them, or you're accessing

the local system somehow. When you need this kind of permission, you can change the configuration of your cloud service to run in a full-trust model. While you're running in a full-trust model, you have access to do just about anything you want. The identity your application is running under is still that of a limited user on the server, which keeps you from creating Windows user accounts and formatting the hard drive. While your code is able to run any opcode through the CLR that can possibly be run, the local user permissions are limited, such that you can only do things for your application and not system wide.

Full trust is enabled by default. To disable full trust, set the `enableNativeCodeExecution` setting to `false`:

```
<WorkerRole name="VerifyOrder" enableNativeCodeExecution="false">
    <ConfigurationSettings>
      <Setting name="InboundQueue"/>
      <Setting name="OutboundQueue"/>
      <Setting name="AccountName" />
      <Setting name="AccountSharedKey" />
      <Setting name="QueueStorageEndpoint" />
      <Setting name="TableStorageEndpoint" />
    </ConfigurationSettings>
  </WorkerRole>
```

Full trust is the doorway to running just about any code you can think of. Another option that will be released soon is the ability to run your own VMs in the Azure data center. There isn't a lot of information about this yet as it was just announced at the PDC 2009, but it promises to let customers run any machine they have today up in the cloud, including any off-the-shelf applications.

That's all you need to know about full trust for now. If trust level is important to your existing code, understand that you have access to the same tools in Azure. Now we're going to look at some scenarios that you might be doing today on-premises. We're going to show how they work the same way in the cloud.

7.2 FastCGI in Windows Azure

FastCGI is a module in IIS 7 that provides a way to run CGI-based applications. CGI is a standard interface that modules can be written to, to plug in to any web server. The web server then pipes each request through these modules, letting each module work on responding to the request. Sometimes a simple static resource module immediately responds with an image file. Other modules execute a whole application to respond with HTML. Modules responding to web requests is how any web server works, especially ASP.NET and PHP. IIS 7 supports this standard with its FastCGI module that's run in the IIS 7 pipeline (when the pipeline is configured correctly).

To take advantage of FastCGI in Azure, you need to configure the FastCGI support for your development workstation, and then configure your web role to enable the PHP interpreter. After you've completed the configuration, your Azure application will be able to host a PHP application and respond to PHP requests from web clients.

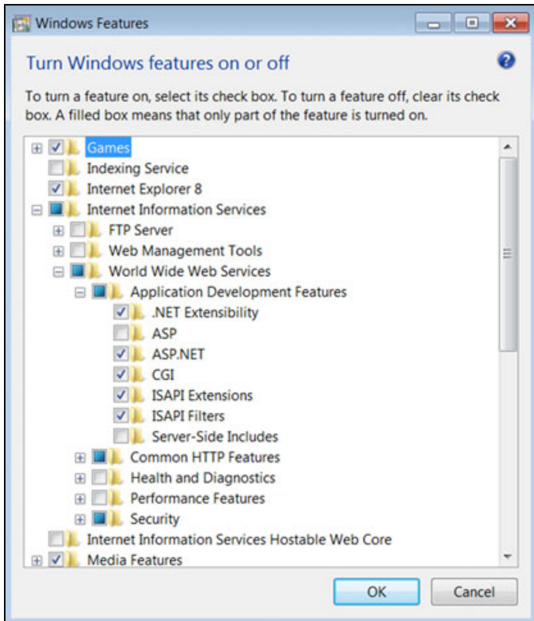


Figure 7.1 If you use Windows 7 for your development workstation, you'll need to enable CGI support for your local instance of IIS. You can do this by using the Windows Features applet in the control panel.

7.2.1 Enabling FastCGI in your local cloud environment

The first step you need to take to run PHP in Azure is a local step. If you want to run and debug PHP running in the local cloud environment, you need to reconfigure the local instance of IIS. The readme for the SDK contains all the gory details for enabling FastCGI. The short steps are to enable the CGI feature of the web server role, if you're running Windows Server 2008. If you're running Vista or Windows 7, you'll need to enable the CGI feature in the Application Development Features group in the Windows Features applet, as shown in figure 7.1.

Easy, right? Now, you've got FastCGI and PHP enabled. Let's configure them.

7.2.2 Configuring Azure for FastCGI and PHP

To run FastCGI in Azure, you need to do more than just tweak one setting in the service definition file, but it isn't too complicated. You need to include a new file in your web role project called web.roleConfig, shown in listing 7.1. The file needs to be in the root of the web project, and needs to be modified from the default contents when you add it. To avoid some of these steps, you can create a CGI web role instead of a normal web role when you create your project. We'll cover that a little later.

Listing 7.1 The web.roleConfig file configures IIS for the use of FastCGI

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.webServer>
    <fastCgi>
```

```
      <application fullPath="%RoleRoot%\approot\myinterpreter.exe"/>
    </fastCgi>
    </system.webServer>
</configuration>
```

After you add the web.roleConfig file, include the CGI interpreter for your platform in the project. If you're upgrading a normal web role project, change the build action of both the interpreter and the web.roleConfig file to `content`. If you're using a CGI Web Role project, this is already done for you. This setting tells Visual Studio to do nothing with the files, but to include them in the output package that is uploaded to Azure. Modify the `fullPath` attribute to point to the interpreter that you include in your project.

Excellent. You've enabled FastCGI, and uploaded and enabled the PHP interpreter that IIS 7 will use to execute your web pages. Now you need to tell IIS 7 what type of requests should be routed to this new interpreter. Should it be every request, or only requests that end in .php? These routing instructions are called *handlers*, and are configured in the plain old web.config file of your web project. The following example shows the handler for PHP:

```
<add name="PHP with FastCGI" path="*.php" verb="*"
➥ modules="FastCgiModule"
➥ scriptProcessor="%RoleRoot%\php\php-cgi.exe"
➥ resourceType="Unspecified" />
```

The handler definition defines a name for the handler and the *path*. The path is what form of requests should be routed to the interpreter that the handler is configuring. In this example, any request to the web server that ends in .php is sent to `FastCgiModule` for processing. `FastCgiModule` then passes the request on to the PHP interpreter for processing.

All file types have a handler in the configuration. Requests for .aspx are executed by ASP.NET with a handler that's configured for you when you install ASP.NET. Requests for static files such as .gif, .jpg, and other simple files are routed to the static file handler. When writing the configuration, keep in mind that you need to use the macro `%RoleRoot%` to point to the root of where your files will be running from. Currently this root is a small drive called E:\ in the Azure server. Don't rely on it to always be E:\; it could change, which is why the `%RoleRoot%` macro is provided.

Add the handler configuration to the `handlers` section of the `system.webServer` part of the web.config file for your web project. Now IIS can set up the handler and start accepting requests.

Let's give IIS some requests to accept by making an application that'll accept and process PHP requests. There are several large applications running on WordPress, on PHP, and on Windows Azure.

7.2.3 Setting up HelloAzureWorld.php

This book isn't about PHP, so our sample is going to focus on simply getting some PHP to work in the cloud. We'll leave what to do with PHP up to you, and perhaps to

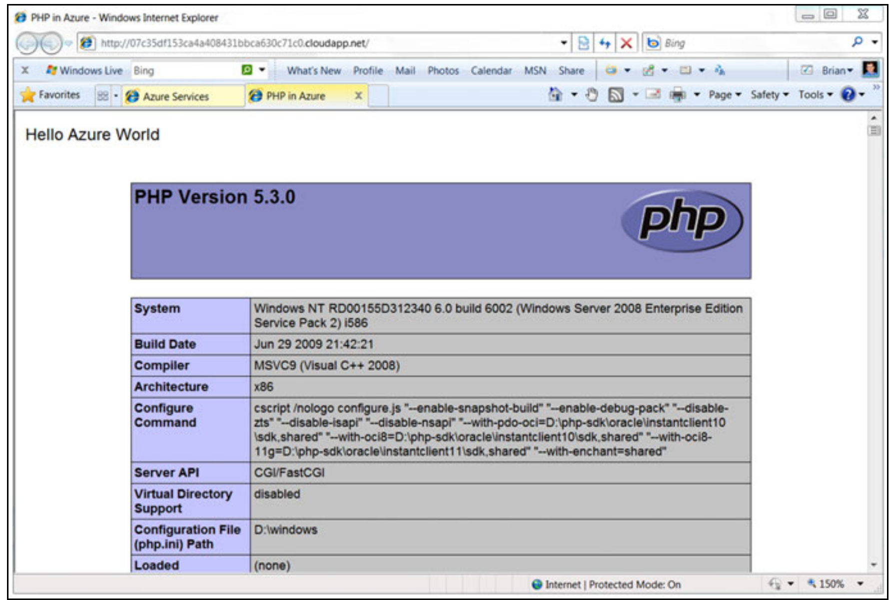


Figure 7.2 Your Hello Azure World PHP web application

another book. For now, you're going to build a simple application that's going to say "Hello Azure World" and display some server information, as shown in figure 7.2.

To build your Hello Azure World PHP application, create a new Cloud Service solution in Visual Studio. Instead of adding a regular web role to the solution, you're going to add a CGI web role. We named ours HelloPHP, as seen in the fabulous figure 7.3. The CGI Web Role template includes the typical changes you'll need to run a CGI-based application in Azure.

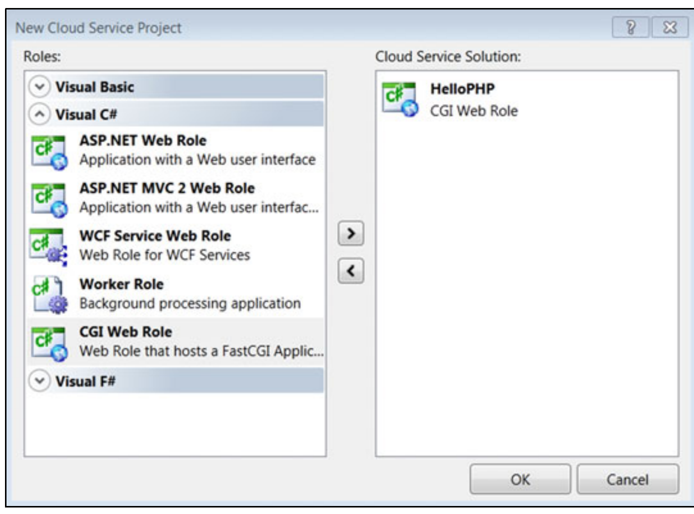


Figure 7.3 Selecting a single web role for your PHP web application. To use FastCGI, choose the CGI Web Role template, which comes prepackaged with the web.roleConfig file you'll need.

The project will already contain a web.roleConfig file, with some notes on how to modify it correctly. The notes remind you to enable native code execution, and to change the path to the interpreter executable (`php-cgi.exe` in this case). You can download the correct version of the PHP interpreter from windows.php.net. (We recommend putting it in a subfolder of the project.) Don't forget to mark any non-.NET files that you add as `content` in the build action for that file. Depending on the version of PHP that you download, you might have to set the time zone in the php.ini file to a valid time zone value, such as `America/New_York`.

The next step is to make some changes to the web.config file. You'll make all these changes in the `system.webServer` section. All lines in this section are used to configure IIS 7, removing the need to use the IIS management tool or to log on to the web server. Not having to use the management tool is great, because now you can put the required web server configuration into the configuration for the web application and it'll travel with your application to whichever web server you deploy it to. You have to configure your application this way in Windows Azure because you can't log in directly to your server.

You need to make two changes to web.config (which are shown in listing 7.2):

- Add a default document section, so that new visitors come to the home page for your web site correctly.
- Add the configuration for the handler, to tell IIS what to do with any web requests that involve PHP.

Listing 7.2 Changing the `web.config` file to run your PHP application

```
<system.webServer>
      <defaultDocument>
        <files>
          <add value="index.php"/>
        </files>
      </defaultDocument>
.
.
.
 <handlers>
.
.
. <add name="FastGGI Handler"
             verb="*"
             path="*.php"
             scriptProcessor="%RoleRoot%\approot\php\php-cgi.exe"
             modules="FastCgiModule"
             resourceType="Unspecified" />
 </handlers>
</sytem.webServer>
```

Your project is set up now, and you can start adding some real PHP code. Add a text file to your web project and name it index.php. Add the following PHP code to index.php:

```
<html>
 <head>
```

```
  <title>PHP in Azure</title>
 </head>
 <body>
  <?php echo '<p>Hello Azure World</p>'; ?> <br />
  <?php phpinfo(); ?>
 </body>
</html>
```

This code does a few basic things. As required by the Union of Demo Code Developers, we have to somehow write to the screen some reference to "Hello World." In this case, you're going to use `echo`, and write out `<p>Hello Azure World</p>.` Lower down in the code, you use the famous `phpinfo()` function. This function writes a bunch of diagnostic information about the version of PHP you're running and information about your web server. You commonly run this function when you first install PHP on a server, to confirm that everything is working correctly. If you deploy your application to the cloud and then browse the results, you'll find some interesting facts about the server running your application: the user name your code is running under is a GUID, and your website files are stored on E:\.

Running PHP in Azure is pretty easy after you set up a few options. That might be why PHP on Azure has proven to be quite popular. Microsoft and WordPress announced at the PDC in 2009 that they're working together, and ICanHazCheeseburger announced that it's been running some of its sites with WordPress running on PHP on Azure.

Now that we have that under our belts, we can start looking at spawning processes in Azure.

7.3 External processes in Windows Azure

Some enterprise applications rely on child processes to be run in an asynchronous way. As a way to show how to work with external processes, you're going to build a small website that converts videos to different video formats. Because we don't want you to write all the video conversion code, you're going to leverage FFmpeg.

FFmpeg is an open source project that provides a cross-platform way to play, convert, and stream media files. You can download it at FFmpeg.org. You'll have to drill around to find the latest build. Also, be aware that it's in a .RAR file, requiring the use of a tool like WinRAR to extract it. For this task, we're using the Windows-compatible binaries. In the package, you'll find an executable called FFmpeg.exe. This is the core executable that you'll be using. Figure 7.4 shows a screen shot of the application you're going to build.

The application will provide a simple way to browse videos in your BLOB account, and to convert one by providing a destination name. The extension of the new filename will tell FFmpeg what format to convert to.

We already have several versions of the Big Buck Bunny trailer uploaded to our BLOB account. Big Buck Bunny is part of the open movie project, and is released under Creative Commons. You can download the whole movie at www.bigbuckbunny.org.

If you enter a destination filename, for example SmallMovie.mpg, the website creates a new background process, executes FFmpeg on the big movie, and then places

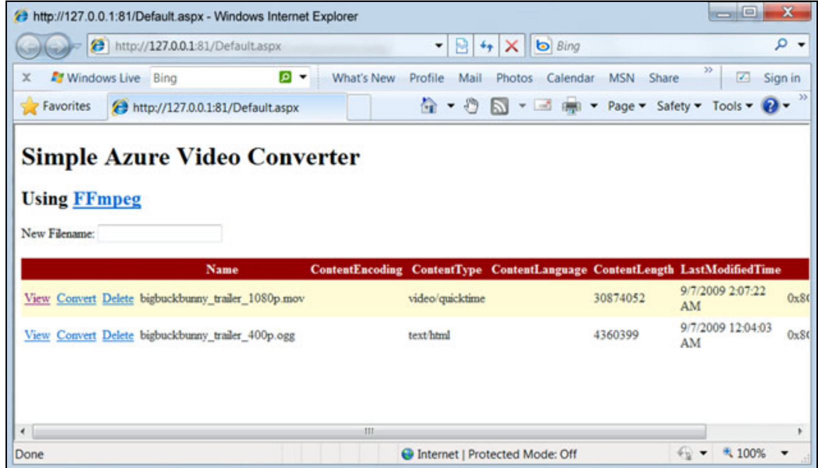


Figure 7.4 A screen shot of the Simple Azure Video Converter, which uses the open source FFmpeg to convert media files. Your web role will call out to the FFmpeg application by launching it as an external process.

the new file (after it's converted into an MPG) back into the BLOB account. Now we're going to tell you how to do that yourself.

7.3.1 Spawning a sample process

You can't run this program in the process that's executing your .NET code, so you'll spawn a new process and have it execute the application for you. The SDK has some great sample code for how to spawn this process and how to interact with it. You're going to use the sample as it is, with some tweaks based on what you're doing. You can find the sample as part of the Full Trust demo, which is in the ExecuteProcess.aspx.cs file. The method is called `Run`.

The `Run` method provides ways to capture any output from the process, so that you can use the output in a way that makes sense for you. In the sample application for this section, we've chosen to capture some basic facts about what's happened and pipe that to the Azure log. Use the following shortened `Run` command; for this project, the command is named `ExecuteCommand`.

```
protected int ExecuteCommand(string cmdPath, string arguments)
{
     var process = new Process();
     var startInfo = process.StartInfo;
     startInfo.UseShellExecute = false;
     startInfo.CreateNoWindow = true;
     startInfo.FileName = Server.MapPath(cmdPath);
     startInfo.Arguments = arguments;
     startInfo.WorkingDirectory =
     ➥ Path.GetDirectoryName(startInfo.FileName);
     startInfo.RedirectStandardError = true;
     startInfo.RedirectStandardOutput = true;
```

```
        process.Start();
        process.BeginErrorReadLine();
        process.BeginOutputReadLine();
        process.WaitForExit();
        var elapasedTime = process.ExitTime - process.StartTime;
        RoleManager.WriteToLog("Information", String.Format("Command: {0}
        ➥ {1}", cmdPath, arguments));
        RoleManager.WriteToLog("Information", String.Format("Exit Code: {0}",
        ➥ process.ExitCode));
        RoleManager.WriteToLog("Information", String.Format("Elapsed Time:
        ➥ {0}", elapasedTime));     return process.ExitCode;
}
```

In this method, you're starting a new process on the operating system and telling it to run your executable. The path to the program to run is set to the `FileName` property, and any command-line arguments you want to pass are set to the `Arguments` property. The process isn't actually started until the `Start` method is called. At the end of your method, you're capturing some runtime information and saving that to the Azure log.

When running in the cloud, you can spawn both 32-bit and 64-bit processes. This makes it easy to move legacy code into the cloud. You can spawn 64-bit processes in the local development fabric as well, unless you're running a 32-bit machine. In that case, you'll only be able to spawn 32-bit processes when you're running locally.

You can start multiple processes in your role, but keep in mind that each new process takes up memory and CPU time. Start too many of them and your instance won't be able to get any real work done.

Like most applications you're likely to spawn, FFmpeg works only on files stored on the local filesystem. To accommodate the application, you're going to copy the movie from BLOB storage to the local storage.

7.3.2 Using BLOB storage

The StorageClient library includes some simple methods to make using BLOB storage easy. You call either the `DownloadToFile` or `UploadFile` method on a `Blob` reference. In the sample application, you'll figure out where the BLOB is and where you want to put it. Use the code in the following listing to download the file, run the conversion code, and then upload the new file back to the BLOB container.

Listing 7.3 Copying a BLOB to local storage and back again

```
public void ConvertVideoFromBlob(string _containerName,
➥ string _inputName, string _outputName)
        {
            string inputName = _inputName.ToLower();
            string outputName = _outputName.ToLower();

            CloudBlobContainer videoContainer =
            ➥ blobClient.GetContainerReference(_containerName);
```

```
        videoContainer.CreateIfNotExist();
        videoContainer.GetBlobReference(inputName)
        ➥ .DownloadToFile(inputName);                          ❶ Downloads BLOB to local file

        ConvertVideo(localDisk.RootPath + inputName,
        ➥ localDisk.RootPath + outputName);                    ❷ Converts video using an external process

        videoContainer.GetBlobReference(outputName)
        ➥ .UploadFile(inputName);                              ❸ Uploads new file to BLOB container

        File.Delete(localDisk.RootPath + inputName);
        File.Delete(localDisk.RootPath + outputName);
    }
```

After the movie is copied down to local storage at ❶, you can run the `FFmpeg` command at ❷. To run `FFmpeg` at the command line, use something like the following:

```
FFmpeg.exe -i BigBuckBunny_Trailer_400p.ogg -y SmallerMovie.mpg
```

To execute this command as a process, call the `ExecuteCommand` method with the following code, which spawns the process, executes FFmpeg, passes in the parameters, and waits for the command to finish executing:

```
string VideoArgs = string.Format(@"-i {0} -y {1}",
➥ localInputFilename, localOutputFilename);
ExecuteCommand(@"ffmpeg\ffmpeg.exe", VideoArgs);
```

No matter which way you run `FFmpeg`, the result is a new movie file, of the proper type, in the same local storage folder where the source movie was.

Your next step is to copy the file back into BLOB storage. Copying the file creates a new BLOB in your videos container that has the same name the one that the user asked for in the web application at ❸. First, you get a BLOB reference with the filename that the user wanted. This BLOB doesn't exist yet, it's just a reference. When you call `UploadFile,` the file is uploaded to BLOB storage. When the file is finally uploaded, you clean up after yourself by deleting both local files.

Spawning processes is an important option to have, but you should use it only when you're migrating an application to Azure that relies on an external dependency. You shouldn't intentionally architect a new system to use this feature. For a new system, you should probably use inter-role communication between different role instances, which gives your solution more flexibility when it's time to scale.

Now you're familiar with two ways to use native code in Azure. Let's look at one more way: calling into a native library. Remember, these are important tools to have, but we wouldn't use them unless we absolutely had to.

7.4 Calling native libraries with P/Invoke

We've looked at two ways to leverage native code in Azure: using FastCGI and spawning processes. The third option at your disposal is to call into a native library with P/Invoke. P/Invoke allows you to directly call a native library, such as a Windows dynamic link

library (DLL). P/Invoke is shorthand for *platform invoke*. You use P/Invoke when you want to call a platform API directly. If you work mostly in .NET, you're rarely calling the platform API directly; instead, you're using classes out of the .NET Framework or the Base Class library.

The limitation with P/Invoke is that you can call only 64-bit native libraries while running in the cloud. If you happen to be developing on a 32-bit machine, you'll be able to call a 32 bit-library locally, but not in the cloud. You can work around this problem by spawning a 32-bit subprocess, as outlined in section 7.3.1, and calling P/Invoke from there. We think this is too much work though; you should stick with 64-bit libraries.

Calling native libraries is a special skill to begin with, so we aren't going to cover everything you need to know about doing that in .NET. Calling them in the cloud is a lot like when you do it locally.

When working with native libraries, you need to first import the DLL into your namespace, and then provide a façade method into the native libraries method. This will involve a great deal of code that will map native data types to .NET CLR types.

Don't forget to allow native code execution in your cloud service definition file; otherwise, you'll receive a security exception.

The easiest way to implement the signatures you need in your code is to get them from http://www.pinvoke.net, like we do. This website provides the signatures for just about any native call you could possibly want to make.

Now let's get to it and use P/Invoke to call a native library. Although you're free to follow along with how we do this, you might consider skipping to the sample code for the book. It'll be easier to see how it all fits together.

7.4.1 Getting started

You're going to build a simple web application for Azure that will list the files and directories in a given folder. You could easily build this with managed code, without P/Invoke, but we thought it would make for a good example.

The first step is to add the interoperability services namespace to your project:

```
using System.Runtime.InteropServices;
```

Then you import the native methods that you want to call:

```
[DllImport("kernel32", CharSet = CharSet.Unicode)]
public static extern IntPtr FindFirstFile(string lpFileName,
➥ out WIN32_FIND_DATA lpFindFileData);
```

In this example code, you're importing a method called `FindFirstFile` from `kernel32`, which is called by the DLL. This line of code defines a series of parameters you'll need to provide to the method for it to work. The first parameter, `lpFileName`, is the path of the folder you want to look in. The second parameter, `lpFindFileData`, is a variable that the results will be stored in. The `FindFirstFile` method also returns a handle to an object that will be your pointer into the filesystem. You'll use this handle to iterate through the folder you're pointing at by calling another imported method, `FindNextFile`. Always remember to close any handle objects you're using

while working with native libraries; otherwise, they'll be left open in memory, causing memory leaks.

It'll often be the case that the method you're importing requires input and output parameters, and return values that use a type that isn't supported in .NET. You'll need to define these types so that they can be used. The following listing shows an example structure that defines the data about the file that the finder just found.

Listing 7.4 Defining a data type to work with a native library

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
        public struct WIN32_FIND_DATA
        {
            public FileAttributes dwFileAttributes;
            public FILETIME ftCreationTime;
            public FILETIME ftLastAccessTime;
            public FILETIME ftLastWriteTime;
            public int nFileSizeHigh;
            public int nFileSizeLow;
            public int dwReserved0;
            public int dwReserved1;
            [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
            public string cFileName;
            [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
            public string cAlternate;
        }
```

After you define these types, you'll be able to call into the method just fine.

7.4.2 Calling into the method

In this example, you're going to make the initial `FindFirstFile` call, and then perform a `Do` loop until `FindNextFile` comes back empty. With each iteration of the loop, you'll copy data from the return data structure to a class, `myFileData`, which you also have to define. Then you'll bind a collection of `myFileData` objects to a simple `GridView` on the web form. All this is shown in listing 7.5.

Listing 7.5 Processing each file that's found

```
hFileFinder = FindFirstFile(currentPath, out foundFile);
   do
   {
      fileList.Add(new myFileData()
      { Filename = foundFile.cFileName,
        Filesize = foundFile.nFileSizeLow,
        isDirectory = (foundFile.dwFileAttributes &
        ➥ FileAttributes.Directory) != 0 });
   }
            while (FindNextFile(hFileFinder, out foundFile));

         FindClose(hFileFinder);

         gvFileList.DataSource = fileList;
         gvFileList.DataBind();
```

Working with native libraries and P/Invoke can be complicated. You should look to using this way to use native code only when you can't possibly do what you need to do with a class in the .NET library. Using native libraries can introduce brittleness into your solution by creating external dependencies.

7.5 Summary

Microsoft has provided broad support for running just about anything on Azure. It didn't take the easy way and limit cloud developers to just .NET code because Microsoft wants Azure to be a usable platform with broad adoption. The ability to run FastCGI, spawn processes, and call native libraries makes it easier for you to port existing applications and to support a broader array of applications.

With Azure's support for FastCGI you can leverage any CGI-compatible module in Azure. These modules include PHP, Ruby, and many other web platforms. With just a few simple steps, you can deploy one of these platforms to your web roles in Azure.

Spawning external processes is important when you have an external dependency in a system you might be migrating to the cloud. You can also spawn them to manually parallelize your application.

Some applications that are built with unmanaged code need to access native libraries for system-level access. Using P/Invoke to access these libraries is available in Azure, but it can be complicated. You should leverage this feature only if you truly need it.

Azure isn't just for Web 2.0 web applications with a viral nature; it's also for serious enterprise applications. Those applications often come with a legacy aspect, whether it's calling into a home-grown DLL that can model and calculate the air speed of an unladen swallow, or being able to leverage a forum for your website that happens to run in PHP.

What we've covered in this chapter, especially native calls and process spawning, are great tools with great power. And with great power comes great responsibility. Make sure you use them wisely, or you'll spend a lot of your weekends figuring out why your application isn't working like you want it to.

In our next chapter, we'll start the conversation about how to store files in the cloud using BLOBs. Don't be scared; we know you've probably had a bad experience with BLOBs and traditional databases. BLOBs in Windows Azure aren't nearly as complex.

Part 4

Working with BLOB storage

Part 4 explores BLOB storage, a simple file storage system for the cloud. Many people call file storage *unstructured storage*, and if you saw our desktops, you would know why people call it that. You wouldn't believe how hard it was in these chapters to avoid cheesy, 1950s sci-fi references to BLOB monsters and the like.

Chapter 8 covers BLOB basics: what they are and why you might use them. Chapter 9 shows you how to work with BLOBs inside your applications, and chapter 10 shows you how to use BLOBs from outside Azure.

What? Yes, outside Azure. Hey, by now you should know the cloud isn't all or nothing; the most common use of Azure will likely be of a hybrid nature.

8 The basics of BLOBs

This chapter covers

- How files are currently shared in retro systems
- How Windows Azure allows us to store files (woo hoo, go Azure)
- How to consume the BLOB storage service

In case you didn't bother reading the blurb at the beginning of part 3, in this chapter (and the next couple of chapters), we'll be looking at how you can store files in Windows Azure's highly scalable, fault tolerant, binary-file storage system (otherwise known as the BLOB storage service).

> **DEFINITION** BLOB stands for *binary large object*. The term has been stolen from the world of relational databases where it used to describe the storage of binary data (such as an image or an MP3 file) in a single entity. We wish they'd used BinLob as the acronym. It more accurately describes what happens when a DBA discovers you stuck a terabyte of data in a single row of his database.

In this chapter, we're going to answer the following questions:

- Why is storing files in a typical scaled-out system so hard?
- How does the BLOB storage service address typical scaling issues?

- How does the BLOB storage service work?
- How can you can get your tools out and start developing against it?
- How do you store BLOBs in the production system?

Before you can appreciate the beauty of the BLOB service, you need to get a little insight into how you might solve the problem of storing files that can be accessed by multiple servers in a scalable fashion.

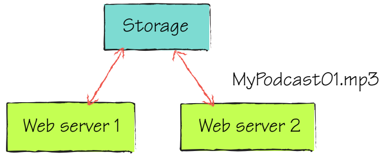
8.1 Storing files in a scaled-out fashion is a pain in the NAS

Unless you have plenty of cash, you're going to experience some pain if you try to share files across machines. No matter what hat one of us puts on (author, presenter, architect, developer, or computer scientist), we're embarrassed by the following statement: sharing files across machines is incredibly hard. It is; it shouldn't be, but it is. Decoding the genome and making robots climb stairs, that should be hard, but sharing files shouldn't be.

> **BLOB content starts in section 8.1.2**
>
> In this section, we'll be looking at the challenges of storing files in a scalable fashion. If you've had too much coffee and just can't wait to get to some BLOB content, feel free to skip along to section 8.1.2.

To contextualize the problem, let's return to the podcast example that we introduced in chapter 1. In that scenario, we wanted to provide a service where users could upload podcasts that would be converted from MP3 to WMA. To support the predicted demand, we decided to load balance the website across two servers. Because users can upload or download a podcast from any server, a shared storage solution is required.

Figure 8.1 shows a logical representation of two load balanced web servers accessing a podcast from a shared storage mechanism.



To be honest, you don't need to be the greatest architect in the world to draw the solution shown in figure 8.1. It's pretty logical, common sense stuff. Two web servers access a common storage area.

Now you're thinking, "Why did they just say it's common sense, when before they said it was hard? Get me another book that says it's easy." Well, before you start reaching for *Mavis Beacon Teaches Windows Azure*, check out the following questions. As you think about the possible answers, you might begin to see why this is a little harder than it seems to be at first.

- Do you have enough space to store all the files you need?
- How do you add more storage capacity?

- If a disk crashes, where does your data go?
- Is the storage block load balanced?
- What if you lose your connection to the block? Is it redundant?
- At what point do you max out your disk, in terms of reading and writing?
- How do you evenly distribute load across all disks?

The good news is that pretty much all of these problems have been answered and solved already. You can even implement these solutions in your traditional noncloud environments today (well, the lead time is probably longer than a day). The bad news is that the cheap, simple solutions are typically not scalable or fault tolerant. The solutions that are scalable and durable are usually expensive. In the Windows Azure BLOB storage service, all that changes.

Before we look at how easy it is to store and access files (in a scalable, durable fashion) across multiple servers in Windows Azure, let's look at some of the options outside Windows Azure.

8.1.1 Traditional approaches to BLOB management

Over the next few sections we'll look at how you might provide a file storage facility in traditional ASP.NET web server farms, using our podcasting example. We'll specifically look at using the following storage options:

- SQL Server
- Network share
- Distributed File System (DFS)
- Network-attached storage (NAS)
- Direct-attached storage (DAS)
- Storage area network (SAN)

Let's start with one with the typical developer solutions to the problem: the database.

SQL SERVER

Because web servers typically have access to a shared SQL Server database, you could store your podcasts in a table. Although this is a common approach used in many solutions, it's probably not the best use of your expensive database server. It's like racing a truck in a Grand Prix; there are cheaper, simpler, higher performing, and more appropriate solutions for storing files.

Unless you're using a high-availability technology (such as clustering, mirroring, or replication), your database server is likely to be a single point of failure in the system. In figure 8.1, SQL Server would be represented by the Storage block (accessed over a typical network connection).

NETWORK SHARE

Another common approach to providing a shared filesystem across web servers is to use a shared network drive that can be accessed by all instances of the website. This low-cost solution is more lightweight than a database, but it still introduces a single

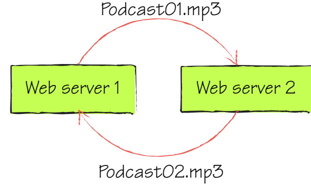
point of failure. This cheapo solution offers no redundancy and provides no ability to scale out. In figure 8.1, an application server with a network share would also be represented by the storage block.

Now that we've looked at some of the lower-end solutions, let's take a look at some of the typical high-scale solutions that are used, starting with Distributed File Systems.

DISTRIBUTED FILE SYSTEM (DFS)

Windows Server 2003/2008 provides a technology known as DFS that allows you to create a peer-to-peer (P2P) filesystem on your network. UNIX/Linux environments have similar tools. If you use DFS to store podcasts, when a new podcast is uploaded, a copy of the file is replicated to all other participating servers. Although this approach requires no new hardware, it's complicated to manage and adds extra performance overhead to all servers involved.

Figure 8.2 shows a DFS solution with a P2P network between two web servers.



Whenever a file is uploaded to a web server, it's automatically replicated to all other servers in the farms. Using replication ensures that there are no single points of failure in this solution and that the data is held on multiple machines. In figure 8.2, Podcast01.mp3 is uploaded to web server 1 and then replicated to web server 2; when Podcast02.mp3 is uploaded to web server 2, it's then replicated to web server 1.

In figure 8.3, the web servers don't hold the files locally, but use a replicated file store held in application servers. In this figure, Podcast01.mp3 was uploaded to app server 1 via web server 1. The file was replicated to app server 2, and then served up to the client from app server 2 via web server 1.

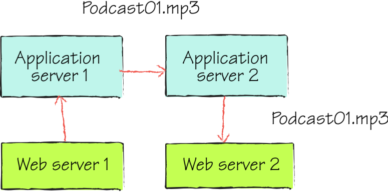


Figure 8.3 Two web servers reading and writing files from a set of replicated file servers

With file replication, any time a file is uploaded to a server there's a small delay between the file being uploaded and it being replicated across all servers. It's therefore possible that the web user could be load balanced onto a server where the file isn't available (because it hasn't been replicated across to that server yet). Although this issue can be alleviated by using sticky sessions, sticky sessions won't help if the original server keels over. Also, using sticky sessions means that incoming requests won't be evenly distributed across all web servers.

Now that we've looked at some of the hook-some-machines-together solutions, we'll look at some of the dedicated disk array–type solutions that are typically used in the market.

> **Sticky sessions**
>
> A sticky session occurs when a load balancer forwards all incoming requests from the same client to the same server for the period of the session.

NETWORK-ATTACHED STORAGE (NAS)

A *network-attached storage* device is a disk array that you can plug into your network and that can be accessed via a network share. NAS devices are responsible for managing the device hardware, the filesystem, and serving files, and can provide varying levels of redundancy, depending on the device and the number of disks in the array.

Although NAS devices reduce load from client operating systems by taking responsibility for file management, they can't scale beyond their own hardware. NAS devices can range from being pretty cheap to very expensive, depending on the levels of scalability, performance, and redundancy that you require from the device. In figure 8.1, the NAS device would be represented by the storage block (connected via the Ethernet).

NAS devices are used to provide capabilities similar to those of a file server, rather than being used as a disk management system in a high-performance application solution.

DIRECT-ATTACHED STORAGE (DAS)

A *direct-attached storage* device is a disk array that you can plug directly into the back of your server and that can be accessed natively by the server. DAS devices are responsible for managing the device hardware and can provide varying levels of redundancy, depending on the device and the number of disks in the array.

Because DAS devices are directly connected to a server, they're treated like a local disk; the server is responsible for the management of the filesystem. DAS devices can support large amounts of data (100 TB or so), can be clustered (there's no single point of failure), and are usually high-performance systems. As such, DAS devices are a common choice for high-performance applications. The cost of the device can range from being pretty cheap to very expensive, depending on the levels of scalability, performance, and redundancy that you require.

Although DAS devices are great, they're limited by the physical hardware. When you reach the physical limits of the hardware (which is quite substantial), you'll be able to scale no further.

In figure 8.1 the DAS device would be represented by the storage block, connected directly to the servers.

STORAGE AREA NETWORK (SAN)

Like DAS devices, SANs are also separate hardware disk arrays; they don't have their own operating system, so file management is performed by the client operating system.

SAN devices are represented on the client operating system as virtual local hard disks that are accessed over a fiber channel. Because you need your web servers to

access shared data, the SAN would need to support a shared filesystem. In figure 8.1, the SAN device would be the storage block, attached to the web servers via fiber channels.

SANs are usually quite expensive, require specialized knowledge, and are rarely used outside the enterprise domain. To give you a clue about how expensive they are, Dell doesn't even list the price on its website. As for installing and managing SANs, that's purely in the domain of the long-haired sandal-wearing bearded types. We mere mortals have no chance of making those things work. SAN devices support replication and are highly scalable (they scale much higher than do DAS devices), fault tolerant, high performing, and incredibly expensive. Due to their performance, price, and scalability, this is the solution of choice in the enterprise space. The rest of us can only dream.

Hopefully we've justified our earlier premise that implementing a file storage solution today isn't as easy as it first looks. All the available choices (beyond a certain size) require extensive IT knowledge, skills, and management, not to mention large amounts of cash or a tradeoff between capacity, redundancy, ability to scale, or performance.

This is the state of affairs with regard to the issues with storing files in traditional on-premises solutions. Let's now look at the Windows Azure BLOB storage service and how it tackles these issues.

8.1.2 The BLOB service approach to file management

As we discovered earlier, the BLOB storage service is the Windows Azure solution to providing file storage. Let's take a look at how Azure implements this service.

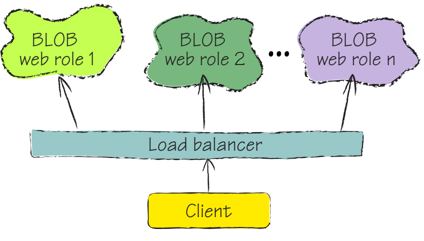
AN API-BASED SERVICE

Rather than building a native network-share-based solution, Microsoft has provided a set of REST-based APIs that allow you to interact with all the storage services over the HTTP stack, using a standard HTTP request. As mentioned earlier, not only can you use these APIs inside the data center, but you can also use them outside the data center.

> **NOTE** Although you can upload and download files outside the data center, you'll be subject to internet speed; it might take you a few hours to upload or download gigabytes of data. Within the data center, you can copy gigabytes of data between BLOB storage and a worker or web role in seconds. This massive speed difference is the result of the co-location of the storage service and the roles.

SCALABILITY

Using HTTP as the underlying transport layer means that Windows Azure can leverage the web role infrastructure inside Windows Azure to host the storage services. By using the web role infrastructure to host the Windows Azure storage service (with tens of thousands of instances), you can be confident that your application will be able scale to that level. Figure 8.4 shows the abstraction of web instances for the BLOB storage service.



Because BLOB storage is built on the web role infrastructure, web roles can also harness the advantages of utility computing. As the demand for the storage services increases, Microsoft can ramp up the number of instances just like it can for any other web role. You don't need to worry about the scalability of any of the storage services (unless Microsoft runs out of pennies).

DISK STORAGE

Just as there are thousands of racks of machines used to host the web and worker roles, there are just as many disk arrays storing your data! Microsoft can grow the storage required in the data center by adding more disks as and when required. This level of enterprise-class storage means that you never need to worry about capacity or scale. Think of the BLOB service as a giant virtual hard disk that will always scale up to meet your demands and never run out of space.

DATA CONSISTENCY WITH REPLICATION

Like the DFS solution, Windows Azure BLOB storage is also a replicated solution (to be honest, you have to be to achieve such massive scale). Although the BLOB service is quite similar to the Amazon Simple Storage System (Amazon S3), replication is one of the areas in which it differs.

With Amazon S3, there's no consistency of data throughout the data center. If you upload a file to Amazon S3 and then request that same file, it's likely that a different server will process that request. As a result of network latency, the file probably won't be available to the new server because the data won't have been replicated from the original server yet. Amazon S3 suffers from the same issues seen with DFS.

This issue of replication latency can never occur in Windows Azure storage services. Windows Azure guarantees a consistent view of your data across all instances that might serve your requests. Internally, inside the Windows Azure storage services, data is replicated throughout the data center as soon as it's written to your storage account. Every piece of data must be replicated at least three times as part of the commit process.

As your data is being replicated across the various disks in Windows Azure, the FC keeps track of which instances can access the latest version of your data. The load balancer will route requests only to an instance that can access the latest version, ensuring that stale data is never served.

Even if a disk failure occurs immediately after the upload, there won't be any data loss; other disks are guaranteed to receive a copy of that data.

So far we've talked about how BLOB storage solves the problems of scalability and fault tolerance, but we haven't talked about performance. Surely performance is going to suffer; it's effectively a REST-based web service, after all.

PERFORMANCE

Sure, the performance of BLOB storage in comparison to SANs or DASs isn't all that great. Ultimately that tradeoff between performance, fault tolerance, and scalability means that performance is lost. However, within the data center, it's generally good enough performance. Because the service is ultimately a load balanced web server, you can expect 50 milliseconds to 100 milliseconds of latency between your role and the storage service. Although the latency is poor, the network connection is fast, so you can expect good enough performance. Sure, you wouldn't allow an application that needs to write to disk very quickly (for example, SQL Server) to write directly to BLOB storage, but not all applications need that kind of speed.

If you do need that level of speed, you can always cache files locally on your role using local storage. This technique will usually give you more acceptable performance for your application. In fact, this is exactly what the Azure Drive (originally called X-Drive) feature uses to ensure performance.

> **What's Azure Drive?**
>
> Although the REST API is flexible and provides great scale, it's no substitute for a good old filesystem. To make life a little easier for those bits of code that are used to talk to directories and files rather than to a web service, Microsoft has provided a new feature called Azure Drive. Azure Drive allows you to mount BLOB storage as a New Technology File System (NTFS) drive, which lets you access BLOB storage just like any other drive. Because this feature is implemented using a special OS driver that was developed specifically for Windows Azure, this feature is only available to your roles; it's not available outside the data center.
>
> As cool as Azure Drive is, it allows only one instance of a role to read and write to the Azure Drive. Multiple role instances can mount the same Azure Drive, but only in a read-only mode, and only against a snapshot of the drive itself.

Now that we've looked at how BLOB storage handles the issues that arise in traditional on-premises solutions, it's worth looking at BLOB storage from a data management perspective.

MANAGEMENT

One of the most compelling arguments for using the Windows Azure storage services is that IT professional management skills aren't required. In traditional systems, a large investment in IT management skills is usually needed to support storage. Management of the storage arrays usually requires expensive specialists who are capable of supporting the data, such as SAN experts, network specialists, technicians, administrators, and DBAs.

To plan such a system, these experts need to be able to design and implement the infrastructure, taking disk management, fault tolerance, networking, lights-out operation, and data distribution into consideration. The day-to-day running of the system

includes hardware replacement, managing backups, optimizing infrastructure, health monitoring, and data cleansing, among other endless tasks.

With Windows Azure, you can let Microsoft manage the storage systems and concentrate on using the system via familiar developer APIs. You can focus on your core skill set, which is building software.

8.2 A closer look at the BLOB storage service

You have an idea how the BLOB storage service is hosted in Windows Azure. Let's look at how files are stored in the service. In this section, we'll look at the three layers of BLOB storage:

- The account
- The container
- The BLOB

To help explain these concepts, we'll use figure 8.5 as a reference. Figure 8.5 shows how an MP3 file might be stored in BLOB storage.

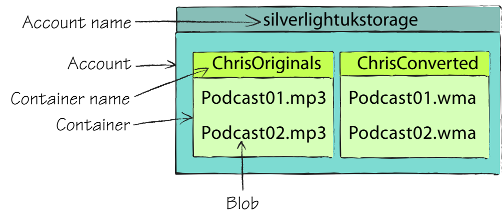


Figure 8.5 Podcast01.mp3 is stored in the ChrisOriginals container in the silverlightukstorage account

Before we get all technical about accounts, containers, and BLOBs, keep this in mind: an account is simply your account. Dave has an account, Jim has an account, and you have an account. An account is about ownership. A container is somewhere you can store your BLOBs. Containers are about access control (public or private access) and some level of organization.

With that in mind, let's look at some of the specifics.

8.2.1 Accessing the BLOB (file)

In figure 8.5, you can see how files (otherwise known as BLOBs) are stored in BLOB storage. The BLOB Podcast01.wma resides in the container ChrisConverted, which resides in the storage account silverlightukstorage. A BLOB can't directly reside in a storage *account* and must live in a storage *container*. If you do need to make the BLOB available as if it's at the top level of the account (as if it doesn't have a container), you can store the BLOB in the root container; we'll explain this in more detail in chapter 10.

Because storage services use a REST-based architecture, you can retrieve a file from BLOB storage by performing an HTTP `GET` request to the URI for the BLOB. The following URI would let you retrieve Podcast01.wma from the ChrisConverted container (held in the silverlightukstorage storage account) from the live BLOB storage service: http://silverlightukstorage.blob.core.windows.net/ChrisConverted/Podcast01.wma.

We could formalize the URI for the live storage account as follows: http://<*storageAccount*>.blob.core.windows.net/<*Container*>/<*BlobName*>.

Let's now take a closer look at accounts, containers, and BLOBs to get a clearer understanding of these components.

8.2.2 Setting up a storage account

When you sign up for Windows Azure, you can create a storage account in the Azure portal. The storage account is the top level for all storage services (BLOBs, queues, and tables) that reside under it.

When you create your storage account, you'll be assigned a subdomain for each storage service. The following three domains are for the storage services:

- http://<*storageAccountName*>.blob.core.windows.net/
- http://<*storageAccountName*>.queue.core.windows.net/
- http://<*storageAccountName*>.table.core.windows.net/

In our previous example, the name of the storage account was silverlightukstorage, which means that the top-level URI for each service in our account would be as follows:

- http://silverlightukstorage.blob.core.windows.net/
- http://silverlightukstorage.queue.core.windows.net/
- http://silverlightukstorage.table.core.windows.net/

For now we're going to focus on the BLOB service, but in later chapters we'll return to the Table service and the Message Queue service.

> **How do you break up your storage accounts?**
>
> There are a couple of things to consider about your storage account, the major one being this: do you have one large account, or a separate account for each application? Although this is ultimately up to you, a good guide would be access control. If you're a small shop, then one overall account is probably suitable; however, a single account wouldn't work so well in, say, Microsoft or IBM. In these larger organizations, separating by application is probably a more suitable approach.

If you don't like the beautiful subdomain assigned to you for BLOB storage (*xxxxx*.blob.core.windows.net) then you can always assign your own domain name.

8.2.3 Registering custom domain names

What we'll do now is step through the process of associating your own domain name with the BLOB storage service. You'll be able to access your WMA file using this URI: http://blobs.chrishayuk.com/ChrisConverted/Podcast01.wma.

To register a custom domain name with a BLOB storage account, you have to do the following:

1. Register a suitable domain with your domain provider.
2. Set up a domain to point at Windows Azure.
3. Validate that you own the domain.
4. Set up the subdomain to point at BLOB storage.

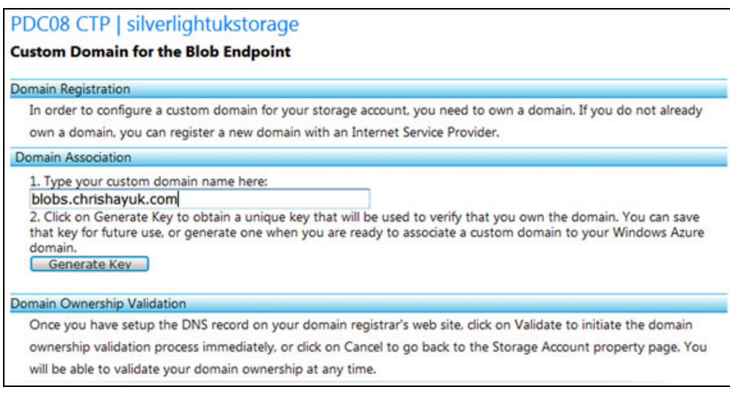


Figure 8.6 Validating in the Azure portal that you're the owner of the domain that you want to point to the BLOB storage account

We're going to skip the registering a suitable domain step. If you don't know how to do that, then I'm sure GoDaddy (or some other provider) will happily provide some instructions so they can extract some lovely dollar bills (or British Pounds, or Euro Euros) from your pocket.

SET UP A SUITABLE DOMAIN

After you've registered your domain (for example, chrishayuk.com), you need to let Windows Azure know that you want to point a suitable subdomain at it. To do that, log in to the Azure portal. Select your storage account (silverlightukstorage), and then click the Manage Domains button. You'll be faced with the page shown in figure 8.6.

After you've entered the name of the domain (including the subdomain) that you want to point to the BLOB storage account, you need to validate the domain.

VALIDATING THAT YOU OWN THE DOMAIN

Validate the domain by clicking the Generate Key button. After you click the button, you'll be presented with the screen shown in figure 8.7.

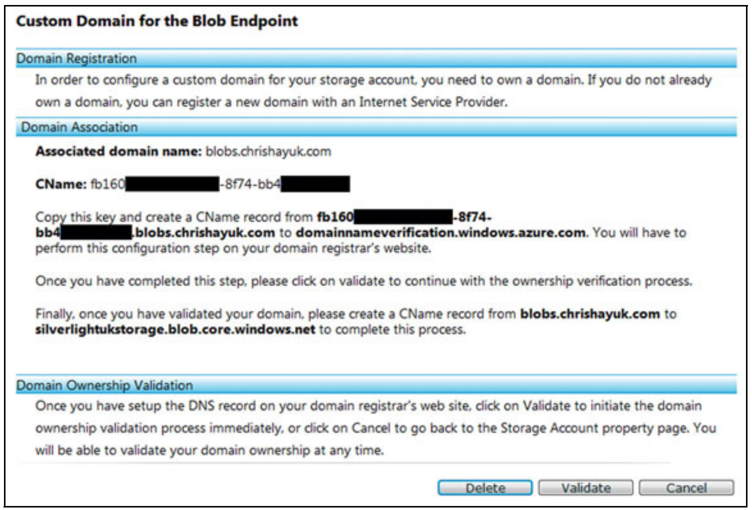


Figure 8.7 Receiving the domain validation CNAME GUID

The window in figure 8.7 indicates that you need to perform two actions:

- Add a new CNAME for the GUID (`fb160. . .`) that points to verify.windowsazure.com.
- Add a new CNAME for the subdomain (blobs.chrishayuk.com) that points to your BLOB storage account (silverlightukstorage.blob.core.windows.net).

Whichever company you used to register your domain probably manages the DNS for your domain name. Using their web control panel, you should be able to create the subdomain using a CNAME. Figure 8.8 shows the CNAMEs for chrishayuk.com in the GoDaddy Domain Manager.

If you manage your own DNS server, you already know how to set up a CNAME; if not, your system administrator will certainly be able to (although he might not be very pleased that you're looking to replace him with an automated system).

After you've set up your CNAMEs, return to the Windows Azure portal a little later to validate the domain (click the Validate button shown in figure 8.7). As soon as the domain has been validated, you'll be able to use your custom domain name.

Why do you need to come back later? Funnily enough, this is all to do with replication. After you've updated the DNS details on the server that's responsible for maintaining your domain records, this update needs to be replicated to all the other DNS servers in the world. This replication delay is the reason that you'll have to come back later (usually 10 minutes to an hour); it'll take a little time for the Windows Azure DNS servers to receive that update. Perhaps the world's DNS servers should use Windows Azure instead.

OK, you've got your custom domains set up and you understand containers; let's look at how you can use them to store BLOBs.

8.2.4 Using containers to store BLOBs

In BLOB storage, you can't store BLOBs directly in a storage account because every BLOB must live in a container. A container is really a top-level folder. Although you can set permissions directly on a BLOB, this can be a pain with a large number of BLOBs. To alleviate that administrative headache, you might want to group similar BLOBs that

CNAMES (Aliases) | Reset to Default Settings | Add New CNAME Record

Host	Points To	TTL	Actions
blog	ghs.google.com	1 Hour	
fb160c4b-d159-41ba-8f74-bb40ce4b65d2	domainnameverification.windows.azure.com	1 Hour	
blobs.chrishayuk.com	silverlightukstorage.blob.core.windows.net	1 Hour	
www	@	1 Hour	

Figure 8.8 The CNAME entries for chrishayuk.com; notice that both the domain verification CNAME and the BLOB storage CNAME are listed

have similar access levels in the same container. Then you can set permissions at the container level rather than at the individual BLOB level.

In BLOB storage, there are two levels of access that you can set on a container: private and public.

PRIVATE CONTAINERS

BLOBs in a private container are restricted to the owner of the account. If you need to list the contents or download a BLOB stored in a private container, you need to make a request signed with your shared authentication key (in the next chapter we'll show you how to do this).

In figure 8.5, the container ChrisOriginals is a private container. If you wanted to access the BLOB podcast01.mp3, you would make a GET request to the following URI (this request must be signed with either your account master key or a pregenerated shared key; we'll explain this later): http://silverlightukstorage.blob.core.windows.net/ChrisOriginals/Podcast01.mp3.

FULL PUBLIC READ ACCESS AND PUBLIC READ-ONLY ACCESS FOR BLOBS

If the container is set to full public read access, then you can retrieve any BLOB held in the container over HTTP without providing authentication credentials. Not only that, you can list all the BLOBs in that container and query data about the container.

With public read-only access for BLOBs, anonymous requests will only be able to read a BLOB (you won't be able to read container data or list the BLOBs in the container).

In figure 8.8, the container ChrisConverted is a public container; anyone on the internet would be able to download the file podcast01.wma by making an HTTP `GET` request to http://silverlightukstorage.blob.core.windows.net/ChrisConverted/Podcast01.wma.

If you need to perform any operations beyond the container permission level (for example, if you need to upload or modify a BLOB), you need to provide authentication credentials (account owner or shared access) because these operations are restricted operations.

So far we've talked only about the live BLOB storage service. Now we'll take some time to look at how you can develop against the BLOB storage service by using a development version of the BLOB service that's in the development storage service.

8.3 Getting started with development storage

Development storage hosts all three storage services (BLOB, Queue, and Table storage services) and exposes local endpoints that implement the same APIs as the live service. The production version of the storage services and the development version are two completely different animals. They might expose the same APIs, but the development version is greatly simplified and suitable only for local development.

When you've finished developing your application against your local development storage, you can easily switch to using the live environment by just changing configuration.

8.3.1 SQL Server backing store

Because the development environments and the data centers of Windows Azure are drastically different (we don't have replicated storage arrays on laptops), the SDK can provide only a simulation of the live storage environment. Although development storage and BLOB storage are API–compatible, the underlying implementations are understandably different.

In the development storage version of BLOB storage, SQL Server is used as the backing store.

> **Installation issues**
>
> By default, the development storage database is created in the SQLEXPRESS named instance of SQL Server on your development machine. This instance is normally installed as part of the Visual Studio installation, which is why the SDK assumes that this instance is present. If you need or want to install the database onto a different SQL Server instance, you can use a tool in the SDK called DSInit.exe. You might want to do this if you prefer to run a full-blown version of SQL Server on your machine or if you skipped installing the SQLEXPRESS instance during the Visual Studio installation.

If you want, you can even run queries against the database to ensure that your data is stored as you expected. Figure 8.9 shows all the tables representing the various storage services in the SQL Server implementation.

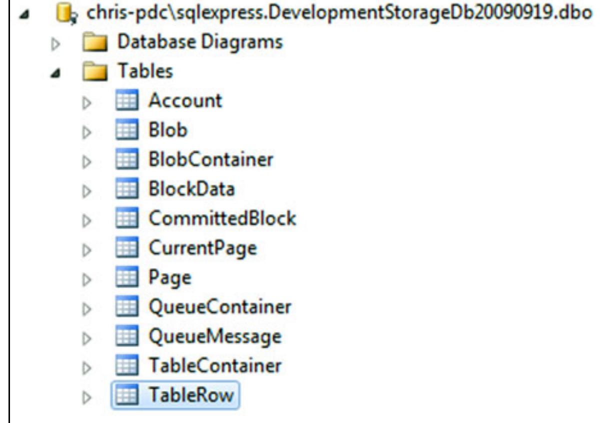


Figure 8.9 Development storage database in SQL Server

Although the development storage system uses SQL Server (as shown in figure 8.9), the real BLOB storage system uses a higher performing, more scalable, custom solution that makes the best use of the Windows Azure infrastructure. You can be assured that your BLOBs aren't stored in some SQL Server table in the live system.

8.3.2 Getting around in the development storage UI

The development storage service is automatically started whenever you run a web or worker role project in Visual Studio. The startup of development storage occurs at the same time as the startup of the development fabric, as described in chapter 2. If you

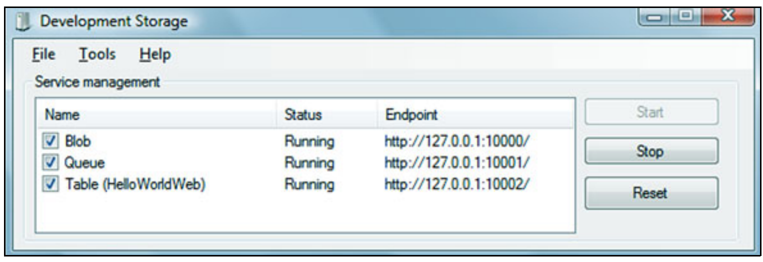


Figure 8.10 Development storage command and control center. You can start, stop, and mess around with all your storage services from here.

right-click the Cloud Services icon in the status bar and select Show Development Storage UI, the Development Storage UI is displayed, as shown in figure 8.10.

The development storage UI shows you the current status of your services and lets you stop and start them if you need to. Although development storage and the development storage UI are automatically launched when you run your application in Visual Studio, you can start them manually using the command line. This can be useful if you're interacting with the storage services from an application that's not hosted in the cloud (a normal WPF application that just uses the BLOB storage service).

Starting and shutting down development storage manually

To start development storage manually, you can use the following command:

```
C:\Program Files\Windows Azure SDK\v1.1\bin\devstore\dsservice.exe
```

To shut down the service, you can use this command:

```
C:\Program Files\Windows Azure SDK\v1.1\bin\devstore\dsservice.exe/shutdown
```

With the basics of both BLOB storage and development storage under your belt, get ready! It's time to write your first application that talks to the BLOB service.

8.4 Developing against containers

Before we start writing some code against the containers, we should probably discuss where this type of functionality is useful.

If you just need a shared storage area where you can read BLOBs, you'll probably eventually use the Azure Drive functionality (discussed in chapter 10) rather than interacting with the BLOB storage APIs directly. Even so, it's still useful to understand how the BLOB storage APIs work, because Azure Drive interacts with these APIs too.

If you need a scalable application in which more than one role needs to write to a single shared storage area, then you'll need to use the StorageClient library or the REST APIs directly.

Over the next few sections, we'll be looking at the kinds of operations you can perform against containers. Using containers is particularly interesting when you need to dynamically create storage areas and assign permissions to different parties in a scalable

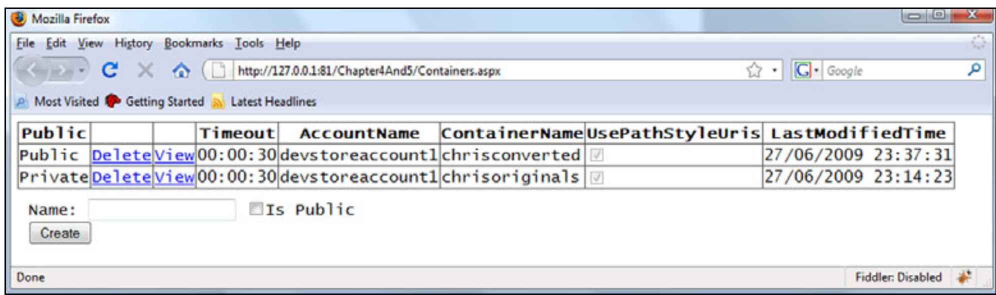


Figure 8.11 When you've finished this section, you'll have an application that can display a list of containers in your storage account using ASP.NET.

fashion. Typical scenarios for using containers are file hosting, enterprise workflows, and data manipulation applications that need to access data. In the next chapter, we'll look at how you can use containers in these kinds of scenarios by generating dynamic keys; we'll also talk about setting permissions on containers.

For now, we'll return to our podcast example. You need a method of creating public and private containers to store your original and converted podcasts in. In this sample application, you'll create an ASP.NET web form page in which to do that. Your final application will look like the screenshot displayed in figure 8.11.

Over the next few sections we'll look at the following topics, which will help you build the web page displayed in figure 8.11:

- Working with the StorageClient library (which is included in the Windows Azure SDK)
- Accessing the development storage account
- Creating a container
- Listing containers
- Deleting a container

By the end of this section, you'll have created the above web page and you'll have a vast knowledge of containers.

Before we get started, you need to create a new web role project in Visual Studio (as described in chapter 1). You should call this web role `PodcastSample`.

8.4.1 Accessing the StorageClient library

There are two ways of interacting with any of the storage services: you can either use the REST API directly or you can use the StorageClient library API. In this book, we're going to look at both methods.

One of the reasons that we'll look at both methods is that the StorageClient library is just a .NET wrapper for the REST API. For new features, Microsoft will often release the REST API call before adding the feature to the StorageClient library. By understanding both methods of interaction, you'll be able to use any new feature immediately (if you need to).

Another reason for looking at the REST API is that the underlying mechanism is heavily abstracted away from you. By understanding the underlying calls, you can make the best decisions architecturally for your application (especially regarding performance).

Now you might be thinking, "If the REST API is so great, why are we using the StorageClient library?" The answer to that is quite simple: as flexible as the REST API is to use, it's one huge pain. Using the REST API directly means we get to write unreadable code; `HttpWebRequest` code with no IntelliSense support. By using the StorageClient library wherever possible, we get to write familiar .NET code (with IntelliSense support), which increases productivity. We can ultimately spend more time doing more important things (like browsing the internet or playing Halo).

Although the StorageClient library (Microsoft.WindowsAzure.StorageClient.dll) is automatically referenced in any new web or worker role projects that you create, you can add the reference manually if you need to. You can find the reference at C:\Program Files\Windows Azure SDK\v1.1\ref\.

You don't need to add this assembly to your project; it's already referenced. But when you're building your own code, you'll probably want to split your code into proper layers. If you do that, you'll need to add the assembly to your own custom assembly.

Now that you know how to reference your assembly, let's look at how you configure the StorageClient library to access your development storage account.

8.4.2 Accessing development storage

To use development storage for storing BLOBs, you need to configure your application to use the development BLOB service in the same way as you would if it were the live system.

There are two ways to tell your code to connect to the local development storage. The first is to use a magic string as your connection string. If you set your connection string to `UseDevelopmentStorage=true,` the development storage fabric will respond to the connections. You can also put in the real development storage fabric connection. The format of the connection string for development storage will look a lot like a production connection string. The following parts are all that's different:

- *Account name*—The only account name that's supported for development storage is the default name, `devstoreaccount1`.
- *Account shared key*—The default name for this value is this:
  ```
  Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50u
  ➥ SRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==
  ```
- *Endpoint*—The BLOB storage endpoint for development storage BLOB services is by default 127.0.0.1:10000. To access the BLOB file in the example in section 8.2.1 from the development storage BLOB service, you would use the following URI: http://127.0.0.1:10000/devstoreaccount1/ChrisConverted/Podcast01.wma. This URI can be formalized as http://127.0.0.1:10000/*<StorageAccountName>*/*<Container>*/*<BlobName>*.

You can change the default values for each of these items in the DSService.exe.config file, but it's recommended that you use the default values.

In development storage, there's no ability to create or host multiple storage accounts and there's no Azure portal, so you can't easily generate a new key. The endpoints and URI structure also differ from the live system.

With that knowledge in hand, let's look at how you can use this information to access your account in code.

STORING ACCOUNT DETAILS IN THE SERVICE CONFIGURATION FILE

To make things a little easier, the Windows Azure SDK provides a property that will spin up a `CloudStorageAccount` object with the default development storage settings:

```
CloudStorageAccount = CloudStorageAccount.DevelopmentStorageAccount;
```

Although this is probably the quickest method of getting started, it isn't best practice. Starting this way, you're effectively hardcoding your application to your development account. You'd have to modify and recompile your code before you could deploy your application to the live system, which can complicate your build process and introduce bugs.

The best practice for storing this information is to store the data in the service configuration file, which gives you the option to change this information without redeploying the whole application. For example, if your shared key is compromised, then you would be able to generate a new shared key and modify your application to use the new key by simply changing the service configuration via the Azure portal.

To help you easily use the ServiceConfiguration.cscfg file to store your account details, the StorageClient library provides a method that can extract the account name, shared key, and endpoint from a configuration setting. The following call is used by the library to extract these values:

```
CloudStorageAccount =
   CloudStorageAccount.FromConfigurationSetting("DataConnectionString");
```

In the above example, your account details will be extracted from a configuration setting named `DataConnectionString`.

Although this code is specific to the StorageClient library, you should still store the account details in the service configuration file, even if you're using the REST API directly. Storing the details there will simplify and standardize your code and allow you to easily use both the StorageClient library and the REST API directly within your application (you don't want to have to modify two different settings to access your account).

Now that you know how easy it is to extract an account from your configuration, let's look at how you define that configuration setting.

DEFINING CONFIGURATION SETTINGS

As explained in chapter 4, you'll first need to define your configuration settings in the service definition file before you can configure them. The following setting is the standard method of defining your storage account in your service definition file.

```
<ConfigurationSettings>
   <Setting name="DataConnectionString"/>
</ConfigurationSettings>
```

Notice in this code that `DataConnectionString` is the same name that was passed to the `FromConfigurationSetting` method to extract the account name, shared key, and endpoint values.

COMMUNICATING WITH DEVELOPMENT STORAGE

After you've defined the configuration settings, you can set the runtime values in your service configuration file. The following configuration settings are the defaults used to talk to development storage:

```
<ConfigurationSettings>
  <Setting name="DataConnectionString"
           value="UseDevelopmentStorage=true" />
</ConfigurationSettings>
```

By setting the value of `DataConnectionString` to `UseDevelopmentStorage=true`, you're effectively telling the storage client to extract your settings from the DSService.exe.config file, which gives you the same result as using the `DevelopmentStorageAccount` property.

The advantage of using the `FromConfigurationSetting` method over the `DevelopmentStorageAccount` property is that you can modify the service configuration file to use the live account details (shown later in this chapter) without having to recompile or redeploy your application.

Now that your application is configured to use development storage via the StorageClient library, you can continue on and create your web page.

8.4.3 Creating a container

Now you're going to create the web page shown in figure 8.11. In your podcast sample web role project, create a new ASPX page called containers.aspx.

At this stage, you only want to write the code that will create your container; you don't need to see the list of containers. At present, your UI needs to display only the new container name text box and the Create Container button.

The following listing shows the ASPX required for the create-container section of the page.

Listing 8.1 ASPX for creating a container

```
<div>
   <div>
   Name: <asp:TextBox ID="txtContainerName" runat="server" />
</div>
<div>
   <asp:Button ID="btnCreate" runat="server"
               Text="Create" OnClick="btnCreate_Click" />
</div>
</div>
```

Creates the button

You've defined your UI. Now you need to create the code that handles the button click. See the following listing, which contains the code-behind for the create button click event.

Listing 8.2 Creating a new container

```
protected void btnCreate_Click(object sender, EventArgs e)
{
   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();                 <- Creates the BLOB client

   CloudBlobContainer container =
      blobClient.GetContainerReference(txtContainerName.Text.ToLower()); <- Gets reference to container

   container.Create();                  <- Creates container
}
```

Before we explain this code, we want to remind you of its purpose. The user will type in the new container name and then click the button to create the new container. Now let's look at the code.

STORAGE ACCOUNT

The first thing you need to do is retrieve an object that allows you to work with the BLOB storage account. Using the `CloudStorageAccount` object that you used earlier to extract your credentials, you can now instantiate a `CloudBlobClient` object that will allow you to mess with things at an account level by issuing the following call:

```
CloudBlobClient blobClient =
      account.CreateCloudBlobClient();
```

After you've retrieved the `CloudBlobClient` object, you can perform the following operations at an account level on BLOB storage:

- Return a list of all containers in the account (`ListContainers`)
- Get a specific container (`GetContainerReference`)
- List BLOBs (`ListBlobsWithPrefix`)
- Get a specific BLOB (`GetBlobReference`)

As well as performing these operations, you can also set some general policies, including the following ones:

- Block sizes
- Retry policy
- Timeout
- Number of parallel threads

In this example, because you're creating a new container, you need to grab a reference to the container that you want to create. Use the `GetContainerReference` method, passing in the name of your new container:

```
CloudBlobContainer container =
      blobClient.GetContainerReference(txtContainerName.Text.ToLower());
```

In this example, you're setting the container name to whatever the user types in the text box.

> **NOTE** The name of the container is converted to lowercase because the BLOB storage service doesn't allow uppercase characters in the container name.

So far you've just set up the container you want to create; you haven't made any communication with the storage service. The `CloudBlobContainer` object that has been returned by `GetContainerReference` can perform the following operations:

- Create a container (`Create`)
- Delete a container (`Delete`)
- Get and set any custom metadata you want to associate with the container
- Get properties associated with the container (for example, `ETag` and last modified time)
- Get and set container permissions
- List BLOBs (`ListBlobs`)
- Get a specific BLOB

Now make a call to create the container:

```
container.Create();
```

As soon as you call the `Create` method, the storage client generates an HTTP request to the BLOB storage service, requesting that the container be created.

> **Default permissions**
>
> In the `Create` container call, you didn't specify any permissions on the container to be created. By default, a container is created as private access only, meaning that only the account owner can access the container or any of the BLOBs contained within it. In the next chapter, we'll look at how you can set permissions on containers and BLOBs.

You should now be able to run your web role and create some containers in your development storage account. At this point, you won't be able to see the containers that you've created in your web page, but you can check that they're there by running a SQL query against the BlobContainer table in the development storage database.

Now that you can create a container from your web page, you need to modify the page so that you can display all the containers in your storage account.

8.4.4 Listing containers

In the figure 8.11, there's a grid that contains a list of all the containers in the account. To create that grid, you need to update your asp.net page to include an ASP.NET `GridView` component (you're going to eventually bind this grid to a list of

containers). You should place the code in the following listing before the code in listing 8.1 in the containers.aspx page.

Listing 8.3 Listing BLOBs with a `GridView`

```
<asp:GridView ID="gvContainers" runat="server"
              AutoGenerateColumns="true"
              onrowcommand="gvContainers_RowCommand"
              onrowdeleting="gvContainers_RowDeleting">
   <Columns>
       <asp:TemplateField>
          <ItemTemplate>
          <asp:LinkButton ID="btnDelete" runat="server"
                          Text="Delete"
                          CommandName="Delete"
  CommandArgument='<%#Eval("Name")%>'/>
          </ItemTemplate>
        </asp:TemplateField>
    <asp:HyperLinkField Text="View"
            DataNavigateUrlFields="Name"
            DataNavigateUrlFormatString="Blobs.aspx?Container={0}" />
   </Columns>
</asp:GridView>
```

Deletes container

Hyperlink for page listing BLOBs

The code in listing 8.3 is the ASP.NET markup for figure 8.11. Notice that you're allowing the grid to autogenerate all the columns (except the Delete button and the View hyperlink) based on the properties of the object bound to the grid. Listing 8.4 shows the code-behind for your web page that gets a list of containers from the account and binds it to the grid.

Listing 8.4 Binding the grid

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
       CloudStorageAccount.SetConfigurationSettingPublisher(
       ➥ (configName, configSetter) =>
            {
              configSetter(RoleEnvironment
              ➥ .GetConfigurationSettingValue(configName));
          });
   BindGrid();
}
}

private void BindGrid()
{
   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();
```

```
gvContainers.DataSource =
   blobClient.ListContainers();                  Sets grid data source as list
                                                 of containers in account
                                 Binds data grid
gvContainers.DataBind();         to its data source
}
```

One interesting thing you'll see in the `Page_Load` method is a call to `SetConfigurationSettingPublisher`. We wouldn't normally put this code here, but it was the easiest place to put it in the book. When you load configuration in ASP.NET, it looks in the web.config file by default. If you're storing your configuration in the .csdef file, ASP.NET will never find it. By including this line of code, you're telling ASP.NET to look in the .cscfg file for the configuration you're trying to load. We would normally put this in the `Role_OnStart` event, or somewhere else where it'll be run once per role instance as it starts up.

Now you have a web page that will display all the containers in your storage account. The page also allows you to create new private containers. To complete your sample, you just need to implement the delete functionality.

8.4.5 Deleting a container

You want to be able to click the Delete button on a particular row in your web page to delete the underlying container. For this to happen, you need to hook in your Delete button. The following listing shows the code-behind for implementing the delete functionality.

Listing 8.5 Deleting the container

```
protected void gvContainers_RowCommand
                       (object sender, GridViewCommandEventArgs e)
{
   if (e.CommandName == "Delete")
   {
       DeleteContainer(e.CommandArgument.ToString());   Calls DeleteContainer
   }                                                    method on receiving
                                                        Delete command
   BindGrid();
}

private void DeleteContainer(string containerName)
{
      CloudStorageAccount account =
         CloudStorageAccount.FromConfigurationSetting
         ("DataConnectionString");

      CloudBlobClient blobClient =
      account.CreateCloudBlobClient();

      CloudBlobContainer container =                    Gets container from
      blobClient.GetContainerReference(containerName);  BLOB account

   container.Delete();          Deletes container
}
```

```
protected void gvContainers_RowDeleting(object sender,
➥ GridViewDeleteEventArgs e)
{

}
```

Empties RowDeleting handler to avoid exceptions

With the Delete button code hooked in, you should be able to run your application and view all the BLOB containers in development storage, add a container, and then delete it from your web page.

Wow, you've done a great job. You've just completed your first Windows Azure BLOB storage application. All that's left is to make this baby work against the live BLOB storage service.

8.5 Configuring your application to work against the live service

To switch your application from the development storage to the live storage account, you need to create a live storage account and switch your configuration to it. In this section, we won't go through the process of creating a storage account; it's pretty simple and the information you require is available in this chapter. We're going to focus on configuring your application to work against your live storage account.

> **Always set affinity for your storage account**
>
> During the process of creating your storage account, always set affinity, as described in chapter 2 when we discussed the Azure portal. If you don't set affinity, it's possible that your web role might be hosted in Washington State, but your storage account is hosted in Chicago. The latency caused by cross-data-center communication will harm the performance of your application. To gain maximum performance, always set affinity on your web roles, worker roles, and storage accounts to the same data center; this ensures that you achieve the best possible network latency.

8.5.1 Switching to the live storage account

To make your application work against the live system, all you need to do is modify the value of your configuration setting in the service configuration file. That's it, the end, nothing else to do. If you remember earlier, you set your storage account configuration setting to the following:

```
<Setting name="DataConnectionString"
            value="UseDevelopmentStorage=true" />
```

Although this is great for the development storage, it doesn't give you a clue to the structure of the setting for when you want to use the live system. The following setting shows how the string should be structured.

```
<Setting name="DataConnectionString"
            value="DefaultEndpointsProtocol=protocol;
            AccountName=storageaccountname;
            AccountKey=storageaccountkey" />
```

To make this run against the live system, plug in the appropriate values:

```
<Setting name="DataConnectionString"
         value="DefaultEndpointsProtocol=http;
         AccountName=silverlightukstorage;
         AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEtlCDX
         ➥ J1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==
" />
```

Now that you've configured the live settings, you can use the live BLOB storage system either from the development fabric or from the live production fabric. The only thing left to do is to configure the access key.

8.5.2 Configuring the access key

If you're unsure where you get the account key for the storage account, you can always refer to the Azure portal. When a new storage account is created in Windows Azure, a primary and secondary access key is generated for you to secure your API requests. The access key for your live account is used by all storage services and is available from the storage account section of the portal. Figure 8.12 shows the window in which you can retrieve your access key. When you retrieve this key, it replaces the development key held in the service configuration file.

If your access key is compromised at any point, you can generate a new key by clicking the Regenerate button shown in figure 8.12. After a new key has been generated, you'll need to update the key in the service configuration for your application. You also have two keys that are valid at one point in time. Both keys are identical in what they allow the user to do with them. Having two keys is a great way to provide rolling key updates without any downtime in your system.

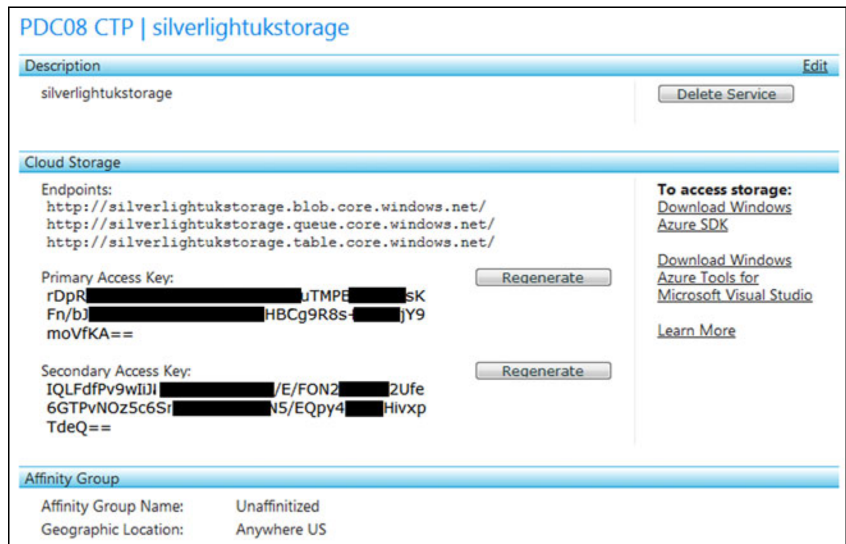


Figure 8.12 Access key in the Azure portal

8.6 Summary

In this chapter, we provided a quick overview of the sorts of problems that you would normally face when trying to provide a shared storage solution in a traditional web farm. Sharing files between multiple servers isn't easy, but Windows Azure provides a neat mechanism that lets you forget about those worries. After a brief introduction to storage services, we showed you how BLOB storage fits into the overall architecture.

Then we jumped right into developing your first application, using the BLOB storage service. First, we used the StorageClient library supplied in the SDK to hit the development environment; then we changed the configuration so it could work against the live production system.

Now you have an appreciation of how easy it is to get started with BLOB storage and containers. Next we're going to look at how to use the APIs that work with BLOB files themselves.

9 Uploading and downloading BLOBs

This chapter covers

- Uploading files
- Downloading files
- Serving files from BLOB storage using an HTTP handler
- Improving performance using local storage
- Using custom metadata
- Blob storage shared access

In the previous chapter, we showed you how to get started with BLOB storage development using the StorageClient library, with a focus on managing accounts and containers. In this chapter, we're going to look at the underlying REST API for the BLOB service and how to manage BLOBs using the StorageClient library.

9.1 Using the REST API

So far we've only used the StorageClient sample library in the SDK and have ignored the REST API. The reason for this is that, as a developer, you're unlikely to

be writing code directly against the REST API. In general, you'll prefer to use a more object-oriented structure that uses familiar-looking .NET classes.

The StorageClient library is useful but it's only a wrapper implementation of the REST API (which is the only official API). So although you'll mainly be working against the StorageClient library, there are some instances when you might need to use the REST API directly.

Windows Azure is an evolving platform and the Windows Azure team typically releases new features exposed via the REST API first. At a later date, they might provide an update to the SDK. If there's a new feature that you badly need to use, you might not have the luxury of waiting for the SDK update.

Another reason that you might need to use the REST API directly is that not all features are implemented (or implemented in the way you might expect) in the SDK; you might need to drop down to the REST API to use that feature. Rather than showing you every single feature with the REST API, we'll try to show you the important parts: how to list BLOBs in a public container and how to authenticate private requests using the REST API.

9.1.1 Listing BLOBs in a public container using REST

In this example, you're going create a small console application that'll return a list of all the BLOBs in a public container using the REST API. To do that, let's return to the funky little podcasting conversion sample that we were developing in the previous chapter. In that application, let's assume that you've converted a bunch of MP3s to WMA, and now you want to list all the converted podcasts. In the console application that you're going to develop, all the BLOBs stored in the ChrisConverted public container (which holds the WMA files) are going to be returned from the silverlightsukstorage BLOB service account in the live production system. Figure 9.1 shows the information that's returned from the request within your console application. This XML output shows that this container contains a single .wma file called mi2limpbiskit.wma.

