Microsoft

# Architecting Mobile Solutions for the Enterprise

Dino Esposito

# Architecting Mobile Solutions for the Enterprise

## Expert guidance for planning and executing a complete mobile web strategy

Rethink your approach to the mobile web and native apps—and build tailor-made solutions to reach customers and clients on a variety of devices. Led by web development luminary Dino Esposito, you'll learn how to create an effective mobile strategy that meets the unique B2C or B2B needs of your enterprise. You'll also gain architectural and implementation guidance for building mobile-specific websites, native and cross-platform applications, and more.

## Discover how to:

- Reach more users with a combination of mobile websites and platform-specific apps
- Architect a mobile-optimized website accessible from many different devices
- Use HTML5 and jQuery Mobile to build sites that look and behave like native apps
- Get started with the basics for building native apps for Windows® Phone, iPhone, and Android
- Implement design patterns specific to mobile application development
- Develop cross-platform app features, such as localization and offline behavior
- Write one application codebase for many platforms using the PhoneGap framework

**Get code samples on the web**
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=247992

*For **system requirements**, see the Introduction.*

### About the Author

**Dino Esposito**, CTO of a company that provides software and mobile services to professional sports, is an expert trainer and web architect. He's the author of several popular books for Microsoft Press® such as *Programming ASP.NET 4* and *Programming ASP.NET MVC 3*.

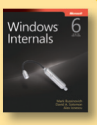
### DEVELOPER ROADMAP

**Start Here!**
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects


**Step by Step**
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook


**Developer Reference**
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples


**Focused Topics**
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples




Microsoft®

ISBN: 978-0-7356-6302-2


**microsoft.com/mspress**

**U.S.A. $39.99**
Canada $41.99
[*Recommended*]

*Programming/Mobile*

Microsoft

# Architecting Mobile Solutions for the Enterprise

Dino Esposito

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472



ISBN: 978-0-7356-6302-2

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.



**Acquisitions and Developmental Editor:** Russell Jones

**Production Editor:** Kristen Borg

**Production Services:** S4Carlisle Publishing Services

**Technical Reviewer:** Marco Bellinaso

**Copyeditor:** Sue McClung

**Indexer:** Margaret Troutman

**Cover Design:** Twist Creative • Seattle

**Cover Composition:** Karen Montgomery

**Illustrator:** S4Carlisle Publishing Services

*To Silvia, because you're stronger than you think.*
*To Michela, because you're just the daughter I always dreamt of.*
*To Francesco, because you're a terrific, quick learner.*

—Dino

# Contents at a Glance

# Contents

# Introduction

As far back as 1999, some smart guys predicted that mobile would become the primary focus of development in only a few years. Although it has taken a bit more time than expected, the era of mobile software has arrived at last. Why did it take so long? The answer is surprisingly simple: mobile software needed a critical mass of users to develop before it could take off. The process of accumulating mobile users probably started with the release of the first iPhone back in 2007, but today, it has reached a large enough mass to trigger all sorts of chain reactions.

Back in 1990 (yes, you read that right), Bill Gates gave a keynote talk at Comdex titled "Information at Your Fingertips." Let's be honest—for 20 years, we pretended we really had information (that we needed) at our fingertips, but at most, we had that information only at hand—which makes a huge difference. Now is the time, though, that we can cover the short distance from hand to fingertips. With mobile devices everywhere, and especially with a revolutionary version of Windows on the horizon, I believe we're truly entering a new era of development—a paradigm shift.

Paradigm shifts just happen—and mobile represents a big one. Mobile enables new business scenarios and new ways of doing the same business. Mobile affects nearly everybody—users, professionals, and clearly developers. Writing mobile applications is a challenge that the vast majority of developers will face in the near future. Overall, mobile applications are simpler than desktop or web applications—but that's true only if you count just the number of functions. The hardest part of mobile development is to identify the right set of use-cases and the right user experience and interaction model. It turns out that the typical mobile application user is much less forgiving than the average user of web or desktop applications. As developers, we forced users to play by the rules of software for decades. In contrast, mobile developers will be forced to play by the rules of user experience and conform to user expectations. This is how software always should have been; but it's definitely not how software has been built for at least the past 20 years. Moreover, before too many more years pass, mobile may well be the only software that we will be called upon to write.

The term mobile refers to a variety of platforms, each with its own set of capabilities and features, and each of which requires significantly different skills: different operating systems, different programming languages, different application programming interfaces (APIs), and even different computers. A mobile application is more sophisticated and more complex than web applications with regard to resource management, data entry, sensors, data storage, and life cycle. Furthermore, each operating system has its own set of development guidelines and a proprietary deployment model.

This book is intended as a quick-but-juicy guide to issues that you may face while developing a mobile project for one or multiple platforms. The book starts by analyzing the various types of mobile solutions, which include websites, websites optimized for mobile devices, and native mobile applications, and then identifies a few design patterns common to all mobile applications and technologies available on the various platforms. Predictive fetch, back-and-save, and guess-don't-ask are just a few of the patterns being discussed and implemented. The book puts considerable emphasis on mobile sites and frameworks, and on techniques to detect browser capabilities accurately. For example, the book offers a chapter on Wireless Universal Resource FiLe (WURFL)—the framework being used by Facebook for mobile device detection—and compares that to the detection capabilities in plain ASP.NET.

Furthermore, the book offers an overview of mobile development for the three major platforms—iOS, Android, and Windows Phone. In particular, this book builds the same application for all three platforms, discussing tools, frameworks, practices, and illustrating architectural and structural differences along the way. Finally, the book covers PhoneGap and HTML5-based development for mobile devices.

After reading this book, you probably won't be a super-expert in any of those platforms, but you'll know enough to start producing code on any of the most popular devices. You'll also know enough to advise your customers and help them define effective mobile strategies for their business.

## Who Should Read This Book

As companies start going mobile, they need a strategy long before they need a mobile site or an iPhone app. But when companies have developed the strategy and start looking into implementing it, they face the rough issue of not having or finding architects and developers that know the mobile world from a variety of angles. Today, they can easily find great iPhone or Android developers, but they can hardly find a consultant that can suggest, based on strong evidence, whether a mobile site is preferable for them.

This book is aimed at providing an architect summary of what you need to know to design and implement mobile solutions. Today, a mobile solution often means arranging the same application for several different platforms (iPhone, Android, Blackberry, and Windows Phone), and doing that using a very specific set of design patterns with little in common with desktop or web apps. Last but not least, the effort must be done in the context of the customer's needs, expectations, and existing business.

## Not a Mobile Developer? Not a Developer!

For a company with a consolidated business, mobile is a way to expand its horizon. The new expansion stage of mobile is reaching out to companies and enterprises and prospecting new ways of doing business. This is a paradigm shift with a deep impact that will give rise to new professional jobs, much as the web itself did more than a decade ago.

That's why I maintain that in only a couple of years, every developer will be either a mobile developer or no developer at all. Being a mobile developer surely includes knowing iOS, Windows Phone, HTML5, and Android, and perhaps BlackBerry, possibly Bada, and even developing for smart TVs—and, of course, for the mobile web. More than anything else, though, developers must acquire a "mobile mindset." You can always figure out fairly easily how to play a video on iOS, or how to make an Android device vibrate. But what isn't as easy to acquire is the intrinsic nature of mobile applications and the patterns behind them, and which aspects to focus on for optimization.

Mobile is different. Overall, it's simpler, but it's also much less forgiving than other types of applications.

Therefore, this book is for everybody who needs to acquire some mobile development insight. The book's contents won't become obsolete in just a few months because I made a serious attempt to reach and report from the heart of the mobile experience. This book discusses technology, but it is not based on any particular technology; therefore, it's an introductory text for any form of mobile development.

# Who Should Not Read This Book

This book won't make you a top-notch iPhone or Android developer; it's intended to help everybody (including those of you who are already top-notch iPhone or Android developers) understand the entire mobile world. The goal is to get readers prepared for architecting effective mobile solutions after a mobile plan has been finalized and accepted. If you're looking for detailed, step-by-step examples of how to play an animation, make the phone vibrate, or making an Internet call on all possible platforms, you won't usually find them here. But I hope that you will find enough to help you get started with every aspect of mobile development.

## Organization of This Book

This book is divided into three sections. Part I, "Going Mobile," is about the possible strategies to approach the mobile world. Part II, "Mobile Sites," covers the architecture and implementation of mobile sites and also touches on HTML5 and jQuery Mobile. Part III, "Mobile Applications," is about the three major mobile platforms of today—iOS, Android, and Windows Phone—and also covers PhoneGap as a way of unifying development in a single codebase.

### Finding Your Best Starting Point in This Book

The different sections of Architecting Mobile Solutions for the Enterprise cover a wide range of technologies associated with mobile development. Depending on your needs and your existing understanding of mobile, you may wish to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

| If you are | Follow these steps |
|---|---|
| New to mobile and spent your entire career doing other software-related work | Read the chapters as they are laid out in the book. |
| A web developer looking into how to build mobile sites | Focus primarily on Part II. |
| A chief technology officer (CTO) or chief architect | Focus on Part I first, and then move to Part II and/or Part III, depending on whether mobile sites or mobile apps are more likely to be relevant in your context. But read the entire book anyway. |
| Familiar with mobile app development in one (or more) platforms | You might want to start with the chapters that cover topics that you are familiar with. These chapters are essential guides, so it is likely that you won't learn anything new there. But if you find that you miss some of the points discussed, then you've got something already from the book. Next, I suggest you focus on Chapter 7, "Patterns of Mobile Application Development," and Chapter 11, "Developing with PhoneGap." |

**Note** This table simply attempts to provide some guidance on how to learn best from this book. In any case, I heartily recommend that you read all the chapters thoroughly.

## Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

## System Requirements

You will need the following hardware and software to set yourself up for development on the various mobile platforms and compile the sample code that accompanies this book:

- For iOS, you need a Mac computer with Xcode and the latest iOS software development kit (SDK). If you plan to use MonoTouch, then you also need to get at least a trial version of the product from *http://www.xamarin.com*. Note that to deploy applications on a iOS device, you also need to be a registered Apple developer enrolled in one of the Apple pay programs.
- For Android, you can use a Windows PC, preferably equipped with Windows 7. Note, however, that you can do Android development from a Mac or Linux PC as well. You can use Eclipse or the IntelliJ IDEA as your integrated development environment (IDE). You will need the Java SDK and the Android SDK installed. You don't need to be a registered developer to compile and deploy Android applications on a device.
- For Windows Phone, you need Microsoft Visual Studio Express for Windows Phone, as well as a Windows PC.

## Code Samples

This book comes with a few examples organized as follows:

- Two ASP.NET websites configured to use WURFL

- The Guess application for iOS
- The Guess application for Android
- The Guess application for Windows Phone
- The HTML5 Guess application for PhoneGap

The sample code contains files that you can incorporate in your own projects using the tools that you prefer.

Many of the chapters in this book include examples that let you try out new material discussed in the main text. You can download all the sample projects from the following page:

*http://go.microsoft.com/FWLink/?Linkid=247992*

Follow the instructions to download the Amse.zip file.

**Note** In addition to the code samples, your system should have Visual Studio 2010 and Microsoft SQL Server 2008 installed. The instructions that follow use SQL Server Management Studio 2008 to set up the sample database used with the practice examples. If available, install the latest service packs for each product.

## Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the Amse.zip file that you downloaded from the book's website (name a specific directory, along with directions to create it, if necessary).
2. If prompted, review the displayed End User License Agreement (EULA). If you accept the terms, select the Accept option, and then click Next.

**Note** If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Amse.zip file.

# Acknowledgments

It took me several months of deep dive to make sense of the many facets of mobile: the customer's angle, the developer's perspective, the architect's vision, and the myriads of devices, operating systems, SDKs, and products. Many friends helped me out along the way.

First and foremost, I want to thank Marco Bellinaso of Mopapp, who first introduced me to the world of mobile apps and then served as an invaluable technical editor for this book. Marco also tried to make me a fan of Objective-C, but I'm afraid his efforts failed in that regard.

Devon Musgrave of Microsoft Press and Russell Jones of O'Reilly believed in this book and made it happen, along with Kristen Borg and the other members of the editing team.

I was surprised to see how many friends asked to review chapters and enthusiastically shared their feedback. I could see an underlying passion and pleasure in their work and I'm not sure my monumental THANK YOU here is enough. In particular, I wish to thank Luca Passani of ScientiaMobile. I met Luca at a web conference in London in 1999, where he tried to sell me mobile as a hot business even back then. It took a bit more time, but his vision was definitely right. I really enjoyed the feedback about mobile site development and HTML5 that I got from Jon Arne Saeteras of MobileTech and Daniele Bochicchio of 5DLabs. IT and Microsoft Regional Director for Italy. The chapters on mobile apps and PhoneGap benefited from the feedback of many people, including Davide Zordan, Ugo Lattanzi, Leon Zandman, Catalin Georghiu, and Davide Senatore. All these people shared their real-world experience with me concerning Windows Phone and PhoneGap.

Near-final thanks go to my team at Crionet and E-tennis.net. As I write these notes, we are finalizing the mobile apps for the worldwide audience of tennis fans following the Rome ATP Masters 1000 tournament. It's the first tournament to offer a comprehensive mobile, web, and social experience and the first one to offer mobile apps on a full range of platforms, including not just iOS and Android, but also Windows Phone and BlackBerry. Working with you guys is a privilege.

What else? Well, just a final note. Take note of this name: Francesco Esposito. I'm sure you'll hear this name in the future. He's 14 and he's already an all-round mobile developer. My use of the word developer is no accident, because that's what he is, irrespective of schooling and age. In his way of coding, learning, thinking, and speaking, I see crystal-clear talent. Being his dad, well, I feel proud.

## Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

*http://go.microsoft.com/FWLink/?Linkid=247993*

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

## We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

*http://www.microsoft.com/learning/booksurvey*

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

## Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

PART I

# Going Mobile

CHAPTER 1

# Pillars of a Mobile Strategy

*In preparing for battle, I have always found that plans are useless, but planning is indispensable.*

*—Dwight D. Eisenhower*

**In this chapter:**

- What Does "Going Mobile" Mean?
- Outlining a B2C Strategy
- Outlining a B2B Strategy
- Summary

The modern era of mobile technology began with the release of the first Apple iPhone in the summer of 2007.

The mobile conquest of the world has been a "soon-to-be" matter for quite some time in the past decade. I still remember the first-ever mobile-related conference being held in Amsterdam in the summer of 2000—the Wrox Wireless Developer Conference. I was a speaker there, and the implicit message for attendees was "Mobile development is here—hurry up."

There was no hurry, actually.

Only a couple of years later, Microsoft released ASP.NET with its own set of mobile controls for optimized mobile websites. Later, mobile frameworks such as Microsoft .NET Compact Framework and Java Micro Edition (J2ME) appeared; meanwhile, richer native operating systems such as Symbian also appeared. However, the mobile conquest of the world never happened—and perhaps hadn't even begun—which begs the question: Why not?

The main reason is that the technology never reached a critical mass of users, and without that, developers and software houses had no good reason to address the mobile space. But when the Apple iPhone appeared, everything changed. Although the iPhone was not an entirely new idea, it was an extremely well-done implementation. And, more importantly, a lot of people (on the order of millions) liked it. That immediately created a breeding ground for new applications and gave mobile technology a new form and immediacy.

The lesson to learn from this is that software is the *effect* (not the cause) of the mobile phenomenon. People buy devices long before they have much compatible software to run on them. Therefore, a compelling device, bought by a critical mass of users, creates a compelling market for specific software over time.

Today, there are a few popular mobile operating systems and a growing number of users willing to pay to get nice applications to run on them. The popularity and convenience of mobile devices drives companies to create their own mobile applications that can reach their customers while they're traveling. Mobile sites are still an excellent way to do that, but whether companies build mobile sites or mobile applications targeted to a particular platform (today, that would include iPhone, Android, BlackBerry, and Windows Phone), companies need to be part of the mobile revolution in much the same way they became part of the web revolution a decade ago.

## What Does "Going Mobile" Mean?

This book is aimed at architects and developers who are willing (or need) to implement mobile solutions for customers. A solution, however, is not necessarily and not simply a mobile application. Today, and even more in the near future, a *mobile solution* will be created as a combination of a classic website for desktop browsers, a website specifically designed for classes of mobile devices (known as an "m-site") and one or more applications for specific mobile operating systems.

The definition of a mobile solution is not carved in stone, for two excellent reasons. First, the mobile industry never sleeps; it churns out requirements and opportunities at an impressive pace, so any current definition of a mobile solution may change to incorporate new aspects in a matter of just one or two years. Second, a mobile solution applies to a particular business scenario. The business scenario ultimately determines the details of the solution and technologies, patterns, and platforms that architects and developers will deal with. As an example, you may need to add some Facebook applets or multiplatform desktop applications if the business has social networking implications. Similarly, you might restrict the range of mobile platforms to just one if you're building a vertical enterprise-class solution for a single customer.

As I see things, going mobile is a far more serious task than simply writing an iPhone application. Companies investing in mobile need a strategy long before they need a mobile site or a set of mobile applications. This means companies must establish goals as well as review processes for achieving those goals—simply put, they must have a strategy.

To paraphrase the quote from Dwight Eisenhower at the beginning of this chapter, in mobile development, plans are useless, but planning is indispensable.

### Toward a Mobile Strategy

So the first step for a company "going mobile" is to define a strategic plan. The strategic plan is more conceptual than it is an operational plan with comprehensive implementation details. The strategic plan is visionary; it identifies the future direction of the business. Outlining a mobile strategy essentially consists of reviewing the current business processes with regard to a few mobile axioms.

## Three Mobile Axioms

Gone are the days in which a website optimized for a bunch of desktop browsers was the only way for a company to deliver an application. Today, there's a growing demand for applications that users can reach from a variety of platforms and browsers. In the past, software architects once reached for the Holy Grail of multiplatform development—and we failed to grasp it. Now, as users increasingly demand multiplatform applications, failure is simply not an option.

Mobile axioms are statements about mobile applications that are self-evident and assumed to be true. You should have these concepts clear in your mind before you start planning your strategy:

- Provide your services through multiple channels.
- Look for new opportunities and new ways to provide your services.
- Aim at making your customers' lives easier.

Like the web a decade ago, mobile is about new ways of doing both a selection of old tasks and entirely new actions. Mobile is highly attractive to users because they can get the services they need in a variety of ways and using a variety of devices. As a company, "going mobile" means being committed to making your customers' lives easier through ad hoc and personal services.

The fundamental point, however, is that this challenge is not limited to just a few segments of the industry; it's a global challenge.

## Multiple Channels

As you can guess, going mobile likely involves significant investments on your side to restructure existing processes, implement new ones, and fix—or at least extend—a portion of your back end software.

Delivering services to a variety of channels is challenging. Mobile channels (tablets, devices, or mobile sites) are more personal and typically involve smaller amounts of information. Your existing back end must be able to serve these new requests effectively while preserving both scalability and performance, and while still ensuring at least the same level of security.

A good example of an application delivered through multiple channels is Facebook; other examples are airline booking and home banking services.

## New Ways to Provide Services

Mobile is both about bringing existing services to people's fingertips and about creating brand-new services. A mobile device is a personal device, so everything that shows up there is potentially "at your fingertips." The real estate of a mobile device is considerably smaller than a laptop, but most applications and websites are padded with extra information (including menus and layout) that is not necessarily required. The advantage of a mobile solution in this context is that it can provide exactly what's needed whenever the user needs it—instantly.

A mobile user is typically traveling around. Your application may query the user's current location and use that information to offer new, unique, and tailor-made services. Location-aware services are really at the heart of the extra power of mobile applications. This is not so much because a desktop site is unable to detect the user's location, but because a site can use the location details in much more compelling and useful ways when the user is out of the office. This is definitely an area to explore if your business is in any way related to location.

As an example, an application that provides information about transportation can use your location data to restrict search or sort results automatically, winnowing out nonessential data for other locations. The same concept applies to mass retail applications, which might notify users of special offers when they are close to a shop, or provide them with free coupons in a nearby shop that they can reach within a few minutes.

## Simplify Customers' Lives

I see more and more companies from a variety of industry segments strongly committed to making their customers' lives easier and better. I believe that is a key challenge for attracting new customers and keeping existing ones. On the other hand, by not going mobile, you risk alienating customers from your brand.

As mentioned earlier, mobile applications are more personal than desktop applications. They're often relatively simpler in terms of logic and complexity, and they often consume smaller amounts of information. That's precisely what makes a customer's life simpler—the application is more focused; ideally, it can handle more related information aggregated from multiple remote sources. Basically, an effective mobile application should be able to give users what they need at any particular moment.

Architecting the system around these new needs is the effort that companies should invest in. It's not simply a matter of software architecture, though. Architects may be able to tell the best way of realizing an idea, but they can hardly identify what makes your users happier. In general, an appropriate analysis and prioritization of use-cases selects the range of features that—once implemented—put more information at the user's fingertips and make life easier.

## Mobility and the Industry

According to a Gartner report presented in the spring of 2010, mobility occupies a relevant position in the list of top priorities for chief information officers (CIOs) of various industry sectors through 2013. According to this report, transportation and retail are the industry sectors that are paying the most attention to mobility.

In these sectors, there's a strong sentiment that it is an "either now or never" matter; there's less and less space left for companies that hesitate or just skip going mobile. The mobile space is open for business (for now) and companies need to establish their presence as soon as possible. If they don't, others will fill the gap and become your toughest competitors.

Also, according to Gartner, beyond transportation and retail, other sectors interested in mobility are healthcare, utilities, education, and—guess what—software publishers. Media and financial services are also there, lower on the list.

The trend that Gartner excerpts from CIOs' priorities may be different from country to country; however, past history shows that a general trend is always a trend that applies worldwide (though at a different pace in various locations).

I can contribute my direct experience in Italy, where most leading mass retail companies are only now experiencing what many experts call the first stage of mobility—merely establishing a static presence. Typically, this process is initiated via nearly functionless mobile sites that go hand in hand with existing primary desktop sites. The next step usually involves adding a bit of context through proactive alerts, and advertising based on location, identity, or perhaps barcode recognition. Finally, the third level of mobility awareness concentrates on providing all-round services at users' fingertips.

## Defining a Mobile Strategy

Each business has its own mission, expressed as purposes and activities. A mobile strategy revisits and extends these purposes and activities in light of new devices and a new lifestyle. The mobile axioms should just inspire a realistic vision for the mobile business.

If this scares you, don't worry: it's nothing new—in fact, you've been there already, a decade ago. Although different in features and results, the mobile revolution follows the same pattern that the web revolution did. Early adopters content themselves with just being there and show customers they're online. Then, executives start developing a new vision of the business and architects actually build it. It's not a waterfall-like process; actually, it has a lot of inherent agility and looks like an intertwined process. In the end, every company ends up with what the management envisioned in their future—good and bad.

### What Do You Want to Achieve?

Personally, I think that for most companies, embarking on mobile projects is not a choice related to gaining an immediate profit. Of course, that mostly depends on the type and size of company. If your business is selling ringtones, then naturally you expect profits from your mobile software right away. However, if your business is selling news, you might want to use mobile channels to make your readers' lives easier, so long as you can add such services at a reasonable cost to you. With the all-free model becoming less affordable every day, going mobile and attracting readers with mobile device capabilities is an immediate expense that hopefully will help achieve better results in the medium term or in the long run.

With a strategy defined in terms of expectations and requirements (covering growth, profitability, and markets), you can look at your overall mobile technology strategy. All in all, there are two (not mutually exclusive) possible expectations: reaching the largest possible audience and improving the experiences of existing customers by building a rich, jaw-dropping application. Implementing each scenario may require a different set of concrete technologies, languages, and platforms. And each scenario may have different costs.

## Reach Out to Users

You reach mobile users by making your application available on the devices they use. This apparently obvious, no-brainer statement hides all the complexity (and costs) of mobile development. Take a closer look at this statement, though, and you'll find two huge questions whose actual implementation determines the actual level of complexity (and costs) of reaching out to users:

- Which devices are your customers using?
- How do you make your application available on all of them?

Before you can answer those questions, you need to think about this: What's a mobile device, anyway?