```
file:///C:/AzureBook/Source/PlayArea/AzurePlayArea/WebLoad/bin/Debug/WebLoad.EXE
<EnumerationResults ContainerName="http://silverlightukstorage.blob.core.windows
.net/chrisconverted">
  <Blobs>
    <Blob>
      <Name>mi2limpbiskit.wma</Name>
      <Url>http://silverlightukstorage.blob.core.windows.net/chrisconverted/mi2l
impbiskit.wma</Url>
      <LastModified>Sun, 28 Jun 2009 17:21:33 GMT</LastModified>
      <Etag>0x8CBC62C04ACF0B0</Etag>
      <Size>5128497</Size>
      <ContentType>application/octet-stream</ContentType>
      <ContentEncoding />
      <ContentLanguage />
    </Blob>
  </Blobs>
  <NextMarker />
</EnumerationResults>_
```

Figure 9.1 Console application that returns a list of BLOBs from a public container, using REST

To create the code that returns this output, create a new console application in Visual Studio and replace the existing static main method with the code in the following listing.

Listing 9.1 Listing the BLOBs in a container via REST

```
static void Main(string[] args)
{
   HttpWebRequest hwr =
      CreateHttpRequest(new Uri(@"http://
     silverlightukstorage.blob.core.windows.net/
   ➥ chrisconverted?restype=container&comp=list"),
   ➥ "GET", new TimeSpan(0, 0, 30));

   using (StreamReader sr =
      new  StreamReader(hwr.GetResponse().GetResponseStream()))
   {
   XDocument myDocument = XDocument.Parse(sr.ReadToEnd());

   Console.Write(myDocument.ToString());
   }

   Console.ReadKey();
}


private static HttpWebRequest CreateHttpRequest(
➥ Uri uri, string httpMethod, TimeSpan timeout)
{
   HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(uri);
request.Timeout = (int)timeout.TotalMilliseconds;
request.ReadWriteTimeout = (int)timeout.TotalMilliseconds;
request.Method = httpMethod;
request.ContentLength = 0;
request.Headers.Add("x-ms-date",
   DateTime.UtcNow.ToString("R", CultureInfo.InvariantCulture));
request.Headers.Add("x-ms-version", "2009-09-19");
         return request;
}
```

1. Sets up the request with correct destination URI
2. Gets stream that represents results returned from GET
3. Adds correct date header
4. Adds correct version header

Wow, that's quite a bit of code. All it really does is list the BLOBs in a public container and output the result to the console (as shown in figure 9.1). Unfortunately, whenever you use the REST API directly, your code will get more complex. (I guess you can see why we prefer to use the StorageClient library.)

Remember that the HTTP requests that were generated by the code in listing 9.1 are the same requests that the StorageClient library generates on your behalf.

In listing 9.1, the `GET` request is created at (1). This verb indicates that you want some data returned from the server rather than have an action performed that'll update the data (such as a create, update, or delete). The request is executed at (2).

Let's now take a deeper look at the rest of the code in listing 9.1; doing so will give you a better understanding of the communication between your clients and the storage accounts.

THE URI

Look at the URI that you're calling at (1). There's some interesting information about the request that's being made.

From the domain, you can determine that you're using the live BLOB storage service (`blob.core.windows.net`) and that the request is being made against the storage account silverlightukstorage. Looking at the request, you can also derive that you want a list of whatever is in the container ChrisConverted (`chrisconverted?comp=list`), which we know are BLOBs (in fact, they're the WMA files that were converted from MP3).

Windows Azure follows a standard naming convention for performing requests; as soon as you're familiar with some of the API calls it's easy to infer what other calls might look like. For example, if you required a list of whatever resides in a storage account (containers), you could use the following URI:

```
http://silverlightukstorage.blob.core.windows.net/?comp=list
```

You would need to sign the request with your access key because it isn't a public operation. Listing BLOBs in a public container can be performed without an authorization key because an authorization key is required only for private containers.

THE REQUEST HEADERS

In the code for the standard `CreateHttpRequest` in listing 9.1, two headers are set: `x-ms-version` and `x-ms-date`.

The `x-ms-version` header (4) is an optional header that should be treated as a required header. The storage service versioning policy is that when a new version of an API is released, any existing APIs will continue to be supported. By providing the correct `x-ms-version` header, you're stating which API you want your request to work against. Using this policy, Microsoft can release new functionality and change existing APIs but allow your existing services to continue to work against the previous API.

> **TIP** You should always check the version of the REST API that you're using to support a particular feature. At (4) we're using the September 19, 2009 version of the API. If a new feature is released and it isn't working, there's a good chance that you forgot to update the version. The good news is that whenever you download the latest version of the StorageClient library, it'll already be using the correct version.

The `x-ms-date` header is a required header that states the time of the client request. We set this value at (3) in listing 9.1. The value set in the request header is a representation of the current time in UTC; for example, "Sat, 27 Jun 2009 23:37:31 GMT". This request header serves two purposes:

- It allows the server to generate the same authorization hash as the client
- It prevents replay attacks by denying old requests

> **TIP** If you suddenly start getting errors whenever you call the storage service, it might be worth checking the time on your machine. If the time of the

request is out of synchronization with the server time in the data centers (older than 15 minutes), the request will be rejected with a 403 response code.

We've looked at how to make non-authenticated requests against a public container and how to make requests to the storage accounts via the SDK. We'll now look at how to make authenticated requests via the REST API and give you an understanding of how the REST API calls are authenticated.

9.1.2 Authenticating private requests

In the previous section, you developed a console application to return a list of the files that reside in your public container (ChrisConverted). Now you're going to modify this code to return all the containers in your development storage account. This sample is the direct REST API equivalent of the storage client calls that we performed in the previous chapter.

Because there's only two containers in the development storage account (ChrisOriginals and ChrisConverted), we expect the following XML output from the console application:

```
<EnumerationResults AccountName="http://128.0.0.1:10000/devstoreaccount1/">
  <Containers>
    <Container>
      <Name>chrisconverted</Name>
      <Url>http://128.0.0.1:10000/devstoreaccount1/chrisconverted</Url>
      <LastModified>Sat, 27 Jun 2009 23:37:31 GMT</LastModified>
      <Etag>0x8CBC5975FB7A0D0</Etag>
    </Container>
    <Container>
      <Name>chrisoriginals</Name>
      <Url>http://128.0.0.1:10000/devstoreaccount1/chrisoriginals</Url>
      <LastModified>Sat, 27 Jun 2009 23:14:23 GMT</LastModified>
      <Etag>0x8CBC594247EFB60</Etag>
    </Container>
  </Containers>
  <NextMarker />
</EnumerationResults>
```

Listing 9.2 contains the code that we used to make this request via the REST API. Before you can run this sample, you'll need to reference the storage client because you're going to use some of its library calls to sign the HTTP request with the shared access key. You'll also need the following `using` statement at the top of your class:

```
using Microsoft.WindowsAzure.StorageClient;
```

Listing 9.2 Listing the containers in the development storage account via REST

```
static void Main(string[] args)
{
   HttpWebRequest hwr = CreateHttpRequest(
      new Uri(@"http://127.0.0.1:10000/devstoreaccount1?comp=list"),
      "GET", new TimeSpan(0, 0, 30));                    <- Creates HTTP request as before
```

```
...CloudStorageAccount.SetConfigurationSettingPublisher
➥ ((configName, configSetter) =>
                 { configSetter(ConfigurationManager
                 ➥ .AppSettings[configName]); });

var account =
   CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

account.Credentials.SignRequest(hwr);

using (StreamReader sr =
          new
     StreamReader(hwr.GetResponse().GetResponseStream()))
{
   XDocument myDocument = XDocument.Parse(sr.ReadToEnd());
                Console.Write(myDocument.ToString());
}

   Console.ReadKey();
}
```

Gets account credentials using StorageClient library

Signs request using StorageClient library

Executes request and gets response

The code in listing 9.2 is similar to the code in listing 9.1 except that we're using a different URI (because we're listing containers rather than BLOBs) and signing the request. The URI for listing containers in the development storage account is http://127.0.0.1:10000/devstoreaccount1/?comp=list. This URI will return a list of the contents of the storage account (a list of containers).

Apart from the URI, the only difference between the two listings is that you sign the HTTP web request with your shared-key credentials. The process of signing the request is quite complicated; it's best to use the storage client code as we've done in listing 9.2.

Use the same code that you wrote earlier to extract the configuration setting:

```
var account =
   CloudStorageAccount.FromConfigurationSetting("DataConnectionString");
```

> **TIP** `CloudStorageAccount.FromConfigurationSetting` will work with both Windows Azure `ConfigurationSettings` and regular .NET application settings.

Now sign the request using the request signing functionality built into the StorageClient library:

```
account.Credentials.SignRequest(hwr);
```

Using the `SignRequest` method also adds the `x-ms-date` header information to the request, so you don't have to add it on your own. Be aware that if you add the `x-ms-date` header on your own and you sign the request with `SignRequest`, the header might not be populated correctly.

Even if you do plan to use only the REST API, it's still worth referencing the StorageClient library just to use that bit of code.

SIGNING THE REQUEST

Although the request signing code is already implemented, let's look at the overall process of signing a request.

If you were to look at the HTTP request that you generated in listing 9.2 (you could stick a breakpoint after the request has been signed), you would see that the request contains an Authorization header. The Authorization header for the request in listing 9.2 is

```
SharedKey devstoreaccount1:J5xkbSz7/7Xf8sCNY3RJIzyUEfnj1SJ3ccIBNpDzsq4=
```

The signature in the header (the long string after `devstoreaccount1`) is generated by canonicalizing the request. The canonicalized request is hashed using a SHA-256 algorithm and then encoded as the signature using Base64 encoding.

> **DEFINITION** Canonicalizing is a defined process that converts a request into a predictable request. You can find more information about the canonicalization process at http://msdn.microsoft.com/en-us/library/dd179428.aspx.

PROCESSING THE REQUEST

After the request is received by the server, the server takes the incoming request and performs the same canonicalization and hashing process with the shared key. If the signature that's generated matches the Authorization header, the server can perform the request. Figure 9.2 shows the validation process between the client and the BLOB storage service. Notice that the authorization key generated by the server matches the original client request.

If the signature generated by the server is different from the Authorization header, then the server won't process the request and returns a 400 response code (`Bad Request`). Being able to generate the same authorization key both client-side and server-side means that users who don't have the shared key are prevented from performing unauthorized requests against the account. Because the shared key is never sent between the client and the BLOB storage service, the risk of the key being compromised is substantially reduced.

The authorization key is generated from both the contents of the request and the shared key. Generating the hash with both pieces of information means that the user can't tamper with the request to perform another operation. If, for example, a hacker

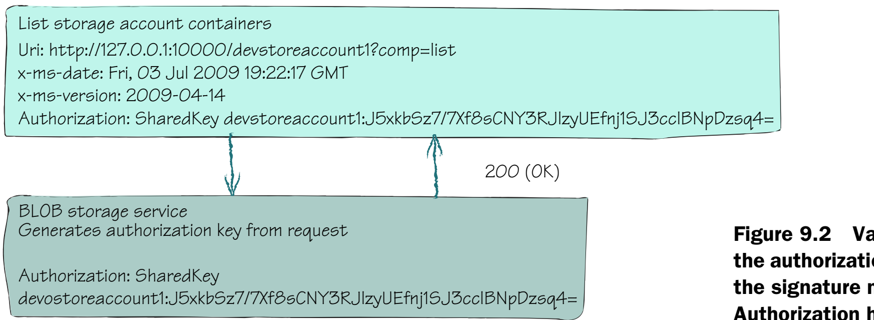


Figure 9.2 Validation of the authorization key when the signature matches the Authorization header

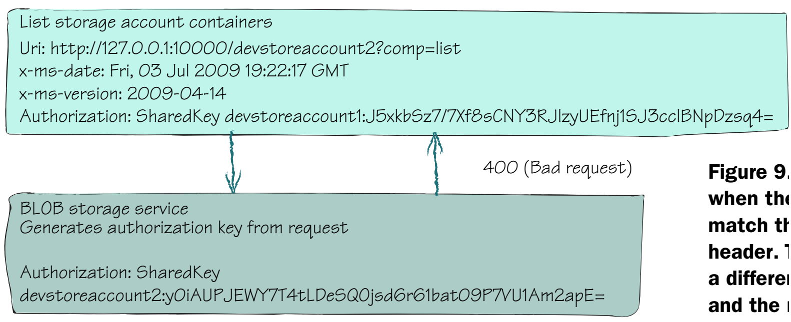


Figure 9.
when th
match t
header.
a differe
and the

were to intercept your previous request, then the generated Authorization header server-side would change and the request would be denied. Figure 9.3 shows the validation process between the client and the BLOB storage service when the original request from figure 9.2 has been tampered with.

In figure 9.3, we've tampered with the original request from figure 9.2 to return any containers from `devstoreaccount2`. Notice that the server generated a different authorization key from the tampered request and therefore the server returns a 400 response code (`Bad Request`).

Now that you understand how the REST API authentication process works, take a break and have a quick beer. Just think, if you had to implement that code yourself rather than Microsoft providing it, you'd probably prefer to have a longer break and drown your sorrows.

9.2 Managing BLOBs using the StorageClient library

Now that you've finished your break and you're rested enough to read this, let's look at how you can use the StorageClient library to list BLOBs in a container, rather than using the REST API directly.

In chapter 8, you developed a sample management application that would allow you to upload podcasts in an MP3 format that were ready to be converted to WMA. You made an ASP.NET page that displayed a grid of all the containers in your storage account. Included in the grid was a hyperlink that would redirect you to another page called blobs.aspx, passing in the name of the selected container in the query string. We're going to extend that example now to develop the page blobs.aspx, shown in figure 9.4. This page is similar to the page that you developed for listing containers in chapter 8.

You can perform the following actions with the page shown in figure 9.4:

- List all BLOBs in the selected container
- Upload a new file
- Delete an existing BLOB
- Download an existing BLOB

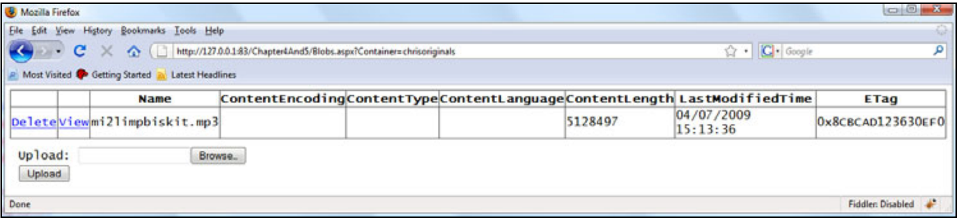


Figure 9.4 An ASP.NET web page that displays all the BLOBs that are in the selected container in a grid control

Let's look at each of these actions in detail.

9.2.1 *Listing BLOBs using the storage client*

In this section, we'll look at how to generate the grid that displays the list of BLOBs. We'll explain how to upload, delete, and view BLOBs later in the chapter. The following listing contains the markup required for the blobs.aspx page shown in figure 9.3.

Listing 9.3 ASPX page that lists the BLOBs in a container

```
<div>
   <asp:GridView ID="gvBlobs" runat="server"
                 onrowcommand="gvBlobs_RowCommand"
                 onrowdeleting="gvBlobs_RowDeleting">
      <Columns>
         <asp:TemplateField>
            <ItemTemplate>
               <asp:LinkButton ID="btnDelete" runat="server"
                               Text="Delete"
                               CommandName="Delete"
                               CommandArgument='<%#Eval("Uri")%>'/>
            </ItemTemplate>
         </asp:TemplateField>
         <asp:TemplateField>
            <ItemTemplate>
               <asp:LinkButton ID="btnView" runat="server"
                             Text="View"
                             CommandName="View"
                             CommandArgument='<%#Eval("Uri")%>'/>
             </ItemTemplate>
          </asp:TemplateField>
         <asp:BoundField HeaderText="File" DataField="Uri"/>
       </Columns>
   </asp:GridView>
   <div>
      <div style="padding: 10px;">
         <div>
            Upload: <asp:FileUpload ID="fu" runat="server" />
         </div>
         <div>
```

1 Grid of BLOBs

2 View BLOB button

3 File upload control

```
            <asp:Button ID="btnUpload" runat="server"
                        Text="Upload"
                        OnClick="btnUpload_Click" />      ④ Upload button
         </div>
      </div>
   </div>
</div>
```

The ASPX code for displaying the BLOBs is similar to the ASPX we used in chapter 8 to list containers, so this shouldn't be too alien to you.

At ① a `GridView` is displayed that lists all the BLOBs in its data source. At ② you're defining a hyperlink button that you'll use to download the BLOB, and at ③ is the standard ASP.NET `FileUpload` control to upload the file.

THE CODE-BEHIND FOR THE WEB PAGE

You've defined how your web page will look. Now you need to bind the grid to the data source on page load. Listing 9.4 shows the code-behind required to display a list of BLOBs for the selected container in your grid.

Listing 9.4 Show the list of BLOBs in the grid

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
      BindGrid();
}

private void BindGrid()
{
   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();

   CloudBlobContainer container =
      blobClient.GetContainerReference(Request["container"] as string);   ① Gets container from request query string

   gvBlobs.DataSource = container.ListBlobs();      ② Lists BLOBs in container
   gvBlobs.DataBind();
```

The code shown in listing 9.4 is again similar to the list containers example in chapter 8. This code is the storage client equivalent to the code you used to display the list of BLOBs with the REST API directly in listing 9.1.

At ① you retrieve an instance of the BLOB container by calling the `GetContainer-Reference` method off the BLOB client. Notice that we're passing in the container name from the request query string. Finally, you retrieve a list of the files held in the container by calling the `ListBlobs` method and binding the result to the `GridView` ②.

Now that you can list and display all of the BLOBs in the selected container, you need to extend the web page so that you can upload new files into the container.

9.2.2 Uploading BLOBs

To upload files from your web page, you're going to use the built-in ASP.NET upload control at (3) in listing 9.3. On click of the upload button at (4), you're going to capture the uploaded file and then upload the captured file to BLOB storage. The following listing contains the code-behind for the upload button click event.

Listing 9.5 Posting the uploaded file to BLOB storage

```
protected void btnUpload_Click(object sender, EventArgs e)
{
   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();

   CloudBlobContainer container =
      blobClient.GetContainerReference(Request["container"] as string);
          var blob =
        container.GetBlobReference(fu.PostedFile.FileName);   <-- (1) Gets reference to new BLOB

     blob.UploadByteArray(fu.FileBytes);   <-- (2) Uploads file contents to BLOB

   BindGrid();
}
```

All the code in listing 9.5 has been discussed in previous examples, except for what's happening at (1) and (2). At (1) you get a reference to the BLOB that you're about to create. We're giving the BLOB the original name of the uploaded file (retrieved from the `UploadFile` control). Then, at (2) you upload the new file to BLOB storage.

> **TIP** You'll notice that we're extracting the filename and the file contents directly from the ASP.NET file upload control. If you want, you can give the file a name different from the original.

Setting the maximum request length

By default, ASP.NET is configured to allow a maximum upload of 4 MB. If you provide a web role frontend to the BLOB storage as we've done in this sample, you might need to increase the maximum request length.

To increase the default value to a larger value, you need to add the following line under the `system.web` element in the web.config file:

```
<httpRuntime executionTimeout="300" maxRequestLength="51200"/>
```

The maximum upload size in the above example is 50 MB.

In the previous example, we used the `UploadByteArray` method to upload the BLOB. Three other methods are provided in the StorageClient library that you can use:

`UploadFile`, `UploadText`, and `UploadFromStream`. Depending on your situation, one of these methods might be easier to use than `UploadByteArray` (for example, `UploadFile` might be a better choice if you have a local file on disk that you want to store in BLOB storage).

> **Splitting BLOBs into blocks**
>
> The maximum size of a BLOB is 1 TB, but if a file is larger than 64 MB, under the covers the StorageClient library splits the file into smaller blocks of 4 MB each. One of the advantages of the StorageClient library is that you don't need to worry about this. If you're using the REST API, you'll need to implement the splitting of BLOBs into blocks and the committal of blocks and retry logic associated with re-uploading failed blocks (yet another good reason to use the StorageClient library).
>
> If you're a sick and twisted individual who wants to mess around with blocks, then feel free to look in more detail at the online documentation at http://msdn.microsoft.com/en-us/library/ee691964.aspx.

Now that you've spent all that time and effort adding the file to BLOB storage, let's delete it (groan).

9.2.3 Deleting BLOBs

Deleting a BLOB is similar to uploading a file except that you're deleting the BLOB instead of uploading it (cute, huh?). Just like the upload file example in listing 9.5, you get the reference to the BLOB, and then delete the BLOB by calling the following:

```
blob.Delete();
```

The following listing shows the code that will delete the BLOB in your web page.

Listing 9.6 Deleting the BLOB

```
protected void gvBlobs_RowCommand
                        (object sender, GridViewCommandEventArgs e)
{
   if (e.CommandName == "Delete")
   {
       DeleteBlob(e.CommandArgument.ToString());
   }

   BindGrid();
}

private void DeleteBlob(string blobName)
{

   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();
```

```
    CloudBlobContainer container =
       blobClient.GetContainerReference(Request["container"] as string);

    var blob =
       container.GetBlobReference(blobName);

    blob.Delete();                                    Deletes the BLOB
}

protected void gvBlobs_RowDeleting(object sender,
➥ GridViewDeleteEventArgs e)
{
}
```

As you can see, deleting a BLOB is pretty simple. Now you can list, upload, and delete BLOBs from your storage account using your ASP.NET management website hosted in your Windows Azure web role. Let's complete the management page example by looking at how you can download BLOBs.

9.3 Downloading BLOBs

In this section we'll look at how to download BLOBs from both a public container and a private container. To take things nice and easy, we'll tell you how to download BLOBs that are stored in a public container first.

9.3.1 Downloading BLOBs from a public container

If your BLOB is hosted in a public container, you can present the URI of the BLOB directly to the user and they'll be able to directly download the file to their browser. In our podcasting sample, the following URI will download podcast01.wma from the ChrisConverted public container in the development storage account: http://127.0.0.1:10000/devstoreaccount1/ChrisConverted/podcast01.wma. Because the BLOB is held in a public container, the user won't need to provide any credentials to access the BLOB.

> **TIP** If you've correctly set the MIME type of your podcast, when the URI is pasted into your browser, the podcast will automatically start playing in Windows Media Player.

In our management web page example, we don't want to expose the podcasts to the world; we want to restrict access to our own credentials. Let's look at how you can do that using your new best friend, the storage client.

9.3.2 Downloading BLOBs from a private container using the storage client

Now we want you to modify your management web page so that if you click the View button for the selected podcast, as shown in figure 9.5, you're prompted to download the file (also shown in figure 9.5).

Although you can store the BLOB in a public container to achieve the same result, in this example you're going to first download the BLOB to your web role and then serve the BLOB from your web role, rather than directly from BLOB storage.

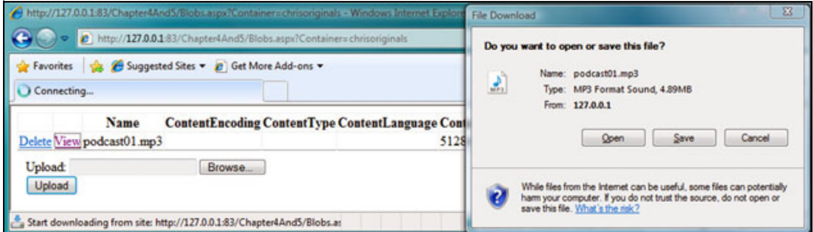


Figure 9.5 Clicking the View button on your BLOBs page opens a Save dialog box

In listing 9.3, we defined the ASPX for your View button; listing 9.7 is where we hook up the code that will download the file when this button is clicked. The code in the following listing will download the selected BLOB and prompt the user with the Save dialog box shown in figure 9.5.

Listing 9.7 Downloading BLOBs from the grid using the storage client

```
protected void gvBlobs_RowCommand(object sender,
➥ GridViewCommandEventArgs e)
{
   if (e.CommandName == "Delete")
   {
      DeleteBlob(e.CommandArgument.ToString());
   }

   if (e.CommandName == "View")
   {
       DownloadBlob(e.CommandArgument.ToString());
   }

   BindGrid();
}

private void DownloadBlob(string blobName)
{
   CloudStorageAccount account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   CloudBlobClient blobClient =
      account.CreateCloudBlobClient();

   CloudBlobContainer container =
      blobClient.GetContainerReference(Request["container"] as string);

   var blob =
     container.GetBlobReference(blobName);


   using (var ms = new MemoryStream());          ❶ Creates memory stream
   {                                               and downloads BLOB
      blob.DownloadToStream(ms);
      Response.ContentType = blob.Properties.ContentType;     ❷ Sets MIME type
      Response.AddHeader("Content-Disposition",
                          "attachment; filename=" blobName);
```

```
        Response.BinaryWrite(ms.ToArray());        ◁ Writes file to
    }                                              ③ response stream
}
```

At ❶ you create a new memory stream that writes the contents of the BLOB that's been downloaded from BLOB storage. At ❷ you set the MIME type of the file so that you can allow the browser to perform the correct action based on that type (for example, launch Windows Media Player, Microsoft Word, or some other action), and you add the content-disposition header so that the browser knows to offer a Save file dialog box to the user. Finally, the downloaded BLOB is made available to the client by writing the file to the response stream ❸.

> **TIP** In the previous example, we used the `DownloadToStream` option to download files from BLOB storage but the storage client also offers these methods to download files: `DownloadText`, `DownloadByteArray`, and `DownloadToFile`.

So far we've shown you how to download files that are in public or private containers. What if you want to do something a little more granular, like control access to your BLOBs or containers? For operations like this, you can use a Shared Access Signature. Later on, we'll look at this feature in a little more detail.

Now that you know how to upload and download BLOBs, let's look at how you can integrate BLOB storage with your existing ASP.NET websites.

9.4 Integrating BLOBs with your ASP.NET websites

In typical ASP.NET websites, you usually distribute your assets with your website. Although this strategy works great for small websites, it's pretty much unmanageable when dealing with larger websites. Do you really want to redeploy your entire website just because you have a new Hawaiian shirt in your product range and you need to add an image of it?

There's a better way. In this section, we'll tell you how you can integrate public assets and private assets (such as purchased MP3 files) with your ASP.NET website. Let's return to the Hawaiian Shirt Shop example from chapter 2 to find out how you can integrate your assets.

9.4.1 Integrating ASP.NET websites with table-driven BLOB content

Currently, the shirt shop website displays a list of all the products that you have for sale, but it doesn't display pictures of the shirts. Because the data is currently hardcoded, you could just store the image on the website directly; any changes to the product line would require you to deploy a new version of the website.

In future chapters, you're going to drive the data from an external data source such as Table storage or SQL Azure, so you need a strategy that lets you update static content on the website when the product line is changed, without redeploying the images to the website.

If you stored the pictures of the shirts in BLOB storage, you could store the URI of the BLOB in your external data source. As you add new items to your data source, you

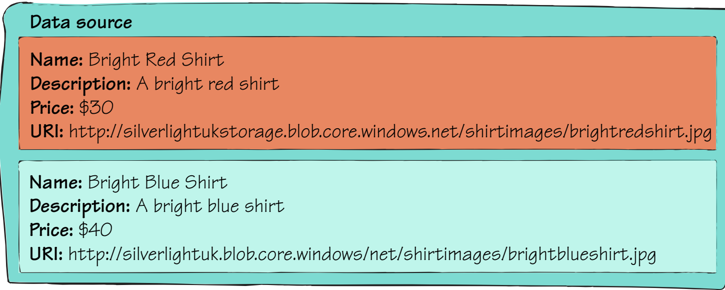


Figure 9.6 Storing URIs in an external data source

can add the associated BLOB at the same time. To expand on the Hawaiian Shirt Shop example, you could store an image of a shirt in BLOB storage as well as add a new shirt to the table. In the table, you would store all the details of the shirt (the name of the shirt, price, description, and so on) and its associated URI. Figure 9.6 shows a representation of this setup.

Because you're happy for the images of your Hawaiian shirts to be in the public domain, you can store the shirt images in a public container and set the URI of your HTML image tag directly to the public URI. The following code shows how this image tag might look:

```
<img src="http://silverlightukstorage.blob.core.windows.net
➥ /shirtimages/greenshirt.jpg" />
```

Using a public container is fine for storing content that you don't mind making available to the world, but it's not an acceptable solution if you want to serve content that you want to keep protected.

> **NOTE** Storing your website assets in BLOB storage is a great way to optimize performance because it takes load away from your web server. In the next chapter, we'll look at how you can take your assets that are stored in BLOB storage and expose them via the Windows Azure Content Delivery Network to make even more performance improvements.

Returning to the podcast example, suppose you decided that you wanted only paid members of your website to be able to download your MP3. In this scenario, you don't want to store the file in a public container; you want to store it in a private container instead.

9.4.2 Integrating protected, private content

Earlier in this chapter, in the BLOB management web page, you implemented a download link that allowed you to serve files stored in a private container to your users on a public website. Now we're going to expand that technique to a more integrated, reusable solution by doing the following things:

1. Creating an HTTP handler that serves MP3 files from BLOB storage
2. Registering the HTTP handler
3. Protecting your handler so that only authorized users can use it

HTTP handlers let specified types of HTTP requests be handled by some custom code. In this example, you're going to build a handler that will intercept requests for MP3 files and return the MP3 files from BLOB storage instead of from the local filesystem.

CREATING THE HTTP HANDLER

You're going to build a small sample that will intercept any incoming requests for MP3 files using an HTTP handler. Rather than attempting to serve the MP3 files from your website's filesystem, the handler will retrieve the files from your private BLOB storage container and serve them to the user who requested them. For this task, we're going to build on the techniques that we used in listing 9.7. You can either serve the files directly (useful for images) or initiate a Save dialog box (as shown in figure 9.5). In this example, you'll download a file directly. For example, when the user requests the following URI, they'll be served the file from BLOB storage:

```
www.mypodcastwebsite.com/podcast01.mp3
```

To implement an HTTP handler, add a new `httphandler` file to your ASP.NET web project. This handler is called `MP3Handler.cs`. Listing 9.8 contains the implementation for `MP3Handler.cs`.

Listing 9.8 An MP3 HTTP handler that serves MP3s from BLOB storage

```
public class MP3Handler : IHttpHandler
{
   public void ProcessRequest(HttpContext context)        <-- 1 Content is the incoming request
   {
      string blobName = context.Request.Path.Trim('/');
      var blobData = GetBlob("chrisoriginals", blobName);

      context.Response.ContentType = "audio/mpeg";
      context.Response.BinaryWrite(blobData);
   }

   public bool IsReusable
   {
      get {return false;}
   }

   private byte[] GetBlob(string containerName, string blobName)   <-- 4 Gets BLOB from BLOB storage
   {
      CloudStorageAccount account =
         CloudStorageAccount
         ➥ .FromConfigurationSetting("DataConnectionString");

      CloudBlobClient blobClient =
         account.CreateCloudBlobClient();

      CloudBlobContainer container =
         blobClient.GetContainerReference(containerName);

      var blob =
        container.GetBlobReference(blobName);
```

```
        blob.DownLoadByteArray();
    }
}
```

The HTTP handler in listing 9.8 will be called whenever an MP3 file request is intercepted at (1). When the request has been intercepted, the name of the .mp3 file that should be downloaded from BLOB storage from the requested path is extracted (2). The handler will then download the requested BLOB from the ChrisOriginals private container at (2) by calling the `GetBlob` method (4). After the file is retrieved from BLOB storage, you set the MIME type as `MP3` (3) (we've hardcoded the MIME type but in a more generic sample you could retrieve it from the BLOB properties) and then write the file back to the client.

REGISTERING THE HTTP HANDLER

To register the HTTP handler with your website, add the following line to your web.config file in the `handlers` section under the `system.webServer` element:

```
<add name="MP3Handler" path="*.mp3" verb="GET" type=
➥ "AzurePlayAreaWeb_WebRole.MP3Handler, AzurePlayAreaWeb_WebRole"
➥ resourceType="Unspecified"/>
```

This code will configure your server to route any web requests that end in .mp3 to your new handler. This is a great way to protect assets on your server that would normally be freely accessible. Your handler could check security permissions, route the call to virtual storage, or deny the request.

AUTHORIZATION

If you want to restrict your MP3 files to logged-in users, you can use the built-in ASP.NET authorization and authentication functionality:

```
<authorization>
  <deny users="?"/>
</authorization>
```

Now you have an integrated HTTP handler that can serve protected BLOB files from BLOB storage to authorized users as if it were part of your website. If your website were serving the same file to hundreds of users every day, then continually retrieving the same file from BLOB storage would be inefficient. What you would need to do in that

> **Shared Access Signatures**
>
> Putting BLOBs in local storage to improve performance is useful for showing you how to integrate local storage and BLOB storage together. It's also useful as an option for integrating content. You can use Shared Access Signatures to provide the same result. Using Shared Access Signatures is the preferred approach to protect a BLOB because it's cheaper and takes load off your servers.

case is use BLOB storage and local storage together, to cache requests, so you wouldn't need to continually request the same BLOB from BLOB storage.

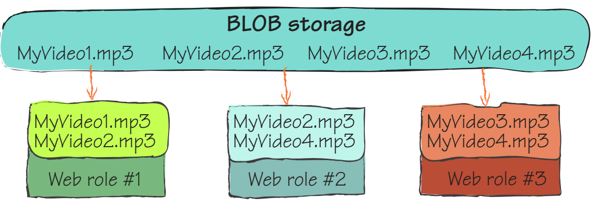
9.5 Using local storage with BLOB storage

In the previous section, you used a file handler to intercept MP3 requests from your website and to serve them from BLOB storage rather than from your website. A more efficient approach would be to check whether the file is already on your local filesystem. If the file already exists, then you can just serve the file straight up. If the file doesn't exist, you can retrieve it from BLOB storage, store it on the local filesystem, and then serve the file. All future requests for the file won't need to continually retrieve the file from BLOB storage.

9.5.1 Using a local cache

When you define Windows Azure web roles, you can allocate a portion of the local filesystem for use as a temporary cache. This local storage area allows you to store semi-persistent data that you might use frequently, without having to continually re-request or recalculate the data for every call.

You must be aware that this local storage area isn't shared across multiple instances and the current instance of your web role is the only one that can access that data. Figure 9.7 shows the distribution of BLOBs in local storage across multiple instances.



Because the load balancer evenly distributes requests across instances, a user won't always be served by the same web role instance. Any data that you need persisted across multiple requests (such as shopping cart or session data) shouldn't be stored in local storage and must be stored in a data store that all instances can access.

9.5.2 Defining and accessing local storage

As we discussed in chapter 4, you can define how much space you require on your local filesystem in your service definition file. The FC uses this information to assign your web role to a host with enough disk space. The following code is how you would define that you need 100 MB of space to cache MP3 files:

```
<WebRole name="ServiceRuntimeWebsite">
   <LocalResources>
      <LocalStorage name="mp3Cache"
                    cleanOnRoleRecycle="true"
                    sizeInMB="100" />
```

```
   </LocalResources>
</WebRole>
```

To access any of the files held in local storage, you can use the following code to retrieve the location of the file:

```
  LocalResource localCache =
    RoleEnvironment.GetLocalResource("mp3Cache");

 string localCacheRootDirectory = localCache.RootPath;
```

After you've retrieved the root directory of local storage (`RootPath`), you can use standard .NET filesystem calls to modify, create, read, or delete files held in local storage. (For more information about local storage, see chapter 4).

Great! You've configured your local storage area. Now you need to modify your HTTP handler to use your local storage area, where possible.

9.5.3 Updating your HTTP handler to use local storage

You're going to modify your HTTP handler to check in local storage to determine whether the requested file exists already. If the file doesn't exist, then you'll retrieve the MP3 file from BLOB storage and store it in local storage. Finally, you'll write the file back to the client from your local filesystem. The following code shows how you modify the `ProcessRequest` method in listing 9.8 to use local storage.

```
public void ProcessRequest(HttpContext context)
{
   string blobName = context.Request.Path.Trim('/');

   var mp3Cache = RoleEnvironment.GetLocalResource("mp3Cache");
   string localFilePath = mp3Cache.RootPath + blobName;

  if (!File.Exists(localFilePath))
  {
     var blobData = GetBlob("chrisoriginals", blobName);
     File.WriteAllBytes(localFilePath, blobData);
  }

  context.Response.ContentType = "audio/mpeg";
  context.Response.WriteFile(localFilePath);
}
```

Using the local storage mechanism in your HTTP handler improves performance by serving the requested file from local storage, rather than always having to retrieve the file from BLOB storage first. Although performance is improved, if the file is changed in BLOB storage, the file that you'll serve will be out of date. Let's look at how you can keep the performance improvement but serve the latest content even if the file has changed in BLOB storage.

> **TIP** Although this sample is focused on web HTTP handlers, this technique can easily be used in worker roles. I currently use this technique with a Windows Azure MapReduce solution that I've built.

```
file:///C:/AzureBook/Source/PlayArea/AzurePlayArea/WebLoad/bin/Debug/WebLoad.EXE
x-ms-request-id : bc06db6f-90f7-4ec6-97c1-3ca2b55e70ac
Content-Length : 5128497
Date : Sun, 05 Jul 2009 20:20:56 GMT
ETag : 0x8CBCB1016B0929A
Last-Modified : Sat, 04 Jul 2009 22:44:13 GMT
Server : Development Storage Blob Service Version 1.0 Microsoft-HTTPAPI/2.0
```

Figure 9.8 The output of a `HEAD` request

9.5.4 Checking properties of a BLOB without downloading it

If you were to use the HTTP `HEAD` verb instead of the `GET` verb, you could check the properties of the file without downloading the file. Figure 9.8 shows the output of a `HEAD` request.

In figure 9.8, the `Last-Modified` tag shows us the last time the file was updated in BLOB storage. By comparing the header value to the local value in your file properties, you know whether the file has changed.

> **x-ms-request-id**
>
> In figure 9.8, you'll notice that there's a tag called `x-ms-request-id`. Every request made is assigned a unique identifier (GUID) that's returned in the response. Every request and response is logged by Microsoft; if you're experiencing issues, you can always pass this ID with a support request—providing the ID lets Microsoft easily investigate any issues you have.

The listing that follows shows the code for the console application shown in figure 9.8.

Listing 9.9 Showing the output of a HEAD request

```
static void Main(string[] args)
{
      HttpWebRequest hwr = CreateHttpRequest(
         new Uri(@"http://127.0.0.1:10000/devstoreaccount1/chrisoriginals/
         ➥ podcast01.mp3"),
         "HEAD", new TimeSpan(0, 0, 30));              ① HEAD request rather than GET

   var account =
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   account.Credentials.SignRequest(hwr);

   var response = hwr.GetResponse();

   foreach (string header in response.Headers)
   {
      Console.WriteLine("{0} : {1}", header, response.Headers[header]);
   }

   Console.ReadKey();
}
```

In the code example shown in listing 9.9, you make the same request that you've made before, but it's a `HEAD` request instead of a `GET` request. You're making a `HEAD` request for the file podcast01.mp3 in the ChrisOriginals container in your development storage account ❶. At ❷ you loop through all the returned headers, outputting the header key and value to the console screen.

USING THE STORAGECLIENT LIBRARY

In listing 9.9, you retrieved the last modified time using the REST API directly, although you could've used the StorageClient library. The following code performs a `HEAD` request that returns the last modified time:

```
blob.FetchAttributes();
var lastModifiedTime = blob.Properties.LastModifiedUtc;
```

`FetchAttributes` is the StorageClient library equivalent of `HEAD`. It returns all the properties and custom metadata of the BLOB, without downloading the actual file. Table 9.1 lists the BLOB properties and their descriptions.

Table 9.1 BLOB properties

Property name	Description
`BlobType`	`PageBlob, BlockBlob`
`CacheControl`	Allows you to instruct the browser on how to cache the BLOB
`ContentEncoding`	Encoding of the header
`ContentLanguage`	Language header of the BLOB
`ContentMD5`	MD5 hash of the content header
`ContentType`	The MIME type of the BLOB
`Etag`	Unique identifier of the request (changes every time a BLOB is modified)
`LastModifiedTimeUtc`	Last time the BLOB was modified
`LeaseStatus`	Used with page blocks: `Locked, Unlocked`; not available in the StorageClient library.
`Length`	Size of the BLOB, in bytes

Now that you're familiar with BLOB properties and how to retrieve them without downloading the file, let's look at how you can use this knowledge to improve the performance of your handler.

9.5.5 Improving your handler to check the last modified time

Because you can retrieve the last modified time, you can modify your handler to check whether the file has changed since you last downloaded it. If it has changed,

you can download another copy of the file. The following listing shows the modified code for your handler.

Listing 9.10 Checking the last modified date

```
blob.FetchAttributes();
   var lastModifiedTime = blob.Properties.LastModifiedUtc;     <-- 1 Gets BLOB properties and metadata

   if (!File.Exists(localFilePath) ||
       File.GetLastAccessTimeUtc(localFilePath) < lastModifiedTime)    <-- 2 Checks whether BLOB has changed
{
    var blobData = GetBlob("chrisoriginals", blobName);
    File.WriteAllBytes(localFilePath, blobData);
    File.SetLastWriteTimeUtc(localFilePath, lastModifiedTime);    <-- 3 Sets last write time
}
```

At (1) you get the last modified time of the BLOB. This method is the same code that you used in section 9.5.4; you can either use the StorageClient library or the REST API directly to retrieve this data.

After you've gotten the last modified time of the BLOB, then you check (2) whether you already have the file locally and whether the local file is older than the server file. The final change that you need to make is that you set the last write time of the local file to the last modified time of the BLOB (3).

So far, we've only looked at the properties of a BLOB. Now let's look at the other part of the BLOB data that's returned in a `HEAD` request: custom metadata.

9.5.6 Adding and returning custom metadata

In listing 9.9, we showed you how to view all the headers associated with a BLOB using a console application. In that example, you returned all the standard headers associated with a BLOB (`last-modified`, `x-ms-request-id`, and so on). If you want to associate some custom metadata with a file, that metadata would also be displayed in the response headers.

DISPLAYING CUSTOM METADATA

Let's modify the BLOB podcast01.mp3 to include the author of the file (me) as custom metadata. Figure 9.9 shows the `HEAD` response in the same console application using the same code that we developed in listing 9.10.

```
file:///C:/AzureBook/Source/PlayArea/AzurePlayArea/WebLoad/bin/Debug/WebLoad.EXE
x-ms-request-id : 2700f84e-b636-45db-b91f-e3131fbf4651
x-ms-meta-author : chris hay
Content-Length : 5128497
Date : Sun, 05 Jul 2009 21:55:38 GMT
ETag : 0x8CBCBD277B8766B
Last-Modified : Sun, 05 Jul 2009 21:55:38 GMT
Server : Development Storage Blob Service Version 1.0 Microsoft-HTTPAPI/2.0
```

Figure 9.9 `HEAD` **response with custom metadata**

Note that with the file podcast01.mp3, there's a new header returned in the response, called `x-ms-meta-author,` with the value `chris hay`. In this example, we returned the metadata from the `HEAD` request, but it's also available in a `GET` request.

SETTING CUSTOM METADATA

If you need to set some custom metadata, you can easily do this with the StorageClient library. The following code sets the metadata for the file podcast01.mp3:

```
blob.Metadata.Add("Author", "Chris Hay");
blob.SetMetadata();
```

This code calls the `UpdateBlobMetadata` method on the container, passing in a `NameValueCollection` of metadata that you want to store against the BLOB.

> **x-ms-meta**
>
> The StorageClient library automatically prefixes all metadata with the tag `x-ms-meta-`. We actually set the key as `author` in the metadata collection, but the response header returned `x-ms-meta-author`. As a matter of fact, the BLOB storage service will only allow you to set metadata with the prefix `x-ms-meta-` and it ignores any attempt to modify any other header associated with a BLOB. Unfortunately, this means you can't modify any standard HTTP header that isn't set by the BLOB storage service.

SCENARIOS FOR USING CUSTOM METADATA

The metadata support for BLOBs allows you to have self-describing BLOBs. If you need to associate extra information with a BLOB (for example, podcast author, recording location, and so on), then this would be a suitable place to store that data; you wouldn't have to resort to an external data source. Any custom attributes associated with a file (such as the author of a Word document) could also be stored in the metadata.

TIP If you need to be able to search for metadata across multiple BLOBs, consider using an external data source (for example, a database or the Table storage service), rather than searching across the BLOBs.

Now that you know how the upload and download operations operate under the covers, let's return to the final part of the uploading and downloading puzzle, namely, copying BLOBs.

9.6 Copying BLOBs

So far in this chapter, you've uploaded files and downloaded files in and out of your account. But you don't always want to transfer files outside your account; sometimes you might want to take a copy of an existing file in the account.

Let's return to the podcasting example for a lesson on why you might want to do this. Let's say that during the conversion of a WMA file to MP3, you decide that you don't want

to make the converted file immediately available. In this case, your converted MP3 file would reside in a private container that isn't available to the general public. At a later date, you decide to make the file available to the public by moving it into your public downloads container. To do this, you copy your converted file from one container to another. Figure 9.10 shows a file being copied from one container to another.

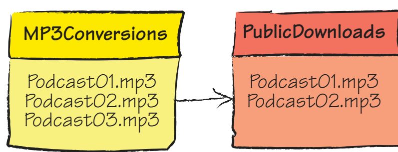


Figure 9.10 Copying podcast03.mp3 in the MP3Conversions container to the PublicDownloads container

In figure 9.10, the BLOB isn't being uploaded or downloaded; the BLOB podcast01.mp3 is just being copied from the MP3Conversions container and the copy is being placed in the PublicDownloads container.

Although you could do this using a download followed by an upload, this would be much slower than just doing an internal copy within the data center. Likewise, performing an upload followed by a download would be incredibly slow and wasteful of network resources if the calling client was based outside the Windows Azure data center.

Listing 9.11 shows the code used to copy a very large filename podcast03.mp3 to the PublicDownloads container from the MP3Conversions container via the REST API.

Listing 9.11 Copying a BLOB between containers

```
    HttpWebRequest hwr = CreateHttpRequest(
       new Uri(@"http://127.0.0.1:10000/devstoreaccount1/publicdownloads/
       ➥ podcast03.mp3"),
       "PUT", new TimeSpan(0, 0, 30));                    <-- 1 PUT request to destination container

    hwr.Headers.Add("x-ms-copy-source",
       "/silverlightukstorage/mp3conversions/podcast03.mp3"); <-- 2 Source location of file

 var account =
    CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

 account.Credentials.SignRequest(hwr);

 hwr.GetResponse();
```

As you can see in listing 9.11, the basics of making the HTTP request is pretty much the same as any other REST request you've made. In this case, you're setting the destination container ❶ as the URI for the request, and you're setting the request to be a `PUT` (rather than a `GET`, `HEAD`, or `POST`).

At ❷ you set the location of the source BLOB to be copied. The header `x-ms-copy-source` is where you define the location of the file; notice that we're including the storage account name (`silverlightukstorage`), the container (`mp3conversions`), and the BLOB name (`podcast03.mp3`) in the header value.

9.6.1 Copying files via the StorageClient library

Listing 9.11 uses the REST API directly to copy the BLOB, but you could make this call by making the simpler StorageClient library call:

```
var sourceBlob = sourceContainer.GetBlobReference("podcast01.mp3");
var targetBlob = targeCcontainer.GetBlobReference("podcast01.mp3");
targetBlob.CopyFromBlob(sourceBlob);
```

This code makes the same call as in listing 9.11, this time using the StorageClient library instead of using the REST API directly.

NOTE Although you can use the REST API, using the StorageClient library is much easier. Save yourself a lot of heartache and use the REST API only when necessary. Try to stick to using the lovely StorageClient library.

Snapshotting BLOBs

The problem of copying BLOBs is that you have to pay storage costs to the Microsoft bean counters for keeping duplicate copies of the data. If you're only copying the BLOB to maintain some sort of version control, you should consider snapshotting instead. A snapshot pins a version of your BLOB at the date and time you created the snapshot. The snapshot is read only (it can't be modified) and can be used to revert to earlier versions of a BLOB. Only changes made between versions of snapshots are chargeable.

To create a snapshot, you can make the following call:

```
var snapshotBlob = blob.CreateSnapshot();
```

To retrieve a snapshot via the REST API, you can use the following URI:

http://accountname.blob.core.windows.net/containername/blobname?snapshot=<DateTime>

You can also retrieve a snapshot using the StorageClient library:

```
var snapshotBlob = container.GetBlobReference("blobname?snapshot=<DateTime>");
```

For more details about snapshotting, go to http://msdn.microsoft.com/en-us/library/ee691971.aspx.

9.7 Setting shared access permissions

Access to a BLOB is controlled by the container that it lives in. If the BLOB lives in a public container, it's available to the world. If the BLOB lives in a private container, you can access it only with your private authentication key.

WARNING Don't distribute your private authentication key. Doing so is a surefire way to have some evildoer trash your data.

These levels of access are a little too extreme; we need a more granular way of controlling access to our BLOBs, namely Shared Access Signatures. Using shared access, you can set a policy on a private container (or BLOB), and anyone who makes a request with the correct signature can perform the appropriate action on the BLOB (say, download the BLOB).

Although you can assign permissions at an individual BLOB level, this is a pain to maintain. It's easier to maintain permissions at a container level (you can always have a container that consists of a single BLOB).

Let's now return to the podcast example and look at how you can control download access to one of your podcasts.

9.7.1 Setting shared access permissions on a container

Let's say your podcasting business has gone well and you've decided to start selling some of your podcasts to the general public. In this scenario, after some rich dude has purchased the podcast, you need to provide a way for them to download the podcast without making it public (you obviously don't want to give them your owner authentication key). To achieve this, you're going to store your podcast (podcast03.mp3) in its own private container (Podcast03), which isn't available to the general public.

After your customer has purchased the podcast, you'll generate a Shared Access Signature that will give that customer permission to read any BLOBs (in this case, podcast03.mp3) in the Podcast03 container, for a period of 24 hours. After the 24-hour period has expired, the customer will no longer be able to download the podcast.

The first thing you need to do is generate a shared access policy that will restrict the download period to the next 24 hours, using the following code:

```
var oneDayDownloadpolicy = new SharedAccessPolicy();
oneDayDownloadpolicy.SharedAccessStartTime = DateTime.Now;
oneDayDownloadpolicy.SharedAccessExpiryTime = DateTime.Now.AddDays(1);
oneDayDownloadpolicy.Permissions = SharedAccessPermissions.Read;
```

As shown in the code, you can specify both a start time and an expiry time for the policy. If you don't specify a start time, the value `now` is substituted as a default. After you've specified this policy, apply it to the container.

```
var permissions = new BlobContainerPermissions();
permissions.SharedAccessPolicies.Clear();
permissions.SharedAccessPolicies.Add("CustomerA", oneDayDownloadPolicy);
container.SetPermissions(permissions);
```

Finally, you can generate a URI that customers will be able to use to download the BLOB, using the following code:

```
string sharedAccessSignature = container.
➥ GetSharedAccessSignature(oneDayDownloadpolicy);
string uri = blob.Uri.AbsoluteUri + sharedAccessSignature;
```

The generated URI will look something like this:

```
https://chrishayuk.blob.core.windows.net/podcast03/podcast03.mp3?st=
➥ 2010-01-04T12%3A08%3A00Z&se=2010-01-05T12%3A08%3A00Z&sr=
    ➥ b&sp=r&sig=ByfV3a1SXOXT04G4GF%2FNQo%2B9cxx4vrRE45kYxbhFhJk%3D
```

And that's about it; you can now dynamically assign permission to read BLOBs that are in containers.

> **Assigning other types of permissions**
>
> What if you want to be able to assign permissions at a BLOB level (rather than at a container level) or if you want to provide more than just read permissions? You can generate Shared Access Signatures that give users permissions to write to certain BLOBs in your container. This scenario is a little too detailed for what we would like to show in this book, but feel free to visit the online documentation for more details at http://msdn.microsoft.com/en-us/library/ee395415.aspx.

9.8 Summary

And....breathe. It's fairly safe to say we've covered how to manage BLOBs stored in BLOB storage in great depth.

In this chapter, we built upon the knowledge you gained in chapter 8 and looked at how you can upload and download files using the StorageClient library. You know how much the StorageClient library can do for you. We figure that the real take-away from that section is that you shouldn't use the REST API and you should stick to the StorageClient library; otherwise, you have to worry about blocks and retries.

Not only did we look at how to use BLOB storage, but you also learned how you can integrate with your ASP.NET website solutions. BLOB storage is the ideal place to store your ASP.NET website assets. It's perfectly suited to providing controlled access to content you want to protect, such as paid MP3 files.

We used two different methods of serving up content: the HTTP handler and the container-level access policy. Although you should probably use the container-level access policy whenever possible (it's cheaper), the techniques that you learned when you built the HTTP handler are invaluable for synchronizing access between local storage and BLOB storage.

In chapter 8, we focused on managing accounts and containers. In this chapter, we showed you how to work with the actual BLOBs. Not only did we look at how to develop against the BLOB service, but we also showed you how you can use the service with your ASP.NET websites.

In the next chapter, we'll build on your BLOB knowledge and look at how you can use the BLOB storage service without a web role, worker role, or server, and run it as a standalone service. We think you'll agree that it opens up some interesting scenarios.

10 When the BLOB stands alone

This chapter covers

- Hosting static websites in BLOB storage
- Hosting Silverlight applications in BLOB storage
- Hosting website assets in BLOB storage
- Using BLOB storage to progressively download video

Although BLOB storage is generally used as a durable storage area for web and worker roles, it can also be used as a standalone service; you can use the service for your applications without hosting a worker or web role.

So far in this book we've focused on using web roles to host websites. Now we're going to tell you how you can use BLOB storage to host a static HTML website without needing a web role.

10.1 Hosting static HTML websites

You learned in chapter 9 that BLOBs held in a public container are accessible to the outside via a public URI over an HTTP connection. Those files can be accessed with

standard web browsers such as Internet Explorer. BLOB storage also lets you configure the MIME types associated with contained files; the web browser can correctly handle the served content. These capabilities make BLOB storage not just a networked hard disk but rather a full-fledged web server farm (as shown in figure 10.1).

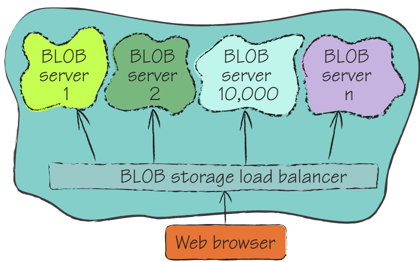


Figure 10.1 BLOB storage acting as a web farm (look, no web roles)

BLOB storage can serve up all modern content types (including standard HTML pages, JavaScript files, CSS files, images, movies, Word documents, PDF documents, Silverlight applications, and Flash applications), making BLOB storage a viable consideration for hosting static websites.

> **Why would I want to use BLOB storage rather than a web role?**
>
> Although web roles are the recommended solution for most scenarios, in some instances it might be more cost effective to use BLOB storage to host your website. If your website is a standard static HTML website that doesn't perform any server-side processing, then it's a suitable candidate for hosting in BLOB storage.
>
> For example, informational sites such as local business websites, static corporate websites, and Silverlight (or Flash) games are good candidates for BLOB-storage hosting. You might conclude that a static website is a fairly uncommon scenario, but it's far more common than you might think. For example, www.sagentia.com is a typical corporate website. Apart from a search facility (which could be implemented using Google and JavaScript), there are no interactive parts to this website. There's no reason why this website couldn't be hosted in BLOB storage rather than using ASP.NET or JSP.
>
> Quite simply, if you want a cheap method of hosting websites in a scalable fashion and you don't need to dynamically generate pages, you can save yourself the cost of web roles and just pay per request.

In this section, we'll show you how BLOB-storage hosting works by showing you how to create and then publish a simple static HTML website. We'll also look at a couple of issues related to using BLOB storage as a host. Now let's get to it and create a static website that you can host in BLOB storage.

10.1.1 Creating a static HTML website

The website that you're about to build is a simple HTML page with a JavaScript calculator, as shown in figure 10.2. When we say calculator, it really just adds two numbers together; abacus++ is probably a more accurate term.

After the user enters two values and clicks the equals button, the result appears (with a gratuitous smiley face graphic), as shown in figure 10.3.

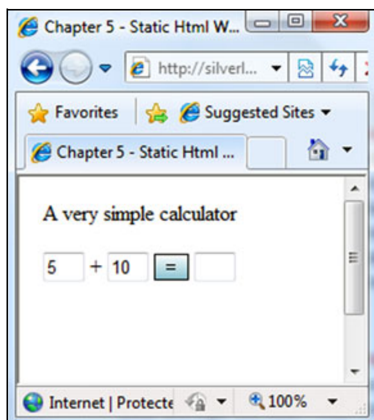


Figure 10.2 A simple static HTML website hosted in BLOB storage

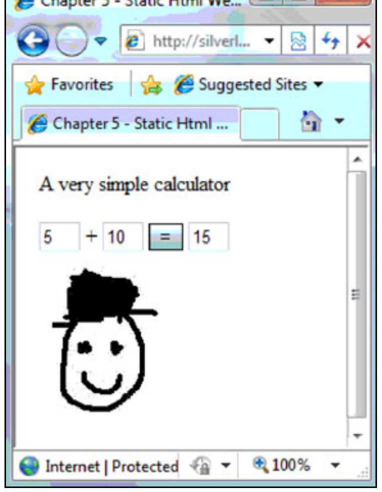


Figure 10.3 On click of the equals button, the result is displayed (along with a badly drawn smiley face)

Although this web page is a simple example, it does consist of all the major parts of a typical website: HTML, CSS, JavaScript, and assets (such as images). Table 10.1 gives a quick overview of the contents of this site.

Table 10.1 A breakdown of all the files in the website shown in figure 10.3

Website file	Purpose
default.htm	HTML page with buttons, text boxes, and stuff.
standard.css	All good web designers use CSS for their markup.
calculator.js	The complicated JavaScript that adds two numbers together.
happy.jpg	The smiley face that appears in figure 10.3.

That's what makes up this website. Now let's take a look at some of the code. The listing that follows contains the HTML for this page.

Listing 10.1 Calculator page website HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
➥ Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
➥ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Chapter 10 - Static Html Website hosted in Blob Storage</title>
    <link rel="stylesheet" type="text/css" href="standard.css" />      Reference to CSS file
    <script type="text/javascript" src="calculator.js"></script>       Reference to JavaScript file
    <script type="text/javascript">
        function calculate() {                                         Calculates the value (1)
            document.getElementById('result').value =
                add(
```

```
                    document.getElementById('number1').value,
                    document.getElementById('number2').value
                   );

            document.getElementById('piccy').className = "visible";
        }
    </script>
</head>
<body>
    <div class="standarddiv">
        A very simple calculator
    </div>
    <div class="standarddiv">

        <input id="number1" type="text"
               value="5" class="standardtextbox"/>

        <span>+</span>
        <input id="number2" type="text"
               value="10" class="standardtextbox"/>

        <input type="submit" value="="
               class="standardbutton"
               onclick="calculate();"/>                    ② Calculate button

        <input id="result" type="text"
               class="standardtextbox"
               readonly="readonly"/>

        <div id="piccy" class="hidden">                    ③ Happy picture
            <img src="happy.jpg"/>
        </div>
    </div></body>
</html>
```

We're not going to explain this code in much detail. It's fairly simple HTML and this is chapter 10; if you've gotten this far, you've probably figured out HTML already. We'll cover the important bits, though. On click of the equals button at ② in listing 10.1, the JavaScript function at ① is invoked. This function takes the contents of the two text boxes (`number1` and `number2`), calls the add function, and displays the result in the result text box. Finally, when the calculation is complete, the style at ③ is changed from `hidden` to `visible` and the smiley face graphic as seen in figure 10.3 is displayed.

Now that we've pushed technology to its absolute limit by building a web page that can add numbers together (woo-hoo, go us), you're going to publish your work of art to BLOB storage for hosting.

10.1.2 Publishing your website to BLOB services

In previous examples, you've programmatically added your BLOBs to the storage service. In this example, you'll make use of a tool called Chris Hay's Azure Blob Browser (can you guess from the title who created it?) to upload the files to BLOB storage. Figure 10.4 shows the addition of the calculator.js file to BLOB storage via the tool.

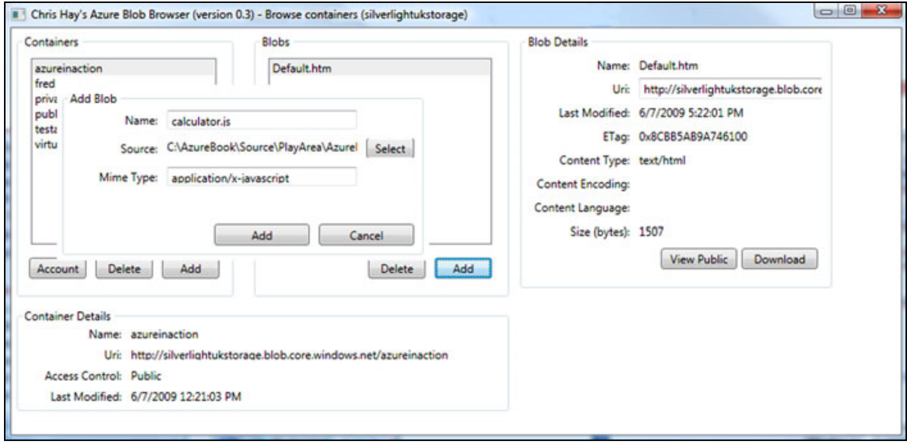


Figure 10.4 Adding calculator.js to BLOB storage via Chris Hay's Azure Blob Browser

Using the BLOB browser, you can add each file of the website (default.htm, calculator.js, standard.css, and happy.jpg) by selecting the file to upload, setting the MIME type (which is automatically predicted for you, but is also editable), and clicking the Add button.

After all the files are uploaded into the appropriate public container, you can access the website with a standard web browser using the following URI structure: http://silverlightukstorage.blob.core.windows.net/azureinaction/default.htm.

Note that you must include the filename (default.htm) because BLOB storage doesn't support default documents.

> **TIP** If you're hosting your website or web application in BLOB storage, you probably don't want your clients to use the BLOB.core.windows.net domain. You need to host your website on your own custom domain, for example, http://www.noddyjavascriptcalculator.com/. For details about how to do this, see chapter 8.

SETTING THE CORRECT MIME TYPE

If the correct MIME type isn't set, the web browser won't be able to render the downloaded file. Some browsers, such as Internet Explorer, will render the content in some situations even if the MIME type is incorrect; other browsers, such as Firefox, require the MIME type to be set explicitly.

Figure 10.5 shows how Firefox handles HTML documents that don't have the correct MIME type (text or HTML) but instead use the default Azure BLOB storage services MIME type (application/octet-stream). Figure 10.5

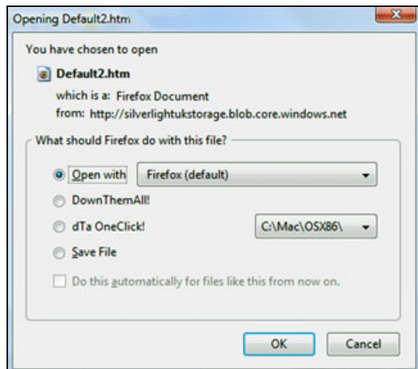


Figure 10.5 Accessing a file that has an incorrect MIME type in Firefox

shows that Firefox can't display the page and requires further instructions from the user. If the MIME type is set correctly, Firefox will show the web page that's shown in figure 10.3.

Now that your first static HTML website is running in BLOB storage, let's look at how you can move beyond flat directory structures and store and retrieve files within hierarchies.

HANDLING DIRECTORY STRUCTURE

In our calculator example, we used the following flat directory structure for the files:

- default.htm
- calculator.js
- standard.css
- happy.jpg

Although this kind of structure is useful for small websites, it's limiting for websites with a large hierarchy. BLOB storage technically doesn't support directories within a container, but you can simulate such support by using a slash (/) as a separator in the BLOB name, which creates a hierarchy for the files. The following list shows how you can represent your files in BLOB storage:

- Chapter10/default.htm
- Chapter10/javascript/calculator.js
- Chapter10/css/standard.css
- Chapter10/images/happy.jpg

Using the slash separator in the BLOB name means that the default.htm page would be accessed using the following URI structure: http://silverlightukstorage.blob.core.windows.net/azureinaction/Chapter10/default.htm.

The `CloudBlobDirectory` class in the StorageClient library provides a way around the limitation of containers not having subcontainers (or subfolders) by providing a set of methods that allows you to pretend that there's a real directory structure in a container. It's driven by the slashes you put in your BLOB filename.

Now that you've made this example website, you can tell that using BLOB storage as a web server is pretty cool and simple. There is, however, one drawback.

LOGGING BLOB REQUESTS

As of version 1, BLOB requests aren't logged. This level of logging is vital for analyzing who's visiting your website, where they've come from, and what parts of your site are popular. If you're used to running your own website, you usually have access to your IIS logs, which contain information such as page requested, IP address, referrer, and so on. Because the BLOB storage service doesn't provide such information, if you're going to use it to host HTML websites, you need to use an external analytics service, such as Google Analytics.

In this section, we showed you how to display static HTML websites in BLOB storage services. Next, we'll look at hosting Rich Internet Applications (RIAs), specifically Silverlight applications.

10.2 Hosting Silverlight applications in BLOB storage

Although you'll usually host your Silverlight applications in a standard Windows Azure web role, you can host your Silverlight applications in BLOB storage, if required. Depending on your web application, you can effectively use BLOB storage to achieve massive scale for minimal cost. For example, in our podcasting example from the previous chapters, it might make financial sense to have your website hosted in a web role, but to also have a Silverlight media player hosted in BLOB storage. If you run a foreign currency exchange website, you might want to host the main site (which could offer the option to purchase currency) in a web role, but host an exchange rate calculator in BLOB storage.

> **NOTE** In this chapter, we're focusing on Silverlight because it's familiar to .NET developers. Although we're looking exclusively at Silverlight, the techniques discussed in this section are applicable to other RIA solutions (for example, Flash, Flex, Air, and so on).

In this section, we'll look at how you can enable such scenarios by showing you how you can do the following using BLOB storage:

- Host a standalone Silverlight application that's contained in a static HTML page
- Host a standalone Silverlight application that's contained in an ASP.NET web page
- Get your Silverlight application that's hosted in BLOB storage to communicate with external web services
- Store external Silverlight application access files

And now for the details that will enable you to do all this wonderful stuff with BLOB storage.

10.2.1 Hosting the Silverlight Spectrum emulator

If you've developed a standalone Silverlight application that requires no interaction with any backend services, BLOB storage is a cost-effective candidate for hosting your Silverlight application. These types of applications typically include games, tax calculators, and other widgets.

For our next example, we'll show you how to host a small Silverlight application in BLOB storage. The application that you'll host is a Silverlight ZX Spectrum emulator, which was an 8-bit home computer of the 1980s, very like a Commodore 64 (actually, that's not true; it was much worse, but it had spirit). Figure 10.6 shows the emulator running from BLOB storage.

The ZX Spectrum emulator not only allows you to play games from the 1980s, but as you can see from figure 10.6, you can even write BASIC programs in it. Now, we don't want you to lose focus on this book as soon as you load this thing. As cool as Jet Set Willy is, it won't help you deliver your amazing cloud-based application. You'll have to continue to pay attention to learn how to do that.

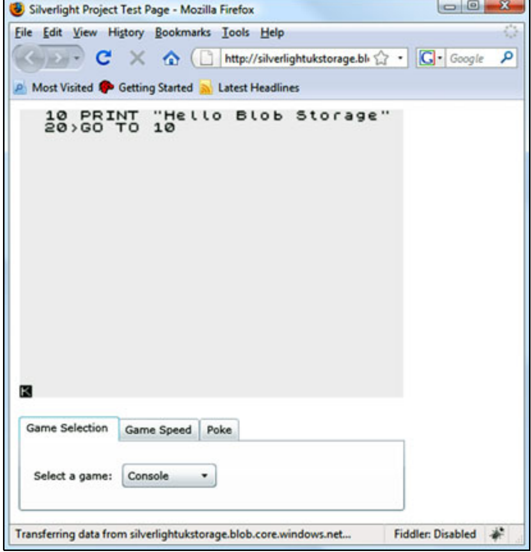


Figure 10.6 Silverlight application running in BLOB storage

To store the application in BLOB storage, upload ZXSilverlight.xap using the same procedure you used in the previous section. (You can download this Silverlight application and its source code from http://www.azureinaction.com.) You must set the BLOB with the correct MIME type; otherwise, the browser won't be able to launch the application. The MIME type for a Silverlight application is `application/x-silverlight-app`.

You're also going to store the HTML page that hosts the Silverlight application in BLOB storage. The following listing shows the HTML that runs the ZX Spectrum emulator Silverlight application (ZXSilverlight.app).

Listing 10.2 HTML for the ZX Spectrum emulator Silverlight application

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Silverlight Project Test Page </title>

    <style type="text/css">
        html, body {height: 100%; overflow: auto;}
        #silverlightControlHost {height: 100%;}
    </style>
    <script type="text/javascript" src="Silverlight.js"></script>
</head>
<body>
     <div id="silverlightControlHost">
            <object data="data:application/x-silverlight,"
                    type="application/x-silverlight-2"
                    width="100%" height="100%">
                    <param name="source" value="ZXSilverlight.xap"/>
                    <param name="background" value="white" />
            </object>
    </div>
</body>
</html>
```

Specifies Silverlight plug-in

Specifies Silverlight 2 application

Specifies location of Silverlight application

By looking at listing 10.2, you can see that the HTML page doesn't have any logic in it. Its job is to host the Silverlight application. Because this web page is just a host for a Silverlight application, using a full-fledged web server would be a little over the top (and expensive); a static HTML page hosted in BLOB storage will do the job perfectly.

> **TIP** If you have an existing web page that you host outside BLOB storage (you're using a web role or an existing web hosting provider to host it), you can still host your Silverlight application in BLOB storage but keep your site with your existing host. To do so, you need to change the `source` parameter of the Silverlight plug-in (see listing 10.2) to point to your BLOB storage URI.

This standalone Silverlight application doesn't require access to any backend web services, but what if you want to host an application that does? You can still host your application in BLOB services, but you'll need to understand how a cross-domain policy works.

10.2.2 Communicating with third-party sites

Suppose your Silverlight application requires communication with an external website or web service (WCF, ASMX, REST, POX, or HTTP). For example, what if you want to host a Silverlight exchange rate calculator?

Your Silverlight application needs to poll a web service for live data, so you might be thinking that you should host your entire solution as an ASP.NET-hosted website. Although this is a perfectly valid solution, it's still quite expensive for what you require. In this scenario, it might be more cost effective to host your web service in ASP.NET (or use a third-party service if one is available), but host your Silverlight application (and website) in BLOB storage. Figure 10.7 shows a Silverlight application communicating with an external web service.

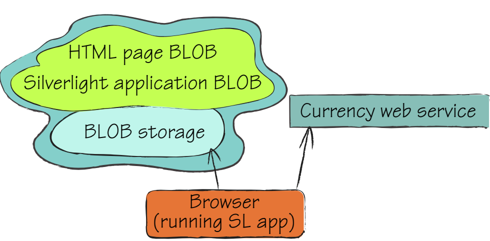


Figure 10.7 Silverlight application communicating with an external web service

For the Silverlight application to communicate with a third-party domain, the external site must host a suitable cross-domain policy.

HOSTING A CROSS-DOMAIN POLICY

The following listing shows a typical cross-domain policy file (ClientAccessPolicy.xml) used to give permissions to a Silverlight application.

Listing 10.3 ClientAccessPolicy.xml file for Silverlight permissions

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
```

```
        <domain uri="http://silverlightukstorage.blob.core.windows.net/"/>   ◁─ 1 URI of BLOB storage domain
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Cross-domain policy

For security purposes, Silverlight applications (and Flash applications) can, by default, communicate only with the domain that the container web page is hosted on. If your Silverlight application needs to communicate with a third-party domain, the external website needs to host a cross-domain policy file (ClientAccessPolicy.xml or CrossDomain.xml) to give your application permission to communicate with it.

The ClientAccessPolicy.xml file displayed in listing 10.3 states at (1) that any Silverlight application hosted at http://silverlightukstorage.blob.core.windows.net/ (the BLOB storage account of your Silverlight application) can access the third-party web service. Your application would also be able to access the service if the domain URI at (1) was set to *, which would indicate that any Silverlight application hosted at any website could access it.

ClientAccessPolicy.xml is a cross-domain policy file that's used solely by Silverlight applications. Silverlight applications can also access web services that host a CrossDomain.xml file (a format that's supported by both Flash and Silverlight). The following listing shows a CrossDomain.xml file that your BLOB storage-hosted application could communicate with.

Listing 10.4 CrossDomain.xml file

```
<?xml version="1.0" ?>
<!DOCTYPE cross-domain-policy  SYSTEM
➥ "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
    <allow-access-from domain="*" />                ◁─ 1 Allows access to all domains
</cross-domain-policy>
```

The CrossDomain.xml file displayed in listing 10.4 states at (1) that any website hosting a browser-based application (Silverlight or Flash) can communicate with this service. If the owner of the web service wanted to restrict access to your application only, then it would replace the `*` at (1) with `silverlightukstorage.blob.core.windows.net` (your BLOB storage account).

> **$root container**
>
> If you need to allow third-party Silverlight or Flash applications to access assets held in your BLOB storage account, store your CrossDomain.xml or ClientAccessPolicy.xml file in a public container named $root.
>
> This special container allows you to store any file in the root of your URI, for example: http://chrishayuk.blob.core.windows.net/crossdomain.xml.

Now let's build a Silverlight web search application that uses Yahoo's Search API to search the internet.

BUILDING A SILVERLIGHT WEB SEARCH APPLICATION

Figure 10.8 shows the Silverlight application that we're going to show you how to build.

The HTML page for the application that you're going to build (shown in figure 10.8) is no different from the one in listing 10.2 (well, there is one small difference: the `source` parameter value references the new Silverlight application (.XAP file).

To create the application, create a new Silverlight application as you normally would; then rename the default XAML (Extensible Application Markup Language) to YahooSearch.xaml. Replace the default `Grid` provided in the template with the XAML shown in the following listing.

Listing 10.5 XAML for Silverlight web search application

```
<Grid x:Name="LayoutRoot" Background="White">
        <StackPanel>
            <StackPanel Orientation="Horizontal">
```

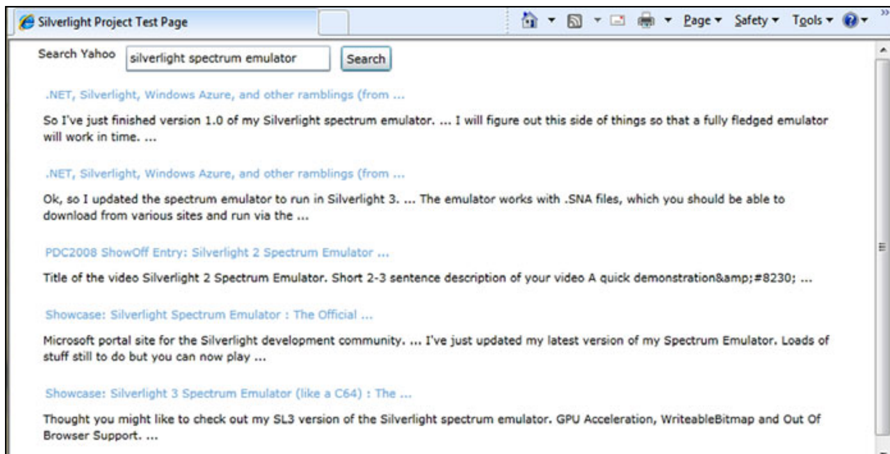


Figure 10.8 A Silverlight search application hosted in BLOB storage that communicates with the Yahoo Search API.

```
            <TextBlock  Text="Search Yahoo"
                        VerticalAlignment="Top"
                        Margin="5"/>
            <TextBox x:Name="txtSearch"                          Enters the
                                VerticalAlignment="Top"          search term
                                Height="25" Width="200"
                                Margin="5"/>
            <Button x:Name="btnSearch"
                    Content="Search"
                    VerticalAlignment="Top"
                    Height="25" Width="50"
                    Margin="5"
                    Click="btnSearch_Click"/>
        </StackPanel>
            <ItemsControl x:Name="itemsResults">                 Displays the
                <ItemsControl.ItemTemplate>                      search results
                    <DataTemplate>
                        <Grid Margin="10">
                            <Grid.RowDefinitions>
                                <RowDefinition Height="25"/>
                                <RowDefinition Height="*"/>
                            </Grid.RowDefinitions>
                            <HyperlinkButton
                               Grid.Row="0"
                               Content="{Binding Title}"
                               NavigateUri="{Binding Url}"/>
                            <TextBlock
                               Grid.Row="1"
                               Text="{Binding Summary}"
                               TextWrapping="Wrap">
                            </TextBlock>
                        </Grid>
                    </DataTemplate>
                </ItemsControl.ItemTemplate>
            </ItemsControl>
        </StackPanel>
    </Grid>
```

After you've pasted the XAML in listing 10.5 into your Silverlight page, you can hook up the code-behind for the page, as shown in listing 10.6.

Listing 10.6 Code-behind for Silverlight web search application XAML page

```
public partial class YahooSearch : UserControl
{
        public YahooSearch()
        {
            InitializeComponent();
        }

        private void btnSearch_Click(object sender, RoutedEventArgs e)
        {
            Search mySearch = new Search();
            mySearch.SearchResultsReturned += new
            ➥ Search.SearchResultReturnedDelegate
            ➥ (mySearch_SearchResultsReturned);
```

```
            mySearch.Execute(txtSearch.Text);          <-- 1 Sends search request to Yahoo API
        }

        void mySearch_SearchResultsReturned(SearchResultSet results)
        {
            itemsResults.ItemsSource = results.Results;   <-- 2 Binds results of search to data grid
        }
}
```

In the code-behind for the Silverlight page shown in listing 10.6, you can see that when the Search button is clicked, the `execute` method of your `Search` class is invoked at (1). The search term is sent to the Yahoo Search API, and then the results are returned asynchronously. When the search results are received, the results are bound to the data grid at (2).

Yahoo Search

Yahoo was a popular search engine many years ago, at a time before we Googled for everything. Here at *Azure in Action*, we like to support the little guy; come on Yahoo, you can be big again.

Actually, the reason we use Yahoo is that it has a nice, simple, REST-based API that we can easily use from Silverlight.

Listing 10.7 shows the code-behind for the Yahoo `Search` class.

Listing 10.7 Code-behind for the Yahoo `Search` class

```
public class Search
{
    public delegate void SearchResultReturnedDelegate
    ➥ (SearchResultSet results);
    public event SearchResultReturnedDelegate SearchResultsReturned;

    private const string _baseUri =
    ➥ "http://search.yahooapis.com/WebSearchService/
    ➥ V1/webSearch?appid={0}&query={1}&start={2}&results={3}";
    private const string _applicationId =
    ➥ @"Jk.kKH_V34FIj5WGgtm6iimK.37hiKF
    ➥ 0A5EVnJkoltzGoydU.Z0notpjqa0DxTiJULlbg--";

        public void Execute(string keyword)
        {

            Uri searchUri = new Uri(string.Format
            ➥ (_baseUri, _applicationId, keyword,1,5));

            Execute(searchUri);                     <-- Calls the search
        }

        private void Execute(Uri address)
        {
```

```
            WebClient searchClient = new WebClient();

            searchClient.DownloadStringCompleted +=
            ➥ new DownloadStringCompletedEventHandler
            ➥ searchClient_DownloadStringCompleted);

            searchClient.DownloadStringAsync(address);
        }

        void searchClient_DownloadStringCompleted
               (object sender, DownloadStringCompletedEventArgs e)
        {

            if (SearchResultsReturned != null)
            {

                XElement xeResult = XElement.Parse(e.Result);

                                    SearchResultSet resultSet =
                                    ➥ new SearchResultSet(xeResult);

                SearchResultsReturned(resultSet);    <-- Returns the search results
            }
        }
}

public class SearchResultItem
{
    public string Title { get; set; }
    public string Summary { get; set; }
    public Uri Url { get; set; }
}

public class SearchResultSet
{

    public SearchResultSet()
    {

        Results = new List<SearchResultItem>();
    }

    public SearchResultSet(XElement resultsXml)
    {

        Results = new List<SearchResultItem>();


        XNamespace ns = "urn:yahoo:srch";

        var xeResults = from xeResult in
        ➥ resultsXml.Elements(ns + "Result")    <-- LINQ to reform the results
             select new SearchResultItem
             {
```

```
                    Title = xeResult
                    ➥ .Element(ns + "Title").Value,
                    Summary = xeResult
                    ➥ .Element(ns + "Summary").Value,
                    Url = new Uri(
                    ➥ xeResult.Element(ns + "Url").Value)
                  };

            Results.AddRange(xeResults);
        }

        public List<SearchResultItem> Results { get; set; }

    }
```

We're not going to spend any time explaining the code in listing 10.7, except to say that it makes an HTTP request to the Yahoo Search API and returns a set of results that you can bind to your Silverlight data grid.

Now that you have an idea of how to store static HTML websites and Silverlight applications in BLOB storage, let's look at how you can use BLOB storage as a media server.

10.3 Using BLOB storage as a media server

In this section, we'll return to our podcasting example. Previously, you've used BLOB storage as a place to store your video and audio podcasts; now you're going to use BLOB storage as a mechanism for serving your videos to customers.

Because BLOB storage provides a URI for any files held in public containers, you could just make the link available to your customers to download the files offline, for example, http://storageaccount.blobs.core.windows.net/container/mypodcast.wmv. This probably wouldn't provide the greatest user experience in the world.

An alternative to downloading an entire media file is to use streaming. When media is streamed, the streaming server starts sending a byte stream of the video to the client. The client media player creates a buffer of the downloaded bytes, and starts playing the video when the buffer is sufficiently full. While the user is watching the video from the buffer, the client continues to download the data in the background.

Media streaming lets the user start watching the video within seconds, rather than requiring the user to wait for the entire movie to download. If the user decides to watch only the first few seconds of the movie, the service provider will have served up only some of the movie, which results in cheaper bandwidth bills. Unfortunately, streaming isn't currently available as an option in Windows Azure and the Windows Azure BLOB storage service. If you want to use such a solution, you need to use a third-party streaming service.

An even better solution is to use progressive downloading, in which a file is downloaded in small chunks and is stitched together by the client application. After a few chunks are downloaded, the client application can start playing the movie while the rest of the file chunks are downloaded in the background. Progressive downloading

has the same performance advantages as streaming and provides a similar user experience. The main difference between progressive downloading and streaming is that the file being streamed never physically resides on disk and remains only in memory.

10.3.1 Building a Silverlight or WPF video player

By default, Silverlight supports the ability to progressively download files. In this example, we'll tell you how to build a small Silverlight video player that will allow you to play movies on your web page that are hosted by BLOB storage. Figure 10.9 shows a small video player that you'll build. The video player is playing a video served directly from BLOB storage using the public URI.

You can build a WPF or Silverlight media player like that shown in figure 10.9 by creating a WPF or Silverlight application and using the following XAML:

```
<MediaElement x:Name="myVideo"
➥ Source="http://storageaccount.blobs.core.windows.net
➥ /container/videopodcast01.wmv"
/>
```

Wow, is that really it? Yup, that's how easy it is to create a progressive downloading video player that shows videos hosted in BLOB storage.