According to one widely accepted definition, a mobile device is one that you might have with you at any time, can be used more or less instantly, is a personal item, and can be used to connect to a network. A laptop, for example, seems to match most of these requirements—except that you are hardly likely to take it with you when you go out for a walk or buy groceries—and laptops don't usually start instantly. Cell phones mostly fall into the category of mobile devices (many cell phones have at least some browsing capabilities). Finally, smartphones and tablets match all the definitional requirements.

**Note** Recently, I used the preceding words, *more or less,* to introduce mobile development challenges to a developer audience. One of the attendees winked and playfully replied: "So, you mean that my Windows Mobile phone is not a mobile device? It takes ages to boot up."

A mobile strategy also depends on the level of control you can exercise over the devices your users have. For example, if in your context, *user* means "employee," then the company can decide to support just one mobile platform and focus development on that. If you think that *user* means "consumer," however, then reaching out to a large audience usually means developing multiple similar applications for various devices. The same applies to scenarios where *user* means "employee," but the company is giving its employees the option to use the device of their choice.

Deciding how to approach the technology is a delicate and critical point of a mobile strategy that I'll address in more detail in the section "Outlining a B2C Strategy," later in this chapter.

## Offer Rich Applications

If you know that a significant share of your users connect to your site using a particular mobile device, or if the content you're offering can best be consumed on specific popular devices, then your mobile strategy should include the development of an ad hoc application optimized for that device. You don't have to target each possible family of devices; instead, you can establish priorities and add new applications progressively.

Suppose that you own a radio station. You want to increase your audience so you can sell more ad slots. Most radio listeners are faithful, so despite the switch to mobile, they may well still be listening to their favorite radio station while out and about. They might be listening via radio-equipped MP3

players, original equipment manufacturer (OEM)–applications using the embedded radio system of a mobile phone, or Internet-based free radio programs. In all cases, users can listen, but they can't interact and increase your site traffic. But if you can develop a specific mobile application and let listeners interact with your back-end systems via the web, consume streamed live music, access podcasts, traffic reports, news, submit feedback, blog, and more, you can gain interactivity and increase user participation.

Should you address all the major mobile platforms at the same time? That mostly depends on both your budget and management's expectations. One common pattern is to build an iPhone application first, and then follow that up with an Android or iPad application. At a radio station, to continue with the example, a tablet device such as the iPad may add little extra value compared to an iPhone. So the second step in your strategy probably would be to develop an Android application, letting iPhone and iPad users share the same application.

I'll return to this point in a moment and address it more specifically in the next chapter, but keep in mind that mobile applications don't necessarily mean iPhone or Android applications. A mobile site can be as functionally rich, and it is usually more cost-effective.

## B2C and B2B

The full spectrum of mobile applications falls into one of these two categories:

- Business-to-Consumer (B2C)
- Business-to-Business (B2B)

A third label is worth mentioning, though: Consumer-to-Consumer (C2C). Although not terribly relevant at the current stage of the industry, C2C provided the spark for the whole mobile revolution. The mobile revolution we're experiencing these days would probably have remained on hold for another 10 years without a lot of (initially) independent developers who enthusiastically embraced iPhone and Android programming and built clever applications (regardless of their usefulness). Some of these developers capitalized on the success and exposure of a single application to build a business and help the mobile revolution thrive.

Going B2C or B2B poses different challenges and drives different implementation choices. For example, in a B2C scenario, a key decision is about how to make the application available and get consumers to notice it—whether it's a free or paid application. In some cases, the question is a no-brainer (the app pretty much has to be free). In other cases, a more sophisticated model that offers a free (but perhaps feature or time-limited) version of the application is offered to entice users to purchase the full-feature paid version. In still others, consumers can select either an ad-supported version or an ad-free paid version.

In contrast, in a B2B scenario, you have a fixed number of users to reach. Here, your focus is on enabling users to return what you expect quickly, effectively, and securely. Security and middleware, in fact, are usually far more important in B2B scenarios.

## Development and Costs

Developing mobile applications is neither cheap nor quick. Many companies find this surprising when they approach mobile projects. But mobile development is only *apparently* similar to web or Windows development; the two have different programming frameworks and often different (and uncommon) programming languages. Furthermore, mobile suffers from the lack of a consolidated set of patterns. Another reason that raises costs for mobile is the need to produce different user interfaces (often both layout and images) for different devices. This has never been a requirement for web or desktop applications. All these factors currently make mobile development significantly *more* expensive and time-consuming than web development, although time will help alleviate some of these issues.

It is commonly believed that outsourcing development is preferable to having in-house development, largely because in-house development means that you first need to invest in training. It's one thing to train a team of developers on ASP.NET and then have them build three sites in a row. But it's quite another to train a team on three different mobile platforms and then have them build the same application three times from scratch—once for each relevant platform you plan to address.

Outsourcing allows you to eliminate in-house training costs and speed up development. In return for this, however, you must pay more for the outside expertise. It's worth exploring some of the reasons that make mobile development more expensive than many executives think at first.

### Targeting Multiple Platforms

The mobile ecosystem is populated by several different platforms, each of which has its own more-or-less unique set of features and capabilities. The most popular platforms today are iPhone, iPad, Android, Windows Phone, and BlackBerry. The list of platforms, however, doesn't end here. Other platforms that you are likely to encounter or need to consider are Symbian, Windows Mobile, Meego, Bada, QT, and webOS. And when you begin to look at using tablet devices, the range of platforms that you may need to take into account grows even more, because there are tablet-specific variations of the aforementioned platforms, including Android Honeycomb, BlackBerry PlayBook, and the upcoming Windows 8.

Each platform has its own operating system, its own programming application programming interface (API), and its own set of programming guidelines. Often, each mobile platform requires applications be written in a specific programming language, such as Java, Objective-C, C#, or C++.

So does this mean that you must port or develop your application from scratch for each of these platforms?

Frankly, very few applications (e.g., content providers) need to address all these platforms. More typically, applications target a subset of no more than three or four of them. If it is crucial for your business to reach the largest possible audience, even those running on low-end devices, then you might want to look at HTML—specifically HTML5—to build a website optimized for mobile devices (i.e., an *m-site*). As you'll see in more detail in the next chapter, m-sites are often the first option that you should consider when targeting multiple platforms is a true business necessity. M-sites, however, are not free of device issues either. In the end, building a mobile site can be considerably more complicated than building a website.

## Addressing the Device Fragmentation Issue

If you felt frustrated by desktop browser fragmentation—too many different browsers to optimize webpages for—you have never explored the mobile jungle. Each device—and by device I don't simply mean smartphones—has its own browser, and each browser has its own user agent string, which changes for each version and operating system update. And, of course, the actual set of capabilities can change for each device as well. The screen size is probably the most important capability to take into account because of real estate and pixel density.

The dimension of the device fragmentation problem is far larger with mobile browsers than with desktop browsers. When it comes to mobile site development, you have thousands of different device models to take into account, not just a few dozen smartphones, often with a pre-fixed set of capabilities. How can you approach such a task?

Writing a set of pages (if not the entire site) on a per-device basis is simply not feasible. The one-size-fits-all approach is viable, but it comes at the cost of leaving a lot of older devices behind and giving up on advanced features that smartphones have. This is typically not good enough for companies whose success depends on online content, such as social networks, or media and news companies. The alternative is *multiserving,* which basically consists of three points:

- Group devices in classes based on their capabilities
- Build a version of the site for each class of devices that you intend to support
- Define a strategy to serve the right site for each connecting device

That's easy to say, but how can you determine the capabilities of a given device? How can you know the size of the screen, the operating system, the quality of video codecs, whether the device supports graphic processors or certain HTML features (e.g., file upload and CSS gradients), the availability and accuracy of location services, and even much more specific capabilities, such as image inlining (the ability to display images from page-embedded Base64-encoded strings)?

For some of these capabilities, such as screen size, you can ask the browser itself. In fact, forums are full of questions about how to determine effectively the "real" size of a screen on a particular device and model. For other capabilities, such as image inlining, there's just no way to make such a query. You just must *know* it.

About 10 years ago, Luca Passani had the vision of starting a community-driven project aimed at collecting reliable information about the effective behavior of mobile devices. He created the WURFL project, short for "Wireless Universal Resource File." Today, WURFL is a centralized database that stores detailed information (more than 500 different capabilities) about more than 15,000 mobile devices and mobile browsers. Today, WURFL is managed by ScientiaMobile (*http://www.scientiamobile.com*) and made available through both commercial and open-source licenses.

Multiserving takes mobile development to a new level of complexity, but this is where WURFL shows its value: WURFL makes multiserving manageable. Multiserving is inherently expensive, but using WURFL can make it considerably less expensive.

I'll return to the topic of mobile site development in Chapter 4, "Building Mobile Websites," and cover WURFL features in detail in Chapter 6, "Developing Responsive Mobile Sites."

**Note** WURFL is the device detection engine that powers a number of very large and popular mobile sites: Facebook, Google, AdMob, and a long list of mobile network operators and virtual network operators.

## Looking for Best Practices

If you are building a desktop website, you can rely on a number of tutorials, widgets, articles, books, and posts that give you guidance. The same isn't true for mobile software.

The importance and complexity of mobile site development is not yet perceived in its entirety. Too many developers (and, worse, architects) succumb to the siren call that m-sites are simply standard websites with different Cascading Style Sheets (CSS) and layout.

Turning to native mobile applications, all you can find are official API references, long and staid official guidelines in the form of white papers, and a ton of useful tips and tricks scattered in a variety of question/answer sites (such as StackOverflow). This is largely because mobile applications are relatively new and the entire space is fragmented; very few developers who program for iPhones know (or are interested in) Android or Windows Phone development. Furthermore, the stereotypical iPhone/Android developer considers mobile sites old-fashioned.

The bottom line is that when you are facing mobile development for business (for example, say your boss told you that you have to build an application in just a few weeks), you have no good place to look for common practices. Even when you can figure out most common practices, it's tough to know whether those common practices are also best practices.

## The Marketplace Tax

Finally, development of mobile applications is subject to appstores. Apple made this model popular with i-tools (such as the iPhone, iPod Touch, and iPad); Microsoft took the same route with Windows Phone (and seems to be inclined to forge ahead with it in Windows 8); Google (for Android) and RIM for BlackBerry left their appstores optional for developers.

The role of appstores is crystal clear: they are there to protect users who buy or download applications from an appstore to their devices. The appstore owner guarantees the quality of published applications. For developers, getting approval from the appstore owner requires more effort to ensure the quality of the final product—which is not a bad thing for consumers. For companies, the appstore model means that there's an extra distribution cost, which I like to call the "marketplace tax." Companies have to pay to gain the right to distribute even free applications, and for paid applications, they typically have to provide about 30 percent of the app's revenue to the appstores.

## Outlining a B2C Strategy

A B2C strategy is built around two pillars: reaching out to users and making them happy. Both pillars are quite generic and can be implemented in various scenarios with slight variations.

You may need to reach the largest audience possible, including holders of low-end devices devoid of flat connectivity rates. Likewise, you may need to focus on holders (and potential holders) of smartphones. You may need to push a mobile application with certain characteristics to keep existing users and make them glad that they chose your brand. Alternatively, you may need a mobile application to attract and engage new users by offering new services or new ways of consuming existing services.

Needless to say, a B2C approach is particularly suitable for companies that already operate their core business in B2C mode. It comes as no surprise that, according to the Gartner report mentioned earlier, the industry sectors most interested in mobile are transportation, retail, healthcare, software publishers, financial services, media, and in general, content providers.

### Focus on Your Audience

Any business that aims at being successful should focus on its potential audience and make projections about the composition of this audience in terms of age and other social and personal aspects. With this consolidated information in hand, you can make better plans. In this regard, a mobile strategy is merely a specific form of business strategy.

A mobile audience is made up of people who own a mobile device and are (or may be) interested in the services you provide. Figure 1-1 depicts these two sets of users and shows how mobile applications fit in with your existing customer base.

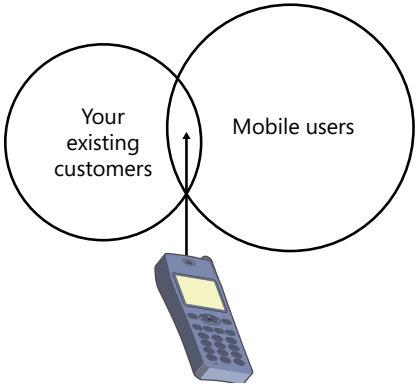



**FIGURE 1-1** Mobile applications as the point of contact between existing customers and mobile users.

**Note** With regard to Figure 1-1, it should be noted that the overlap between "mobile users" and "existing customers" is moving and may change from month to month. When looking at the figure, don't take the size of overlap as truly representative of all businesses. The fact that the overlap is not null is perhaps the really important thing to remember.

Not all of your existing customers will become users of your new mobile infrastructure, but some generic mobile users will join the universe of your customers because of the mobile framework. This also should be read the other way around: If you don't go mobile, you may lose a share of your existing customers who are also mobile users.

## A Quick Look at Global Numbers

It may sound obvious, but I'm going to say this anyway: the world is full of mobile devices. For the most part, these are low-end devices with a basic HTML browser, a quarter VGA (QVGA) screen (240 x 320 pixels), perhaps a camera, an MP3 player, and a few games and utilities.

According to the 2010 statistics of the International Telecommunication Union (ITU)—the agency of the United Nations (UN) responsible for information and communication technologies—there are 78 mobile devices per 100 inhabitants distributed all over the world, and a peak of 114 per 100 inhabitants in developed countries (see *http://www.itu.int/ITU-D/ict/statistics*).

Whichever way you look at it, the data shows that there are a few billion mobile devices of any type out there. How many of these are devices (and users) that you want to reach with your application? Probably as many as possible if you're Facebook or Google; a small fraction is enough otherwise.

The same ITU source reveals that there are about 30 Internet connections per 100 inhabitants all over the world, and 70 per 100 in developed countries. Although the two numbers are not directly related, this statistic gives a better approximation of the size of a potential mobile audience. However, according to eMarketer (*http://www.emarketer.com*), in 2011 the smartphone penetration in the world expressed as a percentage of all mobile devices is around 11 percent. That figure is expected to grow to about 50 percent over the next three years.

The data is more interesting when you look at these numbers for selected areas and countries. For example, the smartphone share grows to 37 percent in North America and 32 percent in Western Europe. It's around 10 percent in Asia and stays below 5 percent in Africa and Latin America. Amazingly, the country with the highest penetration is Italy, with 47 percent currently (expected to grow to 67 percent by 2014). And this in a country—my country—that still has wide areas of digital divide, and where one family out of three doesn't even have a home broadband connection.

The next section presents a few more numbers to help you understand the big picture of mobile connectivity.

## A Deeper Look at Numbers

If you take global numbers literally, then by focusing on an iPhone application and disregarding mobile sites entirely, you cut off 90 percent of the potential worldwide audience—and even more than that if you consider that not all iPhone devices may be capable of running your application because of versioning issues. From this perspective, a mobile site seems to be a very reasonable choice.

**Note** That iPhone users are approximately 10 percent of the total smartphone-using population is an estimate that seems to find many direct and indirect confirmations from a variety of sources. Considering only the U.S. market, iPhone users represent about one-third of the smartphone segment, which is reported to range around 30 percent of the total audience for mobile devices. Statistics, however, depend on a number of factors and often represent little more than an opinion!

Regardless of your final choice, blindly looking at global numbers is not necessarily the correct approach.

Suppose that after running a few customer surveys and having analyzed your website logs, you know that 50 percent of your real customer base use iPhones and connect from Italy. Given those figures, should you really focus your effort only on a mobile site? Probably not. A desktop site that looks decent on most devices, that looks good on iPhones, and features a native iPhone application is the best combination. Note that the costs of implementing the iPhone application dwarf anything else.

On the other hand, if your business is selling ringtones or news, then you need to reach out to the widest possible audience, regardless of the devices they're using. A solution that reaches this objective with the lowest cost is your Holy Grail. Today, this means developing a solution based on HTML and JavaScript.

## Facebook Was Not Built in One Day

In mobile, as well as in any business, time to market is critical. In laying out your strategy, consider applying an agile schema that lets you release applications piecemeal. Figure 1-2 presents the canonical Scrum process adapted to mobile projects.

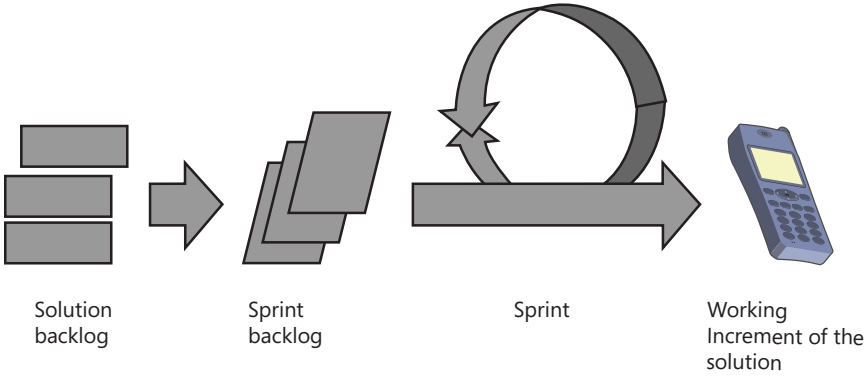



**FIGURE 1-2** A Scrum-like model for mobile solutions.

The entire set of features and applications ("product backlog," according to the Scrum dictionary, and labeled "Solution backlog" in the figure) is partitioned into multiple sprints or iterations. At the end of each sprint, you release a working segment of the entire application (such as an iPhone application) and then are ready to reiterate the same process for another sprint (for example, an equivalent Android application).

More often than not, sprints for mobile solutions also include the following:

- Arranging a website that's usable by both mobile applications and sites. This means exposing the core functions of the website as easily callable, Representational State Transfer (REST)–based, HTTP endpoints. For example, if you're building the website using the ASP.NET Model-View-Controller (MVC), this may mean exposing an ad hoc controller that can serve requests based on the use-cases that you implement in mobile clients.
- Developing a set of pages (scripts, styles, graphics, and presentation logic) for a class of mobile devices. You may want to start with high-end devices and proceed downward to enable more and more lower-end devices to access some fraction of the full site functionality.
- Optimizing the behavior of these pages with more accurate device-detection capabilities.
- Developing native applications for most popular mobile platforms.
- At each level, you can propagate valuable user feedback through the entire stack of applications you've built thus far.

To paraphrase a popular saying, "Rome was not built in one day." I'd say that Facebook was not always the huge platform we know today, either—after all, it's been around only a few years. A mobile solution, therefore, will look increasingly like a small platform of integrated services; it requires hard work and overcoming many challenges to complete.

## Delivery Models

A B2C application is (ideally) distributed worldwide. The costs of spreading the word about its availability (not advertising...) are entirely up to you. A website is immediately available from any place in the world, but again, the costs of spreading the word about it must be borne entirely by your organization.

In the mobile world, appstores rule over the publication and distribution of platform-specific applications. You publish your application to the appstore, giving it instant exposure to users of a particular platform. Each device ships with an applet so that users can access the platform's appstore—where your application gets published. Users can then access your application, read release notes, check requirements (such as that your application requires Internet access, phone calls, text services, local storage, and so on), see some screenshots, test-drive a trial version (if available and supported)—and what then?

What do you expect the return from investing in a mobile application to be? More generally, how do you expect to recover the costs of developing a mobile site and/or a few native applications? That's another part of overall strategy that management has devised.

**Note** Here, I'm talking about "spreading the word" and "publishing," which you get for free for the minimal costs of being a registered developer with the platform of choice. Advertising your application in and out of the appstore is another story entirely.

## The Free/Paid Dilemma

Mobile applications are typically very cheap when they're not entirely free. The cost of the average iPhone application is around $2—even less for games. The average iPhone user is expected to download (and pay for, if that's required) about 80 applications in the course of a year.

Paid applications generate direct income subject to the marketplace tax (and, of course, government taxes). Free applications are generally built for marketing and branding purposes, or as an additional form of customer service.

After reading analysis and projections, expert opinions, and analytics, I formed the idea that mobile applications should be free; they need to generate revenue in some other way. However, if you're an individual or a small company and happen to have a stand-alone (not bound to a strategic business plan) mobile application, why give it away for free? If it's a well-done application that fills a hole in people's mobile lives, you can likely recover your investment, and perhaps even more.

A third option is advertising-supported applications, which are free for users but generate revenue for the author through dynamically inserted ads. Switching to a paid or ad-based model is an important step. If you first release the application for free, you get a lot more downloads, which are good for feedback. It also helps you understand how well received your application is and whether it really fills a hole.

If you look into the most popular appstores such as the Apple App Store, Android Market, Windows Marketplace, and BlackBerry App World, you will find that there are almost always more paid applications than free applications. For example, in the Android market, free applications outnumber paid applications by about a 60/40 ratio.

The free/paid dilemma is not really a dilemma with a binary, black-or-white answer. There are a few other models that mix free and paid content according to different recipes.

## The Freemium Model

The Freemium model is based on the idea that you provide the full application free and then offer users the chance to buy a few extra services. From a realistic business perspective, however, the Freemium model means that the vast majority of your users will consume your application for free; only a minority will pay for any extra services.

So how can this model be worthwhile (financially speaking) for mobile applications? First and foremost, you need a lot of users, preferably on the order of millions, and at minimum on the order of tens of thousands. Maintaining all these users probably has a cost as well. For example, if you need to maintain a website to provide data to the mobile application, then you have a growing cost directly related to the number of users. Even if your application can run as a self-contained device application, you still may have some costs per user because you have to support users and reply to their emails.

An excellent example of a mobile application for which the Freemium model is perfect is Evernote (*http://www.evernote.com*). These mobile applications work entirely on the devices they target; all they need is storage space. According to *http://blog.evernote.com/2011/01/04/evernote-2010-a-year-in-stats*, Evernote has more than 6 million users. Of those, only 3 percent pay an extra subscription fee.

Another example is Searcheeze (*http://www.searcheeze.com*), a new startup that offers collaborative search. Users, both groups and individually, can run and publish a search on a given topic. This search—realized by humans, not search engines—may be left free or published to an internal marketplace. Becoming a Searcheeze user is free unless you want to buy extra services, such as installing a private engine on your company's servers.

## The Premium-with-Free-Sample Model

The premium-with-free-sample model is fairly new in the media industry, but it's already the model toward which most content providers and newspapers are moving. Basically, it consists in making a significant portion of the content available for a small fee but leaving a fraction of the content free for everybody to access.

*The New York Times* pioneered this model. It currently gives you a number of free articles per month, after which you have to pay a fee to access more content. In contrast, the *Boston Globe* locked three-quarters of its digital content and offers free access only to the remaining part. Repubblica.it, Italy's largest news site and second-best-selling newspaper, also uses this latter model. In addition, Repubblica.it charges for access to its mobile site. In contrast, the desktop site is free, but you need a smartphone to read it effectively.

It's worth noting that the *Boston Globe* mobile solution is based on a HTML5-powered mobile site, which maximizes the audience without incurring the costs of developing ad hoc mobile applications and, importantly, without paying the typical 30 percent marketplace tax to an appstore owner. Appstores, in fact, may impose ad hoc policies for in-app payment. During the summer of 2011, Amazon quickly modified the Kindle iPhone and iPad applications to comply with new Apple policies for subscription-based applications. At nearly the same time, the *Financial Times* application—a best-selling program—was pulled from the store because it was patently in violation of the store rules. As a result, the *Financial Times* now encourages customers to use its new HTML5-based mobile site, which—guess what—has been optimized for iPhone browsers and looks nearly the same as a native application.

## The Quid-Pro-Quo Model

As an Italian, I would have used another Latin phrase to express the same concept: *do-ut-des*. According to Wikipedia, the English usage of quid pro quo in fact matches the Italian usage of *do-ut-des* perfectly, meaning "I give so that I can receive."

This model is probably the one I feel most comfortable with. In my personal vision of the world, a mobile application exists as a complimentary feature, a favor that the publisher does for me. I reciprocate the favor by buying some of the publisher's other content or services.

The free applications are entirely free; there are no strings attached. To use them fully, however, you need to buy or consume some other services that the publisher relies on for income. Applications you use in an airport, during a tennis tournament, or at a conference are all examples that fall in this category. You get some services via the application in exchange for the simple fact that you're there (in airports or at conferences): you don't pay directly for these services (mostly information and news),

but you pay in some other way. For example, you probably paid to attend the conference or bought a ticket through that airport.