Although Silverlight does progressively download the video player, the download is performed in a linear fashion, downloading from a single file, from a single website. If you have very large files (they're up to gigabytes), you might want to use a slightly different technique. For large files, you can download video much faster if you split up and chunk your files manually (and eventually split them across multiple servers).

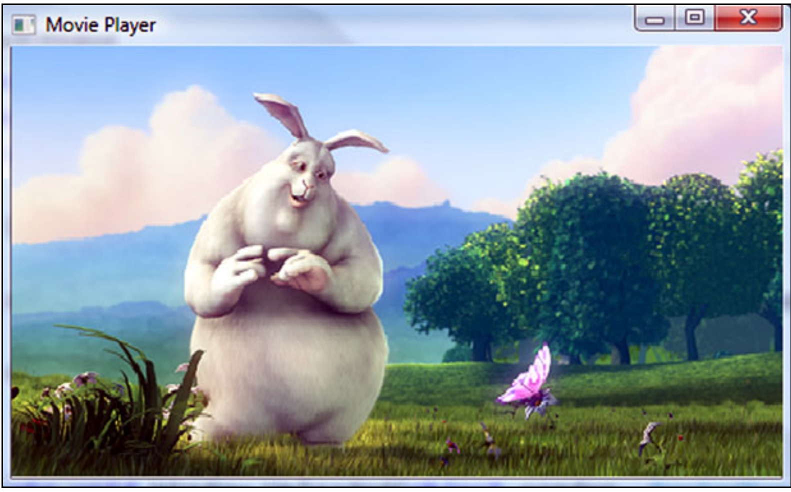


Figure 10.9 WPF video player showing a video that features a giant bunny rabbit (Big Buck Bunny)

10.3.2 A WPF-based adaptive-streaming video player

Now we're going to tell you how to build a WPF-based media player that can adaptively stream your video. Your media player will look exactly like the WPF media player displayed in figure 10.9.

Adaptive streaming is a technique that most content delivery networks (CDNs) use to deliver high-performance video. This technique is also used in IIS adaptive streaming, although BLOB storage won't deliver in multiple bitrates.

> **WARNING** The following code is intended as an example to show you how you can use adaptive streaming. This isn't production-quality code (not by a long shot).

PLAYING THE VIDEO

The following XAML is used to play the video in WPF:

```
<MediaElement x:Name="myVideo" Source="videopodcast01.wmv"
              LoadedBehavior="Manual"/>
```

`MediaElement` is a built-in control that allows you to play movies in WPF applications. In this example, we're downloading the movie (mymovie.wmv) to the same folder that the WPF movie player application resides in. The source attribute states where the `MediaElement` should look for the movie.

By default, the `MediaElement` automatically starts playing the movie on startup. Because the movie hasn't been downloaded yet, you need to prevent the movie from automatically playing by setting the attribute `LoadedBehavior` to `Manual`.

THE CHUNKING METHODOLOGY FOR WPF

You're going to download the movie in chunks of 100 Kb. Only one chunk will be downloaded at a time and each newly downloaded chunk will be appended to the previously downloaded chunk. Listing 10.8 shows that as soon as the movie player is loaded, it starts downloading the chunks. After 10 seconds, the movie starts playing.

Listing 10.8 Chunking movies with the WPF movie player

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   movieStream =
      File.Open("videopodcast01.wmv",
         FileMode.Append, FileAccess.Write, FileShare.ReadWrite);

GetNextChunk();
Thread.Sleep(10000);
myVideo.Play();
}
```

Creates empty movie file

Waits 10 seconds

Before we look more carefully at the code used to progressively download the video, let's talk a bit about the `Range` header.

USING THE RANGE HEADER

When a GET request is made using either the storage client or via an HTTP request, the entire file is downloaded by default. The code shown in listing 10.9 will download videopodcast01.wmv from the public container, podcasts.

Listing 10.9 Using the storage client to download a video file

```
  CloudStorageAccount account =
     CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

CloudBlobClient blobClient =
   account.CreateCloudBlobClient();

CloudBlobContainer container =
   blobClient.GetContainerReference("podcasts");          <- Gets podcasts container

container.GetBlobReference("videopodcast01.wmv");        <- Gets reference to BLOB videopodcast01.wmv
container.DownloadFile("videopodcast01.wmv");             <- Downloads videopodcast01.wmv
```

The code in listing 10.9 will download the entire movie; in this example, we want to split the movie up into manageable chunks. Currently, the storage client sample code doesn't provide the ability to download a specified portion of the file, even though the underlying REST API does support this. The code shown in listing 10.10 will download the entire podcast using HttpWebRequest.

Listing 10.10 Using HttpWebRequest to download a video file

```
string baseUri = @"http://silverlightukstorage.blob.core.windows.net/";
HttpWebRequest hwr =                                   <- (1) Generates HttpWebRequest
    CreateHttpRequest(new Uri(baseUri +"podcasts/videopodcast01.wmv"),
                           "GET", new TimeSpan(0, 0, 30));
// TODO: Range Header goes here                         <- (2) Restricts download size using Range header
DownloadFile(hwr," videopodcast01.wmv");  <- (3) Downloads the file
```

At (1), you generate the HttpWebRequest with the standard required headers using the CreateHttpRequest method that we used in chapter 8. At (3), you use the DownloadFile method to invoke the request and download the file to disk. The implementation of the DownloadFile method is available in the online samples.

If you don't want to download the entire video file, you can use the Range header to specify the range of bytes that you want to download. The following code would restrict the download to the first 100,000 bytes of the file, and can be used in listing 10.10 at (2):

```
hwr.AddRange(0, 100000);
```

Not only can you restrict the number of bytes using the Range header, you can also use it to progressively download the file in chunks.

DOWNLOADING CHUNKS OF DATA

When the WPF movie player was loaded, we made a call to `GetNextChunk()` in listing 10.8. In listing 10.11, we show you how `GetNextChunk` is implemented, and how to use the `Range` header to progressively download the movie.

Listing 10.11 Downloading the data in chunks

```
private int size 100000;
private int nextSize = 0;
private Stream movieStream;

private void GetNextChunk()
{
string baseUri = @"http://silverlightukstorage.blob.core.windows.net/";
HttpWebRequest hwr =
    CreateHttpRequest(new Uri(baseUri +"podcasts/videopodcast01.wmv"),
                           "GET", new TimeSpan(0, 0, 30));

hwr.AddRange(nextRange, nextRange + size);
nextRange += (size + 1);
hwr.BeginGetResponse(new AsyncCallback(webRequest_Callback), hwr);
}

private void webRequest_Callback(IAsyncResult asynchronousResult)
{
   try
   {
      HttpWebRequest request =
         (HttpWebRequest)asynchronousResult.AsyncState;

      HttpWebResponse response =
         (HttpWebResponse)request.EndGetResponse(asynchronousResult);

      SaveChunk(response.GetResponseStream());
      response.Close();

      GetNextChunk();
   }
   catch { }
}

private void SaveChunk(Stream incomingStream)
{
   int READ_CHUNK = 1024 * 1024;
int WRITE_CHUNK = 1000 * 1024;
byte[] buffer = new byte[READ_CHUNK];

Stream stream = incomingStream;

while (true)
{
```

2 Adds Range header to restrict download size

3 Makes request, providing callback

4 The callback function

5 Saves downloaded file section

```
        int read = stream.Read(buffer, 0, READ_CHUNK);
        if (read <= 0)
           break;
        int to_write = read;

        while (to_write > 0)
        {
           movieStream.Write(buffer, 0, Math.Min(to_write, WRITE_CHUNK));
           to_write -= Math.Min(to_write, WRITE_CHUNK);
        }
     }
  }
```

The `GetNextChunk` method will download 100 Kb of data from the BLOB storage service asynchronously. This method will be called every time you need to download a new chunk of video; it's called for the first time when the application is loaded in the fourth line in listing 10.8.

In listing 10.11, you create a standard HTTP web request at ❶; this is the same method that you used to create requests earlier. The video (videopodcast01.wmv) that you're downloading resides in a public container called *podcasts* that's held in your storage account. After you've created the request, you add the `Range` header ❷ to restrict the download to the next 100 Kb chunk.

At ❸ you're making an asynchronous HTTP web request to the BLOB storage service, with a callback to ❹. On the request callback ❹, the downloaded data is retrieved, and you call the `SaveChunk` method ❺, which will append the downloaded chunk to the videopodcast01.wmv file on the local disk. Finally, you call `GetNextChunk` again to get the next chunk of data.

Time-based buffering

In our simple example, we used time-based buffering to delay the playback of the movie. Although time-based buffering is suitable for our example, in a production scenario you should start playing the movie after a portion of the movie has been downloaded. We also don't make any provision for variable download speeds; you might want to extend the sample to handle situations in which the playback speed is faster than the download speed.

Now you have a working version of a WPF progressive-downloading media player, which shows that you can use BLOB storage as a storage service for desktop clients. With some tweaks you can build media players that perform similarly to a streaming service.

10.3.3 A Silverlight-based chunking media player

In the previous section, we showed you how to build a desktop progressive-downloading media player for video podcasts. Although the desktop application is great for that

richer client experience, we also want to show you how to provide a web-based cross-platform Silverlight experience.

CHUNKING VIDEO IN SILVERLIGHT

Go back to the Silverlight player that you created earlier and modify the XAML to the following:

```
<MediaElement x:Name="myVideo" AutoPlay="False"/>
```

Although the XAML is similar to the WPF version, there are some subtle differences. In the Silverlight version, you use `AutoPlay="False"` to specify that you don't want the video to play automatically; in the WPF version, you set `LoadedBehavior="Manual"`. You're also not going to set a source for the `MediaElement;` you'll do this programmatically in the code-behind.

Listing 10.12 shows the `On_Load` event of the Silverlight media player.

Listing 10.12 `On_Load` event of the Silverlight media player

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
            GetNextChunk();                                   ← 1 Retrieves first chunk of video
            Thread.Sleep(10000);                              ← 2 Delays start of video
            using (IsolatedStorageFile isf =                  ← 3 Starts playing from isolated storage
     IsolatedStorageFile.GetUserStoreForApplication())
                {
                    var iosf = isf.OpenFile("videopodcast01.wmv",
                    ➥ FileMode.Open,
                    ➥ FileAccess.Read,
                    ➥ FileShare.ReadWrite);
                    myVideo.SetSource(iosf);
                    myVideo.Play();
                }
}
```

At 1 the code calls `GetNextChunk` to start downloading the chunks of data and at 2 playing the video is delayed for 10 seconds by sending the thread to sleep. At 3 the video starts playing back from isolated storage.

> **WARNING** Sending the thread to sleep is bad production practice. You should never send the UI thread to sleep. In production, use a background timer that monitors the download progress, and start playing the video back at a suitable threshold.

THE CHUNKING METHODOLOGY FOR SILVERLIGHT

You'll use the same methodology to chunk the data that you used in the WPF example. You're going to split the data into manageable chunks and then stitch the chunks back together client-side.

Restricted headers

The browser plug-in model prevents the use of certain request headers because they're considered reserved and are available only to the browser. The following restricted headers are used by BLOB storage APIs and are unavailable to use in Silverlight: `Authorization` (discussed in the section "Setting shared access permissions" in chapter 9), `Date` (you can use `x-ms-date` as a workaround), and `Range` (you can use `x-ms-range` as a workaround).

Although you could split the files using the `x-ms-range` header (instead of using the `Range` header as you did in the WPF version), it might be kind of interesting to presplit the files into chunks and store the chunks in BLOB storage.

To split the files, use the same code that we used in the WPF example, but instead of stitching the file back together, you're going to save a separate file for each chunk. For example, the file VideoPodcast01.wmv would be split into the following chunks: VideoPodcast01_1.wmv, VideoPodcast01_2.wmv, . . ., VideoPodcast01_12.wmv.

Finally, you need to upload each chunk to BLOB storage using a tool like Chris Hay's Azure Blob Browser.

> **TIP** By presplitting the files, you can potentially distribute the files across a greater number of servers (or even domains). Distributing files in this way would be useful if you decided to use a content delivery network (CDN) (discussed later in this chapter) in combination with BLOB storage. You could also instruct the browser to cache the chunks (see chapters 6 and 9); if the file wasn't fully downloaded, the next download would be much quicker.

To use the presplit version of the chunks, your Silverlight application needs to use the following code to replace the WPF version of the `GetNextChunk` method:

```
private void GetNextChunk()
{
string baseUri = @"http://silverlightukstorage.blob.core.windows.net/";
string videoUri = baseUri+"podcasts/videopodcast01_"+nextRange+".wmv";
nextRange++;
HttpWebRequest hwr =
     CreateHttpRequest(new Uri(videoUri), "GET", new TimeSpan(0, 0, 30));
hwr.BeginGetResponse(new AsyncCallback(webRequest_Callback), hwr);
}
```

This code manually requests the chunk that's stored in your container by continually incrementing the chunk in the URI for each download request.

The major difference between this example and using adaptive streaming is that we don't have multiple encodings of the video at different bitrates. You could easily modify these samples to have videos encoded at different bitrates available and to detect the best bitrate for the client.

SAVING TO ISOLATED STORAGE

Silverlight can directly access the filesystem only via a user-initiated action. Silverlight applications can, however, write data directly to an isolated storage area without user initiation.

> **Saving to the actual filesystem**
>
> If you don't mind user-initiated actions, you can always use the `SaveFileDialog` rather than isolated storage.
>
> You need to be aware that if you're downloading files of any real size, you'll probably need a user-initiated action to increase the isolated storage default limits. In this case, you'd probably do just as well to use the `SaveFileDialog`.

In the Silverlight version of the media player, you're going to use the isolated storage area to save the video locally. Although the video podcast has been presplit into chunks, you still have to combine the chunks into a single file so the media player can play back the video. The method of saving and combining the chunks is similar to that used in the WPF version except that you're using isolated storage. Listing 10.13 shows the Silverlight version of `SaveChunk`.

Listing 10.13 `SaveChunk` for Silverlight

```
private void SaveChunk(Stream incomingStream)
{
int READ_CHUNK = 1024 * 1024;
            int WRITE_CHUNK = 1000 * 1024;
            byte[] buffer = new byte[READ_CHUNK];
            using (IsolatedStorageFile isf =
            ➥ IsolatedStorageFile.GetUserStoreForApplication())
            {
                using (IsolatedStorageFileStream isostream = new
                ➥ IsolatedStorageFileStream("videopodcast01.wmv",
                ➥ FileMode.Append, FileAccess.Write,
                ➥ FileShare.ReadWrite, isf))
                {
                    Stream stream = incomingStream;
                    stream.Position = 0;

                    while (true)
                    {
                        int read = stream.Read(buffer, 0, READ_CHUNK);
                        if (read <= 0)
                            break;
                        int to_write = read;

                        while (to_write > 0)
                        {
                            isostream.Write(buffer,
                            ➥ 0,
                            ➥ Math.Min(to_write, WRITE_CHUNK));
```

Gets isolated storage area for Silverlight application

Appends chunk to video in isolated storage

```
                    to_write -= Math.Min(to_write, WRITE_CHUNK);
                }
            }
            isostream.Close();
        }
    }
}
```

Although you've had to do some slight workarounds, now you've got a working version of the media player in Silverlight.

Congratulations! You've built a WPF video player and a Silverlight video player. Now let's look at how you can improve the performance of the video delivery by using content delivery networks (CDNs).

10.4 Content delivery networks

If your website customers are geographically dispersed, using a CDN can significantly improve the user experience. In this section, we'll discuss CDNs and how you can use them in conjunction with the Windows Azure BLOB storage service.

10.4.1 What's a CDN?

A CDN is a large number of web servers that are distributed across the world. These web servers usually sit close to the internet backbone and can quickly serve up large files. When a user makes a request to the CDN for a file, the CDN figures out which data center in the CDN is closest to the user's location and serves up the content from that data center.

Figure 10.10 shows that if you're based in Edinburgh, your files are served from the Dublin data center, rather than from one in Hong Kong.

In a CDN, a user makes a request via the nearest edge server and the origin server answers the request. Let's look at these servers in more detail.

EDGE SERVERS

Figure 10.10 shows that the CDN network has the following data centers

- Los Angeles
- New York
- Dublin
- London
- Dubai

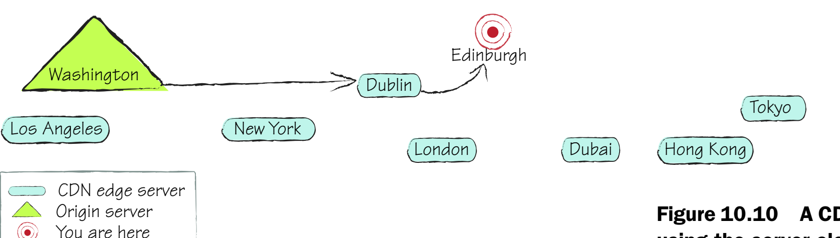


Figure 10.10 A CDN delivers files using the server closest to you.

- Hong Kong
- Tokyo

These data centers, which are represented by circles in figure 10.10, are known as edge servers. An edge server is the name used for one of the geographically dispersed web servers that are responsible for serving your content.

ORIGIN SERVER

The origin server is the web server that contains the original version of the content being distributed. In figure 10.10, the origin server is represented by the triangle and resides in Washington State. For Windows Azure, the origin server could be either your web role or a public container in your BLOB storage account. The content held on your origin server would never be accessed directly by your end user; the origin server only serves content to the edge servers.

When a request is made for a file held in a CDN, the request is redirected to the nearest edge server. If the edge server doesn't have a local copy of that file, it requests the file from the origin server and caches it locally.

> **NOTE** A CDN web server is similar to a BLOB storage web server; it can serve up any static file with the correct MIME type, but it has no backend server processing capabilities. For this reason, a CDN is suitable for serving up static content such as HTML, JavaScript, CSS, Silverlight and Flash applications, PDF documents, audio files, videos, and so on.

10.4.2 CDN performance advantages

Using a CDN network is a simple method of improving performance on your web servers without significantly changing your architecture or code. How so, you ask? Well, read on.

REDUCED LOAD FROM YOUR WEB APPLICATION SERVERS

If your static content is offloaded from your web application servers, your web servers will have more capacity to handle incoming requests. The reduced load will reduce active connections, CPU use, and network traffic on the server.

INCREASED CLIENT-SIDE PERFORMANCE

The largest bottleneck on web applications tends to be the time it takes to download static content from the web server, not the time it takes to serve the HTML page. Figure 10.11 shows the number of requests made to www.manning.com and the length of time it has taken for the content to be served.

If you look carefully at figure 10.11, you'll see that it takes only 1.72 seconds to serve up the HTML page, but an additional 3.5 seconds to serve the static content. This extra time is used because most internet browsers (including Firefox and Internet Explorer) can download only two files simultaneously from the same domain. If more than two files are requested at the same time, all other requests from the same domain are queued. The graph in figure 10.12 shows that the .gif files from manning.com are being downloaded only two files at a time, but the files being served from google-analytics.com can be downloaded at the same time as the manning.com gifs.

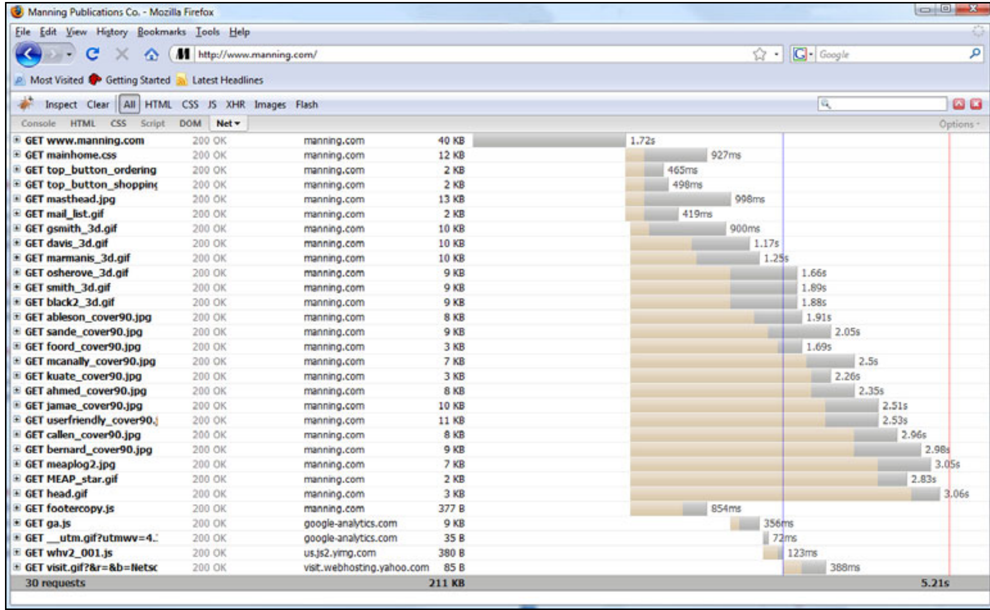


Figure 10.11 Comparison of the number of seconds it takes to serve HTML pages versus the number of seconds it takes to download the static content at www.mannning.com

A CDN moves static content to other subdomains, which allows the browser to simultaneously download more files and reduces the time it takes to render content on the client browser.

With the reduced network latency, faster internet connections, and the increased number of distributed web servers, any static content served from the CDN is delivered more quickly than it would be from a standard web server.

> **Using CDNs means your web server does less work**
>
> Any request that you can push off to the CDN (images, movies, CSS, JavaScript) results in fewer files that the web servers hosting your web application need to serve. Ultimately, this means that to meet the demands of your web traffic, you'll need fewer web role instances, which saves you money. Now, that's a good thing.

OK, you've got the message that you need to use CDNs, so pay close attention while we tell you how to use them with Windows Azure.

10.4.3 Using the Windows Azure CDN

So far, we've looked at CDNs from a generic standpoint; in this section, we'll take a look at what options are available in Windows Azure. In particular, we'll look at how you can use Windows Azure as an origin server and what options you have regarding edge servers.

Although in this section we're showing you how to use the Windows Azure CDN (c'mon, it's a Windows Azure book, after all), you can use other CDN providers, such as Akamai and Amazon Cloud Front. As you're about to discover, using the Windows Azure CDN is probably the easiest option (we have no idea whether it's the cheapest; you'll need to use our JavaScript calculator to work that one out).

USING BLOB STORAGE AS AN ORIGIN SERVER

Although web roles can be used as origin servers with non-Windows Azure CDNs, using them is a more expensive option than using BLOB storage. Because edge servers can serve only static content, there's no need for them to be able to generate dynamic server-side content. An efficient cost-effective solution is to use your storage account as an origin server rather than use a web role.

> **NOTE** With the Windows Azure CDN, you can use only BLOB storage as an origin server; you can't use web roles.

To use our movie player example, the original version of the Big Buck Bunny movie, or even the original presplit chunk files, would be stored in BLOB storage, and BLOB storage would act as the origin server.

ENABLING THE WINDOWS AZURE CDN

To enable the Windows Azure CDN, open the storage account in the Windows Azure Developer portal and click Enable CDN, as shown in figure 10.12.

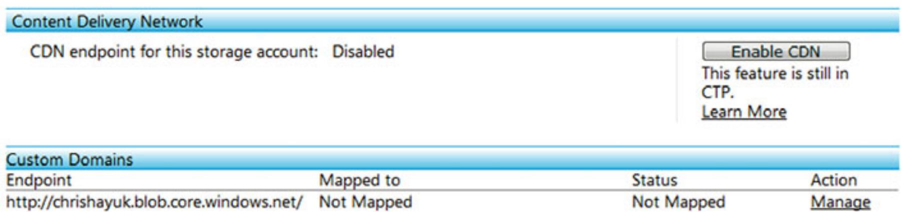


Figure 10.12 Enabling a CDN in the Windows Azure Developer portal

After you click this button, any BLOBs stored in public containers will be available on the CDN after about an hour. Figure 10.13 shows the Developer portal after your CDN account has been enabled.

Content Delivery Network

CDN endpoint for this storage account: Enabled
http://az1903.vo.msecnd.net/

Disable CDN
This feature is still in CTP.
Learn More

Custom Domains

Endpoint	Mapped to	Status	Action
http://chrishayuk.blob.core.windows.net/	Not Mapped	Not Mapped	Manage
http://az1903.vo.msecnd.net/	Not Mapped	Not Mapped	Manage

Figure 10.13 An enabled CDN account in the Windows Azure Developer portal

A new custom domain is assigned to your CDN-enabled BLOBs. In figure 10.13, the CDN domain is http://az1903.vo.msecdn.net/. All you need to do now is replace where we used http://chrishayuk.blob.core.windows.net/ with http://az1903.vo.msecdn.net/, and the files will start being fed from the Windows Azure CDN instead of from our BLOB storage account.

If you don't like the assigned subdomain, you can assign your own custom domain to the CDN domain; for details on how to do that, see chapter 8.

TIP If you want to improve the amount of parallel downloads on your site, assign half your assets to the assigned CDN domain, and the other half to a custom domain name.

10.5 Summary

Over the course of part 4 (chapters 8, 9, and 10), we've explored BLOB storage pretty thoroughly and you should now be able to use this storage service effectively in your applications.

In this chapter, you've discovered that BLOB storage can be more than just a hard disk for your web and worker roles. BLOB storage is a powerful storage mechanism that can do the following things for you:

- Host static websites
- Host RIAs, such as Silverlight
- Act as a media server
- Host assets for your existing websites (Azure or non-Azure hosted websites)
- Act as a CDN

We're going to move away now from BLOB storage and build upon the knowledge you gained with this storage service and look at another part of the storage services puzzle, the Table service.

Part 5

Working with structured data

Now that you have working with files under your belts, we can turn our attention to how to work with structured data. There are two options in Azure.

Chapter 11 covers the first option, Azure Table storage. This stuff is really different from your dad's SQL Server and relational data engines. Take off your mental blinders or your mind will be blown!

Don't like to work with high-level APIs? Do you prefer to set the value of AX yourself? Then chapter 12 is for you. Chapter 12 looks at how to work with Azure tables, using the REST interface. Hardcore stuff, but easy for you.

For those who had their minds blown in chapter 11, we'll put them back together again when we cover SQL Azure in chapter 13. Have an app you want to move to the cloud, but you're using an old-fashioned relational data model? You can easily move to the cloud using SQL Azure and all its foreign-key-indexed-relationships-and-transactions supporting goodness.

Finally, we polish off part 5 with chapter 14 and look at how and when you might choose Azure tables and SQL Azure. We try to end the debate in a peaceful way and help you make solid decisions for your data platform.

11 The Table service, a whole different entity

This chapter covers

- Introducing the Table service
- Getting started developing with the Table service
- Using the Table service in a production environment

In typical web applications, you'd normally store your data in a relational database, such as SQL Server. SQL Server is great at representing relational data and is a suitable data store for many situations, but it's very difficult to design scalable SQL Server databases at low cost. To get around the problems of scalability, Windows Azure provides its own table-based storage mechanism called the Table service.

Problems of scale in relational databases

As the web server load increases for a site, you may need to scale up the number of servers to cope with the increase in demand. But what do you do if you need to

(continued)

increase the capacity of your data store? Unfortunately, most databases aren't designed to scale beyond a single server, which means the only way to cope with the increase in demand is either to scale up your hardware or redesign your application.

Although it's possible to design scalable federated databases with SQL Server, the licensing costs, the design complexity, the cost of development, and the operational costs of running such a system make it very difficult to justify for many companies on a budget.

Let's now take a brief look at what the Table service is.

11.1 A brief overview of the Table service

The Table service component of the Windows Azure storage services (which includes the BLOB service, Table service, and Queue service) is a very simple, highly scalable, cost-effective solution that can be used to store data. In many scenarios it can replace traditional SQL Server–based designs.

> **NOTE** Like all other storage services, the Table service is hosted within the Windows Azure data centers, leveraging the web role infrastructure. Access to the service is provided through an HTTP-based REST API. For more details on the infrastructure of the service see chapter 9.

The Table service provides you with the ability to create very simple tables that you can use to store serialized versions of your entities. Figure 11.1 shows how entities are stored in the Table service.

In figure 11.1 you can see that there are two tables (Products and ShoppingCart) in a storage account (silverlightukstorage). The Products table could represent the product list for the Hawaiian Shirt Shop website that we introduced in chapter 2, and each entity stored in the Products table (Red Shirt, Blue Shirt, and Blue Frilly Shirt) would represent different types of shirts.

It's important to point out that although the Table service offers the ability to store data in tables, it's an entity storage mechanism, not a relational database. That means it doesn't offer the sort of functionality that you may be used to:

- It can't create foreign key relationships between tables.
- It can't perform server-side joins between tables.
- It can't create custom indexes on tables.

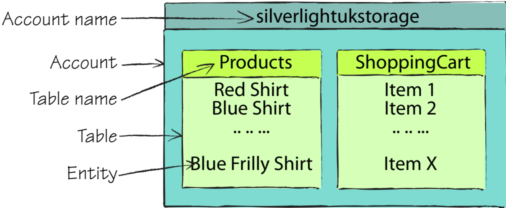


Figure 11.1 The Hawaiian shirts (which are represented as entities) are stored in the Products table. Tables are very similar to BLOB containers (but they hold entities rather than BLOBs). Like BLOB containers, tables are housed within your storage account. Here you can see that the Products and ShoppingCart tables live in the silverlightukstorage account.

If you do require relational database storage, you can look at SQL Azure, which is a Windows Azure platform–hosted SQL Server database.

> **I always need a relational database, don't I?**
>
> We've become a little conditioned to store data in a relational form, even when it's not strictly necessary.
>
> If you can expand your mind and accept that there are other ways of storing data, you can use the Table service to store your data in a highly scalable (and cheaper) fashion.
>
> As you'll see later in this book, many applications (including shopping carts, blogs, content management systems, and so on) could potentially use the simple Table service rather than a relational database.

Now that you know what the Table service is and isn't, it's nearly time to look at how entities are stored in the Table service. But before we do that, let's take a look at how we'd normally represent data in non-Windows Azure environments.

NOTE In the next section, we'll look at how we'd represent an entity in a typical SQL Server solution, and we'll compare this to Table service solutions. If you have no patience or are addicted to caffeine and need to get to Table service code right now, feel free to skip along to section 11.3.

11.2 How we'd normally represent entities outside of Azure

To keep things simple, we'll return to the Hawaiian Shirt Shop website that we introduced in chapter 2. Over the next few sections, we'll look at how we'd typically store shirt data in a noncloud database. We'll focus on the following:

- How would we represent a shirt in C#?
- How would we store shirt data in SQL Server?
- How would we map and transfer data between the two platforms?

By understanding how we'd represent our shirt data in typical solutions, we can then see how this translates to the Table service.

11.2.1 How we'd normally represent an entity in C#

In chapter 2, we defined a Hawaiian shirt using the following data transfer object (DTO) entity.

```
public class Product
{
   public int Id { get; set; }
   public string Name { get; set; }
   public string Description { get; set; }
}
```

The `Product` entity class that we're using to represent Hawaiian shirts contains three properties that we're interested in (`Id`, `Name`, and `Description`).

In chapter 2, we chose to keep the example simple by hardcoding the list of products rather than retrieving it from a data store such as the Table service or SQL Server. The following code is a hardcoded list of the three shirt entities displayed in figure 11.1 (Red Shirt, Blue Shirt, and Blue Frilly Shirt).

```
var products =
   new List<Product>
   {
     new Product
     {
        Id = 1,
        Name = "Red Shirt",
        Description = "Red"
     },
     new Product
     {
        Id = 2,
        Name = "Blue Shirt",
        Description = "A Blue Shirt"
     },
     new Product
     {
        Id = 3,
        Name = "Blue Frilly Shirt",
        Description = "A Frilly Blue Shirt"
     },

};
```

In the preceding code, we simply defined the list of products as a hardcoded list. Obviously this isn't a very scalable pattern—you don't want to redeploy the application every time your shop offers a new product—so let's look at how you can store that data using a non-Windows Azure environment, such as SQL Server.

11.2.2 How we'd normally store an entity in SQL Server

To store an entity in SQL Server, you first need to define a table where you can store the entity data. Figure 11.2 shows how the Products table could be structured in SQL Server.

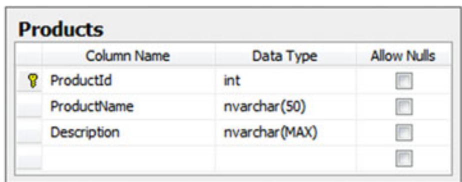


Figure 11.2 A representation of how you could store the Hawaiian shirt data in SQL Server

Figure 11.2 shows a table called Products with three columns (ProductId, ProductName, and Description). In this table, ProductId would be the primary key and would uniquely identify shirts in the table. Table 11.1 shows how the shirt data would be represented in SQL Server.

Table 11.1 Logical representation of the Products table in SQL Server

ProductId	ProductName	Description
1	Red Shirt	Red
2	Blue Shirt	A Blue Shirt
3	Blue Frilly Shirt	A Frilly Blue Shirt

In table 11.1 we've enforced a fixed schema in our SQL Server representation of the Hawaiian shirts. If you wanted to store extra information about the product (a thumbnail URI, for example) you'd need to add an extra column to the Products table and a new property to the `Product` entity.

Now that we can represent the Hawaiian shirt product as both an entity and as a table in SQL Server, we'll need to map the entity to the table.

11.2.3 Mapping an entity to a SQL Server database

Although you can manually map entities to SQL Server data, you'd typically use a data-access layer framework that provides mapping capabilities. Typical frameworks include the following:

- ADO.NET Entity Framework
- LINQ's many varieties, like LINQ to SQL and LINQ to DataSet
- NHibernate

The following code maps the Products table returned from SQL Server as a dataset to the `Product` entity class using LINQ to DataSet.

```
var products = ds.Tables["Products"].AsEnumerable().Select
   (
      row => new Product
      {
        Id = row.Field<int>("ProductId"),
        Name = row.Field<string>("ProductName"),
        Description = row.Field<string>("Description")
      }
   );
```

In this example, we convert the dataset to an enumerable list of data rows and then reshape the data to return a list of `Product` entities. For each property in the `Product` entity (`Id`, `Name`, and `Description`) we map the corresponding columns (ProductId, ProductName, and Description) from the returned data row.

We've now seen how we'd normally define entities in C#, how we'd represent entities in SQL Server, and how we could map the entity layer to the database. Let's look at what the differences are when using the Table service.

11.3 Modifying an entity to work with the Table service

Before we look at how we can start coding against the Table service, you need to understand how your data is stored in the Table service and how that differs from the SQL-based solution we looked at in the previous sections. In the next couple of sections, we'll look at the following:

- How can we modify an entity so it can be stored in the Table service?
- How is an entity stored in the Table service?

As these points suggest, before you can store the shirt data with the Table service, you need to do a little bit of jiggery pokery with the entity definition. Let's look at what you need to do.

11.3.1 Modifying an entity definition

To be able to store the C# entity in the Table service, each entity must have the following properties:

- `Timestamp`
- `PartitionKey`
- `RowKey`

Therefore, to store the `Product` entity in the Azure Table service, you'd have to modify the previous definition of the `Product` entity to look something like this:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Product
{
   public string Timestamp{ get; set; }
   public string PartitionKey { get; set; }
   public string RowKey { get; set; }
   public string Name { get; set; }
   public string Description { get; set; }
}
```

In the preceding code the original `Product` entity is modified to include those properties required for Table storage (`Timestamp`, `PartitionKey`, and `RowKey`). Don't worry if you don't recognize these properties—we'll explain what they mean shortly.

To generate a hardcoded list of shirts using the new version of the `Product` entity, you'd need to change the hardcoded product list (shown earlier in section 11.2.1) to something like this:

```
var products =
   new List<Product>
   {
     new Product
     {
        PartitionKey = "Shirts",
```

```
            RowKey= "1",
            Name = "Red Shirt",
            Description = "Red"
         },
         new Product
         {
            PartitionKey = "Shirts",
            RowKey = "2",
            Name = "Blue Shirt",
            Description = "A Blue Shirt"
         },
         new Product
         {
            PartitionKey = "Shirts",
            RowKey = "3",
            Name = "Frilly Blue Shirt",
            Description = "A Frilly Blue Shirt"
         }

};
```

As you can see from the preceding code, the only difference is that you're now setting a couple of extra properties (`PartitionKey` and `RowKey`).

Look, no `Timestamp`

Notice that the revised object-creation code doesn't set the `Timestamp` property. That's because it's generated on the server side and is only available to us as a read-only property. The `Timestamp` property holds the date and time that the entity was inserted into the table, and if you did set this property, the Table service would just ignore the value.

The `Timestamp` property is typically used to handle concurrency. Prior to updating an entity in the table, you could check that the timestamp for your local version of the entity was the same as the server version. If the timestamps were different, you'd know that another process had modified the data since you last retrieved your local version of the entity.

Now that you've seen how to modify your entities so that you can store them in the Table service, let's take a look at how these entities would be stored in a Table service table.

11.3.2 Table service representation of products

In table 11.1 you saw how we'd normally store our list of Hawaiian shirt product entities in SQL Server, and table 11.2 shows how those same entities would logically be stored in the Windows Azure Table service.

Table 11.2 Logical representation of the Products table in Windows Azure

Timestamp	PartitionKey	RowKey	PropertyBag
2009-07-01T16:20:32	Shirts	1	Name: Red Shirt Description: Red
2009-07-01T16:20:33	Shirts	2	Name: Blue Shirt Description: A Blue Shirt
2009-07-01T16:20:33	Shirts	3	Name: Frilly Blue Shirt Description: A Frilly Blue Shirt

As you can see in table 11.2, entities are represented in the Table service differently from how they'd be stored in SQL Server. In the SQL Server version of the Products table, we maintained a fixed schema where each property of the entity was represented by a column in the table. In table 11.2 the Table service maintains a fairly minimal schema; it doesn't rigidly fix the schema. The only properties that the Table service requires, and that are therefore logically represented by their own columns, are `Timestamp`, `PartitionKey`, and `RowKey`. All other properties are lumped together in a property bag.

EXTENDING AN ENTITY DEFINITION

Because all tables created in the Table service have the same minimal fixed schema (`Timestamp`, `PartitionKey`, `RowKey`, and `PropertyBag`) you don't need to define the entity structure to the Table service in advance.

This flexibility means that you can also change the entity class definition at any time. If you wanted to show a picture of a Hawaiian shirt on the website, you could change the `Product` entity to include a thumbnail URI property as follows:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Product
{
   public string Timestamp{ get; set; }
   public string PartitionKey { get; set; }
   public string RowKey { get; set; }
   public string Name { get; set; }
   public string Description { get; set; }
   public string ThumbnailUri { get; set; }
}
```

Once you've modified the entity to include a thumbnail URI, you can store that entity directly in the existing Products table without modifying either the table structure or the existing data. Table 11.3 shows a list of shirts that include the new property.

Table 11.3 The modified entity with a new property can happily coexist with older entities that don't have the new property.

Timestamp	PartitionKey	RowKey	PropertyBag
2009-07-01T16:20:32	Shirts	1	Name: Red Shirt Description: Red
2009-07-01T16:20:33	Shirts	2	Name: Blue Shirt Description: A Blue Shirt
2009-07-01T16:20:33	Shirts	3	Name: Frilly Blue Shirt Description: A Frilly Blue Shirt
2009-07-05T10:30:21	Shirts	4	Name: Frilly Pink Shirt Description: A Frilly Pink Shirt ThumbnailUri: frillypinkshirt.png

In the list of shirts in table 11.3, you can see that existing shirts (Red Shirt, Blue Shirt, and Frilly Blue Shirt) have the same data that was stored in table 11.2—they don't contain the new `ThumbnailUri` property. But the data for the new shirt (Frilly Pink Shirt) does have the new `ThumbnailUri` property.

11.3.3 Storing completely different entities

Due to the flexible nature of the Table service, you could even store entities of different types in the same table. For example, you could store the `Product` entity in the same table as a completely different entity, such as this `Customer` entity:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Customer
{
   public string Timestamp{ get; set; }
   public string PartitionKey { get; set; }
   public string RowKey { get; set; }
   public string Firstname { get; set; }
   public string Surname { get; set; }
}
```

As you can see from the `Customer` entity, although the entity must contain the standard properties (`Timestamp`, `PartitionKey`, and `RowKey`) no other properties are shared between the Customer and Product entities; they even have different class names.

Even though these entities have very different definitions, they could be stored in the table, as shown in table 11.4. The Table service allows for different entities to have different schemas.

Table 11.4 Storing completely different entities in the same table

Timestamp	PartitionKey	RowKey	PropertyBag
2009-07-01T16:20:32	Shirts	1	Name: Red Shirt Description: Red
2009-07-01T16:20:33	Shirts	2	Name: Blue Shirt Description: A Blue Shirt
2009-07-01T16:20:33	Shirts	FredJones	Firstname: Fred Surname: Jones
2009-07-05T10:30:21	Shirts	4	Name: Frilly Pink Shirt Description: A Frilly Pink Shirt ThumbnailUri: frillypinkshirt.png

CHALLENGES OF STORING DIFFERENT ENTITY TYPES

Although the Table service is flexible enough to store entities of different types in the same table, as shown in table 11.4, you should be very careful if you're considering such an approach. If every entity you retrieve has a different schema, you'll need to write some custom code that will serialize the data to the correct object type.

Following this approach will lead to more complex code, which will be difficult to maintain. This code is likely to be more error prone and difficult to debug. We encourage you to only store entities of different types in a single table when absolutely necessary.

CHALLENGES OF EXTENDING ENTITIES

On a similar note, if you need to modify the definition of existing entities, you should take care to ensure that your existing entities don't break your application after the upgrade.

There are a few rules you should keep in mind to prevent you from running into too much trouble:

- Treat entity definitions as data contracts; breaking the contract will have a serious effect on your application, so don't do it lightly.
- Code any new properties as additional rather than required. This strategy means that existing data will be able to serialize to the new data structure. If your code requires existing entities to contain data for the new properties, you should migrate your existing data to the new structure.
- Continue to support existing property names for existing data. If you need to change a property name, you should either support both the old and new names in your new entity or support two versions of your entity (old definition

and new definition). If you only want to support one entity definition, you'll need to migrate any existing data to the new structure.

Now that you've seen how entities are stored within the Table service, let's look at what makes this scalable.

11.4 Partitioning data across lots of servers

In the last couple of sections, we've skipped past a few topics, namely, accounts, partition keys, and row keys. We'll now return to these topics and explain how the Windows Azure Table service is such a scalable storage mechanism.

In this section, we'll look at how the Table service scales using partitioning at the storage account and table levels. To achieve a highly scalable service, the Table service will split your data into more manageable partitions that can then be apportioned out to multiple servers. As developers, we can control how this data is partitioned to maximize the performance of our applications.

Let's look at how this is done at the storage account layer.

11.4.1 Partitioning the storage account

In this section, we'll look at how data is partitioned, but we'll leave performance optimization to a later section.

In figure 11.1, there were two tables within a storage account (ShoppingCart and Products). As the Table service isn't a relational database, there's no way to join these two tables on the server side. Because there's no physical dependency between any two tables in the Table service, Windows Azure can scale the data storage beyond a single server and store tables on separate physical servers.

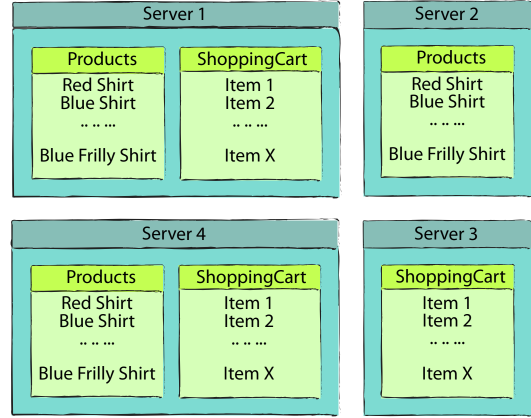


Figure 11.3 Tables within a storage account split across multiple servers

Figure 11.3 shows how these tables could be split across the Windows Azure data center. In this figure, you'll notice that the Products table lives on servers 1, 2, and 4, whereas the ShoppingCart table resides on servers 1, 3, and 4. In the Windows Azure data center, you have no control over where your tables will be stored. The tables could reside on the same server (as in the case of servers 1 and 4) but they could easily live on completely separate servers (servers 2 and 3). In most situations, you can assume that your tables will physically reside on different servers.

Data replication

In order to protect you from data loss, Windows Azure guarantees to replicate your data to at least three different servers as part of the transaction. This data replication guarantee means that if there's a hardware failure after the data has been committed, another server will have a copy of your data.

Once a transaction is committed (and your data has therefore been replicated at least three times), the Table service is guaranteed to serve the new data and will never serve older versions. This means that if you insert a new Hawaiian shirt entity on server 1, you can only be load balanced onto one of the servers that has the latest version of your data. If server 2 was not part of the replication process and contains stale data, you won't be load balanced onto that server. You can safely perform a read of your data straight after a write, knowing that you'll receive the latest copy of the data.

The Amazon SimpleDB database (which has roughly the same architecture as the Windows Azure Table service) doesn't have this replication guarantee by default. Due to replication latency, it isn't uncommon in SimpleDB for newly written data not to exist or to be stale when a read is performed straight after a write. This situation can never occur with the Windows Azure Table service.

Now that you've seen how different tables within a single account will be spread across multiple servers to achieve scalability, it's worth looking at how you can partition data a little more granularly, and split data within a single table across multiple servers.

11.4.2 Partitioning tables

One of the major issues with traditional SQL Server–based databases is that individual tables can grow too large, slowing down all operations against the table. Although the Windows Azure Table service is highly efficient, storing too much data in a single table can still degrade data access performance.

The Table service allows you to specify how your table could be split into smaller partitions by requiring each entity to contain a partition key. The Table service can then scale out by storing different partitions of data on separate physical servers. Any entities with the same partition key must reside together on the same physical server.

In tables 11.2 through to 11.4, all the data was stored in the same partition (Shirts), meaning that all three shirts would always reside together on the same server, as shown in figure 11.3. Table 11.5 shows how you could split your data into multiple partitions.

Table 11.5 Splitting partitions by partition key

Timestamp	PartitionKey	RowKey	PropertyBag
2009-07-01T16:20:32	Red	1	Name: Red Shirt Description: Red

Table 11.5 Splitting partitions by partition key *(continued)*

Timestamp	PartitionKey	RowKey	PropertyBag
2009-07-01T16:20:33	Blue	1	Name: Blue Shirt Description: A Blue Shirt
2009-07-01T16:20:33	Blue	2	Name: Frilly Blue Shirt Description: A Frilly Blue Shirt
2009-07-05T10:30:21	Red	2	Name: Frilly Pink Shirt Description: A Frilly Pink Shirt ThumbnailUri: frillypinkshirt.png

In table 11.5 the Red Shirt and the Frilly Pink Shirt now reside in the Red partition, and the Blue Shirt and the Frilly Blue shirt are now stored in the Blue partition. Figure 11.4 shows the shirt data from table 11.5 split across multiple servers. In this figure, the Red partition data (Red Shirt and Pink Frilly Shirt) lives on server A and the Blue partition data (Blue Shirt and Frilly Blue Shirt) is stored on server B. Although the partitions have been separated out to different physical servers, all entities within the same partition always reside together on the same physical server.

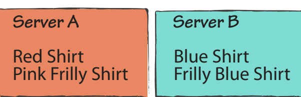


Figure 11.4 Splitting partitions across multiple servers

ROW KEYS

The final property to explain is the row key. The row key uniquely identifies an entity within a partition, meaning that no two entities in the same partition can have the same row key, but any two entities that are stored in different partitions can have the same key. If you look at the data stored in table 11.5, you can see that the row key is unique within each partition but not unique outside of the partition. For example, Red Shirt and Blue Shirt both have the same row key but live in different partitions (Red and Blue).

The partition key and the row key combine to uniquely identify an entity—together they form a composite primary key for the table.

INDEXES

Now that you have a basic understanding of how data is logically stored within the data service, it's worth talking briefly about the indexing of the data.

There are a few rules of thumb regarding data-access speeds:

- Retrieving an entity with a unique partition key is the fastest access method.
- Retrieving an entity using the partition key and row key is very fast (the Table service needs to use only the index to find your data).
- Retrieving an entity using the partition key and no row key is slower (the Table service needs to read all properties for each entity in the partition).

- Retrieving an entity using no partition key and no row key is very slow, relatively speaking (the Table service needs to read all properties for all entities across all partitions, which can span separate physical servers).

We'll explore these points in more detail as we go on.

Load balancing of requests

Because data is partitioned and replicated across multiple servers, all requests via the REST API can be load balanced. This combination of data replication, data partitioning, and a large web server farm provides you with a highly scalable storage solution that can evenly distribute data and requests across the data center. This level of horsepower and data distribution means that you shouldn't need to worry about overloading server resources.

Now that we've covered the theory of table storage, it's time to put it into practice. Let's open Visual Studio and start storing some data.

11.5 Developing with the Table service

Now that you have an understanding of how data is stored in the Table service, it's time to develop a web application that can use it. In the previous section, we defined an entity for storing the Hawaiian shirt product, and we looked at how it would be stored in the Table service. Here you'll build a new application that will manage the product inventory for the Hawaiian Shirt Shop website.

11.5.1 Creating a project

Rather than returning to the solution you built in chapter 2, here you'll develop a new product-management web page in a new web application project. Create a new Cloud Service web role project called `ShirtManagement`. If you need a refresher on how to set up your development environment or how to create a web role project, refer back to chapter 2.

Like the other storage services, communication with the Table service occurs through the REST API (which we'll discuss in detail in the next chapter). Although you can use this API directly, you're likely to be more productive using the StorageClient library provided in the Windows Azure SDK.

Whenever you create a new Cloud Service project, this library will be automatically referenced. But if you're building a brand new class library or migrating an existing project, you can reference the following storage client assembly manually:

- `Microsoft.WindowsAzure.StorageClient`

In addition, you'll need to reference the ADO.NET Data Services assemblies:

- `System.Data.Services`
- `System.Data.Services.Client`

> **ADO.NET Data Services**
>
> The Table service exposes its HTTP REST APIs through its implementation of the ADO.NET Data Services framework. By using this framework, we can utilize the ADO.NET Data Services client assemblies to communicate with the service rather than having to develop or use a custom REST wrapper.
>
> Because ADO.NET Data Services is used by the storage client SDK, you'll need to reference those assemblies too.

Now that you've set up your project, let's look at how you can add the `Product` entity class to the project.

11.5.2 Defining an entity

Before you create your product-management web page, you need to create an entity in the web project. At this stage, we'll just show you how to add the entity directly to the web page, but in the next chapter you'll see how to architecturally separate a class into an entity layer.

To keep this example simple, we'll just store the shirt name and description, as before. Add a new class to your web project named `Product.cs` and define the class as shown in the following listing.

Listing 11.1 `Product` entity

```
public class Product : TableServiceEntity
{
   public string Name { get; set; }
   public string Description { get; set; }
}
```

In listing 11.1, the `Product` class inherits from the `TableServiceEntity` base class (`Microsoft.WindowsAzure.TableService.TableServiceEntity`). This base class contains the three properties required by all table storage entities:

- `Timestamp`
- `PartitionKey`
- `RowKey`

Now that you've set up your project and defined your entity, you need to create a table to store the entity in. The same method can be used in both the development and live environments.

11.5.3 Creating a table

The simplest method of creating a table is to create a PowerShell script or to use one of the many storage clients that are available. In this chapter we'll use Azure Storage Explorer, which you can download from CodePlex: http://azurestorageexplorer.codeplex.com/.

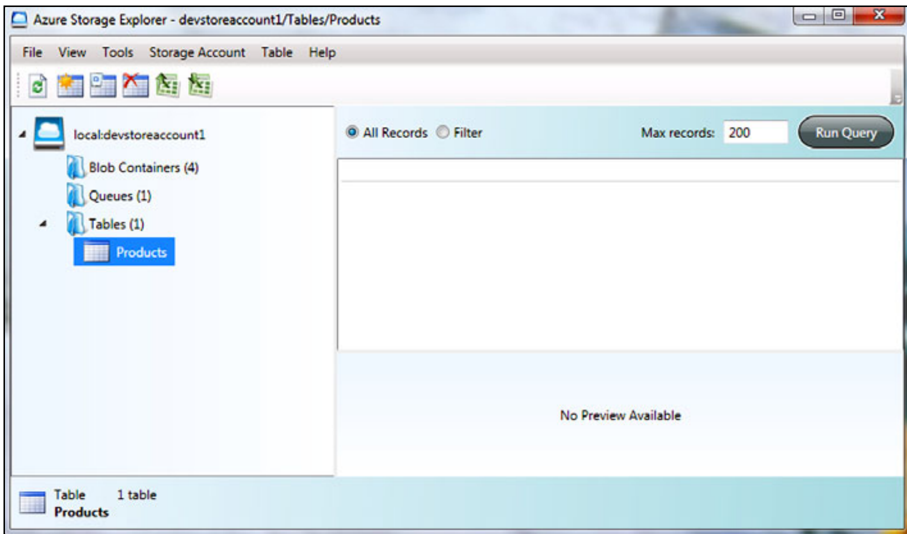


Figure 11.5 The Azure Storage Explorer showing the newly created Products table

In this section, we'll look at how to create a table in two ways: using Azure Storage Explorer and using code.

CREATING A TABLE USING THE AZURE STORAGE EXPLORER

Once you have downloaded and fired up Azure Storage Explorer, it will automatically connect you to your development storage account as long as your local development storage service is running.

To create a new table in your storage account, all you need to is select Table > New Table and enter the name of your table (Products, in this case). Figure 11.5 shows the newly created Products table in Azure Storage Explorer.

Although using a tool such as Azure Storage Explorer is probably the easiest method of creating a new table, you may wish to do this manually in C#.

CREATING A TABLE IN CODE

In this example, you'll manually create a console application that will create a new table in the storage account when it's run. Although we'll have you use a console application in this example, you could easily use a web application, Windows Forms application, or Windows Presentation Foundation application. The deployment application doesn't need to be a cloud application (web or worker role); it can be a standard application that you run locally.

To create the application, perform the following steps:

1. Create a console application targeting .NET 3.5.
2. Add a reference to `System.Data.Services`.
3. Add a reference to `System.Data.Services.Client`.

4 Add a reference to `Microsoft.WindowsAzure.StorageClient`.
5 Add an app.config or web.config entry with your storage account credentials.
6 Add the following code to create the table:

```
CloudStorageAccount.SetConfigurationSettingPublisher((configName,
➥ configSetter) =>
              {
configSetter(RoleEnvironment.GetConfigurationSettingValue(configName));
              });

var storageAccount =

  CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

CloudTableClient tableClient =
   storageAccount.CreateCloudTableClient();

tableClient.CreateTableIfNotExist("Products");
```

The code added in step 6 retrieves storage account information from the app.config and then calls the `CreateTableIfNotExist` method from the `CloudTableClient` object, passing in the name of the table to create (Products).

> **Deploying to live**
>
> The code used to create a new table will work not only on your development storage account, but will also work against the live system. All you need to do to make this code work against the live system is to change the `DataConnectionString` configuration setting to your live account.

Now that you know how to create a table both in the live system and in development storage, it's worth taking a quick peek at how this is implemented in the development storage backing store. Figure 11.6 shows how tables are represented in the development storage SQL Server database.

As you can see in figure 11.6, the SQL Server database that stores the entities is pretty flexible. The TableContainer table keeps a list of all the tables stored in the development storage account. Because you can create tables dynamically, any new table created will contain a new entry in this table.

Each row in the TableRow table in figure 11.6 stores a serialized version of the entity. As you can see from this table definition, the only fixed data that's stored in this table is AccountName, TableName, PartitionKey, RowKey, and TimeStamp. All other properties are stored in the Data column. As you can see, the actual development storage schema relates to the logical representation that you saw in table 11.4.

Now that you've seen how tables are represented in development storage, let's look at how you can start working with your entities.

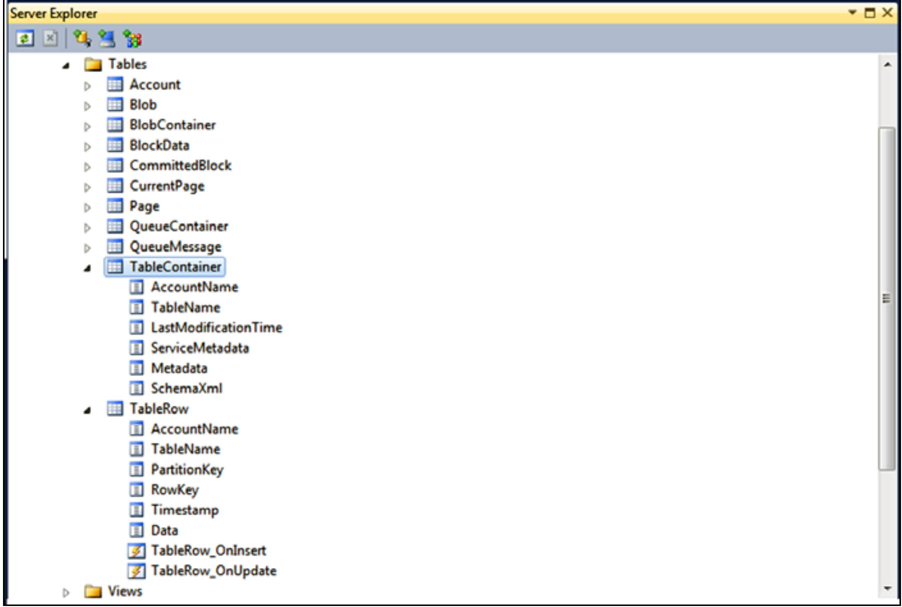


Figure 11.6 How tables are represented in the development storage SQL Server database

11.6 Doing CRUDy stuff with the Table service

In this section, you'll build a new product-management web page to manage the Hawaiian shirt product list stored in the Table service. Through this web page, you'll be able to *create*, *read*, *update*, and *delete* (also known as CRUD) data in the table. Figure 11.7 shows what the web page will look like.

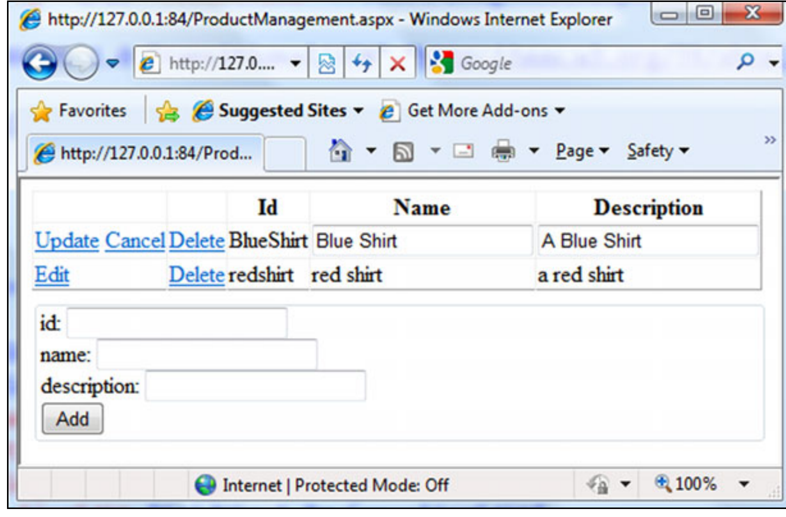


Figure 11.7 Product-management web page

The product-management web page shown in figure 11.7 will allow us to do several things:

- Add new shirts
- List all shirts
- Edit existing shirts
- Delete shirts

You've already set up the entity and registered the entity table in the development storage. It's time to develop the product-management web page shown in figure 11.7. The first step is to set up a context class for your entity.

11.6.1 Creating a context class

In order to work with entities in any way (query, add, insert, delete) using ADO.NET Data Services, you first need to set up a context object.

The context class is really a bridge between an entity and ADO.NET Data Services. It will define the endpoint of the storage account, the name of the table that we'll query, and the entity that will be returned. The following listing shows the context class for the Products table.

Listing 11.2 Product context class

```
public class ProductsContext : TableServiceContext
{
   private static CloudStorageAccount storageAccount =        <-- Reads account details from config
      CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

   public ProductsContext()
      : base(storageAccount. TableEndpoint.AbsoluteUri (),
                storageAccount.Credentials)                  <-- Uses account details to populate context
   {

   }

   public DataServiceQuery<Product> Product
   {
      get
      {
         return CreateQuery<Product>("Products");
      }
   }
}
```

Listing 11.2 shows the context object for the Hawaiian shirt `Product` entity. As you can see, most of the complexity of the context class is abstracted away in the classes that you inherit from. The `TableServiceContext` class inherits from the standard ADO.NET Data Services context class, `DataServicesContext`. The `TableServiceContext` class provides some additional functionality beyond what is provided out of the box with ADO.NET Data Services, including retry policies.

In listing 11.2, the storage account details and credentials are automatically populated from the service configuration. This allows you to simplify your calling classes—you don't need to get the endpoint and credentials every time you wish to use the context class.

Finally, the `Product` property is what you'll use to perform LINQ queries on the Products table.

> **TIP** Code generation is outside the scope of this book, but if you're generating a large number of tables and entities, you may wish to consider using a code generation tool such as T4 to autogenerate code where possible. Typical areas to consider generating code automatically would include table context classes and table-creation scripts.

Let's look at how you can start using this.

11.6.2 Adding entities

Now that you have your context class, you can start creating your product-management web page. To do this, you need to add a new ASP.NET web page called products.aspx.

At this stage, we won't generate the grid listing all the Hawaiian shirts for sale; we'll only write the code required to add shirts to the Products table. Therefore, you only need to add the markup for the bottom section of the products page.

The listing that follows contains the ASPX code that you should add to the products.aspx page.

Listing 11.3 Adding the shirt section of the products.aspx page

```
<fieldset>
   <div>
      <div>id:
<asp:TextBox id="txtId" runat="server" />
</div>                                                   Id text box
      <div>name:
<asp:TextBox id="txtName" runat="server" />
</div>
      <div>
          description:
          <asp:TextBox id="txtDescription"
runat="server" />
      </div>
      <asp:button id="btnAdd" text="Add"
                  runat="server"
onclick="btnAdd_Click" />                                  Add button
   </div>
</fieldset>
```

When the Add button is clicked in listing 11.3, the shirt entity will be added to the Products table.

Listing 11.4 contains the code for the Add button's click event. This code should be added to the products.aspx code-behind.

Listing 11.4 Add the entity to the Products table

```
protected void btnAdd_Click(object sender, EventArgs e)
{
   var shirtContext = new ProductsContext();            // 1 Creates context to connect with table

   var newShirt = new Product                           // 2 Creates object to store in table
       {
          PartitionKey = "Shirts",
          RowKey = txtId.Text,
          Name = txtName.Text,
          Description = txtDescription.Text
       };

   shirtContext.AddObject("Products", newShirt);        // 3 Adds new object to context

   shirtContext.SaveChanges();                          // 4 Commits changes in context to table

   }
```

To add the new shirt details, which were entered on the web page, to the Products table, you need to extract all the information entered about the product (ID, name, and description) and create a new instance of the `Product` class called `newShirt` (2). Once you've created an instance of the shirt (3), you add the shirt entity to a tracking list held in the context object (1) for the product list.

When you add the entity to the tracking list (3), you also specify the table that the entity should be added to. The product context object (`shirtContext`) maintains a list of all objects that you have changed as part of this operation. You can create, update, or delete objects from the product list, and you can add all these changes locally to the tracking list.

Eventually, when you wish to perform all the changes on the server side, you can invoke the `SaveChanges` method on the context object (4), which will apply all the tracked changes on the server side using ADO.NET Data Services via the REST API.

In the next chapter, we'll look at some of the more advanced scenarios for applying changes, such as batching changes, concurrency, retry logic, and transactions.

WAS THE ENTITY ADDED?

You should now be able to run the product-management web page and add new shirts to the product list. We haven't yet implemented the grid to display the list of entities in the product list—we'll do that in the next section. In the meantime you can always check that your entity exists by querying the development storage database using the following statement in SQL Management Studio:

```
SELECT * FROM TableRow
```

11.6.3 Listing entities

It's now time to extend the products.aspx web page to display the list of shirts stored in the Products table. The following listing contains the code required for your grid. This code should be placed above the code in listing 11.3.

Listing 11.5 ASP.NET grid for displaying shirts

```
<asp:GridView ID="GridView1"
              OnRowCommand="GridView1_RowCommand"
              OnRowDeleting="GridView1_RowDeleting"
              OnRowEditing="GridView1_RowEditing"
              OnRowCancelingEdit="GridView1_RowCancelingEdit"
              OnRowUpdating="GridView1_RowUpdating"
              AutoGenerateColumns="false"
              AutoGenerateEditButton="true"
              runat="server">
   <Columns>
      <asp:TemplateField>
         <ItemTemplate>
            <asp:LinkButton ID="btnDelete" runat="server"
                            Text="Delete"
                            CommandName="Delete"
                            CommandArgument=
                            '<%#Eval("RowKey")%>'/>
         </ItemTemplate>
      </asp:TemplateField>
      <asp:BoundField HeaderText="Id"
                      DataField="RowKey"
                      ReadOnly="true"/>
      <asp:BoundField HeaderText="Name"
                      DataField="Name" />
      <asp:BoundField HeaderText="Description"
                      DataField="Description" />
   </Columns>
</asp:GridView>
```

Passes in product ID (row key)

The grid in listing 11.5 will display the product ID, name, and description for each shirt in the Products table. The name and description columns will both be editable, but the product ID won't be.

At this stage, you've defined the markup required to edit and delete the shirts in the table, but we won't write the code-behind for these events just yet. The following listing contains the code-behind for the products.aspx page, which you'll require to populate the new grid with the list of shirts.

Listing 11.6 Populating the list of shirts

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      BindGrid();
```

```
    }
}

private void BindGrid()
{
    var shirtContext = new ProductContext();              ❶ Creates context to connect to table
    GridView1.DataSource = shirtContext.Product;          ❷ Sets grid's data source to Products table
    GridView1.DataBind();                                 ❸ Binds grid
}
```

As you can see, the code to bind the `GridView` to the list of shirts held in the Products table is pretty simple. On the first load of the web page ❶, the `BindGrid` method ❷ is called to populate the grid with the list of shirts retrieved from the Products table.

To retrieve the list of shirts, you instantiate the product context object (`shirtContext`) ❷ and set the data source of the grid to the `Product` property of the context object.

If you look back to the code used to define the product context in listing 11.2, you'll see that the `Product` property returns an `IQueryable` list of products. By returning an `IQueryable` list of products from the context object, you can define a query using LINQ that will be executed on the server side when you enumerate the list of objects, which happens when the grid is data bound ❸.

To keep this example simple, we won't perform any server-side filtering at this stage. We'll simply return a list of all shirts in the Products table as shown at ❷. In the next chapter, we'll look at how to build efficient server-side queries using LINQ.

11.6.4 Deleting entities

In listing 11.5 we included some event definitions in the markup to handle deletes. It's now time to implement those event handlers so that when you click the Delete button in the grid, it will delete the corresponding shirt from the Products table in the Table service.

In the listing that follows you'll see the code-behind that relates to the Delete button.

Listing 11.7 Code-behind to delete shirts

```
protected void GridView1_RowCommand (object sender,
                                GridViewCommandEventArgs e)
{
    if (e.CommandName == "Delete")
    {
        DeleteShirt(e.CommandArgument.ToString());
        BindGrid();
     }
}

private void DeleteShirt(string rowKey)
{
    var shirtContext = new ProductContext();              ❶ Creates context to connect with table
```

```
    var entity = (from item in shirtContext.Product
                  where item.PartitionKey=="Shirts" &&
                  item.RowKey==rowKey
                  select item).First();

    shirtContext.DeleteObject(entity);

    shirtContext.SaveChanges();

    BindGrid();
}

protected void GridView1_RowDeleting(object sender,
➥ GridViewDeleteEventArgs e){}
```

(2) Uses query to fetch record from table

(3) Deletes object locally

The code in listing 11.7 shows how you can delete a shirt from the Products table when the Delete button is clicked in the grid. The `DeleteShirt` method is very similar to the code in listing 11.4 that you used to add a shirt. You instantiate the shirt context (1), add the shirt to be deleted to the context object's tracking list (3), and submit the changes to the Table service. Finally, you rebind the grid to display the updated list.

FINDING THE ENTITY TO DELETE

There are a couple of differences between adding and deleting a shirt. When adding a shirt to the product list, you can simply create a new `Product` object and add that to the tracking list. If you're deleting an object, you first need to get a local copy of the object that you wish to delete (as you did at (2) in listing 11.7) and then add that object to the tracking list (as shown at (3)).

For the sake of simplicity, you can pass the object to be deleted by refetching the object from the Table service. In production code, you should never refetch an object for deletion because this performs an unnecessary call to the Table service—in the next chapter, we'll look at how to delete an object in a production web scenario. In this example, however, you can take the `ProductId` (row key) that was passed as the Delete button's command argument, and then perform a LINQ query to retrieve that entity from the Table service.

The following code shows the LINQ query we used to retrieve the individual shirt:

```
from item in shirtContext.Product
where item.PartitionKey=="Shirts" && item.RowKey==rowKey
select item
```

Looking at the LINQ query, you can see that we're using the `IQueryable Product` from the `shirtContext` object as the data source. Because this query is `IQueryable`, you can modify the query before it's sent to the server to restrict the result set to only return those entities that reside in the Shirts partition whose row key matches the passed-in `ProductId`. Because this query is manipulated before it's sent to the server, all the filtering is performed by the Table service rather than by the client application.

You'll now be able to add, list, and delete shirts in the Products table.

11.6.5 Updating entities

Finally, you need to update the code-behind to allow editing the grid and saving any changes back to the Products table. The following listing contains the code you need to edit the grid.

Listing 11.8 Updating entities

```
protected void GridView1_RowCommand(object sender,
                                  GridViewCommandEventArgs e)
{
   if (e.CommandName == "Delete")
   {
      DeleteShirt(e.CommandArgument.ToString());
      BindGrid();
   }
   if (e.CommandName == "Edit")
   {
      BindGrid();
   }
}

protected void GridView1_RowEditing
                 (object sender,GridViewEditEventArgs e)
{
   GridView1.EditIndex = e.NewEditIndex;
   GridView1.DataBind();
}

protected void GridView1_RowCancelingEdit(object sender,
                                   GridViewCancelEditEventArgs e)
{
   GridView1.EditIndex = -1;
   GridView1.DataBind();
}

protected void GridView1_RowUpdating(object sender,
                                   GridViewUpdateEventArgs e)
{
   GridViewRow row = GridView1.Rows[e.RowIndex];
   string id = row.Cells[2].Text;

   var shirtContext = new ProductContext();

   var entity = (from item in shirtContext.Product
                where item.PartitionKey ==
                    "Shirts" && item.RowKey == id
                select item).First();

   entity.Name = ((TextBox)
                 (row.Cells[3].Controls[0])).Text;
   entity.Description = ((TextBox)
                        (row.Cells[4].Controls[0])).Text;
```

1. Create a context to the table
2. Retrieve the old entity from the table
3. Update the data in the entity from the table

```
    shirtContext.UpdateObject(entity);
    shirtContext.SaveChanges();

    GridView1.EditIndex = -1;
    BindGrid();
}
```

After editing the grid with your new values, you need to store the modified data back to the Products table. Listing 11.8 will be called whenever there's a change to any data in the grid. The `GridView`'s `RowUpdating` event is where you'll perform that update. The process of updating data is very similar to deleting an entity, as shown earlier.

You retrieve the row that has been edited and extract the product ID displayed in the second cell of the row. As before, you instantiate the product context (1), and at (2) you retrieve a copy of the edited entity from the Table service using a LINQ query, passing the partition key and the row key. At (3) you replace the current name and description of the shirt with the modified data extracted from the text boxes of the row being edited. You then add the modified entity to the `shirtContext`'s tracking list via the `UpdateObject` method, commit the changes back to the Table service using the `SaveChanges` method, take the grid out of edit mode, and rebind the grid to a fresh copy of the data returned from the Table service.

11.7 Summary

In this chapter, we gave you an overview of the Table service, explaining how it provides massively scalable storage and how it differs from traditional relational databases, such as SQL Server.

As you've seen, the Table service is a scalable way of storing and querying entities, and it isn't a relational database. We can sometimes assume we need relational databases when there are other ways of representing data in our solution. Designing systems is all about trade-offs, and the trade-off with relational databases is scalability. By using the Table service, you gain scalability, but the cost is that you have to think and design systems in a different way.

You now have enough knowledge to create and deploy web applications that run in the cloud and can store data in the Table service. In the next chapter, we'll expand upon your new knowledge and drill into some of the inner workings of the service by looking at the REST API and seeing how you can efficiently query and update data.

12 Working with the Table service REST API

This chapter covers

- Introducing the StorageClient library and REST API
- Understanding how to effectively modify data
- Querying the Table service

In the previous chapter, we looked at how to get started with the Table service using the StorageClient library and the WCF Data Services client. In this chapter, we'll look at the underlying REST API to gain a better understanding of the communication between applications and the Table service.

WCF Data Services is an implementation of the OData protocol. OData defines how to work with and exchange data over REST-based services. The Table service implements OData, and the StorageClient library acts as a WCF Data Services client. Even with these layers of abstraction, understanding how to query and update data using the REST API will help you to truly understand what is happening under the covers when using the WCF Data Services–based Table service API.

Let's get started by looking at how you can perform operations directly against the REST API.

12.1 Performing storage account operations using REST

For now, we'll concentrate purely on the operations that you can perform against a storage account using the REST API. Although we've already looked at these operations using the StorageClient library, it's still useful to look at the REST API. Ultimately, the StorageClient library is just a wrapper library for the calls we're about to look at. Over the next few sections, we'll look at the following operations:

- Listing tables
- Deleting tables
- Creating tables

In chapters 8–10 on BLOBs, we described how you could interact with the storage services using various endpoints. We won't go over that subject again in this chapter, but it's worth looking at the endpoint URI of the Table service. The URI of the Table service endpoint uses the following structure:

```
http://<storageaccount>.table.core.windows.net/
```

If your storage account was named silverlightukstorage, your URI would be the following:

```
http://silverlightukstorage.table.core.windows.net/
```

For the development storage Table service, you'd use the following URI:

```
http://127.0.0.1:10002/devstoreaccount1/
```

Now that you know what the URIs will look like, let's try using them.

12.1.1 Listing tables in the development storage account using the REST API

In chapter 9, we looked at a small console application that listed all the containers in a BLOB storage account using the REST API. In this section, you'll create a similar console

```
file:///C:/AzureBook/Source/AiAChapter8/RESTAPIConsole/bin/Debug/RESTAPIConsole.EXE
<feed xml:base="http://127.0.0.1:10002/devstoreaccount1/" xmlns:d="http://schema
s.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/
ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Tables</title>
  <id>http://127.0.0.1:10002/devstoreaccount1/Tables</id>
  <updated>2009-07-26T13:08:29Z</updated>
  <link rel="self" title="Tables" href="Tables" />
  <entry>
    <id>http://127.0.0.1:10002/devstoreaccount1/Tables('ProductTable')</id>
    <title type="text"></title>
    <updated>2009-07-26T13:08:29Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Table" href="Tables('ProductTable')" />
    <category term="Microsoft.ServiceHosting.DevelopmentStorage.Table.Table" sch
eme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:TableName>ProductTable</d:TableName>
      </m:properties>
    </content>
  </entry>
</feed>
```

Figure 12.1 A console application that returns a list of tables in a storage account—that's a lot of XML just to return a list containing the name of one table.

application that will list all the tables in a development storage account. Figure 12.1 shows the output of this console application.

If you look at the output in figure 12.1 you can see that the Products table (created in the previous chapter) is returned in the list of storage accounts.

Does that funny-looking XML follow some sort of standard?

As you may have already gathered, normal people don't create APIs that output XML like what you see in figure 12.1. You need a standard to generate that level of verboseness and complexity.

The standard that the Table service uses to expose its data is known as AtomPub. We'll discuss it in more detail later.

If you're interested in being able to identify AtomPub documents in the wild, you can always look at the XML namespace. As you can see in figure 12.1, it's referencing the `Atom` namespace:

```
xmlns=http://www.w3.org/2005/Atom
```

If you wanted to list all the tables in a storage account using the StorageClient library, you could do the following:

```
var storageAccount =
      CloudStorageAccount.Parse(
         ConfigurationManager.AppSettings["DataConnectionString"]);

CloudTableClient tableClient =
   storageAccount.CreateCloudTableClient();

tableClient.ListTables();
```

Let's now take a look at how this could be done using the REST API directly. You might not usually do this directly with REST, but we want you to appreciate what's really happening behind the scenes.

In order to get a list of all tables that exist in the storage account, all you need to do is write some code that will perform a `GET` request against the following URIs:

```
http://127.0.0.1:10002/devstoreaccount1/Tables (for dev storage)
http://<storageaccount>.table.core.windows.net/Tables (for live)
```

To create the application that generated the output shown in figure 12.1, you'll need to create a new console application in Visual Studio. To keep the example simple, we'll reuse the StorageClient library's credential-signing method. You'll need to add a reference to this assembly—this is the same assembly that you used in the previous chapter.

The following listing contains the code for the console application.

Listing 12.1 Listing the tables in a storage account

```
static void Main(string[] args)
{
```

```
HttpWebRequest hwr =
      CreateHttpRequest(new
      ➥ Uri(@"http://127.0.0.1:10002/
      ➥ devstoreaccount1/Tables"),
      ➥ "GET", new TimeSpan(0, 0, 30));

var storageAccount =
        CloudStorageAccount.Parse(ConfigurationManager
        ➥ .AppSettings["DataConnectionString"]);

storageAccount.Credentials.SignRequestLite(hwr);

   using (StreamReader sr =
   ➥ new StreamReader(hwr.GetResponse()
   ➥ .GetResponseStream()))
   {
      XDocument myDocument = XDocument.Parse(sr.ReadToEnd());
      Console.Write(myDocument.ToString());
   }
}

private static HttpWebRequest CreateHttpRequest
➥ (Uri uri, string httpMethod, TimeSpan timeout)
{
   HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(uri);
   request.Timeout = (int)timeout.TotalMilliseconds;
   request.ReadWriteTimeout = (int)timeout.TotalMilliseconds;
   request.Method = httpMethod;
   request.ContentLength = 0;
   request.ContentType = "application/atom+xml";
   return request;
}
```

1 Creates request

2 Uses storage credentials to sign request

3 Processes response

The code displayed in listing 12.1 will make a `GET` request to the development Table service asking for a list of all the tables in the storage account (http://127.0.0.1:10002/devstoreaccount1/Tables).

At **1**, you generate the HTTP request by calling the `CreateHttpRequest` method. This method creates the `HttpRequest` for the given URI (http://127.0.0.1:10002/devstoreaccount1/Tables) and HTTP verb (`GET`) and returns the request to the calling method.

Sign Request Lite

You've probably noticed that listing 12.1 makes use of the storage account credentials from the StorageClient library **2**, even though it's using the REST API. The major reason for this is that signing the HTTP request manually is hard and horrible. Rather than writing that nasty code, it's easier to use the StorageClient library method. We'll discuss request signing in a little more detail in section 12.2.

Finally, with the request generated and signed, you can make the request to the development service (3). The Table service will return an XML response listing all of the tables in your account, which is written to the console window, as you saw in figure 12.1.

SWITCHING TO THE LIVE SERVICE

In listing 12.1 you made a request to the development storage Table service. If you wanted to change the application to query a live service (such as a silverlightukstorage storage account) you'd need to change the URI at (1) to http://silverlightukstorage.table.core.windows.net/Tables.

It's worth pointing out that although the URI in listing 12.1 is hardcoded, you could extract it from the `storageAccount` object:

```
storageAccount.TableEndpoint.AbsoluteUri.ToString();
```

This is just the base URI; you'll still need to append `/tables` to access the Table service.

Now that you know how to list tables, let's take a look at how you can change this code to delete a table (and its underlying data).

12.1.2 Deleting tables using the REST API

To delete a table using the StorageClient library, you need to call the `DeleteTable` method of your `CloudTableClient` object, passing in the name of the table that you wish to delete. The following code would delete the Products table:

```
var storageAccount =
      CloudStorageAccount.Parse(
         ConfigurationManager.AppSettings["DataConnectionString"]);

CloudTableClient tableClient =
   storageAccount.CreateCloudTableClient();

tableClient.DeleteTable("Products");
```

If you wanted to delete the same table using the REST API directly, you could perform an HTTP `DELETE` (rather than a `GET`) request using the following URI:

```
http://silverlightukstorage.table.core.windows.net/Tables('Products')
```

To modify listing 12.1 to delete the Products table, you could replace the code at (1) with the following:

```
HttpWebRequest hwr =
      CreateHttpRequest(
      ➥ new Uri(@"http://silverlightukstorage
      ➥ .table.core.windows.net/Tables('Products')"),
      ➥ "DELETE", new TimeSpan(0, 0, 30));
```

As you can see, this code replaces the original `GET` request with a `DELETE`, and the URI has been modified. Because you no longer need to process an XML response, you'd also need to change the code at (3) as follows:

```
hwr.GetResponse();
```

Finally, because the code now uses the live Table service rather than the development storage version, you'd also need to set the correct credentials.

You've now had a chance to interact with the Table service both via the StorageClient library and by using the REST API directly, so let's look at some of the technologies used to implement the Table service REST API.

12.1.3 WCF Data Services and AtomPub

WCF Data Services (formerly known as Astoria) is a data-access framework that allows you to create and consume data via REST-based APIs from your existing data sources (such as SQL Server databases) using HTTP.

Rather than creating a whole new protocol for the Table service API, the Windows Azure team built the REST-based APIs using WCF Data Services. Although not all aspects of the Data Services framework have been implemented, the Table service supports a large subset of the framework.

One of the major advantages of WCF Data Services is that if you're already familiar with the framework, getting started with the Windows Azure Table service is pretty easy. Even if you haven't used the WCF Data Services previously, any knowledge gained from developing against Windows Azure storage will help you with future development that may use the framework.

WCF DATA SERVICES CLIENT LIBRARIES

WCF Data Services provides a set of standard client libraries that abstract away the complexities of the underlying REST APIs and allow you to interact with services in a standard fashion regardless of the underlying service. Whether you're using WCF Data Services with the Windows Azure Table service or SQL Server, your client-side code will be pretty much the same. As seen in the previous chapter, using these libraries to communicate with the Table service allows you to develop simple standard code against the Table service quickly.