Here's another example of an application that I had the pleasure of knowing from an insider's perspective: I ported it from iPhone and Android to Windows Phone. The application is called Postino; you can find out more about it at *http://www.postinoapp.com*. Postino was originally built for the iPhone and then ported to a number of other platforms, one step at a time, as the result of a classic B2C strategy.

Postino lets you snap a picture as you travel and promptly creates a (virtual) postcard that you can send to a friend. The postcard contains a message, a signature that you draw on the screen with your finger, and an address. If the address is an email address, everything is free. If the address is a physical address, then you must buy a virtual stamp and upload the card to a server, which will print and send a real postcard.

An application built around a simple but good idea can be free, generate income in an indirect way, and still represent a success story for the developer (or the company), which may generate more business.

# Outlining a B2B Strategy

I certainly don't have the expertise and experience to embark on a comprehensive discussion about the differences between B2C and B2B. As far as mobile strategies are concerned, there's only one important difference: B2B often gives you the chance to choose one specific platform and vendor and stick to that. From a software company perspective, B2B means that you're helping another business set up a mobile infrastructure that will be used to serve a limited and largely controlled audience, such as the network of agents that operate in a given region.

For the purpose of this book, the difference between B2C and B2B is the same as the difference between a public Internet site and an intranet site.

## Serve Your (Limited) Audience

Let's review the main traits of a strategy aimed at serving the needs of just one business. The mobile interface is not open to the public; it's consumed by special customers, such as employees, agents, and consultants. Although you don't have to capture a large audience, you instead have a relatively small audience that you must serve in the best possible way. Forcing them to use one particular device or site is part of the deal.

### B2B and the BlackBerry Case

What made BlackBerry so successful and BlackBerry devices so widely used? Sure, it offers email, tasks, and calendaring; it may even support web browsing and a camera. You can do some instant messaging and run a few utilities from an appstore that is one-tenth the size of Apple's. But compared to, say, an iPhone, a BlackBerry device looks like a child's toy.

So why was it so successful (at least before the iPhone arrived)?

The answer lies in the enterprise-class features that it offers. In particular, a BlackBerry device can connect to an in-house enterprise server—the BlackBerry Enterprise Server (BES)—and receive email updates, news, and task alerts in real time. How is that different from today's Microsoft Exchange Server connectivity in Windows Phone? It's not, really; both are basically the same—but BlackBerry was available somewhat earlier, and companies liked its features. As a BES administrator, you can apply policies and prevent a class of users from using the camera or instant messaging; you can force them to use only certain applications or to navigate only certain sites. Moreover, you can install your applications directly to your BlackBerry devices; you don't need to distribute them publicly to an appstore first.

In a nutshell, BlackBerry was a platform created to help members of an organization collaborate with ease and effectiveness.

## Pick One Mobile Vendor

In a B2B scenario, a customer calls the software company and discusses requirements. The advisor has to figure out just one solution that provides the requested services in a mobile way. Most of the time, there are no constraints on existing devices and hardly any constraints to address on the platforms.

If you need a better mobile infrastructure to make employees collaborate, you probably have no reason to build an iPhone application. In addition to the costs of development, you also need to account for the costs of providing an iPhone to all your employees. I can think of a few companies who just did that—but I consider them the exception rather than the rule.

So in a B2B scenario, you should select just one vendor and platform and stick to that. From the customers' perspective, costs are clearly lower, and development time is traceable. Which vendor you settle on depends on a number of factors, including the existing base of devices, deployment needs, special security or middleware constraints, existing skills, and, of course, overall cost and personal preferences.

I'll return to this in a moment, but I think it's important to call attention to that point, because in a B2B scenario, the mobile vendor is not simply—and not necessarily—the vendor of a mobile operating system and API. In some cases, the candidate vendor doesn't even have its own mobile operating system. Instead, it offers its middleware with a bunch of platform-specific presentation layers for users to consume data and applications. According to Gartner's Magic Quadrant for 2010, Sybase is an excellent player in a B2B scenario—and Sybase doesn't have an operating system; instead, it provides a strong and powerful middleware for mobile clients.

## Private Applications

When a company's goal is to build mobile solutions for its workforce, any applications that it develops should be private. A private mobile application is a mobile application that can be installed directly on one or more devices, with no intervening appstore. Consider, for example, an iPhone application written to serve the needs of a particular customer of yours—such as an application for sales agents.

That application is likely built to reflect the use-cases and business processes of that customer. It may have integrated some strong authentication policy. You don't want it to go to the marketplace,

and you don't want others to even look at it, let alone try it or buy it. You want it to work like Windows—you create an application, prepare an installer, run the installer on the machines you want, and that's it—you're done.

Private mobile applications are possible, but the process is not identical across the various platforms. In this regard, Android and BlackBerry are open: you can install just any application on just any device. For BlackBerry, this freedom of installing applications can be controlled and restricted by BES administrators. In Android, the only controller is the owner of the device.

Apple has a special enterprise program that, at the cost of $299/year, allows you to distribute applications freely within the members of your organization, whether through an intranet webpage, a network share, or email channels. Windows Mobile—the predecessor of Windows Phone—is as open as Android; Windows Phone still lacks an enterprise program. Currently, the recommended approach for simulating a private, company-wide marketplace is to make the application public and free and implement logic that unlocks the application only for users who have a specific Personal Identification Number (PIN).

## Mobile Enterprise Application Platforms

In a B2B scenario, you typically choose a mobile vendor by analyzing its mobile enterprise application platform (MEAP). A MEAP indicates the entire stack of mobile technologies, products, and services that a mobile vendor (e.g., Sybase) offers.

### MEAP vs. Stand-Alone Applications

When building a mobile solution, you could proceed by building a few stand-alone front-end applications that are based on an existing middleware or an ad-hoc back end and storage layer. But in doing so, you will likely end up using tools, services, and technologies from different vendors for the various phases of development.

MEAP is beneficial because by choosing a particular vendor, a company can often build a single back end and front end and deploy them to a variety of devices. The mobile device functions as a terminal that simply mirrors the content generated by the back end. A MEAP-based solution relies on proprietary middleware that you can customize and extend by writing applets using a few programming languages. The middleware serves data to the mobile client and controls both the user interface and local in-device logic.

With a MEAP in place, a company can expand its horizons with less effort—no need to invest in writing a new iPhone application; just deploy the same MEAP-specific application to the iPhone presentation layer. No changes are required to the underlying business logic, and the list of mobile front ends can be extended whenever the MEAP adds support for a new mobile platform.

In other words, a MEAP is an all-round business partner that specializes in mobile solutions. In this context, the classic iPhone or Android mobile application is just the tip of the iceberg—the real meat and potatoes are what lies under the surface.

## Gartner's Rule of Three

To explain the importance of MEAPs and, at the same time, give companies an easy way to check their affinity with a MEAP solution, Gartner developed the *Rule of Three*.

According to Gartner, a company should consider a MEAP seriously when the implementation of its mobile strategy requires three or more mobile applications for three or more mobile operating systems to be integrated with three or more back ends. It goes without saying that building a mobile platform from scratch with these requirements is a huge effort that probably requires a monstrous budget. In this context, a MEAP can introduce significant savings and—more importantly—keep the company at the forefront of the technology, ready to release new products in a fraction of the time that non-MEAP-using competitors might require.

## MEAP and Gartner's Magic Quadrant

How do you evaluate a MEAP? And more importantly, which vendors are actually MEAPs? Each year, Gartner applies its proprietary Magic Quadrant methodology to competing players in a given area—in this case, MEAP. The result is a diagram like the one shown in Figure 1-3.

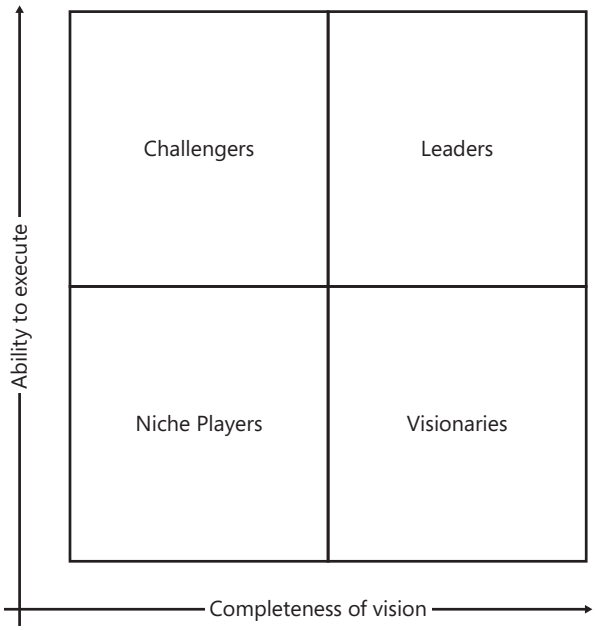



**FIGURE 1-3** Gartner's Magic Quadrant.

The rank the research returns for each evaluated player determines the coordinates in the diagram and, subsequently, the quadrant into which that player falls. In the paper published in 2011, the Leaders quadrant contains companies such as Sybase, Antenna Software, Syclo, RhoMobile, and Pyxis Software.

It is worth noting that, according to Gartner, the MEAP market is steadily heading for a $2 billion volume in sales. So where are the other big companies commonly associated with mobile solutions? To be a MEAP player, a vendor must have a comprehensive set of products and services to develop and test applications and offer security, (cloud) storage, notification, reporting, and synchronization. Microsoft, Apple, and RIM appear in Gartner's Magic Quadrant but are not considered leaders in this segment.

## Summary

No company can afford to ignore the mobile revolution taking place. Not all companies should proceed at the same pace or immediacy, but going mobile is a growing need and will soon be a necessity. The expression "going mobile" refers to the process of defining a strategic plan that sets business objectives that can be reached by restructuring internal processes, adopting innovative technologies, and developing ad hoc new applications to reach users who are traveling or to let one's workforce operate efficiently while away from the office.

This chapter outlined the main aspects of a mobile strategy both in a B2C and a B2B scenario. The next chapter takes a closer look at the two main ways of providing a mobile experience to users, whether customers or employees: mobile websites and platform-specific native applications.

CHAPTER 2

# Mobile Sites vs. Native Applications

*When it is not necessary to make a decision, it is necessary not to make a decision.*

*—Lucius Cary, 2nd Viscount, Falkland*

**In this chapter:**

- Not a Pointless Matter
- Aspects of Mobile Sites
- Aspects of Native Applications
- Summary

The modern era of mobile technology began with the release of the first Apple iPhone in the summer of 2007. No—that's not a mistake: this is exactly the same words that began Chapter 1, "Pillars of a Mobile Strategy." They are repeated here because this statement illustrates the origin of the argument being debated in this chapter so well.

Mobile has been heralded as the next big thing for a decade. Since the late 1990s, it has been repeatedly forecast as the vehicle for a huge innovation that's going to affect our lifestyles. Although the adoption of mobile devices has always been beyond a significant critical mass, the original forecast became reality only a few years ago, starting with the release of the iPhone in 2007.

Inevitably, the hype about mobile has always been bound to (and biased by) the use of a smartphone with its own set of native applications. But in the beginning, when forecasts about the upcoming mobile revolution were first being made, mobile meant something completely different; it meant having websites optimized for mobile devices.

Today, mobile sites and native applications are all too often pitted against each other, much like the playbill of a sensational boxing match. Is this a correct representation of reality? Is putting one "versus" the other a fair matchup? Let's find out.

To anticipate: The quick answer is that both mobile sites and native applications have full reason to exist. The head-to-head approach is primarily a forced interpretation of their differences. As you saw in Chapter 1, it's quite common to find both options (mobile site and native application) implemented in an enterprise-class mobile strategy.

## Not a Pointless Matter

Native applications and mobile sites represent different ways to implement a mobile solution (in whole or in part). Analyzing and understanding the pros and cons of each approach is definitely a good thing, but approaching the topics as a contraposition between two mutually exclusive approaches is pointless. Being an advocate for one camp is fine; preferring one approach over the other is also fine; but making a hard-nosed business decision is never a matter of preference. If you understand the mechanics of native applications and mobile sites, there's always a best route to take.

## A False Dilemma—But True Differences

If you type keywords such as "native apps vs. mobile sites" into any search engine, you will get tons of links, many of which discuss the pros and cons of each approach according to the vision of the particular author. Most of the arguments do make sense, and I recommend that you read the content of some of the first links that Google and Bing return in response to such a search.

The real point, though, is different: the comparison of mobile sites vs. native applications is generally a false dichotomy, but mobile sites and native applications do represent two valid options for a mobile solution. True differences do exist between mobile sites and native applications, and understanding them is the key to pinpointing the pillars of your mobile solution.

### Focus on the Right Question

Contrasting mobile sites versus native applications is not the right way to approach this topic because that approach attempts to find a tough answer to an irrelevant question. For the developers reading this, the subject is not much different than brainstorming whether JavaScript Object Notation (JSON) or XML is preferable when building a distributed application. During the application-building process, there probably will be a time when architects or developers need to make a decision as to whether to employ JSON or XML, but the more critical questions to solve are, for example, how to design the public interface of your services [i.e., Representational State Transfer (REST) vs. Simple Object Access Protocol (SOAP)]; which transportation protocols should be involved; whether security and transactions matter; and so forth.

Back to mobile: If "native apps vs. mobile sites" is not the most relevant question, then what is? As discussed in Chapter 1, you need to focus on the products, services, and audiences for which you are building. Next, you should detail objectives and focus on the budget, resources, and timeline. When all this is clear, you can move on to the architectural side of the project. At that point, you will probably discover that you have just one option left.

At any rate, a few true differences exist between a native application and a mobile site. The next sections focus on the main traits of both.

## The Main Traits of Native Applications

A native application must play by the rules of the host operating system. This means that it's subject to restrictions (for example, limited storage capabilities and processing power). On the other hand, it has the advantage of being able to use location and built-in services such as push notification services, Short Message Service (SMS), and camera, use the full capabilities of the touch user interface, and rely on a seamless install/update mechanism.

Many people tend to justify staying in the "native application" camp with the alleged fact that users prefer native applications. All in all, you should take that as an arbitrary opinion. Although a native application (or lack thereof) may certainly have a marketing impact on your brand, in terms of functionality and features, here are the main traits:

- A native application has the best chance to integrate well with the device and use hardware and built-in software services.
- A native application is much less exposed to network latency.
- A native application may be able to work on totally disconnected devices.
- A native application is an on-premises program, so it doesn't require users to deal with URLs.
- A native application typically offers a native user experience.
- Users download native applications as an all-encompassing package in a single network request.
- A native application must be created for each mobile platform that you intend to support.

**Note** A few of these points deserve further explanation. First, however, I need to note that when I say "native application," I'm deliberating omitting game applications. Typically, game applications have a completely customized user interface and user experience, so some of the points made here (such as the native user experience) simply do not apply to mobile games.

One aspect in which native applications have a clear advantage over mobile applications is their integration with built-in hardware and software services. A native application can be written to use the specific device capabilities, overall offering an unmatched user experience. Controlling camera and device buttons, accessing global positioning satellite (GPS) services and telephony application programming interfaces (APIs) such as SMS, phone calls, and contacts, is a clear plus. But not all applications require such capabilities. In other words, even the first point of the list—which is by far the most compelling—owes its importance to the actual application requirements set by the stakeholders.

Another trait that highlights a remarkable difference between mobile and native applications is the overall user experience. Elements of the user interface (e.g., controls) are a key part of the user experience. A native application can use native user interface elements and the API that implements the device-specific interaction model. In the end, a native application automatically provides the same user experience as other native applications. With a mobile site, achieving a gratifying user experience is a bit harder to do, and the costs of implementation are on the developer's shoulders.

Finally, I wouldn't say that a native application is *necessarily* faster than a mobile site, although it's true that native applications are much less subject to network latency and can recover from network failures more elegantly. Local storage is another feature that provides an advantage to native applications, even though it is a feature that some mobile sites can match if they employ HTML5. (You'll see more about this topic later in the book.)

**Note** The perceived speed of mobile sites depends on the browser and the markup and script that you put on the page. The processing power of mobile devices is not the same as a laptop, so running heavy scripts may be problematic. Animations and page transitions that are coded on a mobile webpage don't offer the same smooth and pleasant experience as a native application does.

## The Main Traits of Mobile Sites

A mobile site is a plain website, just one that is optimized to serve an HTML layout that fits in the tiny real estate of the mobile device screen. Any website can be displayed on a mobile device browser so long as that browser is designed to process the received markup. Most mobile phones today—and all smartphones—can display desktop sites well by simply shrinking the content. Figure 2-1 shows the results of viewing a very simple site such as *Bing.com*.


**FIGURE 2-1** A smartphone shrinks sites that are too large to fit within the available screen real estate.

For users to be able to click links within the page, the ability to "zoom in" is a necessity. Most people are used to zooming, but it's definitely not a contributor to the best possible user experience.

A true mobile site lives at a different URL, whose naming convention is typically *m.xxx.com*. A true mobile site provides a selection of the content in the desktop site served in a manner that complements the capabilities of the mobile browser, including screen real estate. Figure 2-2 shows the mobile version of the same *Bing.com* site.

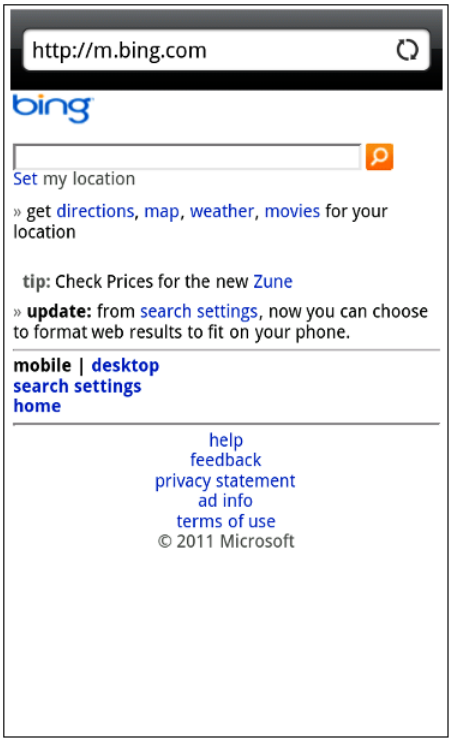



**FIGURE 2-2** The mobile version of the *Bing.com* website.

The page size is considerably smaller; there are no images in the background, meaning that there is no extra Hypertext Transfer Protocol (HTTP) request for the image, and a simpler layout highlights the most crucial use-cases.

**Note** Optimizing content for the capabilities of mobile browsers is a huge topic that the next two chapters will explore. Furthermore, the choice to use two distinct URLs for the desktop and mobile site or to switch automatically to the mobile version if the requester is a mobile device is the architect's decision. However, in some cases, this decision may be influenced by nonfunctional requirements. For example, a content provider company may decide to offer a free desktop site but ask users of the mobile site to pay a fee.

Here are the main traits of a mobile site:

- A mobile site can't access native device features such as the accelerometer, gyroscope, camera, audio, and tactile feedback.
- A mobile site works by interacting with the web server; subsequently, it's subject to network latency and may be sensitive to high-traffic slowdowns.
- A mobile site may not be able to do offline work.
- A mobile site can be reached only by typing the URL or creating a bookmark.
- A mobile site is based on pure HTML (although some sites style the HTML elements to mimic the user interface of a particular mobile operating system).
- A mobile site is subject to the browser's caching and rendering capabilities, which are not the same for all devices but are nearly the same for the major mobile operating systems.
- A mobile site is inherently cross-platform; it's created once and at most may need some ad hoc skins for specific mobile devices (such as the iPhone).
- A mobile site offers nearly the same Search Engine Optimization (SEO) benefits as a desktop site.

If you limit the discussion here to the native browsers of the most popular mobile platforms (iPhone/iPad, Android, Windows Phone, and BlackBerry), including additional browser applications that you may install on these devices (such as Opera Mini), they collectively offer a set of features rich enough to let a mobile site implement local storage, geolocation, and touch events. This rich set of features is provided by the WebKit engine (see *http://webkit.org*) and can be tested using the Modernizr JavaScript library (see *http://www.modernizr.com*). The next two chapters will explore these topics further.

The WebKit engine can't make the user experience for a mobile site fully equivalent to that of a native application, but it definitely provides users with something similar that's tailor-made for a mobile device.

**Note** Chapter 11, "Mobile Applications with PhoneGap," discusses a JavaScript framework for building native applications for a few mobile platforms, including iPhone and Android. This framework is called PhoneGap (*http://www.phonegap.com*). With PhoneGap, you write HTML pages styled with Cascading Style Sheets (CSS) and activated with JavaScript code and optional libraries such as jQuery, jQuery Mobile, and other touch libraries, such as Sencha Touch. PhoneGap offers its own JavaScript API that communicates with a platform-specific engine and allows developers to control cameras and other hardware devices with JavaScript code. It should be noted that this capability is reserved to PhoneGap applications and can't be achieved in the normal context of a mobile site.

### A Sure Differentiator: Offline or Online?

All in all, and even beyond the user experience, the biggest difference between a native application and a mobile site is that the former can be available at any time, even when the device is not connected to any network. A mobile site may hardly be available if the device is disconnected; and anyway, the availability is subject to the caching rules of the browser. On the other hand, if the mobile site makes intensive use of HTML5 features—specifically local storage and offline caching—then it is possible that the site degrades gracefully in a disconnected environment without displaying the ugly empty page of the browser.

**Note** It is not unusual for a user to configure the device to use only WiFi. This means that these users accept being disconnected from the network occasionally. For these users, a native application that works most of the time when disconnected and synchronizes with a remote server when connected is probably the best option.

## Reasons for the Perceived Dilemma

Before I delve a bit deeper into the aspects of mobile sites and native applications that can guide your decisions (development issues, type of application, and pros and cons), let me share a few more thoughts about the native vs. web dilemma.

### Perceived or Real?

I clearly said that in my opinion, the mobile site vs. native application dilemma is the wrong way to approach the development of a mobile strategy. Yet this is a very common question raised in forums, training sessions, and interviews, and explained in blogs and articles.

After companies have outlined which business objectives they intend to pursue with a mobile strategy, they need to figure out whether they need both a mobile site and a full set of mobile applications, or if they would do better to concentrate their available resources on one fully funded project. At this point, companies wonder what's best for their users and call experts to find the answer.

The question has just one right answer: *It depends*.

Although technically correct, this is not an answer at all, because it doesn't really address any of the original concerns and questions. A concrete answer *depends* on the business goals, objectives, and overall mobile strategy.

So the right way of asking the question is: Given the following objectives and strategy, and these specific resources and budget, what would you recommend?

That's the tricky point.

## Too Many Forces

As you probably already understand, whatever answer one can provide to such a question needs to be considered over a number of approval sessions. At each level, however, a different force may be applied to stretch the result toward a particular direction.

Some executives are impressed by that really great iPad application they've just seen. So they definitely want to have an iPad application for themselves, and maybe even an iPhone application for the customer base.

Then there are program managers who mostly consider the amount of work involved and concentrate on possible resource shortages. They see that they may need people with various skills to do the work, and wonder whether they would need to outsource development to other companies or hire freelancers (both of which always have elements of uncertainty). Program managers also recognize the complexity of having to deal with multiple projects (iPhone, iPad, and Android) and the synchronization required across the teams to add new features or change existing ones, and to fix bugs.

Next, are the developers, who almost always have a strong preference for one camp or the other. It's the nature of developers to convince themselves that what is cool and geeky is also the most useful solution for the company. (This is just another instance of the Pet Technology anti-pattern.)

Finally, there are the users, who have the bad habit of demanding more and more functionality every day, no matter what technology they're using.

The result is that people in these different camps tend to oversimplify. Hence, in every company at the end of the umpteenth meeting, someone stands up and asks: so what are we doing? Are we building a mobile site or a native application? And the question, still unanswered, goes to StackOverflow.

**Note** Here's a real anecdote reported by a customer while explaining the company's needs for a mobile class. The boss showed up and said, "We think we need some mobile applications; why don't you guys take some classes and figure out what to do?" The team replied, "Well, we practiced a bit with some mobile platforms; we can port our internal applications to Windows Phone in days!" And the boss said, "Nice, but we want iPhone apps."

## More Concrete Reasons to Waver

In the minds of most developers and managers, going mobile requires no more (and no less) than planning an iPhone application. The reason for this appears at the very beginning of both this chapter and the previous one.

Going with an iPhone application has been a good approach for the past few years. Today, however, and in the future, I doubt that's still a sufficient strategy. When a company develops a thorough mobile strategy, the result may be that having an iPhone application is not the first step; in fact, it may not be a step at all.

The growing support for HTML5 and JavaScript libraries means that mobile site solutions become more powerful all the time. The possibility of linking a mobile site to a framework like Wireless Universal Resource File (WURFL) for accurate device detection adds another arrow to your quiver.

And finally, there's the marketplace tax mentioned in Chapter 1. From the real world, I'll select two great examples of companies who left the native application route (specifically, iPhone) to embrace a mobile site solution based on HTML5: the *Boston Globe* and the *Financial Times*.

My gut feeling is that the potential of mobile sites in combination with HTML5, WURFL, JavaScript, and CSS has been only partially explored.

# Aspects of Mobile Sites

A few pages ago, I outlined the main traits—positive and negative—of a typical mobile site. Let's look into this subject a bit closer to find out more.

## What's Good About Mobile Sites

A mobile site is not a perfect solution for every possible scenario, or else I wouldn't be discussing it here and this book would probably have another title and contents. A mobile site, however, has some very interesting aspects that you want to evaluate with due attention while looking at the ideal steps to implement a strategy.

### The Server-Side Solution

First and foremost, a mobile site is a website. As such, it is hosted on a web server and the development is entirely based on production of HTML pages and JavaScript code. This means that the development team doesn't need to become familiar with any new API, programming language, or, in some cases, even a new computer (such as a Mac). You use the products and technologies that you know are adept at creating websites, including ASP.NET, PHP, Java, and more.

Using a server-side solution has both benefits and disadvantages, as you'll see. However, I like to mention the benefit of keeping development costs low because such solutions require less code to write, test, and maintain. In addition, ASP.NET and PHP developers are more numerous (and probably currently less expensive) than iPhone or Android developers. The real value of a mobile site, however, stems from its design and architecture. And those are not so cheap.

### One Site Fits (Almost) All

A mobile site is *one* solution deployed to a web server and accessible to any mobile device via the installed browser. You don't strictly need to create a mobile website for iPhone, another for Android, and yet another for BlackBerry or Windows Phone 7.

Sounds like the perfect trick, then. Unfortunately, things are not as smooth as those words may make it seem. (Note, I don't really think I made any inaccurate or wrong statements here; it's just that I temporarily omitted a few side notes.)

In particular, because of its inherently cross-platform, web-based nature, a mobile site can be written once and then viewed from everywhere, including mobile devices. However, not all mobile devices are the same and, as mentioned, device fragmentation is huge. So, to make sure that you reach your audience properly, you may need to consider adding multiple "skins" to your mobile site, one for each class of devices you're targeting. This is probably not as costly as addressing different platforms with a native application, but it is certainly more expensive than writing a single desktop-style website.

The bottom line is that the slogan "One-site-fits-all" is not entirely accurate in the mobile space due to the enormous fragmentation of device capabilities. In the end, you may find yourself optimizing (but not creating) the site for iPhone, Android, Windows Phone 7, and more.

**Note** Device fragmentation is a real and painful problem, but it won't affect you much as long as your strategy is focused on smartphone users. In other words, if you don't care much about reaching owners of low-end devices, the mobile site development model is nearly the same as for a desktop site. However, the implementation of a mobile site should adhere to a new set of patterns and practices.

### Hassle-Free Deployment

After you begin offering your services through a mobile site, any updates you have must be deployed only to the production server to update all users immediately. Users just need to type the URL of a site every time they want to access it, or else bookmark it—no more and no less than they need to do on a PC.

**Note** Users of a mobile site never receive alerts that a new (or just fixed) version of the application is available and waiting for them to download it. When you update the site, users passively receive the update. Often, users don't even see a list of changes or new features. This means users have little control over all sites (not just mobile sites). In contrast, native applications tend to be much more user-centric than sites.

## What's Bad About Mobile Sites

The effect of the contrasting forces mentioned earlier in this chapter produced the equation "mobile equals iPhone or a native application." It's possible to save a good deal of work (and money) by implementing a mobile strategy, but mobile sites are not as easy (or quick) to arrange as one might expect. The next sections discuss the main drawbacks.

### No Access to Hardware Capabilities

Just as desktop website can't interact with the hardware and file system of your laptop or PC, a mobile site can't just interact with the hardware of a mobile device, including the camera, accelerometer, and radio system. A mobile site is sandboxed in the mobile browser, in much the same way that a desktop site is sandboxed in the desktop browser.

It's up to the browser to release some of these constraints—and as more browsers add support for HTML5, local storage and geolocation are no longer taboos on either desktop or mobile browsers.

## Varied Browser Capabilities

A mobile site is restricted to run in a mobile browser. Although you write a mobile site with generic technologies, such as HTML, CSS, and JavaScript, there's no guarantee that a given page will look the same across the full spectrum of available mobile browsers. Does that sound familiar? It should, because it's the same problem that site developers faced for desktop web browsers for years (the "browser wars") and that was resolved (or at least significantly mitigated) only with the advent of powerful JavaScript libraries such as jQuery. Of course, you can use jQuery and the excellent jQuery Mobile in mobile sites as well, so what's the problem?

The problem is that device fragmentation—the number of devices with different capabilities (some 7,000+)—is huge. And there's also a huge number of capabilities (500+) that may be relevant to check for each device. Currently, you can't check some of these capabilities by querying the browser because some browsers are not reliable in their response and because the browser's object model doesn't expose any property that can provide a direct answer. I'll explore this point further with a brief example now, and then I'll return to this discussion in Chapter 6, "Developing Responsive Mobile Sites."

It is critical for mobile sites to try to reduce traffic because there's no guarantee that users have a flat rate or are connected via a wireless network. In other words, the bandwidth your site requires could cost your users a lot of money. For example, all browsers send an additional request to the server when they encounter an image that they must download and display. Most browsers, however, have the ability to process inline images, rendered as Base64-encoded strings and embedded in the page. No browser is currently able to tell you that. As you can see, knowing about this capability can be critical for mobile sites that aim at a very large audience. (Chapter 4, "Building Mobile Websites," will discuss WURFL as a way to address device fragmentation.)

The bottom line is that site development must address device fragmentation. The spread between browsers, however, is significantly reduced if you limit your target to smartphones and browsers that support the WebKit engine.

## Network Latency

A mobile site runs by navigation; each link that users follow may generate additional traffic to and from the device. As mentioned previously, additional traffic may be expensive for users, so the architect's responsibility is to minimize the need for traffic. But how?

Using inline images and sprites are nice tricks, as are minifying and aggregating scripts and CSS files. Another trick is to use a Content Delivery Network (CDN) with subdomains for static resources. Yet another possibility is to cache data locally (according to the capabilities of the actual browser). If the browser supports HTML5, then local caching is no big deal; you can do that quite comfortably. Otherwise, your storage capabilities are very limited.

## Web-Based Navigation

If the device is a touch device, following a link is as easy as pointing a finger and tapping. If not, usability decreases significantly as users are forced to use other navigation techniques such as relying on *focus* (the highlighted text has the focus) and a *cursor* (moving a cursor among clickable items using the arrow keys).

But navigation is hardly optimal, even on touch displays. The finger, in fact, is not a stylus, and users often end up touching the wrong link. For this reason, mobile sites should be designed very carefully, and ideally, they would follow the guidelines of device vendors. Most of the time, these guidelines are available only for native applications, but they are useful because they reflect some general usability rules that work for mobile sites as well. Adopting any measures that limit the input effort for users and avoiding crowded pages with more than three or four links are two key best practices. Another is to design the site map so that people can reach each destination with no more than three taps. You'll learn more about common practices for mobile site design in Chapter 4.

That being said, I have noticed that the majority of mobile sites offer a poor experience, which is a good reason for architects and developers to focus more on the design of effective m-site functionality in the future.

## Appstores and Payments

Websites are simply deployed to a server; they're not like applications, which can be packaged and offered through an appstore. This means that users won't find an icon on their launch pad that starts the site. All this makes mobile sites less enticing to many users, but it probably is not a valid reason to drop them from your strategy.

A mobile site has total freedom to require membership and accept payments—even through mobile devices. This is both good and bad news. It's good because you are not subject to marketplace taxes, and there's no risk that someone will come your way asking you to share subscriber details. But it's bad because you have to add the development costs of a membership system and payment platform and lose the instant visibility to millions of users that popular appstores can offer.

It should be noted, though, that the additional development costs for a custom payment system apply only once. You build this system in the back end of the site, and it will work for any mobile client that connects to the site. In contrast, adding a payment system to a native application requires more work because you have to code it for each supported platform—and you'll likely need to learn a new API every time. On the other hand, a positive aspect of appstores is that you don't have to manage hosting, advertising, or deployment costs.

## Audience for a Mobile Site

If you look at absolute numbers, you can conclude that by using mobile sites, you can potentially reach a larger share of users. But how many of them are *your* users or are interested in *your* services? (That's what the phrase "relativity of numbers" means.)

Although building a mobile site as the first sprint in the implementation of a mobile strategy is hardly a bad choice, there are scenarios where other options are probably more effective. So it really depends on the type of application you're building.

For example, if you're serving news or content that any individual may be interested in, then a mobile site provides both a cost-effective solution and a return in terms of image. Recently, I went to a popular tennis tournament with a friend. My friend could use a custom iPhone application to get live scores and news from all the courts. With my Android, however, I had to be content with the shrunken version of the desktop site, meaning that I spent most of my time zooming in and out. A well-done mobile site would have offered nearly the same experience to everybody, regardless of the device.

# Aspects of Native Applications

The main trait of mobile applications is that they are platform-specific. An iPhone application is simply an application that works on only a particular family of devices. However, for many companies, mobile applications and iPhone applications are essentially the same.

## What's Good About Native Applications

What's good about native applications is often what's bad about mobile sites, and vice versa. Let's find out more.

### Fast and Fully Integrated

A native application is usually as fast as the hardware allows, with no slowdown due to network traffic and latency. Most native applications work by placing HTTP calls or accessing streamed data over some protocol or web service. This interaction can certainly introduce a delay, but for native applications, this delay is limited to specific operations; it doesn't affect the rendering of the user interface or in-app navigation.

Native applications have full access to all sorts of hardware devices and, more importantly, to software services such as background agents, broadcasters, and push notification services.

**Note** The availability of background agents and services varies considerably, depending on the platform. For example, applications running on iOS and Windows Phone face many more restrictions than, say, on Android.

### Appstore Integration

Users acquire or purchase native applications through a special store. That's not only good for users but is often also great for companies, because users have a defined place to look for applications. The appstore provides search capabilities, categorization, and exposure, as well as handling payments.

From the user's perspective, an application that comes from an appstore is certified and guaranteed to work and be safe. The appstore can't guarantee the quality of the application itself, but for that, the appstore infrastructure offers a user-driven feedback engine that can help potential users make better decisions.

Depending on the platform, the appstore may be the only way (or just the easiest way) to get a native application. For example, Apple and Microsoft make using their appstores mandatory (see the Note), and provide specific software (iTunes and Zune) to synchronize content between computers and devices. Google (Android) and RIM (BlackBerry) have their own appstores, but users are also free to install whatever they want from wherever they like.

**Note** Devices based on iOS (Apple) and Windows Phone 7 (Microsoft) can be unlocked in a number of legal ways or through some free software that *jailbreak* the device. In the summer of 2010, the United States declared jailbreaking legal, over Apple's objections. As a user of a jailbroken device, however, you lose support from the manufacturer.

### User Experience

A native application is called *native* because it shares the same set of controls and user interface widgets used by original equipment manufacturer (OEM) applications. Therefore, it is possible to make the overall user experience for native applications superior to anything else, including mobile sites and mixed applications (such as those written using frameworks like Rhodes, Titanium, and PhoneGap).

Most native applications perform common tasks in much the same way. For example, the process of picking a date or displaying an alert box functions the same across applications—thus generating a feeling of continuity for users. And, finally, let's assume that users love native experiences that make their particular phone cool. Often, these users are the same top managers who determine the mobile strategy of the company.

## What's Bad About Native Applications

At the cost of repeating myself, the main trait of mobile applications is that they are platform-specific. This is bad news for companies addressing mobile development.

### Isolated Mobile Continents

Each mobile operating system is a closed environment. An application developed for iPhone can run only on another device that hosts a compatible version of the same operating system: iPad and iPod Touch. There are no chances that the same application can run, full or sandboxed, on an Android or BlackBerry device.

It means that to develop the same applications for multiple platforms, you need to learn different languages, programming frameworks, and development tools, and, in some cases, also buy specific hardware or install specific operating systems to develop. Table 2-1 lists programming languages, frameworks, tools, and hardware required to write applications for the different mobile platforms.

**TABLE 2-1** Equipment for Developing Applications for Most Popular Mobile Platforms

| Platform | Computer Operating System | Tools | Programming |
|---|---|---|---|
| iOS | Mac OS | Xcode<br>MonoDevelop<br>AppCode | Objective-C and iOS SDK<br>C# and MonoTouch |
| Android | Mac OS<br>Windows<br>Linux | Eclipse<br>IntelliJ IDEA | Java and Android SDK<br>C# and MonoDroid |
| Windows Phone | Windows | Microsoft Visual Studio | C# and Microsoft .NET Framework 4 |
| BlackBerry | Mac OS<br>Windows<br>Linux | Eclipse | Java and BlackBerry SDK |
| Symbian | Mac OS<br>Windows<br>Linux | Qt Quick | C++ and Qt SDK |

You'll see that the "Tools" column of the table lists a couple of third-party products, such as MonoDevelop from Xamarin and AppCode and IntelliJ IDEA from JetBrains. In the "Programming" column, you'll find a reference to MonoTouch and MonoDroid, which are frameworks from Xamarin.

Cross-platform mobile development is the Holy Grail that everybody is seeking these days. I certainly don't expect to see one common mobile platform, either now or in the future. However, a common language that lets developers share at least pieces of logic across various applications is already close to reality today. That language is C#, which can be used for Windows Phone, iPhone, and Android through Xamarin's MonoTouch and MonoDroid platforms. HTML5 and JavaScript are other valid options that reduce the distance between mobile continents. Today, frameworks such as PhoneGap and Titanium make it relatively easy and cheap to migrate HTML projects between the various mobile platforms, and such applications provide an excellent mix of web and native features. The topic of native application development will be covered in more detail later in this book, in Part III, "Mobile Applications."

While waiting for true cross-platform development possibilities to appear in the future, enterprises today face multiple development costs for each platform that they want to target. Costs are a function of both acquiring skills and of reuse—of both code and skills.

The main purpose of this book is not to make you a great iPhone or Android developer, but a better mobile developer or architect in general, who can move quickly from one platform to the next. Such adaptability exemplifies the type of skill that is in high demand in the mobile arena.

## Minimized SEO

With a mobile site, the SEO tools you can use are the same as those for desktop web applications. Each page of your mobile site can be indexed and exposed individually to search engines, so every piece of your mobile content is reachable by users through search engines. Obviously, that's not true for native applications; however, any native application with a sound strategy behind it comes with a (promotional) companion website. The content in this website can be indexed and ranked properly by

search engines. The problem is not so much making your mobile application visible to search engines, but what SEO strategy you can apply to push them. You likely need to create a companion website solely to serve the purposes of increasing your application's visibility through search engines.

In a nutshell, the real point is that with a mobile application (and without ad hoc efforts through a website), Google or Bing will not return related links unless the searcher specifically enters your application or brand name. That's not necessarily a problem for everyone, but it is an issue to consider.

### Natural Targets for Native Applications

The range of mobile applications that lend themselves well to be developed as native applications is certainly larger than the range of applications that will work well on mobile sites. Games fall into that range, as are applications that can run locally, without needing connectivity. This latter category includes personal utilities (notes, to-do lists, calendars, shopping lists, and so forth) as well as telephony utilities (contact managers, history, phone usage, and so on). Finally, popular applications such as maps and front ends for social networks can provide the expected appealing user experience only if they are coded as native applications.

## Summary

Addressing the question of going mobile as if it were a simple matter of choosing either mobile sites or native applications is patently wrong and pointless. Increasingly, the future for companies involves planning a strategy that reaches the majority of mobile users—and you can devise a strategy by looking at numbers, both absolute and relative, budget, and how well it meets declared objectives. At this point, the implementation of the strategy may require a complex architecture and multiple nonsimultaneous steps.

You look at where your users—existing or potential—are, and figure out how to reach those users and draw in others as well. You may be able to achieve this with any possible combination of mobile sites and carefully crafted applications—some for specific devices. There are no hard and fast rules. Everything depends on context. I've been working in software architecture and mobile development long enough to be able to say unambiguously that there are no easy questions and no easy answers. Mobile sites or native applications are part of the answer, but making a binary choice between them is most certainly not the right question.

PART II

# Mobile Sites

CHAPTER 3

# Mobile Architecture

*Before I came here, I was confused about this subject. Having listened to your lecture, I am still confused. But on a higher level.*

*—Enrico Fermi*

**In this chapter:**

- Focusing on Mobile Use-Cases
- Mobile-Specific Development Issues
- Summary

About a decade ago at a conference, when a wireless-specific flavor of HTML was being pushed as the foundation of mobile site development, I heard someone describing the bottom line for websites and mobile websites by saying: "Well, the difference is more or less like Italian and Spanish. As a Spaniard or Italian, you can understand Italian or Spanish, but speaking it is quite another story."

That fits my perspective: a mobile site and a website have a lot in common, but they are also radically different. This sounds like an oxymoron, and to some extent it is an oxymoron. Just changing the layout of the master page and replacing a few Cascading Style Sheets (CSS) files may work in practice, but it doesn't lead to an effective and functional solution. Creating the site to be smart and responsive to changes of resolution is nice, but it mostly works only for desktop browsers. Such approaches have a number of performance-related drawbacks for mobile devices.

To design an effective mobile solution, you first must recognize that mobile devices and browsers are different tools operating in different environments. So you must find the proper solutions that work effectively in the mobile environment. Overall, building mobile sites is not that hard—I daresay it's even easier and faster than building desktop sites. The primary reason is that a mobile site is much simpler and needs to implement only a small subset of the features that you might want to have on a full-size desktop site.

However, especially in software, simplicity is a hard concept to expand on. In mobile, the boundary between what's *simple* and what's *simplistic* is often blurred. As an architect, your main responsibility is to make this boundary clear and ensure that the implementation doesn't trespass over it.

In this chapter, I'll outline the personal rules and practices that I've employed in mobile solutions to date. I say "to date" because that's just what I mean—mobile development is a relatively new field, and new and better practices may be discovered and applied every day. Overall, though, I believe that

the points below are quite generic and aim at the heart of the mobile design experience. Still, I urge you to take all this with a grain of salt and evaluate it from the viewpoint of your own needs.

## Focusing on Mobile Use-Cases

The term *website* needs nearly no explanation today—websites have been part of everyday life for the past decade, so I'll be brief. A website is a stable and consolidated solution that relies on a number of practices and resources, which—as developers—we don't even think about. For years, the content served by websites have been viewed through desktop computers and laptops containing a significant amount of RAM, huge disks, large screens, and mostly constantly powered by AC current. More importantly, the content served by websites has been viewed by users while sitting comfortably at a desk.

The key point is not so much whether you also can view a website on a mobile device, but whether—as a user—you really like doing so. Mobile devices, including high-end smartphones—are not the same as laptops or computers. Subsequently, the mobile web experience cannot approach the familiar desktop web experience unless you change layout, style, types of interactions, and the content served to a mobile audience.

That is, it can't unless you reconsider the mobile web experience from scratch.

### Stereotypes to Refresh

Most developers tend to look at a mobile solution as a spin-off of work already completed for the desktop website. I don't think the term *spin-off* is an incorrect word choice here; for the most part, it is just right. The devil is in the details of how you actually spin the mobile solution off from the website.

Lack of specific practices and expert guidance has propagated a few myths around mobile web development that I want to address now.

**Note** Recognizing myths is probably just a first step toward building a number of widely accepted practices for engineering effective mobile solutions. Hopefully we will be able to produce valuable guidelines from analyzing and recognizing these myths.

#### Myth #1: People Don't Like Mobile Sites: Why Bother?

This myth asks: Why bother designing and building mobile sites when people don't even know the difference? They may even complain if they're sent off to another view when visiting their favorite site.

I'm not surprised that most users don't recognize the difference between mobile and desktop sites—they just navigate to a site with a laptop, a tablet, or a phone and expect to view and use it in the best possible way.

For example, when they connect with a laptop, they expect to be able to use the entire screen real estate to do things. And they expect the same experience when they connect with a tablet or a phone.

Clearly, a phone doesn't have the same screen size as a laptop; it also doesn't have the same power or even necessarily the same stable connectivity. It's up to the development team to produce something that works well. I wonder whether the limited mobile traffic to sites isn't really due to the bad experiences that many of these sites offer when viewed from a mobile browser!

Also, users should not be asked to decide whether they want a mobile or desktop site. I see it as the code's responsibility to detect the capabilities of the device and adjust the user experience accordingly, while always providing a link somewhere so users can switch to the full site if they wish.

## Myth #2: You Don't Need Mobile Sites at All

This myth insists that because modern smartphones are powerful and feature-rich, developers can save themselves the effort of building a mobile site—the plain website will work just fine.

Frankly, even the most cutting-edge smartphone these days is still too small to view comfortably a page created for a desktop site. You can manage to read articles and post comments from a smartphone—I do that every day—but for regular use, there are a number of drawbacks. For one thing, you strain your eyesight and stress your neck muscles. Further, using desktop sites on a smartphone requires intensive use of pinch-and-zoom and continuously panning of the content. Is that pleasant?

Mobile devices have about one-sixth the physical screen real estate of laptops, and they have about half the screen resolution (if not less). These numbers do matter. A well-done mobile site should provide functionality equivalent to the full site without any need for users to zoom and pan. If that's not possible, it's better to just cut that function entirely.

Finally, the world is not populated only by smartphones, even though smartphone users probably drive most of the mobile traffic to websites. For your business, though, it might be crucial to be able to provide at least a subset of functionality to simpler devices, with only a fraction of the capabilities of an iPhone or a Windows Phone 7 device.

## Myth #3: A Tiny HTML Page Will Do the Trick

This myth recognizes the need for a mobile site to be different from the full site, but it oversimplifies the problem by saying that all you need is a plain HTML page with a bunch of links and bullet points.

I don't really know whether it's users who prefer to receive desktop pages and then proceed by zooming and panning, or whether it's developers who mostly address mobile needs through native iPhone and Android applications rather than with HTML, ASP.NET, and JavaScript. But the fact is, either enterprises need to take their mobile site development seriously as a stand-alone project (or segment thereof) or they'd be better off letting their users zoom and pan on their phones.

Serving any mobile devices with a basic, scanty HTML page with a fixed width set around 300 pixels is not a brilliant idea. It may be the only option for low-end devices, but it makes for an exceptionally poor experience on more advanced devices, which is what most users have now.

Mobile site development is about flexibility and finding the right balance between feature implementation and constraints set by hardware and software capabilities.

### Myth #4: One Site Fits All

This myth recognizes the need to have a site that serves both desktop and mobile users. However, it fails because it doesn't acknowledge that mobile sites can't just be simple spin-offs of the full site. So it takes the tack that all you need is to build just *one* website that automatically adjusts to different resolutions.

This is a subtle point. I definitely like the idea of a site that users can visit with any device (laptops, tablets, phones, e-readers, smart TVs, and more), but I don't believe you can achieve this realistically with a single codebase and by using client-side capabilities such as CSS and JavaScript.

In my opinion, we should aim at building *one web experience that automatically adjusts to different devices*. This is quite different from having one site that fits all possible devices.

To me, the idea of *just-one-website* doesn't work as a rule, but it may work as an exception.

## Analysis First

The interaction between users and the system in a mobile site is likely different than in the full-size version of the same site, for the same reasons mentioned previously in the discussion of the myths of mobile development: screen size, power, and sometimes unstable connectivity. You also should consider the frequent lack of a hardware keyboard (and even when a hardware keyboard exists, using it is not as seamless as with a laptop), and touch capabilities that may lead you to redesign the way some features are presented and implemented.