ATOMPUB

The Windows Azure Table service uses the WCF Data Services implementation of the Atom Publishing Protocol (AtomPub) to interact with the Table service. AtomPub is an HTTP-based REST-like protocol that allows you to publish and edit resources. AtomPub is often used by blog services and content management systems to allow the editing of resources (articles and blog postings) by third-party clients. Windows Live Writer is a well-known example of a blog client that uses AtomPub to publish articles to various blog platforms (Blogspot, WordPress, Windows Live Spaces, and the like). In the case of Windows Azure storage accounts, tables and entities are all considered as resources.

Although WCF Data Services can support other serialization formats (such as JSON) the Table service implementation of WCF Data Services only supports AtomPub. In this book, we won't look at the AtomPub protocol specifically, but we'll point out its usage.

If you were to look at all the previous examples in this chapter (listing and deleting tables in a storage account) and compare them to the AtomPub protocol, you would see that the REST APIs map directly.

If you're interested in reading more about the AtomPub protocol (RFC 5023) you can read the full specification here: http://bitworking.org/projects/atom/rfc5023.html.

Now that you have a basic awareness of AtomPub, we can look at how the AtomPub protocol and the Atom document format are used to create a table using the Table service REST API.

12.1.4 Creating a table using the REST API

In the previous chapter, you created a table using the following StorageClient library call:

```
var storageAccount =
      CloudStorageAccount.Parse(
         ConfigurationManager.AppSettings["DataConnectionString"]);

CloudTableClient tableClient =
   storageAccount.CreateCloudTableClient();

tableClient.CreateTableIfNotExist("ShoppingCartTable");
```

Ultimately the StorageClient library just wraps the REST API that's exposed by the Table service. Let's take a look at how this is done.

CREATING A TABLE USING ATOMPUB

To create a new table, you must perform a POST request against the URI you used earlier to list and delete tables in the silverlightukstorage storage account.

```
POST http://silverlightukstorage.table.core.windows.net/Tables
```

Because the Table service implements the AtomPub protocol, the body of the POST request needs to be in the Atom document format. The following Atom document instructs the Table service to create a new table called ShoppingCartTable in the storage account:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
   <entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
      <title />
      <updated>2009-03-18T11:48:34.9840639-07:00</updated>
      <author>
         <name/>
      </author>
      <id/>
      <content type="application/xml">
        <m:properties>
          <d:TableName>ShoppingCartTable</d:TableName>      <-- Specifies name of table to create
        </m:properties>
      </content>
   </entry>
```

As you can see from this verbose piece of junk, creating a simple table requires a whole bunch of useless information that's never used.

Our two cents about the REST API method of creation

This is a complete rant, and we do apologize for it, but it has to be said. The REST API method of creating tables is unnecessarily complex and verbose. We know you probably won't care because you'll use the StorageClient library to create tables rather than the REST API. We also know that it's not Microsoft's fault—they're just following the standard. But could we not have a simpler API call?

The method of deleting the ShoppingCartTable is pretty simple; it's the HTTP verb `DELETE` with the appropriate URI, such as this one:

```
http://silverlightukstorage.table.core.windows.net/Tables
➥ ('ShoppingCartTable')
```

It's pretty simple, isn't it? Why does AtomPub have that mad method of creating a table? We can't help thinking that a simpler method of creating tables would be to use the same URI as `DELETE` and change the HTTP verb from `DELETE` to `POST`.

Instead of using AtomPub to create tables with a crazy amount of XML, you can do the same thing with the slightly easier-to-use REST API. We'll eventually get to the easiest way to create a table, which is with the StorageClient library.

CREATING THE TABLE USING THE REST API IN A CONSOLE APPLICATION

It's time for you to create a small console application that will generate the ShoppingCartTable in a storage account using AtomPub and the REST API. The listing that follows contains the code for the console application.

Listing 12.2 Creating tables using the REST API and AtomPub

```
static void Main(string[] args)
{
   HttpWebRequest hwr = CreateHttpRequest(new Uri(
     @"http://silverlightukstorage.table.core.windows.net/Tables"),
     "POST", new TimeSpan(0, 0, 30));

  string xml = string.Format(@"<?xml version=""1.0"" encoding=""utf-8""
  ➥ standalone=""yes""?><entry
  ➥ xmlns:d=""http://schemas.microsoft.com/ado/2007/08/dataservices""
  ➥ xmlns:m=""http://schemas.microsoft.com/ado/2007/08/dataservices/metadata""
  ➥ xmlns=""http://www.w3.org/2005/Atom""><title/><updated>{0:yyyy-MM-
  ➥ ddTHH:mm:ss.fffffffZ}</updated><author><name/></author><id/><content
  ➥ type=""application/xml""><m:properties><d:TableName>{1}</d:TableName>
  ➥ </m:properties></content></entry>"
  ➥ , DateTime.UtcNow, "ShoppingCartTable");

   byte[] bytes = Encoding.UTF8.GetBytes(xml);
   hwr.ContentLength = bytes.Length;

   var storageAccount =
      CloudStorageAccount.Parse(
```

```
        ConfigurationManager
        ➥ .AppSettings["DataConnectionString"]);          Gets
                                                           credentials
                                                                     Signs
    storageAccount.Credentials.SignRequestLite(hwr);                 request

    using (Stream requestStream =                        Writes message
    ➥ hwr.GetRequestStream())                        ❶  to the wire
    {
        requestStream.Write(bytes, 0, bytes.Length);
    }

    using (StreamReader sr =
        new StreamReader(hwr.GetResponse()               Makes request
        ➥ .GetResponseStream()))                         to Table service
    {
        XDocument myDocument =
            XDocument.Parse(sr.ReadToEnd());

        Console.Write(myDocument.ToString());
    }
}
```

The console application in listing 12.2 is pretty much the same as the one in listing 12.1 that listed tables. But there are a few differences between the two applications. In this case, you use the URI of your table endpoint, and you need to change the HTTP verb from `GET` to `DELETE`. You also need to convert the Atom XML you used earlier from a string to a byte array. Finally, you write the Atom XML `byte[]` to the request body.

Over the past few sections, we've looked at how you can interact with the REST API directly. In each example, you've used the StorageClient library to sign the request, but we haven't spent any time explaining that. Let's take a little time out to do that now.

12.2 Authenticating requests against the Table service

In chapter 9 (section 9.7) we described the Shared Key authentication method for BLOBs. This method of authentication is used by both the BLOB and Queue services. The Table service, however, supports two different authentication mechanisms:

- Shared Key authentication for Table service
- Shared Key Lite authentication for Table service

Let's take a look at Shared Key authentication first.

12.2.1 Shared Key authentication

Shared Key authentication for Table service is the most secure method of authenticating against the Table service using the REST API. The method for generating an authentication key is similar to the method used for BLOBs (with a few subtle differences).

In order to generate a shared key, you need to canonicalize the HTTP request and then hash it using a SHA-56 algorithm, storing the hashed value in the Authorization

header. The following value represents the Authorization header for a request, hashed using the Shared Key mechanism:

```
SharedKey devstoreaccount1:J5xkbSz7/7Xf8sCNY3RJIzyUEfnj1SJ3ccIBNpDzsq4=
```

The major difference between the shared key for the BLOB service (as well as the Queue service for that matter) and the Table service is that you don't include canonicalized headers in the signature. The following code shows how you would generate the string prior to SHA-256 hashing:

```
unhashedString =
   VERB + "\n" + Content-MD5 + "\n"
+ Content-Type + "\n" + Date + "\n"
+ canonicalizedRequest;
```

Once you've generated the string, you can hash it and stuff it into the Authorization header.

Although you can use the Shared Key mechanism with the Table service, it can only be used with the REST API directly—the WCF Data Services client doesn't support the Shared Key mechanism. Fortunately, the Table service also supports Shared Key Lite as an authentication mechanism (which is supported by the WCF Data Services client). Let's take a look at that authentication mechanism.

12.2.2 Shared Key Lite authentication

The Shared Key Lite authentication mechanism for signing a request is similar to the Shared Key method. Like the Shared Key mechanism, it will canonicalize and hash the request using a SHA-256 algorithm, storing the result in the Authorization header. The following value represents the Authorization header for a request hashed using the Shared Key Lite mechanism:

```
SharedKeyLite devstoreaccount1:0c4bknVWVmQ+L1r5jCIYFiNDkSXHata8ZYW8mjQhPLo=
```

Although Shared Key Lite follows the same process of hashing a request and uses the same key to hash the request as the Shared Key mechanism, Shared Key Lite is a lighter and less secure mechanism. The Shared Key mechanism includes more data as part of the hash, meaning a hacker would have a better chance of tampering with the request when you're using Shared Key Lite rather than Shared Key.

If you look back to listing 12.2, you can see that the request is signed prior to writing the Atom XML to the request body (at 1). That means the XML isn't part of the hash and can be tampered with.

If an HTTP request is intercepted within the Shared Access Signature time window, a hacker would be able to modify the request to create a table of a different type (such as a NastyHackerTable). The hacker would not be able to perform any other types of requests, however, because the hash prevents the HTTP verb from being tampered with.

Which authentication method should you use?

If you're using WCF Data Services to communicate with the Table service and your application runs purely within the data center, you can continue to use the Shared Key Lite mechanism. (To be honest, you don't have a choice, as it's the only authentication mechanism supported by the WCF Data Services client.)

If you're using the REST API directly, and you have an application communicating with the Table service API outside of the Windows Azure data center, or you want the highest possible security, you should use the REST API directly with Shared Key authentication.

By now, you should have a good feel for storage accounts, the REST API, AtomPub, and which operations you can perform on a storage account using both the REST API directly and the StorageClient library. Now let's look at how to perform CRUDy stuff (inserts, updates, and deletes—we'll do querying later) in conjunction with the REST API and the StorageClient library.

12.3 Modifying entities with the REST API is CRUD

Over the next few sections, we'll focus on how the REST API can be used to communicate with the Table service in regard to entities. In particular, we'll look at

- Inserting entities
- Deleting entities
- Updating entities

Before you can delete or update an entity, you'll need to insert one first.

12.3.1 Inserting entities

Before we look at how to insert an entity using the REST API, let's look at the code you'd write to store an entity in a table using the StorageClient library:

```
var shirtContext = new ProductContext();

shirtContext.AddObject("Products",
   new Product
   {
      PartitionKey = "Shirts",
      RowKey = "RedShirt",
      Name = "Red Shirt",
      Description = "A Red Shirt"
   });

shirtContext.SaveChanges();
```

The preceding example inserts a new instance of the `Product` entity into the Products table. The new `Product` will be stored in the `Shirts` partition with a `RowKey` value of

RedShirt. For a detailed description of how data is stored in the Table service and how to use a context class to insert data into a table, please refer to chapter 11.

> **The Unit of Work pattern**
>
> WCF Data Services, and therefore the StorageClient library, implement the Unit of Work pattern for saving data back to the database. This means that all changes are tracked locally (when you use the AddObject method) and then all changes are saved back to the Table service when the SaveChanges method is called.
>
> A process that doesn't use the Unit of Work pattern would not track the changes to the entities locally and apply the changes directly to the Table service when the AddObject method is called. This removes any batching or cancellation capabilities easily provided by the Unit of Work pattern.

Now that we've reminded ourselves of how to insert a new entity into a table using the StorageClient library, let's look at how this would be done using the REST API directly.

USING THE REST API

To use the AtomPub-based REST API to insert a new entity into a table, you'd need to perform an HTTP POST request to the following URI:

```
http://<storageaccountname>.table.core.windows.net/<TableName>
```

To insert a new entity into the Products table in the silverlightukstorage storage account, you'd use the following URI:

```
http://silverlightukstorage.table.core.windows.net/Products
```

To insert the entity we created at the beginning of section 12.3.1 with the StorageClient library, you'd need to define the request body with the following Atom XML:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/
➥ ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/
➥ ado/2007/08/dataservices/metadata"
➥ xmlns="http://www.w3.org/2005/Atom">
  <title />
  <updated>2009-07-27T14:22:48.8875037Z</updated>
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
      <d:Description>A Red Shirt</d:Description>
      <d:Name>Red Shirt</d:Name>
      <d:PartitionKey>Shirts</d:PartitionKey>
      <d:RowKey>RedShirt</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">0001-01-01T00:00:00</d:Timestamp>
    </m:properties>
```

```
  </content>
</entry>
```

If you look at the preceding AtomPub XML, you can see that it follows a similar format to the XML used to create the storage account table in section 12.1.4 (in the "Creating a table using AtomPub" subsection). As you can see from the preceding Atom XML, not only are the values of each property of the entity included in the XML, but so is the name and type of each property (notice that the `Timestamp` property is of type `Edm.DateTime`). Because a Table service table is effectively schemaless, and each row could contain an entity with an entirely different set of properties, it's important that the entities being inserted into the table be self-describing.

If you wanted to modify the console application in listing 12.2 to insert an entity instead of creating a table, you could replace the URI generation with the URI we defined earlier and replace the Atom XML with the preceding document.

Now that you've created your lovely entity, let's nuke it!

12.3.2 Deleting entities

In the product-management web page you built in chapter 11, you could delete shirts from the product list using the following call:

```
var shirtContext = new ProductContext();
shirtContext.DeleteObject(shirtToDelete);
shirtContext.SaveChanges();
```

Deleting an entity is similar to adding an entity when using the StorageClient library. If you wished to delete a shirt from the Products table, you'd need to add the shirt to be deleted to the context object's (`shirtContext`) tracking list using the `DeleteObject` method. All changes are again tracked locally by the context object and are only saved back to the Table service when the `SaveChanges` method is called (following the Unit of Work pattern).

Let's now take a look at using the REST API for deleting entities from a table. To delete an entity you need to make a `DELETE` request to the appropriate URI, passing in the correct table name, partition key, and row key:

```
http://silverlightukstorage.table.core.windows.net
➥ /Products(PartitionKey="Shirts", RowKey="RedShirt")
```

The preceding URI would delete an entity called `RedShirt` from the `Shirts` partition of the Products table from a storage account named silverlightukstorage. See the following listing for the code required to the delete the entity from a console application.

Listing 12.3 Deleting entities using the REST API

```
static void Main(string[] args)
{
      HttpWebRequest hwr = CreateHttpRequest(
      ➥ new Uri(@"http://silverlightukstorage
      ➥ .table.core.windows.net/Products(PartitionKey='Shirts'
      ➥ , RowKey='RedShirt')
      ➥ "), "DELETE", new TimeSpan(0, 0, 30));
```

Deletes entity from table

```
        hwr.Headers.Add("If-Match", "*");

    var storageAccount =
        CloudStorageAccount.Parse(
          ConfigurationManager
          ➥ .AppSettings["DataConnectionString"]);

    storageAccount.Credentials.SignRequestLite(hwr)

    hwr.GetResponse();
}
```

Sets up credentials and executes request

OPTIMIZING DELETE PERFORMANCE IN WEB GRIDS

As explained previously, to delete an entity using the StorageClient library, you first need to add a local copy of the entity to your context object's tracking list. In Windows client applications, this isn't such a big issue, as you'll already have a local copy of the entities. In an ASP.NET application, if you listed the entities in a grid (as in the product-management web page), you'd no longer have a local copy of the entity because each web page call is stateless.

Although you could store a copy of all entities in the ASP.NET page state, this would massively increase the page size and reduce overall performance. Similarly, storing the entities in the session would put unnecessary overhead on the web server, again affecting the performance. Even if the grid were populated from a cache, unnecessary calls to the cache would still have a slight impact on performance. So let's look at some other options.

In the product-management web page in chapter 11, we fetched the entity from the Table service (for the sake of simplicity) and then added that entity to the `shirtContext`'s tracking list. The following code shows how we used a LINQ query to fetch the data from the Table service:

```
var entity = new ProductContext();

shirtToDelete = (from item in shirtContext.Products
        where item.PartitionKey == "Shirts"
        && item.RowKey == rowkey
                 select item).First();

shirtContext.DeleteObject(entity);

shirtContext.SaveChanges();
```

Refetching the entity to be deleted isn't something you should consider in a production environment, as any unnecessary calls to the Table service will impact the performance of your application and add to the overall running cost of your service (calls to the Table service are billable).

As you saw in our discussion of the REST API, you really don't need all the data of the object. In fact, you only need the partition key and row key, so rather than fetching the whole object, you could construct a lightweight version of the object to be deleted.

To do this, you can define a lightweight sister class of the Product class that only contains the PartitionKey and RowKey values. As long as the object held in the tracking list holds the correct PartitionKey and RowKey (which uniquely identify an entity), you'll have enough information to perform the delete—there's no need to fetch every property of the entity.

The following code shows how you can delete an entity from the Table service using a lightweight instance:

```
var shirtContext = new ProductContext();
shirtContext.DeleteObject
(
   new ProductKey
   {
      PartitionKey = "Shirts",
      RowKey = e.CommandArgument.ToString()
   }
);
   shirtContext.AttachTo("Product", entity, "*");
shirtContext.SaveChanges(saveOptions);
```

As you can see in the preceding code, there's no need to fetch the entity from the Table service because you already know the PartitionKey (Shirts), and the RowKey can be extracted from the command argument of the Delete button. This optimization saves you that extra fetch.

Now that we've explored both inserting and deleting, it's time to complete the set and look at updates.

12.3.3 Updating entities

When deleting an object using the StorageClient library, you need to keep track of the objects to be deleted in the context object for the Products table. You can use similar logic to update objects in your application.

Here's an example:

```
var shirtContext = new ProductContext();

shirtToUpdate = (from item in shirtContext.Products
                 where item.PartitionKey == "Shirts"
                 && item.RowKey == "RedShirt"
                 select item).First();
shirtToUpdate.Description = "I have been modified";
shirtContext.UpdateObject(shirtToUpdate);
shirtContext.SaveChanges(saveOptions);
```

The preceding code retrieves the RedShirt entity from the Shirts partition in the Products table. The code then modifies the description of the entity and saves the changes back to the Products table.

MERGING DATA

By default, the SaveChanges method will *merge* any changes made to the object back to the entity stored in the Table service, rather than performing a replacement update.

Before we can explain what this means, let's look at an extract of Atom XML that describes the entity held in the Table service:

```
<content type="application/xml">
    <m:properties>
      <d:Description>A Red Shirt</d:Description>
      <d:Name>Red Shirt</d:Name>
      <d:PartitionKey>Shirts</d:PartitionKey>
      <d:RowKey>RedShirt</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">0001-01-01T00:00:00</d:Timestamp>
    </m:properties>
  </content>
```

By choosing to merge the data, you can efficiently send data back to the Table service by only sending the modified data instead of the full entity. Table 12.1 shows how this would work in three scenarios:

- *Remote*—A remote copy of the entity is stored in the Products table.
- *Local*—A local copy of the entity is used.
- *Merged*—The changes in the local version of the entity are merged with the remote version.

Table 12.1 Merging data with updates

Scenario	Partition key	Row key	Name	Description
Remote	Shirts	RedShirt	RedShirt	A Red Shirt
Local	Shirts	RedShirt	RedShirt	A Pink Shirt
Merged	Shirts	RedShirt	RedShirt	A Pink Shirt

As you can see in table 12.1, the only property that has changed for the entity is the description. This means that the client application doesn't need to send back the name property in the Atom XML describing the entity. The following extract of the Atom XML describes what would be returned to the Table service as part of the merge operation:

```
<content type="application/xml">
    <m:properties>
      <d:Description>A Pink Shirt</d:Description>
    </m:properties>
  </content>
```

If you need to replace the entity stored in the Table service with your local version rather than performing a merge, you can use the following setting in your client code:

```
shirtContext.SaveChangesDefaultOptions =
    SaveChangesOptions.ReplaceOnUpdate;
```

In this case, the Atom XML sent using the REST API would contain the full description of the entity rather than just the changed properties.

USING THE REST API TO MERGE OR UPDATE

When you use the REST API to update or merge data, you're really using a combination of the delete and insert REST API functions.

The URI to update or merge the local entity back to the Table storage is the same URI as for the delete operation:

```
http://silverlightukstorage.table.core.windows.net
➥ /Products(PartitionKey='Shirts', RowKey='RedShirt')
```

As you can see, the URI needs to specify the `PartitionKey` and `RowKey` of the entity being modified. Depending on the operation you're performing, you should set the HTTP verb to either `MERGE` or `PUT`. Finally, the body of the HTTP request should be set to the AtomPub XML document that describes the entity (this is the same as the XML used to create the entity).

If you wish to modify the console application from section 12.1.1 to merge instead of insert, you would need to change the URI and HTTP verbs in the code.

Finally, you would need to add a new `If-Match` header to the request. This header is used to ensure that the data held in the remote version of the entity has not changed since you grabbed the local version. If you wish to ensure that data is only modified if the data is unchanged, you should set the `If-Match` header to the e-tag that was originally returned with the entity.

If you wish to perform an unconditional update, the value of the `If-Match` header should be set to `"*"`.

In this section, you've learned how to perform inserts, updates, and deletes against your entities. But you're unlikely to work with single entities, so it's time to learn about some of the complications of updating data—batching and transactions.

12.4 Batching data

In the previous sections, you used both the StorageClient library and the REST API to insert new entities into the Products table. In this section, we'll look at how you can both improve performance and perform transactional changes by batching up data.

The following code inserts multiple entities into the Products table using the StorageClient library:

```
var shirtContext = new ProductContext();

for (int i = 0; i < 10; i++)
{
   shirtContext.AddObject("Products",
   new Product
   {
      PartitionKey = "Shirts",
      RowKey = i.ToString(),
      Name = "Shirt" + i.ToString(),
      Description = "A Shirt"
   });
}
shirtContext.SaveChanges();
```

```
12 21.950029   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
16 22.241154   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
17 22.251847   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
22 22.633432   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
23 22.641256   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
27 22.898821   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
28 22.906475   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
32 23.163614   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
33 23.171723   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
37 23.429996   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
38 23.438125   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
42 23.694816   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
43 23.702653   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
47 23.960189   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
48 23.967479   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
52 24.226081   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
53 24.233846   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
57 24.491098   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
58 24.499442   192.168.0.5     65.55.33.205    HTTP   POST /ProductTable HTTP/1.1  (application/atom+xml)
62 24.756697   65.55.33.205    192.168.0.5     HTTP   HTTP/1.1 201 Created  (application/atom+xml)
```

Figure 12.2 By default, the context object will save each entity with an individual request rather than saving them all in a batch.

The preceding code will create 10 new shirts and add each new shirt to a list of objects that are to be tracked; it does this by calling the `AddObject` method on the `shirtContext` object. Following the Unit of Work pattern, the context object won't send any changes to the Table service until the `SaveChanges` method is called. It will then iterate through the list of tracked objects and insert them into the Products table.

By default, the `SaveChanges` call will insert the entities into the table one by one rather than batching the inserts into a single call. Figure 12.2 shows the HTTP traffic for the preceding call, captured by using Wireshark (a packet-sniffing tool).

As you can see from figure 12.2, to insert 10 shirts, the application must perform 10 HTTP `POST` requests to the Table service. This method can cause performance problems if you're inserting a large number of entities and your application is outside of Windows Azure or your web or worker role isn't affinitized to the same data center as your storage account.

> **WARNING** Due to latency, inserting 10 shirts using the preceding code took 4 seconds between our local machine and the live Table service. When running the same code as a web role in the Windows Azure data center, it took milliseconds.

Although minimizing latency will give large performance benefits, you can gain larger performance improvements by batching up inserts into single calls using entity group transactions.

> **NOTE** Due to the flexible nature of the Windows Azure platform, you can host your storage account and your web and worker roles in different data centers. As you can see from the previous example, this flexibility comes at a price: latency. For the best performance, always affinitize your web roles, worker roles, and storage service to the same data center to minimize latency.

12.4.1 Entity group transactions

Entity group transactions are a type of batch insert where the whole batch is treated as a transaction, and the whole thing either succeeds or is rolled back entirely. First, let's look at how batch inserts are done.

Passing `SaveChangesOptions.Batch` as a parameter into the `SaveChanges` method calls will batch up all changes into a single HTTP `POST`:

```
shirtContext.SaveChanges(SaveOptions.Batch);
```

Batching up the data like this reduced our insert of 10 shirts (from the local machine to the live service) from 4 seconds to 1 second.

The `SaveOption` parameter can also be passed in with the call to the `SaveChanges` method to specify what happens if the inserts aren't entirely successful:

- `SaveOptions.None`—By default, when no `SaveOption` is passed, or when `SaveOptions.None` is passed, as part of the `SaveChanges` method, and a tracked entity fails to be inserted, the context object will stop attempting to save any further entities. Any entities that were saved successfully won't be rolled back and will remain in the table.
- `SaveOptions.ContinueOnError`—If this option is passed as part of the `SaveChanges` call, and an entity fails to save, the context object will continue to save all other entities.
- `SaveOptions.Batch`—If this option is passed as part of the `SaveChanges` call, all entities will be processed as a batch in the scope of a single transaction—known as an entity group transaction. If any of the entities being inserted as part of the batch fails to be inserted, the whole batch will be rolled back.

These are the rules for using entity group transactions:

1. A maximum of 100 operations can be performed in a single batch.
2. The batch may not exceed 4 MB in size.
3. All entities in the batch must have the same partition key.
4. You can only perform a single operation against an entity in a batch.

In this book, we won't discuss the REST implementation of entity group transactions due to the complexity of the implementation. But it's worth noting that if you decide to use the REST implementation, the Table service only implements a subset of the available functionality. As of the PDC 2009 release, the Table service only supports single changesets (a changeset being a set of inserts, updates, or deletes) within a batch.

> **NOTE** If you're interested in looking at the REST implementation of batching, you should look up the "Performing Entity Group Transactions" MSDN article: http://msdn.microsoft.com/en-us/library/dd894038.aspx.

Entity group transactions are executed using an isolation method known as *snapshot isolation*. This is a standard method of isolation used in relational databases such as SQL Server or Oracle; it's also known as multiversion concurrency control (MVCC). A snapshot of the data is taken at the beginning of a transaction, and it's used for the duration of the transaction. This means that all operations within the transaction will use the same set of isolated data that can't be interfered with by other concurrent processes. Because the data is isolated from all other processes, there's no need for locking on the

table, meaning that operations can't be blocked by other processes. On committing the transaction, if any modified data has been changed by another process since the snapshot began, the whole transaction must be rolled back and retried.

12.4.2 Retries

In order to handle the MVCC model, your code must be able to perform retries. The ability to handle retries is built into the StorageClient library and can be configured using the following code:

```
shirtContext.RetryPolicy =
    RetryPolicies.Retry(5, TimeSpan.FromSeconds(1));
```

The preceding retry policy will reattempt the `SaveChanges` operation up to five times, retrying every second. If you don't wish to set a retry policy, you can always set the policy as `NoRetry`:

```
shirtContext.RetryPolicy = RetryPolicies.NoRetry;
```

If you need more complicated retry polices with randomized back-off timings, or if you wish to define your own policy, this can also be achieved by setting an appropriate retry policy. Unfortunately, if you're using the REST API directly, you'll need to roll your own retry logic.

In order to make use of the standard retry logic, you'll need to use the `SaveChangesWithRetries` method rather than the `SaveChanges` method, as follows:

```
shirtContext.SaveChangesWithRetries();
```

> **Use retries for queries too**
>
> Although retry policies are vital when using entity group transactions, they can also be useful when querying data. Your web and worker roles are based in the cloud and can be shut down and restarted at any time by the Fabric Controller (such as in a case of a hardware failure), so to provide a more professional application, it may be advisable to use retry policies when querying data.

So far we've covered the modification of data in quite a lot of detail. But entity group transactions can also be useful for querying data. With that in mind, it's worth breaking away from data updates and focusing on how to retrieve data via the REST API.

12.5 Querying data

In this section, we'll take a look at how to query data held in the Table service by using both the StorageClient library and the REST API directly. The knowledge that you gain from this section will come in useful when we look at how to store data efficiently. In particular we'll look at how you can

- Retrieve entities using the REST API
- Query using LINQ

- Filter data using the REST API
- Filter data using LINQ
- Select data using LINQ
- Page data

To get started, let's look at how to retrieve data from a table using the REST API.

12.5.1 Retrieving all entities in a table using the REST API

In this section we'll look at how to build a small console application that will display all the entities in the Product table. This application is similar to the one that you built earlier to list tables in storage accounts.

The base URI used to query the Products table is the same base URI you used to insert, update, and delete table entities. To return all shirts stored in the Products table in the development storage account, you would make an HTTP `GET` request using the following URI:

```
http://127.0.0.1:10002/devstoreaccount1/Products
```

The code in listing 12.4 will return all entities in the Products table and display the result in the console window.

Listing 12.4 Listing the entities in the Products table using the REST API

```
static void Main(string[] args)
{
   HttpWebRequest hwr =
      CreateHttpRequest(new Uri(@"http://127.0.0.1:10002
      ➥ /devstoreaccount1/Products"), "GET",
      ➥ new TimeSpan(0, 0, 30));

   var storageAccount =
      CloudStorageAccount.Parse(
        ConfigurationManager
        ➥ .AppSettings["DataConnectionString"]);

   storageAccount.Credentials.SignRequestLite(hwr)

   using (StreamReader sr =
          new StreamReader(
          ➥ hwr.GetResponse()
          ➥ .GetResponseStream()))
   {
      XDocument myDocument = XDocument.Parse(sr.ReadToEnd());
      Console.Write(myDocument.ToString());
   }
}
```

(1) Specifies URI for request

Executes request and displays output

The code in listing 12.4 is pretty much the same code as in listing 12.1 (which listed all the tables in a storage account). The only modification you need to make to that code is to change the URI for the HTTP request at (1) to the URI of the Products table.

The REST API call that you used to list all product entities adheres to the AtomPub protocol, so the result that's returned to the console window will display the entities from the Products table in Atom XML format, as shown in the following listing.

Listing 12.5 Atom XML output from the console application in listing 12.4

```
<feed xml:base="http://127.0.0.1:10002/devstoreaccount1/" xmlns:d="http://
     schemas
.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://
     schemas.microsoft.com/
ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Products</title>
  <id>http://127.0.0.1:10002/devstoreaccount1/Products</id>
  <updated>2009-08-01T11:23:48Z</updated>
  <link rel="self" title="Products" href="Products" />
  <entry m:etag="W/&quot;datetime'2009-07-23T19%3A55%3A38.7'&quot;">
    <id>http://127.0.0.1:10002/devstoreaccount
    ➥ /Products(PartitionKey='Shirts',RowKey='BlueShirt')</id>
    <title type="text"></title>
    <updated>2009-08-01T11:23:48Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Products" href="Product(PartitionKey=
    ➥ 'Shirts',RowKey='BlueShirt')" />
    <category term="AiAChapter7Web_WebRole.Products"
    ➥  scheme="http://schemas.
microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:Name>Blue Shirt</d:Name>
        <d:Description>A Blue Shirt</d:Description>
        <d:Timestamp m:type="Edm.DateTime">2009-07-
        ➥ 23T19:55:38.7</d:Timestamp>
        <d:PartitionKey>Shirts</d:PartitionKey>
        <d:RowKey>BlueShirt</d:RowKey>
      </m:properties>
    </content>
  </entry>
  <entry m:etag="W/&quot;datetime'2009-07-23T19%3A13%3A28.09'&quot;">
    <id>http://127.0.0.1:10002/devstoreaccount
    ➥ 1/Products(PartitionKey='Shirts',RowKey='RedShirt')
    </id>
    <title type="text"></title>
    <updated>2009-08-01T11:23:48Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Products" href=
    ➥ "Product(PartitionKey='Shirts',RowKey='RedShirt')" />
    <category term="AiAChapter7Web_WebRole.Products"
    ➥ scheme="http://schemas.microsoft.com/ado
    ➥ /2007/08/dataservices/scheme" />
    <content type="application/xml">
```

```
        <m:properties>
          <d:Name>red shirt</d:Name>
          <d:Description>a red shirt</d:Description>
          <d:Timestamp m:type="Edm.DateTime">2009-07-
          ➥ 23T19:13:28.09</d:Timestamp>
          <d:PartitionKey>Shirts</d:PartitionKey>
          <d:RowKey>RedShirt</d:RowKey>
        </m:properties>
      </content>
    </entry>
</feed>
```

The Atom XML output in listing 12.5 returns all entities stored in the Products table (`RedShirt` and `BlueShirt`). As you can see, the returned XML is very descriptive (and also verbose), including the name of the property, the value, and the data type.

The verbosity of the returned XML means that the returned datasets are usually pretty large. You should be careful to cache data whenever possible and return the minimum amount of data required.

The verbosity of Atom XML

Hopefully, in the future, the Windows Azure Table service team will support a less verbose serialization format, such as JSON, and will also support local data shaping (explained later on in this section). JSON would be an ideal format to support because WCF Data Services (but not the Table service implementation) already supports this method of serialization.

Using JSON would require few changes to your application code, but you'd gain large benefits in terms of reduced bandwidth. The following code shows how the previously returned Atom XML could be represented in JSON:

```
Products:
 [
     {
        "Name" : "blue shirt"
        "Description" : "a Blue Shirt"
        "Timestamp" : "2009-07-23T19:13:28.09"
        "PartitionKey" : "Shirts"
        "RowKey" : "BlueShirt"}
    },
    {
        "Name" : "red shirt"
        "Description" : "A Blue Shirt"
        "Timestamp" : "2009-07-23T19:13:28.09"
        "PartitionKey" : "Shirts"
        "RowKey" : "RedShirt"}
    },
]
```

As you can see, the JSON representation is much more readable and terse, meaning that the size of the returned documents would be greatly reduced and will therefore improve the speed of your application. Hopefully this will be supported in future versions of the Table service.

12.5.2 Querying with LINQ

Because the Table service is implemented using ADO.NET Data Services, you can use the WCF Data Services client library to perform server-side queries using LINQ rather than querying the REST API directly.

The following code shows how the Products table was exposed in the `ProductContext` class created in listing 11.2 (in chapter 11):

```
public DataServiceQuery<Product> Products
{
   get{return CreateQuery<Product>("Products");}
}
```

Rather than executing and returning a list of products from the Products table stored in the Table service, the `Products` property will generate and return a new query that won't be executed until the collection is enumerated. Because the execution of the `DataServiceQuery` is deferred, you can modify the returned query to include any filters that you may require prior to executing the query.

Because `DataServiceQuery` implements the `IQueryable` interface, you can define the query that should be executed by the Table service in your application by using LINQ. The following code is a LINQ query that will return all the products in the `Shirts` partition of the Products table:

```
var shirts = from shirt in shirtContext.Products
             where shirt.PartitionKey == "Shirts"
             select shirt;

foreach (var shirt in shirts)
{
}
```

In the preceding query, the `Products` property of `shirtContext` is `IQueryable`, so you can make this the data source of a LINQ query. Because the query won't be passed to the Table service for execution until the `for` loop is executed, you can add additional filter criteria to the query (such as restricting the returned data to only those shirts that reside in the `Shirts` partition).

> **NOTE** If you don't include the where criteria in the preceding LINQ statement, the underlying REST API that's executed would be the same as the call made in listing 12.4.

12.5.3 Filtering data with the REST API

In the previous section, we looked at a LINQ query that included a `WHERE` clause, restricting the data returned from the Table service to include only those shirts in the `Shirts` partition. We'll now look at how you can modify your use of the REST API to perform server-side filtering.

RETURNING A SINGLE ENTITY

As stated previously, the combination of the PartitionKey and RowKey uniquely identifies an entity in a table. If you wish to return a single entity from a table, and you know these two values, you can efficiently return the entity. The following URI would return the RedShirt entity from the Shirts partition:

```
http://127.0.0.1:10002/devstoreaccount1
➥ /Products(PartitionKey='Shirts',RowKey='RedShirt')
```

To execute this query, you could modify the console application in listing 12.4, replacing the URI with one here. The Atom XML returned from the query would be similar to the data returned in listing 12.5, but it would only contain the RedShirt entity.

QUERYING ENTITIES

If you need to return zero or more entities based upon some filter criteria (such as all shirts that cost $10), you could use the following URI to define the REST query:

```
http://127.0.0.1:10002/devstoreaccount1/Products$filter=<query>
```

Just replace the <query> in the URI with the filter that you want to run server-side. Again, you can modify the console application in listing 12.4 to use this URI. When the query is executed, all entities that match the query will be returned in Atom XML format, as in listing 12.5.

QUERY EXPRESSIONS

Let's take a quick look at the syntax of the query expression applied in the REST API. (We'll look at using these queries in the more familiar LINQ syntax in the next section.)

As of the PDC 2009 release, the Table service only supports the query expressions listed in table 12.2.

Table 12.2 Query expressions supported by the Table service

Supported query expression	Description (C# equivalent operator)
eq	Equals (==)
gt	Greater than (>)
ge	Greater than or equal to (>=)
lt	Less than (<)
le	Less than or equal to (<=)
ne	Not equal to (!=)
and	And (&&)—Boolean properties only
not	Not (!)—Boolean properties only
or	Or (||)—Boolean properties only

You could apply these queries to a REST API query to return all shirts with the description “A Red Shirt”. The URI would look like this:

```
http://127.0.0.1:10002/devstoreaccount1/Products?$filter=
➥ Description%20eq%20'A%20Red%20Shirt'
```

To return all shirts in the `Shirts` partition that have the description “A Red Shirt”, you’d use the following URI:

```
http://127.0.0.1:10002/devstoreaccount1/Products?$filter=
➥ PartitionKey%20eq%20'Shirts'%20and%20Description%20eq%20
➥ 'A%20Red%20Shirt'%
```

12.5.4 Filtering data with LINQ

In the previous section, we looked at how to filter queries server-side using the REST API. We’ll now look at how the REST API maps onto the LINQ queries.

As you may have guessed, LINQ queries eventually get resolved to the REST API URIs like the ones we looked at in the previous section. This means that although LINQ has a large and rich syntax, only those methods that map directly to the REST API can be supported.

While you’re debugging a LINQ query in Visual Studio, you can either hover over or put a watch on a context object (such as `shirtContext` in figure 12.3) and you’ll be able to see the underlying REST API query. Figure 12.3 shows the REST API query for a LINQ query that returns all products in the `Shirts` partition.

```
var shirtContext = new ProductContext();
    shirts | {http://silverlightukstorage.table.core.windows.net/ProductTable()?$filter=PartitionKey eq 'shirts'}
var shirts = from shirt in shirtContext.ProductTable
             where shirt.PartitionKey == "Shirts"
             select shirt;
```

Figure 12.3 Mapping a LINQ query back to the REST API

Let’s now look at the typical queries that you’ll be able to perform.

EQUALITY COMPARISONS

As you can see from the list in table 12.2, only equality, range comparisons, and Boolean comparisons can be performed using the Table service. The following queries are typical equality comparisons that can be performed:

```
where shirt.RowKey == "Red Shirt"
```

or

```
where shirt.Description != "A Red Shirt"
```

or

```
where shirt.Partition == "Shirts"
  && shirt.Description != "A Red Shirt"
```

RANGE COMPARISONS

The Table service supports the filtering of range data using range queries. For example, the following `WHERE` clause will return those shirts priced at $50 or more, and less than $70:

```
where shirt.Price >= 50 && shirt.Price < 70
```

Because data is stored in the Table service as native types, rather than as string representations, the Table service will perform comparison routines using the native types rather than string comparisons. The following query will return all shirts whose price is greater than or equal to $50.20:

```
where shirt.Price >= 50.20
```

If this query were performed as a string comparison (which you would have to do with Amazon SimpleDB), it would not return shirts priced at $60 (because there are fewer characters in the string than 50.20) unless the price were stored as 60.00.

In Windows Azure Table service, the only time you need to worry about performing equivalent string comparisons is if you store a non-native string type as a partition or row key. Partition and row keys are always represented as strings in the Table service, so if you need to perform range comparisons on these entities, you'll need to ensure that the string lengths of the stored data are correct.

BOOLEAN LOGIC

As stated earlier, the Table service does respect property types. This means you can perform Boolean logic against entity properties that are defined as `bool`. For example, you could perform the following `WHERE` clause against a shirt that's marked as a genuine Hawaiian shirt:

```
where shirt.IsMadeInHawaii && shirt.Price > 50
```

PREFIX QUERIES

Using the range comparison and Boolean logic, you can manipulate your LINQ and REST queries to return all entities that start with a particular string. For example, if you wanted to return all shirts that were present in any of `partition1`, `partition2`, `partition3`, or `partition4`, you could use the following query:

```
where shirt.PartitionKey.CompareTo("Partition1") >= 0 &&
     shirt.PartitionKey.CompareTo("Partition5") < 0
```

LINQ TO OBJECTS QUERIES

Even though only a small subset of the LINQ syntax is available to be executed by the Table service, you can still perform in-memory LINQ queries (LINQ to Objects). In-memory LINQ queries do provide full access to the LINQ syntax, but all queries are executed on the client side, so they require the full dataset to be returned by the Table service first. This approach isn't suitable for situations where you're working with a large set of data.

By now you should have a taste of the types of queries that you can perform against the Table service. Let's now look at how you can shape the data that's returned from your queries.

12.5.5 Selecting data using the LINQ syntax

As you'll have noticed in the supported LINQ syntax list (table 12.2), there was no mention of the SELECT statement. You can use the SELECT statement to return the entire entity, but you can't use SELECT to instruct the Table service to only return a subset of the entity properties.

RETURNING AN ENTIRE ENTITY USING SELECT

To illustrate the limitations of using SELECT, let's look again at a LINQ query that returns a product entity in its entirety:

```
var shirts = from shirt in shirtContext.Products
             where shirt.PartitionKey == "Shirts"
             select shirt;
```

This LINQ query was used earlier to return all entities that reside in the Products table. The following code is an Atom XML extract of one of the entities returned by the preceding LINQ query:

```
<content type="application/xml">
   <m:properties>
    <d:PartitionKey>Shirts</d:PartitionKey>
    <d:RowKey>shirts0</d:RowKey>
    <d:Timestamp m:type="Edm.DateTime">
       2009-07-29T21:14:45.022Z
    </d:Timestamp>
    <d:Description>A Shirt</d:Description>
    <d:Name>shirtshirts0</d:Name>
 </m:properties>
</content>
```

As you can see from the XML for the returned entity, every property of the product entity is returned by the Table service (PartitionKey, RowKey, Timestamp, Description, and Name).

If the Products table was held in SQL Server rather than the Table service, and the LINQ statement was executed against the database using LINQ2SQL or LINQ2Entities, the following SQL statement would be generated and executed on the SQL Server database:

```
SELECT PartitionKey, RowKey, Timestamp, Description, Name
FROM Products
WHERE PartitionKey = 'Shirts'
```

SHAPING THE QUERY

If you're using LINQ2SQL or LINQ2Entities with a SQL Server database, and you don't need to return the entire entity, you might choose to write a more efficient LINQ query that only requests and returns specific columns from the SQL Server Database. The following SQL statement requests just the Name and Description properties:

```
SELECT Name, Description
FROM Products
WHERE PartitionKey="Shirts"
```

The preceding SQL statement is less intensive to execute on the server (as there is less data being queried) and it will also use less network bandwidth due to the reduced dataset being returned to the application.

When you're using LINQ2SQL or LINQ2Entities, you can modify your less efficient LINQ statements, like this:

```
select entity
```

to generate the more efficient SQL statement:

```
select new
   {
      Name = newShirt.Name,
      Description = newShirt.Description
   };
```

This would modify the previous `select entity` LINQ statement so it looks like this:

```
var shirts = from shirt in shirtContext.Products
             where shirt.PartitionKey == "Shirts"
             select new
             {
                Name = newShirt.Name,
                Description = newShirt.Description
             };
```

Unfortunately, because the Table service doesn't support data shaping using the `SELECT` statement, you'd get a nasty exception if you attempted to run the preceding LINQ query. As a result, whenever you execute queries against the Table service, every property of the entity will always be returned as part of the query.

If you really do need to shape the returned data in your application, and you don't mind that the entire entity will be returned from the server, you can always shape it locally using the following code:

```
var shirts = from newShirt in
             (
                from shirt inshirtContext.Products
                where shirt.PartitionKey == "Shirts"
                select shirt
             ).ToList()
             select new
             {
                Name = newShirt.Name,
                Description = newShirt.Description
             };
```

The preceding code uses the same LINQ query as in section 12.5.2 to filter the data in the Table service, but this time it returns the entire entity. By calling the `ToList` method on the inner LINQ query, you can ensure that the server-side query will return all properties of the entity.

Finally, the result of the `ToList` method is fed into the outer LINQ2Object query, which performs in-memory shaping of the data, returning a new anonymous type containing the two properties that you want.

You should be aware that although this query returns the entities shaped as you specify, it won't improve server-side or bandwidth efficiency. If you have a very large entity with an infrequently used property that you don't need in a particular query, this unused property will still be returned by the Table service.

12.5.6 Paging data

By default, `SELECT` queries will only return 1,000 items in a single result set. Not only is this the default amount of data returned, but it's also the maximum amount of data returned.

If you wish to return a smaller amount of data, you can set this with the `Take` statement in LINQ, as follows:

```
(from shirt inshirtContext.Products
where shirt.PartitionKey == "Shirts"
select shirt).Take(100);
```

The preceding LINQ statement will return the first 100 items in the `Shirts` partition. The LINQ `Take` extension method will be resolved to the following query string parameter in the URI for the REST API call:

```
&top=100
```

If more items could be returned by the query than are present in the result set, continuation tokens will be provided to allow you to retrieve the next set of data in the query. This method of using continuation tokens effectively provides a method of paging.

If you wanted to return all items in the `Shirts` partition of the Products table, but it potentially contains more than 1,000 items, you could run the following REST API query:

```
http://silverlightukstorage.table.core.windows.net/Products?$filter
➥ =PartitionKey%20eq %20'Shirts'
```

Because more than 1,000 items would normally be returned in the query, you'll receive the following continuation tokens in the response:

```
x-ms-continuation-NextPartitionKey: Shirts
x-ms-continuation-NextRowKey: 1001
```

If you wanted to return all the items in the S`hirts` partition that were not returned as part of the original query, you could retrieve the next set of data using the following query:

```
http://silverlightukstorage.table.core.windows.net/Products?$filter
➥ =PartitionKey%20eq %20'Shirts'&NextPartitionKey=Shirts&NextRowKey=1001
```

The preceding query would return all products in the `Shirts` partition from `RowKey` 1001 onwards, or at least the next 1,000 entities.

12.6 Summary

In this chapter, we've taken quite a deep dive into using both the StorageClient library and the Table service REST API.

You've learned how to use the REST API to modify tables in a storage account, and to perform CRUD functions against those tables. We also looked at the AtomPub format and at how this impacts your applications. Finally, we looked at how you can efficiently update and query data.

By gaining an understanding of the REST API, you can maximize the performance of your applications and understand the limits of the service.

Based on the last two chapters, you should now have a pretty good idea of how to use the Table service. Later on in the book, we'll look at how you can apply this knowledge in your applications.

For now, we'll continue with our focus on structured data and look at SQL Azure, the relational database in the cloud.

13 SQL Azure and relational data

This chapter covers

- Leveraging the power of SQL Server for cloud applications
- Easy ways of migrating an on-premises database to the cloud
- Avoiding potholes during migrations

Most applications that work with data today use a relational data model. It's a model we're all familiar with, and we know how to manage and develop with it. SQL Server has been with us for many years, and it's going to be with us as we move into the cloud.

Over the years, SQL Server has matured to meet the different needs of its customers. It started as a spunky departmental server and moved into the desktop space, mobile device space, and the enterprise space. The relational data engine has been the first component to make each of these moves. The rest of the components usually follow shortly after, such as Integration Services (SSIS), Reporting Services (SSRS), and Analysis Services (SSAS). The cloud isn't any different. The SQL

Server team is bringing the data engine to the cloud first, and the rest of the components of the system will follow it quickly.

13.1 The march of SQL Server to the cloud

When Azure was first released as a CTP in November of 2008 at the PDC, SQL Azure wasn't on stage. There was something like SQL involved, but it wasn't a relational engine. It was more like the Azure Table service, but geared for true enterprise needs (beyond the massive scale Tables gives you).

Developers were scared and confused. If they were going to move to the cloud, they surely needed a data platform they were comfortable with—something that made it easy for them to migrate their applications without having to rewrite the data tier. Microsoft heard this feedback, tabled what they had (pun unfortunately intended), and delivered what we now call SQL Azure.

Developers wanted something like SQL in the cloud because they needed a data platform they could easily migrate to from on-premises databases. SQL Azure not only gives you the relational database you're used to, but also supports all of the common tools and APIs you're used to working with, such as ADO.NET and TDS.

You can easily port a traditional database from SQL Server to SQL Azure in a matter of days, if not hours. There are some restrictions, and we'll cover those, but for the most part it's easy and straightforward.

Future versions of SQL Azure are sure to contain the other components of SQL Server. Many customers we speak with want to keep their applications local but move all of their business intelligence applications and their heavy computing needs to the cloud. This is a scenario we have no doubt will be supported at some future date.

13.2 Setting up SQL Azure

The first step in creating a database in SQL Azure is to log into your Azure portal and select the SQL Azure tab on the left.

If you haven't created a SQL Azure server yet, you'll be prompted to create a SQL Azure management account, which is like the SA account for a normal SQL Server installation. You'll need to provide a username and password, and to specify in which data center region you'd like the server to be provisioned. You'll most likely want it in the same region that your Azure applications are running in. The steps are pretty easy, as seen in figure 13.1.

Once your account is created, your server will be provisioned and you'll be given the server name. This screen is where you're able to manage your databases and your SQL Azure firewall settings. In figure 13.2, the server name is mlwwmqca6u. Sometimes you'll need the fully qualified server name, which in this case is mlwwmqca6u.database.windows.net.

Once you have a SQL Azure server provisioned, you'll be able to create a database in that server.

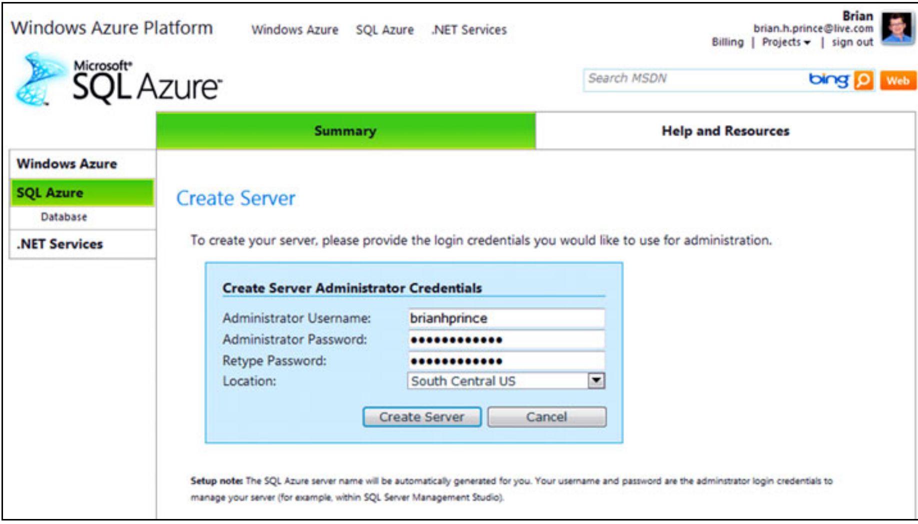


Figure 13.1 Creating a server administrator account for SQL Azure is easy. This is essentially an SA account in the cloud. You can't leave your password blank.

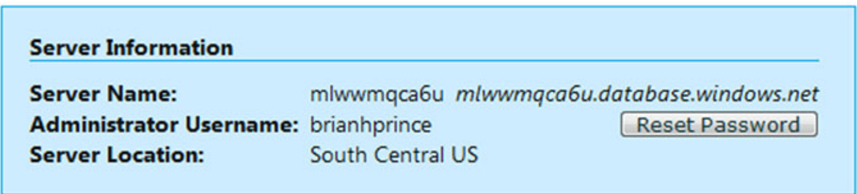


Figure 13.2 Your SQL Azure server name. It's short and easy to remember. I know I can turn this one into a backronym somehow.

An on-premises SQL Server setup has a lot of infrastructure concerns that go along with it: how much RAM it has, what the file group configuration is, what resources are assigned to which instance, what the appropriate resource governing levels should be, and so on. You have none of these concerns with a SQL Azure server, which is just a logical grouping of the databases you've created and doesn't represent a pool of resources at all. The only resource shared by a SQL Azure server is the firewall configuration. Your SQL Server in the cloud isn't really a server, but a collection of settings. It mimics a server when you connect to it, but your databases are actually spread across a farm of commodity hardware running SQL Server.

13.2.1 Creating your database

When you set up a SQL Azure account, you'll be given your own master database, which is listed in the Databases tab shown in figure 13.3. You'll need to use the Create Database button at the bottom of the list to start creating your own databases through the portal.

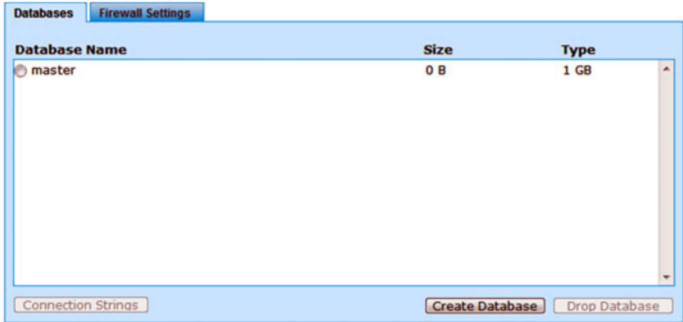


Figure 13.3 Your list of databases and the buttons to manage them. You haven't created any yet, but you were given your own master database. You aren't charged for the master database.

You can also create a database manually by connecting to the master database with the SQLCMD command prompt application and executing a `CREATE DATABASE` command. A sample is shown in figure 13.4. You can provide a `MAXSIZE` parameter to select either the 1 or 10 GB size. If you don't select a size, a 1 GB database will be created. As of the writing of this book, a 50 GB database has been announced, but not yet released.

> **NOTE** You won't be able to connect with SQLCMD, or anything else, until you configure the SQL firewall. We'll cover this in a little bit.

13.2.2 Connecting to your database

Once you click the Create Database button, you'll be prompted to provide a database name and choose the size limit for the database. Your options for a size limit are currently 1 GB and 10 GB. The reasons for these sizes will be discussed later in this chapter.

For this example, we created a database called AzureInAction, with a size of 1 GB. Once you have created your database, you can retrieve the connection string to that

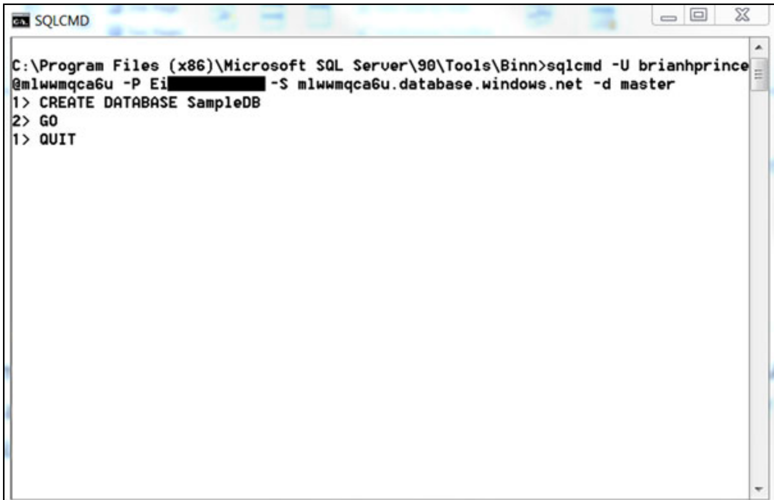


Figure 13.4 Adding a new database to SQL Azure with the SQLCMD tool, just like in the old days. You have to have the firewall configured before you do this.

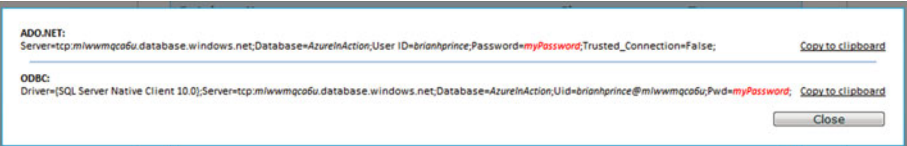


Figure 13.5 SQL Azure connection strings are displayed in the Azure portal, making it easy to know what you need to use in your code. Sometimes changing the connection string is all you need to do to move your application's data to the cloud.

database by selecting it in the list and using the Connection Strings button near the bottom of the list. The connection string displayed (as shown in figure 13.5) will contain the full address of the server, your username, and a place for your password.

You'll eventually create additional SQL user accounts in your database. You should create a special account for each user or application using the database, and not let them use the SA account. The SA account should only be used by you for management purposes.

Notice the brianhprince@mlwwmqca6u username in the ODBC connection string. Some tools will require this form of a username because of how they implement the TDS protocol, which is what underlies all connections to SQL Server.

One last thing you need to do to start using your database is to allow Azure-based applications to connect to the database. This is done on the Firewall Settings tab of the main administration screen. You'll need to check the Allow Microsoft Services Access to This Server check box, as shown in figure 13.6. This will create a default rule allowing Azure-based applications the right to connect to your server. The firewall will be discussed in more detail later in this chapter.

When you've completed these few steps, you'll have created a server, created a database on that server, and retrieved the connection strings you'll need to start using the server. In our example, we chose a 1 GB database; our next step is to discuss why we only have two options.

13.3 Size matters

There are two basic size limits you can choose from in SQL Azure: 1 GB and 10 GB. These are the maximum sizes that databases can be, and these sizes are the basis for how Microsoft will bill you for usage. The 1 GB database (the Web Edition) is charged

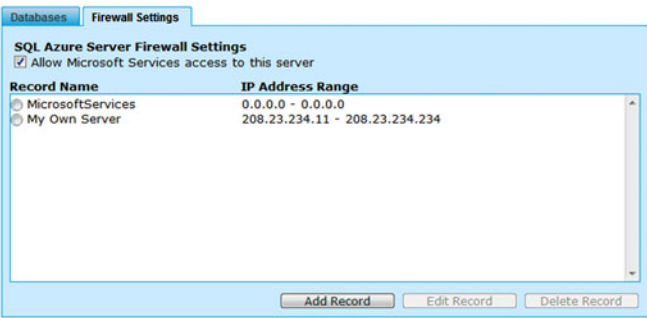


Figure 13.6 Enabling the SQL firewall to allow Azure application traffic to connect to your database. Only IP addresses listed on this screen are allowed to connect to your databases. The check box at the top allows for all Azure services under your account to connect.

at $9.99 per month, and the 10 GB (called the Business Edition) is charged at $99.99 per month. The charges are fixed and don't take into account the real size of the database, just the potential maximum size it can have. The size of your master database, logs, and indexes don't count toward the total size of the database when it comes to the size limits and charges.

You'll also be charged for any bandwidth used by your SQL Azure server that crosses a data center boundary. For example, if you have a mobile application accessing the SQL Azure database, bandwidth used by SQL Azure to those clients will be metered and billed for. If your Azure application is using the database, and they both reside in the same data center (aka geographic region), the bandwidth used won't be billed for because you aren't crossing a data center boundary.

The size of a database is fixed when it's created. If a 1 GB database grows too large, it will stop inserting new data. Databases won't automatically upgrade to a larger size. If you need a larger size, you'll need to use an `ALTER DATABASE` command and modify the `MAXSIZE` property.

The biggest reason for this limited size selection is the SLA that Microsoft provides to its customers. SQL Azure runs on a scalable and robust platform, with your data being replicated to different physical servers and disks in real time. The 1 GB and 10 GB sizes are the only sizes they feel they can currently support with the SLA defined as "Monthly Availability" of 99.9 percent. At this time, they couldn't guarantee a high SLA like this with a massive database. How quickly could you restore full function to a 32 TB database after a cataclysmic failure?

You should select the smallest size necessary to run your application. Currently there aren't any differences between the options, except for the size limit. In the future, the higher-end options will include extra features, such as autopartitioning, CLR support, and fanouts.

If your database is larger than 10 GB, you aren't out of luck. You do have a couple of options that are common in the SQL world when a database starts to overwhelm the hardware it's running on: partitioning and sharding. In this case, we'll use these strategies to work around the size limits that SQL Azure presents us.

13.3.1 Partitioning your data

The first option you have in dealing with a database that's too large for SQL Azure is to partition your data.

Imagine a normal application database schema. There are lots of tables, but all tables can be grouped into families. In our fictional DayOldSushiOnline.com website, we might have one set of tables that focus on customer data, a second set focusing on orders and order history, and a third set for product data and pricing, as shown in figure 13.7.

Data partitioning is the strategy of dividing up your database in a vertical manner, breaking your schema along family lines. In our example, this would create three databases with the names DOSO_customers, DOSO_orders, and DOSO_products (see

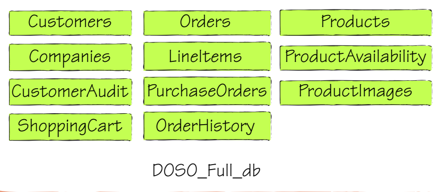


Figure 13.7 A sample application database schema for DayOldSushiOnline.com. The tables naturally fall into three families: customer data, orders and order history, and product data and pricing.

figure 13.8). This does tend to break relationships and queries across these families, requiring a significant amount of rework in your application code related to data handling, as well as in revising the stored procedures and queries in your database.

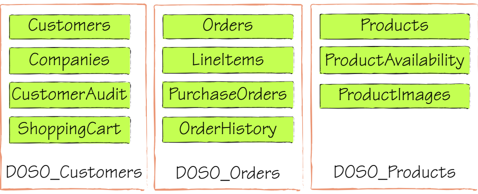


Figure 13.8 Our schema partitioned into families of tables

This strategy will create three smaller databases, each hopefully fitting into the 10 GB limit of SQL Azure. This is commonly done with on-premises databases because it allows the infrastructure team that supports the SQL Servers to tune each database, and its storage system, to the needs of that database. In our example, the products database is going to receive a lot of read traffic, with little write traffic. On the other hand, the orders database is likely to get a mixture of both, with more emphasis on write-related performance. This is a great approach for keeping the load on one part of the database from negatively impacting another part of the database.

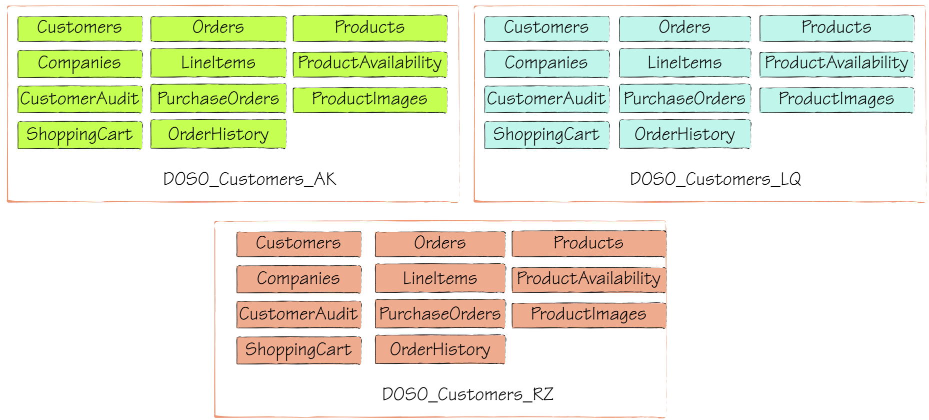
This strategy can lead to more complex backup and restore operations, because you'll need to guarantee transactional consistency across the different databases. That isn't much of a challenge when everything is in the same database.

13.3.2 Sharding your data for easier scale

Partitioning your data is fine, but it can become troublesome once your system reaches the true scale only the internet can provide. You might be able to break a family of tables down once more, but you're eventually backed into a corner and have to resort to expensive hardware tricks to continue to scale. This also continues to make your data-access code more and more complex.

Another option is to shard your data. Partitioning involves breaking your schema along vertical lines, whereas sharding is breaking your data along horizontal lines. You can see in figure 13.9 how this might look for the DayOldSushiOnline.com database. Notice we used the word *schema* with the partitioning strategy, and the word *data* with the sharding strategy.

When you shard a database, you create several databases with each new shard having the exact same schema as the original. Then you break the data into chunks, and place each chunk into its own shard. The strategy you use to define the boundaries of your chunks will vary based on the business and growth natures of your system. A simple strategy to start with, but one that will need rebalancing shortly, is to break the



chunks on the customer name field, as in figure 13.9. Each shard has its own copy of common or static data—in this case, the product data. This helps performance by providing local replicas of commonly needed data. This data could be partitioned out, but this would result in more complexity in your infrastructure.

In our example, the first shard will have all of the data that's related to any customer with a last name starting with A to K. The other two shards will cover the ranges L to Q, and R to Z. As new customer records are created, they're routed to the appropriate database. This approach can keep your data-layer code fairly straightforward, as all of the code is identical regardless of which database the customer is located in. The only code that needs to be added is some logic to look up which database connection string should be used for that customer. This can easily be isolated in your data-access layer, and the connection strings can be managed through the configuration system.

As we mentioned previously, your shards can become out of balance when you use this simple strategy. Some shards may become significantly larger than others, and this leads to a disproportionate use of the resources on each server. There are better strategies for how you might shard your data. One approach is to mix all of the customers up, and assign each new customer to the latest, smallest database. This, in effect, shards the database based on when the customer account was created, and not by customer name. This makes it easy to create new shards in the future, as the size of each database reaches its effective limit. When the last shard reaches capacity, you dynamically create a new shard, enroll it in the configuration system, and start adding new customers to it.

13.4 How SQL Azure works

Although we say that a SQL Azure database is just SQL Server database in the sky, that's not entirely accurate. Yes, SQL Server and Windows Server are involved, but not like you might think. When you connect to SQL Azure server, and your database, you aren't connecting to a physical SQL Server. You're connecting to a simulation of a server. We'd use the term *virtual*, but it has nothing to do with Hyper-V or application virtualization.

13.4.1 SQL Azure from a logical viewpoint

The endpoint that you connect to with your connection string is a service that's running in the cloud, and it mimics SQL Server, allowing for all of the TDS and other protocols and behavior you would expect to see when connecting to SQL Server.

This "virtual" server then uses intelligence to route your commands and requests to the backend infrastructure that's really running SQL Server. This intermediate virtual layer is how the routing works, and how replication and redundancy are provided, without exposing any of that complexity to the administrator or developer. It's this encapsulation that provides much of the benefit of the Azure platform as a whole, and SQL Azure is no different. The logical architecture of how applications and tools connect with SQL Azure is shown in figure 13.10.

Figure 13.10 The logical shape of SQL Azure, and how your code will see it. The SQL Azure server is really just a service that emulates a real SQL Server. Each of your databases are spread across a farm of SQL Servers.

As a rule of thumb, any command or operation that affects the physical infrastructure isn't allowed. The encapsulation layer removes the concern of the physical infrastructure. When creating a database, you can't set where the files will be, or what they will be called, because you don't know any of those details. The services layer manages these details behind the scenes.

13.4.2 SQL Azure from a physical viewpoint

The data files that represent your database are stored on the infrastructure as a series of replicas. The SQL Azure fabric controls how many replicas are needed, and creates them when there aren't enough available.

There's always one replica that's elected the leader. This is the replica that will receive all of the connections and execute the work. The SQL Azure fabric then makes sure any changes to the data are distributed to the other replicas using a custom replication fabric. If a replica fails for any reason, it's taken out of the pool, a new leader is elected, and a new replica is created on the spot. The physical architecture, relating the different parts of SQL Azure together, is shown in figure 13.11.

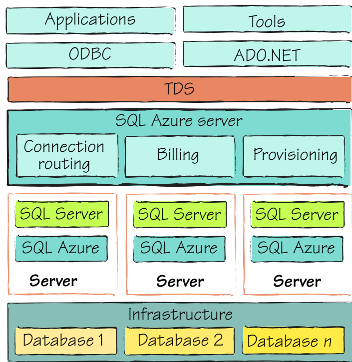


Figure 13.11 How SQL Azure works behind the scenes, encapsulating the physical infrastructure. Your data is replicated three times, and moved around on the SQL Azure fabric as needed to gain performance and reliability.

When a connection is made, the routing engine looks up where the current replica leader is located and routes the request to the correct server. Because all connections come through the router, the lead replica can change and the requests will be rerouted as needed. The fabric can also move a replica from one server to another for performance reasons, keeping the load smooth and even across the farm of servers that run SQL Azure.

What's really happening behind this encapsulation is quite exciting. The infrastructure layer contains the physical disks and networks needed to redundantly and reliably store the bits that are part of your database. This is similar to the common storage area network (SAN) that many database server infrastructures use. The redundancy of the disks and the tight coupling of the networks provide both performance and reliability for your data.

Sitting on top of this infrastructure layer is a series of servers. Each server runs a set of management services, SQL Server itself, and the SQL Azure fabric. The SQL Azure fabric is the component that communicates with the other servers in this layer to help them communicate with one another. The fabric provides the replication, load balancing, and failover features for the platform.

On top of the servers is a series of services that manages the connection routing (including the firewall features), billing, and provisioning. This services layer is the layer that you connect with and the layer that hides all of the magic.

Deep down under all of these covers, SQL Server really is running. Microsoft has added these layers to provide an automated and redundant platform that's easily managed and reliable.

13.5 Managing your database

SQL Azure is heavily automated, and it protects you from the concerns of the physical infrastructure. This frees you up to focus completely on managing your database. There are several aspects to managing a database, and many of them remain the same when the database is in the cloud.

If you ever have a script that needs to know what version of SQL it's running in, you can check the `Edition` and `Engine Edition` values of the server. When running in SQL Azure, you'll receive `SQL Azure` and `5` respectively. This is the query you would execute:

```
SELECT SERVERPROPERTY ('edition'), SERVERPROPERTY ('engineedition')
```

Usually, the first task you have after creating a database is moving your data up into the cloud.

13.5.1 Moving your data

Just because your database is in the cloud and you don't have to manage the disks doesn't mean you don't have to back up your data.

It's true that SQL Azure provides a robust and reliable platform for your data. It stores your data in multiple copies and load balances requests across those copies. In some cases, this might be enough to meet your disaster recovery needs.

But remember that, regardless of the vendor, there isn't any service in the cloud that protects you from stupid, there's no service that makes a bad application work correctly, and there's no platform that can fix bad decisions. What it can do is make them stupider, badder, and break faster, for less money. Don't be lulled into complacency by the scale and redundancy the cloud gives you. You still need to think of the risks to

your system, and to your data, and plan for them. The redundancy protects you against the common types of catastrophic loss, related to failing disks and other common issues, but not all loss is catastrophic in nature. For example, this will not protect you if a dinosaur eats Chicago or if you hire that guy Bob. Bob might execute a poorly written update SQL script that renames all of your customers to Terrance. This means you still need to back up your data, to protect your data from yourself and your code.

All of this great redundancy and scale won't protect against the disaster of the data center getting hit by a falling satellite, a radioactive monster, or an oil-eating bacteria. In these cases, having your bits on several servers in the same data center won't help you—you'll need geographic diversity. Only you, and the business you support, can determine what level of disaster recovery you need, and how much you're willing to afford.

Disaster isn't your only risk. What if an upgrade goes awry, and you accidentally delete a customer's data in your system? Or if an automated job takes a left turn and wipes out the order history for the past month? SQL Azure protects you against faults in the platform, but it doesn't protect you against faults in your own code or policies.

Even if you don't need to back up your data, you'll likely need to move data at some point. You'll run into this when you're migrating an existing on-premises database to the cloud, or when you want to move the data from SQL Azure to another location.

Right now there are only three ways to work with SQL Azure data. The first option is to use a developer's API to access the data and save it in some format that you can back up and restore from. There are a lot of options in this case; you could use ADO.NET, ODBC, or WCF Data Services, but this would be tedious and breaks our rule of not writing any plumbing code. We don't want to have to write our own backup tool.

The second option is to use the Bulk Copy Program (BCP), which is used quite often to move large amounts of data into and out of a SQL Server database. It's one of your best bets. When using BCP, it's common for the DBA to disable indexing until the import is completed, to speed up the transfer process. Once the data is loaded, the indexes are enabled and updated based on the new data.

To extract the data from your local SQL Server, you'll need to run BCP from a command prompt. The command is quite simple:

```
bcp databasename.schema.tablename out filename.dat -S servername
➥ -U username -P password -n -q
```

The `bcp` command is straightforward. When exporting data with BCP, you need to use the `out` keyword. Provide the fully qualified table name, the name of the file to write the data to, the server name, and your username and password. The parameters tell BCP to keep the SQL data types native (don't convert them), and to use quoted identities in the connection string. Depending on the size of your table, the copy might take a few minutes. Once it's done, you can open the data file and see what it looks like. When you're done poking around, you can use BCP in the other direction to blast the data into your SQL Azure database. You'll use a similar BCP command to insert the data into your SQL Azure database.

BCP operates at a table level, so you'll need to run it several times, once for each table. A common trick is to write a script that will export all of the tables, one at a time, so you don't have to do it manually. This also helps with automating the procedure, reducing the risk of mistakes, and making it easy to reproduce the process at any time.

Your last option for working with data you have stored, or want to store in SQL Azure, is SQL Server Integration Services (SSIS). SSIS is SQL Server's platform for extracting, transforming, and loading data. It's a common tool used to move data in and out of on-premises SQL servers, and it can connect to almost any data source you need it to. You can also migrate your existing SQL Server 2008 packages to SQL Azure (Azure can't run packages from SQL Server 2005). If you're going to connect with SQL Server Management Studio (SSMS), you'll need to use SSMS 2008 R2 or later; the earlier versions don't support SQL Azure. You can trick them into doing it, but that's material for a shady blog post, and not for a high-profile, self-respecting book.

Regardless of the tool you use to connect to your database, your connection will have to make it past the SQL Azure firewall.

13.5.2 Controlling access to your data with the firewall

Before an incoming connection will be routed to the current leader server for your replicas, the source IP address is checked against a list of allowed sources. If the source address isn't in this list, the connection is denied.

The list of approved sources is a true whitelist, meaning that the IP address must be on the list to be allowed in. There are no other rules supported, just the list of allowed addresses. The list is stored in the master database for your SQL Azure database server.

The most common way to adjust the firewall settings is the one shown earlier in this chapter, namely, by using the SQL Azure portal and making the changes through the admin pages. You can also add entries directly into the master database by using the stored procedures provided. The rules are stored in a table called firewall_rules (shown in figure 13.12), and you can query it like a normal table. Because the table is in the master database, the connection to the database performing the query must be under the system administrator account you set up.

To manage your firewall rules through code, you can create a connection to the master database with your administrator account and use the provided stored procedures: `sp_set_firewall_rule` will create a firewall rule, and `sp_delete_firewall_rule` will remove a rule. When creating a rule, you'll need to provide a unique name, and the two

SQLQuery1.sql - ...lwwmqca6u (150))*

```
select * from sys.firewall_rules
```

Results | Messages

	id	name	start_ip_addr...	end_ip_addr...	create_date	modify_date
1	3	BP Home	99.17.241.139	99.17.241.139	2009-10-17 17:03:38.490	2009-10-17 17:03:38.490
2	1	MicrosoftServices	0.0.0.0	0.0.0.0	2009-10-16 03:23:15.190	2009-10-16 03:23:15.190

Figure 13.12 You can see what firewall rules are being enforced with `select * from sys.firewall_rules`.

IP addresses that form the range of addresses allowed. These addresses can be the same if you want to grant access to a single IP address. When deleting a rule, you only need to provide the name of the rule to be removed.

Even applications running in Windows Azure can't connect to your server until you grant them permission to connect. Because application servers in Azure use virtual IP addresses, and the roles could shut down and restart anywhere, causing their addresses to change, there's a special rule you can use in the firewall. If you create a rule with 0.0.0.0 as the starting and ending addresses, any Azure application will be able to connect to your server. That connection must still provide a valid username and password.

A common problem is making a change to the firewall and then immediately trying to access the server from that IP address, which then fails. The cache that's used to speed up firewall-rule checks refreshes every 5 minutes. If you make a rule change, it's worth waiting up to 5 minutes before trying to see if the change works.

The second line of defense is the use of SQL credentials to verify that people are allowed to connect to your database.

13.5.3 Creating user accounts

Managing user accounts in SQL Azure works much the same way as in on-premises SQL Servers. A few small differences can be found, mostly with the names of specific roles, but in general it's the same.

Your first step in granting someone access to a database is to create a login for the server itself. You need to connect to the master database in your server using an account with the `loginmanager` role. This would normally be the administrator account you created when you created your server, but any other account with that role will work.

Once connected, you can list the existing logins by executing the `sys.sql_logins` view. To create a login, use the same command you're used to, as shown in figure 13.13:

```
CREATE LOGIN theloginname WITH password='strongpasswordhere';
```

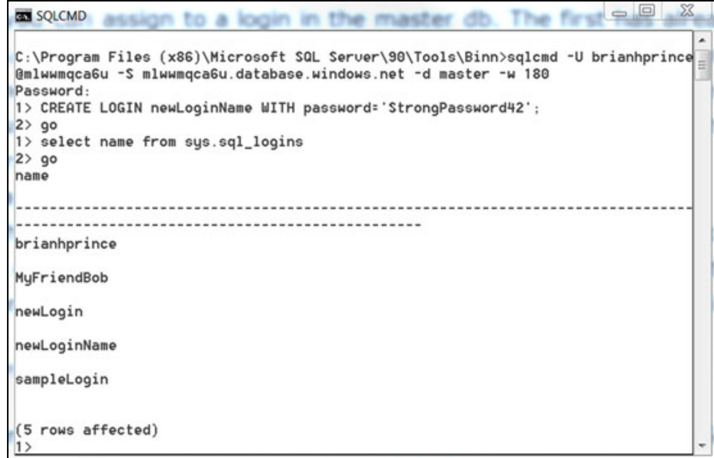


Figure 13.13 Creating a new SQL Azure login the same as with SQL Server on-premises, using the `CREATE LOGIN` command.

There are two roles that you can assign to a login in the master database. The first has already been mentioned, the `loginmanager` role. This role allows the user to manage user accounts and logins on the server. The second is the `dbmanager` role. Granting this role to a user account gives it all the possible permissions it could have, making it equivalent to the administrative account you first created.

Once you've created a server login, you can create a user account in a specific database based on that login. This is the same approach as in on-premises SQL Server installations. Although you can easily do this in SSMS, many DBAs like to do this with SQL commands. You would execute the following command while connected to the database you want to create the user account in:

```
CREATE USER newusername FROM LOGIN theloginname;
```

Once the user is created in the database, you can grant and revoke privileges as you would in a normal SQL database, either with the admin tools, or with the `GRANT` and `REVOKE` commands. SQL Azure only supports SQL users; Windows-based authentication isn't supported by SQL Azure.

Many people ask us when they should use SQL Azure and when they should use the Azure Table service. This is a complex question, and one we've dedicated the whole next section to.

13.6 Migrating an application to SQL Azure

One of the goals Microsoft had in delivering SQL Azure was to make it as easy as possible to migrate an existing application and its database to their Azure platform. Before SQL Azure was released, people were worried they would have to re-architect their systems in order for them to run in the cloud.

There are two basic ways to migrate your on-premises SQL Server data to the cloud. The first is to extract it, and copy it up. This is the same way you might move it from server A to server B in your own data center. The second way is to use an open source tool, SQL Azure Migration Wizard, that was created to help ease the migration process.

13.6.1 Migrating the traditional way

The easiest way to migrate a database to SQL Azure, and one that we have used ourselves, is to find a DBA who already knows how to do it and buy them lunch. Barring that, there are a few steps you can go through, following a traditional approach.

The first step is to make sure that you've created your SQL Azure server and know the administrative username and password. Once that's done, you can create the database that will house your schema and data. Your goal is to script out the existing database, including both schema and data, and then run that script against your cloud server database to recreate it all.

For this example, you'll use SQL Server Management Studio to script out your source database. Navigate to the database in SSMS, and right-click on it. Select Generate Scripts from the Tasks menu, and then walk through the wizard like normal. Make

sure that you select the option to script all objects in the database. Then, on the options screen, make your selections according to this list:

- Convert UDDTs to Base Types—True
- Script Extended Properties—False
- Script Logins—False
- Script USE DATABASE—False
- Script Data—True

Finish the wizard. When you're done, the entire script will be loaded in a window.

You now need to remove some parts of the script to make sure it'll work in the cloud:

- Delete any occurrence of `SET ANSI_NULLS ON`.
- Delete any occurrence of `WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]` in a `CREATE TABLE` command.
- Delete any occurrence of `ON [PRIMARY]` in a `CREATE TABLE` command.

Once you've made these changes to the script, you may connect to the destination database and execute the script. You can use SQLCMD for this, or SSMS.

This is a lot of work, and it can be tedious making sure you have modified the script to avoid anything SQL Azure might not support. Thankfully there is an open source tool that can help.

13.6.2 Migrating with the wizard

The previous section might have scared you away from ever trying to migrate data to SQL Azure. It wasn't meant to, but there are a lot of little details to be considered. Fortunately, there's an open source project hosted on CodePlex called SQL Azure Migration Wizard that can help you out with this process. The project URL is http://sqlazuremw.codeplex.com.

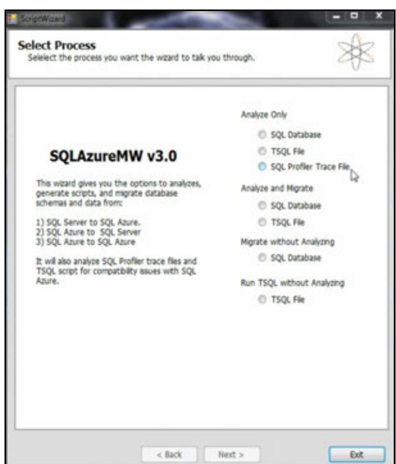


Figure 13.14 The SQL Azure Migration Wizard is a great tool when migrating or planning to migrate data from SQL Server to SQL Azure. It can help in modifying your database so it's compliant, and it also helps move the data.