However, the top-tier reason for carefully selecting the use-cases for a mobile site is that users are likely to use the site in a different way. Sometimes the actions they want to perform are a subset of the full site; sometimes, instead, the two sets just overlap, and new scenarios must be added. Finally, the mobile site may drive the need for entirely new features.

This means that before you embark on a mobile site project, you should have a clear idea of which features you are going to implement, and you should spend an appropriate amount of time on a preliminary analysis.

The analysis of the features required for a mobile site is the most delicate part of this work. Coding is often fast because the number of features (and, subsequently, the number of sprints) is smaller than on a desktop site. In addition, you often can reuse a lot of the code and services of the existing application's back end.

### Selection of Use-Cases

Any application of any form and shape is built around a bunch of use-cases. A *use-case* is simply the description—more or less formal—of a possible interaction between the user and the system.

A mobile site is often a subset of a larger site or application. Hence, the mobile site exists to make it easier for traveling users to consume the content of the site. For a mobile solution, simplified access to the back end of the site means providing ad hoc user interfaces, location-aware prompts, and direct links. Overall, this means making an accurate selection of the (few) use-cases to implement.

A practical rule of mobile development states that nearly 80 percent of the desktop site is of little use to users who can't resist connecting to your site while using a mobile device. Your efforts should focus on use-cases, ordered by priority and selected through interviews, statistics, and guesses to determine that 20 percent of functionality that users would like to have constantly available at their fingertips. Focus on context. When designing functionality, ask yourself where users might be, why they might be using the site, and in what ways you can help.

The small number of use-cases automatically will make menus, lists, and navigation paths reasonably short and easy to reach.

## From Web to Mobile: A Practical Example

Figure 3-1 presents a demo version of a fully functional website for booking tennis courts. You can experiment with it live at *http://www.easycourt.net/contosoen*.

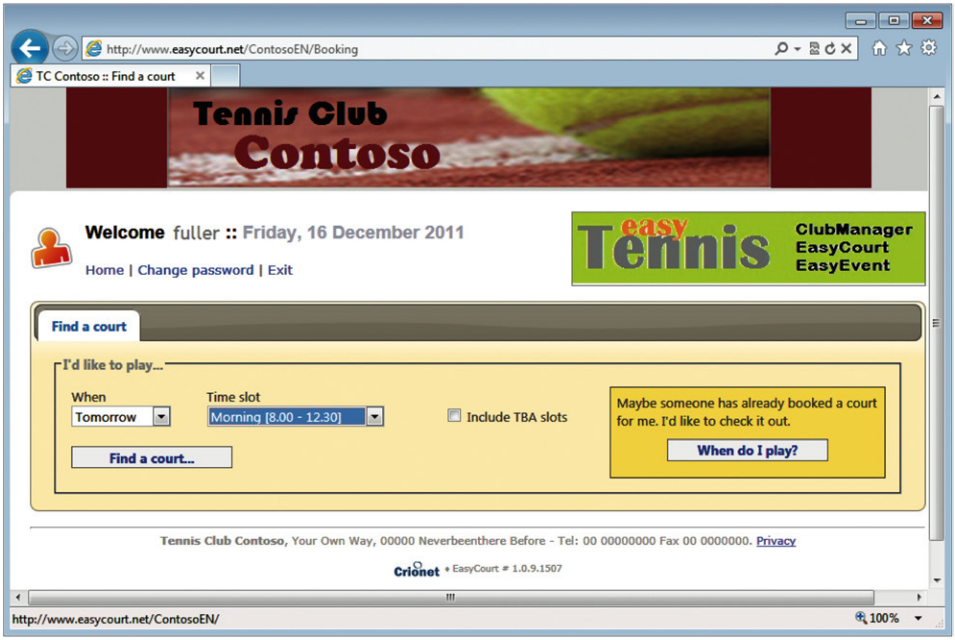



**FIGURE 3-1** The full site.

Figure 3-2 presents the same site as you would see it on an Android smartphone (zoomed a bit to become usable).

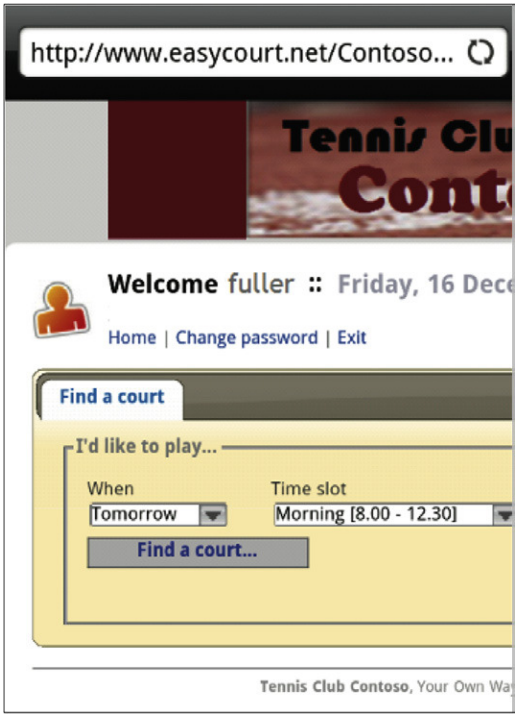



**FIGURE 3-2** The full site viewed on a smartphone.

Let's skip the obvious differences in the graphical layout required for a mobile page, and just focus on the functional differences that you should take into account. The home page for a logged-on user in the full site offers the functions listed in Table 3-1.

**TABLE 3-1** Functions of the EasyCourt Home Page

| Function | Description and Rationale |
|---|---|
| Log out | The user logs out.<br>*The feature is there because the site can be viewed on a publicly available computer (for example, in the clubhouse).* |
| Change password | A user changes his or her password.<br>*This is a required feature in any application based on membership.* |
| Book an available court | Users restrict the time slots to the desired ranges and then proceed with actual booking.<br>*This is a core function of the site.* |
| Look up existing bookings | Users can check whether bookings exist under their name (for example, if someone else booked a court for them).<br>*This is a core function of the site.* |

In a mobile site, you might want to maintain the two core functions, slightly reworked from a usability and presentation perspective.

The role of the logout function is application-specific in a mobile site. The browser (and the device) is much more personal; such devices are rarely shared with other people. At the same time, though,

a mobile device may be lost or stolen more easily than a laptop. Particularly for sites that manage sensitive information, the logout function is required; where the type of information managed is not critical, you can give the logout function a lower priority.

For the implementation of the Change-Password use-case, the user is required to enter both an old and a new password, perhaps (usually) entering the new password twice. That's too much work to do on a mobile device, where typing is the hardest activity, and the input scope of soft keyboards conflicts with the need to enter a strong password. To avoid changing or extending the logic of password management, such as by accepting a simple numeric personal identification number (PIN) from mobile users, you should drop this feature entirely from the mobile site. To change the password, users must log on to the full site to proceed.

Figure 3-3 shows a revised home page for the logged-on user of the mobile site.

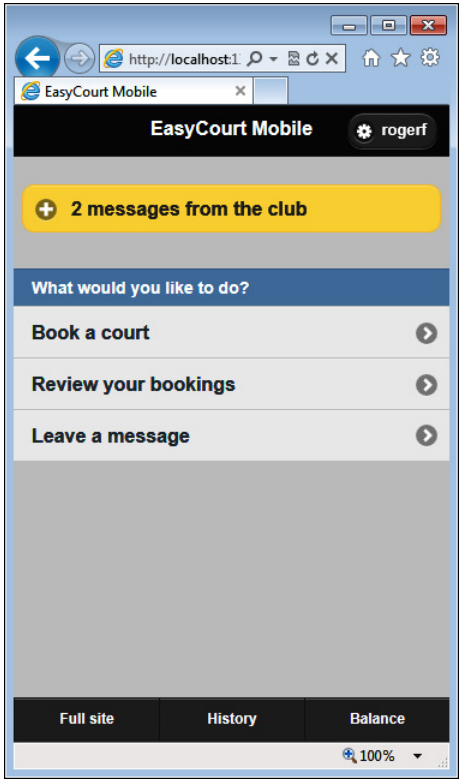



**FIGURE 3-3** A revised home page for a logged-on mobile user.

The logout feature is supported here: By adding an extra link button to the caption bar, you can keep the feature handy for users without sacrificing valuable screen real estate. The core functions take up one menu item each and give users immediate feedback about what they can do: book a court, review bookings, or leave a message.

The other links address features that were either absent from the main site or were presented differently here.

## Restructuring Existing Use-Cases

A mobile site doesn't have to be a clone of the full site, even when implementing the same general use-case. When designing the EasyCourt site, I focused on replicating the typical actions that a club member would take to book a court. Based on feedback that I gathered, it turned out that checking their balance (knowing how many hours you can book in a given period) was not on top of the list. So the site warns users if they're exceeding their balance when they try to enter a new booking, and they can check their balance every time they enter a new booking.

Personal messages, such as expiration of their medical certificates, are displayed when required, whereas general messages, such as a summer party being held at the club, appear only in specific public areas of the site.

In Figure 3-3, you saw a Balance link at the bottom of the page that points to a plain page like that shown in Figure 3-4.

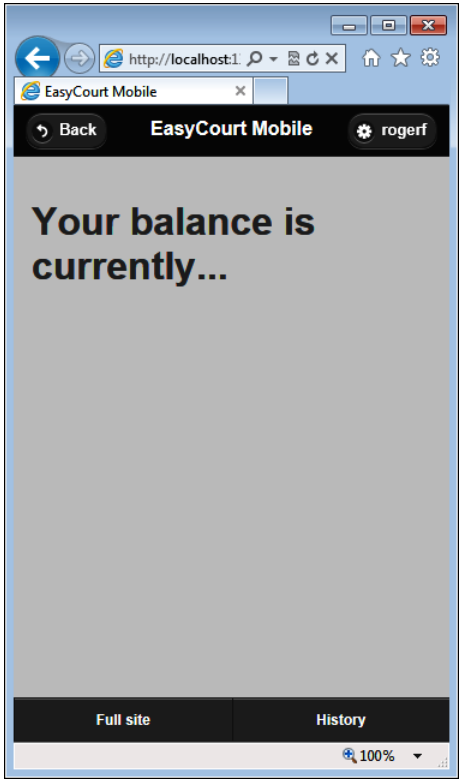



**FIGURE 3-4** Providing specific information quickly.

The benefit of this type of interaction is that you give users a quick way to find specific information with only a couple of touches. Users will hardly connect to the desktop site just to find their current balances, but that's exactly the type of quick information they may want to get from a mobile device—highlights of relevant facts and news rather than a full-blown page.

### Inventing New Use-Cases

Setting up a mobile site starting from an existing website also provides an opportunity to review the use-cases for the existing solution and may spark ideas about new features. The EasyCourt full site doesn't have a History page where users can review the list of bookings they placed and the other players they faced on the court. However, when I solicited feedback about which new possible features to include on a mobile site, History showed up prominently.

I wondered why History was considered (by nearly the same set of users) more important to have in a mobile site than in the full site.

The answer lay in the way that people tend to use mobile devices. For example, people often use them when they have nothing better to do (such as when they are hanging around in an airport, standing in line, or stuck in some waiting room). At such times, you can unleash your curiosity. You have time to revisit how often you play (and pay) at your club or how often you played your best friend.

In general, a mobile site (and to a good extent, a mobile native application) should be designed around the user much more than a regular site, even for nearly the same functions. Understanding the priorities of mobile users is key; that's why a code-first approach—or, worse yet, a code-rework approach—just won't pay off. Mobile solutions are all about getting up close and personal with users.

## Mobile-Specific Development Issues

Sometimes, when reworking a site for an ideal mobile experience, you face the need to have new queries and maybe new forms of updates incorporated into the application's back end. A quick comparison between the full site (Figure 3-1) and mobile site (Figure 3-3) should be enough for an experienced architect.

The features "Messages From The Club," "History," and "Balance" that you see in the mobile page have no direct counterparts in the full site. That means that you probably need different methods in the business and data access layers to provide the requested data. In all cases, this concerns data that is already in the database; you are just creating different types of queries and different view models to work with.

How would you tackle this need architecturally?

### Toward a Mobile Application Layer

Any software of any reasonable complexity is best organized in layers. Each layer represents one logical section of the system. A layer will be hosted on a physical tier (i.e., a server), and multiple layers may be hosted on the same tier.

Here, let's briefly recap the basics of layered applications and then look at how that design applies to mobile sites.

## Layered Applications

Most applications are built on three layers—presentation, business, and data access. Of the three layers, the business layer is by far the most blurred. The name *business layer* is quite generic, and the architectural details depend strictly on business cases. Today, the business layer often results from the aggregation of two interfacing layers—the domain layer and the application layer (see Figure 3-5).

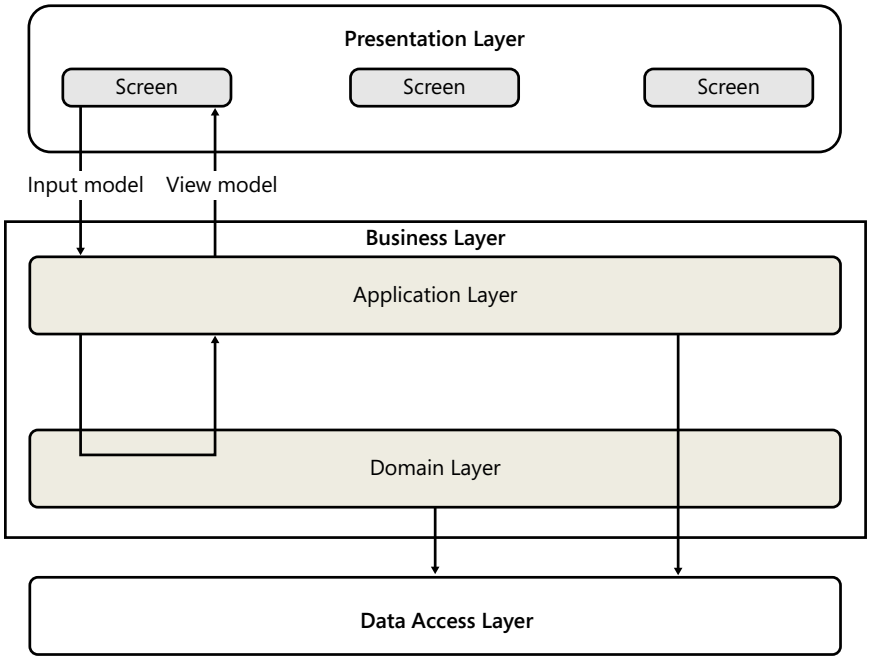



**FIGURE 3-5** A typical layered architecture of modern web applications.

The domain layer is where you describe the model of your domain data and where you orchestrate and expose domain-specific services. For example, the domain layer includes a domain model that you persist via an O/RM framework. Classes in the domain model have properties and methods (domain logic). A *Booking* class fits naturally in the domain model of an application like the sample EasyCourt site. Around this class, you likely need a number of services ranging from CRUD (create, read, update, and delete bookings) to more sophisticated queries such as all the bookings for a given member in a given period. These services apply to domain entities and use data access tools.

All this should be nothing new to most developers and architects.

In a realistically complex application, CRUD is never plain CRUD. Adding a booking, for example, is not simply a matter of writing one record to one database table. More often than not, the system response to the user demand of entering a booking is a workflow that must be orchestrated at some level. This is where the application layer (or service layer) fits in.

The application layer contains the application logic. It is that part of the business logic that contains endpoints, as required by use-cases. The application logic is the layer that you invoke directly from the presentation layer. The application layer coordinates calls to the domain model, workflows, services, and data access layer to orchestrate the behavior required by the various use-cases.

## Identifying Mobile-Specific Endpoints

As the name seems to suggest, *application logic* is application-specific; therefore, it is not a layer of code that you would expect to reuse much. A mobile site is a different application, so having a separate application layer makes total sense.

After identifying the use-cases for the mobile site, the next step is to check how much support the existing application layer can provide for the new site. A new or modified use-case likely will require new code to orchestrate the back end. A different user interface, on top of the same behavior, likely will require passing different data types around, and subsequently a modified application layer.

The domain layer—both model and services—is, instead, highly reusable because it represents the core logic behind the domain. From within the domain layer, inserting a booking requires exactly the same action, regardless of who orders it or whether they did so from the full or the mobile website.

A mobile site needs its own application layer, and probably some extensions to the domain layer for specific core operations not initially accounted for. In this example (see Figure 3-3), I had to extend the domain layer to support the history feature and to query for and push messages.

The key takeaway here is: Don't be afraid to reinvent use-cases, and make sure that you supply mobile devices with just the data they need—shaped appropriately. If this requires ad hoc new services, don't be afraid to write them as necessary.

## Defining an Application Layer for Mobile Clients

A mobile site is a website; as such, it may not need a web-based application programming interface (API). A web-based API is essentially a collection of Hypertext Transfer Protocol (HTTP) endpoints that connect the site to any middleware you may have. Such an API is a true necessity for native applications, but it is not necessarily required for a mobile site. A mobile site can simply be an ASP.NET website made up of .aspx pages or resulting from a bunch of controllers.

However, if you opted for a single-page interface and the site is built around a single template and extended and configured via Ajax calls, then you likely need a well-defined collection of endpoints for mobile view callbacks. Specific endpoints will be responsible for serving just the data that the mobile view expects. Figure 3-6 illustrates the big picture.

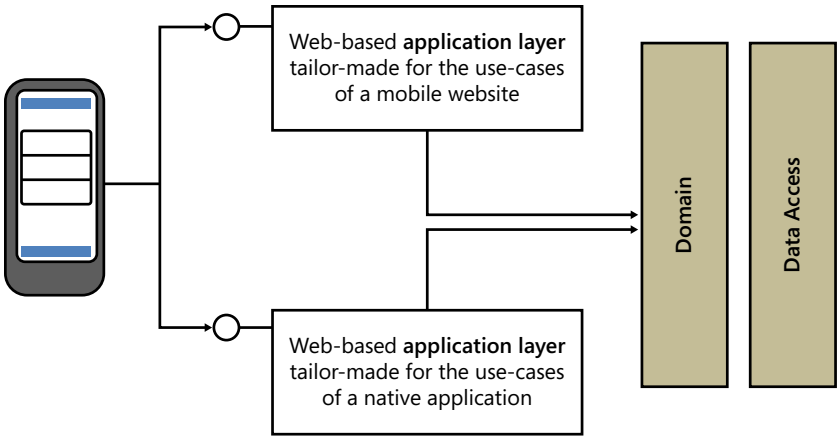



**FIGURE 3-6** An application layer for mobile clients.

To define a web-based application layer, you currently have a variety of options.

If your site is based on an ASP.NET Model-View-Controller (MVC), you can simply add a new controller and adjust routes to its action methods. Alternatively, you can host a classic Windows Communication Foundation (WCF) service on the server side configured to expose a web API. These two options require different programming models and slightly different APIs. As part of the Microsoft .NET Framework 4.5, you also can take advantage of the new ASP.NET web API, which combines the power of WCF and MVC controllers. The result is that you can have an API that works the same under both ASP.NET MVC and Web Forms and lets you create web endpoints with full support for ASP.NET MVC–specific features such as model binding, dependency injection, and routing, as well as WCF-specific features such as content negotiation and query composition via OData.

## Data Access Practices

Strictly related to the definition of a mobile web API is the definition of a format in which data should be exposed. Should that be JavaScript Object Notation (JSON)? Should it be a feed format such as AtomPub? Should you access data using a query/update protocol such as OData?

If you build your mobile site around a relatively fixed template that you flesh out with requested data, then JSON, returned by a plain or an OData service, is probably an excellent option. Consuming a plain JSON stream that models your domain objects doesn't require any external libraries in most mobile browsers; however, the same is not true for JSON if it is returned by an OData service, which probably requires a specific library.

If you set up a classic site that simply serves up HTML, then data access takes place on the server, and you can arrange to get the data through canonical tools, including plain ADO.NET commands or O/RM frameworks. When the mobile site is the first step of a larger project that will eventually include platform-specific applications, then setting up a layer of public endpoints that return JSON streams makes much more sense.

Should these endpoints be exposed through an OData service or a plain service that returns data in a more generic format such as JSON?

OData is a web protocol designed to enable performing CRUD operations on publicly exposed resources. An OData service is an HTTP-based data service that exposes data as name/value pairs taken from a variety of collections (e.g., a relational table). The stream of data can be formatted as an AtomPub or a JSON feed. In a way, OData is the web-based rework of the old idea behind Open Database Connectivity (ODBC) and Object Linking and Embedding Data Base (OLE DB)—a common API for accessing existing data in read/write mode.

Figure 3-7 provides an overview of the options that you have when you are creating an application layer that provides data to a mobile client.

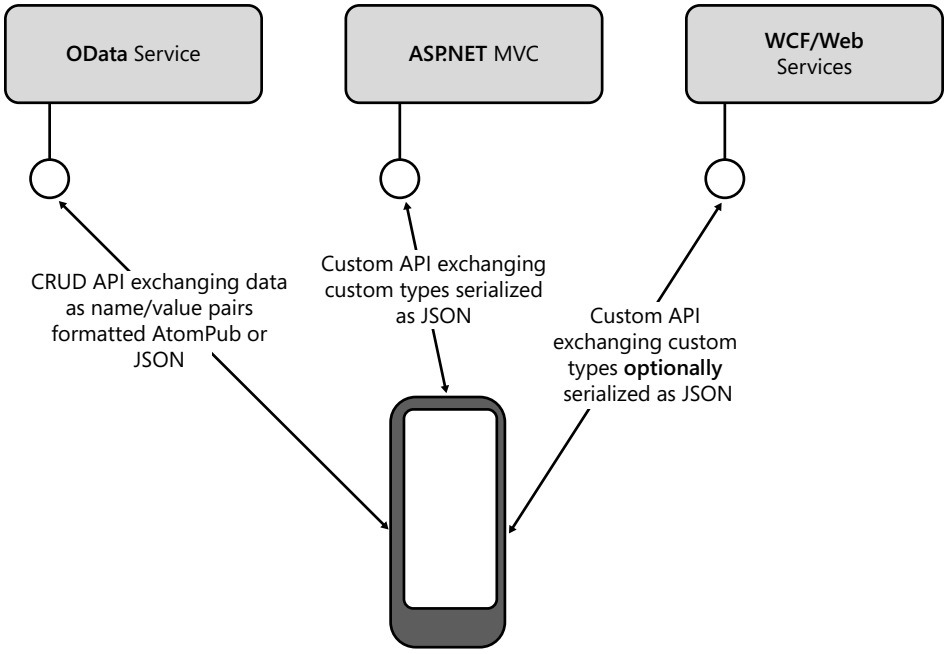



**FIGURE 3-7** Options for a mobile application layer.

The difference between an OData service and a plain web service is in the API that you invoke from the client. With OData, you have an API optimized for a CRUD scenario. You get raw data and adapt it to the format you need on the client. With other approaches, you have to put more effort into the design of the application layer but gain more freedom to return the data types that you prefer. You can design the flow of data to match the needs of the client perfectly. As usual, neither option is clearly preferable: you should choose the option that results in the best compromise between implementation costs, extensibility, and performance.

## Local Output Caching

A key factor for a successful mobile site is using any possible tricks to save requests. In classic ASP.NET programming, you can mark pages or sections of pages with the *@OutputCache* directive, meaning that a request for a page or view is resolved quickly at the server gate without physically running the request against the application's back end. The output cache is a great feature to have in those pages or views that are relatively static and change infrequently.

The output cache, however, still causes the client to place a request to the server. From a mobile perspective—where HTTP requests have a higher cost than on laptops—the classic output cache trick is much less compelling.

If the browser supports HTML5 local storage, you can try to implement your own local form of output caching. Support for HTML5 local storage is required; otherwise, you have no chance to cache data on the client beyond the small amount that you can store in a cookie (assuming that third-party cookies are enabled on the mobile browser). Most smartphones have HTML5-ready browsers that support local storage, so if those are your target, you can code application-specific caches of data and/or markup.

Figure 3-8 shows a view of the History page containing a list of the most recently updated bookings.

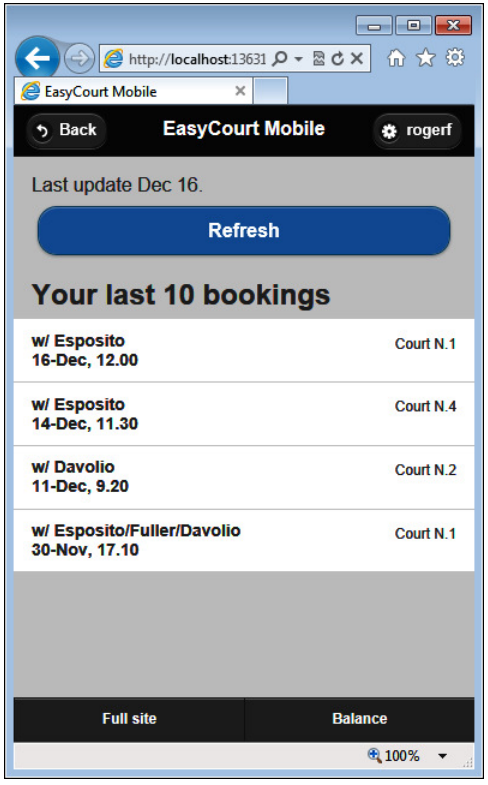



**FIGURE 3-8** A page that uses local caching.

A local output or data cache requires that your site makes intensive use of Ajax to download data. You create a page like the one in Figure 3-8 by placing a call to some remote service and then populating a list view with downloaded data. If local storage is supported, the downloaded data is then cached locally as a JSON string and reused for future requests. The duration of the cache is also totally under your control.

**Note** Chapter 5, "HTML5 and jQuery Mobile," will return to HTML5 and cover improvements in the markup language, as well as some key software development kits (SDKs) such as local storage that add new functionality to the browser environment.

## Server-Side Device Detection

Several thousand mobile browsers can connect to your site. Many of these have varying capabilities that may not match the site's markup and logic requirements. In general, anyone who is developing a mobile site that goes beyond displaying some basic HTML should be concerned about the capabilities of the requesting browser and should plan different versions for each page to accommodate the various clients. In general, almost no mobile sites are created for just one type of browser audience.

### The Rationale Behind Server-Side Device Detection

Mobile site development requires a couple of critical decisions: which devices you intend to serve and how best to serve them the site content. Each request received by the server contains a signature that identifies the requesting browser.

**Note** To be precise, this is not an absolute fact; there are some exceptions. For the purposes of this discussion, however, you can assume that every request contains information identifying the browser.

On the server, therefore, you can determine the requesting browser and match it to some *known* capabilities for that browser—for example, whether that browser supports JavaScript or local storage, or whether it can upload files. You also can discover the size of the screen—or simply determine whether it is a mobile device, a tablet, or perhaps a smart TV.

How would you arrive at the decision about which mobile browsers to support and how?

First and foremost, you never would base such a decision on specific browser versions, such as Windows Internet Explorer 9. Instead, you focus on browser capabilities. Unfortunately, though, very few of the capabilities we're interested in (including the basic capability of determining whether a device is a phone or a tablet) can be detected programmatically from the information sent by the browser alone. This fact created the need for *device description repositories* (DDRs), which are databases that store information about thousands of different devices and expose that information via an API.

Today, a DDR is nearly mandatory for any mobile website that needs to check more than one or two basic capabilities. As you'll see in the next chapter, you probably can arrange your own code to detect (quite reliably) whether a given device is mobile or not. You can hardly go beyond that point, though. Without a DDR, you have two options: serving the same view to any mobile device regardless of the effective capabilities, and relying on some CSS and JavaScript tricks to serve adjusted pages.

Both options have drawbacks. If you don't distinguish between devices, then you expose yourself to the risk of misbehavior and poor rendering on some devices, with a possible loss of business. If you opt for a client-only solution, then you have control over what's rendered, but you can't really serve optimized content to a specific mobile device. (For example, you can decide whether some content should be resized or hidden, but you can't manage to download just the content that fits the browser's capabilities.)

Now, let's briefly explore two different approaches for creating sites for a multiple audience: *multiserving* and *responsive design*.

**Note** Chapter 6, "Developing Responsive Mobile Sites," will discuss multiserving, responsive design, and DDRs in more detail. Specifically, that chapter focuses on Wireless Universal Resource FiLe (WURFL)—one of the most popular DDRs today.

## Multiserving

The idea behind multiserving is that for each mobile request, the site composes the resulting page by assembling views that are optimized for the profile to which that particular device belongs. As a developer, you build (or rent from the DDR of your choice) a mechanism that maps the user-agent string of the requesting browser to a device profile. Based on the profile name, this mechanism will return the actual name of the view (or page) to load to serve the request.

You typically create your own device profiles and customize them for the needs of your site and business. A common strategy consists of having three profiles: Smartphone, Medium-level phones, and Legacy phones. However, there's no limitation on the number of profiles and, more important, on the rules that restrict each profile. In general, you don't want to have more than four or five profiles because otherwise the workload becomes excessive.

The details of defining what devices fall in the various profiles are up to you. It is also up to you to define the capabilities that restrict devices to one profile or another. As an example, consider that the Medium-level profile might include devices that provide basic Extensible Hypertext Markup Language (XHTML) and CSS rendering, JavaScript, basic Document Object Model (DOM) manipulation, Ajax, and a screen that's less than 300 pixels in height. Similarly, the Smartphone profile includes devices that support HTML5, animation, local storage, geolocation, and CSS3. A tablet profile could be considered as powerful as a smartphone, but with a larger screen size.

Multiserving applies the principle of *Interface Segregation* to mobile pages. The Interface Segregation principle is about creating software modules (mobile views) that are simple, highly cohesive, and subsequently easy to edit and refactor. The principle fights scenarios in which you end up with fat modules that become unmanageable; it recommends that you split them into simpler blocks.

**Note** Project Liike is the product of the Microsoft's Patterns-and-Practices group, whose aim is to define best practices for mobile sites. I've been a proud member of their advisors board. The project takes the multiserving approach and identifies three device experiences along the previous guidelines. The approach is named W-W-W (for Wow, Works, Whoops) where the three words describe the reaction in front of a phone. Wow devices are Smartphones; Works devices are Medium-level devices; and finally, Whoops devices are Legacy devices. In Project Liike, the W-W-W approach is implemented using WURFL as the DDR. The definition of what makes a device Wow, Works, or Whoops is application-specific. For more information on the project, visit *http://liike.github.com*.

## Just One Web

The term *One Web* refers to a vision in which there's just one web, and sites should be designed to provide the same experience regardless of the device used to view them. Device detection may be necessary, and pages may require some reflowing, but site functionality, layout, and content should not be radically different.

As you can see, this approach—the same one we have used for years when designing desktop websites—entails a lot of work for mobile sites, much more work than was required for desktop websites. Why? A lot more special cases exist to take into account in mobile environments than for desktops.

How would you attempt to make the One Web vision reality? A few approaches exist, but not all of them may be considered pure from the original One Web perspective. In this book, I call a site *One Web compliant* if it has just one source file per page, yet still adapts to a variety of devices, as illustrated in Figure 3-9. In particular, Figure 3-9 compares One Web to multiserving: the difference lies in where you run the logic that produces the markup.

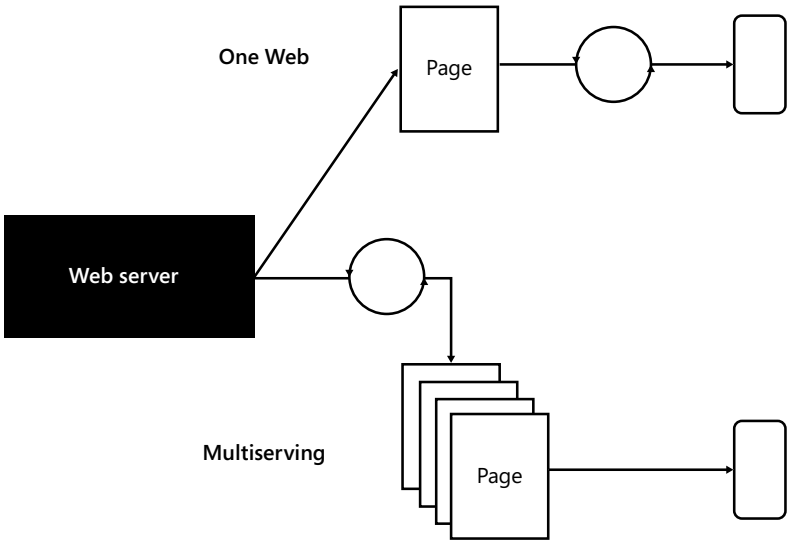



**FIGURE 3-9** One Web vs. multiserving.

In One Web, the logic that produces the markup is embedded in the served page. In multiserving, the logic is applied on the server instead, which then serves the final markup optimized for the device.

Figure 3-10 shows a possible flowchart for the logic that picks up the most optimized content for the requesting device in a multiserving scenario.

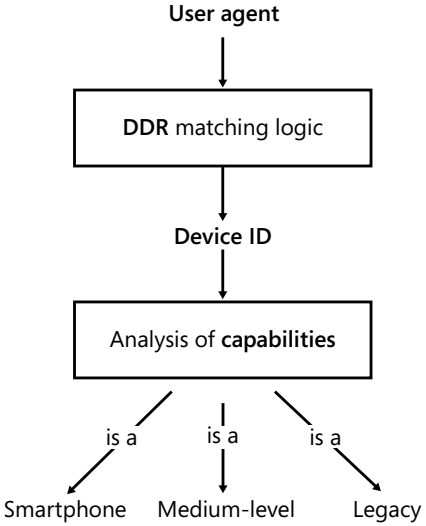



**FIGURE 3-10** Serving most optimized content for a device.

One way to implement *One Web* is by using CSS styles and media queries. Media queries help select a CSS style based on the capabilities associated with the user agent. Media queries provide for conditional download and application of a CSS style. Conditional download is based on the evaluation of a Boolean expression that involves only a limited set of properties, such as device width.

A CSS-based solution fits with the One Web philosophy because it doesn't change the functionality of the site when viewed on a mobile device; it simply limits or adjusts the layout as necessary. Media queries alone, however, can't work all the magic. A realistic One Web solution likely needs some JavaScript code.

An emerging trend is *Responsive Web Design*, which makes extensive use of CSS optimization and media queries to create dynamic layouts. Based on actual device width, pages resize their content dynamically—and do it without code, by simply relying on percentages. Such pages use these tricks to resize images, too. You can find a nice introduction to Responsive Web Design here: *http://bit.ly/blRWZc*.

As an alternative to CSS-only or CSS-based solutions, you can have code-only solutions for maintaining a single codebase. You can write that all by yourself, or you can use one of the available JavaScript frameworks.

For example, using the jQuery Mobile library shields you from a lot of trouble when it comes to dealing with various mobile browsers. The jQuery Mobile library offers a unified programming interface and abstracts away differences between browsers. However, you should note that

jQuery Mobile doesn't work in the same way on all possible browsers. Currently there are three levels of browser support in jQuery Mobile. The lowest level means that rendering will fall back to plain HTML and basic CSS. Chapter 5 will cover the topic of jQuery Mobile more fully.

## Summary

Overall, building a mobile site is not really a difficult task, at least compared to building a full desktop site. Building a mobile site, however, is a delicate task that has some key requirements. First and foremost, you should select carefully the use-cases you want to focus on. Second, you should re-architect carefully the application's back end to pass the mobile front end only the data it needs, and in the format that it needs.

That probably means that you need a different set of endpoints to call, or perhaps a reworked application layer, if your mobile site partners with an existing desktop site. But you should not need to make any changes to the domain layer or data access layer.

Beyond that, a mobile site is a plain website consumed by special browsers. To build it, you use the same set of tools and frameworks that you use for a regular site. ASP.NET Web Forms and ASP.NET MVC are both great options. The single-page application pattern is also an option.

When it comes to mobile, you should instead focus a lot of attention on the user interface to optimize collecting input and presenting data. A new family of components—and a new perspective—that use touch and take full advantage of the limited screen space are required. Mobile devices augment user capabilities. For example, they can access new types of information (such as geolocation) that may originate new use-cases and require new scenarios. The power of mobile sites lives here; planning a mobile site solely as an appendix of your full desktop site is a big mistake. Often, in fact, deriving the design of the mobile site from the full site is a mistake.

Finally, you should be ready to face the myriad implementations of mobile browsers out there. A responsive application—namely, an application that knows how to adjust itself to the requesting device—is the answer. To implement this, there are two main approaches: multiserving, which is based on server-side device detection and DDRs; and the One Web approach, which relies on device capabilities that can be tested on the client and uses CSS and JavaScript to adjust the markup.

The next chapter provides a few examples of mobile-optimized user interfaces and device detection, whereas Chapter 6 is fully dedicated to providing a developer's perspective of device detection and offers an up-close-and-personal perspective of WURFL, the popular DDR.

CHAPTER 4

# Building Mobile Websites

*Intelligence is the ability to adapt to change.*

*—Stephen Hawking*

**In this chapter:**

- From Web to Mobile
- Development Aspects of a Mobile Site
- The Device-Detector Site
- Summary

A mobile site, much like a native mobile application, is most likely to be consumed by users on the move: people who are standing up, in a hurry, busy, or waiting in line. Under these conditions, users are probably willing to connect to a site using a tiny device because they really believe that they derive some benefit from the site. The site, therefore, must be direct, concise, and accurate.

It is essential that a mobile site should load quickly and allow users to reach all main functionalities in just a few clicks or taps. The user interface should be extremely clear, but also clean and flawless to show the options available at any time, yet still making the act of choosing an option easy. After becoming familiar with the site, users often end up working with it semi-automatically, so even the position of a single button can have an impact on the quality of feedback that you receive. Note that the mobile Human-Computer Interaction (HCI) research field, although new, is very active, and a lot of studies exist about the dos and don'ts of interaction between mobile users and mobile devices and software. Luca Chittaro has a paper that effectively summarizes what mobile HCI means to developers and architects. You can read it here: *http://goo.gl/lSG3s*.

The previous chapter emphasized the importance of accurately selecting the use-cases to implement. The *number* of use-cases, however, should be kept small so that the site doesn't end up as a shrink-wrapped version of the full site. A pragmatic (and then not necessarily exact) rule is that a mobile site rarely needs more than 20 percent of the features available in the full site. I'd even go further by saying that sometimes, not even the 20 percent of features you take from a parent website should not necessarily be reimplemented "as is." You might want to restructure some of these use-cases and even add new ad hoc use-cases for related scenarios.

A mobile site is a brand-new project that can inherit some code and services from an existing site—more often than not, it inherits large shares of the business logic. This chapter covers a number

of issues and open points that you will want to solve before embarking on building a mobile site. After addressing all these points, building the mobile site is reduced to the work of writing a relatively simple and small website.

# From Web to Mobile

You rarely build a mobile site without also having a full site in place. Most of the time, you build a mobile site to serve mobile users better. In some cases, you start building both full and mobile sites at the same time. Currently, it's quite unlikely that you build a mobile site as a stand-alone project. Whatever the case may be, however, this section aims at isolating the issues that differentiate a mobile site from a full website.

If you're building a mobile site as a spin-off of an existing website, then the chances are good that you will be able to reuse large portions of the existing application's back-end code. Those include the data access layer, the domain layer, and possibly a bunch of other distinct components (such as services and workflows) that you may already have in place, which will provide bits and pieces of business logic. If you can replicate some views without too much modification, you may even end up being able to reuse webpages and ASP.NET Model-View-Controller (MVC) controllers—more generally, parts of your presentation and application layers.

As you move towards the presentation layer, though, the chances of reuse diminish. How would you deal with scripts, images, style sheets, and markup? For images and style sheets, there's probably an easy answer—you just have to reduce their size. For scripts and markup, the answer is less obvious and likely is influenced by context.

## Application Structure

Before the community of web developers (re)discovered AJAX a few years ago, websites were always built as a navigable collection of distinct pages. Jumping from one page to the next required links and forms. The browser handled each request, which resulted in a full replacement of the current page.

AJAX changed the course of things by using the services of a small browser-hosted component—the *XmlHttpRequest* (XHR) object—through which your script code can play the role of the browser and conduct server requests autonomously. The net effect is that webpages can contain ad hoc script code that use XHR to download data or partial pages. After being downloaded, the content is processed by some other script code and used to update the currently displayed page. AJAX can be used for some specific (perhaps even critical) features, or it can be used extensively throughout the site. When that happens, the site architecture is usually referred to as the *Single-Page Interface* (SPI) model.

### The Single-Page Interface Model

In brief, the SPI model refers to a web application that behaves more or less like a desktop application—it has a primary user interface (UI) layout that is adjusted and reconfigured via AJAX calls. The page downloads everything it needs from the source site via script-controlled actions.

The SPI model has been recently formulated as a manifesto. You can read about it here: *http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php*.

As an early AJAX adopter myself, I've always been a fan of the SPI model, but I've also always seen it as a vector rather than a concrete pattern to implement. As a result, I have not fully implemented the SPI model in a production site yet, primarily due to lack of confidence, proper facilities, and tools. Moreover, I always found it difficult to sell a 100-percent JavaScript site to a customer—at least, that was true up until two or three years ago. Today, the situation is different. The SPI manifesto dates back to the summer of 2011.

With that said, I'm not completely sure that a SPI model is appropriate for just any mobile site. An SPI model requires a lot of JavaScript, partly written by you and for the most part imported from external libraries. You likely need quite a few of these libraries to provide for generic UI manipulation, templates, and data binding. Some of these libraries are based on jQuery and jQuery Mobile. These two libraries alone total some 200 KB (uncompressed) of script and style sheets.

**Important** In a production site, you typically apply minification and GZIP compression to script and other resources, thus reducing significantly the size of the download. *Minification* is the process of removing unnecessary characters from source code without breaking any functionality. Applied to script files, minification also adds a (thin) layer of obfuscation to your code, making it much harder for humans to read. GZIP is, instead, a popular compression format. Once properly minified and gzipped for a production site, the jQuery library is 31 KB and a bit less for jQuery Mobile.

In addition to script, SPI requires helpers for data binding and UI refresh, which adds a few more tens of kilobytes. Popular libraries in this segment include JsRender, JsViews, Knockout, and Upshot.

In summary, the size of the JavaScript assets that a client will need to download can consume a few hundred kilobytes, which can become a problem. Once downloaded, of course, the browsers will cache the scripts; they don't download the code over and over again. But the browser needs to do a lot of work to render SPI pages—work that goes far beyond simply requesting and rendering markups. The more advanced the device browser is, the more the SPI model becomes affordable (from a resource standpoint) for mobile sites.

Personally, I wouldn't adopt the SPI model without first performing deep analysis of the context. The main challenges with an SPI implementation can be summarized as follows:

- In case of intermittent connectivity, it's difficult to figure out whether the problem is with the application or the network. This may be frustrating. Not that this same problem doesn't exist for other models (i.e., the full-page refresh model), but at least tiny, non-SPI pages download well, even with limited bandwidth. Moreover, with no connectivity at all, you immediately grasp the nature of the problem—when nothing shows up, the problem is not the application.
- Many sites require users to log on before they will serve content. If the users' session expires, the full-page refresh model redirects them to the logon page at the first successive access. The

problem is both manifest and easily resolved. With an SPI site, although user authentication is also AJAX-based, it may take hours before you figure out the root cause of the misbehavior you're observing. In SPI, user authentication requires due attention and effective client and server-side implementation to work smoothly.

So let's explore some other available options.

**Note** ASP.NET MVC 4 ships with a new project template that promotes the use of the SPI model. The template uses Knockout and Upshot.

## Full Page Refresh

At the extreme opposite of SPI lies the classic *Full Page Refresh* (FPR) model. FPR is how the web worked for years: the browser makes a request for a new page, the web server returns the new page, and the browser replaces the current page with the new rendered content. This process occurs repeatedly for each user action.

AJAX (on which SPI is heavily based) made the classic FPR web experience much less compelling, and it also made it more natural for users to expect that only fragments of the current page would be updated as the result of a given action. In a desktop scenario, the FPR model is cumbersome, so more and more large sites are progressively adding AJAX-based capabilities.

However, the significant impact that FPR has had on desktop sites (which have large displays and numerous auxiliary resources for downloading, caching, and refreshing content), is less so on mobile sites because mobile pages are considerably smaller and lighter to begin with. Still, loading several small pages may still be less engaging than updating bits and pieces of the currently displayed page. Updating the current page may still be slow over a slow connection, but at least it doesn't have a dramatic impact on the user experience.

## Partial Page Refresh

Some middle ground between SPI and FPR can be found in the partial rendering that both ASP.NET Web Forms and ASP.NET MVC support. In terms of traffic, partial page refresh (PPR) falls between the two extremes—it is not as efficient as SPI, but it's not as poor a user experience as FPR. The idea is that the browser places a request as usual, but that request is captured at the script level—via embedded script—and silently transformed into an AJAX request.

On the server side, the web server handles requests as if they were regular full page requests, but the response is packaged as an HTML fragment. The data being transferred is a mix of data and markup—just the delta between the markup of the current page and the requested new page.

The benefit of the PPR model is that it offers many of the benefits of AJAX without the cost of having to learn new programming features. In ASP.NET Web Forms, PPR happens relatively automatically through the *UpdatePanel* control; in ASP.NET MVC, it occurs through the *Ajax.BeginForm* HTML helper.

In a nutshell, using PPR means that an FPR approach won't require special skills on the development side. In contrast, an SPI model may represent a complete paradigm shift for many developers. PPR represents some middle ground.

### Context Is King

Which of the previous options is preferable? There's no obvious and clear answer. If you're determined to find just one answer, then the best approach depends strictly on the context and the knowledge that you have about the devices that are going to access your site. If smartphone traffic is predominant, you definitely can opt for SPI. But if you're interested in reaching the widest possible audience, then you probably should opt for FPR.

The point, though, is that there's probably no ideal solution that works in all cases. Mobile devices are so different that you really should consider partitioning your audience into a few distinct classes of devices and arrange a different solution for each. That could mean serving plain HTML pages to older browsers while implementing a nicer SPI solution for smartphones.

**Note** In general, you always should consider AJAX seriously because it reduces the amount of network traffic. Extensive use of AJAX, however, actually may *raise* the number of Hypertext Transfer Protocol (HTTP) requests. In a mobile scenario, an HTTP request is more expensive than in a desktop scenario because connections are slower, limited in number (reduced parallelism in download), and also sometimes processed in a more convoluted way, especially if the connection doesn't happen over a WiFi network.

## Amount of JavaScript

The amount of JavaScript that you might want to use for the pages of your mobile site is another huge point. Processing JavaScript is crucial to minimize web traffic; on the other hand, it also affects performance. Like it or not, mobile devices (even high-end smartphones) are not as powerful as laptops. Among other things, this means that a mobile device may not be able to tolerate effectively the same amount of JavaScript that you could employ in a full-fledged website. In this context, "not able to tolerate" just means consuming more battery power even if the perceived page performance is acceptable.

**More Info** The consumption of battery power tends to increase with the number of HTTP requests. Subsequently, a JavaScript-intensive page is critical from the resource management perspective. Here's a reference and some numbers: *http://goo.gl/EKcyj*. Other excellent resources for making sense of the amount of JavaScript and its performance are Steve Souders's blog (*http://stevesouders.com*) and *http://goo.gl/jyhV*.

## The jQuery Family of Libraries

Today, the jQuery library is a must for nearly any website. I'm personally dreaming of the day when jQuery will be native to browsers and be integrated into every browser's JavaScript engine. Until that day comes (if ever), jQuery must be linked and downloaded if you plan to use it.

Note that jQuery is required even if you plan to use only jQuery Mobile, which offers a variety of ready-made components for scaffolding a mobile site. Plug-ins for jQuery Mobile can put a "skin" on your mobile site so that it looks like a native application with animations, navigation effects, and snazzy presentation.

All in all, once minified and gzipped, all this JavaScript is likely affordable for use with high-end mobile browsers, even though JavaScript-based effects may emulate some native effects closely but are never the same in terms of performance. With that said, you'll need to be able to limit the quantity of JavaScript in your pages if you want to enlarge your mobile horizons to non-smartphone devices.