The wizard can be used in several different scenarios. It can handle moving or analyzing a database from a SQL Server to SQL Azure, from SQL Azure to a SQL Server, or from SQL Azure to SQL Azure. When you run the tool, you can analyze the database for compatibility, generate a compatible script, and migrate the data. You can see the options available to you in figure 13.14.

When you start the tool, you'll need to point it at the source database. You'll be given the options to select the objects in the database you want to analyze and migrate.

The wizard is driven by a powerful rules engine with over 1,000 compatibility rules already programmed in. These rules are in an XML file, and it's easy to modify them to meet your needs. The rules detect a condition that isn't supported by SQL Azure, and specify how to fix it. This might be as simple as converting a data type to a type that's supported, or something more intense.

Once the analysis is done, you'll receive a report on the objects that need to be changed to become compatible with SQL Azure. You can stop here and use the report to make decisions, or you can keep going to migrate your data.

You'll need to provide a destination database connection string and credentials if you want the wizard's help in migrating your data. It'll connect to SQL Azure and execute the new script. It'll handle creating the tables and other objects, as well as migrating the data.

Migrating the data tends to be the most difficult part. If you're doing this by hand, even after you sort out the changes you need to make to your script, you still need to worry about timeouts in the connection and batching your data during the upload. You don't want all of your data thrown out because the connection was reset. The migration wizard handles all of this for you, so you don't have to worry about anything.

Because SQL Azure is running in the cloud, there are some limitations. Some of these are related to schemas, and some are related to functional capabilities.

13.7 Limitations of SQL Azure

Although SQL Azure is based on SQL Server, there are some differences and limitations that you'll need to be aware of. We've mentioned some of these in various places in this chapter, but we'll try to cover them all in this section.

The most common reason for any limitation is the services layer that sits on top of the real SQL Servers and simulates SQL Server to the consumer. This abstraction away from the physical implementation, or the routing engine itself, is usually the cause. For example, you can't use the `USE` command in any of your scripts. To get around this limitation, you'll need to make a separate connection for each different database you want to connect with. You should assume that each of your databases are on different servers.

Why you can't use `USE`

You can't use the `USE` command in SQL Azure because the routing layer is stateful, because the underlying TDS protocol is session-based. When you connect to a server, a session is created, which then executes your commands. When you connect in SQL Azure you still have this session, and the fabric routes your commands to the physical SQL Server that's hosting the lead replica for your database. If you call the `USE` command to connect to a different database, that database may not be on the same physical server as the database you're switching from. To avoid this problem, the `USE` command isn't allowed.

Any T-SQL command that refers to the physical infrastructure is also not supported. For example, some of the `CREATE DATABASE` options that can configure which `filegroup` will be used aren't supported, because as a SQL Azure user, you don't know where the files will be stored, or even how they will be named. Some commands are outright not supported, like `BACKUP`.

You can only connect to SQL Azure over port 1433. You can't reconfigure the servers to receive connections over any other port or port range.

You can use transactions with SQL Azure, but you can't use distributed transactions, which are transactions that enroll several different systems into one transaction update. SQL Azure doesn't support the network ports that are required to allow this to happen. Be aware that if you're using a .NET 2.0 TransactionScope, a normal transaction may be elevated to a distributed transaction in some cases. This will cause an error, and you won't know where it's coming from.

Each table in your database schema must have a clustered index. Heap tables (a fancy DBA term for a table without an index) aren't supported. If you import a table without a clustered index, you won't be able to insert records into that table until one has been created.

All commands and queries must execute within 5 to 30 minutes. Currently the system-wide timeout is 30 minutes. Any request taking longer than that will be cancelled, and an error code will be returned. This limit might change in the future, as Microsoft tunes the system to their customers' needs.

There are some limitations that are very niche in nature, and more commands are supported with each new release. Please read the appropriate MSDN documentation to get the most recent list of SQL Azure limitations.

13.8 Common SQL Azure scenarios

People are using SQL Azure in their applications in two general scenarios: *near data* and *far data*. These terms refer to how far away the code that's calling into SQL Server is from the data. If it's creating the connection over what might be a local network (or even closer with named pipes or shared memory), that's a near-data scenario. If the code opening the connection is anywhere else, that's a far-data scenario.

13.8.1 Far-data scenarios

The most common far-data scenario is when you're running your application, perhaps a web application, in an on-premises data center, but you're hosting the data in SQL Azure. You can see this relationship in figure 13.15. This is a good choice if you're slowly migrating to the cloud, or if you want to leverage the amazing high availability and scale SQL Azure has to offer without spending $250,000 yourself.

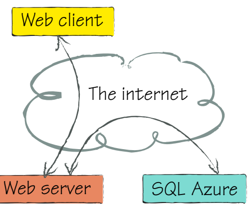


Figure 13.15 A web server using SQL Azure in a far-data scenario. The data is far away from the code that's using it. In this case, the web server is on-premises, and the data is in the cloud with SQL Server.

In a far-data scenario, the client doesn't have to be a web browser over the internet. It might be a desktop WPF application in the same building as the web server, or any other number of scenarios. The one real drawback to far data is the processing time and latency of not being right next to the data. In data-intensive applications this would be a critical flaw, whereas in other contexts it's no big deal.

Far data works well when the data in the far server doesn't need to be accessed in real time. Perhaps you're offloading your data to the cloud as long-term storage, and the real processing happens onsite. Or perhaps you're trying to place the data where it can easily be accessed by many different types of clients, including mobile public devices, web clients, desktop clients, and the like.

13.8.2 Near-data scenarios

A near-data scenario would be doing calculations on the SQL Server directly, or executing a report on the server directly. The code using the data runs close to the data. This is why the SQL team added the ability to run managed code (with CLR support) into the on-premises version of SQL Server. This feature isn't yet available in SQL Azure. Figure 13.16 shows what a near-data scenario looks like.

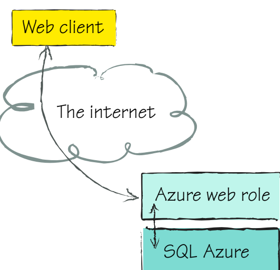


Figure 13.16 Hosting a data service in an Azure web role helps your application be in a near-data scenario. This improves the performance of the application when it comes to working with the data.

One way to convert a far-data application to a near-data one is to move the part of the application accessing the code as close to the data server as possible. With SQL Azure, this means creating a services tier and running that in a role in Azure. Your clients can still be web browsers, mobile devices, and PCs, but they will call into this data service to get the data. This data service will then call into SQL Server. This encapsulates the use of SQL Azure, and helps you provide an extra layer of logic and security in the mix.

13.8.3 SQL Azure versus Azure Tables

SQL Azure and the Azure Table service have some significant differences, which we've tried to cover in this chapter and the chapters on Azure Tables (chapters 11 and 12). These differences help make it a little easier to pick between SQL Azure and Azure Tables, and the deciding factor usually comes down to whether you already have a database to migrate or not.

If you do have a local database, and you want to keep using it, use SQL Azure. If moving it to the cloud would require you to refactor some of the schema to support partitioning or sharding, you might want to consider some options.

If size is the issue, that would be the first sign that you might want to consider Azure Tables. Just make sure the support Tables has for transactions and queries meets your needs. The size limit surely will be sufficient, at 100 TB.

If you're staying with SQL (versus migrating to Azure Tables) and are going to upgrade your database schema to be able to shard or partition, take a moment to

think about also upgrading it to support multitenant scenarios. If you have several copies of your database, one for each customer that uses the system, now would be a good time to add the support needed to run those different customers on one database, but still in an isolated manner.

If you're building a new system that doesn't need sophisticated transactions, or a complex authorization model, then using Azure Tables is probably best. People tend to fall into two groups when they think of Tables. They're either from "ye olde country" and think of Tables as a simple data-storage facility that'll only be used for large lookup tables and flat data, or they're able to see the amazing power that a flexible schema model and distributed scale can give them. Looking at Tables without the old blinders on is challenging. We've been beaten over the head with relational databases for decades, and it's hard to consider something that deviates from that expected model. The Windows Azure platform does a good job of providing a platform that we're familiar and comfortable with, while at the same time giving us access to the new paradigms that make the cloud so compelling and powerful.

The final consideration is cost. You can store a lot of data in Azure Tables for a lot less money than you can in SQL Azure. SQL Azure gives you a lot more features to use (joins, relationships, and so on), but it does cost more.

13.9 Summary

SQL Azure is a powerful data platform that's familiar to us. It just happens to be running in the cloud. This makes it easier to move an application to the cloud, or to build something new, using paradigms and tools that we know and love.

We looked at how to create and manage a database in the cloud. The processes and approaches are eerily similar to what you would do when working with a local database, which makes it easy to adopt SQL Azure for your application. There are some limitations because of the "virtual" nature of the SQL Server, and because the database is running in a shared environment.

The sophisticated data engine that is SQL Azure isn't the last stop for SQL in the cloud—it's just the beginning. The tools will be upgraded for full support, and the rest of the SQL family will move into the cloud over time.

Although the firewall and other security features make it easier to trust putting your data in the cloud, you still need to think critically about whether it's the right place for your data, or whether a blended approach is best for your scenario.

Now that you've moved your data to the cloud, it's time to learn more about how best to query and use data in a cloud application. Chapter 14 will look at this from several different angles.

14 Working with different types of data

This chapter covers

- Working with static data
- Working with dynamic data
- Working with infrequently changing data

In previous chapters, we've shown you what you can do with the Table service, SQL Azure, and caching. In this chapter, we'll look at how you can choose when to use these three technologies. Rather than focusing specifically on the technologies, we'll look at the types of data you can store with each of the technologies and at how each technology will help you store different types of data.

We'll look at three types of data in particular: static data, dynamic data, and infrequently changing data. We'll also focus on where you can store the data and how you can efficiently retrieve it.

We'll start with static data.

14.1 Static reference data

Every application typically has some sort of frequently accessed static reference data. This data is usually very small and typically used for data normalization purposes. Let's return to the Hawaiian Shirt Shop website and look at an example.

For each shirt displayed in the Hawaiian Shirt Shop web page, you might wish the customer to be able to specify the following criteria about the shirt they want to buy:

- Shirt personage type (men, ladies, boys, girls)
- Shirt size type (small, medium, large, extra large)
- Shirt material (cotton, silk, wool)

As you can see, the data listed is fairly static, and it's applied across all shirts. All shirts will have a size and a material (admittedly not wool). This data can be considered static, because it's unlikely that it would ever be changed once it's defined. (Hawaiian shirts are unlikely to suddenly start being made in platinum.)

Figure 14.1 shows a page of the website where that data can be selected. The customer can browse shirts that are designed for men or ladies, and for a particular shirt they can choose the size or material (style).

As you can see from figure 14.1, the web page represents the static material (style) and size data with drop-down lists, whereas the personage type is represented as a hyperlink that will perform a search for different shirt types. For now we'll focus on the two drop-down lists (material and size).

The first question you're probably asking is, "Where and how do we represent this data?" Let's take a look at how this could be done using each of these technologies:

- SQL Azure
- Table service
- Cache

Figure 14.1 Product detail page of the Hawaiian Shirt Shop website

We'll start with SQL Azure, as this is probably the most familiar way of representing data.

14.1.1 Representing simple static data in SQL Azure

As you learned in chapter 13, SQL Azure is a relational database, so you would use a typical relational model to store the data. Figure 14.2 shows a database diagram for the Hawaiian Shirt Shop website in SQL Azure.

In figure 14.2 you can see that the data for each of the drop-down lists (size types and materials) is currently stored in their own tables. As of yet, we haven't defined any

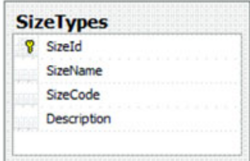


Figure 14.2 A database with Shirts, SizeTypes, and Materials tables in SQL Azure with no relationships defined

relationships between the static tables and the Shirts table (which is the most central table in the relationship).

Now let's take a look at how you would retrieve data from this database and populate the drop-down lists on the web page.

POPULATING DROP-DOWN LISTS

To populate the materials or size types drop-down list directly from a database, you can make a standard ADO.NET call to the database (either using ADO.NET directly or your favorite data-access layer technology, such as Linq2SQL, ADO.NET Entity Framework, or NHibernate).

The following code shows how you could bind the size drop-down list using ADO.NET directly:

```
DataSet ds = new DataSet();
using (SqlConnection conn =
   new SqlConnection(mySqlAzureDBConnectionString))
{
   conn.Open();

   using (SqlDataAdapter da =
      new SqlDataAdapter("SELECT SizeId, SizeName FROM SizeTypes",
          conn))
   {
      da.Fill(ds);
   }
}

sizeDropDown.DataSource = ds.Tables[0];
sizeDropDown.DataTextField= "SizeName";
sizeDropDown.DataValueField = "SizeId";
sizeDropDown.DataBind();
```

The preceding code shows that you can bind your drop-down lists to a table in SQL Azure just as easily as you can with a regular SQL Server database.

> **WARNING** The preceding code obviously isn't up to production standards. You shouldn't mix data-access code with presentation-layer code, but it does illustrate the point.

Cost issues with SQL Azure

SQL Azure uses a fixed-price model, and if your database is tied up servicing static data calls (which always return the same set of data), you may hit the limits of your database quickly and unnecessarily, requiring you to scale out to meet the demand. In this situation, caching the data is probably the most cost-effective approach.

As stated earlier, SQL Azure isn't the only method of storing static data. Let's look at how you could store this data using the Table service.

14.1.2 Representing simple static data in the Table service

Just as easily as each type of static data could be represented in a SQL Azure table, the data could also be represented as entities in a Table service table. The following C# class could represent the `SizeType` entity in the Table service.

```
public class SizeType : TableServiceEntity
{
   public string SizeCode { get; set; }
   public string Description { get; set; }
}
```

In this class, the `PartitionKey` for the `SizeType` entity isn't relevant due to the size of the table, so you could make all entities in the table have the same partition key. You could use the `SizeCode` property to represent the `RowKey`.

> **NOTE** Because the Table service isn't a relational database and you don't need clustered indexes here, you have no need for the `SizeTypeId` surrogate key that's present in the SQL Azure implementation.

The following code represents the data service context class for the SizeTypes table.

```
public class SizeTypeContext : TableServiceContext
{
  private static CloudStorageAccount storageAccount =
  CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

  public SizeTypeContext()
     : base(storageAccount.TableEndpoint.ToString(),
            storageAccount.Credentials)
   {

   }

  public DataServiceQuery<SizeType> SizeTypeTable
  {
     get
     {
        return CreateQuery<SizeType>("SizeTypeTable");
     }
  }
}
```

NOTE For a detailed explanation of the context class, refer to chapter 11.

To store a size in the SizeTypeTable table in the Table service, you could use the following code:

```
var sizeTypeContext = new SizeTypeContext();

var newSizeType = new SizeType
{
          PartitionKey = "SizeTypes",
          RowKey = "Small",
          SizeCode = "S",
          Description = "A shirt for smallish people"
};

sizeTypeContext.AddObject("SizeTypeTable", newSizeType);
sizeTypeContext.SaveChanges();
```

This code will store the "Small" size entity in the SizeTypeTable table. The method used to store the data in the table is explained in more detail in chapter 11, which discusses the Table service.

TIP In this particular example, the SizeTypeTable table will never grow beyond a few rows, so it's not worth splitting the table across multiple servers. Because all the size data will always be returned together, you can store the data in a single partition called SizeTypes.

Once you have a fully populated SizeTypeTable table, you can bind it to the drop-down list using the following code:

```
var sizeTypeContext = new SizeTypeContext ();
sizeDropDown.DataSource = sizeTypeContext.SizeTypeTable;
sizeDropDown.DataTextField = "RowKey";
sizeDropDown.DataValueField = "SizeCode";
sizeDropDown.DataBind();
```

In this example, the drop-down list is populated directly from the Table service.

Cost issues with the Table service

Because you're charged for each request that you make to the Table service, reducing the number of requests will reduce your monthly bill. In a website where you receive 1,000 page views of the Product Details page, this would translate to 4,000 Table service requests. Based on these figures, your hosting bill could get very costly very quickly if you follow this model. Caching the data is probably the most efficient thing to do in this situation.

We've already talked a little about the performance disadvantages of using SQL Azure or the Table service for accessing static data. Let's look more closely at why you should avoid chatty applications in Windows Azure.

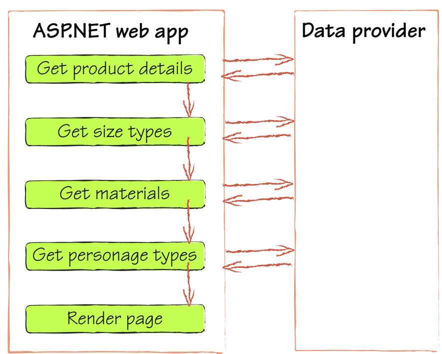
14.1.3 Performance disadvantages of a chatty interface

In the previous sections, we've discussed how you could store static data, such as shirt sizes and materials, in SQL Azure or the Table service. We'll now look at the call sequences you'd need to make to render the web page (shown previously in figure 14.1) and discuss the pros and cons of each approach.

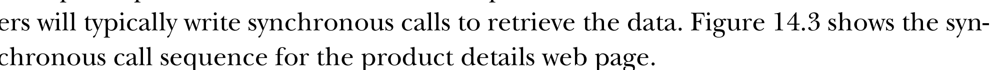
SYNCHRONOUS CALLS

To retrieve all the data required to display the product details page shown in figure 14.1, you'd need to make at least four calls to the storage provider:

- Retrieve the product details
- Retrieve the list of size types
- Retrieve the list of materials
- Retrieve the list of personage types (men, ladies, and so on)



When developing an ASP.NET web page, you should consider making asynchronous calls to improve performance, but most develop-



ers will typically write synchronous calls to retrieve the data. Figure 14.3 shows the synchronous call sequence for the product details web page.

As you can see, the synchronous nature of the page means you have to wait until you receive data back from the data provider (SQL Azure or the Table service) before you can process the next section of the page. Due to the blocking nature of synchronous calls and the latency involved in cross-server communication, the rendering of this page will be much slower than it needs to be.

ASYNCHRONOUS CALLS

Because size types, materials, and personage types are sets of data that are both independent of each other and independent of the returned product details, you could use asynchronous calls instead. Retrieving the data from the Table service asynchronously means you don't have to wait for one set of data to be returned before retrieving the next set.

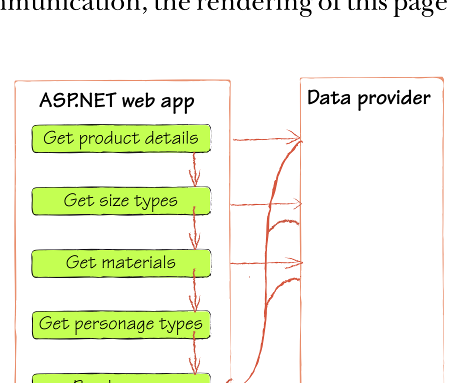


Figure 14.4 shows the same call sequence as in figure 14.3, but this time using asynchronous calls.

As you can see in figure 14.4, you no longer have to wait for data to be returned before you process the next statement. Once all the data is returned, you can render the page.

TIP Here we've used static data calls as our example. You can, however, use asynchronous calls whenever you don't have any relationship between sets of data rendered on a page.

Now that you understand how you can store and retrieve static data, let's take a look at how you can improve performance (and reduce the hosting bill) by using cached data instead.

14.1.4 Caching static data

Regardless of your chosen storage platform (SQL Azure or the Table service), you should consider caching static data rather than continually retrieving the same data from a table. This will bring large performance and cost benefits.

Because static reference data hardly ever changes and is usually a pretty small set of data, the in-process memory cache (such as the default ASP.NET cache) is a suitable caching choice. You can always use a cache dependency to keep the cache and the underlying backing store synchronized. For more details on how to set up an in-process cache, refer to chapter 6.

Let's now take a look at how you can use the cache in the Hawaiian Shirt Shop website.

POPULATING THE CACHE

For frequently accessed static data, you should probably populate the web role cache when the application starts up. The following code, placed in the Global.asax file, will do this:

```
protected void Application_Start(object sender, EventArgs e)
{
   var sizeTypeContext = new SizeTypeContext();
   HttpRuntime.Cache["sizeTypes"] = sizeTypeContext.SizeTypeTable;
}
```

In this example we've populated the cache with data from the Table service, but the technique is the same for using SQL Azure. We're using a cache because we're working with static data, and we don't want to hit the data source too often. You might want to consider populating the caching when your role instance starts, instead of when the ASP.NET application starts.

POPULATING THE DROP-DOWN LISTS

Now that you have your data in the cache, let's take a look at how you can populate the drop-down lists with that data:

```
sizeDropDown.DataSource = (IEnumerable<SizeType>)Cache["sizeTypes" ];
sizeDropDown.DataTextField = "RowKey";
sizeDropDown.DataValueField = "SizeCode";
sizeDropDown.DataBind();
```

As you can see from this code, you no longer need to return to the data store to populate the drop-down list—you can use the cache directly.

Because the cache will be scavenged when memory is scarce, you can give static data higher priority than other cache items by using the cache priority mechanism (meaning that other items will be scavenged first):

```
var sizeTypeContext = new SizeTypeContext();
HttpContext.Current.Cache.Insert("sizeTypes",
➥  sizeTypeContext.SizeTypeTable, null, new DateTime(2019, 1, 1),
➥  Cache.NoSlidingExpiration, CacheItemPriority.High, null);
```

In the preceding code, the `SizeTypes` list will be stored in the cache with a `High` priority, and it will have an absolute expiration date of January 1, 2019. If the web role is restarted, the cache will be flushed, but if the process remains running, the data should remain in memory until that date.

If the static data might change in the future, you can set a cache dependency to keep the cache synchronized or manually restart the role when updating the data.

PROTECTING YOUR CODE FROM AN EMPTY CACHE

Because cache data is volatile, you might wish to prevent the cached data from being flushed by checking that the data is cached prior to populating the drop-down list:

```
private IEnumerable<SizeType> GetSizeTypes()
{
   if (Cache["sizeTypes"] == null)
   {
      var sizeTypeContext = new SizeTypeContext();
      Cache["sizeTypes"] = sizeTypeContext.SizeTypeTable;
   }

   return (IEnumberable<SizeType>)Cache["sizeTypes"];
}

sizeDropDown.DataSource = GetSizeTypes();
sizeDropDown.DataTextField = "RowKey";
sizeDropDown.DataValueField = "SizeCode";
sizeDropDown.DataBind();
```

In this code, before the drop-down list is populated, a check is run to make sure that the data is already stored in the cache. If the data isn't held in cache, it repopulates that cache item.

By effectively caching static data, you can both reduce your hosting bill and improve the performance of your application. By using an in-memory cache for static data on the product details page, you now have one data storage call per application start up rather than four.

Using in-memory cache for static data also means that your presentation layer no longer needs to consider where the underlying data is stored.

Please be aware that the examples in this section aren't production-level code and have been simplified to illustrate the concepts. If you're implementing such a solution, you should also take the following guidelines into consideration:

- Abstract your caching code into a separate caching layer
- Don't use magic strings (such as `Cache["sizeTypes"]`)—use constants instead
- Use cache dependencies

- Prioritize your cache properly
- Check that your cache is populated prior to returning data
- Handle exceptions effectively

14.2 Storing static reference data with dynamic data

In the previous section, we looked at how you could represent static data for the purposes of data retrieval. This is only one side of the picture, because typically you'll want to associate the static data with some dynamic data.

For example, people viewing the product details web page shown in figure 14.1 will hopefully purchase the shirt displayed. When they do, how should you store that data so that you can easily retrieve it?

Depending on the implementation of the web page, you can either allow the user to purchase the item directly or add the item to a shopping cart. In either case, the method of storing the data will be the same, so let's look at storing items in a shopping cart.

14.2.1 Representing the shopping cart in SQL Azure

You're probably familiar with relational databases, so we'll first look at how you can store shopping cart data in SQL Azure. (We'll look at using the Table service in section 14.2.3).

THE SHOPPING CART DATA MODEL

The shopping cart can be persisted across sessions, and if the user is a registered logged-in user, you can associate this account with their user ID. Figure 14.5 represents a typical data model for a shopping cart.

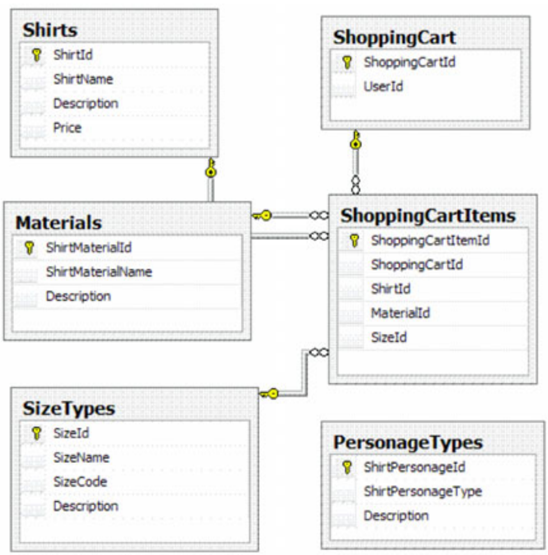


Figure 14.5 Data model for a shopping cart

In figure 14.5, the shopping cart is represented as two tables (ShoppingCart and ShoppingCartItems). The ShoppingCart table represents the shopping cart for each user, and each item in the shopping cart is stored in the ShoppingCartItems table. For each item in the cart, the ShirtId, MaterialId, and SizeId are stored, and the appropriate foreign-key relationships between tables are established.

If the website user is a registered user, the UserId would be stored in the ShoppingCart table; if the user is unregistered, you could use the session ID.

NOTE Although this data model represents a shopping cart, you could model an orders table using the same data structure.

RETRIEVING THE DATA

Because SQL Azure is a relational database, you could query across multiple tables by using `JOIN` clauses in your SQL statements, if required. For example, to return all items in the shopping cart for `UserId 12345`, including the shirt name, size, and material, you could issue the following SQL query:

```
SELECT sci.ShoppingCartItemId,
       s.ShirtName,
       s.Description,
       s.Price,
       sz.SizeId,
       sz.SizeName,
       m.MaterialId,
       m.MaterialName
FROM ShoppingCart sc
JOIN ShoppingCartItems sci ON sci.ShoppingCartId = sc.ShoppingCartId
JOIN Shirts s ON s.ShirtId = sci.ShirtId
JOIN SizeTypes sz ON sz.SizeId = sci.SizeId
JOIN Materials m ON m.MaterialId = sci.MaterialId
WHERE sc.UserId = '12345'
```

Because the shopping cart table will never hold a large amount of data, and the static data is held in an in-memory cache, there's no need to return the static data as part of your SQL query. You can, therefore, use the following query instead:

```
SELECT sci.ShoppingCartItemId,
       s.ShirtName,
       s.Description,
       s.Price,
       sci.SizeId,
       sci.MaterialId,
FROM ShoppingCart sc
JOIN ShoppingCartItems sci
   ON sci.ShoppingCartId = sc.ShoppingCartId
JOIN Shirts s ON s.ShirtId = sci.ShirtId
WHERE sc.UserId = '12345'
```

Not returning the static data (SizeTypes and Materials) both improves the performance and reduces the complexity of the query by reducing the number of joins.

Once the data is returned from the database, you can combine it with the in-memory cached data using LINQ to Objects.

> **NOTE** This strategy is perfectly acceptable when dealing with static reference data because the cached data tends to be very small. This would not be an acceptable strategy when dealing with millions of rows of data.

14.2.2 Partitioning the SQL Azure shopping cart

The data model in figure 14.5 looks like it's only usable for single-server databases, but this isn't strictly the case. That model will easily scale out, as we'll explain in a second.

One of the issues with using SQL Azure is that currently there's no built-in method of partitioning data across multiple servers. To avoid bottlenecking your application

on a single database, you need to partition (or shard) your data across multiple servers in your application layer. In chapter 13, we spent a little time talking about sharding, but here we'll look at how to apply it in relation to the Hawaiian Shirt Shop.

SPLITTING THE DATA MODEL ACROSS MULTIPLE SERVERS

As we said, the data model in figure 14.5 will work across multiple servers—you just have to be a little smart about how you do this. Because the shopping cart tables have no dependencies on any other part of the application, you can separate those tables into their own separate database.

In the application layer, you could therefore separate the shopping cart's data-access layer methods and connection strings into their own layer. By keeping the functionality logically separated, you can split your application across multiple databases if required. In the Hawaiian Shirt Shop example, you could easily maintain separate databases for the shopping cart, orders, customers, products, and static data, if required.

All those databases—isn't that expensive?

The good news about this type of design is that you can still start with a single SQL Azure database when your application has a small number of users. As the traffic increases, you can then split the shopping cart tables into their own separate database when needed. To split out your databases, you could simply create a new SQL Azure database, migrate the shopping cart data across, and then change the connection string for the shopping cart's data layer.

PARTITIONING DATA FURTHER

In your web application, you might get to the point where the shopping cart database is bottlenecking and you need to partition the data further. This isn't as difficult as it sounds.

Because the shopping cart data is only used by a single user and you never query across multiple users, you can easily partition out the data by user. For example, you might have 100,000 registered users who maintain a shopping cart. If you currently have a single database and wish to split it into two or more databases, you could use a partitioning function in your application:

```
If (userId.ToString()[0] < 'N')
{
   // Use Connection String for Database 1
}
else
{
  // Use Connection String for Database 2
}
```

In the preceding example, if the first character of the `userId` begins with `N` or later, they would use the second shopping cart database; otherwise, they'd use the first database. You could break this down into further partitions if required.

As stated earlier, this type of partitioning can only be done in the application layer at present. It's planned that in future releases of SQL Azure there'll be some partitioning built-in on the server side.

Maintaining referential integrity

One of the issues with splitting data across multiple servers is keeping referential integrity.

In the shopping cart example, the data that the shopping cart is referencing is either static (size types, materials, or personage types) or infrequently changing (and administratively controlled), so referential integrity isn't such a big deal. You're ultimately controlling the data, so you can break foreign-key relationships and store the various tables on separate servers.

If you do need to maintain referential integrity, you could keep a copy of the static data on each instance of the database. Although there is data duplication, there's such a small amount of data that you can easily fix the problem by synchronizing the databases. In this case, you'd generally keep one master database allowing one-way synchronization.

In the shopping cart example, because there's no need to query across multiple tables (beyond the cache), you can safely break referential integrity.

Querying across partitions

Unfortunately there is no way of efficiently querying across multiple databases in the current implementation of SQL Azure, which is why you can only partition functionally independent data.

If you need to query across partitions when reporting (such as when reporting on all customer orders in the past week), you can always export data from your SQL Azure real-time database to a large reporting database (outside of Windows Azure) where you can make use of a full-blown version of SQL Server with BI capabilities.

In this example, we've looked at how you can use SQL Azure with Windows Azure when storing dynamic data with static data (and partitioning where necessary). We've purposely not looked at how this works with large sets of infrequently changing data because we'll look at this in section 14.3 of this chapter.

Let's transition over to how we would implement the same concept, but using the Azure Table service instead.

14.2.3 Representing the shopping cart's static data in the Table service

In the previous section, we looked at how you could store the shopping cart in SQL Azure and how you could scale it out horizontally if necessary. Although it's possible to scale out a SQL Azure database, it still requires you to add some manual partitioning

code at the application layer. We'll now look at how you could represent the shopping cart table in the Table service, and at how you can use the built-in partitioning model to scale out your tables.

Due to the architecture of the Table service, there's no facility to perform a server-side join between a Shirts table and the ShoppingCart table. This effectively leaves you two options when you have data that resides on two different tables:

- Duplicate the data
- Join the data on the client side

For now, we'll ignore the duplicate data option (we'll cover that in the next section), and we'll focus on client-side data joining. But before we look at joining the data on the client side, let's take a peek at how you could represent the ShoppingCart table in the Table service.

SHOPPINGCART ENTITY

In the SQL Azure implementation of our shopping cart data model, there were two tables (ShoppingCart and ShoppingCartItems). Because the Table service isn't a relational database and doesn't support joining between tables, you can represent the shopping cart as a single table (ShoppingCart). Within the ShoppingCart table you can store the entity as follows:

```
public class ShoppingCartItem : TableServiceEntity
{
   public string Shirt {get;set;}
   public string Material { get; set; }
   public string Size { get; set; }
}
```

Consider the definition of the `ShoppingCartItem` entity. Because both the material and size data are cached in memory, you can simply store a reference to the data (the row key for the material and size) and then perform a client-side join between the shopping cart entity and the cached versions of the Material and SizeType entities. Because the cached data is a small set of static reference data, and it's being joined to a small set of shopping cart data, this technique is appropriate.

> **Partitioning with the Table service**
>
> Unlike SQL Azure, partitioning the Table service implementation of the shopping cart is pretty simple. All you need to do is set a reasonable value for the partition key for the shopping cart table, and the Windows Azure Table service will take care of the rest.
>
> In this case, you'll only be retrieving the shopping cart items for one user at a time, so it would make sense to partition the data by `UserId`. By setting the PartitionKey as the UserId, the data can be physically partitioned to as many servers as necessary, and the data for a single user's shopping cart will always physically reside together.
>
> For more details on how Table service partitioning works, refer back to chapter 11.

To join these two sets of data, you can define a new entity that will represent a strong version of your shopping cart item, as follows:

```
public class StrongShoppingCartItem
{
   public Shirt SelectedShirt { get; set; }
   public SizeType Size { get; set; }
   public Material Material { get; set; }
}
```

As you can see, this code represents the shopping cart item with a reference to each entity rather than the using an ID reference.

Now that you have the stronger version of the entity, you need to populate it, like this:

```
var materials = (IEnumerable<Material>)Cache["materials"];
var sizeTypes = (IEnumerable<SizeType>)Cache["sizeTypes"];

var shoppingCartItemContext = new ShoppingCartItemContext ();
var shoppingCartItems =
   shoppingCartItemContext.ShoppingCartItem.ToList();

var q = from shoppingCartItem in shoppingCartItems
        join sizeType in sizeTypes
           on shoppingCartItem.Size.RowKey equals sizeType.RowKey
        join material in materials
           on shoppingCartItem.Material.RowKey equals material.RowKey
        select new StrongShoppingCartItem
        {
           SelectedShirt = new Shirt(),
           Material = material,
           Size = sizeType
        };
```

The cool thing about the preceding query is that because the size type and materials are cached, you don't need to make any extra table service calls to get the reference data. This is all performed in-memory using LINQ to Objects.

> **WARNING** This technique is super cool for small datasets. Make sure you check your performance on large datasets, such as when you have millions of rows, because it may not meet your needs.

In-memory joins for static data save money

Not only will in-memory joins improve the performance of your application, they'll save you lots of cash. The fewer calls you make to the Table service or SQL Azure, the more money you'll save.

With the Table service, you save money directly by making fewer requests; with SQL Azure, you save money indirectly by requiring fewer SQL databases to service your queries.

The previous example improved performance and saved money by joining static data. Although this works well for static data, it doesn't work so well for nonstatic data—dynamic data or infrequently changing data.

14.3 Joining dynamic and infrequently changing data together

In this section we'll look at the options we have for joining dynamic and infrequently changing data.

In the previous example, we skipped over how you'd represent your shirt in the shopping cart. You have two choices: perform a client-side join (as you did in the previous section with the static data) or duplicate the data. In this section we'll look at how you can duplicate data, and join uncached data.

Let's first look at how we can duplicate data.

14.3.1 Duplicating data instead of joining

Because the shirt data isn't static data but is infrequently changing data, you need to find a different way of associating that data with the dynamic shopping cart item. For this example, you can duplicate the shirt data within the shopping cart item, as shown here:

```
public class ShoppingCartItem : TableServiceEntity
{
   public string ShirtName { get; set; }
   public string ShirtDescription { get; set; }
   public int Price { get; set; }
   public Material Material { get; set; }
   public SizeType Size { get; set; }
}
```

The preceding code stores a complete copy of the selected shirt as part of the shopping cart item's entity.

> **NOTE** This approach will mean that we won't need to perform a join with the Shirts table, but it does mean that our table will be much larger than a traditional relational table, meaning higher storage costs.

Now that you have your shopping cart item, you need to correctly display your `StrongShoppingCartItem` object. The following code shows how you could modify the earlier query (from section 14.2.3) to do this.

```
select new StrongShoppingCartItem
        {
           SelectedShirt = new Shirt {PartitionKey="Shirts",
                             RowKey=shoppingCartItem.ShirtName,
                       Description=shoppingCartItem.ShirtDescription,
                       Price=shoppingCartItem.Price},
           Material = material,
           Size = sizeType
        };
```

As you can see from the preceding code, you can project the duplicate data into the `StrongShoppingCartItem` object by instantiating a new `Shirt` object with the data from the `ShoppingCartItem`.

> **Data synchronization**
>
> Apart from it taking up more space, another issue with duplicating data is data synchronization.
>
> Although the shirt data is infrequently changed, when a change does occur (such as the price), all items that are present in a customer's shopping basket won't be automatically updated with the new price. If your business model allows you to have stale data, this is obviously not a problem. But if your pesky customers want the correct price to be reflected in the shopping basket, you'll need some method of synchronizing the master table to all the duplicates.
>
> A simple method of keeping the data synchronized is to publish a message to a queue, stating that an item has changed. Then you can have a worker role pick up that message and update all items in the table with the correct data.

14.3.2 Client-side joining of uncached data

If data synchronization is a big concern and your dynamic data is a very small set of data, you could take the hit of performing a client-side join. To do that, you'd need to modify the `ShoppingCartItem` to support the join:

```
public class ShoppingCartItem : TableServiceEntity
{
   public Shirt Shirt {get;set;}
   public Material Material { get; set; }
   public SizeType Size { get; set; }
}
```

In the preceding code, the duplicate shirt properties have been replaced with a reference to the shirt.

Now that the entity stores a reference, you need to modify your query to join the data together. The following code shows how you could do this:

```
var materials = (IEnumerable<Material>)Cache["materials"];
var sizeTypes = (IEnumerable<SizeType>)Cache["sizeTypes"];

var shoppingCartItemContext = new ShoppingCartItemContext ();
var shoppingCartItems =
   shoppingCartItemContext.ShoppingCartItem.ToList();

var shirtsContext = new ShirtContext();

var q = from shoppingCartItem in shoppingCartItems
        join sizeType in sizeTypes
           on shoppingCartItem.Size.RowKey equals sizeType.RowKey
        join material in materials
           on shoppingCartItem.Material.RowKey equals material.RowKey
```

```
select new ShoppingCartItem
{
   SelectedShirt = (from shirt in shirtsContext.ShirtTable
                    where shirt.PartitionKey=="Shirts" &&
                    shirt.RowKey ==
                    shoppingCartItem
                     .SelectedShirt.RowKey
                    select shirt).First(),
   Material = material,
   Size = sizeType
};
```

The key thing to note about the preceding example is that the shirt query isn't cached, and it will invoke a call to the Table service for each item returned. The following extract shows where this is performed:

```
SelectedShirt = (from shirt in shirtsContext.ShirtTable
                 where shirt.PartitionKey=="Shirts" &&
                 shirt.RowKey == shoppingCartItem.SelectedShirt.RowKey
                 select shirt).First(),
```

Because the data returned from the shopping cart is small, this is a pretty useful technique. If you were dealing with a much larger set of shopping cart data (such as hundreds of items), this would start to perform badly.

> **TIP** If you're working with infrequently changing data (such as the shirt data), you may wish to consider using SQLite to host a local cached version of your data. This would allow you to perform SQL queries on local cached data without calling out to the Table service or SQL Azure.

Not having the ability to join data across tables does present challenges, but not always impossible ones. For the most part, you can use different techniques to get around those limitations. But in some circumstances it's going to be impossible to use the Table service. If you have lots of dynamic data and need to perform live queries across various table joins, you're not going to be able to represent that easily in the Table service. In these instances, SQL Azure is the most appropriate choice. Similarly, if you need to perform transactions across various tables, SQL Azure is again the right choice.

Lucene.NET

If you're storing your data in SQL Azure or the Table service and you need to perform text searches, you can export your data out of these databases and perform searches with Lucene.NET: http://lucene.apache.org/lucene.net/.

14.4 Summary

In this chapter, we've tried to break away from automatically building pure relational database solutions and instead tried to build hybrid solutions that offer the benefits of both platforms.

You've seen that by breaking foreign-key constraints, you can make effective use of the cache to store your static data, allowing you to use either the Table service or SQL Azure as your underlying storage platform.

You've also seen that if you choose to make use of the Table service (which is much more scalable than SQL Azure), you can easily join your dynamic data back to your static reference data without financial or performance penalties.

Although SQL Azure isn't as naturally scalable as the Table service, we looked at how you can shard SQL Azure to build a highly scalable relational database solution.

Finally we saw that when it comes to infrequently changing reference data (such as the shirt data), what you want to do with that data should influence what technologies you choose. In some circumstances, such as live fresh data, transactional data, or data with many joins across dynamic tables, SQL Azure is the only choice. But if you're looking to join dynamic data with infrequently changing data, you can still use the Table service by thinking about your queries and your performance. Take time to work out how your queries should be structured, consider how many round trips will be made, and think about how wasteful some of your "smaller" queries might be. REST is easy to use, but there can be overhead costs because small queries become chatty over the network.

Think about what your application needs to do before you choose a technology. Don't automatically reach for SQL Azure (which is awesome); consider whether the Table service will meet your needs.

Now that you've moved your data to the cloud, it's time to think about how to accomplish the complex backend processing usually associated with relational databases.

Part 6

Doing work with messages

Part 6, the last leg of this journey, covers several advanced topics.

Chapter 15 discusses worker roles. We covered web roles early in the book; most of that also applies to worker roles. This chapter covers some aspects of worker roles, focusing on how they're different from web roles. We also discuss some more advanced topics we saved for the back of the book so you would get to them when you were good and ready.

Chapter 16 covers the last part of Azure storage: queues. We'll look at how you can use queues to decouple your system and peek at some advanced patterns for queues.

Chapter 17 looks at the grandly titled Windows Azure platform AppFabric services. The Access Control Service (ACS) and the Service Bus help you connect to and protect the services you're running in Azure (or, really, that you're running anywhere).

The final chapter, chapter 18 for those keeping count, focuses on how to use the service management API to watch and control your Azure environment. If you want to gather logs and diagnostics, head to this chapter.

That's it, that's the end of the book. You can now put Cloud Surfer on your resume and retire with fame and wealth.

15

Processing with worker roles

This chapter covers

- Scaling the backend
- Processing messages
- Using the service management APIs to control your application

In Azure there are two roles that run your code. The first, the *web role*, has already been discussed. It plays the role of the web server, communicating with the outside world. The second role is the *worker role*. Worker roles act as backend servers—you might use one to run asynchronous or long-running tasks. Worker roles are usually message based and will usually receive these messages by polling a queue or some other shared storage. Like web roles, you can have multiple deployments of code running in different worker roles. Each deployment can have as many instances as you would like running your code (within your Azure subscription limits).

It's important to remember that a worker role is a template for a server in your application. The service model for your application defines how many instances of

that role need to be started in the cloud. The role definition is similar to a class definition, and the instances are like objects.

If your system has Windows services or batch jobs, they can easily be ported to a worker role. For example, many systems have a series of regularly scheduled backend tasks. These might process all the new orders each night at 11 p.m. Or perhaps you have a positive pay system for banking, and it needs to connect to your bank each day before 3 p.m., except for banking holidays.

The worker role is intended to be started up and left running to process messages. You'll likely want to dynamically increase and decrease the number of instances of your worker role to meet the demand on your system, as it increases and decreases throughout your business cycle.

When you create worker roles, you'll want to keep in mind that Windows Azure doesn't have a job scheduler facility, so you might need to build your own. You could easily build a scheduling worker role that polls a table representing the schedule of work to do. As jobs need to be executed, it could create the appropriate worker instance, pass it a job's instructions, and then shut down the instance when the work is completed. You could easily do this with the service management APIs, which are discussed in chapter 18.

We're going to start off by building a simple service using a worker role. Once we have done that we'll change it several times, to show you the options you have in communicating with your worker role instances.

15.1 A simple worker role service

When it's all said and done, working with worker roles is quite easy. The core of the code for the worker role is the normal business code that gets the work done. There isn't anything special about this part of a worker role. It's the wrapper or handler around the business code that's interesting. There are also some key concepts you'll want to pay attention to, in order to build a reliable and manageable worker role.

In this section, we'll show you how to build a basic worker role service. You have to have some way to communicate with the worker role, so we'll first send messages to the worker through a queue, showing you how to poll a queue. (We won't go too deep into queues, because they're covered thoroughly in chapter 16.) We'll then upgrade the service so you can use inter-role communication to send messages to your service.

We'll use the term *service* fairly loosely when we're talking about worker roles. We see worker roles as providing a service to the rest of the application, hopefully in a decoupled way. We don't necessarily mean to imply the use of WS-* and Web Service protocols, although that's one way to communicate with the role.

Let's roll up our sleeves and come up with a service that does something a little more than return a string saying "Hello World." In the next few sections, we'll build a new service from scratch.

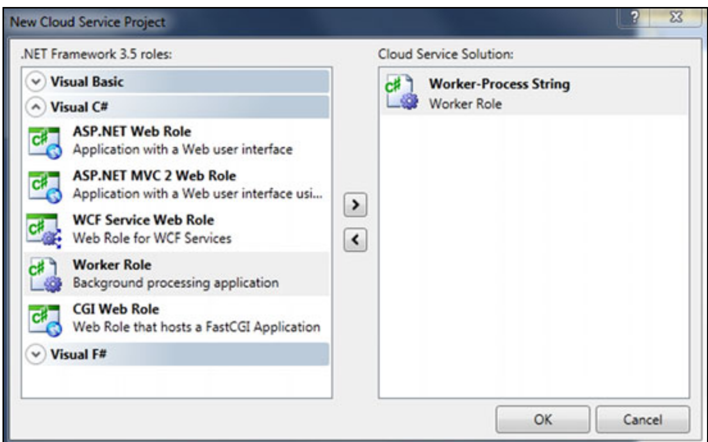


Figure 15.1 To build the service, you'll start with a worker role. It'll do all of the work and make it easy to scale as your business grows, especially during the string-reversal peak season.

15.1.1 No more Hello World

Because Hello World doesn't really cut it as an example this late in any book, we're going to build a service that reverses strings. This is an important service in any business application, and the string-reversal industry is highly competitive.

There will be two parts to this service. The first part will be the code that actually does the work of reversing strings—although it's composed of some unique intellectual property, it isn't very important in our example. This is the business end of the service. The other part of the service gets the messages (requests for work) into the service. This second part can take many shapes, and which design you use comes down to the architectural model you want to support. Some workers never receive messages; they just constantly poll a database, or filesystem, and process what they find.

To build this string-reversal service you need to open up Visual Studio 2010 and start a new cloud project. For this project, add one worker role, and give it the name *Worker-Process String*, as shown in figure 15.1.

At the business end will be our proprietary and award-winning algorithm for reversing strings. We intend to take the string-reversal industry by storm and really upset some industry captains. The core method will be called `ReverseString`, and it will take a string as its only parameter. You can find the secret sauce in the following listing. Careful, don't post it on a blog or anything.

Listing 15.1 The magical string-reversal method

```
private string ReverseString(string originalString)
{
   int lengthOfString = originalString.Length;
   char[]reversedBuffer = new char[lengthOfString];

   for (int i = 0; i < lengthOfString; i++)
```

```
    {
       reversedBuffer [i] = originalString[lengthOfString - 1 - i];
    }

    return new string(reversedBuffer);
}
```

The code in the previous listing is fairly simple—it's normal .NET code that you could write on any platform that supports .NET (mobile, desktop, on-premises servers, and so on), not just for the cloud. The method declares a character array to be a buffer that's the same length as the original string (because our R&D department has discovered that every reversed string is exactly as long as the original string). It then loops over the string, taking characters off the end of the original string and putting them at the front of the buffer, moving along the string and the buffer in opposite directions. Finally, the string in the buffer is returned to the caller.

For this example, we'll put this business logic right in the `WorkerRole.cs` class. Normally this code would be contained in its own class, and would be referenced into the application. You can do that later if you want, but we want to keep the example simple so you can focus on what's important.

We've chosen to put this service in a worker in the cloud so that we can dynamically scale how many servers we have running the service, based on usage and demand. We don't want to distract our fledgling company from writing the next generation of string-reversal software with the details and costs of running servers.

If you ran this project right now, you wouldn't see anything happen. The cloud simulator on your desktop would start up, and the worker role would be instantiated, but nothing would get done. By default, the worker service comes with an infinite polling loop in the `Run` method. This `Run` method is what is called once the role instance is initialized and is ready to run. We like that they called it `Run`, but calling it `DoIt` would have been funnier.

Now that you have your code in the worker role, how do you access it and use it? The next section will focus on the two primary ways you can send messages to a worker role instance in an active way.

15.2 Communicating with a worker role

Worker roles can receive the messages they need to process in either a push or a pull way. Pushing a message to the worker instance is an active approach, where you're directly giving it work to do. The alternative is to have the role instances call out to some shared source to gather work to do, in a sense pulling in the messages they need. When pulling messages in, remember that there will possibly be several instances pulling in work. You'll need a mechanism similar to what the Azure Queue service provides to avoid conflicts between the different worker role instances that are trying to process the same work.

Keep in mind the difference between roles and role instances, which we covered earlier. Although it's sometimes convenient to think of workers as a single entity, they

don't run as a role when they're running, but as one or more instances of that role. When you're designing and developing your worker roles, keep this duality in mind. Think of the role as a unit of deployment and management, and the role instance as the unit of work assignment. This will help reduce the number of problems in your architecture.

One advantage that worker roles have over web roles is that they can have as many service endpoints as they like, using almost any transport protocol and port. Web roles are limited to HTTP/S and can have two endpoints at most. We'll use the worker role's flexibility to provide several ways to send it messages.

We'll cover three approaches to sending messages to a worker role instance:

- A pull model, where each worker role instance polls a queue for work to be completed
- A push model, where a producer outside Azure sends messages to the worker role instance
- A push model, where a producer inside the Azure application sends messages to the worker role instance

Let's look first at the pull model.

15.2.1 Consuming messages from a queue

The most common way for a worker role to receive messages is through a queue. This will be covered in depth in chapter 16 (which is on messaging with the queue), but we'll cover it briefly here.

The general model is to have a `while` loop that never quits. This approach is so common that the standard worker role template in Visual Studio provides one for you. The role instance tries to get a new message from the queue it's polling on each iteration of the loop. If it gets a message, it'll process the message. If it doesn't, it'll wait a period of time (perhaps 5 seconds) and then poll the queue again.

The core of the loop calls the business code. Once the loop has a message, it passes the message off to the code that does the work. Once that work is done, the message is deleted from the queue, and the loop polls the queue again.

```
while (true)
{
      CloudQueueMessage msg = queue.GetMessage();
      if (msg != null)
      {
          DoWorkHere(msg);
          queue.DeleteMessage(msg);
      }
      else
      {
           Thread.Sleep(5000);
      }
}
```

You might jump to the conclusion that you could easily poll an Azure Table for work instead of polling a queue. Perhaps you have a property in your table called `Status` that defaults to `new`. The worker role could poll the table, looking for all entities whose `Status` property equals `new`. Once a list is returned, the worker could process each entity and set their `Status` to `complete`. At its base, this sounds like a simple approach.

Unfortunately, this approach is a red herring. It suffers from some severe drawbacks that you might not find until you're in testing or production because they won't show up until you have multiple instances of your role running.

The first problem is of concurrency. If you have multiple instances of your worker role polling a table, they could each retrieve the same entities in their queries. This would result in those entities being processed multiple times, possibly leading to status updates getting entangled. This is the exact concurrency problem the Azure Queue service was designed to avoid.

The other, more important, issue is one of recoverability and durability. You want your system to be able to recover if there's a problem processing a particular entity. Perhaps you have each worker role set the `status` property to the name of the instance to track that the entity is being worked on by a particular instance. When the work is completed, the instance would then set the `status` property to `done`. On the surface, this approach seems to make sense. The flaw is that when an instance fails during processing (which will happen), the entity will never be recovered and processed. It'll remain flagged with the instance name of the worker processing the item, so it'll never be cleared and will never be picked up in the query of the table to be processed. It will, in effect, be "hung." The system administrator would have to go in and manually reset the `status` property back to `new`. There isn't a way for the entity to be recovered from a failure and be reassigned to another instance.

It would take a fair amount of code to overcome the issues of polling a table by multiple consumers, and in the end you'd end up having built the same thing as the Azure Queue service. The Queue service is designed to play this role, and it removes the need to write all of this dirty plumbing code. The Queue service provides a way for work to be distributed among multiple worker instances, and to easily recover that work if the instance fails. A key concept of cloud architecture is to design for failure recoverability in an application. It's to be expected that nodes go down (for one reason or another) and will be restarted and recovered, possibly on a completely different server.

Queues are the easiest way to get messages into a worker role, and they'll be discussed in detail in the next chapter. Now, though, we'll discuss inter-role communication, which lets a worker role receive a message from outside of Azure.

15.2.2 Exposing a service to the outside world

Web roles are built to receive traffic from outside of Azure. Their whole point in life is to receive messages from the internet (usually from a browser) and respond with some message (usually HTML). The great thing is that when you have multiple web role instances, they're automatically enrolled in a load balancer. This load balancer automatically distributes the load across the different instances you have running.

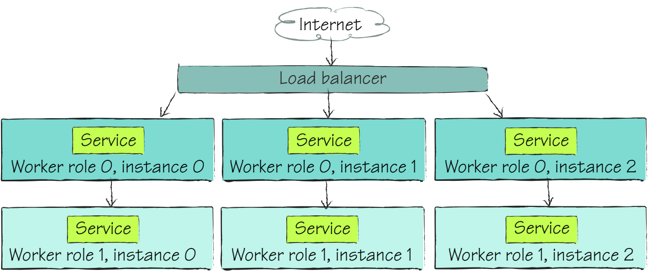


Figure 15.2 Worker roles have two ways of exposing their services. The first is as an input service—these are published to the load balancer and are available externally (role 0). The second is as an internal service, which isn't behind a load balancer and is only visible to your other role instances (role 1).

Worker roles can do much the same thing, but because you aren't running in IIS (which isn't available on a worker role), you have to host the service yourself. The only real option is to build the service as a WCF service.

Our goal is to convert our little string-reversal method into a WCF service, and then expose that externally so that customers can call the service. The first step is to remove the loop that polls the queue and put in some service plumbing. When you host a service in a worker role, regardless of whether it is for external or internal use, you need to declare an endpoint. How you configure this endpoint will determine whether it allows traffic from sources internal or external to the application. The two types of endpoints are shown in figure 15.2. If it's configured to run externally, it will use the Azure load balancers and distribute service calls across all of the role instances running the server, much like how the web role does this. We'll look at internal service endpoints in the next section.

The next step in the process is to define the endpoint. You can do this the macho way in the configuration of the role, or you can do it in the Visual Studio Properties window. If you right-click on the Worker-Process String worker role in the Azure project and choose Properties, you'll see the window in figure 15.3.

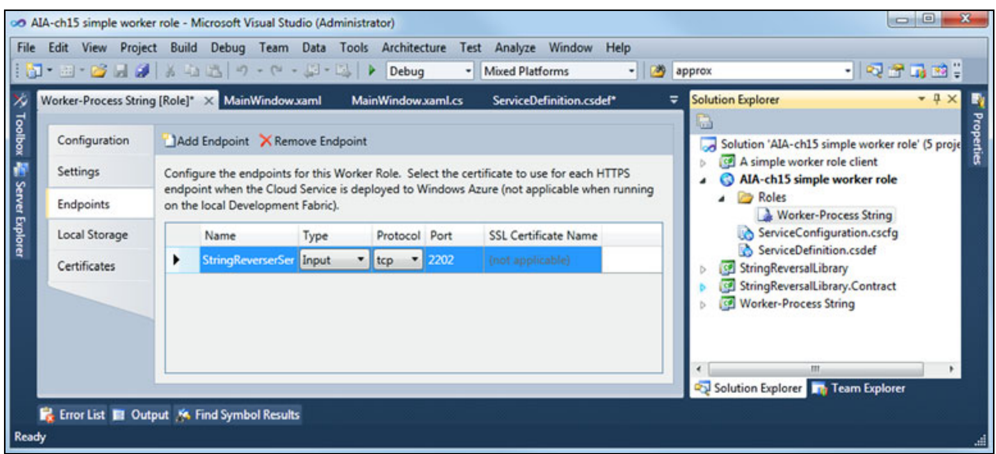


Figure 15.3 Adding an external service endpoint to a worker role. This service endpoint will be managed by Azure and be enrolled in the load balancer. This will make the service available outside of Azure.

Name the service endpoint `StringReverseService` and set it to be an input endpoint, using TCP on port 2202. There's no need to use any certificates or security at this time.

After you save these settings, you'll find the equivalent settings in the ServiceConfiguration.csdef file:

```
<Endpoints>
 <InputEndpoint name="StringReverserService" protocol="tcp" port="2202" />
</Endpoints>
```

You might normally host your service in IIS or WAS, but those aren't available in a worker role. In the future, you might be able to use Windows Server AppFabric, but that isn't available yet, so you'll have to do this the old-fashioned way. You'll have to host the WCF service using `ServiceHost`, which is exactly that, a host that will act as a container to run your service in. It will contain the service, manage the endpoints and configuration, and handle the incoming service requests.

Next you need to add a method called `StartStringReversalService`. This method will wire up the service to the `ServiceHost` and the endpoint you defined. The contents of this method are shown in the following listing.

Listing 15.2 The `StartStringReversalService` method wires up the service

```
private void StartStringReversalService()
{
   this.serviceHost = new ServiceHost(typeof(ReverseStringTools));

   NetTcpBinding binding = new NetTcpBinding(SecurityMode.None);

   RoleInstanceEndpoint externalEndPoint = RoleEnvironment
   ➥ .CurrentRoleInstance
   ➥ .InstanceEndpoints["StringReverserService"];

serviceHost.AddServiceEndpoint(
        typeof(IReverseString),
        binding,
        String.Format("net.tcp://{0}/StringReverserService",
        ➥ ExternalEndPoint.IPEndpoint));

   try
   {
      this.serviceHost.Open();
   }
   catch (Exception ex)
   {
      Trace.TraceError("Could not start string reverser servicehost. {0}",
      ➥ ex.Message);
   }

   while (true)
   {
      Thread.Sleep(500000);
   }
}
```

(1) Retrieves endpoint settings from configuration

(2) Starts service host

(3) Sleeps forever so service host stays open

Listing 15.2 is an abbreviated version of the real method, shortened so that it fits into the book better. We didn't take out anything that's super important. We took out a series of `trace` commands so we could watch the startup and status of the service. We also abbreviated some of the error handling, something you would definitely want to beef up in a production environment.

Most of this code is normal for setting up a `ServiceHost`. You first have to tell the service host the type of the service that's going to be hosted (1). In this case, it's the `ReverseStringTools` type.

When you go to add the service endpoint to the service host, you're going to need three things, the ABCs of WCF: address, binding, and contract. The contract is provided by your code, `IReverseString`, and it's a class file that you can reference to share service contract information (or use MEX like a normal web service). The binding is a normal TCP binary binding, with all security turned off. (We would only run with security off for debug and demo purposes!)

Then the address is needed. You can set up the address by referencing the service endpoint from the Azure project. You won't know the real IP address the service will be running under until runtime, so you'll have to build it on the fly by accessing the collection of endpoints from the `RoleEnvironment.CurrentRoleInstance.InstanceEndpoints` collection (2). The collection is a dictionary, so you can pull out the endpoint you want to reference with the name you used when setting it up—in this case, `StringReverserService`. Once you have a reference to the endpoint, you can access the IP address that you need to set up the service host.

After you have that wired up, you can start the service host. This will plug in all the components, fire them up, and start listening for incoming messages. This is done with the `Open` method (3).

Once the service is up, you'll want the main execution thread to sleep forever so that the host stays up and running. If you didn't include the sleep loop (3), the call pointer would fall out of the method, and you'd lose your context, losing the service host. At this point, the worker role instance is sitting there, sleeping, whereas the service host is running, listening for and responding to messages.

We wired up a simple WPF test client, as shown in figure 15.4, to see if our service is working. There are several ways you could write this test harness. If you're using .NET 4,

Figure 15.4 A simple client that consumes our super string-reversing service. The service is running in a worker role, running in Azure, behind the load balancers. kltpzyxM! kltpzyxM! kltpzyxM!

it's very common to use unit tests to test your services instead of an interactive WPF client. Your other option would be to use WCFTestClient.exe, which comes with Visual Studio.

Exposing public service endpoints is useful, but there are times when you'll want to expose services for just your use, and you don't want them made public. In this case, you'll want to use inter-role communication, which we'll look at next.

15.2.3 Inter-role communication

Exposing service input endpoints, as we just discussed, can be useful. But many times, you just need a way to communicate between your role instances. Usually you could use a queue, but at times there might be a need for direct communication, either for performance reasons or because the process is synchronous in nature.

You can enable communication directly from one role instance to another, but there are some issues you should be aware of first. The biggest issue is that you'll have direct access to an individual role instance, which means there's no separation that can deal with load balancing. Similarly, if you're communicating with an instance and it goes down, your work is lost. You'll have to write code to handle this possibility on the client side.

To set up inter-role communication, you need to add an internal endpoint in the same way you add an input endpoint, but in this case you'll set the type to `Internal` (instead of `Input`), as shown in figure 15.5. The port will automatically be set to `dynamic` and will be managed for you under the covers by Azure.

Using an internal endpoint is a lot like using an external endpoint, from the point of view of your service. Either way, your service doesn't know about any other instances running the service in parallel. The load balancing is handled outside of your code when you're using an external endpoint, and internal endpoints don't have any available load balancing. This places the choice of which service instance to consume on the shoulders of the service consumer itself.

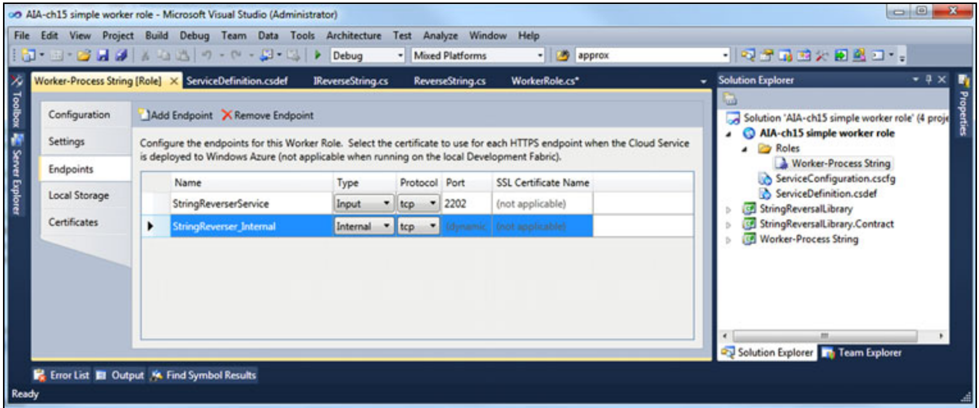


Figure 15.5 You can set up an internal endpoint in the same way you set up an external endpoint. In this case, though, your service won't be load balanced, and the client will have to know which service instance to talk to.

Most of the work involved with internal endpoints is handled on the client side, your service consumer. Because there can be a varying number of instances of your service running at any time, you have to be prepared to decide which instance to talk to, if not all of them. You also have to be wily enough to not call yourself if calling the service from a sibling worker role instance.

You can access the set of instances running, and their exposed internal endpoints, with the `RoleEnvironment` static class:

```
foreach (var instance in
➥ RoleEnvironment.CurrentRoleInstance.Role.Instances)
{
   if (instance != RoleEnvironment.CurrentRoleInstance)
      SendMessage(instance.InstanceEndpoints["MyServiceEndpointName"]);
}
```

The preceding sample code loops through all of the available role instances of the current role. As it loops, it could access a collection of any type of role in the application, including itself. So, for each instance, the code checks to see if that instance is the instance the code is running in. If it isn't, the code will send that instance a message. If it's the same instance, the code won't send it a message, because sending a message to oneself is usually not productive.

All three ways of communicating with a worker role have their advantages and disadvantages, and each has a role to play in your architecture:

- Use a queue for complete separation of your instances from the service consumers.
- Use input endpoints to expose your service publicly and leverage the Azure load balancer.
- Use internal endpoints for direct and synchronous communication with a specific instance of your service.

Now that we've covered how you can communicate with a worker role, we should probably talk about what you're likely to want to do with a worker role.

15.3 Common uses for worker roles

Worker roles are blank slates—you can do almost anything with them. In this section, we're going to explore some common, and some maybe not-so-common, uses for worker roles.

The most common use is to offload work from the frontend. This is a common architecture in many applications, in the cloud or not. We'll also look at how to use multithreading in roles, how to simulate a worker role, and how to break a large process into connected smaller pieces.

15.3.1 Offloading work from the frontend

We're all familiar with the user experience of putting products into a shopping cart and then checking out with an online retailer. You might have even bought this book

online. How retailers process your cart and your order is one of the key scenarios for how a worker role might be used in the cloud.

Many large online retailers split the checkout process into two pieces. The first piece is interactive and user-facing. You happily fill your shopping cart with lots of stuff and then check out. At that time, the application gathers your payment details, gives you an order number, and tells you that the order has been processed. Then it emails all of this so you can have it all for your records. This is the notification email shown in figure 15.6.

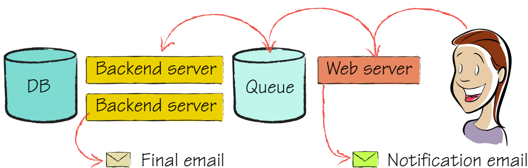


Figure 15.6 The typical online retailer will process a customer's order in two stages. The first saves the cart for processing and immediately sends back a thank you email with an order number. Then the backend servers pick up the order and process it, resulting in a final email with all of the real details.

After the customer-facing work is done, the backend magic kicks in to complete the processing of the order. You see, when the retailer gave you an order number, they were sort of fibbing. All they did was submit the order to the backend processing system via a message queue and give you the order number that can be used to track it. One of the servers that are part of the backend processing group picks up the order and completes the processing. This probably involves charging the credit card, verifying inventory, and determining the ability to ship according to the customer's wishes. Once this backend work is completed, a second email is sent to the customer with an update, usually including the package tracking number and any other final details. This is the final email shown in figure 15.6.

By breaking the system into two pieces, the online retailer gains a few advantages. The biggest is that the user's experience of checking out is much faster, giving them a nice shopping experience. This also takes a lot of load off of the web servers, which should be simple HTML shovels. Because only a fraction of shoppers actually check out (e-tailers call this the conversion rate), it's important to be able to scale the web servers very easily. Having them broken out makes it easy to scale them horizontally (by adding more servers), and makes it possible for each web server to require only simple hardware. The general strategy at the web server tier is to have an army of ants, or many low-end servers.

This two-piece system also makes it easier to plan for failure. You wouldn't want a web server to crash while processing a customer's order and lose the revenue, would you?

This leaves the backend all the time it needs to process the orders. Backend server farms tend to consist of fewer, larger servers, when compared to the web servers. Although you can scale the number of backend servers as well, you won't have to do that as often, because you can just let the flood of orders back up in the queue. As long as your server capacity can process them in a few hours, that's OK.

Azure provides a variety of server sizes for your instances to run on, and sometimes you'll want more horsepower in one box for what you're doing. In that case, you can use threading on the server to tap that entire horsepower.

15.3.2 Using threads in a worker role

There may be times when the work assigned to a particular worker role instance needs multithreading, or the ability to process work in parallel by using separate threads of execution. This is especially true when you're migrating an existing application to the Azure platform. Developing and debugging multithreaded applications is very difficult, so deciding to use multithreading isn't a decision you should make lightly.

The worker role does allow for the creation and management of threads for your use, but as with code running on a normal server, you don't want to create too many threads. When the number of threads increases, so does the amount of memory in use. The context-switching cost of the CPU will also hinder efficient use of your resources. You should limit the number of threads you're using to two to four per CPU core.

A common scenario is to spin up an extra thread in the background to process some asynchronous work. Doing this is OK, but if you plan on building a massive computational engine, you're better off using a framework to do the heavy lifting for you. The Parallel Extensions to .NET is a framework Microsoft has developed to help you parallelize your software. The Parallel Extensions to .NET shipped as part of .NET 4.0 in April of 2010.

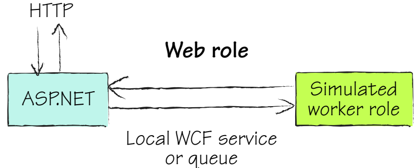
Although we always want to logically separate our code to make it easier to maintain, sometimes the work involved doesn't need a lot of horsepower, so we may want to deploy both the web and the worker sides of the application to one single web role.

15.3.3 Simulating worker roles in a web role

Architecting your application into discrete pieces, some of which are frontend and some of which are backend, is a good thing. But there are times when you need the logical separation, but not the physical separation. This might be for speed reasons, or because you don't want to pay for a whole worker role instance when you just need some lightweight background work done.

MAINTAINING LOGICAL SEPARATION

If you go down this path, you must architect your system so you can easily break it out into a real worker role later on as your needs change. This means making sure that while you're breaking the physical separation, you're at least keeping the logical separation. You should still use the normal methods of passing messages to that worker code. If it would use a queue to process messages in a real worker instance, it should use a queue in the simulated worker instance as well. Take a gander at figure 15.7 to see what we mean. At some point, you'll need to break the code back out to a real worker role, and you won't want to have to rewrite a whole bunch of code.



Be aware that the Fabric Controller will be ignorant of what you're doing, and it won't be able to manage your simulated worker role. If that worker role code goes out

of control, it will take down the web instance it's running in, which could cascade to a series of other problems. You've been warned.

If you're going to do this, make sure to put the worker code into a separate library so that primary concerns of the web instance aren't intermingled with the concerns of the faux worker instance. You can then reference that library and execute it in its own thread, passing messages to it however you would like. This will also make it much easier to split it out into its own real worker role later.

UTILIZING BACKGROUND THREADS

The other issue is getting a background thread running so it can execute the faux worker code. An approach we've worked with is to launch the process on a separate thread during the `Session_Start` event of the global.asax. This will fire up the thread once when the web app is starting up, and leave it running.

Our first instinct was to use the `Application_Start` event, but this won't work. The `RoleManager` isn't available in the `Application_Start` event, so it's too early to start the faux worker.

We want to run the following code:

```
Thread t = new Thread(new ThreadStart(FauxWorkerSample.Start));
t.Start();
```

Putting the thread start code in the `Session_Start` event has the effect of trying to start another faux worker every time a new ASP.NET session is started, which is whenever there's a new visitor to the website. To protect against thousands of background faux workers being started, we use the Singleton pattern. This pattern will make sure that only one faux worker is started in that web instance.

When we're about to create the thread, we check a flag in the application state to see if a worker has already been created:

```
object obj = Application["FauxWorkerStarted"];

if (obj == null)
{
    Application["FauxWorkerStarted"] = true;
    Thread t = new Thread(new ThreadStart(FauxWorkerSample.Start));
    t.Start();
}
```

If the worker hasn't been created, the flag won't exist in the application state property bag, so it will equal `null` in that case. If this is the first session, the thread will be created, pointed at the method we give it (`FauxWorkerSample.Start` in this case), and it will start processing in the background.

When you start it in this manner, you'll have access to the `RoleManager` with the ability to write to the log, manage system health, and act like a normal worker instance. You could adapt this strategy to work with the `OnStart` event handler in your webrole.cs file. This might be a cleaner place to put it, but we wanted to show you the dirty work around here.

Our next approach is going to cover how best to handle a large and complex worker role.

15.3.4 State-directed workers

Sometimes the code that a worker role runs is large and complex, and this can lead to a long and risky processing time. In this section, we'll look at a strategy you can use to break this large piece down into manageable pieces, and a way to gain flexibility in your processing.

As we've said time and time again, worker roles tend to be message-centric. The best way to scale them is by having a group of instances take turns consuming messages from a queue. As the load on the queue increases, you can easily add more instances of the worker role. As the queue cools off, you can destroy some instances.

In this section, we'll look at why large worker roles can be problematic, how we can fix this problem, and what the inevitable drawbacks are. Let's start by looking at the pitfalls of using a few, very large workers.

THE PROBLEM

Sometimes the work that's needed on a message is large and complicated, which leads to a heavy, bloated worker. This heaviness also leads to a brittle codebase that's difficult to work with and maintain because of the various code paths and routing logic.

A worker that takes a long time to process a single request is harder to scale and can't process as many messages as a group of smaller workers. A long-running unit of work also exposes your system to more risk. The longer an item takes to be processed, the more likely it is that the work will fail and have to be started over. This is no big deal if the processing takes 3 seconds, but if it takes 20 minutes or 20 hours, you have a significant cost to failure.

This problem can be caused by one message being very complex to process, or by a batch of messages being processed as a group. In either case, the unit of work being performed is large, and this raises risk. This problem is often called the "pig in a python" problem (as shown in figure 15.8), because you end up with one large chunk of work moving through your systems.

We need a way to digest this work a little more gracefully.

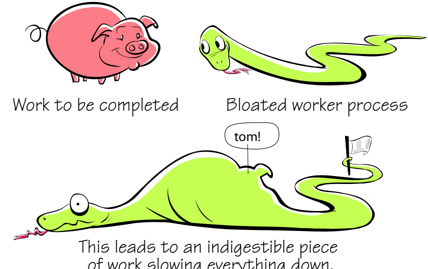


Figure 15.8 The "pig in a python" problem can often be seen in technology and business. It's when a unit of work takes a long time to complete, like when a python eats a pig. It can take months for the snake to digest the pig, and it can't do much of anything else during that timeframe.

THE SOLUTION

The best way to digest this large pig is to break the large unit of work into a set of smaller processes. This will give you the most flexibility when it comes to scaling and managing your system. But you want to be careful that you don't break the processes down to sizes that are too small. At this level, the latency of communicating with the queue and

other storage mechanisms in very chatty ways may introduce more overhead than you were looking for.

When you analyze the stages of processing on the message, you'll likely conceive of several stages to the work. You can figure this out by drawing a flow diagram of the current bloated worker code. For example, when processing an order from an e-commerce site, you might have the following stages:

1. Validate the data in the order.
2. Validate the pricing and discount codes.
3. Enrich the order with all of the relevant customer data.
4. Validate the shipping address.
5. Validate the payment information.
6. Charge the credit card.
7. Verify that the products are in stock and able to be shipped.
8. Enter the shipping orders into the logistics system for the distribution center.
9. Record the transaction in the ERP system.
10. Send a notification email to the customer.
11. Sit back and profit.

You can think of each state the message goes through as a separate worker role, connected together with a queue for each state. Instead of one worker doing all of the work for a single order, it only processes one of the states for each order. The different queues represent the different states the message could have. Figure 15.9 compares a big worker that performs all of the work, to a series of smaller workers that break the work out (validating, shipping, and notifying workers).

There might also be some other processing states you want to plan for. Perhaps one for really bad orders that need to be looked at by a real human, or perhaps you have platinum-level customers who get their orders processed and shipped before normal run-of-the-mill customers. The platinum orders would go into a queue that's processed by a dedicated pool of instances.

You could even have a bad order routed to an Azure table. A customer service representative could then access that data with a CRM application or a simple InfoPath

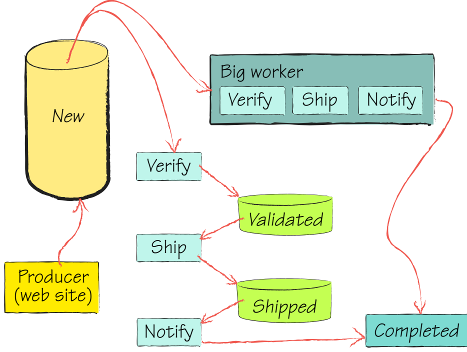


Figure 15.9 A monolithic worker role compared to a state-driven worker role. The big worker completes all the work in one step, leading to the "pig in a python" problem of being harder to maintain and extend as needed. Instead, we can break the process into a series of queues and workers, each dedicated to servicing a specific state or stage of the work to be done.

form, fix the order, and resubmit it back into the proper queue to continue being processed. This process is called *repair and resubmit*, and it's an important element to have in any enterprise processing engine.

You won't be able to put the full order details into the queue message—there won't be enough room. The message should contain a complete work ticket, representing where the order data can be found (perhaps via an order ID), as well as some state information, and any information that would be useful in routing the message through the state machine. This might include the service class of the customer, for example—platinum versus silver.

As the business changes over time, and it will, making changes to how the order is processed is much easier than trying to perform heart surgery on your older, super complicated, and bloated work role code. They don't say spaghetti code for nothing. For example, you might need to add a new step between steps 8 and 9 in our previous list. You could simply create a new queue and a new worker role to process that queue. Then the worker role for the state right before the new one would need to be updated to point to the new queue. Hopefully the changes to the existing parts of the system can be limited to configuration changes.

EVEN COOLER—MAKE THE STATE WORKER ROLE ITS OWN AZURE SERVICE

How you want to manage your application in the cloud should be a primary consideration in how you structure the Visual Studio solution. Each solution becomes a single management point. If you want to manage different pieces without affecting the whole system, those should be split out into separate solutions.

In this scenario, it would make sense to separate each state worker role to its own service in Azure, which would further decouple them from each other. This way, when you need to restart one worker role and its queue, you won't affect the other roles.

In a more dynamic organization, you might need to route a message through these states based on some information that's only available at runtime. The routing information could be stored in a table, with rules for how the flow works, or by simply storing the states and their relationships in the cloud service configuration file. Both of these approaches would let you update how orders were processed at runtime without having to change code. We've done this when orders needed different stages depending on what was in the order, or where it was going. In one case, if a controlled substance was in the order, the processing engine had to execute a series of additional steps to complete the order.

This approach is often called a *poor man's service bus* because it uses a simple way of connecting the states together, and they're fairly firm at runtime. If you require a greater degree of flexibility in the workflow, you would want to look at the Itinerary[1] pattern. This lets the system build up a schedule of processing stops based on the information present at runtime. These systems can get a little more complicated, but they result in a system that's more easily maintained when there's a complex business process.

---

[1] For more information on the Itinerary pattern, see the *Microsoft Application Architecture Guide* from Patterns & Practices at Microsoft. It can be found at http://apparchguide.codeplex.com.

OOPS, IT'S NOT NIRVANA

As you build this out, you'll discover a drawback. You now have many more running worker roles to manage. This can create more costs, and you still have to plan for when you eventually will swallow a pig. If your system is tuned for a slow work day, with one role instance per state, and you suddenly receive a flood of orders, the large amount of orders will move down the state diagram like a pig does when it's eaten by a python. This forces you to scale up the number of worker instances at each state.

Although this flexibility is great, it can get expensive. With this model, you have several pools of instances instead of one general-purpose pool, which results in each pool having to increase and then decrease as the pig (the large flood of work) moves through the pipeline. In the case of a pig coming through, this can lead to a stall in the state machine as each state has to wait for more instances to be added to its pool to handle the pig (flood of work). This can be done easily using the service management APIs, but it takes time to spin up and spin down instances—perhaps 20 minutes.

The next step to take, to avoid the pig in a python problem, is to build your worker roles so that they're generic processors, all able to process any state in the system. You would still keep the separate queues, which makes it easier to know how many messages are in each state.

You could also condense the queues down to one, with each message declaring what state the order is in as part of its data, but we don't like this approach because it leads to favoritism for the most recent orders placed in the processors, and it requires you to restart all of your generic workers when you change the state graph. You can avoid this particular downfall by driving the routing logic with configuration and dependency injection. Then you would only need to update the configuration of the system and deploy a new assembly to change the behavior of the system.

The trick to gaining both flexibility and simplicity in your architecture is to encapsulate the logic for each state in the worker, separating it so it's easily maintainable, while pulling them all together so there's only one pool of workers. The worker, in essence, becomes a router. You can see how this might work in figure 15.10. Each message is routed, based on its state and other runtime data, to the necessary state

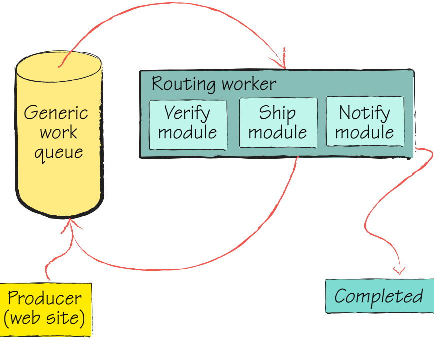


Figure 15.10 By moving to a consolidated state-directed worker, we'll have one queue and one worker. The worker will act as a router, sending each inbound message to the appropriate module based on the message's state and related itinerary. This allows us to have one large pool of workers, but makes it easier to manage and decompose our bulky process.

processor. This functions much like a factory. Each state would have a class that knows how to process that state. Each state class would implement the same interface, perhaps `IorderProcessStage`. This would make it easy for the worker to instantiate the correct class based on the state, and then process it. Most of these classes would then send the message back to the generic queue, with a new state, and the cycle would start again.

There are going to be times when you're working with both web and worker roles and you're either importing legacy code that needs access to a local drive, or what you're doing requires it. That's why we'll discuss local storage next.

15.4 Working with local storage

There are times when the code you're working with will need to read from and write to the local filesystem. Windows Azure allows for you to request and access a piece of the local disk on your role instance.

You can create this space by using the configuration of your role. You won't have control over the path of the directory you're given access to, so you should make sure that the file path your code needs to access is part of your configuration. A hardcoded path will never remain accurate in the cloud environment.

We recommend that you only use local storage when you absolutely have to, because of some limitations we'll cover later in this section. You'll likely need to use local storage the most when you're migrating to the cloud existing frameworks or applications that require local disk access.

15.4.1 Setting up local storage

You can configure the local storage area you need as part of your role by adding a few simple lines of configuration to your role. The tag we're going to work with is the `LocalStorage` tag. It will tell the Fabric Controller to allocate local file storage space on each server the role instance is running on.