So except for smartphones, consider dropping jQuery and derived libraries entirely. On these non–high-end devices, the chances of encountering quirks, bugs, unsupported features, and unexpected behavior is quite high; therefore, why take the risk? By simplifying your page structure, you still should be able to include some JavaScript-based dynamic behavior that relies only on Document Object Model (DOM) support and basic language tools. A good rule of thumb is that (with the notable exception of smartphones and tablets) any quantity of JavaScript beyond 10 KB can begin to degrade load time and performance.

**Note** Whether you use jQuery or not, *unobtrusive* JavaScript always should be your guiding star. Unobtrusive JavaScript means having a place at the start of the code where you attach handlers to elements so that degradation in the case of unsupported features is easier to handle.

## JavaScript Microframeworks

For a desktop site, nobody really minds having a few hundreds of kilobytes in script code. Gmail, for example, fully loaded, usually exceeds the range of kilobytes for loaded JavaScript code. The idea of loading a full-fledged, monolithic JavaScript framework in a mobile site is often unaffordable, especially if you want to target more than just the top iPhone and top Android devices. Still, it's useful to be able to use the services of existing code and ready-made frameworks. JavaScript microframeworks come to the rescue.

Microframeworks are small, highly focused libraries whose overall size is often only a few kilobytes; sometimes even 1 KB or so. There's no magic and no special tricks—microframeworks are so small not only because they are minified and gzipped, but also because they take on just one or two tasks.

You probably will want to pick up a library for asynchronous (async) loading: one for optimized DOM traversing, one for touch gestures, and perhaps a few more. You can find an interesting list of such microframeworks at *http://microjs.com;* their average size is around 1 KB.

One microframework for general-purpose JavaScript programming in the context of mobile sites is XUI (see *http://xuijs.com*). XUI is not specific to a single task as are some of the libraries listed on *microjs.com*; instead, it's specifically designed for mobile scenarios, so you can consider it a competitor to other larger mobile frameworks such as jQuery Mobile and Sencha Touch. Unlike these larger frameworks, XUI doesn't force you into a given programming paradigm or page structure. It focuses on a few tasks (e.g., DOM/CSS management) and totals only 5 KB gzipped. XUI is a good alternative to jQuery libraries for mobile sites and offers a powerful alternative to jQuery libraries for lower-end devices that support JavaScript, DOM, and Cascading Style Sheets (CSS).

Chapter 5, "HTML5 and jQuery Mobile," will cover the whole topic of jQuery Mobile. Sencha Touch is a JavaScript framework, originally created to add touch capabilities to mobile pages, but which is now a full-fledged framework for developing mobile web applications that look and feel native on most mobile platforms such as iOS and Android. You can find out more about SenchaTouch at *http://www.sencha.com*.

**More Info** A good resource for microframeworks, and for comparing their pros and cons against larger script frameworks such as jQuery, is Addy Osmani's work at *http://goo.gl/jnE9t*.

## Application Device Profiles

Device fragmentation is huge in the mobile space. If the differences between browsers in desktop site development scare you, then be aware that the mobile space is much worse.

On rare occasions, you can get by with just one set of pages for a mobile site. Ideally, you need pages that can adjust intelligently to the characteristics of the requesting browsers. Sometimes you just end up having multiple versions of the same page—one for each device (or class of device) that you intend to support. Alternatively, sometimes you may have one common page template that you fill up with device-specific content. But at the end of the day, these are implementation details.

The crucial point is that you need to split your expected mobile audience into a few classes. Then, for each page or view, you provide class-specific markup. This is the essence of *multiserving*, as briefly described in Chapter 3, "Mobile Architecture." Chapter 6, "Developing Responsive Mobile Sites," will illustrate multiserving with an example.

**Note** In most developed markets today, it is possible to cover most devices with good semantic markup of some kind and then progressively enhance the presentation and interaction using CSS and JavaScript. In this regard, the concept of device classes applies to CSS as well: however, if you want to present different content to different classes of devices, then you need to use multiserving because the markup will change between classes.

## Practical Rules to Class Profiles

In the mobile space, neither the team building a given site nor the team producing a general-purpose library can afford optimizing pages on a per-device basis—the number of potential devices to take into account is just too large (in the order of several thousand). Hence, a common practice has become to classify all the devices you're interested in into a few classes. How many classes do you need to have, and how do you define a class?

A device class typically gathers all devices that match a given set of capabilities. You define the classes based on the business cases and scenarios. A basic (but still arbitrary) classification may consist in splitting the whole range of requesting devices into three categories: smartphones, tablets, and all other browsers. You provide a rich web experience to smartphones, serve the full site to tablets, and offer plain HTML to all others.

How would you define a smartphone? Everyone would agree that an iPhone is a smartphone while, say, a Nokia 7110 is not. [The Nokia 7110, released in the fall of 1999, was the first device equipped with a Wireless Access Protocol (WAP) browser.] In contrast, the Nokia 6600 certainly was a smartphone when it came out in 2004, but nobody would consider it a high-end phone today. But there are no hard and fast rules. You should know that not only the device classes, but also the rules that determine which class a given device belongs to, are highly variable and strictly specific to a business scenario. At the same time, this lack of fixed rules and practices makes it possible for everybody to define classes in a way that best fits a given business. All you need is a reliable infrastructure that first identifies the device and then tells you about its real capabilities. I'll return to that infrastructure in a moment.

As a purely intellectual exercise, here are some requirements that identify a modern smartphone in 2012. Note that these are likely to change, even in the near future.

- *Operating system (minimum version)*: all versions of iOS, Android 2.2, Windows Phone 7, RIM OS 6, Samsung Bada 2.0, Symbian Anna, and Nokia Meego
- *Input mode*: touchscreen
- *Screen size*: 320 × 480 pixels

Admittedly, this definition is rather arbitrary, and it may sound too restrictive for some while being way too relaxed for others. Above all, this definition will become progressively less pertinent as more powerful devices hit the market.

A tablet has two main characteristics—it is a mobile device (not a desktop browser), and it has a larger screen than a smartphone. You can probably set the lower boundary to 640 pixels today.

It is important to note that grouping all the remaining devices into a single class may be a tough decision. Whether you really want to serve plain static HTML to all of them or further split the remaining devices into two or more classes is up to you.

How can you discover the capabilities of the browser for each device? How can you be sure that the requesting browser is really a mobile device? If it's sufficient for your needs to just check the screen size and screen resolution, then you might decide to go with CSS media queries for high-end

browsers, and use script code that simulates that on older browsers. Although this approach may work in some cases, it's not a route that will take you far. Instead, relying on a commercial device description repository (DDR) is probably the best way to go. Chapter 6 discusses a few DDR frameworks, focusing in particular on WURFL.

### Dealing with Older Browsers

When I talk to executives planning the mobile strategy of their companies, I often get the impression that when they say "mobile," they just mean the iPhone and iPad. While it can't be denied that iPhones and other smartphones are actually responsible for most of the mobile traffic to sites, the mobile universe contains many other types of cell phones and devices as well. Unfortunately, not all of them have the same characteristics, but they are so numerous that you must find a common denominator approach.

In the process that you use to identify the device profiles to support, you must define the bottom of the stack at some point. Devices that fall into this sort of catchall group typically are served plain HTML or, at least, the simplest markup that your mobile site can serve.

What's the best way to handle this?

When you end up having a catchall device profile, it also means that there's some rich library you're relying on for higher-end devices. The jQuery Mobile library is an excellent example. Such mobile libraries sometimes offer to scale down the otherwise rich markup they produce on older browsers automatically.

That's apparently a fantastic deal for you: you write the mobile markup once, and it gets downgraded automatically for less powerful browsers. Unfortunately, my current experience has not been particularly positive on this point. Although most libraries do fulfill their promises of downgrading the markup, the quality of the HTML that they produce when that happens is often below your desired standards. You probably want to take care of the HTML being served to older devices yourself rather than blindly relying on the kind of hard-coded markup served by some libraries.

The bottom line is that while jQuery Mobile (and other libraries) can truly downgrade HTML based on the requesting browsers, you'll achieve a better final effect if you manually fix up the output. Currently, I'm inclined to use jQuery Mobile—but only to serve smartphones and tablets.

**Note** Chapter 5 will cover jQuery Mobile in more detail, including more about its browser-graded support matrix. That feature is orthogonal to performing your own device capability detection, but overall, I prefer to skip the automatic downgrading, at least with the current version of most libraries.

## Optimizing the Payload

Minimizing the number of HTTP requests to websites is always a good thing, and it should be a central aspect of any strategy aimed at improving the performance of a site.

If you have ever tried to use a mobile device to connect to a very basic site with a few plain HTML pages, a bit of CSS, and one or two images over a 3G data connection, you have experienced a delay, or *latency*. This latency is relevant if the device is not one of the latest smartphones with a powerful processor. In the mobile space, minimizing the total amount of data transferred and the number of requests is not simply a matter of optimization; it is a crucial development point.

Over the years, a number of recommended practices have been worked out to help developers build fast websites. Yahoo! has been quite active in this field, publishing a very valuable document that you can read here: *http://developer.yahoo.com/performance/rules.html*. The rules in that document are written for a generic website, but for the most part, they can be applied equally well to both desktop and mobile sites. Here's a summary of the key suggestions:

- Take care of the page structure and find the right place for scripts and style sheets.
- Reduce the number of HTTP requests.
- Reduce the size of resources.
- Maximize the use of the browser cache.

Let's briefly go through the optimization aspects that are most relevant to mobile sites next.

## The Page Structure

A mobile browser may load pages significantly slower than a laptop browser. This means that not just raw performance but also *perceived* performance is important. Little tricks, such as placing style sheets at the top of the page in the <*head*> section help, because doing that means that the body of the page will be ready to render as it is downloaded. The overall download time doesn't change, but at least users see some results a bit sooner.

Similarly, placing scripts at the bottom of the page is helpful because it reduces the impact that synchronously downloading scripts may have on page rendering. When browsers encounter a <*script*> tag, they stop page rendering and proceed to download the script synchronously. Page rendering resumes only after the script files have been downloaded, parsed, and executed. When all the scripts are at the bottom of the page, browsers don't need to interrupt the page rendering process to load scripts; therefore, browsers are free to concentrate on displaying an early view of the page.

## Reduce the Number of Requests

Too many HTTP requests are the primary cause of latency in websites. This statement is even truer for mobile sites. Executing an HTTP request is an expensive operation, especially when that entails connecting to a radio cell. In this case, to preserve battery power, the device sometimes cuts off the connection right after receiving the HTTP response, meaning that to execute another request, a new handshake is required, which consumes both time and resources.

For this reason, it is doubly sinful to let links to duplicate resources go unnoticed. Linking a script twice doesn't affect the rendering of the page, but while the performance hit is negligible on a laptop, it becomes a serious performance hit in a mobile scenario.

The same can be said for redirects. Each redirect requires two HTTP requests. Avoiding redirects from a site is one way to reduce the number of requests that devices visiting that site have to place. Most ASP.NET MVC books [including my book *Programming ASP.NET MVC*, 2nd ed. (Microsoft Press, 2011)] recommend that the Post-Redirect-Get pattern is appropriate for input forms because it saves applications from unwanted F5 refreshes. In a mobile space, that also introduces extra requests; make sure that you make a sound decision about your projects on this point. If you can afford to use AJAX, that would probably be an ideal compromise.

## Compacting Resources

When your goal is to reduce the number of HTTP requests, there's little better than merging two or more files. Image sprites, for example, illustrate just this point. A *sprite* is an image that results from the concatenation of two or more images. That way, the browser can make a single request, downloading and caching multiple images at one time.

Image sprites are not always ideal. Using sprites works great with very small images, such as button icons that are widely used across the pages, but sprites may not be as ideal for the relatively large images used by distinct pages. Downloading a 50 KB image may not be easy over a 3G connection and with an older browser, so if the image is part of a sprite and the sprite size is 100 KB or more, downloading it would take even more resources—probably enough to make the user experience unpalatable.

Sometimes, to save an HTTP request, you can decide to encode a small image (one on the order of just a few kilobytes) as a Base64 string and embed it directly in the page. The data Uniform Resource Identifier (URI) scheme just serves this purpose.

The data URI scheme defines a standard for embedding data within the page instead of linking it as an external resource. The scheme is defined in RFC 2397; you can read about this RFC on Wikipedia at *http://en.wikipedia.org/wiki/Request_for_Comments*. The net effect of applying the data URI scheme is that the content of the *src* attribute of the *<img>* element matches the following template:

```
data:[<content-type>][;base64],<bytes of the image>
```

First, you place the data keyword followed by the Multipurpose Internet Mail Extensions (MIME) type of the image. Next, you place the Base64 keyword followed by the Base64-encoded representation of the binary image. If you were embedding an image manually, here's the markup you would need:

```
<img class="image" src="..." alt="" />
```

The Base64 image encoding (also known as *image inlining*) saves a few HTTP requests and improves both the *real* performance and *perceived* performance of the page. It is not a feature to use for just any image and page!

**Note** Most modern device browsers support image inlining, with the sole notable exception of the Windows Internet Explorer browser embedded in the versions of Windows Phone prior to version 7.5. However, few older browsers support this feature, whose importance decreases as the capabilities and processing power of the browser increase.

## Improve Your Control over the Browser Cache

If the primary objective of a mobile site is to minimize the number of requests, browser caching is the primary tool that you have to manage. Moving auxiliary resources (i.e., style sheets and scripts) to external files often helps. Initially, the number of HTTP requests is higher, but after the first access auxiliary resources are cached, no more requested expiration occurs.

External resources are really beneficial if such resources are widely referenced from a variety of pages. In home pages and rarely visited pages, inlining of resources (where possible) may be a better option. In addition, you can drive the browser behavior about caching by using e-tags and *Expires* headers on your critical resources.

**Note** In addition to optimizing the cache and minimizing HTTP requests, you might want to employ all possible techniques that reduce the amount of data to download. This certainly includes fixes to application logic to return the smallest possible amount of data and markup, but it also includes actions on the infrastructure, such as enabling compression at the web server level and minifying scripts and style sheets. Finally, note that you should always return data as JavaScript Object Notation (JSON) strings rather than XML strings. (JSON was once named the "fat-free alternative" to XML.)

The browser cache also applies to AJAX responses. The use of AJAX makes the request go unobtrusively for the user, who remains in total control of a still responsive page. However, it doesn't mean that the request will end in a matter of milliseconds. Sometimes caching the AJAX response helps to make most responses really instantaneous. This pattern, however, doesn't work for all types of requests. If you place a request to read the current balance of an account, you don't want it to be cached. If you use an AJAX request to auto-complete a text field, you want to cache as much as possible. This is to say that control over the AJAX cache must exist, and in real-world scenarios, it must be applied on a per-URL basis. Most libraries, though, offer an all-or-nothing control, which is hardly the right thing for a mobile site.

**Important** As the Back button is the most popular button in a browser, you need to remember this point: you don't want the page to reload when you hit that button. To avoid reloading, it is not always enough to set the right cache headers. The size of the elements also matters. The following link provides some numbers. It is a post from a couple years ago, but it is still worth reading: *http://www.stevesouders.com/blog/2010/07/12/mobile-cache-file-sizes*.

## The Offline Scenario

A mobile site represents an excellent shortcut to implementing a mobile strategy because it brings products and content to a variety of devices without writing a different application for each mobile platform that you intend to support. At the same time, a mobile site requires constant connectivity to work well. This aspect of mobile sites is going to be more and more of a showstopper for sites, and it highlights the key difference between mobile sites and native applications. It's not coincidental that offline applications have been given a role in using HTML5.

### Offline Sites with HTML5

An offline experience without an HTML5-enabled browser is quite hard to achieve; while it's not impossible, it does require a strong commitment. In other words, it's not a feature that you would want to offer as a free add-on to a customer!

When most people talk about offline sites, they mostly mean online sites that perhaps occasionally experience long downtime periods. The key to surviving such lack of connectivity is caching—in particular, to take advantage of the browser's ability to cache resources that a user may navigate to later, during a downtime window. These resources include not only auxiliary files, but also AJAX responses.

HTML5 lets you create a manifest file and link to it from the *<html>* tag of the home page. In this manifest code, you list the files you want to keep cached, which resources are a fallback for other (possibly missing) resources, and which resources are available only while online:

```
<!DOCTYPE html>
<html manifest="/offline.appcache">
  ...
</html>
```

The browser's ability to cache a subset of the full site on the client isn't a great thing, per se. It requires more than that to be truly effective. That means is that there is little value for users in just navigating through a few static pages. While that's considerably better than getting a 404 error message from the browser, it's not really a decisive change.

### Persisting Application Data

Persisting application data locally is another aspect of websites strictly related to surviving an offline status interval. In an HTML5-enabled browser (again, most browsers in today's smartphones are HTML5-ready), you use the local storage application programming interface (API) to write name/value pairs in an application-restricted area managed by the local browser. In the near future, the flat name/value format may become even more sophisticated and evolve into a table-based and indexed format.

Persistence is also related to synchronization. The ability to persist data locally fully enables occasionally connected application scenarios. At the same time, in an occasionally connected scenario, you might want to offer a read-only view of the data or enable updates. When this happens, you have either the problem of queuing operations or the problem of editing a local cache of data to be synced with the server when the connection is re-established.

Overall, if you want to build a full-fledged, occasionally connected scenario, you'd better endow yourself with a solid sync framework and/or consider using the OData protocol and facilities for your data exchange.

# Development Aspects of a Mobile Site

Based on the discussion in Chapter 3, the most important task when planning a mobile site planning is selecting use-cases. This doesn't mean, however, that use-case selection is unimportant when developing full sites or other types of applications. It's just that a mobile application and site are structurally built around a few (and well-chosen) use-cases.

Even when you're simply picking up a use-case from the root site, the way in which you implement it for a mobile audience may require significant changes—possibly a different user interface and perhaps even a different workflow.

At this point, let's assume that you have a well-defined (and hopefully well-chosen) set of use-cases; and let's also suppose that you have everything required for the back end already in place. You are now ready to start producing markup. But, first and foremost, how do you reach the mobile site?

## Reaching the Mobile Site

A mobile site can be a stand-alone site located at its own unique URL, or it can result from the application logic serving appropriate content to desktop and mobile browsers. In the former case, you just create a new ASP.NET project, design your pages for the mobile devices that you intend to support, give your images and style sheet the size and properties they need, and go. In the latter case, you have a single project where you just handle the desktop full-web case as a special case of a mobile site. Let's investigate the two options.

### One Site, One Experience

Maybe partly influenced by the One Web vision, I initially approached mobile development with the idea of offering my users just one endpoint and host name. The plan was to hard-code the server with the ability to detect device capabilities and serve the most appropriate content. So my first mobile project was a mere extension of an existing site: I just released a new version of the desktop site with the additional ability to detect mobile browsers and serve ad hoc markup. I had just one ASP.NET project with two distinct sets of pages/views for desktop and mobile.

Honestly, this didn't take much effort to design and implement. It took only a little extra engineering to set up a page/view router to distinguish between and serve both mobile and desktop requests. However, testing the site was painful. The only reliable source of information was to use a real mobile device; switching the user agent string on desktop browsers just didn't work effectively. We'll return to these test issues later in this chapter, and also demonstrate this one-site, one-experience approach while presenting a demonstration of device capability detection.

The noteworthy point is that when you have just one site that can handle both mobile and desktop browsers, you actually have one set of pages for full browsers and then multiple sets of pages for each class of mobile devices that you support. Really, the desktop becomes the special case!

What about different use-cases? This isn't a big issue. You always have a home page in both mobile and desktop environments, so your mobile home page will just offer a different set of links and start a different type of navigation. You only need some logic that, when a new browser session starts, the home page request for *http://www.yoursite.com* produces the output of *default.aspx*, *default.mobile.aspx,* or perhaps *default.iphone.aspx*.

## Two Sites, One Experience

Mostly for ease of development and testing, I soon switched to a different model: two sites, one experience. The mobile site is a neatly separated entity and has its own URL. You have a stand-alone mobile site reachable as a */mobile* virtual directory of the main site or as a subdomain, such as *http://m.yoursite.com*. Sometimes, the site takes its own extension, such as *http.//www.yoursite.mobi*. Any option that you choose here is equally good, in my opinion, and mostly depends on other aspects of the mobile strategy. In any case, it is always a good advice to provide users (at least on smartphones and tablets) with a link to browse the full site.

Because the mobile site is isolated from the principal site, you can test it much more easily—and you can use desktop browsers or mobile generic emulators (such as Opera Mobile Emulator) to perform quick tests aimed primarily at evaluating the user interface and experience. Obviously, it's crucial to test on real devices, but for quick tests on markup, colors, position, and flow, using a desktop program makes the process seamless.

Because you now have two distinct sites, you need an automatic mechanism to switch users to the right site based on the capabilities of the requesting device.

## Routing Users to the Right Site

It's a mistake to assume a one-to-one correspondence between desktop and mobile pages. This may happen but should not be considered a common occurrence. Note that by saying "page correspondence," I simply mean that both applications can serve the same URL; I'm not saying anything about what each page actually will serve.

All in all, we can safely consider only the host name of any requested URL. If the host name belongs to the desktop site and the requesting browser is detected to be a desktop browser, then everything works as expected. Otherwise, the user should be displayed a landing page, where she will be informed that she's trying to access a desktop site with a mobile device. The user is given a chance to save her preference for future similar situations. The preference is stored to a cookie and checked next.

If the request refers to a URL in the mobile site and the user seems to have a desktop browser, consider showing another landing page rather than simply letting the request occur as usual. Finally, if a request is placed from a mobile device to the mobile site, it will be served as expected; namely, by looking into the device capabilities and figuring the most appropriate view. Figure 4-1 presents a diagram of the algorithm.

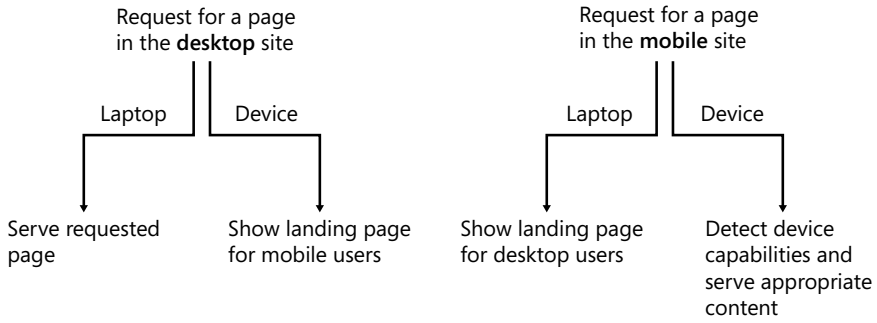



**FIGURE 4-1** The desktop/mobile view switcher algorithm.

How would you implement this algorithm?

In ASP.NET, the natural tool to implement this routing algorithm is an HTTP module that is active on both sites and capturing the *BeginRequest* event. The module will use plain redirection or, if possible, URL rewriting to change the target page as appropriate.

Here's some code that implements the aforementioned algorithm in the desktop site:

```
public class MobileRouter : IHttpModule
{
    private const String FullSiteModeCookie = "FullSiteMode";
    public void Dispose()
    {
    }
    public void Init(HttpApplication context)
    {
        context.BeginRequest += OnBeginRequest;
    }

    private static void OnBeginRequest(Object sender, EventArgs e)
    {
        var app = sender as HttpApplication;
        if (app == null)
            throw new ArgumentNullException("sender");

        var isMobileDevice = IsRequestingBrowserMobile(app);

        // Mobile on desktop site, but FULL-SITE flag on the query string
        if (isMobileDevice && HasFullSiteFlag(app))
        {
            app.Response.AppendCookie(new HttpCookie(FullSiteModeCookie));
            return;
        }

        // Mobile on desktop site, but FULL-SITE cookie
        if (isMobileDevice && HasFullSiteCookie(app))
            return;

        // Mobile on desktop site => landing page
        if (isMobileDevice)
```

```
            ToMobileLandingPage(app);
    }

    #region Helpers
    private static Boolean IsRequestingBrowserMobile(HttpApplication app)
    {
        return app.Context.Request.IsMobileDevice();
    }

    private static Boolean HasFullSiteFlag(HttpApplication app)
    {
        var fullSiteFlag = app.Context.Request.QueryString["m"];
        if (!String.IsNullOrEmpty(fullSiteFlag))
            return String.Equals(fullSiteFlag, "f";
        return false;
    }

    private static Boolean HasFullSiteCookie(HttpApplication app)
    {
        var cookie = app.Context.Request.Cookies[FullSiteModeCookie];
        return cookie != null;
    }

    private static void ToMobileLandingPage(HttpApplication app)
    {
        var landingPage = ConfigurationManager.AppSettings["MobileLandingPage"];
        if (!String.IsNullOrEmpty(landingPage))
            app.Context.Response.Redirect(landingPage);
    }
    #endregion
}
```

Once installed in the desktop site, the HTTP module captures every request and checks the requesting browser. If the browser runs within a mobile device, the module redirects to the specified landing page. The landing page will be a mobile optimized page that basically offers a couple of links: one to the home of the desktop site and one to the home of the mobile site. Figure 4-2 shows a sample landing page viewed with an Android 2.2 device.