In the configuration element, you need to name the storage space. This name will become the name of the folder that's reserved for you. You'll need to define how much filesystem space you'll need. The current limit is 20 GB per role instance, with a minimum of 1 MB.

```
<LocalResources>
      <LocalStorage name="FilesUploaded" cleanOnRoleRecycle="false"
      ➥ sizeInMB="15" />
      <LocalStorage name="VirusScanPending" cleanOnRoleRecycle="true"
      ➥ sizeInMB="5" />
</LocalResources>
```

You can declare multiple local storage resources, as shown in the preceding code snippet. It's important that the local file storage only be used for temporary, unimportant files. The local file store isn't replicated or preserved in any way. If the instance fails

and it's moved by the Fabric Controller to a new server, the local file store isn't preserved, which means any files that were present will be lost.

> **TIP** There is one time when the local file storage won't be lost, and that's when the role is recycled, either as part of a service management event on your part, or when the Fabric Controller is responding to a minor issue with your server. In these cases, if you've set the `cleanOnRoleRecyle` parameter to `false`, the current files will still be there when your instance comes back online.

Instances may only access their own local storage. An instance may not access another instance's storage. You should use Azure BLOB storage if you need more than one instance to access the same storage area.

Now that you've defined your local storage, let's look at how you can access it and work with it.

15.4.2 Working with local storage

Working with files in local storage is just like working with normal files. When your role instance is started, the agent creates a folder with the name you defined in the configuration in a special area on the C: drive on your server. Rules are put in place to make sure the folder doesn't exceed its assigned quota for size. To start using it, you simply need to get a handle for it.

To get a handle to your local storage area, you need to use the `GetLocalResource` method. You'll need to provide the name of the local resource you defined in the service definition file. This will return a `LocalResource` object:

```
public static LocalResource uploadFolder =
➥ RoleEnvironment.GetLocalResource("FilesUploaded");
```

After you have this reference to the local folder, you can start using it like a normal directory. To get the physical path, so you can check the directory contents or write files to it, you would use the `uploadFolder` reference from the preceding code.

```
string rootPathName = uploadFolder.RootPath;
```

In the sample code provided with this book, there's a simple web role that uses local storage to store uploaded files. Please remember that this is just a sample, and that you wouldn't normally persist important files to the local store, considering its transient nature. You can view the code we used to do this in listing 15.3. When calling the `RootPath` method in the local development fabric, Brian's storage is located here:

```
C:\Users\brprince\AppData\Local\dftmp\s0\deployment(32)\res\deployment(32)
➥ .AiA_15___Local_Storage_post_pdc.LocalStorage_WebRole.0\directory\File
➥ sUploaded\
```

When we publish this little application to the cloud, it returns the following path:

```
C:\Resources\directory\0c28d4f68a444ea380288bf8160006ae.LocalStorage
➥ _WebRole.FilesUploaded\
```

Listing 15.3 Working with local file storage

```
protected void Page_Load(object sender, EventArgs e)
{
 litName.Text = uploadFolder.Name;
 litMaxSize.Text = uploadFolder.MaximumSizeInMegabytes.ToString();
 litRootPath.Text = uploadFolder.RootPath;
}

protected void cmdUpload1_Click(object sender, EventArgs e)
{
 if (FileUpload1.HasFile)
 {
  FileUpload1.SaveAs
  ➥ (uploadFolder.RootPath + FileUpload1.FileName);
 }

 litUploadedFilePath.Text = uploadFolder.RootPath +
 ➥ FileUpload1.FileName;
 litUploadedFileContents.Text =
 ➥ System.IO.File.OpenText(uploadFolder.RootPath +
 ➥ FileUpload1.FileName).ReadToEnd();
}
```

(1) Path to our local folder

(2) Path to the file uploaded

Now that we know where the files will be stored, we can start working with them. In the sample application, we have a simple file-upload control (1). When the web page is loaded, we write out the local file path to the local storage folder that we've been assigned (2). Once the file is uploaded, we store it in the local storage and write out its filename and path (3). We then write the file back out to the browser using normal file APIs to do so. Our example code was designed to work only with text files, to keep things simple.

The local storage option is great for volatile local file access, but it isn't durable and may disappear on you. If you need durable storage, look at Azure storage or SQL Azure. If you need shared storage that's super-fast, you should consider the Windows Server AppFabric distributed cache. This is a peer-to-peer caching layer that can run on your roles and provide a shared in-memory cache for your instances to work with.

15.5 Summary

In this chapter, we've looked at how you can process work in the background with the worker role in Azure. The worker role is an important tool for the cloud developer. It lets you do work when there isn't a user present, whether because you've intentionally separated the background process from the user (in the case of a long-running check-out process) or because you've broken your work into a discrete service that will process messages from a queue.

Worker roles scale just like web roles, but they don't have a built-in load balancer like web roles do. You'll usually aggregate worker roles behind a queue, with each instance processing messages from the queue, thereby distributing the work across

the group. This gives you the flexibility to increase or decrease the number of worker instances as the need arises.

It's quite possible to have an Azure application consist of only worker roles. You could have some on-premises transaction systems report system activity (such as each time a sale is made) to a queue in the cloud. The worker role would be there to pick up the report and merge the data into the reporting system. This allows you to keep the bulk of your application on-premises, while moving the computing-intensive back-end operations to the cloud. A more robust way of doing this would be to connect the on-premises system with the cloud system using the Windows Azure platform AppFabric Service Bus, which is discussed in chapter 17.

In this chapter we talked a lot about how to work with worker roles, and how to get messages to them. One of the key methods for doing that is to use an Azure queue. We'll work closely with queues in the next chapter.

16 Messaging with the queue

This chapter covers

- Loosely coupling your system
- Distributing work to a group of service providers
- Learning how to use messaging

Queues are the third part of the Azure storage system (after BLOBs and tables). The concept of queues has been around a long time, and it's likely that you've worked with some technology related to queues already.

A common architectural goal during design is to produce a system that's tightly integrated, but also loosely coupled. Any sizable system usually has several components, and whether these components are running in the same memory space, or on different boxes, they need to work closely together. This is what is meant by "tightly integrated." These different components should work as a team to provide the value of the system in an easy and cohesive manner.

If your only goal is tight integration, you'll often end up with a system where the components are tightly coupled as well. Tight coupling leads to a system that's brittle and that responds poorly to changes. This makes it difficult to manage the system and to extend it to meet future needs. In a brittle system, a change in one

component can ripple through the whole system, requiring changes in many other components. Such a system is difficult to understand, maintain, and troubleshoot.

The easiest way to create a loosely coupled system is to provide a way for the components to talk with each other through messages, and these messages should follow a "tell, don't ask" approach. You shouldn't ask an object for a bunch of data, do some work with it, and then give the results back to the object for recording. You should just tell the object what you want it to do. This approach should be applied at a component and system level as well—this approach helps to create code that's well abstracted and compartmentalized.

Loose coupling also helps you isolate change from one component to another. For example, an e-commerce website may be communicating with a backend ERP system. When the company chooses to change ERP vendors, the queue will act as a buffer, keeping the change from rippling over the queue boundary. All the producer knows is that it puts messages in a certain format in the queue. The producer has no knowledge of the consuming system, and doesn't care what happens.

The queue also becomes a pivot point for scaling. Later on we'll talk about how you can monitor the length (some use the term *depth*) of a queue to determine whether messages are being consumed quickly enough. If not, you can scale out the number of consumers processing the messages, which reduces the number of waiting messages in the queue.

Be careful, however, that you don't put so much effort into loosely coupling your system that you end up building an overly complex monstrosity that's completely unmanageable. As with many things, balance is the key.

16.1 Decoupling your system with messaging

There are many ways to decouple your system, but they usually center on messaging of some sort. One of the most common ways is to use queues in between the different parts of the system, or between completely different systems that must work together.

Queues have several major components, and we'll walk through each of them in turn. We're first going to look at how queues work in general—how they pass messages around. Then we'll examine what messages are, the shape they have, and how they work. Finally we'll look closely at how an Azure queue works—what its limits are and how to get the most out of it.

16.1.1 How messaging works

Queues have two ends. The producer end, where messages are put into the queue, is usually represented as the bottom. The other end is the consumer end, where a consumer will pull messages off of the top of the queue.

Performance is critical to every part of Azure, and queues are no exception. Each queue, like the rest of the Azure storage services, exists as three instances, each of which is protected by different fault and update domains. This strategy protects your queue from completely failing when a switch goes down or a patch is rolled out.

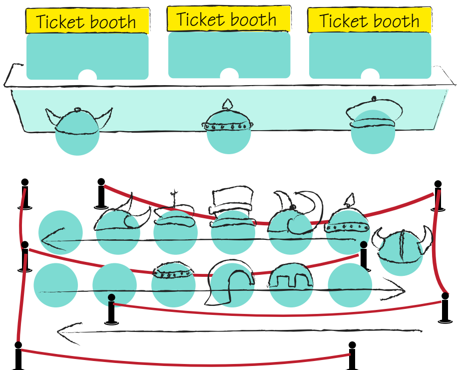


Figure 16.1 A queue forms for tickets on the opening night of a new blockbuster movie. Movie-goers enter (while wearing their fanboy outfits) at the bottom, or end of the line. As the ticket booth (consumer) processes the ticket requests, the movie-goers move forward in the queue until they're at the head of the line.

As the demand for a queue increases, the storage fabric will start serving the requests out of a memory cache. This dramatically increases the performance of the queue and reduces the latency of using the queue.

A queue is a FIFO structure: first in, first out. This contrasts with a stack, which is LIFO: last in, first out. A real-world example of a queue is the line for tickets at a movie theater, as illustrated in figure 16.1. When people arrive, they stand at the end of the line. As the consumer (the ticket booth) completes sales, it works with the next person at the head of the line, and as people buy their tickets, the line moves forward.

At a busy movie theater, there may be many ticket booths consuming customers from the line. Management may open more ticket booths, based on the length of the line or based on how long people have to wait in the line. As the processing capacity of the ticket counter is increased, the theatre is able to sell tickets to more customers each minute. If the line gets short at a particular time, the theater manager might close down ticket booths until only one or two are left open.

Your system can use a similar concept with a queue. As the producer side of your system (the shopping cart checkout process, for example) produces messages, they're placed in the queue. The consumer side of the system (the ERP system that processes the orders and charges credit cards) will pull messages off of the queue. In this way, the two systems are tightly integrated but loosely coupled, because of the queue in between.

Queues are one-way in nature. A message goes in at the bottom, moves towards the top, and is eventually consumed, as you can see in figure 16.2. In order for the consumer message to communicate back to the producer, a separate process must be used. This could be a queue going in the other direction, but it's usually some other mechanism, like a shared storage location.

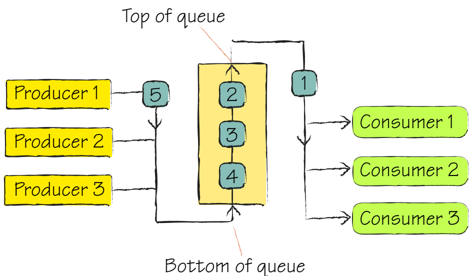


Figure 16.2 Producers place messages into the queue, and consumers get them out. Each queue can have multiple produces and consumers.

There's an inherent order to a queue, but you can't usually rely on queues for strict ordered delivery. In some scenarios,

this can be important. A consumer processing checkouts from an e-commerce website won't need the messages in a precise order, but a set of doctor's orders for a patient might. It won't matter which checkout is processed first, as long as it's in a reasonable order, but the order of what tests, drugs, and surgeries are performed on a patient is likely important. We'll explore some ways of handling this situation in Azure later in this chapter.

16.1.2 What is a message?

Your Azure storage account can have many queues; at any time, a queue can have many messages. Messages are the lifeblood of a queue system and should represent the things that a producer is telling a consumer. You can think of a queue as having a name, some properties, and a collection of ordered messages.

In Azure, messages are limited to 8 KB in size. This low limit is designed for performance and scalability reasons. If a message could be up to 1 GB in size, writing to and reading from the queue would take a long time. This would also make it hard for the queue to respond quickly when there were many different consumers reading messages from the top of the queue.

Because of this limit, most Azure queue messages will follow a work ticket pattern. The message will usually not contain the data needed by the consumer itself. Instead, the message will contain a pointer of some sort to the real work that needs to be done.

For example, following along with figure 16.3, a queue that contains messages for video compression won't include the actual video that needs to be compressed. The producer will store the video in a shared storage location (1), perhaps a BLOB container or a table. Once the video is stored, the producer will then place a message in the queue with the name of the BLOB that needs to be compressed (2). There'll likely be other jobs in the queue as well.

The consumer will then pick up the work ticket, fetch the proper video from BLOB storage, compress the video (3), and then store the new video back in BLOB storage (4). Sometimes the process ends there, with the original producer being smart enough to look in the BLOB storage for the compressed version of the video, or perhaps a flag in a database is flipped to show that the processing has been completed.

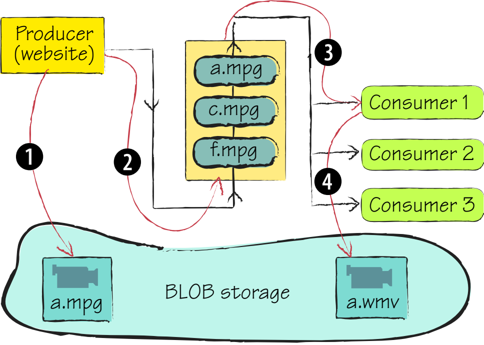


Figure 16.3 Work tickets are used in queues to tell the consumer what work needs to be done. This keeps the messages small, and keeps the queue scalable and performant. The work ticket is usually a pointer to where the real work is.

The content of a queue message is always stored as a string. The string must be in a format that can be included in an XML message and be UTF-8 encoded. This is because a message is returned from the queue in an XML format, with your real message as part of that XML. It's possible to store binary data, but you need to serialize and deserialize the data yourself. Keep in mind that when you're deserializing, the content coming out of the message will be base64 encoded.

The content of the message isn't the only part of the message that you may want to work with. Every message has several important properties, as you can see in the following listing.

Listing 16.1 A message in its native XML format

```
<QueueMessagesList>
 <QueueMessage>
  <MessageId>20be3f61-b70f-47c7-
  ➥ 9f87-abbf4c71182b</MessageId>
  <InsertionTime>Fri, 07 Aug 2009
  ➥ 00:58:41 GMT</InsertionTime>
  <ExpirationTime>Fri, 14 Aug 2009 00:58:41 GMT</ExpirationTime>
  <PopReceipt>NzBjM2QwZDYtMzFjMC00MGVhLThiOTEtZ
  ➥ DcxODBlZDczYjA4</PopReceipt>
  <TimeNextVisible>Fri, 07 Aug 2009 00:59:16 GMT</TimeNextVisible>
  <MessageText>PHNhbXBsZT5zYW1wbGUgbWVzc2FnZTwvc2FtcGxlPg==</MessageText>
 </QueueMessage>
</QueueMessagesList>
```

1 Unique message ID

2 Date and time message was placed on queue

The ID property is assigned by the storage system, and it's unique 1. This is the only way to uniquely differentiate messages from each other because several messages could contain the same content.

A message also includes the time and date the message was inserted into the queue 2. It can be handy to see how long the message has been waiting to be processed. For example, you might use this information to determine whether the messages are becoming stale in the queue. This timestamp is also used by the storage service to determine if your message should be garbage collected or not. Any message that's about a week old in any queue will be collected and discarded.

Now that we've discussed what messages are, we're ready to discuss what contains them—the queue itself.

16.1.3 What is a queue?

The queue is the mechanism that holds the messages, in a rough order, until they're consumed. The queue is replicated in triplicate throughout the storage service, like tables and BLOBs, for redundancy and performance reasons.

Queues can be created in a static manner, perhaps as part of deploying your application. They can also be created and destroyed dynamically. This is handy when you need a way to organize and direct messages in different directions based on real-time data or user needs.

Each queue can have an unlimited number of messages. The only real limit is how fast you can process the messages, and whether you can do so before they're garbage collected after one week's time.

Because a queue's name appears in the URI for the REST request, it needs to follow the constraints that DNS names have:

- It must start with a letter or number, and can contain only letters, numbers, and the hyphen (-) character.
- The first and last letters in the queue name must be alphanumeric. The hyphen (-) character may not be the first or last character.
- All letters in a queue name must be lowercase. (This is the requirement that gets me every time.)
- A queue name must be from 3 to 63 characters long.

A queue also has a set of metadata associated with it. This metadata can be up to 8 KB in size and is a simple collection of name/value key pairs. This metadata can help you track and manage your queues. Although the name of a queue can help you understand what the use of the queue is, the metadata can be useful in a more dynamic situation. For example, the name of the queue might be the customer number that the queue is related to, but you could store the customer's service level (tin, silver, molybdenum, and gold) as a piece of metadata. This metadata then lives with the queue and can be accessed by any producer or consumer of the queue.

Queues are both a reliable and persistent way to store and move messages. They're reliable in that you should never lose a message—we'll look at how this works in section 16.4 when we discuss the message lifecycle. Queues are also strict in how they persist your messages. If a server goes down, the messages aren't lost, they remain in the queue. This differs from a purely memory-based system, in which all of the messages would be lost if the server were to have a failure.

16.1.4 StorageClient and the REST API

There are two basic ways to interact with a queue and its messages. The first is the StorageClient library that ships with the Azure SDK. The other mechanism for interacting with queues is to use the REST API directly. You can create and consume REST messages in any way you want. Although this is a little more work, it's worth learning how the REST API works, so that you understand more fully how the storage system works.

The REST entry point will be your key way to access Azure storage when you don't have a handy API lying around, like the StorageClient. Microsoft and several open source teams are working to build SDKs similar to the StorageClient library for every platform, including Python and PHP. All of these libraries use the REST protocols under the hood.

Each call into the REST API has a request header that includes some basic information. The header needs to include which version of the service you're targeting, the

date and time of the request, and the authorization header. You can see a sample header in the following listing.

Listing 16.2 A sample REST request header

```
POST /queue21b3c6dfe8626450880b9e16c70e2425e/messages?timeout=30 HTTP/1.1
x-ms-date: Fri, 07 Aug 2009 01:26:38 GMT
Authorization: SharedKey hsslog:Iow8eYFGeodLGqXrgbEcwDuA+aNOR0emEC9uy3Vnggg=
Host: hsslog.queue.core.windows.net
Content-Length: 80
Expect: 100-continue
```

The service version header is useful for preventing an update to the queue service from disrupting your system. You can force your requests to be processed by a specific version of the storage service, allowing you to control when you support and leverage new features in a newer version of the service. If you omit the version header, your request will be routed to the default version of the service.

A queue can't be made public or anonymous. Every operation against a queue must be authenticated with the shared key method. Constructing the authorization header for queue requests is the same as for BLOBs and tables.

Now that you know how to forge the header, or the envelope, for a message, let's look at how to send commands to the queue.

16.2 Working with basic queue operations

To show you how to use the basic queue API operations, we're going to build a queue browser. This little tool (shown in figure 16.4) will help you debug any system you're

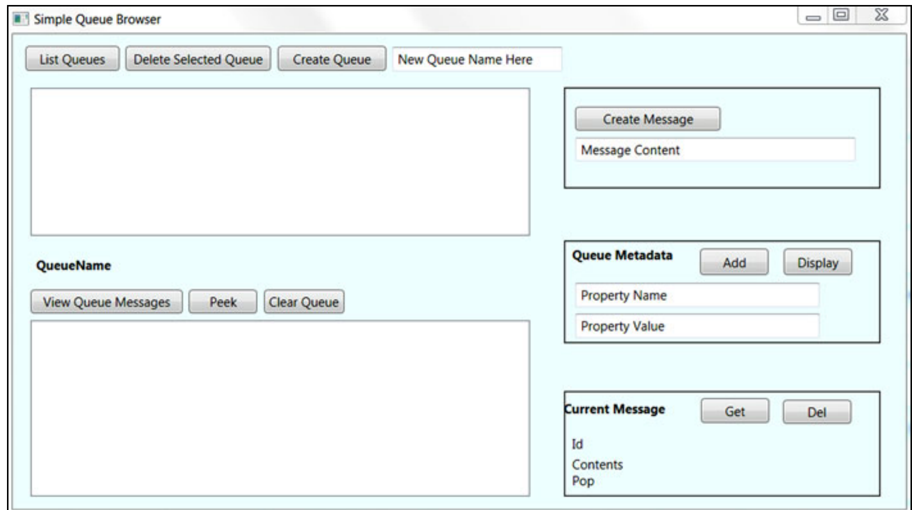


Figure 16.4 A screenshot of the simple queue browser we'll build in this chapter. It's important to note that your authors, while charming, aren't graphic designers or UX specialists. This tool will act as a vehicle for understanding the basics of working with queues.

building, by letting you look at the queues that are being used and see how they're working.

We'll be focusing on creating methods that perform each of the following operations of the browser:

- `ListQueues()`—Lists the queues that exist in your storage account
- `Create()` or `CreateIfNotExist()`—Creates queues in your account
- `SetMetadata()`—Writes metadata
- `Clear()`—Clears a queue of all of its pending messages
- `Delete()`—Deletes a queue or a message from the system

We aren't going to focus on how WPF works, or the best application architecture for this little application. This is meant to be "me-ware"—something that works for you and doesn't have to work for anyone else. You should use it as a harness to play with the different APIs and learn how they work.

16.2.1 Get a list of queues

There are several basic queue operations that you'll need to be able to work with. The first one we're going to look at is a method that will tell you what queues exist in your account. You may not always need this. Usually, you'll know what the queue for your application is and provision it for yourself.

To get a list of the available queues, you need to first connect to the queue service and then call the method. You can see how in the following listing.

Listing 16.3 Connecting to the queue service and getting a list of queues

```
private CloudQueueClient Qsvc;
private IEnumerable<CloudQueue> qList;

CloudStorageAccount storageAccount =
➥ CloudStorageAccount.FromConfigurationSetting("DataConnectionString");

Qsvc = storageAccount.CreateCloudQueueClient();    <-- (1) Creates queue client to interact with queue

qList = Qsvc.ListQueues();
```

You'll use something like line (1) quite often. This creates a connection to the service, similar to how you create a connection object to a database. You'll want to create this once for a block of code and hold it in memory so that you aren't always reconnecting to the service. In this little application, we create this object in the constructor for the window itself and store it in a variable at the form level. This makes it available to every method in the form and saves you the trouble of having to continuously recreate the object.

The `CloudStorageAccount` serves as a factory that creates service objects that represent the Azure Queue storage service, called the `CloudQueueClient`. There are several ways to create the `CloudQueueClient`. The most common approach is to create it as in listing 16.3, by using the `FromConfigurationSetting` method. This looks into

your configuration and sets up all of the URIs, usernames, account names, and so on. This is better than having to set four or five different parameters when you're newing up the connection.

Once you have a handle to the Queue service, you can call `ListQueues`. This doesn't just return a list of strings, as you might expect, but instead returns a collection of queue objects. Each queue object represents a queue in your account and is fully usable. The equivalent call to get a list of queues in REST would be like the following code. You can see that it's a simple `GET`.

```
GET http://hsslog.queue.core.windows.net/
➥ ?comp=list&maxresults=50&timeout=30
```

If you don't have any queues, you'll get back an empty collection. In this case, your next step would be to create a queue.

16.2.2 Creating a queue

Once you're connected to the queue service you can create a queue with the queue client. You do this by creating a reference to the queue, even if the queue doesn't exist yet. Then you can create the queue using the reference as a control point:

```
CloudQueue q = Qsvc.GetQueueReference("newordersqueue");
q.CreateIfNotExist();
```

In the preceding code, we first create a `CloudQueue` object. This is an empty object, and this line doesn't actually connect to the service. It's merely an empty handle that doesn't point to anything. Then the `CreateIfNotExist` method is called on the queue object. This will check if a queue with that name exists, and if it doesn't, it'll create one. This is very handy.

You can check whether a queue exists before you try to create it by using `q.DoesQueueExist`. This method will return a Boolean value telling you whether the queue exists or not.

Your next step is to attach some metadata to the queue.

16.2.3 Attaching metadata

You can store up to 8 KB of data in the property bag for each queue. You might want to use this to track some core data about the queue, or perhaps some core metrics on how often it should be polled. To work with the metadata, use the following code:

```
CloudQueue q = Qsvc.GetQueueReference("newordersqueue");
q.Metadata.Add("ProjectName", "ElectronReintroductionPhasing");
q.Metadata.Remove("BadKeyNoDoughnut");
q.SetMetadata();
```

The metadata for a queue is attached as a property on the `CloudQueue` object. You work with it like any other name value collection. In this code, we first add a new entry to the metadata called `ProjectName`, with a value of `ElectronReintroductionPhasing`. But this new entry won't be saved back to the queue service until we call `SetMetaData`, which connects to the service and uploads the metadata for the queue in the cloud.

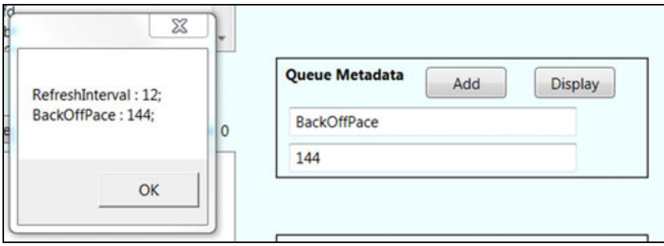


Figure 16.5 Displaying the metadata set on a queue. You can attach up to 8 KB of metadata to a queue. This can help in managing the queue, such as by specifying what the backoff pace rate should be.

You can remove existing properties from the bag if you no longer need them. In this example, the `Remove` method removes `BadKeyNoDoughnut` from use. When you remove an item from the metadata collection, you must follow that with a `SetMetaData` call to persist the changes to the cloud.

In our queue example application, we've set some metadata properties, namely `RefreshInterval` and `BackOffPace`. You can see how we set and fetch these in figure 16.5.

Now that you've created a queue and set its metadata, let's look at how you can delete a queue. On occasion, you'll want to dynamically create and destroy queues.

16.2.4 Deleting a queue

It's good practice to clear a queue before you delete it. This removes all of the messages from the queue. The clear queue method is handy for resetting a system or clearing out poison messages that may have stopped up the flow.

Deleting a queue is as simple as this, when using the client library:

```
CloudQueue q = Qsvc.GetQueueReference("newordersqueue");
q.Clear();
q.Delete();
```

The equivalent REST call would be as follows:

```
DELETE http://hsslog.queue.core.windows.net/newordersqueue?timeout=30
```

Being able to create and destroy queues in a single line of code makes them simple objects to work with. In the past, using a queue in your system would require days, if not weeks, of installing several queue servers (for redundancy purposes). They would also require a lot of care and feeding. Queues in the cloud are much easier to work with and don't require any grooming or maintenance. But the real power of queues is in the messages that flow through them, which we'll delve into in the next section.

16.3 Working with messages

Now that you know how to work with queues, let's look at how you can work with messages. As we mentioned above, a queue is a FIFO structure, similar to a line at the movie theater. The first action we usually take with a queue it to put a message in it, or *enqueue* a message.

16.3.1 Putting a message on the queue

When you put a message on the queue, the new message is placed onto the bottom or end of the queue. When you get a message from the queue, it's from the top or front of the queue. Here we have a few lines of code that show how to add a message to a queue:

```
CloudQueue q = Qsvc.GetQueueReference("newordersqueue");
CloudQueueMessage theNewMessage = new CloudQueueMessage("cart:31415");
q.AddMessage(theNewMessage);
```

To add a message to the queue, you need to get a reference to the queue, as we did in the preceding code. Once you have the queue reference, you can call the `AddMessage` method and pass in a `CloudQueueMessage` object. Creating the message is a simple affair; in this case we're simply passing in text that will be the content of the message. You can also pass in a byte array if you're passing some serialized binary data. Remember that the content of each message is limited to 8 KB in size. When you put a message on a queue, a REST message is generated and sent to the queue. The following listing shows you a sample of what that might look like. This sample is for entertainment purposes only.

Listing 16.4 An example of putting a message onto a queue with REST

```
POST /my-special-queue/messages?timeout=30 HTTP/1.1
x-ms-date: Fri, 07 Aug 2009 01:49:25 GMT
Authorization: SharedKey hsslog:3oJPdtrUK47gMSpHfwrmasdnT5nJpGpszg=
Host: hsslog.queue.core.windows.net
Content-Length: 80
Expect: 100-continue

<QueueMessage><MessageText>cart:31415</MessageText></QueueMessage>
```

In this example, we're adding a message with an order number that can be found in the related Azure table. The consumer will pick up the message, unwrap the content, and process the cart numbered 31415. Their shopping cart is probably filled with pie and pie related accessories.

Before we show you how to get a message, we want to talk about peeking.

16.3.2 Peeking at messages

Peeking is a way to get the content of a message in the queue without taking the message off of the queue. This leaves the message still on the queue, so someone else can grab it. Many people peek at messages to determine if they want to process the message or not, or to determine how they should process the message. You can see how to peek in this snippet of code:

```
CloudQueueMessage m = q.PeekMessage();

private IEnumerable<CloudQueueMessage> mList;
mList = q.PeekMessages(10);
```

Peeking at messages is easy. Calling the `PeekMessage` method returns a single message—the one at the front of the queue. You can peek at more than one message by calling `PeekMessages` and providing the number of messages you want returned. In the preceding example, we asked for 10 messages.

Now that you've peeked at the messages, you're ready to get them.

16.3.3 Getting messages

You don't have to peek at a message before getting it, and many times you won't use peek at all. Getting a message off of the queue is simple, as shown here:

```
private CloudQueueMessage currentMsg;
currentMsg = q.GetMessage();
```

If you already have a reference to your queue, getting a message is as simple as calling `GetMessage`. There's one override that lets you determine the visibility timeout of the get. We'll discuss the lifecycle of a message in section 16.4.

Getting the contents of the message, so that you can work with it, is quite simple, especially if it was string data and not binary data:

```
string s = currentMsg.AsString;
```

Once you have a message, you can use the `AsString` or `AsBytes` properties to access the contents of the message. This is the meat of the message, and the part you're most likely interested in.

Once you've processed a message, you'll want to delete it. This takes it off of the queue.

16.3.4 Deleting messages

Deleting a message is as easy as getting it:

```
q.DeleteMessage(currentMsg);
```

To delete a message, you need to pass the message back into the `DeleteMessage` method of the queue object. You can also do it easily with REST. You can only delete one message at a time. The `DELETE` command in REST would look like the following example. Notice that all of the pertinent data needed is in the query string.

```
DELETE http://hsslog.queue.core.windows.net/my-special
➥ -queue/messages/f5104ff3-260c-48c8-
➥ 9c35-cd8ffe3d5ace?popreceipt=AgAAAAEAAAAAAAAA1vkQXwEXygE%3d&timeout=30
```

Regardless of how you delete the message, through REST or the API, be prepared to handle any exceptions that might be thrown.

One aspect of messages that we haven't looked at yet is their lifecycle. What happens when a message is retrieved? How does the queue keep the same message from being picked up by several consumers at the same time? Is a message self-aware? These are important questions for a queue service. Never losing a message (known as durability) is critical to a queue.

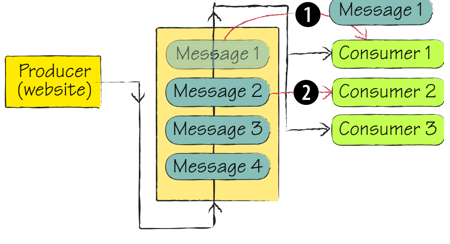
16.4 Understanding message visibility

A key aspect of a queue is how it manages its messages and their visibility. This is how the queue implements the message durability developers are looking for. The goal is to protect against a consumer getting a message, and then failing to process and delete that message. If that happened, the message would be lost, and this isn't good news for any processing system.

Visibility timeouts and idempotency are the two best tools for making sure your messages are never lost. Understanding how these concepts relate to the queue and understanding the lifecycle of a message are important to the success of your code.

16.4.1 About message visibility and invisibility

Every message has a visibility timeout property. When a message is pulled from the queue, it isn't really deleted; it's just temporarily marked as invisible. The consumer is also given a receipt (called the pop receipt) that's unique to that `GetMessage()` operation. The duration of invisibility can be controlled by the consumer, and it can be as long as 2 hours. If not explicitly set, it will default to 30 seconds. While a message is invisible, it won't be given out in response to new `GetMessage()` operations.



As an example, let's say a producer has placed four messages in a queue, as shown in figure 16.6, and we have two consumers that will be reading messages out of the queue.

Consumer 1 gets a message (msg 1), and that message is marked invisible (1). Seconds later, consumer 2 performs a get operation as well. Because the first message (msg 1) is invisible, the queue responds with the next message (msg 2) (2).

Not long thereafter, consumer 1 finishes processing msg 1 and performs a delete operation on the message. As part of the delete operation, the queue checks the pop receipt consumer 1 provides when it passes in the message. This is to make sure consumer 1 is the most recent reader of the message in question. The receipt matches in this case, and the message is deleted.

Consumer 1 then does an immediate read, and gets msg 3. Consumer 1 fails to complete processing within the invisibility time window and fails to delete msg 3 in time. It becomes visible again.

Just at that time, consumer 2 deletes msg 2 and does a get. The queue responds with msg 3, because it's visible again. While consumer 2 is processing msg 3, consumer 1 does finally finish processing msg 3 and tries to delete it. This time, the pop receipt, which consumer 1 has, doesn't match the most recently provided pop receipt, which was given to consumer 2 when msg 3 was handed out for a second time. Because the pop receipt doesn't match, an error is thrown, and the message isn't deleted. You'll

likely see a 400 (`Bad Request`) error when this happens. The inner exception details will explain that there aren't any messages with a matching pop receipt available in the queue to be deleted.

16.4.2 Setting visibility timeout

You can set the length of the visibility timeout when you get the message. This lets you determine the proper length of the timeout for each individual message.

When you specify the visibility timeout, you want to balance the expected processing time and how long it will take for the system to recover from an error in processing. If the timeout is too long, it will take a long time for the system to recover a lost message. If the timeout is too short, too many of your messages will be repeatedly reprocessed.

This leads us to an important aspect of queues in general, but specifically the Azure queue system.

16.4.3 Planning on failure

The Queue service guarantee is worded as promising that every message will be processed, *at least once*. You can see this "at least once" business in the previous scenario. Because consumer 1 failed to delete the message in time, the queue handed it out to another consumer. The queue has to assume the original consumer has failed in some way.

This is very useful because it provides a way for your system to take a hit (a server going down) and keep on going. Cloud architecture plans on failure and makes that central to the structure of the system. Queues provide that capability quite nicely.

The downside is that it's possible that a consumer doesn't crash but just takes longer to process the message than intended. In this case, you need to make sure that your processing code is either idempotent, or that it checks before processing each message to make sure it isn't a duplicate copy. Because the message being reprocessed is actually the same message, its ID property will be the same. This makes it easy to check a history table or perhaps the status of a related order before processing starts.

This little bit of complexity might make you think about deleting a message as soon as you receive it—before you process it. Doing so is dangerous and unwise because there will be failure along the way, and when that happens, the message would be lost forever.

16.4.4 Use idempotent processing code

The goal of a messaging system is to make sure you never lose a message. No matter how small or large, you never want to lose an order, or a set of instructions, or anything else you might be processing.

To avoid complexity, it's best to make sure your processing code is *idempotent*. Idempotent means that the process can be executed several times and the system will

result in the same state. Suppose you're working with a piece of software that tracks dog food delivery. When the food is delivered to the physical address, the handheld computer sends a message to your queue in the cloud. The software uploads the physical signature of the recipient to BLOB storage and submits an order-delivered message to the queue. The message contains the time of delivery and the order number, which happens to also be the filename of the signature in BLOB storage.

When this message is processed, the consumer copies the signature image to permanent storage, with the proper filename, and marks the order as delivered in the package-tracking database.

If this message were to be processed several times, there would be no detriment to the system. The signature file would be overwritten with the same file. The order status is already set to delivered, so we'd just be setting its status to delivered again. Using the same delivery time doesn't change the overall state of the system.

This is the best way to handle the processing of queue messages, but it isn't always possible. The next section will discuss some common queue-processing patterns, and some of them deal with working around this issue.

16.5 Patterns for message processing

As simple as queues are, they can prove valuable in a lot of complex scenarios. This section will focus on some common approaches developers tend to use with queues.

16.5.1 Shared counters

You might run into a scenario where a piece of work is broken into many small pieces, and you need to make sure all of those small pieces are completed before you move on to the next step in your process. Sometimes these pieces are subsets of the main problem.

This is called *single instruction, multiple data*. The same processing will be performed to each piece of data, but each piece is a subset of the whole. Consider working on an image. If you break the image into 100 pieces and apply the same process to each piece, you need to know for sure that all 100 pieces are completed before you can stitch them back into the larger picture.

If you just break the image into 100 pieces and throw them into the queue, it can be difficult to know for sure when all of the 100 units of work have been completed. This has to do with the visibility timeout and the nondeterministic nature of queues. You might think that you could simply check the estimated length of the queue using `q.ApproximateMessageCount`:

```
q.RetrieveApproximateMessageCount();
if (q.ApproximateMessageCount != null)
   int remainingMsg = q.ApproximateMessageCount;
```

If you do, you must call `RetrieveApproximateMessageCount`, which fetches the information into the `ApproximateMessageCount` property from the queue in the cloud,

before you call the property itself. This property returns an approximate count of the items in the queue, not an exact count, for two reasons. The first is that the queue is running in triplicate, and an add operation might have completed on one instance but not the other two, which would lead to an inconsistent result. The second reason is that you might get a zero back from your check, only to have an invisible message turn visible again when its timeout expires. Then you would have a message in the queue you didn't know about.

You need a deterministic way to know for sure that all 100 pieces have been processed. One way to do this is to use a shared counter, perhaps in an Azure table. You can see a visualization of this process in figure 16.7.

When the processing starts, a table is made with a counter set to 0. As items are submitted into the queue by the producer (2), the counter is incremented (1). As the work is completed in the consumer and the messages are deleted (3), the counter is decremented (4).

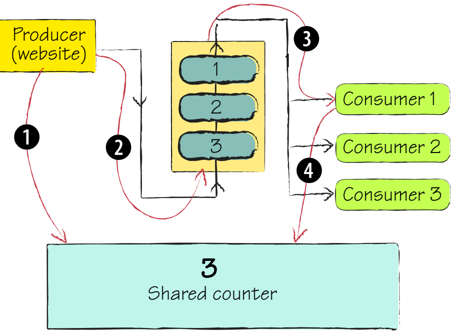


Figure 16.7 Using a shared counter is one way to deterministically track how many messages have been processed. This is a good approach if you have a specific number of messages to process and you need a precise count.

There is one small flaw this approach suffers from, and it's a flaw all shared counters have: it's possible to run into a concurrency problem. If one process reads the counter, adds 1 to it, and then writes it back to the table while another process is doing the same thing, they could end up overwriting each other, resulting in losing track of the count. In order to fix this, you need to use locking on the counter. Another solution, which is used in eventually consistent scenarios, is to read the counter a second time before you write to it, to make sure it wasn't changed by someone else while you weren't looking.

This approach will give you a simple count indicating the progress of the work. What if you want to know which pieces are done and which aren't? No problem. You can do this with a small change to the previous approach.

Instead of writing to a shared counter on each put operation, create a new record in the shared table. This will result in one record per queue message. As they're processed and the queue messages are deleted, the corresponding row should be deleted in the table. Another option would be to mark a property of the row in the table as complete, or store a completed time and date for performance tracking.

In either of these ways—with the shared counter or the shared message tracking table—you can know with a simple query whether all of the work has been completed or not. You should think about wiring up a management portal that monitors the counter or table to show the progress to an administrator.

16.5.2 Work complete receipt

The preceding scenario works when you can control the producer and you have a closed loop. But what if you don't own the producer, or there are too many producers for you to make them also manage a counter? In this case, you can use a return receipt, or a work complete receipt.

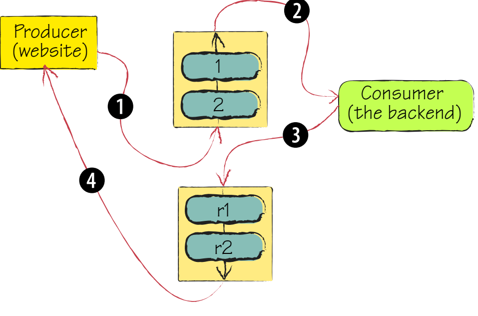
In this approach, as work is completed, a message is sent through a separate channel, perhaps another queue, back to the producer. This alerts the producer that the work is done. This is common in scenarios where the process takes a long time to complete, and the producer wants an asynchronous notification when the work is done.

Instead of using a return queue, we've also had the consumer call a small notification web service on the producer side, sending a simple message regarding the status of the work. This makes the consumer an active part of the process, and it removes the need for the producer to monitor a queue and become a consumer itself.

16.5.3 Asymmetric queues versus symmetric queues

Queues are decidedly one-way. They're a way for one or more producers to communicate with one or more consumers, but not the other way around.

Using one queue in this manner is an *asymmetric queue*. Generally, in an asymmetric queue, the producer finds out about the work being completed in a passive way. The new file happens to be in the right place when the user hits Refresh, or the customer receives an email when the order is shipped, or any number of other scenarios.



As was discussed in section 16.5.2 about work complete receipts, sometimes using symmetric queues can be useful. This makes the response from the consumer back to the producer an active one. Using a queue to do this does help decouple the two halves of the system, but it can lead to too much complexity. This also turns the original producer into a consumer in its own right, which can be hard to implement if the original producer was a website. Because websites only respond to outside requests (a person performs a `GET` or `POST` with a web browser to view the catalog), they don't have a running process to proactively read the queue and respond to it. How this might work is laid out in figure 16.8.

In figure 16.8, you can see two queues connecting the consumers and the producers together. The top queue is the inbound queue, sending messages to the consumers. The bottom queue is the outbound queue, bringing messages back to the consumers. A typical message pattern would be for the producer to send a message to

the consumer by putting it in the inbound queue ❶. The consumer picks up the message ❷, does some critical business work, like ordering pudding, and submits a confirmation message back to the producer by putting it in the outbound queue ❸. The producer finally receives the confirmation message ❹.

16.5.4 Truncated exponential backoff

It's quite common for a worker role to have an infinite loop that polls the queue, processes the work, and pauses for a period of time if the queue is empty. The following listing shows the typical infinite polling loop.

Listing 16.5 The typical infinite loop to poll a queue

```
while (true)
{
   CloudQueueMessage msg = queue.GetMessage();      <-- ❶ Gets message
   if (msg != null)
   {
      string messageContent = msg.AsString;
      DoWork(messageContent);
      queue.DeleteMessage(msg);      <-- ❷ Does work, deletes message
   }
   else
   {
      System.Threading.Thread.Sleep(1000);      <-- ❸ Sleeps if no messages
   }
}
```

In this listing, a permanent `true` condition starts the loop off, looping until `true` equals `false`, which shouldn't ever happen (and if it does, we're all in for a world of hurt). We grab the first message off of the queue and check to see if it's `null`. If there aren't any messages on the queue, the `GetMessage` ❶ would return a `NULL` message. If there is a message, we process it, not deleting it until the real work is fully completed ❷.

If there wasn't a new message retrieved, the loop sleeps for a period of time ❸, and then tries again.

Sometimes you might find that a queue is polled too often. If this is a concern, you can dynamically change the wait time in the bottom of the loop. A common algorithm used in networking is called *truncated exponential backoff*. You can see an example of how to implement this in listing 16.6. Under this system, each time a queue check doesn't return a message, the loop delay is extended exponentially until a certain ceiling is reached. If the check does return a message, the loop delay is decreased, either back to the lowest setting, or to the next lowest setting in the progression.

Listing 16.6 Adjusting the delay in a polling loop

```
while (true)
{
      CloudQueueMessage msg = q.GetMessage();
      if (msg != null)
      {
```

```
            q.DeleteMessage(msg);

        if (useGradualDecrease)
            if (currentInterval > intervalFloor)
                currentInterval =
                  ➥ currentInterval / 2;
            else
                currentInterval = intervalFloor;          ❶ Decreases sleep interval
        else
            currentInterval = intervalFloor;

    }
    else
    {
        if (currentInterval < intervalCeiling)
        currentInterval = currentInterval * 2;            ❷ Increases sleep interval

        Thread.Sleep(currentInterval * 1000);
    }
}
```

For example, suppose the start value for the delay loop was 2 seconds. After an empty poll, the length would be doubled to 4 seconds. Successive empty checks would result in the time delay increasing to 8, 16, 32, 64, 128, 256, and 512 seconds. You can see the code that produces this progression at ❷. Figure 16.9 shows how the delay is increasing in this example as the queue remains empty.

At 512 seconds, the counter reaches the maximum set for the system, so it doesn't rise above 512 seconds. After some time a message appears. The system processes the message, and then checks the length of the queue. Because there was activity on the queue, the delay setting is set to the next step down the ladder, to 256 seconds at line ❶. The code also supports an immediate drop to the floor interval if you want an aggressive increase in the rate of polling.

Figure 16.9 shows the backoff polling in action. The sample code will randomly put messages in the monitored queue—there's a one-in-three chance of it doing so. The interval started at 2 seconds, and was immediately bumped to 4. Then a message was processed, so it was dialed back down to 2. Then there was a succession of empty calls, leading the code to rapidly increase the polling interval all the way up to 16 seconds between checks.

```
Output
Show output from: Debug
'QueueBrowser.vshost.exe' (Managed (v4.0.21006)): Loaded 'C:\Wi
'QueueBrowser.vshost.exe' (Managed (v4.0.21006)): Loaded 'C:\Wi
QueueBrowser.vshost.exe Information: 0 : Interval '4'
Chance: '1 of 3'
QueueBrowser.vshost.exe Information: 0 : Dequeued '*****'
QueueBrowser.vshost.exe Information: 0 : Deleted '*****'
QueueBrowser.vshost.exe Information: 0 : Interval '2'
Chance: '2 of 3'
QueueBrowser.vshost.exe Information: 0 : Interval '4'
Chance: '2 of 3'
QueueBrowser.vshost.exe Information: 0 : Interval '8'
Chance: '2 of 3'
QueueBrowser.vshost.exe Information: 0 : Interval '16'
```

Figure 16.9 The output from running the truncated exponential backoff polling algorithm. You can see the polling interval exponentially increase from 2 to 4 to 8 to 16 as the queue continues to be empty.

16.5.5 Queue creation on startup

One advantage of the Azure storage system is the easy creation and deletion of queues. A common trend is to inspect the storage system on system startup to determine if the needed entities (BLOBs, tables, and queues) exist. If they don't exist, they're created on the spot. This makes it easy to deploy a system, knowing that it will self-provision the storage resources it needs during startup.

A possible drawback to this approach is forgetting to manage the state of these resources carefully. If you're rolling out an upgrade to the system, and the initialization steps clear out the work queues and other storage entities, it's possible that you could lose valuable data or work in progress. Make sure that you test both new deployments and upgrades to your system in a safe environment before relying on the self-provisioning code. The automatic provisioning may also impact performance on system startup, because it will be busy checking the infrastructure and configuring things as needed.

16.5.6 Dynamic queues versus static queues

Most queues in your applications will be static in nature. The design of your system will require whatever queues it needs, and these will be provisioned when the application is deployed and left to run as is.

Alternatively, you can create queues dynamically. For example, you can create a new custom queue for each new order that's being processed by the system. This helps your application dynamically scale, and it also helps you separate different concerns in your system.

Perhaps you're building a system for a value-added network that manages the flow of purchase order messages from vendors and suppliers. All day long you're signing up new vendors and suppliers, and some occasionally stop using your service. A great way to automate the provisioning of the queues that are needed for each customer that signs up is to dynamically create the queues and infrastructure as they sign up and are approved as users of the system.

You want to pay careful attention to the state of each customer, and make sure that any leftover data is cleaned up, and entities are deprovisioned as customers stop using your service. You don't want to have to pay for unneeded infrastructure that's forgotten and left lying around.

Another scenario would be even more short term than the preceding one. Think back to the image-processing scenario. Because each image needs a queue to manage its breakdown and processing, you could dynamically create a queue for each new image job that's submitted. When the processing is complete, the queue could be torn down.

16.5.7 Ordered delivery

Some scenarios require guaranteed ordered delivery, such as some EDI scenarios, but the queue, as it is, doesn't support ordered delivery. There are several approaches for adding this capability on top of the normal queue service.

The simplest approach hinges on the series of messages having a header message that declares the length of the series, or the existence of a trailer message that tells the system when the last message has been received.

The basic approach would be for a process to monitor the normal queue, pull the messages off, and store them in a temporary Azure table. The table should have an integer property that stores the order of that message in the series. Once the defined number of messages are received, or when the trailer message is received, the messages can be properly ordered (usually by some element present in the message themselves) and then sent on to the final system that needed the messages ordered properly.

An optimization would be to have the process that's ordering the messages start trickling them on to the final destination when it knows that it has some of the messages already in order. For example, if messages arrive in the order 1, 2, 3, 5, 6, 4, the message collator could almost immediately send messages 1, 2, and 3. The collator would have to wait for message 4 to arrive before it could forward messages 4, 5, and 6.

16.5.8 Long queues

Most of the queues we've discussed so far have been in the form of an immediately serviced queue, where there are one or more consumers actively processing the messages in the queue.

There are times when it's important to have a long queue in play. This might be a queue that receives messages all day long, without an active consumer. The messages would be processed in a batch later that evening, when the consumer comes online. You might have an application in the field that sends messages into the cloud during the day; then, in the evening, a backend system comes online, processes all of the messages, and goes back offline. This would be a useful scenario when the consuming system isn't always available to process the messages you're holding in the queue.

16.5.9 Dynamically scaling to meet queue demand

The promise of queues is that the processing of messages is decoupled from the production of those messages. By decoupling the backend, you're free to scale the backend to meet the demands of the number of messages in the queue.

In a traditional environment, the number of producers is fairly static. The pool of consumers can be scaled, but it requires all the work of buying an additional server, provisioning it, and deploying it to the pool. This can be time consuming, and you're likely to miss the spike in demand while you're waiting for hardware to be shipped from your vendor.

The promise of cloud computing is the true dynamic allocation of resources to your computing needs. A management tool can be deployed that monitors the length of the queue in question. The tool can then dynamically create additional consumers (by increasing the number of deployed worker role instances using the service management API) based on the length of the queue. The management tool should define

a cap on the number of instances that can be created, and also rules as to when instances should be created or destroyed.

For example, you might define the minimum number of instances as zero, with a maximum of five instances. The rule of thumb would be one instance per 20 messages in the queue. The management tool would need to determine the length of the queue on a regular basis, perhaps every 3 minutes. Your rules should always allow for a little reserve buffer capacity. If you run too close to actual demand, the slight delay it takes (a few minutes) to bring on new instances could have a deleterious effect on the performance of your system.

Because Azure is billed based on the number of active instances, and not the actual use of the CPU, this can be a way to not only meet spikes in demand with grace, but also to minimize the costs of the solution.

16.6 Summary

Azure queues are a great way to break your system into pieces that still work together to get the work done, and they're easy to work with. They don't have to connect Azure web and worker roles together. They can be used to help cloud applications, mobile applications, and enterprise applications communicate together. Instead of a mobile application needing to punch through a firewall to submit a new repair ticket, it can submit the ticket into the cloud, where it can be picked up later by the on-premises system.

Queues are often the only on-ramp to a backend system. In this role, they act as the service endpoint for the capability the backend system represents. We showed how simple it is to create and manage queues. They provide a durable way of passing messages, and they're high performance as well.

Although queues are pivotal in leveraging the dynamic power of Azure with their ability to act as load balancers for worker role instances, the real power is in the messages in the system. Messages are the lifeblood of a decoupled system, allowing different components and subsystems to work together without being required to understand dirty implementation details of external parts.

The key to the power and reliability of queues lies in the message lifecycle and how the visibility timeout is managed. This timeout provides a recovery mechanism in case a message is lost in a failed server.

We also explored several patterns that can be used in the design of your application. The two most common manage the polling of the queue with a backoff polling algorithm, and dynamically provision the proper number of consumers based on the depth of the queue.

We'll next explore how to connect all of your on-premises and cloud services together using the Windows Azure Platform AppFabric Service Bus, and how to secure your services with the ACS service. These will allow you to connect to anything anywhere, and to keep it all secure.

17

Connecting in the cloud with AppFabric

This chapter covers

- Securing your services with ACS
- Introducing the Service Bus
- Connecting to your service from anywhere

The Windows Azure platform AppFabric (hereafter referred to as AppFabric) is an important piece of the Windows Azure puzzle. It's part of the larger Azure ecosystem and provides some fundamental features for working with hybrid applications. It performs two major functions: securing REST services and connecting them together.

AppFabric is a big topic—one that deserves its own book. Whenever you start talking about security, you get into long conversations. The same goes for service buses. That's because both of these topics involve a lot of terminology that the average developer is likely not familiar with.

Our goal in this chapter is to give you enough of a look at AppFabric to understand the core scenarios it can be used for, and to understand enough to confidently dive into a detailed book on your own. We'll be visiting the two key services

(Access Control Service and Service Bus) with a simple and straightforward example. You'll need to know a little about Windows Communication Foundation (WCF), but don't worry—WCF isn't scary.

17.1 The road AppFabric has traveled

AppFabric is arguably the most mature part of Windows Azure, at least if you measure by how long it has been publicly available, if not broadly announced. AppFabric started life as BizTalk Services. It was seen as a complementary cloud offering to BizTalk Server. BizTalk is a high-end enterprise-grade messaging and integration platform, and indeed the services fit into that portfolio well. Some joke that it was called BizTalk Services as a clever way to keep it a secret, because BizTalk is one of the most underestimated products Microsoft has. Just ask a BizTalk developer.

When Windows Azure was announced at PDC 2008, the BizTalk Services were renamed to *.NET Services*. Over the following year, there was a push to get developers to work with the services and put the SDK through its paces. Out of that year of real-world testing came a lot of changes.

When Windows Azure went live in early 2010, the services were renamed again to Windows Azure platform AppFabric to tie it more closely to the Windows Azure platform. Some people were confused by the older .NET Services name, thinking it was just the runtime and base class library running in the cloud, which makes no sense whatsoever.

17.1.1 The two AppFabrics

Don't confuse the AppFabric we'll be covering in this chapter with the new Windows Server AppFabric product. They're currently related by name alone. Over time they'll merge to become the same product, but they aren't there quite yet.

Windows Server AppFabric is essentially an extension to Windows Activation Service (WAS) and IIS that makes it easier to host WCF and Windows Workflow Foundation (WF)-based services in your own data center. It supplies tooling and simple infrastructure to provide a base-level messaging infrastructure. It doesn't supply a local instance of the Access Control Service (ACS) or Service Bus service at this time. Likewise, Windows Azure platform AppFabric doesn't provide any of the features that Windows Server AppFabric does, at least today. In early CTPs of Windows Azure platform AppFabric, there was the ability to host WF workflows in the cloud, but this was removed as it moved toward a production release.

The AppFabric we're going to cover in this chapter makes two services available to you: Access Control Service and the Service Bus.

17.1.2 Two key AppFabric services

AppFabric is a library of services that focus on helping you run your services in the cloud and connect them to the rest of the world.

Not everything can run in the cloud, as we've discussed several times already in this book. For example, you could have software running on devices out in the field,

a client-side rich application that runs on your customer's computers, or software that works with credit card information and can't be stored off-premises.

The two services in AppFabric are geared to help with these scenarios.

- *Access Control Service (ACS)*—This service provides a way to easily provide claims-based access control for REST services. This means that it abstracts away authentication and the role-based minutia of building an authorization system. Several of Azure's parts use ACS for their access control, including the Service Bus service in AppFabric.
- *Service Bus*—This service provides a bus in the cloud, allowing you to connect your services and clients together so they can be loosely coupled. A bus is simply a way to connect services together and route messages around. An advantage of the Service Bus is that you can connect it to anything, anywhere, without having to figure out the technology and magic that goes into making that possible.

As we look at each of these services, we'll cover some basic examples. All of these examples rely on WCF. The samples will run as normal local applications, not as Azure applications. We did it this way to show you how these services can work outside of the cloud, but also to make the examples easier to use.

Each example has two pieces that need to run: a client and a service. You can run both simultaneously when you press F5 in Visual Studio by changing the startup projects in the solution configuration.

17.2 Controlling access with ACS

Managing identity, authentication, and authorization is hard. It takes a lot of work by developers to get it right. One wrong step and you leave a gaping hole that a bad guy can take advantage of and land your company on the front page of the newspaper. Security is always a high priority on any project, but it's loaded with special terms and more complexity than developers generally want to deal with.

We're going to cover how you can integrate the user's identity inside your company with applications running in the cloud. We'll do this by leveraging claims-based authorization, which allows you to federate your internal identities with applications in the cloud by using standards-based tokens.

17.2.1 Identity in the cloud

When you move all of these concerns out of your own network and into the cloud, these concerns become even bigger issues. The application is no longer sitting right next to the source of authentication; the identity boundaries have been broken. How do you fix this problem?

The short answer is to build yet another identity store (a place to store usernames and passwords) and give every user yet another username and password to remember. They won't remember them—they'll either write it down on a sticky note on their monitor, or they'll call you once a month to remind them what it is. This is a bad experience for the user, and it's dangerous for the owner of the application.

We call this the *identity fishbowl*. Your identity (who you are and what you can do) is fairly easily maintained on your network, but the second you go off-premises you leave your fishbowl. The application in the cloud doesn't have any way to connect to your Active Directory to authenticate you. The way to bridge these worlds is with open standards and the concept of a federated identity.

There are some other challenges, as well. Let's go beyond the previous scenario where you have an internal user trying to access your application in the cloud. What if the user doesn't work for you, but is a customer or vendor of some sort. What if you need to provision 100 user accounts for that new customer, or 10,000 accounts? This is a lot of work for the administrator, it gives your end user yet another identity to remember, and it exposes you to a risk of not deleting an account when it should be. You're on the hook for managing those accounts, not the customer or vendor. If you have user accounts in your identity store that are active and belong to someone who has left the company, you're leaving a wide open hole for them (or an outside evildoer) to compromise that account and access your application when they shouldn't.

What you want is an easy and secure way for your internal users and external users (customers, vendors, and so on) to be able to access your application using the identities they already have. This approach also has to have low impact on the service code. You don't want to have to fix the code every time you enroll a new customer, or find a new protocol to support. You want to write applications, not become enterprise security ninjas. Well, most of us do anyway.

ACS handles all of these concerns for us in a brilliantly elegant way. Before we can really talk about ACS, though, we need a common understanding of some of the core concepts ACS is built on.

17.2.2 Working with actors

The security field is loaded with special vocabulary and concepts that scare most developers. Even knowing the importance of security, many developers just find some sample code and paste it in. They either don't have the understanding needed to work with the code, or they won't change it for fear of making a mistake.

There are several actors in this security play that we need to define. The most important is your service or application—the resource you're trying to protect. It's commonly called the *protected resource* or the *relying party* because it's relying on the security infrastructure.

The next actor is also easily defined: the *client*. This is the application that's trying to access the relying party. Clients are also sometimes called *issuers*. This side can get a little complicated when you have a second client that's being delegated through the first client to the protected resource.

Finally we have the ACS service itself. In security lingo, ACS is called the *authorization server* or *trusted authority*. It provides the security infrastructure the client must use to authenticate and use the protected resource.

In our play, the authorization server is the director, the protected resource is the lead actor, and the client is the supporting actor.

17.2.3 Tokens communicate authorization

Our three actors need some way to communicate, and they do this by passing tokens around. There are many formats for tokens, and there are many ways to pass them around. The messages they pass around usually include a set of claims, which we'll get to in a moment.

You likely use a token every day—your driver's license. You can use it to prove your identity or to prove a claim. A bar might demand you satisfy (prove) your claim that you're over 21. You never have to give them your birth certificate; you just give them your driver's license. You proved who you were, and when you were born, when you applied for your license. The Department of Motor Vehicles validated your credentials and provided you with a token that proves your claim of age on the license. The department is the trusted source, the bar with the beer is the protected resource, and your license is the token. You're the client.

The following figure shows this relationship between you, the bar, and the Department of Motor Vehicles. You first authenticate to the DMV. Once they're satisfied, they give you a secure token that provides data for some claims. You can then use this token to get a frosty beverage at any bar, even if not everybody knows your name. You can see this relationship in figure 17.1.

Some token formats are proprietary and some are an open standard. Microsoft has worked with Google and Yahoo! to develop a new, simple open standard that can be used across the cloud and the web. This standard is called OAuth, short for *open authorization*. There are other open standards that are popular, such as SAML (Security Assertion Markup Language), but they tend to be more complex and geared for SOAP instead of REST. OAuth's goal is to provide a simple token and protocol that makes it easy to secure REST-based services.

> **NOTE** You can read the OAuth specification at http://groups.google.com/group/oauth-wrap-wg.

ACS is based on OAuth, but it knows how to read other types of tokens as well. There are two halves to OAuth. The first is the Web Resource Authorization Protocol (WRAP). This is the protocol used to make authorization requests and to move

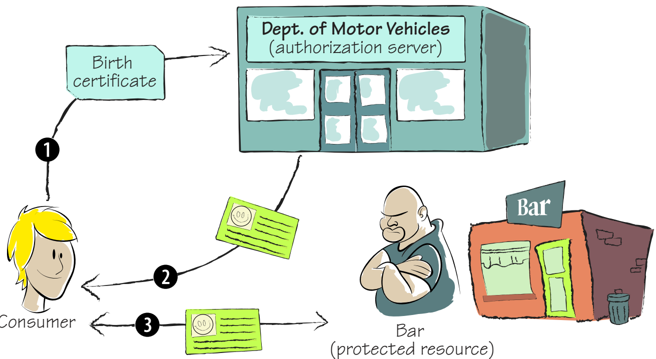


Figure 17.1 A bar doesn't require that you prove your birth date; instead you provide them with a valid and secure token from a trusted authority. You had to prove your birth date to the authority 1, and they gave you a token 2. You then use that token to get into the bar 3.

requests and tokens around on the wire. The other half of OAuth is the Simple Web Token (SWT), which defines the token itself. SWT tokens are simple to read and understand. The goal in designing OAuth and SWT was to build protocols that any platform could leverage to secure REST-based services.

All tokens coming from ACS will be SWT tokens. Here's a sample token:

```
CustomerId%3d31415%26Issuer%3dhttps%253a%252f%252fstringreversalinc.
➥ accesscontrol.windows.net%252f%26Audience%3dhttp%253a%252f%252
➥ flocalhost%252fprocessstring%26ExpiresOn%3d1266231958%26HMACSHA256%
➥ 3dI5g66yaiECux9IQ8y7Ffm2S1p%252bAXF73HWfzSNPyPLOE%253d
```

Notice that this token is URL-encoded. Later on we'll look at code that will let us shred this token and understand the separate parts. If you've ever seen a SAML token, this is far simpler and easier to work with. There isn't even any XML!

The OAuth working group's website includes the specification for these protocols and formats. They're surprisingly easy to read, but they do lack an interesting plot. They're much easier to read than the WS-* and SAML/STS specifications, if that means anything. You can read them all in about 30 minutes.

17.2.4 Making claims about who you are

With all these tokens flying around, we need something to put in them. Although there are other pieces of data stored in tokens, the real reason tokens exist is to deliver what is called "a set of claims."

A *claim set* is a list of claims made about the client or user by the authorization server. What a claim represents is completely open and can be defined by the systems using it. The claim must serialize to plain text—there isn't any fancy XML in OAuth, just a name-value collection of claims. A claim set might include a user's name, their birth date, their customer level, a list of roles or groups they belong to, or anything else your service (the protected resource) needs.

Your service will use these claims to make security and behavior decisions. The first decision is whether the user is allowed to access your service. Then, once you have let them in, you can use the claims to determine what they can do. If their role claim includes *manager*, you might let them apply discounts to existing orders. If the role claim is *staff*, maybe they can only create normal orders. What you ask for in claims and what you do with them once you get them is up to you.

This use of claims moves us away from the traditional role-based access control (RBAC) and toward claims-based access control (CBAC). The concepts are the same; the difference is in how we get the data regarding the user's identity and how we make decisions based on that data.

As you implement ACS, it's possible to add the use of SWT tokens to your system without ripping out the old way of managing identity. This is useful if you're trying to transition to the new platform without breaking what already exists.

17.3 Example: A return to our string-reversing service

In chapter 15, we talked about our amazing string-reversal company, since named String Reversal Inc., and our service that used a new and innovative way to reverse strings. In the time it has taken you to read the intervening chapter, the company has grown and prospered. String-reversal user groups and industry conferences are springing up all over the world.

But our emerging company is running into some trouble. Every time we add a customer, we have to do a lot of work to provision that customer in the system. This overhead is getting in the way of our rapid expansion.

It has also led to a few problems. While you were in the middle of reading the chapter on queues, one of the company's customers, Maine Reversal, was forced to fire an employee, Newton Fernbottom, for insider string-reversing, a horrible, horrible crime. Because that customer didn't notify us that Newton was let go, we never disabled his account. Newton ran home and starting a competing firm, Downeast Reversing, using Maine Reversal's account with our company.

What we need to do now is provide a better way to authorize customers to use our service, and we want to minimize the amount of code we need to change. Because anything can be a client, and anything can be a protected resource with ACS, we're going to move our sample service to a normal local WCF service to make it easier to focus on how ACS works. Everything ACS does can easily be applied to both services and clients running in Azure.

17.3.1 Putting ACS in place

Your first step in upgrading the service is to support a simple scenario where customers will have a shared secret (similar to a username and password) to access the service. Whoever they give that secret to will be able to use the service. They'll be able to change the secret when they need to, just like changing your password every 30 days.

Your first step is to create an AppFabric namespace. This namespace is a lot like a container in BLOB storage—it holds the settings for how you're using the ACS service. You could have several namespaces if you wanted to, perhaps to isolate different services with different configurations.

To create the namespace, you'll use the Azure portal, shown in figure 17.2. Besides creating the namespace, the portal doesn't do much with regards to ACS. There are other tools for that.

To create a namespace, log in to the Azure portal and choose AppFabric on the left side. You'll then see a list of your existing namespaces and a button for creating a new one.

To create a new namespace, you simply need to provide a globally unique name for your namespace. In figure 17.2, you can see that we have selected `StringReversalInc` for our namespace. Once you click the Create button, AppFabric will provision its systems with your namespace.

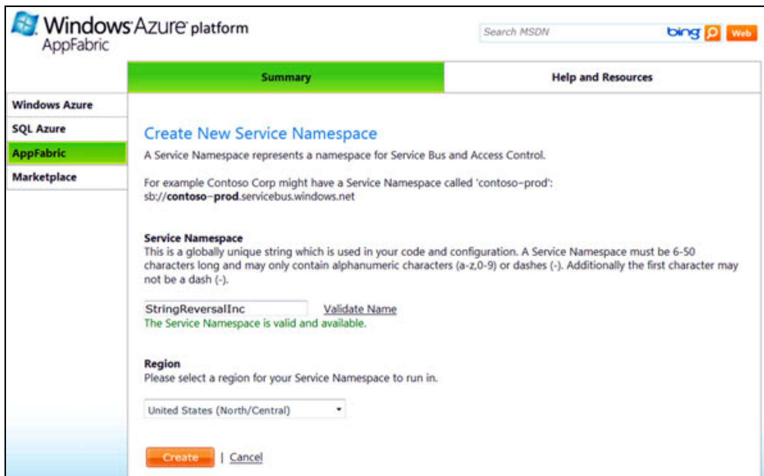


Figure 17.2 To start using AppFabric, you must first create a namespace. This acts like a container for the entire configuration of ACS and the Service Bus. The name of the namespace has to be globally unique.

As you can see in figure 17.3, ACS has configured both a Service Bus and an ACS service for your namespace. The service endpoints for both services will be displayed as shown in the figure. Notice that the namespace is the hostname of the service endpoints.

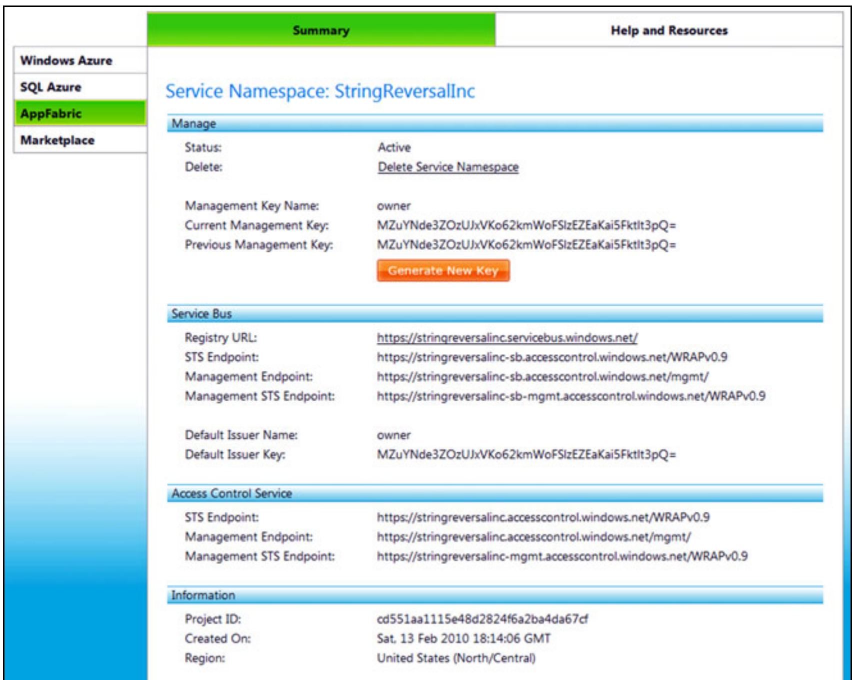


Figure 17.3 Once you create a namespace, AppFabric will provision that namespace with ACS, Service Bus, management endpoints, and security keys.

A management key will be created for you as well. This 32-byte symmetric key is what you'll use when accessing the AppFabric management service to perform operations on your namespace. We won't explore the management service in this chapter, but you should check it out. These keys should not be shared outside your organization, or published in a book where anyone can get ahold of them.

17.3.2 Reviewing the string-reversal service

For this chapter's purposes, we'll use a local REST version of the string-reversal service developed in chapter 15. You can find the complete code for this revised service in the sample code for this chapter. We've removed the entire worker role and Azure-related code to do this. ACS is about securing REST-based services, and our old service used a TCP-based binding. We've changed it to use REST by using the `WebServiceHost` and the `WebHttpBinding` classes.

The following listing shows how we're building our simple little service. This code will start up the service and wait for calls to the service.

Listing 17.1 A simple REST service

```
using System.ServiceModel;
using System.ServiceModel.Web;

public class svcProcessString
{
  public static void Main(string[] args)
  {
    Console.WriteLine("Starting string reversal servicehost...");

    WebServiceHost serviceHost = new
    ➥ WebServiceHost(typeof(ReverseStringTools));          <- Creates host to run service code

    WebHttpBinding binding = new WebHttpBinding(WebHttpSecurityMode.None);

    serviceHost.AddServiceEndpoint(typeof(IReverseString), binding, new
    ➥ Uri("http://localhost/processstring"));

    try
    {
      serviceHost.Open();                                  <- Starts service host with specified configuration
      Console.WriteLine("String reversal servicehost started.");
    }
    catch (Exception ex)
    {
      Console.WriteLine("Could not start string reverser servicehost. {0}",
      ➥ ex.Message);
    }

    Console.ReadLine();
  }
}
```

If you run this sample string-reversal service, you can make all of the requests to the service you want. The sample code includes a simple client that will call the service.

The next few steps are going to center around adding code to the service so that it can read and use SWT tokens. Once that's done, you can upgrade the client so it can fetch a token from ACS and use it during a request to the service.

17.3.3 Accepting tokens from ACS

You'll need to upgrade the service so it can receive and work with ACS tokens. This code is fairly trivial, and much of it is supplied in the AppFabric SDK, which you'll have to install in order to follow these next steps. You can find the SDK on the Azure portal. It also includes several tools that we'll look at in the next section.

Exactly how you get the token and where you process it might change, depending on your business situation and system architecture, but the steps will be generally the same.

The first step is to grab the token from the incoming message. The token will usually be included in the header as an authorization entry. In some situations, it can also be in the URL or in the body of the message, depending on the capabilities of the client.

Exactly how you grab this header will differ based on how you're receiving the message. In WCF it's best to do this in a custom `ServiceAuthorizationManager` class that's added to the WCF pipeline when you set up the channel. Every message will flow through this class, and there you can make a decision about whether to let it through or deny it access.

In a normal WCF service, you need to use the `WebOperationContext` to retrieve the header from the request:

```
string authorizationHeader =
➥ WebOperationContext.Current.IncomingRequest.Headers
➥ [HttpRequestHeader.Authorization];
```

This code will get the raw header. You now need to do a few things to make sure this token is valid, and then you can use it to make decisions.

The SDK has all the sample code you need to build a class called `TokenValidator`. We've included a refactored version of this class in the chapter's sample code that's a little easier to use. The validator will do a series of checks for you, and if they all pass, it'll return `true`. If the validation fails, the validator will deny access.

```
validator = new
➥ ACSTokenValidator("dqSsz5enDOFjUvUnrUe5p1ozEkp1ccAfUFyuYpawGW0=",
➥ "StringReversalInc", "http://localhost/stringservice");

if (!validator.ValidateAuthorizationHeader(authorizationHeader))
  DenyAccess();
```

To initialize the validator, you need to pass in three pieces of information:

- The signing key
- The ACS namespace to check against
- The URL of the service the message was sent to

You're passing in the key, the namespace you set up, called `StringReversalInc`, and the URL of the service you're protecting, http://localhost/stringservice.

You then call the `ValidateAuthorizationHeader` on the header you pulled off the message. If this returns `false`, you'll deny access by calling a simple little method, `DenyAccess`, that sets up the deny message:

```
private static void DenyAccess()
{
  WebOperationContext.Current.OutgoingResponse.StatusCode =
  ➥ HttpStatusCode.Unauthorized;
          WebOperationContext.Current.OutgoingRequest.Headers.Add("WWW-
          ➥ Authenticate", "WRAP");
  }
```

That's all you need to receive the header. Most of the work involves making sure it's a valid header and something you can trust. This is the same job the bouncer at the bar does, when he looks at your driver's license to make sure it's a real license and hasn't been tampered with.

17.3.4 Checking the token

We've put all of the token-checking logic into the `ACSTokenValidator` class, and we've just discussed how to new up a validator. The validator includes some custom methods, namely `Validate` and `IsHMACValid`. When you pass in the header, the validator will verify several aspects of it to make sure it's valid. All of these checks test for the negative; if the test passes, you have a bad token and the validator returns `false`.

Table 17.1 summarizes the checks that we do in the code.

Table 17.1 Validation checks performed on a token

Check to be made	Purpose
`string.IsNullOrEmpty(authHeader)`	Makes sure you received a header.
`!authHeader.StartsWith("WRAP ")`	Ensures the header starts with `WRAP`.
`nameValuePair[0] != "access_token"`	Checks that there are two pieces to the header, and that the first is equal to `access_token`.
`!nameValuePair[1].StartsWith("\"") || !nameValuePair[1].EndsWith("\""))`	Checks that the second piece starts and ends with a slash.
`!Validate(GetTokenFromHeader(authHeader))`	Grabs the token part of the header and makes sure it's valid.
`IsHMACValid(token, signingKey)`	Makes sure the token has been signed properly. If this is correct, you know who sent it.
`this.IsExpired(token)`	Checks that the token hasn't expired. Tokens are vulnerable to replay attacks, so this is important.

Table 17.1 Validation checks performed on a token *(continued)*

Check to be made	Purpose
`this.IsIssuerTrusted(token)`	Ensures the sender is recognized as a trusted source. We'll cover this shortly.
`this.IsAudienceTrusted(token)`	Checks that the audience is the intended destination.

If the header passes all of these checks, you know you have a secure token from a trusted source, and that it's meant for you. This is the minimum you'll want to do to allow the message through to the service. You may also want to crack open the claim set in the token to look at what claims have been sent, and make decisions on those claims. In our example, we've mapped in some claims. One is the customer ID number, and the other is the customer's service level. This might be used to determine how long the strings they submit to our service can be. They might have to pay more to reverse longer strings.

That's all you have to do to enable the service and consume and use ACS tokens for authorization. Next we'll look at how you can configure a client to add the authorization header to their requests.

17.3.5 Sending a token as a client

In this section, you're going to build a simple command-line client. This will make it easier to focus on the service aspects of the code. Feel free to make it sexy by using Silverlight or WPF.

For this client, we'll share the contract by sharing a project reference. (Many projects do this, especially when you own both ends of the conversation.) You should either share the contract through a project reference, or through a shared assembly.

In this case, our biggest customer, Maine Reversal, will be building this client. We've set up a trusted relationship with them by swapping keys and configuring them in ACS—we'll look at how to do this in the next section. Maine Reversal won't be sending in any custom claims of their own, just their issuer identity. This process essentially gives them a secure username and password.

We've created a helpful utility class called `ACSTokenValidator` (found in the sample code for this chapter) that encapsulates the process of fetching an ACS header from the AppFabric service. Again, this code is mostly from the SDK samples with some tweaks we wanted to make. (Why write new code when they give us code that works?)

To call the `GetTokenFromACS` method, you'll pass in the service namespace (`StringReversalInc`), the issuer name (the client's name, which is `MaineReversal` in this case), the signing key that Maine Reversal uses, and the URL that represents your protected resource. This doesn't have to be the real URI of the intended destination, but

in many cases will be. In security parlance this is referred to as the *audience*. The method call looks like this:

```
string Token = GetTokenFromACS("StringReversalInc",
                "MaineReversal",
                "ltSsoI5l+8DzLSmvsVOhOmflAsKHBYrGeCR8KtCI1eE=",
                "http://localhost/processstring");
```

The `GetTokenFromACS` method performs all the work. It uses the `WebClient` class to create the request to ACS. If everything goes well, the ACS service will respond with a token you can put in your authorization header on your request to the string-reversal service.

The following listing shows how you can request a token from the ACS service.

Listing 17.2 How a client gets a token from ACS

```
private static string GetTokenFromACS(string serviceNamespace, string
➥ issuerName, string issuerKey, string scope)
{
  WebClient client = new WebClient();
  client.BaseAddress = string.Format("https://{0}
  ➥ .accesscontrol.windows.net", serviceNamespace);          ❶ Specifies ACS service address

  NameValueCollection values =                    ❷ Sends authorization data
  ➥ new NameValueCollection();
  values.Add("wrap_name", issuerName);
  values.Add("wrap_password", issuerKey);
  values.Add("wrap_scope", scope);

  byte[] responseBytes = client.UploadValues("WRAPv0.9", "POST", values);

  string response = Encoding.UTF8.GetString(responseBytes);    ❸ Gives token to caller

  return response
         .Split('&')
         .Single(value => value.StartsWith("wrap_access_token=",
         ➥ StringComparison.OrdinalIgnoreCase))
         .Split('=')[1];
}
```

You have to provide the `GetTokenFromACS` method with the base address for the request ❶. This is a combination of the ACS service address, `accesscontrol.windows.net`, and the namespace for the ACS account, `StringReversalInc`.

To make the call, you need to provide three pieces of data: the issuer name (your name in the ACS configuration), the signing key, and the namespace of the service you're trying to reach ❷.

At this point, ACS will check your credentials. The issuer name is basically your username, and the signing key is your password. If everything checks out, ACS will respond with a valid token that you can attach to your request to the service ❸.

17.3.6 Attaching the token

Attaching the token to the header of the request is fairly simple on most platforms. You can also put the token information in the URL or the message body. Doing either isn't as good as using an authorization header, so only do this if your system doesn't support an authorization header.

To add the token to the authorization header, you can add it to the `OutgoingRequest.Headers` collection:

```
string authorizationHeader =
➥ string.Format("WRAP access_token=\"{0}\"",
➥ httpUtility.UrlDecode(Token));
WebOperationContext.Current.OutgoingRequest.Headers
➥ .Add("authorization", authorizationHeader);
```

To attach the token to the header, you need to use the `UrlDecode` method to decode it, and then wrap it with the WRAP leading text. This tells the destination service that the token is a WRAP token. This text will be stripped off by the server once the token is validated. Then you add the header to the outgoing request using the `WebOperationContext` class.

That's all the client needs to do. Your client should be robust enough to handle any errors in the ACS service call or the ACS token request being denied.

In order for the token validation and generation to work, you have to set up some configuration in the ACS service: a trusted relationship with the issuer, and some rules.

17.3.7 Configuring the ACS namespace

The ACS needs to be configured for your service. You've already learned how to define a namespace, and the namespace is a container for the rest of the ACS configuration. You can also chain namespaces together, which is the key mechanism for providing simple delegation.

Each namespace has four components: issuers, scopes, rules, and token policies. These elements work together to help secure your REST service.

The AppFabric SDK provides two tools for configuring your service, both of which run locally and call into the management service: ACM.exe (used from the command line) and the Azure configuration browser. (You can use the management service as a third option, but that'll require more work on your part.) Beyond the tool that sets up the namespace, there aren't any management tools on the ACS portal.

The ACM.exe tool can be found in the tools folder where you installed the AppFabric SDK. ACM is most useful when you're automating a process or want to script the configuration. But keep in mind that calls to the AppFabric management endpoint aren't free, like the Windows Azure management endpoints are.

The Azure configuration browser is shipped with the SDK, but as a sample in source-code form in a separate download file. You need to load the solution and compile it to use it. This distribution approach is really useful because you can then extend the tool to meet your needs, and the browser is a lot easier to use than the command-line tool.

The configuration browser does have a few limitations. First, it's really ugly, but that's OK. The second is that, at this time, it can't update an existing cloud configuration; it can only deploy a complete configuration. This means that any time you make a change, you have to delete the configuration in the cloud and completely redeploy the new configuration. An advantage of this approach is that you can store your configuration locally in an XML file, which aids in backup and configuration management.

You'll need to provide your service name and your management key with either tool. For the ACM.exe application, you can put your settings in the app.config file, which saves you from having to type them in as part of your commands every single time.

ISSUERS

Issuers are actors in the ACS system and represent consumers of your service. When you create an issuer, you need to provide both a display name and an official issuer name. Once the issuer is created, a signing key will be generated for you. This is the key the issuer must sign their requests with when they ask the ACS service for a token.

To create an issuer from a command line, you would use the following command:

```
acm create issuer
         -name:MaineReversal
         -issuername:MaineReversal
         -autogeneratekey
```

In the configuration browser you'll need to right-click on the Issuers node and choose Create. Figure 17.4 shows how to set up your first client, Maine Reversal.

Setting up an issuer in the system is akin to creating a user. The issuer needs a name (comparable to a username) and a signing key (which is like a password). These are the credentials the issuer will use to request an ACS token.

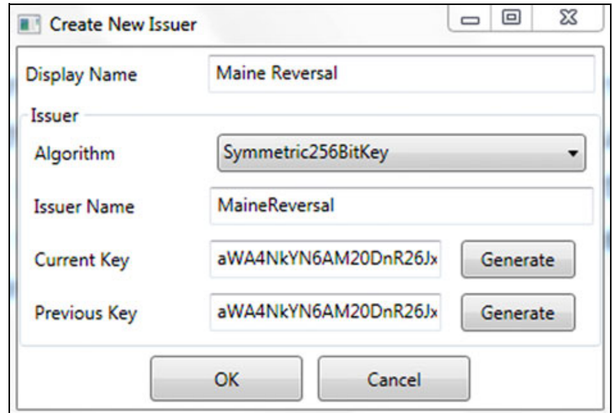


Figure 17.4 Creating an issuer is easy with the ACS configuration browser. You'll need to provide both a display name and an official name for the issuer. You can use the tool to automatically create the signing keys.

TOKEN POLICY

A token policy defines how you want your tokens to be created. Because token-based systems can be vulnerable to token-replay attacks, you'll first want to set a lifetime timeout for the token. This is expressed in seconds. When the token is created, it'll be stamped with the time when the token will expire. Once it's expired, the token will be considered invalid and won't be accepted by a service. You have to check for this expiration explicitly when you validate the token. We check for this in the sample code for the chapter, as seen in the `ACSTokenValidator` class.

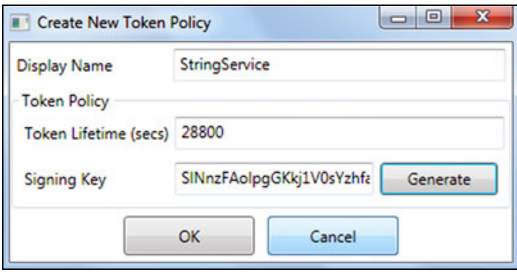


Figure 17.5 You'll need to create a token policy. This will determine the lifetime of your tokens, and the key that will be used to sign your ACS tokens.

The command for creating a token policy at the command line is as follows:

```
acm create tokenpolicy -name:StringReversalInc -autogeneratekey
```

To create a token policy in the configuration browser, right-click on the Token Policy node and select Create. Figure 17.5 shows the Create New Token Policy dialog box, where you can create a policy for your string service.

The second piece of data you'll need for your token policy is the signing key, which can be generated for you. This is the key that will be used to sign the tokens generated for you by the ACS.

SCOPES

A scope is a container that's tied to a service URI. It brings together the rules that you want applied to your service, as well as the token policy you want used for that service.

To create a scope at the command-line level, you'll need the ID of the token policy you want to assign to the scope. You can get the `tokenpolicyid` from the output of the `create tokenpolicy` command discussed in the previous section. This is the command for creating a scope:

```
acm create scope -name:StringServiceScope
           -appliesto:http://localhost/processstring
           -tokenpolicyid:tp_4cb597317c2f42cba0407a91c2553324
```

When you're using the configuration browser, you won't need to provide the token policy ID—you'll be able to choose it from the drop-down list. You can associate a policy to a namespace by creating a scope, as shown in figure 17.6.

There are several advanced uses for scopes that we won't go into in this chapter. These include managing a large number of service endpoints with a few scopes, and chaining namespaces together for delegation.

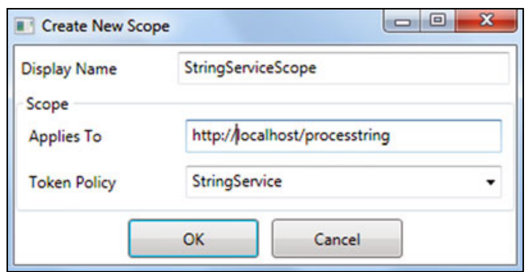


Figure 17.6 It's easy to create a scope. A scope acts as a container for a set of rules for your service. It also associates a token policy with the service. You'll need to define the URI for the service the scope applies to.

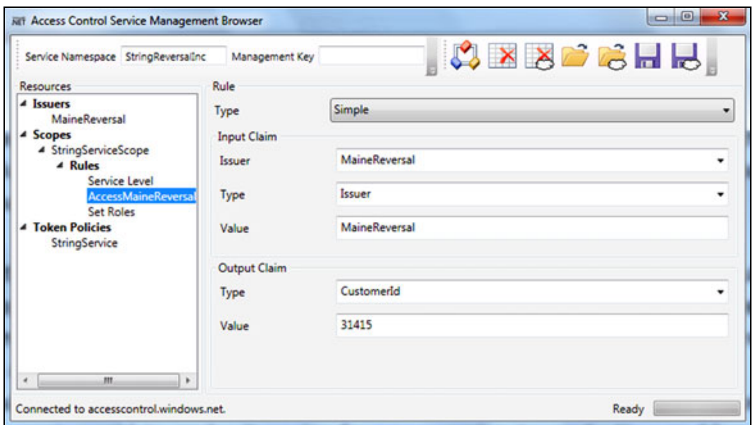


Figure 17.7 Creating a rule to insert a claim that includes the customer's `customerid`. In this case, you're relying on the issuer of the inbound request to know which customer it is.

RULES

Rules are the heart of the ACS system. When a token is created by ACS, the rules for the related scope are executed. This is the process that allows you to transform the consumer's request into something your application can act on. Rules can also be used to create claims out of thin air, and put them in the resulting token.

For example, suppose you wanted to place a claim in the token that represents the consumer's `customerid`, to make it easier for your service to identify the account the request is related to. You could create a rule that says, "If this is for issuer `MaineReversal`, add a claim called `customerid` with a value of `31415`." Figure 17.7 shows how you could create this rule.

Another rule you could use would assign a new role value based on mappings you've worked out with the customer. Perhaps their system tracks their users with a role of ServiceManager—this would be a group the user belongs to at Maine Reversal. Your system doesn't know that role name, and you don't want to add all of your customers' role types to your system—that would get very complex very quickly. The rule in figure 17.8 creates the roles claim with the manager value.

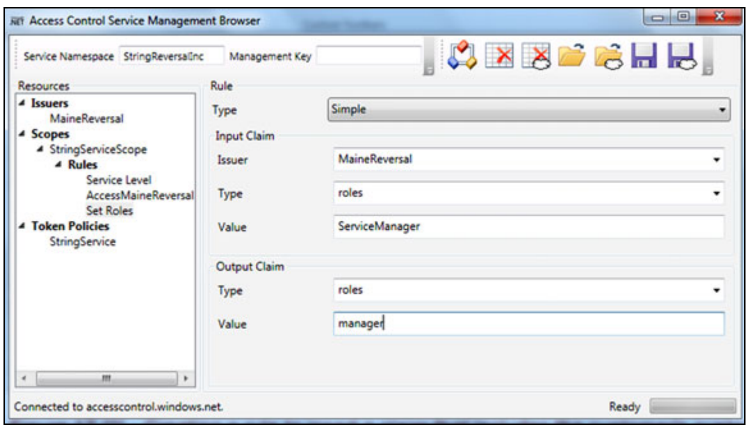


Figure 17.8 Creating a rule that substitutes the inbound roles claim for a new one. Using this rule, you can map the ServiceManager role value that your system doesn't know to one your system does know—manager.

You can then create a rule that finds a claim called `roles` with a value of `SalesManager`, and replaces it with a claim called `roles` that has a value of `manager`. In this way you've moved the customer configuration and mapping out of your service and into the authorization service where it belongs.

Creating a rule at the command line is a little more complex than using the configuration browser:

```
acm create rule -name:MaineReversalMap
➥ -scopeid:scp_e7875331c2b880607d5709493eb2751bb7e47044
➥ -inclaimissuerid:iss_6337bf129f06f607dac6a0f6be75a3c287b7c7fa
➥ -inclaimtype:roles -inclaimvalue:ServiceManager
➥ -outclaimtype:roles -outclaimvalue:manager
```

To find the IDs of the scope and issuer, you can use these commands: `acm getall scope` and `acm getall issuer`.

17.3.8 Putting it all together

You've come a long way in stopping illicit use of your service. Now you can control who uses it and how they use it. You've updated your service to consume tokens, you've updated the client to submit tokens with service requests, and you've prepared the ACS service with your configuration.

How does this all work? In this simple scenario, the client requests an access token from ACS, providing its secret key and issuer name. ACS validates this and creates a token, using the scope rules you set up to create claims in the new token. The client then attaches the token to the message in the authorization header.

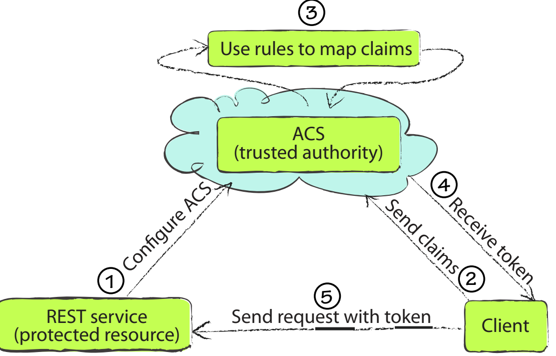


Figure 17.9 How the three actors work together to securely access a REST service. The service configures ACS; the client asks for a token; ACS creates a token, based on rules; and then the client submits this token with its service request.

Figure 17.9 should look familiar; it's much like the DMV example (see figure 17.1), but it shows the technical actors and how they interact.

When your service finally receives the message, you'll grab the token from the header and verify it. You want to make sure that it's valid and hasn't been tampered with. Once you trust it, you can make decisions on what to do.

In our example, you can take the `customerid` value and verify that they're still a paying customer, and if so, respond to their request. You can stop using the token at this point and respond like normal, or you can shred the token and use the claims throughout the application.

If you were protecting an ASP.NET website instead of a REST-based WCF service, you could take those claims, put them in a class that implements `IPrincipal`, and use the class to make role decisions throughout your code.

We've finished a quick lap around ACS. ACS's sibling is the Service Bus, which will let us connect anything to anywhere, with just a little bit of WCF and cloud magic.

17.4 Connecting with the Service Bus

The second major piece of Windows Azure platform AppFabric is the Service Bus. As adoption of service-oriented architecture (SOA) increases, developers are seeking better ways of connecting their services together. At the simplest level, the Service Bus does this for any service out there. It makes it easy for services to connect to each other and for consumers to connect to services.

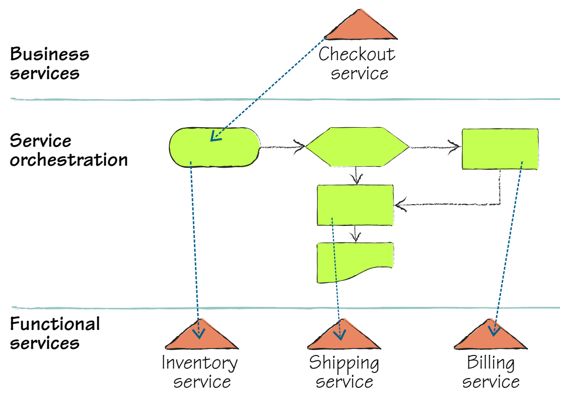
In this section, we're going to look into what the Service Bus is, why you'd use a bus, and, most importantly, how you can connect your services to it. You'll see how easy it is to use the Service Bus.

17.4.1 What is a Service Bus?

Enterprise service buses (ESBs) have been around for years, and they've grown out of the SOA movement. As services became popular, and as the population of services at companies increased, companies found it harder and harder to maintain the infrastructure. The services and clients became so tightly coupled that the infrastructure became very brittle. This was the exact problem services were created to avoid. ESBs evolved to help fix these problems.

ESBs have several common characteristics, all geared toward building a more dynamic and flexible service environment:

- *ESBs provide a service registry*—Developers and dynamic clients needed ways to find available services, and to retrieve the contract and usage information they needed to consume them.
- *ESBs provide a way to name services*—This involves creating a namespace around services so there isn't a conflict in the service names and the message types defined.
- *ESBs provide some infrastructure for security*—Generally, this includes a way to allow or deny people access to a service, and a way to specify what they're allowed to do on that service.
- *ESBs provide the "bus" part of ESB*—The bus provides a way for the messages to move around from client to service, and back. The important part of the bus is the instrumentation in the endpoints that allows IT to manage the endpoint. IT can track the SLA of the endpoint, performance, and faults on the service.
- *ESBs commonly provide service orchestration*—Orchestration is the concept of composing several services together into a bigger service that performs some business process.



A common model for ESBs is shown in figure 17.10. This is similar to the typical n-tier architecture model, where each tier relies on the abstractions provided by the layer below it.

The orchestration has become not only a way to have lower-level services work together, but it also provides a layer of indirection on top of those services. In the orchestration layer you can route messages based on content, policy, or even service version. This is important as you connect services together, and as they mature.

17.4.2 Why an ESB is a good idea in the cloud

The problem for ESBs is that they usually only connect internal services and internal clients together. It's hard to publish a service you don't control to your own bus. External dependencies end up getting wrapped in a service you own and published to your ESB as an internal service. Although this avoids the first problem of attaching external services to your ESB, it introduces a new problem, which is yet more code to manage and secure.

If you wanted to expose a service to several vendors, or if you wanted a field application to connect to an internal service, you'd have to resort to all sorts of firewall tricks. You'd have to open ports, provision DNS, and do many other things that give IT managers nightmares. Another challenge is the effort it takes to make sure that an outside application can always connect and use your service.

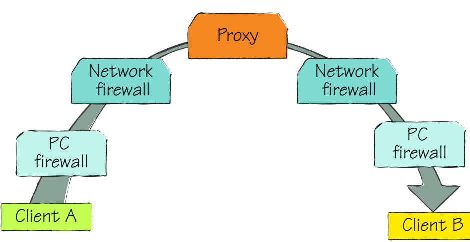


Figure 17.11 Modern networks provide a great number of barriers to easy point-to-point communication. Many computers these days have a local firewall, with one or more firewalls on their network. There are also proxies, NATs, and other devices in the way making it hard to connect in an old-fashioned, direct way.

To go one step farther, it's an even bigger challenge to connect two outside clients together. The problem comes down to the variety of firewalls, NATs, proxies, and other network shenanigans that make point-to-point communication difficult. For example, figure 17.11 might represent the layers between your local software and the services it's calling across the internet.

Take an instant messaging client, for example. When the client starts up, and the user logs in, the client creates an

outbound, bidirectional connection to the chat service somewhere. This is always allowed across the network (unless the firewall is configured to explicitly block that type of client), no matter where you are. An outbound connection, especially over port 80 (where HTTP lives) is rarely a problem. Inbound connections, on the other hand, are almost always a problem.

Both clients have these outbound connections, and they're used for signaling and commanding. If client A wants to chat with client B, a message is sent up to the service. The service uses the service registry to figure out where client B's inbound connection is in the server farm, and sends the request to chat down client B's link. If client B accepts the invitation to chat, a new connection is set up between the two clients with a predetermined rendezvous port. In this sense, the two clients are bouncing messages off a satellite in order to always connect, because a direct connection, especially an inbound one, wouldn't be possible. This strategy gets the traffic through a multitude of firewalls—on the PC, on the servers, on the network—on both sides of the conversation.

There is also NATing (network address translation) going on. A network will use private IP addresses internally (usually in the 10.x.x.x range), and will only translate those to an IP address that works on the internet if the traffic needs to go outside the network. It's quite common for all traffic coming from one company or office to have the same source IP address, even if there are hundreds of actual computers. The NAT device keeps a list of which internal addresses are communicating with the outside world. This list uses the TCP session ID (which is buried in each network message) to route inbound traffic back to the individual computer that asked for it.

The "bounce it off a satellite" approach bypasses this problem by having both clients dialing out to the service. Figure 17.12 illustrates how this works.

The Service Bus is here to give you all of that easy messaging goodness without all of the work. Imagine if Skype or Yahoo Messenger could just write a cool application that helped people communicate, instead of spending all of that hard work and time figuring out how to always connect with someone, no matter where they are.

The first step in connecting is knowing who you can connect with, and where they are. To determine this, you need to register your service on the Service Bus.

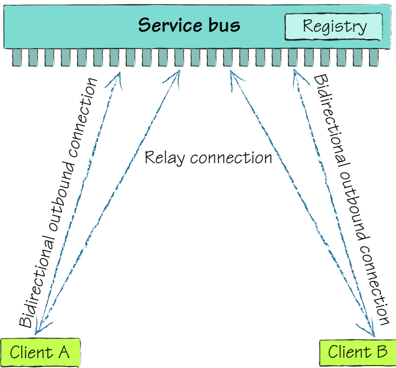


Figure 17.12 Any client can communicate with any other client (which may be a service) from anywhere, on any network, by using the relay bindings with the Service Bus. Each client first registers with the bus so it knows where they're connected. Then when client A wants to connect with client B, they can use the registry to find each other.

17.5 Example: Listening for messages on the bus

Previously in this chapter, as you were configuring ACS, you had to create a namespace. That namespace held configuration for the services you wanted to protect with ACS. Now you'll use that same namespace to register and manage your service on the bus.

Almost all of the magic of the Service Bus is embedded in the new WCF service bindings that are shipped in the AppFabric SDK. All of these bindings start with a protocol identifier of *sb* (you're probably familiar with *http* and others). The binding names also almost always have the word *relay* in them, to indicate that you'll be relaying messages through the Service Bus to their destination.

17.5.1 Connecting the service to the bus

Changing your application to listen for messages from the bus instead of the HTTP endpoint is easy. You need to change the binding and address information to point to the bus.

We've moved the configuration of the service from code to the app.config file to make these changes easier, and you can see these changes in the Service Bus sample code for this chapter. Two things still need to be set up. You need to configure the address and binding information, and you need to configure the service for authentication to the bus.

First, in the configuration, you need to change the address of the endpoint to your namespace on the bus. For this example, you should change it from http://localhost/processstring to sb://stringreversalinc.servicebus.windows.net/processstring. This change tells WCF to use the Service Bus relay bindings, and that the service you're publishing should be registered in the `stringreversalinc` namespace.

```
<endpoint address=
"sb://stringreversalinc.servicebus.windows.net/processstring"
behaviorConfiguration="sharedSecretClientCredentials"
binding="netTcpRelayBinding"
contract="StringReversalLibrary.Contract.IReverseString" />
```

Your service also has to authenticate to the Service Bus when it starts up and registers with the bus. You use ACS to do this. In this example, you can use the simple shared secret we used earlier in this chapter. This will attach the credentials (essentially a username and password) to your request to connect to the Service Bus:

```
<behavior name="sharedSecretClientCredentials">
   <transportClientEndpointBehavior credentialType="SharedSecret">
      <clientCredentials>
         <sharedSecret issuerName="owner"
         ➥ issuerSecret="MZuYNde3ZOzUJxVKo62kmWoFSlzEZEaKai5Fktlt3pQ=" />
      </clientCredentials>
   </transportClientEndpointBehavior>
</behavior>
```

This shared secret behavior that you attach to your service will authenticate to the bus with your issuer name and signing key automatically.

Once you make these changes, you can run the service and it'll start up, authenticate, and start listening for messages coming from the bus. You'll then need to perform similar actions on the client.

17.5.2 Connecting to the service

In the previous section, we looked at what you had to do to connect a service to the Service Bus. You had to change the bindings to point to the bus and update the address. You also had to add some authentication so that the bus knew you were allowed to use your namespace.

You now need to follow the same steps to change the app.config for the client. You need to change the client binding so it's sending messages to the bus. For this example, you can name your endpoint `SBRelayEndpoint`, with the same address the service used.

```
<client>
   <endpoint
      name="SBRelayEndpoint"
      address=
"sb://stringreversalinc.servicebus.windows.net/processstring"
      binding="netTcpRelayBinding"
      contract="StringReversalLibrary.Contract.IReverseString"
      behaviorConfiguration="sharedSecretClientCredentials"
   />
</client>
```

The client is going to have to authenticate to the Service Bus as well—you can configure it to use a shared secret. Use the Maine Reversal issuer from section 17.3.7 of this chapter. Keep in mind that there are two endpoints: one for the ACS service, and one for the Service Bus. They don't share issuers. You can configure the credentials by changing the behavior of the service in the app.config file:

```
<behavior name="sharedSecretClientCredentials">
   <transportClientEndpointBehavior credentialType="SharedSecret">
      <clientCredentials>
         <sharedSecret issuerName=" MaineReversal"
         ➥ issuerSecret=" 1tSsoI5l+8DzLSmvsVOhOmflAsKHBYrGeCR8KtCI1eE=" />
      </clientCredentials>
   </transportClientEndpointBehavior>
</behavior>
```

Now the client can call your service from anywhere and always be able to connect to it. This makes it easy to provision new customers. In the old, direct method, you had to reconfigure the firewalls to let the new customer through. Now you can point them to the Service Bus address and give them the credentials they'll need.

The binding we used in this example is based on TCP, which is one of the fastest bindings you can use in WCF. It adds the relay capabilities to allow us to message through the bus instead of directly. There are other bindings available that support using the relay.

Now that we've covered what AppFabric can do today, let's consider what its future might hold.

17.6 The future of AppFabric

We normally don't talk about futures in a book because it's entirely possible that priorities will shift and particular features will never ship. The team has been open about what they consider their next steps, but any of this could change, so don't base important strategy on unofficial announcements.

With that said, here are a couple of current goals:

- *Extending OAuth for web identity*—The ACS team wants to extend its use of OAuth and SWT to include common web identity platforms. These are identities on the web that people commonly have, instead of traditional enterprise identities from Active Directory. It will be possible for a user to choose to log into a site not just with an ACS-provisioned issuer ID, or a SAML token, but also with their Google, Yahoo!, and Live ID accounts. This is accomplished by ACS federating with those directories with the updated OAuth protocols. This will also include Facebook and any OpenID provider. This is exciting for people who are building consumer-centric applications.
- *Support for all WS-* protocols*—Right now you can only protect REST services with ACS, and the ACS team wants to extend support for SOAP-based services as well. Once they have support for WS-*, they'll have support for WS-Federation and WS-Trust, which makes it easier to federate enterprise identities. This change will also give them support for CardSpace, which is Windows' identity selector for claims-based authentication systems.

17.7 Summary

This chapter gave you a tour of the Windows Azure platform AppFabric. We looked at how it's a cousin of Windows Server AppFabric, and how maybe over time they'll merge.

The first component of AppFabric we looked at was ACS. ACS's primary concern is securing REST-based services, and it does this with the OAuth and SWT open standards. ACS makes it easy to federate in other token protocols, like SAML, and makes it easy to transform tokens from other parties into a form that your application can consume. You can configure ACS with either the ACM command-line application, or the ACS configuration browser. Both applications use the underlying management REST service, which you can also use directly if you want.

We looked at how a client first authenticates with ACS to prove who they are, and then ACS gives them a token to use to gain access to the service. This removes the concern of authentication from the application, which in turn simplifies the codebase, and makes it easier to adjust to new rules as they evolve.

The second component of AppFabric we looked at was the Service Bus. The Service Bus is a migration of a common enterprise service bus into the cloud. The Service Bus's goal is to make it easy for consumers and services to communicate with each other, no matter where they are or how they're connected.

We adjusted our example to use a `netTCP` binding, but also to relay the messages through the cloud to bypass any firewalls or proxies that might be in the way. You had to make a few adjustments on the service and the client to make this possible. Both sides have to authenticate to ACS before they connect to the bus—this is how you secure your service when it's connected to the bus.

Although this was a simple chapter designed to help you get a feel for AppFabric, we hope that you're comfortable enough with the basics to know when it might make sense to leverage AppFabric. You should be prepared enough to explore on your own and maybe dive deep with a dedicated book on the topic.

The next chapter will zoom out from all of this detail and help you use the diagnostics manager to understand what your applications are doing. The diagnostics manager will help you determine what's happening, and the service management API will help you do something about it.

18 Running a healthy service in the cloud

This chapter covers

- Getting to know the Windows Azure Diagnostics platform
- Using logging to determine what's happening with your service
- Using the service management APIs
- Making your service self-aware with logging and service management

Building an application and deploying it to Azure are just the first steps in a hopefully long application lifecycle. For the rest of its life, the application will be in operation mode, being cared for by loving IT professionals and support developers. The focus shifts from writing quality code to running the application and keeping it healthy. Many tools and techniques are out there to help you manage your infrastructure.

What *healthy* means can be different for every application. It might be a measure of how many simultaneous users there are, or how many transactions per second

are processed, or how fast a response can be returned to the service caller. In many cases, it won't be just one metric, but a series of metrics. You have to decide what you're going to measure to determine a healthy state, and what those measurements must be to be considered acceptable. You must make sure these metrics are reasonable and actionable. A metric that demands that the site be as fast as possible isn't really measurable, and it's nearly impossible to test for and fix an issue phrased like that. Better to define the metric as an average response time for a standard request.

To keep your application healthy, you need to instrument and gather diagnostic data. In this chapter, we're going to discuss how you perform diagnostics in the cloud and what tools Azure provides to remediate any issues or under-supply conditions in your system.

18.1 Diagnostics in the cloud

At some point you might need to debug your code, or you'll want to judge how healthy your application is while it's running in the cloud. We don't know about you, but the more experienced we get with writing code, the more we know that our code is less than perfect. We've drastically reduced the amount of debugging we need to do by using test-driven development (TDD), but we still need to fire up the debugger once in a while.

Debugging locally with the SDK is easy, but once you move to the cloud you can't debug at all; instead, you need to log the behavior of the system. For logging, you can use either the infrastructure that Azure provides, or you can use your own logging framework. Logging, like in traditional environments, is going to be your primary mechanism for collecting information about what's happening with your application.

18.1.1 Using Azure Diagnostics to find what's wrong

Logs are handy. They help you find where the problem is, and can act as the flight data recorder for your system. They come in handy when your system has completely burned down, fallen over, and sunk into the swamp. They also come in handy when the worst hasn't happened, and you just want to know a little bit more about the behavior of the system as it's running. You can use logs to analyze how your system is performing, and to understand better how it's behaving. This information can be critical when you're trying to determine when to scale the system, or how to improve the efficiency of your code.

The drawback with logging is that hindsight is 20/20. It's obvious, after the crash, that you should've enabled logging or that you should've logged a particular segment of code. As you write your application, it's important to consider instrumentation as an aspect of your design.

Logging is much more than just remote debugging, 1980s-style. It's about gathering a broad set of data at runtime that you can use for a variety of purposes; debugging is one of those purposes.

18.1.2 Challenges with troubleshooting in the cloud

When you're trying to diagnose a traditional on-premises system, you have easy access to the machine and the log sources on it. You can usually connect to the machine with a remote desktop and get your hands on it. You can parse through log files, both those created by Windows and those created by your application. You can monitor the health of the system by using Performance Monitor, and tap into any source of information on the server. During troubleshooting, it's common to leverage several tools on the server itself to slice and dice the mountain of data to figure out what's gone wrong.

You simply can't do this in the cloud. You can't log in to the server directly, and you have no way of running remote analysis tools. But the bigger challenge in the cloud is the dynamic nature of your infrastructure. On-premises, you have access to a static pool of servers. You know which server was doing what at all times. In the cloud, you don't have this ability. Workloads can be moved around; servers can be created and destroyed at will. And you aren't trying to diagnose the application on one server, but across a multitude of servers, collating and connecting information from all the different sources. The number of servers used in cloud applications can swamp most diagnostic analysis tools. The shear amount of data available can cause bottlenecks in your system.

For example, a typical web user, as they browse your website and decide to check out, can be bounced from instance to instance because of the load balancer. How do you truly find out the load on your system or the cause for the slow response while they were checking out of your site? You need access to all the data that's available on terrestrial servers and you need the data collated for you.

You also need close control over the diagnostic data producers. You need an easy way to dial the level of information from `debug` to `critical`. While you're testing your systems, you need all the data, and you need to know that the additional load it places on the system is acceptable. During production, you want to know only about the most critical issues, and you want to minimize the impact of these issues on system performance.

For all these reasons, the Windows Azure Diagnostics platform sits on top of what is already available in Windows. The diagnostics team at Microsoft has extended and plugged in to the existing platform, making it easy for you to learn, and easy to find the information you need.

18.2 Diagnostics in the cloud is just like normal (almost)

With the challenges of diagnostics at cloud-scale, it's amazing that the solution is so simple and elegant. Microsoft chose to keep everything that you're used to in its place. Every API, tool, log, and data source is the same way it was, which keeps the data sources known and well documented. The diagnostics team provides a small process called MonAgentHost.exe that's started on your instances.

The `MonAgentHost` process is started automatically, and it acts as your agent on the box. It knows how to tap into all the sources, and it knows how to merge the data and move it to the correct locations so you can analyze it. You can configure the process on the fly without having to restart the host it's running on. This is critical. You don't

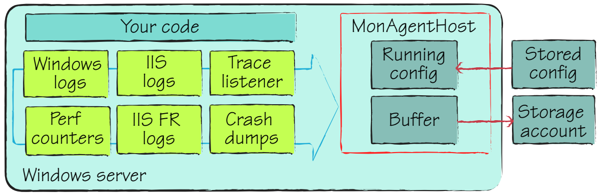


Figure 18.1 The MonAgentHost.exe process gathers, buffers, and transfers many different sources of diagnostic data on your behalf. It's the agent we'll be focusing on in this section.

want to have to take down a web role instance just to dial up the amount of diagnostic information you're collecting. You can control data collection across all your instances with a simple API. All the moving parts of the process are shown in figure 18.1. Your role instance must be running in full-trust mode to be able to run the diagnostic agent. If your role instance is running in partial trust, it won't be able to start.

As the developer, you're always in control of what's being collected and when it's collected. You can communicate with `MonAgentHost` by submitting a configuration change to the process. When you submit the change, the process reloads and starts executing your new commands.

18.2.1 Managing event sources

The local diagnostic agent can find and access any of the normal Windows diagnostic sources; then it moves and collates the data into Windows Azure storage. The agent can even handle full memory dumps in the case of an unhandled exception in one of your processes.

You must configure the agent to have access to a cloud storage account. The agent will place all your data in this account. Depending on the source of the data, it'll either place the information in BLOB storage (if the source is a traditional log file), or it'll put the information in a table.

Some information is stored in a table because of the nature of the data collection activity. Consider when you're collecting data from Performance Monitor. This data is usually stored in a special file with the extension *.blg*. Although this file could be created and stored in BLOB storage, you would have the hurdle of merging several of these files to make any sense of the data (and the information isn't easily viewed in Notepad). You generally want to query that data. For example, you might want to find out what the CPU and memory pressure on the server were for a given time, when a particular request failed to process.

Table 18.1 shows what the most common sources of diagnostic information are, and where the agent stores the data after it's collected. We'll discuss how to configure the sources, logs, and the (tantalizingly named) arbitrary files in later sections.

Table 18.1 Diagnostic data sources

Data source	Default	Destination	Configuration
Arbitrary files	Disabled	BLOB	`DirectoryConfiguration` class
Crash dumps	Disabled	BLOB	`CrashDumps` class

Table 18.1 Diagnostic data sources *(continued)*

Data source	Default	Destination	Configuration
Trace logs	Enabled	Azure table	web.config trace listener
Diagnostic infrastructure logs	Enabled	Azure table	web.config trace listener
IIS failed request logs	Disabled	BLOB	web.config `traceFailedRequests`
IIS logs	Enabled	BLOB	web.config trace listener
Performance counters	Disabled	Azure table	`PerformanceCounterConfiguration` class
Windows event logs	Disabled	Azure table	`WindowsEventLogsBufferConfiguration` class

The agent doesn't just take the files and upload them to storage. The agent can also configure the underlying sources to meet your needs. You can use the agent to start collecting performance data, and then turn the source off when you don't need it anymore. You do all this through configuration.

18.2.2 It's not just for diagnostics

We've been focusing pretty heavily on the debugging or diagnostic nature of the Windows Azure Diagnostics platform. Diagnostics is the primary goal of the platform, but you should think of it as a pump of information about what your application is doing. Now that you no longer have to manage infrastructure, you can focus your attention on managing the application much more than you have in the past.

Consider some of the business possibilities you might need to provide for, and as you continue to read this chapter, think about how the diagnostic tools can make some of these scenarios possible.

There are the obvious scenarios of troubleshooting performance and finding out how to tune the system. The common process is that you drive a load on the system and monitor all the characteristics of the system to find out how it responds. This is a good way to find the limits of your code, and to perform A/B tests on your changes. During an A/B test, you test two possible options to see which leads to the better outcome.

Other scenarios aren't technical in nature at all. Perhaps your system is a multitenant system and you need to find out how much work each customer does. In a medical imaging system, you'd want to know how many images are being analyzed and charge a flat fee per image. You could use the diagnostic system to safely log a `new image` event, and then once a day move that to Azure storage to feed into your billing system.

Maybe in this same scenario you need a rock-solid audit that tells you exactly who's accessed each medical record so you can comply with industry and government regulations. The diagnostic system provides a clean way to handle these scenarios.

An even more common scenario might be that you want an analysis of the visitors to your application and their behaviors while they're using your site. Some advanced

e-commerce platforms know how their customers shop. With the mountains of data collected over the years, they can predict that 80 percent of customers in a certain scenario will complete the purchase. Armed with this data, they can respond to a user's behavior and provide a way to increase the likelihood that they'll make a purchase. Perhaps this is a timely invitation to a one-on-one chat with a trained customer service person to help them through the process. The diagnostics engine can help your application monitor the key aspects of the user and the checkout process, providing feedback to the e-commerce system to improve business. This is the twenty-first-century version of a salesperson in a store asking if they can help you find anything.

To achieve all of these feats of science with the diagnostic agent, you need to learn how to configure and use it properly.

18.3 Configuring the diagnostic agent

If you're writing your code in Visual Studio, the default Azure project templates include code that automatically starts the diagnostic agent, inserts a listener for the agent in the web.config file, and configures the agent with a default configuration.

You can see this code in the `OnStart()` method in the WebRole.cs file.

```
public override bool OnStart()
{
   DiagnosticMonitor.Start("DiagnosticsConnectionString");   <-- (1) Starts diagnostic agent
   RoleEnvironment.Changing += RoleEnvironmentChanging;
   return base.OnStart();
}
```

The agent starts (1) with the default configuration, all in one line. The line also points to a connection string in the service configuration that provides access to the Azure storage account you want the data to be transferred to. If you're running in the development fabric on your desktop computer, you can configure it with the well-known development storage connection string `UseDevelopmentStorage=true`. This string provides all the data necessary to connect with the local instance of development storage.

You also need to create a trace listener for the diagnostic agent. The trace listener allows you to write to the Azure trace log in your code. Create a trace listener by adding the following lines in your web.config. If you're using a standard template, this code is probably already included.

```
<system.diagnostics>
  <trace>
    <listeners>
      <add type="Microsoft.WindowsAzure.Diagnostics
      ➥ .DiagnosticMonitorTraceListener,
      ➥ Microsoft.WindowsAzure.Diagnostics,
      ➥ Version=1.0.0.0, Culture=neutral,
      ➥ PublicKeyToken=31bf3856ad364e35"
      ➥ name="AzureDiagnostics">
        <filter type="" />
      </add>
```

```
    </listeners>
  </trace>
</system.diagnostics>
```

After you've set up the trace listener, you can use the trace methods to send information to any trace listeners. When you use them, set a category for the log entry. The category will help you filter and find the right data later on. You should differentiate between critical data that you'll always want and verbose logging data that you'll want only when you're debugging an issue. You can use any string for the trace category you want, but be careful and stick to a standard set of categories for your project. If the categories vary too much (for example, you have critical, crit, and important), it'll be too hard to find the data you're looking for. To standardize on log levels, you can use the enumerated type `LogLevel` in `Microsoft.WindowsAzure.Diagnostics`. To write to the trace log, use a line like one of the following:

```
using System.Diagnostics;

System.Diagnostics.Trace.WriteLine(string.Format("Page loaded on {0}",
➥ System.DateTime.Now, "Information");

System.Diagnostics.Trace.WriteLine("Failed to connect to database. ",
➥ "Critical");
```

Only people who have access to your trace information using the diagnostics API will be able to see the log output. That being said, we don't recommend exposing sensitive or personal information in the log. Instead of listing a person's social security number, refer to it in an indirect manner, perhaps by logging the primary key in the customer

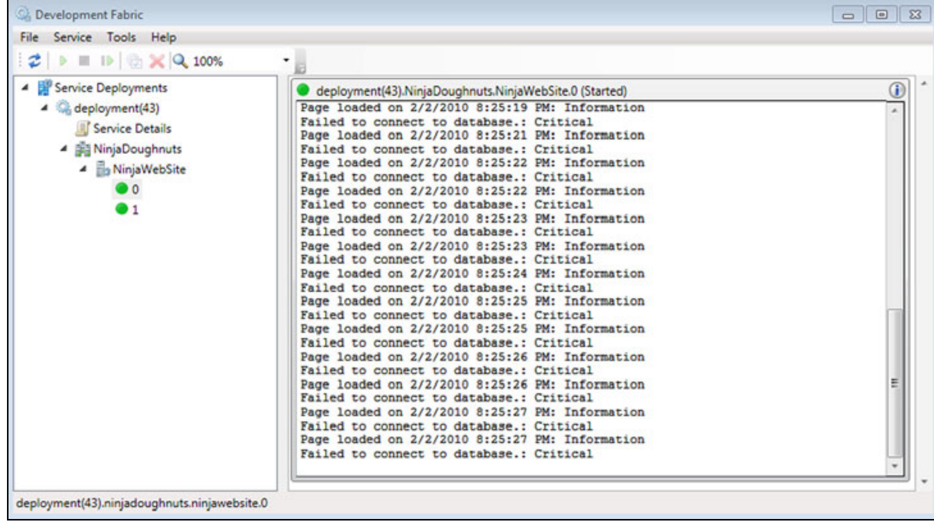


Figure 18.2 When writing to the trace channel, the entries are stored by the Windows Azure diagnostic trace listener to the Azure log, which can then be gathered and stored in your storage account for analysis. The trace output is also displayed in the dev fabric UI during development.

table. That way, if you need the social security number, you can look it up easily, but it won't be left out in plain text for someone to see.

Another benefit of using trace is that the trace output appears in the dev fabric UI, like in figure 18.2.

At a simple level, this is all you need to start the agent and start collecting the most common data. The basic diagnostic setup is almost done for you out of the box because there's so much default configuration that comes with it.

18.3.1 Default configuration

When the diagnostic agent is first started, it has a default configuration. The default configuration collects the Windows Azure trace, diagnostic infrastructure logs, and IIS 7.0 logs automatically. These are the most common sources you're likely to care about in most situations.

When you're configuring the agent, you'll probably follow a common flow. You'll grab the current running configuration (or a default configuration) from the agent, adjust it to your purposes, and then restart the agent. This configuration workflow is shown in figure 18.3.

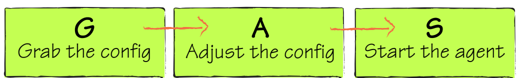


Figure 18.3 Use the GAS process to configure and work with the diagnostic agent. Grab the config, adjust the config, and then start the agent. Sometimes you'll grab the default config and sometimes the running config.

By default, the agent buffers about 4 GB of data locally, and ages out data automatically when the limit is reached. You can change these settings if you want, but most people leave them as is and just transfer the data to storage for long-term keeping.

Although the agent ages out data locally to the role instance, the retention of data after it's moved to Azure storage is up to you. You can keep it there forever, dump it periodically, or download it to a local disk. After it's been transferred to your account, the diagnostic agent doesn't touch your data again. The data will just keep piling up if you let it.

In the next few sections, we'll look at some of the common configuration scenarios, including how to filter the log for the data you're interested in before it's uploaded to storage.

18.3.2 Diagnostic host configuration

You can change the configuration for the agent with code that's running in the role that's collecting data, code that's in another role, or code that's running outside Azure (perhaps a management station in your data center).

CHANGING THE CONFIGURATION IN A ROLE

There will be times when you want to change the configuration of the diagnostic agent from within the role the agent is running in. You'll most likely want to do this during an `OnStart` event, while an instance for your role is starting up. You can change the configuration at any time, but you'll probably want to change it during

startup. The following listing shows how to change the configuration during the `OnStart` method for the role instance.

Listing 18.1 Changing the configuration in a role at runtime

```
using Microsoft.WindowsAzure.Diagnostics;

public override bool OnStart()
{
   var config = DiagnosticMonitor.GetDefaultInitialConfiguration();      <- 1 Grabs the default configuration

   config.PerformanceCounters.DataSources.Add(
    new PerformanceCounterConfiguration()
    {
      CounterSpecifier = @"\Memory\Available MBytes",     <- 2 Tracks the amount of available memory
      SampleRate = TimeSpan.FromSeconds(5.0)     <- 3
    });

config.PerformanceCounters.ScheduledTransferPeriod =
   TimeSpan.FromMinutes(1.0);     <- 4 Sets scheduled transfer time

   DiagnosticMonitor.Start("DiagnosticsConnectionString", config);

   return base.OnStart();
}
```

The first step to change the configuration is to grab the default configuration (1) from the diagnostic agent manager. This is a static method and it gives you a common baseline to start building up the configuration you want running.

In our sample, we're adding a performance counter called `\Memory\Available MBytes` (2). The `CounterSpecifier` property is the path to the performance counter. You can easily find performance counter paths if you use the Performance Monitor, as shown in figure 18.4.

Browse to the counter you want to track to find the specifier (which is like a file path) in the corner. Tell the agent to sample that performance counter every second (3), using the `PerformanceCounterConfiguration` class. Each data source the agent has access to has a configuration class. For each piece of data you want collected, you need to create the right type of configuration object, and add it to the matching configuration collection, which in this case is the `PerformanceCounterConfiguration` collection.

You also want to aggressively upload the performance counter data to Azure storage. Usually this value is set to 20 minutes or longer, but in this case you probably don't want to wait 20 minutes to see whether we're telling you the truth, so set it to once a minute (4). Each data source will have its own data sources collection and its own transfer configuration. You'll be able to transfer different data sources at different intervals. For example, you could transfer the IIS logs once a day, and transfer the performance counters every 5 minutes.

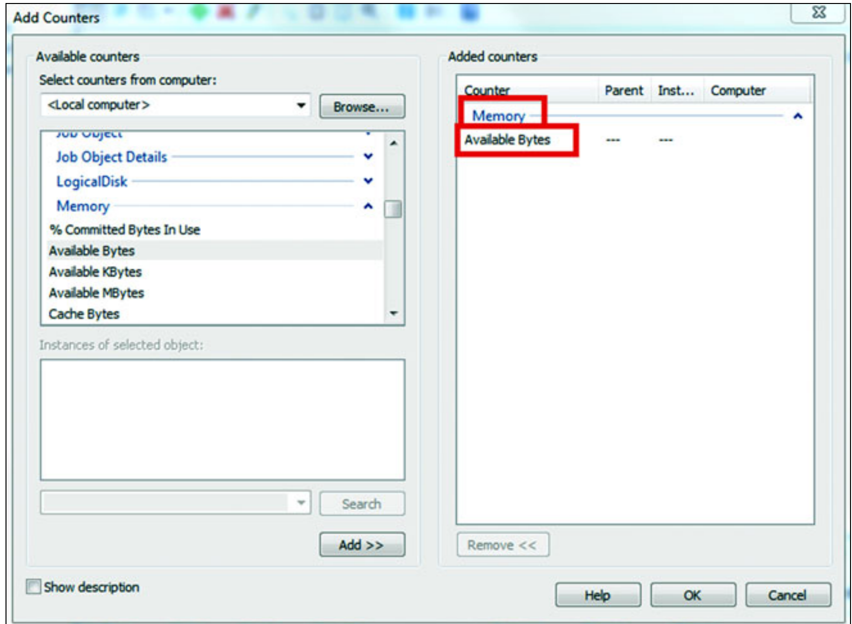


Figure 18.4 Finding the counter specifier using Performance Monitor. Browse to the correct category on the left, choose a counter, and then click Add. You join the category name and counter name with slashes, just like in a folder path. This example is showing `\Memory\Available Bytes`.

Finally, start the diagnostic agent. You need to provide it with the connection string to the storage account that you want the data uploaded to, and the configuration object you just built. The connection string defaults to the value `DiagnosticsConnectionString`, which is an entry in the cloud service configuration file. When you're playing with this on your development machine, you set the value to `UseDevelopmentStorage=true`, but in production you set it to be a connection string to your storage account in the cloud.

The data will be uploaded to different destinations, depending on the data source. In the case of performance counters, the data will be uploaded to an Azure table called WADPerformanceCountersTable, an example of which is shown in figure 18.5.

Figure 18.5 shows the results of the performance counter configuration. The agent tracked the available memory every 5 seconds, and stored that in the table. The entries were uploaded from the role instance to the table every minute, based on the configuration. The high order of the tick count is used as a partition key so that querying by time, which is the most likely dimension to be queried on, is fast and easy.

The RoleInstance column contains the name of the instance, to differentiate entries across the different role instances. In this case, there's only one instance.

Tracking log data can generate a lot of data. To make all this data easier to use, the diagnostic agent supports filters.

Timestamp	EventTickCount	DeploymentId	Role	RoleInstance	CounterName	CounterValue
2010-02-03 03:05:24	634007630871710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3452
2010-02-03 03:05:24	634007630921730000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3459
2010-02-03 03:05:24	634007630971710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3459
2010-02-03 03:07:04	634007631021710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3460
2010-02-03 03:07:04	634007631071700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3462
2010-02-03 03:07:04	634007631121720000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3462
2010-02-03 03:07:04	634007631171710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3462
2010-02-03 03:07:04	634007631221710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3460
2010-02-03 03:07:04	634007631271710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3432
2010-02-03 03:07:04	634007631321870000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3426
2010-02-03 03:07:04	634007631371710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3419
2010-02-03 03:07:04	634007631421710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3422
2010-02-03 03:07:04	634007631471710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3424
2010-02-03 03:07:04	634007631521710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3425
2010-02-03 03:07:04	634007631571700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3426
2010-02-03 03:08:04	634007631621710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3426
2010-02-03 03:08:04	634007631671710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3425
2010-02-03 03:08:04	634007631721720000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3427
2010-02-03 03:08:04	634007631771700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3427
2010-02-03 03:08:04	634007631821710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3427
2010-02-03 03:08:04	634007631871710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3421
2010-02-03 03:08:04	634007631921710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3420
2010-02-03 03:08:04	634007631971710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3417
2010-02-03 03:08:04	634007632021700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3430
2010-02-03 03:08:04	634007632071710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3436
2010-02-03 03:08:04	634007632121710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3436
2010-02-03 03:08:04	634007632171710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3432
2010-02-03 03:08:25	634007632221740000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3435
2010-02-03 03:08:25	634007632271700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3435
2010-02-03 03:08:25	634007632321710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3434
2010-02-03 03:08:25	634007632371710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3456
2010-02-03 03:08:25	634007632421710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3456
2010-02-03 03:08:25	634007632471710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3452
2010-02-03 03:08:25	634007632521710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3446
2010-02-03 03:08:25	634007632571710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3454
2010-02-03 03:08:25	634007632621710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3454
2010-02-03 03:08:25	634007632671710000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3452
2010-02-03 03:08:25	634007632721700000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3452
2010-02-03 03:08:25	634007632771720000	deployment(52)	NinjaWebSite	deployment(52).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3447

Figure 18.5 The performance counter data is stored in a table called WADPerformanceCountersTable. In this example, we're tracking the amount of available memory. You can see that the available memory starts at 3,452 MB and slowly drops to 3,436 MB.

FILTERING THE UPLOADED DATA

The amount of data collected by the diagnostic agent can become voluminous. Sometimes you might want to track a great deal of data, but when you're trying to solve a particular problem, you might want only a subset of data to look through. The diagnostic agent configuration provides for filtering of the results.

The agent still collects all the data locally. The filter is applied only when the data is uploaded to Azure storage. Filtering can narrow down the data you need to sift through, make your transfers faster, and reduce your storage cost. You can set the following property to filter based on the log level of the records that you've specified:

```
transferOptions.LogLevelFilter = LogLevel.Error;
```

All the log data remains local to the agent; the agent uploads only the entries that match or exceed the filter level you set.

CHANGING THE CONFIGURATION FROM OUTSIDE THE ROLE

Being able to change the configuration inside the role instance is nice, but you'll probably do this only during the startup of the instance. Dynamic changes to the configuration are more likely to come from outside the role instance. The source of these changes will probably be either an overseer role that's monitoring the first role, or a management application of sorts that's running on your desktop.

The agent's configuration is stored in a file local to the role instance that's running the agent. By default, this file is polled every minute for any configuration changes. You can change the polling interval if you want to by using the `DiagnosticMonitorConfiguration.ConfigurationChangePollInterval` property. You can set this property only from within the role the agent is running in.

To update the configuration remotely, either from another role or from outside Azure, you can use two classes. The `DeploymentDiagnosticManager` class is a factory that returns diagnostic managers for any role you have access to. You can use this manager to change the configuration remotely by using it to create `RoleInstanceDiagnosticManager` objects. Each `RoleInstanceDiagnosticManager` object represents a collection of diagnostic agents for a given role, one for each instance running in that role.

After you've created this object, you can make changes to the configuration like you did in the previous section. The trick is that you have to change the configuration for each instance individually. The following listing shows how to update the configuration for a running role.

Listing 18.2 Remotely changing the configuration of a role's diagnostic agent

```
using Microsoft.WindowsAzure.Diagnostics.Management;
using Microsoft.WindowsAzure.Diagnostics;
using Microsoft.WindowsAzure.ServiceRuntime;

DeploymentDiagnosticManager myDDM = new
➥ DeploymentDiagnosticManager(
➥ RoleEnvironment.GetConfigurationSettingValue
➥ ("DiagnosticsConnectionString"),
➥ txtDeploymentID.Text);

var myRoleInstanceDiagnosticManager =
        myDDM.GetRoleInstanceDiagnostic
        ➥ ManagersForRole("NinjaWebSite");

PerformanceCounterConfiguration CPUTime =          ❶ Creates new configuration
new PerformanceCounterConfiguration()
{
  CounterSpecifier = @"\Processor(_Total)
  ➥ \% Processor Time",
  SampleRate = TimeSpan.FromSeconds(5.0)
};
```

```
foreach (var instanceAgent in myRoleInstanceDiagnosticManager)   ❷
{
DiagnosticMonitorConfiguration
➥ instanceConfiguration =
➥ instanceAgent
        .GetCurrentConfiguration();
    instanceConfiguration
    ➥ .PerformanceCounters.DataSources.Add(CPUTime);   ❸ Adds new performance counters

instanceAgent.SetCurrentConfiguration(instanceConfiguration);
}
```

The first thing you do to update the configuration for a running role is get an instance of the `DeploymentDiagnosticManager` for the deployment. One object oversees all the roles in your deployment. Give it a connection string to your storage account for logging. This constructor doesn't take a configuration element like the `DiagnosticsMonitor` class does. You have to pass in a real connection string, or a real connection. The code in listing 18.2 grabs the string out of the role configuration with a call to `GetConfigurationSettingValue`.

From there, you ask for a collection of `RoleInstanceDiagnosticManager` objects for the particular role you want to work with. In this example, we're changing the configuration for the `NinjaWebSite` role. You'll get one `RoleInstanceDiagnosticManager` object for each instance that's running the `NinjaWebSite` role.

Next, you create the new part of the configuration you want to add to the agent ❶. In this example, you'll build another performance counter data source that will track the percentage of CPU in use. Then you'll iterate over your collection ❷ and add the new performance counter `CPUTime` to the current configuration ❸. This process is different from that used when you're changing the configuration in a role. Here you want to add to the configuration, not completely replace it. Finally, you update the configuration for that instance, which updates the configuration file for the diagnostic agent. When the agent polls for a configuration change, it'll pick up the changes and recycle to load them.

Figure 18.6 shows the results of the configuration changes that you've made.

In this sample, we've put this code in the role, but this code would work running from any application that's running outside Azure as well. The only difference would be how you provide the connection string to storage, and how you provide the deployment ID.

We've looked at the standard data sources for Windows Azure diagnostics, but there's one hole remaining. What if you want to manage a diagnostic source that isn't on the official list? This situation is where the escape hatch called *arbitrary diagnostics sources* comes into play.

18.3.3 The other data sources

Up until now, we've discussed how to configure a performance counter data source and how to enable the trace listener in the web.config file. Now let's look briefly at the other data sources that are configured in similar ways.

EventTickCount	DeploymentId	Role	RoleInstance	CounterName	CounterValue
634007737309650000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3412
634007737359870000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	26.455785
634007737359870000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3410
634007737409610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	12.953047
634007737409610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3409
634007737459610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	21.378814
634007737459610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3418
634007737509620000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	10.321144
634007737509620000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3418
634007737559600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	9.487033
634007737559600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3418
634007737609600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	10.771193
634007737609600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3419
634007737659610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	9.679234
634007737659610000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3419
634007737709630000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	18.414919
634007737709630000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3418
634007737759600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Processor(_Total)\% Processor Time	22.143216
634007737759600000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.1	\Memory\Available MBytes	3411
634007737817950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	23.094753
634007737817950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3404
634007737867950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	14.653984
634007737867950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3404
634007737917950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	8.119289
634007737917950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3403
634007737967950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	15.763015
634007737967950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3395
634007738017950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	34.482345
634007738017950000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3395
634007738068060000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Processor(_Total)\% Processor Time	26.687914
634007738068060000	deployment(54)	NinjaWebSite	deployment(54).NinjaDoughnuts.NinjaWebSite.0	\Memory\Available MBytes	3394

Figure 18.6 You can change the configuration of the Windows Azure diagnostic agent running in each instance quite easily. In this example, we added the % Processor Time performance counter to the agent. You can do this remotely, even from outside Azure.

CRASH DUMPS

Crash dumps are the log file of last resort. They aren't really a log file, but a dump of the status of the computer when a horrible problem has arisen. There are two types of crash dumps. The normal dump includes a copy of all the memory on the machine. The mini dump holds only the most important information, without a complete copy of everything.

If you're running a web application, ASP.NET should handle any errors that aren't handled by your application (in code or in the global.asax). A crash dump usually occurs only during a truly catastrophic error. When your code is running in a worker role, without the soft embrace of ASP.NET, you're likely to see these dumps more often.

Crash dump files are stored in the local data buffer and transferred with the common logs. You can choose which size dump you want by passing in `true` to the `EnableCollection` or `EnableCollectionToDirectory` method for a full dump and passing in `false` for a mini dump.

IIS FAILED REQUEST LOGS

The failed request logs for IIS are a new feature in IIS 7. IIS tracks log data for requests as they come through, but keeps the data only if certain configurable conditions are met. A condition for keeping the log data might be that a response takes too long to complete the request. If the response is completed fast enough, then the log buffer is flushed. You can configure how IIS manages this process in the `tracing` section of the `system.webServer` part of your web.config. After you configure failed request tracing for IIS in this way, the logs are collected with the rest of the data logs.

WINDOWS EVENT LOGS

Windows event logs can provide important clues to serious problems with your applications. Some applications create custom event log sources for their own logging. The diagnostic agent in Azure can collect these logs and transfer them to storage for you.

You need to subscribe to the event data that you want to receive using an XPath expression. Because of the security profile your processes run on, you won't be able to read the security windows event log. If you add this log to your configuration, nothing will be logged, and it won't work correctly until you remove it.

To capture Windows event logs, you can use the following expression:

```
diagConfig.WindowsEventLog.DataSources.Add("System!*");
```

This code grabs everything from the Windows System Event log, which is where you usually want to start your investigations.

18.3.4 Arbitrary diagnostic sources

Windows Azure Diagnostics covers a lot of the diagnostic sources you might use to troubleshoot an issue with your system. It covers IIS logs, performance counters, Windows event logs, and several other things. Over time, you'll probably devise your own diagnostic source; maybe a log you're creating that you need to track. Perhaps this is custom billing data, or a compressed log of the images used in production, or it might just be the output of a third-party logging framework you've chosen to use.

The agent can transfer anything you want. All you need to do is get that data into a file in a designated folder. When you configure the agent, you tell it which custom directories you want it to monitor. You need to configure some local storage and then write your log files to it.

Each data source has a configuration class that you add to the agent's configuration, and custom log locations aren't any different. We'll use the `DirectoryConfiguration` class to tell the agent to monitor a folder. You can set how large that directory is allowed to become, as well as scheduled transfer characteristics, as shown in the following code:

```
DirectoryConfiguration specialLogsDC = new DirectoryConfiguration();

specialLogsDC.Path =
➥ RoleEnvironment.GetLocalResource("specialLogs").RootPath;    // Source directory
```

```
specialLogsDC.Container = "speciallogs";
instanceConfiguration.Directories
➥ .DataSources.Add(specialLogsDC);
```

Destination storage container

Windows Azure Diagnostics is a powerful tool you can use to help troubleshoot and diagnose problems. But it isn't just for problems, as we've discussed; you can also use it to help monitor the behavior of the system, or the actions of your users. Although Windows Azure Diagnostics is wonderful, it's only a source of data. You still need to analyze the data to turn it into information, and then take action. Sometimes the action you need to take is to change the configuration of the service model you're running your application in. For example, you might need to add some instances to respond to a spike in traffic. Regardless of the result of your analysis, you'll need to store the data you collect in Azure storage to be able to use it. To do this, you use transfers.

18.4 Transferring diagnostic data

The diagnostic agent does a great job of collecting all the local data and storing it on the machine it's running on. But if the diagnostic information is never moved to Azure storage, it won't be any good to anyone. This is where transfers come into play.

There are two types of transfers, one of which you have already seen in play. We've already talked about the scheduled transfer, which sets up a timer and transfers the related data on a regular basis to your storage account. Each data source category has its own transfer schedule. You can transfer performance counter data at a different rate than you transfer the IIS logs.

The second type of transfer is an on-demand transfer. You usually perform an on-demand transfer when you have a special request of the data.

Let's look at each of these kinds of data transfers in more detail.

18.4.1 Scheduled transfer

In our sample in listing 18.1, a scheduled transfer of the performance counter data is set (4) to occur every minute. As we covered earlier, transferring every minute is quite aggressive, and is probably reasonable only in a testing or debugging environment.

In our next example, we're going to show you how to transfer the IIS logs to storage on a daily basis. The IIS logs are automatically captured by default by the diagnostic agent, so you don't need to add them as a data source. You can set the transfer interval to once a day with this line:

```
instanceConfiguration.Logs.ScheduledTransferPeriod = TimeSpan.FromDays(1.0);
```

Any log files that are captured are sent to a container in BLOB storage, not to a table. Each transfer results in one file in the container. A container hierarchy similar to what you would see on the real server is created for you by the diagnostic agent. Your IIS logs will be in a folder structure similar to what you're used to.

If logs that you don't want to transfer are collected, you can set `ScheduledTransferPeriod` to `0`. This setting disables the transfer of any data for that data source. We

typically do this for the Azure diagnostics log themselves, at least until there's a problem with the diagnostic agent itself that requires troubleshooting.

That's how you schedule a transfer. Now let's discuss how you can trigger a transfer on demand.

18.4.2 On-demand transfer

An on-demand transfer lets you configure a onetime transfer of the diagnostics data. This kind of transfer gives you the ability to pick and choose what is transferred and when. A typical scenario is you want an immediate dump of logs because you see that something critical is happening. You can set up an on-demand transfer in much the same way as you would a normal transfer, although there are some differences.

In the following listing, we're initiating an on-demand transfer from within one of the instances, but you can also initiate the transfer from outside the role with an administrative application.

Listing 18.3 Initiating an on-demand transfer

```
DeploymentDiagnosticManager myDDM = new
➥ DeploymentDiagnosticManager(RoleEnvironment
➥ .GetConfigurationSettingValue("DiagnosticsConnectionString"),
➥ txtDeploymentID.Text);

var myRoleInstanceDiagnosticManager =
➥ myDDM.GetRoleInstanceDiagnosticManagersForRole("NinjaWebSite");

DataBufferName datasourceToTransfer =
➥ DataBufferName.PerformanceCounters;                    ← 1 Transfers performance counters
OnDemandTransferOptions transferOptions = new OnDemandTransferOptions();

transferOptions.From = DateTime.UtcNow -
➥ TimeSpan.FromHours(1.0);                               ← 2 Selects time filter for data to be sent
transferOptions.To = DateTime.UtcNow;
transferOptions.NotificationQueueName = "transfernotificationqueue";   ← 3 Provides queue name for notification messages

foreach (var instanceAgent in myRoleInstanceDiagnosticManager)
{
  Guid requestID = instanceAgent
  ➥ .BeginOnDemandTransfer
  ➥ (datasourceToTransfer, transferOptions);              ← 4 Starts transfer for each instance in role
  System.Diagnostics.Trace.WriteLine("on demand started:" +
  ➥ requestID.ToString(), "Information");
}
```

Like in the remote configuration example, you need to get a reference to the `RoleInstanceDiagnosticManagersForRole` class. This reference will let you work with the configuration manager for each instance. In this example, you're going to be transferring the performance counter data over to an Azure table (1). This transfer will include all counters you might have running from prior configuration changes.

You use the `OnDemandTransferOptions` class to configure how the transfer should happen. This class has several parameters that you'll want to set. Set a time filter at (2),

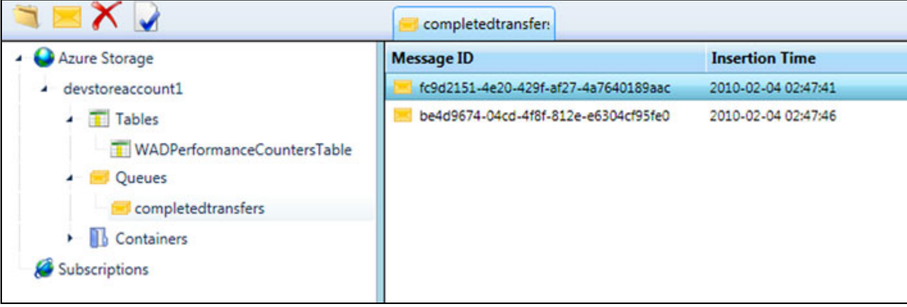


Figure 18.7 You can configure an on-demand transfer to notify you when a transfer is complete by placing a message in a queue you specify. In this example, the role had two instances, so two transfers were completed, one for each instance.

which tells the agent to send over only the performance counter data that's been generated in the last hour. If you're transferring a log, you can also specify a log level filter (informational, critical, and so on).

When you start the transfers, each instance performs its own transfer. The transfer operation is an asynchronous operation; you start it and walk away. In most cases, you'll want to know when the transfers have completed so that you can start analyzing the data. You can have the transfer agent notify you when the transfer for an instance is complete by passing in a queue name ③. As each transfer is started, you'll be given a unique ID ④ to track. You can see the completion messages in figure 18.7. Each message contains one of the unique IDs associated with each transfer.

When each transfer is complete, the agent drops a small message onto the queue you designated with that same ID. The message lets you track which transfers have been completed.

A great place to use an on-demand transfer is in the `OnStop` method in your `RoleEntryPoint` override class. Whenever a role is being shut down, either intentionally or otherwise, this method fires. If you do an on-demand transfer in this method, you'll save your log files from being erased during a reboot or a move. These log files can help you troubleshoot instance failures. Because you might not have enough time to transfer gigabytes of log files, make sure that you're transferring only the critical information you need when you do this.

Now that you know how to get data about what's happening with your service, we need to tell you about the APIs that'll help you do something about what you see happening.

18.5 Using the service management API

After your application is up and running in Azure, you'll want to automate some of the management functions. Automation can include scaling your roles, changing configuration, and automating deployments. Almost anything you can do through the Azure portal you can do through the service management API.

The service management API is built like all the other APIs in Azure. It uses REST and XML under the hood, wrapped in a pleasant .NET library. You can use the service management API directly with REST, but most people use either the library or use a tool that calls the APIs.

All the management APIs we're going to discuss can be called from inside or from outside Azure. All management calls are free; they incur no cost to call or execute. The Azure team has said that they monitor the use of the APIs and can throttle back your calls if they're abused.

To start using the service management API, you need to configure your account with certificates for API authentication. After you've done that, you'll be able to send it commands. After we show you how to configure your account, we're going to look at how you can work with your services and containers, how to automate a deployment to the cloud, and how you can use the management API to scale your service up or down.

18.5.1 What the API doesn't do

A little earlier we said that the service management API can do almost as much as the portal. However, you must use the portal to do the following things:

- *Access billing data*—The portal has several tools you can use to monitor your usage and billing in near real time. Monitoring allows you to estimate your charges as they occur. The final numbers are crunched at the end of the month to generate your bill.
- *Create subscriptions and create compute or storage services*—After you've created the subscription and services, you can do everything else from the management API.
- *Deploy management certificates*—You can't use the management service to deploy a management certificate; you have to do this manually.

To make calls with the API, you need to sign them with a certificate, which we'll discuss next.

18.5.2 Setting up the management credentials

The service management API has a lot of power, so all of its calls and responses must be secure. All calls are transferred over HTTPS, using a signed certificate that you associate with your Azure account. Whether you're calling the REST by hand or using the .NET library, you'll need to attach a certificate to your Azure account trusts.

You can use any X.509 v3 certificate that you want to use. Because you have control over which certificates your account trusts, you can use self-signed certificates if you want to. You can also use certificates that you've purchased from a certificate authority like VeriSign.

Your account can hold up to five certificates. You can distribute those certificates to different people or processes, and then eventually revoke them if you need to. All a person needs in order to use the management API on your services is that certificate and your subscription number. We'll look at how to revoke a certificate later in this section.

SETTING UP A CERTIFICATE

To set up your management certificates, you need a certificate to upload. We're going to walk you through the process of creating a certificate locally and then uploading it to your account.

The goal is to create a .cer file that holds the public key for your certificate. You never share the private half of the key. This public key will be uploaded to Azure, and Azure will use it to verify that your private key was used to sign the management API request.

You need to use IIS 7 to create a self-signed certificate. Open the IIS manager and look for the Features view. Listed there is a link for Server Certificates. Click Create Self-Signed Certificate in the Actions pane and follow the steps. You'll give the certificate a name, which will be used whenever you're working with the certificate.

You can also use the Visual Studio command prompt to create a self-signed certificate. Open the command prompt (make sure to run it as an administrator), and then enter the following command:

```
makecert -r -pe
➥ -n "CN=CompanyName"
➥ -a sha1
➥ -len 2048
➥ -ss My "filename.cer"
```

This command creates a certificate that you can use in the local directory.

IMPORTING AND REVOKING A CERTIFICATE

Importing a certificate is as easy as logging into your Azure portal and going to the Account tab. Choose Manage My API Certificates. The window, shown in figure 18.8,

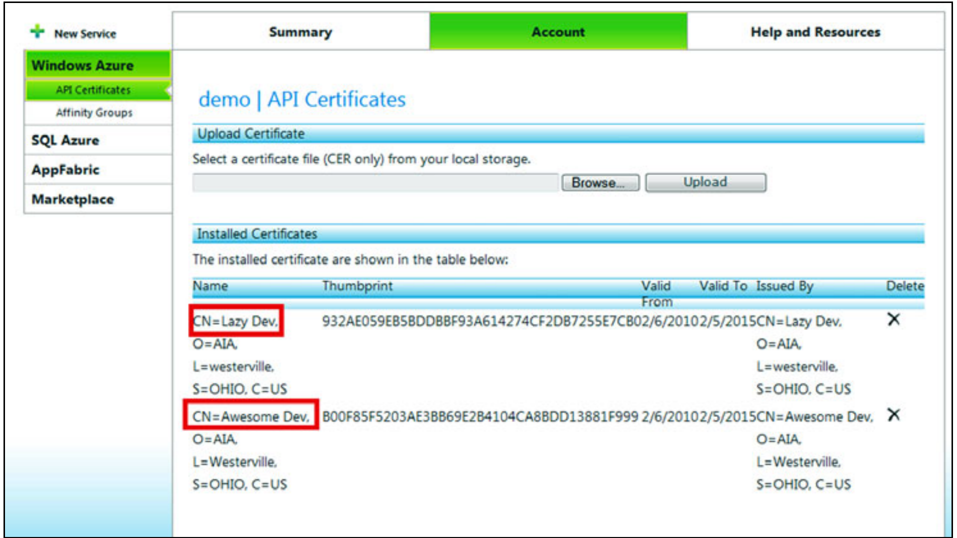


Figure 18.8 This window shows that two X.509 certificates have been imported; one for Lazy Dev and one for Awesome Dev. Certificates are used to authenticate to Azure when you're using the service management API. Guess whose certificate we're going to revoke in the next example?

displays the certificates you've uploaded; you can also upload a new certificate from this window. Your certificate must be in a .cer file. If you have a different format, you can easily convert it by importing it into your Windows certificate store, and then exporting it in the format you want.

You can have up to five certificates in your account at a time; take advantage of them. Each person or system that's using the management API should have their own certificate. If you provide certificates in this way, you'll have an easy way to revoke their access. To revoke a certificate, click the Delete X icon next to the one you want to revoke.

You need to attach your certificate to each request that you send to the API. Attaching your certificate ensures that the message is signed with your private key, which only you should have. When Azure receives your message, it'll check that the message came from you by opening it with the public key you uploaded in the .cer file.

You've got some certificates now and you're ready to learn about some of the things you can use the service management API for.

18.5.3 Listing your services and containers

You can save a lot of time if you learn how to automate your deployments instead of doing them by hand through the portal. We're going to start this section by showing some code you can use to get a list of the service and storage accounts you've created in Azure. This code is fairly primitive and uses the REST API directly. We'll eventually start using a tool that will abstract away the raw REST so that you have something nicer to work with.

You're going to use the `WebRequest` class to work with the REST call you'll be making. You need to pass in the URI of the call you want to make. The following listing shows how to use REST to query for a list of services.

Listing 18.4 Querying for the list of services with REST

```
var request = (HttpWebRequest)WebRequest.Create(
➥ "https://management.core.windows.net/7212af99-206f-dem0-9334-
➥ 380d0f841d0b/services/hostedservices");                          ❶

request.Headers.Add("x-ms-version:2009-10-01");
request.ClientCertificates.Add(
➥ X509Certificate2.CreateFromCertFile(@"C:\
➥ \awesomedev.cer"));                          ❷ Adds certificate to the request

var responseStream = request.GetResponse().GetResponseStream();

var services = XDocument.Parse(new
➥ StreamReader(responseStream).ReadToEnd());
```

In the previous listing, you pass in a string that includes the root of the call, `https://management.core.windows.net/`, and the subscription ID (which can be found on your Accounts tab in the portal) at ❶.You pass this string because you want a list of

the services that you've created. Every request into the management service needs this base address. Because for this example we want to include a list of the hosted services you have in your account, add `/services/hostedservices` to the end of your URL.

You also need to attach a version header and your certificate for authentication. The version header tells Azure which version of the management service you intend to call. Right now the latest version to call is the one that was published in October of 2009, so let's use that one. The certificate is easily attached at line (2). You're attaching it from the file you generated above. You could have used the certificate that's in your secure certificate store.

When you run this code, the result you get in raw XML format is shown in the following listing. All three services that are running in your subscription (`aiademo1`, `aiademo2`, and `aiademo3`) are listed in `HostedService` elements.

Listing 18.5 The raw XML that comes back from our request

```
<HostedServices xmlns="http://schemas.microsoft.com/windowsazure"
➥ xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <HostedService>
    <Url>https://management.core.windows.net
    ➥ /7212af99-206f-dem0-9334-80d0f841d0b
    ➥ /services/hostedservices/aiademo1</Url>
    <ServiceName>aiademo1</ServiceName>
  </HostedService>
  <HostedService>
    <Url>https://management.core.windows.net
    ➥ /7212af99-206f-dem0-9334-380d0f841d0b
    ➥ /services/hostedservices/aiademo2</Url>
    <ServiceName>aiademo2</ServiceName>
  </HostedService>
  <HostedService>
    <Url>https://management.core.windows.net
    ➥ /7212af99-206f-dem0-9334-380d0f841d0b
    ➥ /services/hostedservices/aiademo3</Url>
    <ServiceName>aiademo3</ServiceName>
  </HostedService>
</HostedServices>
```

Simple name for your service

After you've received the XML, you can use something like Language-Integrated Query (LINQ) to XML to parse through the results. You'll want the URL for each service for later when you're calling back to reference that particular service. You can use the same code to get a list of storage accounts in your Azure account by changing the address of the request from `/hostedservices` to `/storageservices`.

You'll be able to use these endpoints to see the storage and service accounts you have. This part of the API is read-only. To create service or storage accounts, you'll have to use the portal.

Now that you can make simple queries of the management service, let's flip over to csmanage.exe and use it to deploy a simple web application to Azure.

18.5.4 Automating a deployment

Even though you can do everything you need through the naked REST API of the service management service, it's a lot easier to use something that provides a higher level of abstraction. REST is fine, but it's too low-level for us on a daily basis.

There are two popular options to go with. The first is a collection of PowerShell commandlets that have been provided by Microsoft. These are useful when you're integrating cloud management into your existing management scripts. You can find these commandlets at http://code.msdn.microsoft.com/azurecmdlets.

We're going to use csmanage.exe for the rest of this chapter. This small utility is provided by the Azure team and can be found at http://code.msdn.microsoft.com/windowsazuresamples.

We might be using a higher-level tool, but you still need to provide the tool with the subscription ID of your Azure account and the thumbprint for the certificate you'll be using to manage your account. You can enter these into the configuration file for the tool, csmanage.exe.config. The easiest place to find the ID and the thumbprint is on the Azure portal.

The `csmanage` application can work with one of three resources online: hosted services, storage services, or affinity groups. Each command you send will include a reference to the resource you want to work with.

GETTING A LIST OF HOSTED SERVICES

To get a simple list of the hosted services you have in your account, you can execute the following command at a command prompt. This command executes the same query you previously made manually.

```
csmanage.exe /list-hosted-services
```

When you run this command, the application connects to your account in the cloud and returns a list of the services you have. When we executed this command, we had just one hosted service, as shown in figure 18.9.

After you have a list of your services, you next task is to create a deployment of your application.

Figure 18.9 You can use the `csmanage` command-line tool to execute management commands against your account in the cloud. In this example, we used a simple command to list the hosted services we have.

CREATING A DEPLOYMENT

First, you need to create the package for your code. You've probably done this a thousand times by now, but right-click the Azure project and choose Publish.

When you're using the management service, a service package has to be in BLOB storage for you to deploy it; you can't upload it as part of the actual `create-deployment` command. What you can do is upload the package through any tool you want to use. In this example, we just want to upload the cspkg file. This BLOB container can be public or private, but it should probably be private. You don't want some jokester on the interweb to download your source package. Because the management call is signed, it'll have access to your private BLOB containers, so you won't have to provide the credentials. The configuration file will be uploaded when you run the `create-deployment` command. The following listing shows the command-line code for deploying a package.

Listing 18.6 Pushing a deployment to the cloud from the command line

```
csmanage.exe
    /create-deployment        /slot:staging
    /hosted-service:aiademo1
    /name:ninjas
    /label:build1234
    /config:ninja.cscfg
    /package:http://aiademostore.blob.core.windows.net/
➥ deployments/NinjaDoughnuts.cspkg
```

Local path of configuration file to upload (points to `/config:ninja.cscfg`)

URL of package to deploy (points to `/package:...`)

That's a lot to type in by hand. The most common use of `csmanage` is in an automated deployment script. You could make the deployment completely hands-off with enough script and PowerShell. When you execute the command in listing 18.6, there'll be a slight pause as the configuration is uploaded and the management service deploys your bits. The output of the command is shown in figure 18.10.

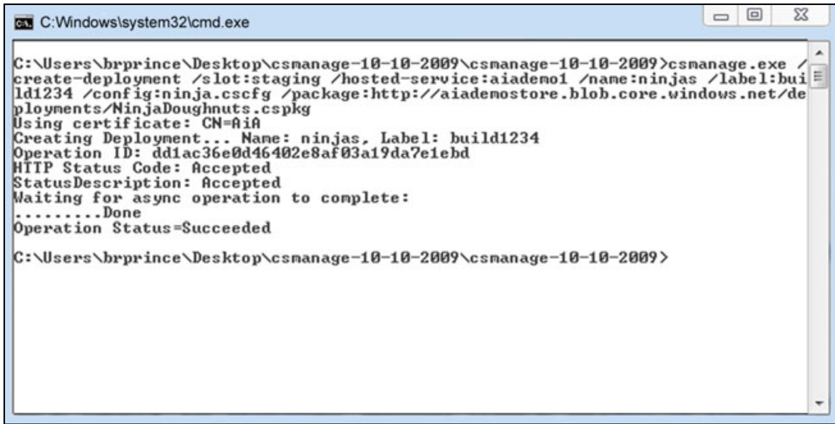


Figure 18.10 The output from `csmanage` when you script out a deployment to staging. We're deploying our ninja doughnut application, which has nothing to do with doughnuts or ninjas.

All this is wonderful, but you don't want to just upload and deploy your code; you need to start it.

STARTING THE CODE

You need to start your code so that the FC can start your site. The command to start everything is quite simple:

```
csmanage
➥ /update-deployment
➥ /slot:staging
➥ /hosted-service:aiademo1
➥ /status:running
```

This command tells the management API that you want to update the deployment of your service that's in the staging slot to running. (You can use the same command to set the status to suspended.) After you set the status to running, it can take a few minutes for the FC to get everything up and running. You can check on the status of each instance of your roles by using the `view-deployment` command:

```
csmanage /view-deployment /slot:staging /hosted-service:aiademo1
```

When you execute this command, you'll get a detailed view of each instance. In this example, both of our instances were busy, as shown in figure 18.11. If we just wait a few minutes, they'll flip to ready.

When your instance status reads as ready, you can use a command to perform a VIP swap.

PERFORMING A VIP SWAP

Remember, a VIP swap occurs when you swap the virtual IPs for production and staging, performing a clean cutover from one environment to the other. Using a VIP swap

```
C:\Windows\system32\cmd.exe

C:\Users\brprince\Desktop\csmanage-10-10-2009\csmanage-10-10-2009>csmanage /view
-deployment /slot:staging /hosted-service:aiademo1
Using certificate: CN=AiA
Getting Deployment
Name:ninjas
Label:build1234
Url:http://49dfc896de8a43d7bbdc47c52e3cf5a9.cloudapp.net/
Status:Running
DeploymentSlot:Staging
PrivateID:49dfc896de8a43d7bbdc47c52e3cf5a9
UpgradeDomainCount:2
RoleInstanceList contains 2 item(s).
    RoleName: NinjaWebSite
    Role InstanceName: NinjaWebSite_IN_0
    Role InstanceStatus: Busy
    RoleName: NinjaWebSite
    Role InstanceName: NinjaWebSite_IN_1
    Role InstanceStatus: Busy
Operation ID: 2f0de1c14b5146a69849a658f6ef718e
HTTP Status Code: OK
StatusDescription: OK

C:\Users\brprince\Desktop\csmanage-10-10-2009\csmanage-10-10-2009>
```

Figure 18.11 You can check the detailed status of each role instance by using the `view-deployment` option. In this example, our instances are busy because we've just deployed the package. In a moment, the status will change to ready.

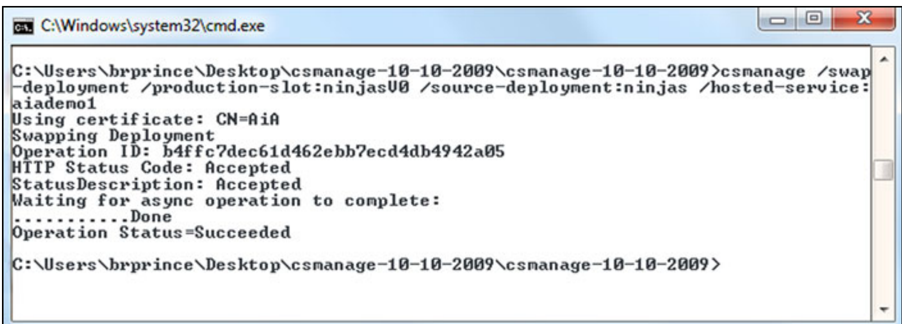


Figure 18.12 The VIP swap has successfully swapped the staging and production slots. You need to provide the names of the deployments in each slot.

is the simplest way to deploy a new version. You can always do an update domain walk, which is more complicated but provides the capability for a rolling upgrade.

You need a deployment in each slot to perform a VIP swap through the `csmanage` or REST interfaces. If you're deploying for the first time to an empty environment, either deploy your first version into the production slot, or rerun the `swap-deployment` command and adjust the `slot` parameter to `production`.

To perform the VIP swap, execute the following command:

```
csmanage /swap-deployment /production-slot:ninjasV0 /source-deployment:ninjas
     /hosted-service:aiademo1
```

Be careful when you type in this command. The third parameter, `source-deployment`, is different from what you would expect. Because the second parameter is `production-slot`, you would expect the third to be `staging-slot`. The naming isn't terribly consistent. Another mystery is why you need to define the slot names at all. You can have only one slot of each anyway. Go figure.

Figure 18.12 shows the successful completion of the VIP swap.

Now that you've swapped out to production and have fully tested the slot, you can tear down the staging slot.

TEARING DOWN A DEPLOYMENT

Use the following command to suspend the state of the servers, and then delete the deployment:

```
csmanage /update-deployment /slot:staging /hosted-service:aiademo1 /
     status:suspended

csmanage /delete-deployment /slot:staging /hosted-service:aiademo1
```

You need to execute these two commands back-to-back. You can't delete a service while it's running, so you have to wait for the first command to suspend the service to finish.

Figure 18.13 shows the successful completion of these commands.

You've successfully automated the deployment of a cloud service, started it, moved it to production, and then finally stopped and tore down the old version of the service.

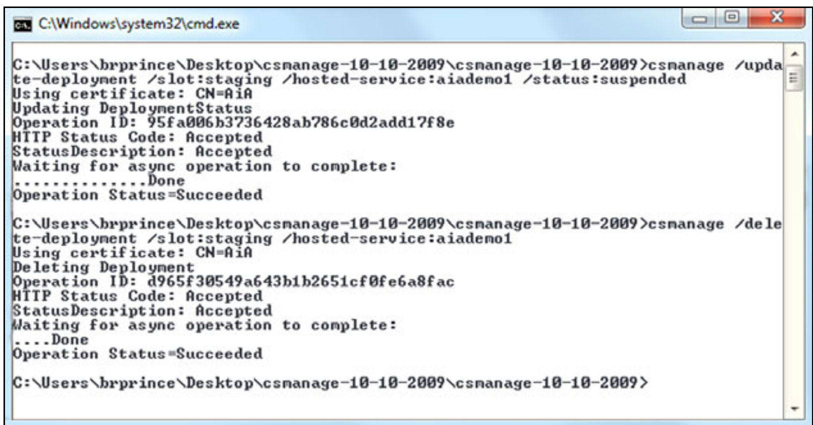


Figure 18.13 After deploying and swapping your new version to production, you suspend the staging environment and then tear it down to stop the billing. This process involves two steps: first you must suspend the service, and then you delete it.

You've done this without having to use the portal. Even so, you still had to use the portal to upload your management certificates and create the initial service container.

Now to what people really care about in the cloud: dynamically scaling the number of instances that are running your service.

18.5.5 Changing configuration and dynamically scaling your application

One of the golden promises of cloud computing is the dynamic allocation of resources to your service. It's really cool that you can deploy a service from nothing to 20 servers, but you also want to be able to change that from 20 servers to 30 servers if your service experiences a spike of some sort.

CHANGING THE SERVICE CONFIGURATION FILE

As we've discussed in chapters past, your service is based on a service model that's defined in your service configuration file. This file defines how many instances per role your service defines. You can change this file in one of three ways.

The first way to change the file is to edit it online in the portal. This is the simplest way to change it, but it's also the most primitive. You can't wire this up into an automated system or into an enterprise configuration management system.

The second method can use configuration files generated by your enterprise configuration management change system. You can upload a new file (version them with different file names so you can keep track of them) into BLOB storage, and then point to that file when you change the configuration from the portal.

The third option lets you upload a new configuration through the service management REST API. If you don't want to use REST, you can use `csmanage`, like we've been doing in the past few sections.

The FC responds in different ways depending on how you've changed your configuration and how you've coded your `RoleEnvironmentChanging` event. By default, if

you've changed anything besides the instance count of a role, the FC tears down and restarts any instances for the affected role. That part of the service comes to a screeching halt while the instances are restarted. The length of the outage isn't very long, but it's there. Be sure to think about which changes should restart your roles and which shouldn't; you can adjust the code in the `RoleEnvironmentChanging` event accordingly.

If you increased the instance count, none of the instances are affected. The FC creates new instances, deploys your bits, configures them, and wires them into the network and load balancer like it does when you've started the role instances from scratch. This process can take several minutes. You won't see an immediate availability, so be patient. Take this slight delay into consideration when you're developing the logic you'll use to detect when you need to add instances.

If you've reduced the number of instances, then the FC uses an undocumented algorithm to decide which instances are shed. They're unwired from the load balancer so that it doesn't receive any new traffic, and then they're torn down.

After you've made changes to the configuration file, you have to deploy it.

DEPLOYING THE NEW CONFIGURATION FILE

You can use `csmanage` to deploy a new configuration file. The command should be fairly easy to understand by now. The .cscfg file that will be uploaded by the following command is in the same folder as `csmanage`. Your file probably isn't in the same folder, so you'll need to provide a path to the configuration file.

```
csmanage /change-deployment-config /config:ninja_daytime.cscfg /
     slot:production /hosted-service:aiademo1
```

When you're using the management API (and `csmanage`), you have to upload a locally stored configuration file. In figure 18.14, you can see that it took quite a while for the

Figure 18.14 We've uploaded a new configuration file for our service. Because we changed only the number of instances for each role, we didn't have any downtime. The FC has spun up new instances to meet our needs.

management service to spin up the new instance. The total time it took in this instance was about 4 minutes.

This might seem like a lot of work. Couldn't Azure do all this for you automatically?

WHY DOESN'T AZURE SCALE AUTOMATICALLY?

Many people wonder why Azure doesn't auto-scale for them. There are a couple of reasons, but they all fall into the "My kid sent 100,000 text messages this month and my cell phone bill is over a million dollars" category.

The first challenge for Azure with auto-scaling your application is that Azure doesn't know what it means for your system to be busy. Is it the depth of the queue? Which queue? Is it the number of hits on the site? Each system defines the status of *busy* different than any other system does. There are too many moving parts for a vendor such as Microsoft to come up with a standard definition that'll make all its customers happy.

Another reason is that Microsoft could be accused of too aggressively scaling up, and not scaling down fast enough, just to increase the charges on your account. We don't think they would do that, but the second someone thinks the algorithm isn't tuned to their liking, Microsoft would get sued for overbilling customers.

Another scenario this approach protects against is a denial of service (DoS) attack. In these attacks, someone tries to flood your server with an unusually high number of requests. These requests overcome the processing power on your servers and the whole system grinds to a halt. If Azure automatically scaled up in this scenario, you would come in on Monday the day after an attack and find 5,000 instances running in production. You would get enough mileage points on your credit card to fly to the moon and back for free, but you probably wouldn't be too happy about it.

Microsoft has given us the tools to manage scaling ourselves. We can adjust the target number of instances at any time in a variety of ways. All we have to add is the logic we want to use to determine what *busy* means for us, and how we want to handle both the busy states and the slow states.

We'll look at some approaches for how to scale your Azure environment in the next section.

18.6 Better together for scaling

Everyone expects that they'll be able to dynamically scale their service in Azure. Dynamic scaling is possible, but it requires some heavy lifting on the developer's part. Over time, vendors will provide this as a service on top of Azure. In the meantime, you'll want to provide some sort of control over the amount of resources allocated to your service.

In this section, we're going to follow the same model that our homes use for heating and cooling. Our homes are driven by a sensor that detects a healthy condition (is the temperature in a pleasant range?), a mechanism to change that temperature, and some simple rules that keep the heater from running for 24 hours straight. We can take this approach and apply it to a cloud service. We'll instrument the cloud service

with the diagnostic engine, use the management API to control the infrastructure, and provide some code to control how all that works. You want to respond to events and keep your system healthy so you don't come in on Monday to find that you have 1,500 instances running in the cloud.

In keeping with our heating and cooling metaphor, let's start with the thermostat.

18.6.1 The thermostat

The thermostat in your home is a simple component of a common control system. Other examples include the cruise control in your car, the autopilot in a plane, and many manufacturing systems. Each of these systems has three components.

The first component is the system itself: the car, the plane, the furnace in the house. In Azure, the system is the service you're running. The system needs to have inputs to be able to control important aspects of itself. In the furnace example, you can send it a turn-on signal or a turn-off signal. These signals cause it to generate more heat or to stop generating heat.

The second component is the measurement or input device. This device measures the control aspects of the system. In our house example, it's the thermometer in the thermostat. In Azure, the measurement might be any number of things. A thermostat uses a simple dial to determine what the desired temperature is, as shown in figure 18.15.

The hard part for systems in Azure is deciding what *busy* is for the system. There's not a simple dial, but likely an amalgam of several inputs; perhaps the depth of a messaging queue, the number of pending requests in the IIS queue, and the running average of response time for each web request. Every measurement point you want to monitor to help decide what *busy* means needs to be something you can measure across all your instances.

Busy is sometimes represented as an absolute measure. For example, you could define a concrete amount of time a response is allowed to take under normal conditions. The system is either beneath or above that allotted time. Some definitions for a busy state are relative in nature. Saying that your system is busy whenever there are more than 50 messages in the queue won't work very long. You might instead want to

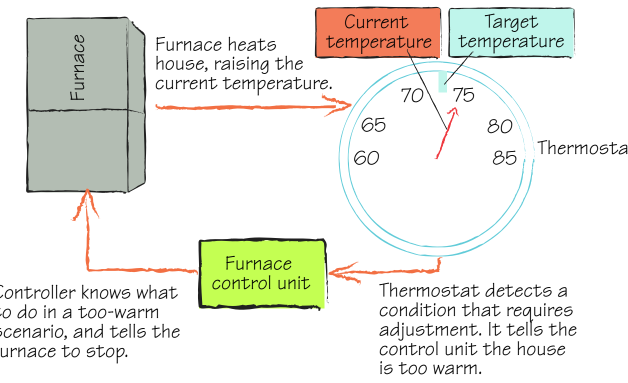


Figure 18.15 A control system you might have in your house. The furnace is the system you want to control. The thermometer measures the temperature of the house and provides feedback on the performance of the system to the controller. The controller is the electronics in your thermostat, which tells the furnace to turn on or off based on the input from the thermostat.

measure 50 messages per instance. Then, if you have five instances, a scenario that has 200 messages in the queue is OK. In this way, you can scale up the definition of *busy* as you scale up the amount of available resources.

The last component is the control logic itself. This is the piece that determines whether anything needs to be done, and how it should be done.

18.6.2 The control system

Your control system has only a few things that it can do with regard to managing performance or scale. In general, the only things the control system can do is add or remove instances of roles. That's about it.

Of course, there are plenty of scale patterns you can implement, and some will help prevent a dramatic scale failure from happening. You'll want to look into shunting, bulkheads, and partitioning. Just look in one of those enterprise patterns books on your bookshelf.

You need to instrument your control system yourself so you know what decisions it's making and how. You want to be able to figure out what went wrong when your service lumbers out of control and eats a village.

The control system can run as a simple role in Azure or as an on-premises application. You might assume an on-premise is a better solution, but remember that you'll need the input of the diagnostic logs; you'll have to download them all the time to make decisions. Having the control for the system running in the cloud puts the code near the data, which makes it both faster and cheaper.

18.6.3 Risks and managing them

A lot of risk is associated with implementing an auto-scaling component for your service; it's not trivial. You'll have to take into consideration a lot of issues and complexities. If things go haywire, they can go haywire badly.

On one hand, you could end up with a large Azure bill if your code goes crazy and spins up 400 instances. On the other hand, if it fails to work properly, you'll end up not responding to a busy state at all, leading to lost orders and unhappy users. Scaling requires a fine balance, and you'll want some protective measures in place.

In your logic, make sure you have an absolute upper boundary in place. No matter what's happening in the feedback system, the scaler won't go above this boundary. If the scaler reaches its ceiling, it should call a human and ask for help (by sending an email or a text message). You should set this boundary to something that's high enough to handle expected spikes (the big spike at the end of month), plus 15 percent for a buffer. In addition to this ceiling, set a floor to the scale value. You might want to make sure you always have two instances for reliability. Of course, some applications are OK with just a single instance, and others are OK to be completely shut down if there isn't any load.

At some point, the spike that caused the controller to create all these instances will pass. After it does, be sure that your controller starts shedding instances to bring the amount of resources deployed back into an acceptable range for the current load.

Pick an algorithm that matches how your load tends to fall off. If it tends to fall very quickly in a steep spike, then you should use an aggressive backoff strategy. In this case, you could use a halving technique in which for each polling cycle that the controller deems is excessive, it cuts the amount of resources by half (going from 32 instances to 16, for example).

In other services, you'll need a slower backoff process, dropping only one instance every time the measure drops by a certain percentage. Do extensive testing on the behavior of your controller to make sure it's working under stressful situations the way you want it to.

Another risk is that the controller might flood the channel with conflicting messages. If the polling cycle is too fast and the traffic too unpredictable, you might end up sending conflicting messages through the channel to the service. If you send a message to add an instance and then immediately follow it with a message to shed an instance, you'll end up thrashing your infrastructure. You also don't want to accidentally send a message to add an instance several times when you want only one net new instance started. To avoid this problem, make sure your controller is stateful, tracking the commands it has sent and whether those commands have been executed yet. You might even want to suspend all polling until the chosen action is completed.

No matter how clever you get with your controlling logic, make sure that you always include wetware somehow. The controller should always have a way to notify a human as to its behavior. If you run into the "we accidentally started up 400 instances last night" problem, you'll likely have a "don't have a job on Monday" problem.

We strongly recommend that you initially build tools that help you watch performance and easily maintain it. Keep the decision to add or remove instances in the control of humans, at least until you have an absolute understanding of how the system responds to stress, and how it responds to increases and decreases in resources.

The cloud does a great job of abstracting away the need to manage the platform, but you still need to manage the application.

18.6.4 Managing service health

Managing service health is critical to your system. Just because the system is running in the cloud doesn't mean there aren't failures and that your system doesn't need to be managed. You still need to manage a system in the cloud, and you need to take into consideration all the aspects that you consider when the system is running on-premises. The cloud doesn't fix problems in a bad system; it makes those problems more obvious.

As you're building your system, think about how you're going to manage disaster recovery, backups, and the ongoing health of the system. The system will be reliable within the Azure data center where the FC can monitor it, but that doesn't protect you against the worst scenario: that data center gets wiped out. In some scenarios, it might be OK to not manage the slight risk of a whole data center disappearing. On the other hand, many companies spend a lot of money running duplicate data centers that aren't near one another, just in case. You need to think about this. If you need to reduce the risk of depending on one data center, then you have a few options. Just to

be clear: the FC will manage the state of your system in a data center, but it won't, at this time, manage it across data centers. We think that in the fullness of time, the fabric will be that powerful, but at that time it's likely to be renamed to Microsoft SkyNet.

If the loss to your business isn't likely to be great, and you can deal with a few hours downtime, you can simply plan on redeploying to another data center in the event of a catastrophe. You need to keep a copy of the production bits and service configuration handy. You also need a backup of any data in the cloud. With these in hand, you could completely redeploy to another Azure data center in a matter of minutes to hours (depending on the amount of data that needs to be uploaded).

If you need to minimize downtime, you can run two copies of your service in Azure. Set each copy with a different geographic affinity in the portal. Azure is then forced to run each copy in a different data center. Perhaps the first copy is running in the Chicago data center, and the second is running in the Southwest data center. You could then use a DNS server that is geo-aware, and have it route users to each system, based on their location. In this situation, you'd have to replicate your data across the two systems. One way to do so would be to run them in complete separation, if your business processes can handle that. Then, once a night, you could merge the data sets with a background operation.

In all of these situations, you need to be able to understand the health of your system. One way to do that is to use the management APIs to understand the load on your system, and report the status in a recurring manner.

18.7 Summary

Although this chapter might have seemed like two disparate topics jammed together in one chapter to save space, at this point, you should understand how diagnostics, service management, and good service health instrumentation is important to running a healthy service in the cloud.

We dug deep into the diagnostics API, showing how you collect all sorts of data from many different instances, and merge it all into one place to make it easy to analyze the result. Each data source has its own configuration mechanism and destination. You can tell each data source to transfer its data on a schedule, or you can force an on-demand transfer.

The Azure platform also provides a rich service management API over REST. You can do almost anything over this API that you can do in the portal except for creating hosting and storage services. Although REST is awesome, some people will likely opt to use csmanage.exe, especially when they're automating deployment with a build system.

Finally, we merged diagnostics and the service management API together to discuss how an auto-scaling feature might look, and why Microsoft doesn't provide one out of the box. You now have all of the tools you need to rock out in the cloud, scale your system up and down, and know for sure what's happening with your services. Remember that as you move to the cloud, you want to stop managing servers and start managing services.

index

Symbols

.NET framework 150
.NET Services. *See* AppFabric
$queue browser 366

Numerics

32 bit 148
  native libraries 150
400 - Bad Request 370
64 bit 148, 150
  native libraries 150

A

A/B test 408
Access Control Service 381–384
  accepting SWT tokens 388–389
  actors 382
  authorization server 382
  authorization tokens 383–384
  configuring the namespace 392–396
  future of 402
  identity stores 381
  issuers 393
  management key 387
  rules 395–396
  scopes 394
  See also AppFabric ACS
  setting up 385–387
  SWT 384
  token policies 393
  trusted authority 382
access key 179
accessing
  development storage 171–173
  local storage 199
  runtime settings 101
AccountName parameter 45
accounts
  administrator 308
  SQL Azure user accounts 308–309
ACM.exe tool 392
ACS. *See* Access Control Service
actors 382
  client 382
  issuers 382
  protected resources 382
  relying party 382
adaptive streaming 225
adding certificates 91–92
AddMessage method 367
addresses 343
administration 54
administrator accounts 308
ADO.NET Data Services 253
  and context class 257
affinity 178, 436
affinity groups 41, 426
AJAX 114
  asynchronous requests 124
ALTER DATABASE command 301
Amazon Simple Storage System 161
Amazon SimpleDB database
  compared to Windows Azure 250
analytical and billing tools 39
analyzing databases 310
and query expression 289
ANSI nulls 310
APIs
  for BLOB data 160
  Service Management 421–432
app.config file 60, 400
AppFabric 4, 25–26, 342
  accepting SWT tokens 388–389
  Access Control Service 381–384
  ACM.exe tool 392
  AppFabric ACS 25
  AppFabric Service Bus 25–26
  audience 391
  checking tokens 389
  configuration browser 392
  creating a namespace 385
  future of 402
  identity stores 381
  in Visual Studio 381
  key services 380
  Service Bus 381, 397–399
AppFabric ACS 25
  introduction to 25
AppFabric Caching 25
AppFabric configuration browser 392
AppFabric Service Bus 25–26
AppFabric Service Workflow and Management 25
application servers 14
application state 348

applications
  adding Microsoft.Windows-Azure.ServiceRuntime assembly 79
  building 9–13
  building a media player 224
  building with Visual Studio 10–13
  communicating with third-party sites 217–223
  configuring build configuration 104
  creating 33–39
  creating with Visual Studio 267
  CSS and XHTML 34–36
  deploying 45–47, 427
  deploying with Azure portal 39–47
  fault tolerance 21
  flipping 42
  hosting 6–8
  hosting in BLOB storage 215–223
  logging 43
  migrating to SQL Azure 309–311
  migrating to Windows Azure 80
  moving to production 47
  multiple instances 119–120
  packaging 45–47
  running 6–8
  running across multiple servers 7
  running in development fabric 38
  running locally 37
  scaling 117–118
  scaling dynamically 430–432
  scaling out 117
  scaling out automatically 117
  scaling up 118
  server applications 7
  setting up storage environment 43–45
  Silverlight Spectrum emulator 215–217
  simulating extreme load 115
  state 348
  supported types 9
  tearing down 429
  testing under load 123
  tweaking configuration settings 104
  types 6
  under extreme load 116
  under normal load 114
  using local storage 32, 88
  web search application 219–223
Application_Start event 348
ApproximateMessageCount property 371
arbitrary diagnostics sources 416
arbitrary files 407
architecture
  flexibility 352
  n-tier model 398
Arguments property 148
arrays
  characters arrays 338
AsBytes property 368
ASP.NET 143
  authorization 198
  caching 135–137
  integrating BLOB data 195–199
  integrating private content 196–199
  maximum request length 191
  persisting data 127
ASP.NET 3.5 SP1 9
ASP.NET applications
  configuring runtime settings 99
  optimizing delete performance in 278–279
ASP.NET MVC2 Web Role template 30
ASP.NET Web Role template 30
assemblies
  Diagnostics 34
  Microsoft.WindowsAzure.ServiceRuntime 79–81, 107
  Microsoft.WindowsAzure.StorageClient 252
  ServiceRuntime 34
  StorageClient 34
  System.Data.Services 252
  System.Data.Services.Client 252
AsString property 368
asymmetric queues 373
asynchronous 373
asynchronous calls
  performance 320
asynchronous work 347
AtomPub standard 267, 270, 286
  creating tables using 271
  verbosity 287
attributes
  enableNativeCodeExecution 141
  enableNativeExecution 86
  LoadedBehavior 225
audience 390–391
authenticating
  requests 273–275
authentication 400, 425
  shared key 273
  Shared Key Lite 274
authorization 198
  accepting SWT tokens 388–389
  attaching tokens 392
  audience 391
  checking tokens 389
  claims 384
  management key 387
  OAuth standard 383
  setting up Access Control Service 385–387
  SWT 384
  tokens 383–384
  validating key 187
Authorization header
  shared key authentication 274
  Shared Key Lite authentication 274
authorization header 363, 388
authorization key
  validating 187
Authorization request header 187
authorization server 382
authorization tokens 383–384
automatic scaling 432
automating
  deployment 424–430
automation
  lights-out operations 54
Azure Diagnostics 405
Azure Drive 162, 169
Azure portal
  affinity groups 41
  creating namepaces 385
  deploying with 39–47
  importing certificates 423–424
  introduction 40
  managing certificates 91

Azure portal *(continued)*
- modifying runtime settings 102–103
- revoking certificates 423–424
- rolling upgrade 67
- setting up domains 165
- setting up services 41
- setting up SQL Azure 297–300
- setting up storage account 164
- static upgrades 66
- subscription ID 426
- uploading new service configuration file 102
- validating domains 165

Azure Queue 338
Azure storage 376
Azure Storage Explorer 253

B

backend server 346
background process 146
background threads 348
BackOffPace property 366
BACKUP command 312
backups 305–307
bandwidth 301
BASIC 216
Batch parameter 283
batch processing 377
batching data 281
BCP tool 306
bin folder 29
binary files. *See* BLOB data
binding 343, 401
- data in code-behind 36

BizTalk Services. *See* AppFabric
Blank Cloud Service template 30
Blob Browser tool 212, 230
BLOB data 16, 32, 148–149, 160–179, 354
- and custom metadata 203–204
- and local caching 199
- APIs 160
- as media server 223–232
- as origin server 235
- associating domain with 164
- Azure Drive 162
- blocks 17
- checking last modified time 202
- checking properties 201–202
- checking properties with StorageClient library 202
- consistency 161
- container 163
- containers 166, 169–178
- copying 204–206
- deleting 192
- disk storage 161
- downloading 193–195
- downloading from private containers 193–195
- downloading from public containers 193
- endpoint 171
- hosting applications 215–223
- hosting Silverlight Spectrum emulator 215
- hosting static websites 209–210
- in public container 196
- integrating with ASP.NET 195–199
- listing 189
- listing with REST 182–185
- managing 162
- managing with StorageClient library 188–193
- media player 224
- MIME type 193, 195, 213
- optimizing performance 196
- performance 162
- publishing websites to 212–215
- replication 161
- scalability 160
- setting shared access permissions 206
- setting up domains 165
- setting up storage account 164
- snapshotting 206
- splitting into blocks 192
- storage account 163
- table-driven 195
- uploading 191–192
- versus Shared Access Signatures 198
- versus traditional approaches 157
- versus web roles 210
- with content delivery networks 232–236
- with local storage 199–204

BLOB storage. *See* BLOB data
blobClient class
- GetContainerReference method 190

BlobProperties class
- BlobType property 202
- CacheControl property 202
- ContentEncoding property 202
- ContentLanguage property 202
- ContentMD5 property 202
- ContentType property 202
- Etag property 202
- LastModifiedUtc property 202
- LeaseStatus property 202
- Length property 202

BlobType property 202
Boolean logic 291
browser sessions 127
buffers 338
building
- applications 9–13

Bulk Copy Program. *See* BCP tool
bulkheads 434
business logic 338
busy state 433
byte arrays 367

C

C# 11
- representing entities 241

CacheControl property 202
caches. *See* caching
caching 135–138, 359
- ASP.NET 4.0 137
- ASP.NET cache 135
- distributed 136–137
- empty caches 322
- in-process 135
- local 199
- Memcached 136–137
- populating cache 321
- static data 321–322
- Windows Server AppFabric Caching 137

canonicalization 187
CAS 140
Cascading Style Sheets. *See* CSS
Cassini web server 31
CBlox 53
CDNs. *See* content delivery networks
certificate authorities 422

certificate thumbprints 98
certificates 90–93
  adding 91–92
  adding to development fabric 91
  adding to production fabric 91
  and service definition file 92
  certificate authorities 422
  configuring 97–98
  configuring HTTPS endpoint 92
  generating 90
  importing 423–424
  makecert tool 90
  management certificates 423
  revoking 423–424
  self-signed 423
  thumbprints 98
CGI 141–146
  enabling in Windows 7 142
  enabling in Windows Server 2008 142
  enabling in Windows Vista 142
CGI web role 144
CGI Web Role template 30
Changed event 103
changing
  configuration 430–432
  service configuration file 430
changing configuration 430
Changing event 103
character arrays 338
Chris Hay. *See* Blob Browser tool
chunking 223–225
  downloading chunks 227–228
  sharding 302
  with Silverlight 228–232
claim set 384, 390
claims 384
  CBAC 384
  claim set 384, 390
claims based access control 384
classes
  CloudBlobClient 174
  CloudBlobContainer 175
  CloudBlobDirectory 214
  CloudQueueClient 364
  CloudQueueMessage 367
  CloudStoragAccount 174
  CloudStorageAccount 172, 364
  ConfigurationManager 61
  context class 257
  CrashDumps 407
  DataServicesContext 257
  DeploymentDiagnostic-Manager 416
  DirectoryConfiguration 407, 418
  HostedService 425
  LocalResource 354
  OnDemandTransferOptions 420
  PerformanceCounter-Configuration 408, 412
  RoleEntryPoint 421
  RoleEnvironment 80, 109, 345
  RoleInstanceDiagnostic-Manager 415
  RoleInstanceDiagnostic-ManagersForRole 420
  RoleManager 348
  ServiceAuthorizationManager 388
  ServiceHost 342
  WebClient 391
  WebHttpBinding 387
  WebOperationContext 388, 392
  WebRequest 424
  WebServiceHost 387
  WindowsEventLogsBuffer-Configuration 408
cleanOnRoleRecyle parameter 354
Clear method 364
clients 382
  connecting to each other 398
  sending tokens 390–391
cloud computing
  abstraction 56
  affordability 19
  and SQL Server 23–24
  and SQL Server. *See* SQL Azure
  as a utility service 19
  capacity 21
  cloud operating system 5
  conceptualizing 37
  diagnostics 405–421
  differences from local environment 32
  history 51
  how it works 56–57
  identity 381
  introduction 4–8
  metered services 4
  reasons for using 18
  scalability 19
  Service Bus 398–399
  shared storage 16
  storing data 15–18
  troubleshooting 406
  using PHP 143–146
cloud operating system 5
Cloud Service templates 29
  ASP.NET MVC2 Web Role template 30
  ASP.NET Web Role template 30
  Blank Cloud Service template 30
  CGI Web Role template 30
  WCF Service Web Role template 30
  Worker Role template 30
CloudBlobClient class 174
  GetBlobReference method 174
  GetContainerReference method 174
  ListBlobsWithPrefix method 174
  ListContainers method 174
CloudBlobContainer class 175
  Create method 175
  Delete method 175
  ListBlobs method 175
CloudBlobDirectory class 214
CloudQueue 365
CloudQueue class
  AddMessage method 367
  Clear method 364
  Create method 364
  CreateIfNotExist method 364
  Delete method 364
  DeleteMessage method 368
  GetMessages method 368
  PeekMessage method 368
  PeekMessages method 368
  SetMetadata method 364
CloudQueueClient class 364
  ListQueues method 364
CloudQueueMessage class 367
CloudStorageAccount class 172, 174, 364
  FromConfigurationSetting method 364
  SetConfigurationSetting-Publisher method 177
CloudTableClient class
  CreateTableIfNotExist method 255
  DeleteTable method 269

CLR 140–141, 313
clustered indexes 312
CNAME entries 166
code
  centralizing 61
  file-reading 61
code access security. *See* CAS
code-behind 36, 120, 190
CodePlex 310
command line
  FFmpeg 149
commanding 399
commands
  ALTER DATABASE 301
  BACKUP 312
  CREATE DATABASE 299, 312
  CREATE TABLE 310
  create tokenpolicy 394
  create-deployment 427
  GRANT 309
  makecert 423
  publish 45
  REVOKE 309
  swap-deployment 429
  USE 311
common code base 105–109
common scenarios
  far-data scenarios 312
  near-data scenarios 313
communicating
  across roles 344–345
  with worker roles 338–345
compatibility 311
concurrency 340, 372
configuration 59–61, 407
  changing at runtime 61
  roles 62
  separation of concerns 61
configuration managers 420
configuration settings
  defining the interface 107
  pluggable 107–109
  sharing 106
  tracking 103
  tweaking 104
  using inversion of control pattern 107–109
ConfigurationChangePollInterval property 415
ConfigurationManager class 61
ConfigurationSettings node 60
ConfigurationSettings.aspx file 99
configuring
  application build configuration 104
  certificates 97–98
  database connection strings 104
  diagnostic agents 409–419
  diagnostic hosts 411–416
  endpoints 105
  roles from outside 415
  runtime settings 98–101
  service definition file 100
  service model 37
  the ACS namespace 392–396
confirmation message 374
connecting
  to databases 299–300
  to Service Bus 400
  to services 401
connection strings 171, 299, 303, 311, 413
  UseDevelopmentStorage option 409
console applications
  creating tables using REST 272
consumer 358
containers
  and REST 182–185
  creating 173
  default permissions 175
  deleting 177
  downloading BLOBs from 193–195
  for BLOB data 163, 166
  full public read acess 167
  listing 175, 185
  listing with Service Management API 424–425
  public 196
  public read-only access 167
  setting shared access permissions 207
  when to use 169–178
content delivery networks 225, 232–236
  advantages of 233–234
  defined 232
  edge servers 232
  Windows Azure CDN 234–236
ContentEncoding property 202
ContentLanguage property 202
ContentMD5 property 202
ContentType property 202
context class 257
  adding entities to 258–259
ContinueOnError parameter 283
contracts 343
control logic 434
  upper boundary 434
control system 434
controllers
  risks 435
  stateful 435
controlling
  access 381–384
controls
  MediaElement 225
conversion rate 346
copying
  BLOBs 204–206
  with StorageClient library 206
cost 314
  in-memory joins 328
  SQL Azure 318
  Table service 319
count 372
CounterSpecifier property 412
crash dumps 407, 417
CrashDumps class 407
CREATE DATABASE command 299, 312
Create method 175, 364
CREATE TABLE command 310
create tokenpolicy command 394
create-deployment command 427
CreateHttpRequest method 268
CreateIfNotExist method 364–365
CreateTableIfNotExist method 255
creating
  an AppFabric namespace 385
  context class 257
  databases 298, 304
  queues 365
  queues at startup 376
  simple web page 33–39
  static websites 210–212
  tables 253–255
  tables using AtomPub standard 271
  tables using REST 271–273
  user accounts 300, 308–309
critical 406
cross-domain policy file 217
CRUD operations
  with Table service 256–264

csmanage tool 426
  changing configuration 430
CSmonitor tool 29
CSpack tool 29, 45
CSrun tool 29
CSS 33–36
CurrentRoleInstance property 343

D

DAS. *See* direct-attached storage
data
  and firewalls 307
  Azure Drive 162
  backing up 305–307
  batching 281
  binding in code-behind 36
  BLOB data 16, 160–163
  blocks 17
  caching 135–138, 321–322
  direct-attached storage 159
  drop-down lists 317
  duplicating versus joining 329
  filtering 414
  filtering with LINQ 290–291
  filtering with REST 288–290
  infrequently changed 329–331
  joining 329–331
  joining uncached data 330–331
  merging 279
  merging or updating with REST 281
  MIME type 193, 195
  moving 305–307
  network-attached storage 159
  on-demand transfers 420–421
  paging 294
  partitioning 249–252
  partitioning with SQL Azure 301–302
  peer-to-peer 158
  persisting in ASP.NET 127
  querying 284–294
  referential integrity 326
  replication 158, 250
  retrieving 324
  retrieving from Memcached 137
  scheduled transfers 419
  selecting using LINQ syntax 292–294
  serialized binary 367
  setting up storage environment 43–45
  sharding 302–303
  shared network drive 157
  shared storage 16
  sharing across machines 156
  sharing with DFS 158
  sharing with SQL Server 157
  shopping cart example 323–331
  single instruction, multiple data 371
  static reference data 316–322
  storage area networks 159
  storing 323–329
  storing in Memcached 137
  storing in tables 18
  storing in the cloud 15–18
  storing session data 128–130
  synchronizing 330
  transferring diagnostic data 419–421
  validating 350
  working with in SQL Azure 306
data access layer 303
data centers
  administration 54
  CBlox 53
  density 53
  edge data centers 54
  edge servers 232
  first generation 52
  Generation 2 53
  Generation 3 53–54
  Generation 4 55
  hardware 55
  history 52
  lights-out operations 54
  managing 21
  modular 54
  power and cooling costs 52
  reducing carbon footprint 52
  System Center Operations Manager 54
data model
  splitting across multiple servers 325
data replication 250
data sources
  arbitrary files 407
  arbitrary sources 416, 418
  common 407
  crash dumps 407, 417
  diagnostic 407
  diagnostic infrastructure logs 408
  failed request logs 418
  IIS failed request logs 408
  IIS logs 408
  performance counters 408
  trace logs 408
  Windows event logs 408, 418
data synchronization 330
database connection strings
  configuring 104
database size limits 300–303
  1 GB 300
  10 GB 301
databases
  allowing Azure connections 300
  analyzing 310
  configuring connection strings 104
  connecting to 299–300
  connection string 299, 311
  creating 298, 304
  dbmanager role 309
  loginmanager role 308
  managing 305–309
  mapping entities to 243
  scaling issues 239
  size limits 300–303
DataServicesContext class 257
dbmanager role 309
DDOS. *See* distributed denial-of-service attacks
debugging 405–406
decoupling 377
  with messaging 358–363
defining
  endpoints 82–85, 341
  local storage 199
  services 81–90
Delete method 175, 364
DELETE verb 269
DeleteMessage method 368
DeleteTable method 269
deleting
  BLOB data 192
  containers 177
  entities 261–262, 277–279
  messages 368
  optimizing delete performance 278–279
  queues 366
  tables using REST 269
denial of service attacks. *See* distributed denial-of-service attacks

dependency injection 352
deploying
  applications 45–47, 427
  new service configuration file 431
  tearing down 429
  with Azure portal 39–47
deployment
  automating 424–430
DeploymentDiagnosticManager class 416
design patterns
  Itinerary pattern 351
  message processing 371–378
  Singleton pattern 348
  Unit of Work pattern 276
developing
  common code base 105–109
  with Table service 252–255
development fabric 9, 56
  adding certificates 91
  defined 14
  development fabric service 31
  diagnostics 409
  Fabric Controller 14, 57–59
  load balancer 120
  load balancer process 122
  running applications 38
  running web pages 12
  selecting ports 84
  UI 39, 411
development storage 167–169, 171–173
  communicating with 173
  installation issues 168
  listing containers 185
  listing tables 266–269
  service definition file settings 172
  SQL Server backing store 168
  starting and stopping manually 169
  switching to live storage 178–179
  user interfaces 168
DevelopmentStorageAccount property 172
DFS
  sharing data with 158
DFUI tool 29
diagnostic agents
  buffering 411
  configuring 409–419
  configuring within a role 411–413
  default configuration 411
  diagnostic infrastructure logs 411
  GAS process 411
  IIS logs 411
  listeners 409
  See also logging
  trace listeners 409
diagnostic hosts
  configuring 411–416
diagnostic infrastructure logs 408, 411
diagnostic managers 415
diagnostic sources
  managing 407–408
DiagnosticMonitorConfiguration class
  ConfigurationChangePollInterval property 415
diagnostics 405–421
  arbitrary sources 416
  Azure Diagnostics 405
  configuring diagnostic agent 409–419
  configuring the host 411–416
  crash dumps 407
  default configuration 411
  diagnostic infrastructure logs 411
  diagnostic managers 415
  exceptions 407
  GAS process 411
  IIS logs 411
  managing sources 407–408
  MonAgentHost tool 406
  on-demand transfers 420–421
  scheduled transfers 419
  transferring diagnostic data 419–421
  Visual Studio 409
diagnostics agents
  configuring roles from outside 415
  filtering data 414
Diagnostics assembly 34
diagnostics. *See* system logs
DiagnosticsConfigurationString 413
direct-attached storage 159
directory structure 214
DirectoryConfiguration class 407, 418
disaster recovery 306
distributed denial-of-service attacks 83, 432
Distributed File System. *See* DFS
distributed transactions 312
DLLs
  kernel32.dll 150
  Microsoft.WindowsAzure.StorageClient.dll 171
  with P/Invoke 150
DNS 41
DoesQueueExist method 365
domains
  associating with BLOB data 164
  CNAME entries 166
  setting up 165
  validating 165
DownloadByteArray method 195
downloading
  BLOB data 193–195
  BLOBs from private containers 193–195
  BLOBs from public containers 193
  progressive. *See* chunking
DownloadText method 195
DownloadToFile method 148, 195
drawbacks
  queuing 340
drivers
  and Fabric Controller 58
drop-down lists 317
  populating 321
DSinit tool 29
durability 340, 368
dynamic data
  joining with infrequently changed data 329–331
  shopping cart example 323–329
  storing with reference data 323–329
dynamic ports 344
dynamic queues 376

E

echo 146
Eclipse 29, 140
edge data centers 54
edge servers 232
EDI 376
elements
  traceFailedRequests 408
EnableConnection method 417

EnableConnectionToDirectory method 417
enableNativeCodeExecution attribute 141
enableNativeExecution attribute 86
enabling
- native execution 119
- Windows Azure CDN 235

endpoints
- BLOB data 171
- configuring 105
- configuring for certificates 92
- defining 82–85, 341
- external 105
- for Table service 266
- input 342, 345
- internal 344–345
- service endpoints 339, 343
- worker role endpoints 85
- worker roles 341

enterprise service bus. *See* Service Bus
entities
- adding to context class 258–259
- and context class 257
- challenges of extending definitions 248
- CRUD operations with REST 275–281
- defining 253
- deleting 261–262, 277–279
- dissimilar 247–249
- entity group transactions 282–284
- extending definitions 246
- inserting 275–277
- inserting with REST 276
- listing 260–261
- mapping to databases 243
- modifying definitions 244–245
- modifying to use Table service 244–249
- optimizing delete performance 278–279
- querying with REST 289
- representing in C# 241
- representing in Table service 245–247
- retrieving using REST 285
- returning with REST 289
- shopping cart 327
- SizeType entity 318
- SQL Server versus Table service 245
- storing in SQL Server 242
- updating 263–264, 279–281

entity definitions 244–245, 253
- challenges of extending 248
- extending 246

entity group transactions 282–284
enumeration types
- LogLevel 410

eq query expression 289
equality comparisons 290
Etag property 202
events
- Application_Start 348
- Changed 103
- Changing 103
- OnStart 411
- RoleEnvironmentChanging 430
- Session_Start 348
- Stopping 89

eventually consistent scenarios 372
exceptions 370, 407
- Out of Memory 131

ExecuteCommand method 147, 149
expired tokens 389
Extensible Hypertext Markup Language. *See* XHTML
external process 146–149

F

Fabric Controller 14, 42, 57–59, 347
- and service model 59–62
- and services 58
- as kernel 15, 57
- defined 15
- driver model 58
- fault domains 63
- how it works 58
- instance management 59
- resource allocation 58
- service configuration file 94–103
- starting a site 428
- update domains 63

fabric. *See* development fabric; production fabric; and storage fabric
factory 353
failed instances 340
failed request logs 418
far-data scenarios 312
FastCGI 140–146
- configuring Windows Azure for 142–143
- enabling 142

FastCgiModule 143
fault domains 63
fault tolerance 21
- separate server racks 21

federated identity 382
FFmpeg 146–149
- at command line 149

FIFO. *See* First in First Out
FileName property 148
file-reading code
- centralizing 61

files
- app.config 60, 400
- ConfigurationSettings.aspx 99
- cross-domain policy file 217
- directory structure 214
- global.asax 348
- Microsoft.WindowsAzure.StorageClient.dll 171
- MIME type 193, 195
- php-cgi.exe 145
- service configuration file 59, 94–103, 342, 430
- service definition file 59, 62, 81–90, 354
- ServiceConfiguration.cscfg 38, 45
- ServiceDefinition.csdef 38
- temporary 353
- web.config 60, 99, 143, 409
- web.roleConfig 142, 145
- WebRole.cs 409

filtering
- data 414
- web role traffic 82
- with LINQ 290–291
- with REST 288–290

FindByThumbprint method 98
FindFirstFile method 151
- lpFileName parameter 150
- lpFindFileData parameter 150

FindNextFile method 150–151
firewall_rules table 307
firewalls 307
- firewall_rules table 307
- managing with code 307

First in First Out 359
flipping 42
  VIP swap 42
flow diagram 350
folders
  bin 29
  inc 29
  Roles folder 34
forklift upgrades. *See* static upgrades
FromConfigurationSetting method 173, 364
frontend
  offloading work 345–346
full trust 139, 141

G

garbage collection 361
ge query expression 289
Generation 2 data centers 53
Generation 3 data centers 53–54
  administration 54
  hardware 55
  lights-out operations 54
  System Center Operations Manager 54
Generation 4 data centers 55
GET verb 201, 267, 365, 373
GetBlobReference method 174
GetConfigurationSettingValue method 106, 109, 416
GetContainerReference method 174, 190
GetLocalResource method 88, 354
  MaximumSizeInMegabytes property 88
  Name property 88
  RootPath property 88
GetMaximumSizeInMegabytes property 88
GetMessage method 368–369
GetTokenFromACS method 391
global.asax file 348
GRANT command 309
grids
  optimizing delete performance 278
GridView component 175
gt query expression 289
guaranteed ordered delivery 376

H

handlers 143
HEAD verb 201–203
headers
  Authorization header 274, 363, 388
  If-Match 281
  version header 425
healthy services 404
heap tables 312
hosted services 426
HostedService class 425
hosting
  applications in BLOB storage 215–223
  Silverlight Spectrum emulator 215–217
  static websites 209–210
HSS Web App role 39
HTTP Get requests
  for BLOB data 163
HTTP handlers 196–198
  and local storage 200
  checking last modified time 202
  creating 197
  registering 198
HTTP protocol 127, 160
  and web roles 82
  handlers 197–198
  HEAD verb 201–202
  internal endpoints 85
HTTPS protocol 32, 422
  and web roles 82
Hyper-V 68–72
  booting servers 69–71
  core-and-socket parking 68
  optimization for Windows Azure 69

I

ICanHazCheeseburger 146
idempotent 370
identity 381
  federated 382
  identity fishbowl 382
  identity stores 381
  web identity 402
identity fishbowl 382
identity stores 381
If-Match header 281
IIS 143, 380
IIS 7.0 log 411
IIS failed request logs 408
IIS logs 408
IIS. *See* Internet Information Services
images 71
  updating 71
inbound queue 373
inc folder 29
indexes
  clustered 312
indexing
  and partitioning 251
indirection 398
infinite loops 374
  ceiling 374
  delay 374
  floor 375
  sleeping 374
infrastructure layer 305
Initialize method 109
in-memory joins 328
input endpoints 342, 345
installing
  Windows Azure SDK 9
InstanceEndpoints property 343
instances
  failed 340
  multiple 119–120
  setting number of 96
integrating
  BLOBs with ASP.NET 195–199
  private content 196–199
interfaces
  defining 107
  implementing in web applications 107
  implementing in web roles 107
  IPrincipal 397
internal endpoints 344–345
Internet Information Services
  FastCGI 141–146
  hostable web core 76
  reconfiguring local instance 142
  running web roles 80
  viewing host process 75
inter-role communication 340, 344–345
inversion of control pattern
  configuration settings 107–109
  Unity Application Block framework 107–108

IoC pattern. *See* inversion of control pattern
IP addresses 308
IPrincipal interfaces 397
IsAvailable method 109
IsAvailable property 80, 106
isolated storage 231
issuers 382, 392–393
Itenerary pattern 351

J

job scheduler 336
joins 329–331
  client-side 330–331
  compared to duplication 329
  in-memory 328

K

kernel 15
kernel32.dll 150
  FindFirstFile method 150
  FindNextFile method 150
keys
  management key 387
  public 423
  signing 388
  symmetric 387

L

Last in Last out 359
Last-Modified tag 201
LastModifiedUtc property 202
le query expression 289
LeaseStatus property 202
least priveleged trust 140
legacy code 353
Length property 202
libraries
  storage client library 148
  StorageClient 170, 188–193
  WCF Data Services 270
LIFO. *See* Last in Last out
lights-out operations 54
limitations 311–312
links 37
LINQ
  filtering with 290–291
  querying with 288
  syntax 292–294
LINQ to DataSet 243
LINQ to Objects 291
LINQ2Entities 292
LINQ2SQL 292
ListBlobs method 175
ListBlobsWithPrefix method 174
ListContainers method 174
listen 400
listeners 409
listing
  entities 260–261
  queues 364
  services and containers with Service Management API 424–425
  services with REST 424
ListQueues method 364
live IDs 39
live storage
  access key 179
  switching from development storage 178–179
load balancer 13, 340, 344
  asynchronous AJAX requests 124
  balancing a multi-instance application 119–120
  development fabric 120
  how it works 118–127
  in live environment 124–127
  primary functions 14
  quirkiness 126
  requests 252
  simulating 120
  testing in live environment 126
  testing in staging environment 124–126
  testing with multiple browser instances 122
  Visual Studio Team System Web Load Tester 124
  WaWebHost service 120
LoadedBehavior attribute 225
local drives 353
local storage 353–355
  accessing 199
  and HTTP handlers 200
  benefits of 87
  configuring 87–90
  defining 199
  RequestRecycle method 89
  setting up 87, 353
  temporary files 353
  using 88
  with BLOB storage 199–204
LocalResource class 354
  RootPath property 354
LocalStorage tag 353
logging 43, 405
  diagnostic agent 43
  diagnostic infrastructure logs 411
  IIS 411
  LogLevel enumerated type 410
  requests 214
logical separation 347
login 308
loginmanager role 308
LogLevel enumeration type 410
long queues 377
loose coupling
  with messaging 358–363
lpFileName parameter 150
lpFindFileData parameter 150
lt query expression 289
Lucene.NET 331

M

magic string 171
makecert tool 90, 423
management account 297
management certificates 423
management key 387
managing
  BLOB storage 162
  caches 135–138
  databases 305–309
  diagnostic data sources 407
  diagnostic sources 407–408
  in-process sessions 130–132
  risk 434–435
  service health 435
  worker roles 352
managing with StorageClient library
  managing BLOB data 188–193
master database 298
MAXSIZE parameter 299
media server
  BLOB storage as 223–232
  streaming 223
MediaElement control 225
  LoadedBehavior attribute 225
mega data centers. *See* Generation 3 data centers
Memcached 136–137
  retrieving data from 137
  storing data 137

memory caching 135
memory consumption 131
memory leak 151
MERGE verb 281
message queues 17
message size 360
message visibility 369–371
  explained 369–370
  setting timeout 370
messages
  adding to queue 367
  asynchronous 373
  confirmation 374
  consumer 358
  content 361
  decoupling 358–363
  defined 360–361
  deleting 368
  failure 370
  getting 368
  guaranteed ordered delivery 376
  how messaging works 358–360
  idempotent code 370
  message size 360
  peeking at 367
  persisting 362
  poison messages 366
  pop receipt 369
  processing 371–378
  producer 358
  properties 361
  pull 338
  push 338
  queuing 339–340, 360
  setting timeout 370
  shared counters 371–372
  single instruction, multiple data 371
  stale 361
  state 350
  timeout 369
  truncated exponential backoff 374–375
  visibility 369–371
  work complete receipt 373
messaging. *See* messages
metadata
  and BLOB data 203–204
  queuing 362, 365
metered services 4
methods
  AddMethod 367
  Clear 364
  Create 175, 364
  CreateHttpRequest 268
  CreateIfNotExist 364–365
  CreateTableIfNotExist 255
  Delete 175, 364
  DeleteMessage 368
  DeleteTable 269
  DoesQueueExist 365
  DownloadByteArray 195
  DownloadText 195
  DownloadToFile 148, 195
  EnableConnection 417
  EnableConnectionToDirectory 417
  ExecuteCommand 147, 149
  FindByThumbprint 98
  FindFirstFile 150–151
  FindNextFile 150–151
  FromConfigurationSetting 173, 364
  GetBlobReference 174
  GetConfigurationSettingValue 106, 109, 416
  GetContainerReference 174, 190
  GetLocalResource 88, 354
  GetMessage 369
  GetMessages 368
  GetTokenFromACS 391
  Initialize 109
  IsAvailable 109
  ListBlobs 175
  ListBlobsWithPrefix 174
  ListContainers 174
  ListQueues 364
  OnStart 409
  OnStop 421
  Page_Load 177
  PeekMessage 368
  PeekMessages 368
  phpinfo 146
  RequestRecycle 89
  RetrieveApproximateMessage-Count 371
  Run 147, 338
  SetConfigurationSetting-Publisher 177
  SetMetadata 364–365
  SignRequest 186
  UploadBlobMetadata 204
  UploadFile 148, 192
  UploadFromStream 192
  UploadText 192
  UrlDecode 392
me-ware 364
MEX 343
Microsoft
  history 51
  Ray Ozzie 52
Microsoft Windows. *See* Windows
Microsoft.WindowsAzure name-space 34
Microsoft.WindowsAzure.Diag-nostics namespace 410
Microsoft.WindowsAzure.Ser-viceRuntime assembly 79–81, 107
  adding to applications 79
  RoleEnvironment class 80, 88, 109
Microsoft.WindowsAzure.Stor-ageClient assembly 252
Microsoft.WindowsAzure.Stor-ageClient namespace 185
  CloudBlobDirectory class 214
  DownloadByteArray method 195
  DownloadText method 195
  DownloadToFile method 195
  See also StorageClient library
  UploadFile method 192
  UploadFromStream method 192
  UploadText method 192
Microsoft.WindowsAzure.Stor-ageClient.dll 171
migrating 149
  SQL Azure Migration Wizard 310–311
  SQL Server to SQL Azure 307
  to SQL Azure 309–311
  traditional approach 309
MIME type 193, 195, 213
modifying
  entities for Table service 244–249
  entity definitions 244–245
modules
  FastCGIModule 143
MonAgentHost tool 406
multithreading 347

N

Name property 88
namespaces
  ACS namespace 392–396
  AppFabric 385
  issuers 392–393
  Microsoft.WindowsAzure 34

namespaces *(continued)*
  Microsoft.WindowsAzure.
    Diagnostics 410
  Microsoft.WindowsAzure.StorageClient 185
    See also StorageClient library
  rules 392, 395–396
  scopes 392, 394
  System.Runtime.InteropServices 150
  token policies 392–393
NAS. *See* network-attached storage
NAT 399
native code
  in processes 146–149
  with FastCGI 141–146
  with P/Invoke 149–152
native execution
  enabling 119
native libraries 149
  32-bit 150
  64 bit 150
ne query expression 289
near-data scenarios 313
network address translation 399
network-attached storage 159
nodes
  ConfigurationSettings 60
None parameter 283
not query expression 289
n-tier architecture 398
NULL 374

O

OAuth standard 383
  future of 402
  SWT 384
on-demand transfers 420–421
OnDemandTransferOptions
    class 420
OnStart event 411
OnStart method 409
OnStop method 421
open source 140, 146
or query expression 289
orchestration 397
origin server 233
  BLOB storage as 235
Out of Memory exception 131
outbound queue 373
overhead 350

P

P/Invoke 149–152
  and native libraries 150
  pinvoke.net website 150
packaging
  applications 45–47
Page_Load method 177
paging data 294
Parallel Extensions for .NET
    347
parameters
  AccountName 45
  Batch 283
  cleanOnRoleRecycle 354
  ContinueOnError 283
  lpFileName 150
  lpFindFileData 150
  MAXSIZE 299
  None 283
partial trust 140
partitioning 434
  across many servers 249–252
  and indexing 251
  querying across partitions 326
  row keys 251
  shopping cart example
    324–326
  splitting the data model 325
  storage account 249–250
  tables 250–252
  with SQL Azure 301–302
  with Table service 327
PartitionKey property 244, 253
password 385
pattern 378
PDC 297, 380
PDC09. *See* Windows Azure platform
peeking 367
PeekMessage method 368
PeekMessages method 368
performance 361, 376
  asynchronous calls 320
  queuing 358
  synchronous calls 320
performance counters 408, 412
PerformanceCounterConfiguration class 408, 412
permissions
  shared acess permissions 206
persisting
  messages 362
PHP 141, 146
  configuring Windows Azure
    for 142–143
  handlers 143
  in the cloud 143–146
  running in Windows Azure 142
php-cgi.exe file 145
phpinfo method 146
physical implementation 311
pig in a python 349, 352
pinvoke.net website 150
pluggable configuration settings
    107–109
  calling correct implementaion
    108
  defining interface 107
  implementing interface 107
podcasting 15, 17, 156, 170, 182,
    204
poison messages 366
polling 336
  floor 375
  infinite loops 374
  tables 340
pop receipt 369
populating
  drop-down lists 321
populating caches 321
ports 312
  dynamic 344
POST verb 271, 373
Preboot Execution Environment
    70
prefix queries 291
processes 149
  32-bit 148
  64-bit 148
  background 146
  external 146–149
  hosting process 75
  RDAgent 76
  small 349
  viewing 74
  WaWebHost 75
processing
  at runtime 351
  messages 371–378
  requests 187
processing engine 351
producer 358
production fabric
  adding certificates to 91
  replicas 304
progressive downloading. *See*
    chunking
properties
  ApproximateMessageCount
    371

properties *(continued)*
Arguments 148
AsBytes 368
AsString 368
BackOffPace 366
BlobType 202
CacheControl 202
ConfigurationChangePoll-Interval 415
ContentEncoding 202
ContentLanguage 202
ContentMD5 202
ContentType 202
CounterSpecifier 412
CurrentRoleInstance 343
DevelopmentStorageAccount 172
Etag 202
FileName 148
GetMaximumSizeInMegabytes 88
InstanceEndpoints 343
IsAvailable 80, 106
LastModifiedUtc 202
LeaseStatus 202
Length 202
Name 88
PartitionKey 244, 253
RefreshInterval 366
RootPath 88, 354
RowKey 244, 253
ScheduledTransferPeriod 419
Timestamp 244–245, 253
protected resources 382
protocols
HTTP 82, 127, 160
HTTPS 82, 422
TDS 24, 300, 311
WRAP 383
public keys 423
publish command 45
pulling messages 338
pushing messages 338
PUT verb 281
PXE 70
Python 139

Q

queries
query expressions 289
query expressions 289
and 289
eq 289
ge 289
gt 289
le 289
lt 289
ne 289
not 289
or 289
query shaping 292
querying
across partitions 326
Boolean logic 291
data 284–294
entities with REST 289
equality comparisons 290
LINQ syntax 292–294
LINQ to Objects 291
LINQ2Entities 292
LINQ2SQL 292
prefix queries 291
query shaping 292
range comparisons 290
SELECT statement 292
with LINQ 288
queue browser 363
queue depth 358
queue operations 364
Queue service 340
queuing
adding messages to queue 367
and REST 362
and state 350
and StorageClient library 362
asymmetric 373
basic operations 363–366
caching 359
concurrency 340
creating at startup 376
creating queues 365
decoupling 358–363
defined 361
delay 374
deleting messages 368
deleting queues 366
depth 358
drawbacks 340
durability 340
dynamic versus static 376
failure 370
First In First Out 359
floor 375
GET verb 365
getting messages 368
guaranteed ordered delivery 376
idempotent code 370
inbound 373
Last In Last Out 359
listing queues 364
long queues 377
loop ceiling 374
message content 361
message properties 361
messages 339–340, 360
metadata 362, 365
naming conventions 362
one-way 359
outbound 373
performance 358
persisting messages 362
polling 336
queue browser 363–366
recoverability 340
scaling dynamically 377
shared counters 371–372
single instruction, multiple data 371
strict ordered delivery 359
symmetric 373
truncated exponential backoff 374–375
work complete receipt 373

R

range comparisons 290
Range request header 226
Ray Ozzie 52
RDAgent process 76
recoverability 340
reducing software maintenance 22
redundancy 306, 361
reference data
caching 321
shopping cart example 323–329
static 316–322
storing with dynamic data 323–329
references
referential integrity 326
referential integrity 326
RefreshInterval property 366
relational databases
scaling issues 239
when to use 241
relay 400
relying party 382
repair and resubmit 351
replay attacks 393

replicas 304, 307
replication 250
  BLOB storage 161
  latency 161
  with DFS 158
representing
  static data 316–318
  static data in Table service 318–319
request headers 184, 362
  Authorization 187
  Range 226
  x-ms-copy-source 205
  x-ms-date 184, 186
  x-ms-version 184
request signing 268
RequestRecycle method 89
requests
  authenticating 185–188, 273–275
  headers 184
  load balancing 118–127, 252
  logging 214
  maximum request length 191
  naming convention 184
  private 185–188
  processing 187
  request signing 268
  signing 187
resources
  allocating 58
  dynamic allocation 430
  protected 382
response headers 203
  custom metadata 203
REST 160, 170, 368, 379, 392
  and queuing 362
  and storage account 266–273
  and Table service 252
  API 181
  authenticating requests 185–188
  changing configuration 430
  checking API version 184
  creating tables in console applications 272
  creating tables with 271–273
  CRUD operations 275–281
  deleting entities 277–279
  deleting tables with 269
  filtering data 288–290
  inserting entities 276
  listing BLOB data 182–185
  listing services 424
  listing tables 266–269
  merging or updating data 281
  query expressions 289
  querying data 284–294
  request headers 362
  retrieving entities 285
  returning single entity 289
  Service Management API 422
  shared key authentication 273
  Shared Key Lite authentication 274
  updating entities 279–281
REST API 181
  checking version 184
restarting
  WaWebHost service 123
retries 284
RetrieveApproximateMessageCount method 371
retrieving
  data 324
reversing strings 337
REVOKE command 309
risk 306, 434–435
  backoff process 435
  controllers 435
role based access control 384
RoleEntryPoint class 421
RoleEnvironment class 80, 109, 345
  accessing runtime settings 101
  callback to RoleEnvironment 109
  callback to RoleHost 109
  Changed event 103
  Changing event 103
  CurrentRoleInstance property 343
  GetConfigurationSettingValue method 106, 109
  GetLocalResource method 88
  Initialize method 109
  IsAvailable method 109
  IsAvailable property 80, 106
  RequestRecycle method 89
  Stopping event 89
RoleEnvironmentChanging event 430
RoleInstance class
  InstanceEndpoints property 343
RoleInstanceDiagnosticManager class 415
RoleInstanceDiagnosticManagersForRole class 420
RoleManager class 348
RoleRoot macro 143
roles 8, 62
  and shared storage 16
  configuring diagnostic agents 411–413
  configuring from outside 415
  configuring in Visual Studio 62
  configuring runtime settings 98–101
  creating 11
  dbmanager 309
  HSS Web App role 39
  inter-role communication 344–345
  loginmanager 308
  modifying runtime settings 102
  recycling 89
  service definition file 59
  virtual machine sizes 62
  web roles 72–76
  worker roles 14
Roles folder 34
rolling upgrades 66
  automatic 66
  manual 66
RootPath property 88, 354
routing logic 349, 352
row keys
  partitioning 251
RowKey property 244, 253
Ruby 139
rules 392, 395–396
Run method 147, 338
running
  web pages 12
runtime 351
  changing configuration 61
runtime settings
  accessing 101
  configuring 98–101
  for ASP.NET applications 99
  modifying in Azure portal 102–103
  tweaking 104

S

SAN 305
SAN. *See* storage area networks
SaveChangesOptions
  Batch parameter 283

SaveChangesOptions enumeration
 ContinueOnError parameter 283
 None parameter 283
scalability 346
 BLOB storage 160
 data sharding 302–303
 on-demand 19
 queuing 377
 SQL Azure 24
 traffic surges 19
 varied usage patterns 20
scaling
 automatically 432
 bulkheads 434
 dynamically 430–432
 improving 432–436
 out 117
 out automatically 117
 out versus up 118
 partitioning 434
 risk 434–435
 shunting 434
 thermostat example 433
 up 118
scheduled transfers 419
ScheduledTransferPeriod property 419
schema 312
 sharding 302
scopes 392, 394
SDK. *See* Windows Azure SDK
secure tokens 390
security 382
 accepting SWT tokens 388–389
 attaching tokens 392
 audience 391
 authorization tokens 383–384
 checking tokens 389
 claims 384
 denial of service attacks 432
 management key 387
 OAuth standard 383
 replay attacks 393
 setting up Access Control Service 385–387
 shared secret 385
SELECT statements 292
self-signed certificates 423
separation of concerns 61
serialized binary data 367
server applications 7
 building 9–13
 building with Visual Studio 10–13
 creating 33–39
 CSS and XHTML 34–36
 deploying with Azure portal 39–47
 fault tolerance 21
 flipping 42
 logging 43
 running across multiple servers 7
 running in development fabric 38
 running locally 37
 setting up storage environment 43–45
 supported types 9
 using local storage 32
server login 309
server name 297
servers
 administration 54
 authorization server 382
 backend 346
 booting 69–71
 CBlox 53
 partitioning 249–252
 scaling 117–118, 346
 scaling out 117
 scaling up 118
 server name 297
 simulating extreme load 115
 trusted authority 382
 under extreme load 116
 under normal load 114
Service Bus 381, 397–399
 connecting to 400
 connecting to services 401
 defined 397–398
 reasons to use 398–399
service bus 351
service configuration file 59, 94–103, 342
 adding settings to 100
 changing 430
 configuring certificates 97–98
 configuring multiple roles 96
 deploying new version 431
 format 95
 runtime settings 98–101
 service model 430
 setting number of instances 96
 standard settings 96
 storing account details 172
 switching to live storage 178
 updating with Azure portal 102
service definition file 59, 62, 81–90, 354
 adding configuration settings to 100
 and certificates 92
 ConfigurationSettings section 100
 defining endpoints 82–85
 editing in Visual Studio 83
 enabling native execution 119
 format 81
 Instances section 86
 Internal Endpoint section 84
 selecting ports 84
 setting trust level 86
 setting up local storage 87
 Startup Action section 86
 storage settings 172
 worker role endpoints 85
service endpoints 339, 343
service health 435
service host 343
Service Level Agreement 301
service management API 78, 421–432
 listing services and containers 424–425
 REST 422
 rolling upgrades 67
 setting up 422–424
 static upgrades 66
 things it can't do 422
service model 59–62
 configuration 59–61
 example 64
 fault domains 63
 roles 62
 service definition file 81–90
 update domains 63
 virtual machines 62
service models 430
service names 41
service oriented architecture 397
service packages 427
service registries 397
ServiceAuthorizationManager class 388
ServiceConfiguration.cscfg file 38, 45

ServiceConfiguration.cscfg file. *See* service configuration file
ServiceConfiguration.csdef file 342
ServiceDefinition.csdef file 38
ServiceHost 342
ServiceHost class 342
ServiceRuntime assembly 34
services
  Access Control Service 381–384
  affinity groups 41
  and Fabric Controller 58
  choosing a name 41
  configuration 59–61
  connecting clients to each other 398
  connecting to 401
  connecting to Service Bus 400
  defining 81–90
  development fabric service 31
  development storage service 31
  enterprise service bus. *See* Service Bus
  exposing 340–344
  exposing to multiple vendors 398
  fault domains 63
  healthy 404
  hosted 426
  listing with service management API 424–425
  managing health 435
  naming 397
  new service wizard 41
  orchestration 397
  Queue 340
  Service Bus 381, 397–399
  service configuration file 59
  service definition file 59
  service model 59–62
  service packages 427
  service registries 397
  service-oriented architecture 397
  setting up 41
  supporting with SQL Server 31
  Table service 240–264
  tracking configuration settings 103
  update domains 63
  upgrading 64–68
  upgrading. *See* upgrading
  version 362
  WaWebHost 109
  WCF 341
  worker role service 336–338
sessions 127–134
  cache-based provider 138
  how they work 127
  killing WaWebHost service 130
  managing in-process 130–132
  memory consumption 131
  SQL session state 135
  sticky 158
  storing session data 128–130
  Table storage state 132–134
  with multiple web role instances 131
Session_Start event 348
SetConfigurationSettingPublisher method 177
SetMetadata method 364–365
setting
  MIME type 213
  permissions 206
sharding 302–303
shared access permissions 206
Shared Access Signatures 198, 207
shared counters 371–372
shared key authentication 273
Shared Key Lite authentication 274
shared secret 385, 400
sharing
  across machines 156
  BLOB data 160–163
  difficulties 156
  shared network drive 157
  storage area networks 159
  with DFS 158
  with SQL Server 157
shopping cart scenario 345
shunting 434
signaling 399
signing key 388
signing requests 187
SignRequest method 186
Silverlight 390
  isolated storage 231
Silverlight applications
  building a media player 224
  chunking 228–232
  communicating with third-party sites 217–223
  hosting 215–223
  restricted headers 230
  Spectrum emulator 215–217
  web search application 219–223
single instruction, multiple data 371
Singleton pattern 348
SizeType entity 318
SLA. *See* Service Level Agreement
sleeping
  infinite loops 374
snapshot isolation 283
snapshotting 206
SOA. *See* service-oriented architecture
Software Development Kit. *See* Windows Azure SDK
source 416
spawn process 149
sp_delete_firewall_rule stored procedure 307
sp_set_firewall_rule storage procedure 307
SQL Azure 4, 23–24
  ALTER DATABASE command 301
  and SQL Server Management Studio 307
  bandwidth 301
  billing 305
  Business Edition 301
  checking SQL version 305
  common scenarios 312–314
  connecting to databases 299–300
  connection routing 305
  cost issues 318
  CREATE DATABASE command 299
  creating databases 298
  creating user accounts 300, 308–309
  database size limits 300–303
  defined 24
  far-data scenarios 312
  firewalls 307
  history 297
  how it works 303–305
  infrastructure layer 305
  limitations 311–312
  logical structure 304
  management account 297
  MAXSIZE parameter 299
  migrating from SQL Server 307
  migrating to 309–311

SQL Azure *(continued)*
Migration Wizard 310–311
near-data scenarios 313
partitioning with 301–302
physical structure 304–305
provisioning 305
querying across partitions 326
referential integrity 326
replicas 304
representing static data 316–318
scalability 24
security 24
setting up 297–300
shopping cart example 323–331
SQL Server Integration Services 307
traditional migration approach 309
Web Edition 300
working with data 306
SQL Azure Business Edition 301
SQL Azure firewall 297
SQL Azure Migration Wizard 310–311
SQL Azure Web Edition 300
SQL Data Services. *See* SQL Azure
SQL Firewall 299
SQL Server
and cloud computing. *See* SQL Azure
backing store 168
checking version 305
in the cloud 23–24
mapping entities to databases 243
migrating to SQL Azure 307, 309–311
sharing data with 157
SQL Azure 23–24
SQL Server Integration Services 307
storing entities 242
supporting services with 31
versus Table service 246
SQL Server Data Services. *See* SQL Azure
SQL Server Integration Services 307
SQL Server Management Studio 307, 309
SQL Serverr 9
SQLCMD tool 299, 310
SSIS. *See* SQL Server Integration Services
SSMS. *See* SQL Server Management Studio
staging environment
differences from production environment 125
inducing failover 126
testing load balancer 124–126
stale messages 361
starting code 428
start-up costs 19
state 350
busy state 433
state machines 351–352
state-directed worker roles 349–353
state machines 351–352
static data 303
asynchronous performance 320
caching 321–322
in Table service 326–329
in-memory joins 328
reference data 316–322
representing 316–318
representing in Table service 318–319
synchronous performance 320
static queues 376
static upgrades 65–66
sticky sessions 158
Stopping event 89
storage
accessing development storage 171–173
Azure Drive 162, 169
BLOB storage. *See* BLOB data
breaking up account 164
creating containers 173
development storage 167–169
direct-attached 159
isolated 231
local 87–90, 353–355
network-attached 159
peer-to-peer 158
setting up account 164
StorageClient library 170
switching from live to development storage 178–179
storage account 174
access key 179
BLOB data 163
breaking up 164
listing tables 266–269
partitioning 249–250
registering domain 164
setting affinity 178
setting up 164
storing details in service configuration file 172
using REST 266–273
storage area networks 159, 305
storage clients
Azure Storage Explorer 253
storage environment 9
setting up 43–45
storage fabric 359
storage procedures
sp_set_firewall_rule 307
storage services 426
StorageClient assembly 34
StorageClient library 148, 170, 185
and queuing 362
as REST API wrapper 182
checking BLOB properties 202
copying with 206
deleting entities 277
downloading BLOBs with 193–195
inserting entities 275, 281
listing tables 267
request signing 268
retries 284
setting custom metadata 204
Unit of Work pattern 276
stored procedures
sp_delete_firewall_rule 307
streaming 223
adaptive 225
with WPF 225–228
strict ordered delivery 359
strings
reversing 337
stubs 36
subscription ID 426
suspend 429
swap-deployment command 429
SWT 384
SWT tokens 388–389
attaching 392
checking 389
validating 388
symmetric key 387
symmetric queues 373
synchronous calls
performance 320

synchronous communication 345
system administrator account 307
System Center Operations Manager 54
system configuration file
  DiagnosticsConfiguration-String 413
system logs 39
System.Data.Services assembly 252
System.Data.Services.Client assembly 252
System.Runtime.InteropServices namespace 150
System.ServiceModel namespace
  ServiceHost class 342
system.webServer 143

T

Table service 240–264
  adding entities 258–259
  and REST 252
  AtomPub standard 267, 270
  authenticating requests 273–275
  batching data 281
  Boolean logic 291
  context class 257
  cost issues 319
  CRUD operations 256–264
  CRUD operations with REST 275–281
  deleting entities 261–262, 277–279
  developing with 252–255
  endpoint URI 266
  equality comparisons 290
  extending entity definitions 246
  inserting entities 275–277
  limitations 240
  LINQ to Objects queries 291
  listing entities 260–261
  merging data 279
  modifying entities for 244–249
  optimizing delete performance 278
  overview 240
  partitioning 249–252, 327
  prefix queries 291
  query expressions 289
  querying data 284–294
  querying with LINQ 288
  range comparisons 290
  representing entities 245–247
  representing static data 318–319
  shared key authentication 273
  Shared Key Lite authentication 274
  shopping cart example 326–329
  SizeType entity 318
  storing dissimilar entities 247–249
  storing table size 319
  updating entities 263–264, 279–281
  versus SQL Server 246
  WCF Data Services 270
Table storage
  cleanup 133
  performance 133
  session state 132–134
  testing 133
tables 18
  creating 253–255
  creating in code 254
  creating using AtomPub standard 271
  creating using REST 271–273
  creating using REST in console applications 272
  deleting with REST 269
  heap tables 312
  listing with REST 266–269
  listing with StorageClient library 267
  partitioning 250–252
  polling 340
  retrieving entities using REST 285
  row keys 251
Tabular Data Stream protocol. *See* TDS protocol
tags
  Last-Modified 201
  LocalStorage 353
  x-ms-request-id 201
TCP 401
TDS protocol 24, 300, 311
templates
  worker role template 339
temporary files 353
testing
  load balancer 122
  under load 123
third-party sites
  communicating with 217–223
threads 343, 347
  background 348
tick count 413
tight integration 357
time filters 420
time synchronizing 184
timeout 369
  setting 370
Timestamp property 244–245, 253
token policies 392–393
tokens 383–384
  attaching 392
  checking 389
  expired 389
  OAuth standard 383
  replay attacks 393
  secure 390
  sending as a client 390–391
  token policies 392–393
  validating 388
tools
  analytical and billing tools 39
  Azure Storage Explorer 253
  BCP 306
  Blob Browser 212, 230
  csmanage 426
  Lucene.NET 331
  makecert 90
  MonAgentHost 406
  SQL Azure Migration Wizard 310–311
  SQLCMD 299, 310
trace listeners 409
  web.config 408
trace logs 408
traceFailedRequests element 408
tracing 418
tracking
  service configuration settings 103
transactional consistency 302
transactions 312
  distributed 312
transferring
  diagnostic data 419–421
  on-demand transfers 420–421
  scheduled transfers 419
troubleshooting 406
  performance 408

truncated exponential backoff 374–375
trust 140–141
  full trust 86, 141
  least priveleged 140
  partial trust 86, 140
  setting trust level 86
trusted authority 382

U

UI. *See* user interfaces
Unit of Work pattern 276
unit tests 344
Unity Application Block framework. *See* inversion of control pattern
unmanaged code 139
update domains 63
  defining number of 63
updating
  update domains 63
upgrading 64–68, 376
  automatic rolling upgrades 66
  manual rolling upgrades 66
  rolling upgrades 66
  single role 67
  static upgrades 65–66
UploadBlobMetadata method 204
UploadFile method 148–149, 192
UploadFromStream method 192
uploading
  BLOB data 191–192
UploadText method 192
upper boundary 434
URIs
  queue names 362
UrlDecode method 392
USE command 311
UseDevelopmentStorage option 409
user accounts
  creating 308–309
  SQL Azure 300
user interfaces
  development fabric 39, 411
  development storage 168
using
  local storage 88

V

validating
  data 350
  tokens 388
verbs
  DELETE 269
  GET 201, 267, 365, 373
  HEAD 201–203
  MERGE 281
  POST 271, 373
  PUT 281
version header 425
VHDs. *See* virtual hard drives
viewing processes 74
VIP swap 42, 428
  See also static upgrades
virtual hard drives 70
virtual machines 7
  anatomy of 8
  Hyper-V 68–72
  images. *See* images
  RDAgent process 76
  running your own 141
  sizes 62, 346
  viewing hosting process 75
  viewing processes 74
  web role details 72
  web roles 72–76
Visual Basic 11
Visual Basic templates
  ASP.NET MVC2 Web Role template 30
  ASP.NET Web Role template 30
  Blank Cloud Service template 30
  CGI Web Role template 30
  WCF Service Web Role template 30
  Worker Role template 30
Visual Studio 144
  and AppFarbic 381
  building applications with 10–13
  Cassini web server 31
  certificates 97
  Cloud Service templates 29
  configuring roles 62
  creating applications 33–39, 267
  creating roles 11
  creating self-signed certificates 423
  creating web pages 12
  CSmonitor tool 29
  CSpack tool 29
  CSrun tool 29
  debugging LINQ queries 290
  DFUI tool 29
  diagnostics 409
  DSinit tool 29
  editing service definition file 83
  makecert tool 90
  reversing strings 337
  running web pages 12
  service configuration file 95
  setting up CGI 143
  setting up local storage 87
  supported versions 10
  Team System Web Load Tester 124
  templates 9
  WCFTestClient.exe 344
  worker role templates 339
VMs. *See* virtual machines
volatile file access 355

W

WAS. *See* Windows Activation Service
WaWebHost process 75
WaWebHost service 109
  caching 136
  killing 126
  killing to kill a session 130
  multiple instances 120
  restarting an instance 123
WCF 343, 388, 400
  addresses 343
  binding 343
  contracts 343
  services 341
WCF Data Services 270
  querying with LINQ 288
  Unit of Work pattern 276
WCF Service Web Role template 30
WCF services 105
WCF. *See* Windows Communication Foundation
WCFTestClient.exe tool 344
web applications
  implementing interfaces 107
web identity 402
web pages
  creating with Visual Studio 12
  running 12

Web Request Authorization Protocol 383
web roles 72–76, 144, 340
  and service configuration file 96
  caching 135
  configuring runtime settings 98–101
  details 72
  enableNativeExecution attribute 86
  hosting locally 84
  implementing interfaces 107
  in staging environment 125
  making available to other roles 84
  Microsoft.WindowsAzure.ServiceRuntime assembly 79
  modifying runtime settings 102
  multiple instances for session state 131
  RDAgent process 76
  receiving messages 82
  recycling 89
  running on IIS 80
  scaling out 117
  scaling out automatically 117
  scaling up 118
  selecting ports 84
  setting number of instances 86, 96
  simulating worker roles 347–349
  versus BLOB data 210
  versus worker roles 8
  viewing hosting process 75
  viewing processes 74
web traffic
  extreme load 116
  normal load 114
  scaling 117–118
  simulating extreme load 115
  testing under load 123
  traffic surges 19
  usage patterns 20
web.config file 60, 99, 143, 409
  appSettings section 99
  handlers section 143
  httpHandlers section 104
  httpModules section 104
  issues with using 99
  moving elements to cloud service definition 105
  system.webServer section 418
  webServer section 143, 145
web.config trace listener 408
web.roleConfig file 142, 145
WebClient class 391
WebHttpBinding class 387
WebOperationContext class 388, 392
WebRequest class 424
WebRole.cs file 409
WebServiceHost class 387
websites
  directory structure 214
  publishing to BLOB services 212–215
  static 209–210
while loops 339
Windows
  supported versions 9
Windows 7
  enabling CGI 142
Windows Activation Service 380
Windows Azure
  abstraction 56
  affinity groups 41
  affordability 19
  and SQL Server. *See* SQL Azure
  and Visual Studio 10
  and Windows XP 10
  AppFabric 4, 25–26
  as a utility service 19
  as cloud operating system 5
  Azure portal 39–47
  building applications 9–13
  capacity 21
  conceptualizing 37
  configuring for FastCGI and PHP 142–143
  creating roles 11
  creating web pages 12
  creating worker roles 14
  determining busy state 433
  development fabric 9, 56
  edge servers 232
  external processes 146–149
  Fabric Controller 14, 57–59
  FastCGI 141–146
  fault tolerance 21
  history 51
  hosting applications 6–8
  how it works 56–57
  improving scaling 432–436
  infrastructure 13
  introduction 4–8
  IP addresses 308
  live IDs 39
  load balancer 13
  logging 43
  managing data centers 21
  message queues 17
  migrating to 80
  multiple instances 20
  naming convention 184
  origin server 233
  packaging and deploying applications 45–47
  PHP 142
  platform 4–8, 23–26
  reacting to varied usage patterns 20
  reduced licensing costs 22
  reducing software maintenance 22
  roles 8
  running applications 6–8
  scalability 19
  scaling automatically 432
  SDK 9, 28
  service management API 78
  service model 37, 59–62
  setting up certificates 90–93
  setting up SQL Azure 297–300
  setting up storage environment 43–45
  shared storage mechanism 16
  signing up for 39
  SQL Azure 4
  start-up costs 19
  storage environment 9
  storing data 15–18
  tables 18
  trust 140–141
  using PHP 143–146
  virtual machines 7
Windows Azure CDN 234–236
  enabling 235
Windows Azure Diagnostics
  analyzing visitors 408
  arbitrary data sources 418
  billing 408
  multitenant systems 408
  other uses 408
  troubleshooting performance 408
Windows Azure platform 23–26
Windows Azure platform AppFabric. *See* AppFabric
Windows Azure SDK 28, 147
  bin folder 29
  Cloud Service templates 29

Windows Azure SDK *(continued)*
  creating applications with 33–39
  CSmonitor 29
  CSpack tool 29, 45
  CSrun tool 29
  DFUI tool 29
  differences from local environment 32
  DSinit tool 29
  folder structure 28
  inc folder 29
  installing 9
Windows Communication Foundation 76
Windows event logs 408, 418
Windows Live Writer 270
Windows PE 70
Windows Server 2008
  enabling CGI 142
Windows Server AppFabric 25, 380
Windows Server AppFabric Caching 137
Windows Vista
  enabling CGI 142
Windows XP
  lack of support for 10
WindowsEventLogsBufferConfiguration class 408
WordPress 146
work
  asynchronous 347
work complete receipt 373
work ticket 351, 360
worker role service 336–338
worker role template 30, 339
worker roles
  and local storage 353–355
  application servers 14
  as Azure services 351
  background threads 348
  common uses 345–353
  communicating with 338–345
  configuring runtime settings 98–101
  creating 14
  drawbacks of large roles 349
  enableNativeCodeExecution attribute 141
  endpoints 85, 341
  infinite loops 374
  logical separation 347
  managing 352
  message queues 17
  offloading work 345–346
  pig in a python 349, 352
  pulling messages 338
  pushing messages 338
  service endpoints 339
  shopping cart scenario 345
  simulating 347–349
  small processes 349
  state-directed 349–353
  threads 347
  versus web roles 8
  worker role service 336–338
WPF 390
  chunking 225
  MediaElement control 225
  streaming 225–228
WRAP 389, 392
wrapper 336
WS-* 384

X

x.509 422
X.509 certificates. *See* certificates
XHTML 33–36
XML Schema Definition language 83
XML standards 267
x-ms-copy-source request header 205
x-ms-date request header 184, 186
x-ms-request-id tag 201
x-ms-version request header 184
XPath 418
XSD. *See* XML Schema Definition language

MICROSOFT .NET/CLOUD

Azure IN ACTION

Chris Hay • Brian H. Prince

Microsoft Azure is a cloud service with good scalability, pay-as-you-go service, and a low start-up cost. Based on Windows, it includes an operating system, developer services, and a familiar data model.

Azure in Action is a fast-paced tutorial that introduces cloud development and the Azure platform. The book starts with the logical and physical architecture of an Azure app, and quickly moves to the core storage services—BLOB storage, tables, and queues. Then, it explores designing and scaling frontend and backend services that run in the cloud. Through clear, crisp examples, you'll discover all facets of Azure, including the development fabric, web roles, worker roles, BLOBs, table storage, queues, and more.

This book requires basic C# skills. No prior exposure to cloud development or Azure is needed.

What's Inside

- Data storage and manipulation
- Using message queues
- Deployment and management
- Azure's data model

A Microsoft MVP specializing in high-transaction databases, **Chris Hay** is a popular speaker and founder of the Cambridge, UK, .NET usergroup. **Brian H. Prince** is a Microsoft Architect Evangelist who helps customers adopt the cloud.

For online access to the authors and a free ebook for owners of this book, go to manning.com/AzureinAction

"Easy to read, easy to recommend."
—Eric Nelson, Microsoft UK

"I doubt even the Azure team knows all of this."
—Mark Monster, Rubicon

"An educational ride at an amusement park—great information and lots of humor."
—Michael Wood
Strategic Data Systems

"Highly recommended, like all Manning books."
—James Hatheway
i365, A Seagate Company

"This book will get you in the cloud... and beyond."
—Christian Siegers, Cap Gemini

ISBN 13: 978-1-935182-48-1
ISBN 10: 1-935182-48-X

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]