If the user insists on viewing the full site, then you can't simply redirect to the plain home page. For its nature, the HTTP module will intercept the new request and redirect again to the mobile landing page. From the landing page, you can simply add a specific query string parameter that the HTTP module will detect on the successive request. Here's the actual link that results in Figure 4-2:

```
<a href="http://www.easycourt.net/contosoen?m=f">Full site</a>
```

You are responsible for defining the query string syntax; in this case, *m* stands for mode and *f* for full. The task is not finished yet, though. At this point, users navigate to the home page of the site. What about any other requests? Those requests, in fact, will be intercepted by the HTTP module. By adding a cookie, you can provide additional information to the HTTP module about requests deliberately sent to the desktop site from a mobile device.

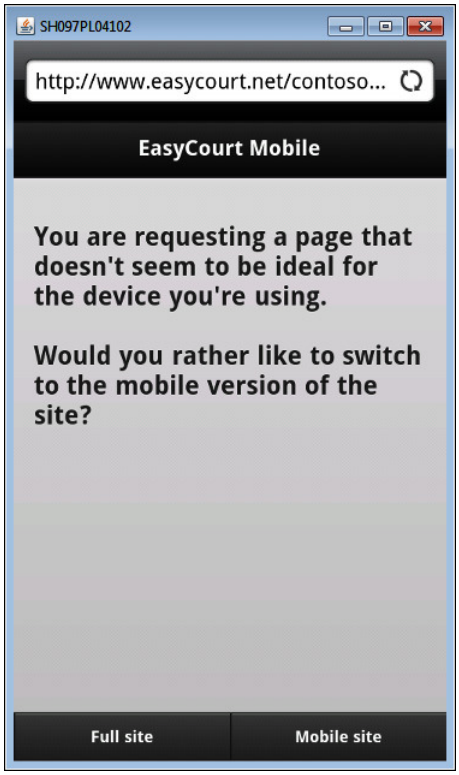



**FIGURE 4-2** The landing page of the EasyCourt demo site.

How can the user switch back to the mobile site? Ideally, any desktop site with a sister mobile site should offer a clearly visible link to switch to the mobile version (and vice versa when the full site is viewed on a mobile device). If not, the user won't be offered a chance to choose the full or mobile site until the cookie expires or is cleared. To clear cookies, users deal with the Settings page of the mobile browser.

## Adding Mobile Support to an Existing Site

Where do you place the landing page? Is it on the desktop or on the mobile site? In general, it doesn't matter; however, if you put it on the mobile site, then you really can enable a scenario in which you deploy a mobile site with all the required routing logic without touching codebase of the existing desktop site.

In the demo site of EasyCourt, a commercial booking system for tennis courts, which I introduced in Chapter 3, I just edited the Web.config file of the desktop site and deployed a library with the HTTP module in the Bin folder. No changes were made to the source code. Here's the configuration script to add a router HTTP module to the desktop site:

```
<system.webServer>
     <modules>
         <add name="MobileRouter" type="..." />
```

```
    </modules>
    ...
</system.webServer>
```

The Welcome page was defined on the mobile site. Note that the Welcome page always should be visible, and it never should need authentication. Depending on how you deploy the mobile site—a distinct root site/application or a child application/directory—you may need to tweak the Web.config file of the mobile site to disable the HTTP module. If the mobile site is a distinct application, then it needs its own Web.config file that has been fully configured with the HTTP module. If the mobile site is, instead, hosted as a child directory in the desktop site, then it inherits the configuration settings of the parent site (the desktop site), including the HTTP module. To speed up requests, you might want to disable the HTTP module in the mobile site.

Here's the configuration script that you need in the mobile site's Web.config file. The script clears the list of HTTP modules required by the mobile site:

```
<system.webServer>
    <modules>
        <clear />
    </modules>
    ...
</system.webServer>
```

In addition, you need to instruct the parent application/site explicitly to stop the default inheritance chain of settings. Here's what you need:

```
<location path="." inheritInChildApplications="false">
<system.webServer>
    <modules>
        <add name="MobileRouter" type="..." />
    </modules>
    ...
</system.webServer>
</location>
```

Also, notice that when the mobile site is a child application/directory, then it inherits a bunch of settings (for the section where inheritance is not disabled) that don't need to be repeated (for example, connection strings and membership providers).

**Note** This example is based on the default Internet Information Services (IIS) 7.5 configuration—integrated pipeline mode. If you're using the classic pipeline mode, then instead of *system.webServer/modules,* you should operate on the *system.web/httpModules* section.

## Design of the Mobile Views

Mobile websites normally show a subset of the content offered by a desktop site. For example, if you have a three-column layout in a desktop site, you probably want to remove (or move to additional pages) content displayed in two of the three columns. Reducing the amount of information improves the load time of the site and makes better use of the available space. "Shrink-and-fit" is a popular slogan.

An ideal layout for pages of a mobile site is based on a single column. The font size is large enough to allow easy reading without zoom. The search bar (if any) ideally will go at the top, and navigation links are commonly placed at the bottom. You might want to place a link to the desktop version of the site somewhere on the mobile site.

Scrolling is accepted, especially vertical scrolling, but endless lists are annoying. Let's briefly review some common scenarios now.

### Input Elements

On a typical mobile site, most of the functionality is read-only. This fact seems to suggest that you are not going to have that many chances to write input forms, and perhaps even that input forms are not an aspect of development you want to invest much time in.

This is just wrong.

Exactly because most of the functionality is read-only and a mobile device is tiny and not as powerful as a laptop, providing specific and restricted parameters is key. To type query parameters, or to specify settings, input forms are a common presence on mobile pages anyway. Some typical web and Windows controls need fixes, though, especially in light of the touch capabilities of many devices.

Typing text on mobile devices is hard and should be minimized. As we'll see in later chapters, this is easier to achieve with native applications, where you can control the input scope of soft keyboards and attempt to display an optimized subset of keys to the user. In mobile sites, all is left to the browser, although developers can use forms of auto-completion via AJAX.

If the browser understands HTML5, then you can just use the most appropriate *type* attribute on the <*input*> element and let the browser do the rest. To be honest, what you get may differ quite a bit, even on smartphones. For example, numbers and ranges are well supported on both iPhone and Android at present, but the date type produces just the plain text box on Android. In iOS5, however, the browser recognizes your intentions and displays the compelling iPhone date picker element (see Figure 4-3).

On the other hand, iOS still lacks the ability to upload files from the browser, which is a feature available in Android mobile browsers.

In general, data entry should be redesigned and even rethought case by case. For example, when it comes to booking a tennis court in EasyCourt, the site offers a drop-down list with ready-made dates, as shown in Figure 4-4. It should be noted, however, that this particular screen is not mobile-specific but simply represents the plain transposition of a desktop page. You always should wonder if there may be an innovative way of letting users enter their choices. Don't stop at the classic, desktop-oriented way of building input forms. Mobile is different and user-centered.


**FIGURE 4-3** Entering dates in iOS5.

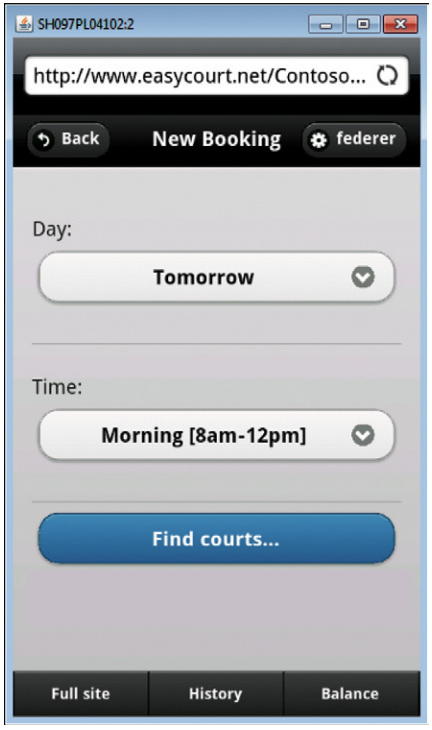



**FIGURE 4-4** Setting parameters for a query on available courts.

And finally, here's a quick tip related to passwords. Strong passwords typically require mixing lowercase and uppercase letters and numbers and symbols, but that's too much work in a mobile website. If possible, consider using numeric personal identification numbers (PINs).

## Radio Buttons and Check Boxes

A common presence in many input forms, radio buttons and check boxes have just one problem in mobile forms: they tend to be too small and hard to select with a touch. It is recommended that you style these elements accordingly using padding and the companion *label* element. Don't place such elements too close to other elements, including other possibly related buttons.

**Note** On iOS, the tappable region is 44 points, which corresponds to a square of 7 mm. This should be considered if you are creating a mobile site without using a framework such as jQuery Mobile, which shields you from many details.

It is worth noting that jQuery Mobile completely rewrites the canonical markup for such input elements, as follows:

```
<input type="checkbox" name="rememberme" id="rememberme" data-theme="d" />
<label for="rememberme">Keep me logged</label>
```

The output produced by the preceding markup is shown in Figure 4-5.

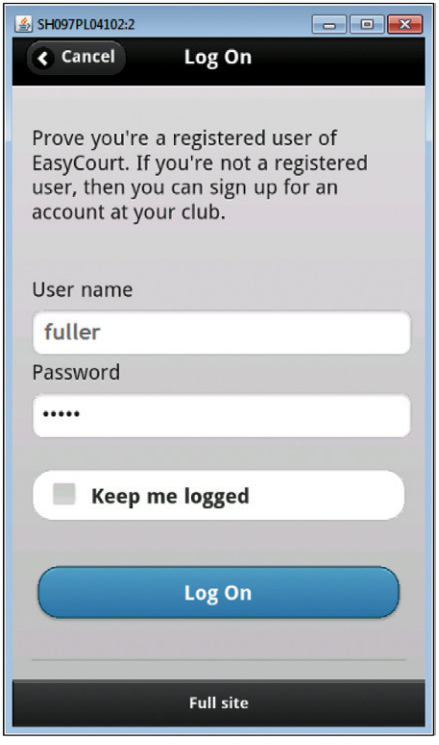



**FIGURE 4-5** A check box that looks like a button and is really easy to tap.

In particular, jQuery Mobile surrounds the base HTML markup with padded DIVs to make it easier for the user to tap. Compare the experience of tapping this with a regular check box that may even need zooming to be read and selected (see Figure 4-6).

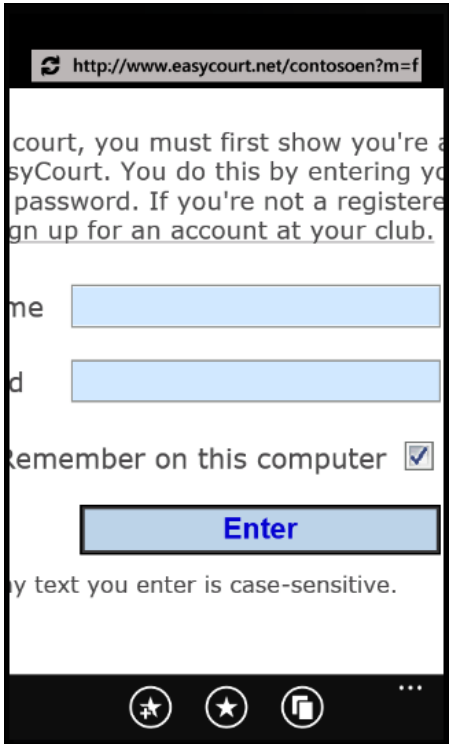



**FIGURE 4-6** A regular HTML check box, but zoomed.

When the user's choice is a *yes/no* or *on/off* choice, it is common in mobile to use flip-switch artifacts that are larger and more comfortable to use and easier to see. Although differently styled, flip-switches are available in iOS, Android, and Windows Phone.

## Scrollable and Drop-Down Lists

Radio buttons are a valid solution if you need only one selection out of just a few options. When the number of options gets closer to 10, or more, you might want to consider a combo box or a plain scrollable list.

One of the original Windows controls, the combo box provides the same experience on desktop applications and websites. The same points previously discussed for radio buttons and check boxes apply to combo boxes too—they're usually too small to be touch-enabled effectively. You can play with style sheets and use a larger font, but you manage to improve the result only a little bit—it remains far from ideal. Figure 4-7 compares native and adjusted combo boxes.

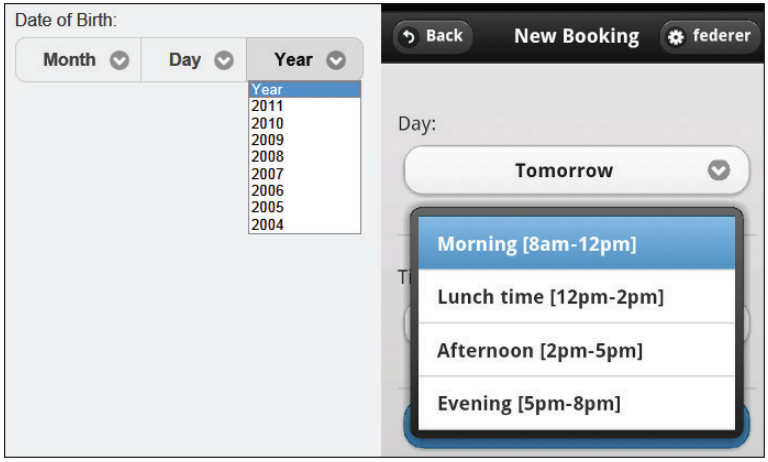



**FIGURE 4-7** Native and adjusted combo boxes.

Figure 4-7 shows the results that you get when you use jQuery Mobile. The bottom line, though, is that you hardly want to use plain <*select*> elements on a mobile page without some graphical adjustments for size and touchability. On the other hand, <*select*> elements are more important than ever in mobile pages because they can save users from typing free text.

Scrolling vertically is quite a natural gesture in a mobile scenario. However, the more you can group related items, the better the final experience for the user is. Once again, focus is king. Figure 4-8 shows a couple of mobile views featuring grouped items and collapsible elements that effectively save valuable screen real estate while providing optional information to users. Grids are effectively rendered on mobile only rarely; smartly built vertical lists are a much more common and preferable choice.

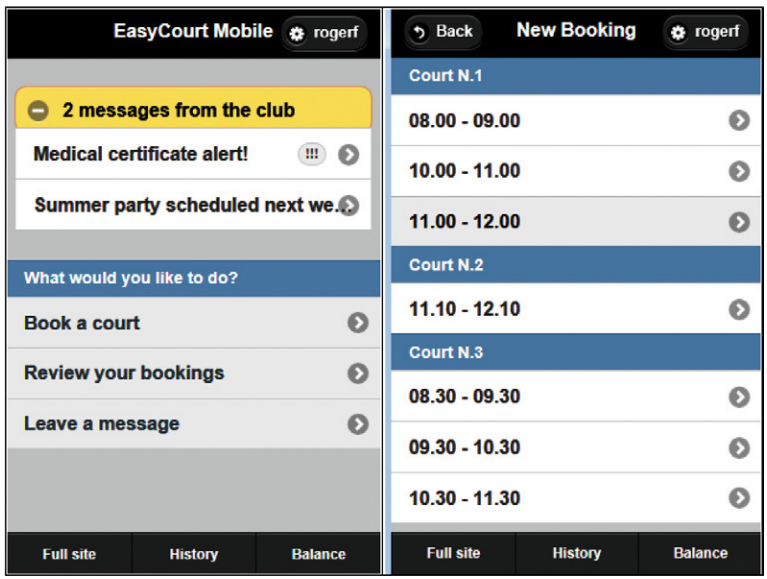



**FIGURE 4-8** Collapsible and grouped elements.

In Figure 4-8, a collapsible element is the block that shows messages. As the icon suggests, it is an area of the screen that can be collapsed. Other blocks are tables with one or multiple sections (to use a terminology popular to iOS and Android developers). Some of the items serve as dividers of the sections and are styled differently, communicating the idea of grouped items.

If the list of items is particularly long, you might want to introduce some pagination. Essentially, pagination works by first loading a fixed number of items and then displaying a link at the end to get more. Alternatively, more items can be loaded automatically when the user reaches the bottom of the list. It is up to you to ensure that the DOM doesn't end up containing too many items after a few click requests for more items. If this happens, you might want to remove older items and move them to another page that the user can request if needed.

**Note** If you want, you can play live with EasyCourt at *http://www.easycourt.net/contosoen*. By reaching the site with a laptop and a mobile device, you can experience the different levels of usability. The desktop site didn't undergo any change to support mobile devices except for the configuration file. For this reason, the desktop site currently doesn't include a link to the mobile site. This means that if you switch to the full site from a mobile one, you can't change back until you remove cookies for the site manually using the browser's Settings page.

## Free Text and Auto-Completion

No matter your efforts to save free text typing, sometimes users are just requested to enter a name or a comment. There's not much you can do to reduce the hassle of typing on a mobile soft keyboard if you can't figure out ways to allow the browser to offer a close-enough input scope. With native applications, it's slightly better, but text typing remains a very sore point for mobile developers.

Auto-completion does help, but it costs you quite a bit of script code, which in turn increases the payload for the page. Auto-completion must be coded through a plug-in—the jQuery UI auto-complete plug-in works well with mobile pages. A plug-in that downloads data from an external source is the most viable option when the potential number of options is in the order of hundreds. For a smaller number of items, you can also consider the jQuery Mobile filter bar. The base markup is the following:

```
<ul>
  <li> ... </li>
  <li> ... </li>
  <li> ... </li>
</ul>
```

Additional attributes will give the final markup a strong auto-completion flavor, as in Figure 4-9. Auto-completion also is sometimes a way to avoid long lists of hundreds of items that are very boring to scroll through.

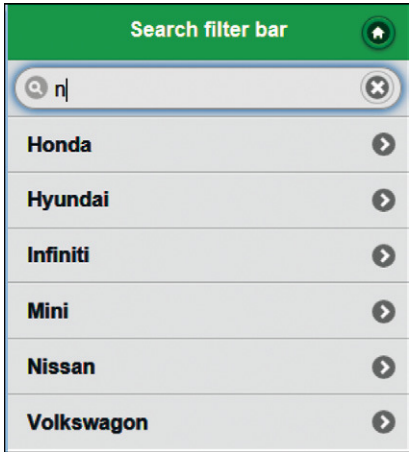



**FIGURE 4-9** A jQuery Mobile filter bar selecting list items with "n."

## Testing the Mobile Site

Testing a mobile site is not an easy task. You may use a mobile emulator (possibly more than one to better simulate the various scenarios), but emulators are not enough. Testing on real devices is what really matters and gives you the real perception of the application performance and user experience.

However, before you deploy to a few test devices (at least one per device class), you might want to check user interface and logic on a more comfortable desktop machine. Emulators and user-agent switchers can do some of the work for you.

### Desktop Emulators

An *emulator* is a desktop application typically running on Windows that aims to mimic the behavior and functions of a specific browser. In mobile development, emulators are time-saving tools for first-pass software testing.

Device/OS emulators are more specific than browser emulators. Device/OS emulators (i.e., the Windows Phone emulator) are created by device manufacturers and are generally really close to the device.

Browser emulators are generally written by third-party companies and serve mostly for verifying the look and feel of the site at a given size and resolution. You should not consider a browser emulator for verifying the rendering of pages that have been optimized for certain classes of devices.

In general, emulators are a first-aid device, and they can be used to test the behavior of the application on types of devices that you have no access to.

### User Agent Switching

Most desktop browsers come with tools to switch the user agent (UA) string. By changing the default UA string to that of a mobile device, you allow the browser to present the website with the credentials of a particular mobile browser. Subsequently, the website will serve the mobile content that it has in store. UA switchers are a special case of browser emulators.

The Firefox browser was the first to offer such a powerful add-on. Today, you find the same tool for Google Chrome and Internet Explorer 9, as shown in Figure 4-10.

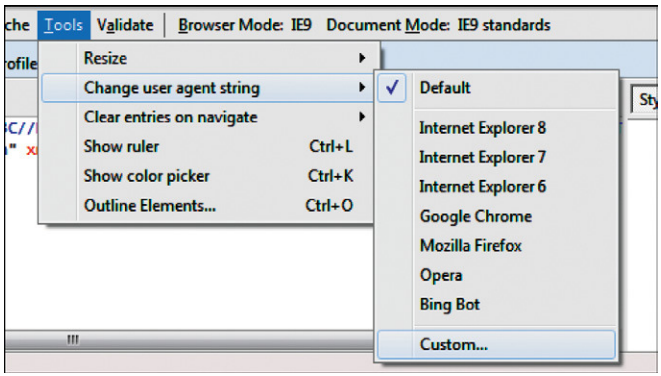



**FIGURE 4-10** The User Agent Switcher tool in Internet Explorer 9.

Opera Mobile Emulator is another interesting tool through which you can experience your site as if you are visiting it through a number of specified devices. Compared to the User Agent Switcher Tool, the Opera Mobile Emulator is more flexible; more important, it is a single tool that can serve multiple scenarios. In Figure 4-11, you see the emulator configured to give the site being tested the appearance of running on a tablet.

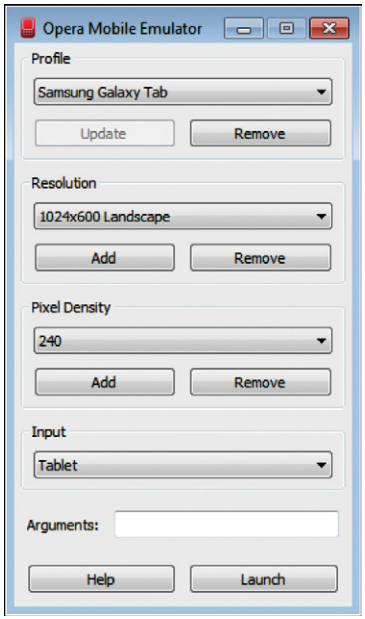



**FIGURE 4-11** Selecting an emulator.

With all this said, it is worth making this point even clearer: you should test your mobile site on a variety of mobile devices. In particular, don't trust too much the apparent magic of UA switchers.

Let me illustrate a specific scenario. The UA switcher forces the browser to send a particular UA string and, subsequently, your server-side code recognizes the requesting browser as mobile. This is the only certain fact.

At this point, your server-side code may inject some JavaScript that checks the browser's capabilities and adapts the markup. Your JavaScript code, however, will be querying the desktop browser, not the mobile browser! Furthermore, a generic emulator like the Opera Browser Emulator may not be able to imitate the real browser perfectly, which makes the test experience significantly less valuable.

The bottom line is that you should use switchers for quick testing, then move to device emulators, and finally test on a selection of actual sample devices—the only source of truth.

**Note** If you want to test your mobile views on specific devices (well, mostly smartphones), all you need to do is get a hold of the same emulators that you use for testing native applications and then use the browsers that they offer.

In addition to buying or renting as many devices as you can, you can use a device-testing service such as Keynote Device Anywhere (*http://www.deviceanywhere.com*). Another tool that is great for remote debugging is Weinre (*http://phonegap.github.com/weinre*).

## The Device-Detector Site

As an example of a mobile site that serves device-specific content, let's consider what it takes to build a device-detector website with ASP.NET MVC. Figure 4-12 shows the desktop version of the site that is being made mobile. It has a three-column layout and a variety of links. This site doesn't do any particularly fancy things: it is limited to displaying the UA string and a few other browser capabilities.

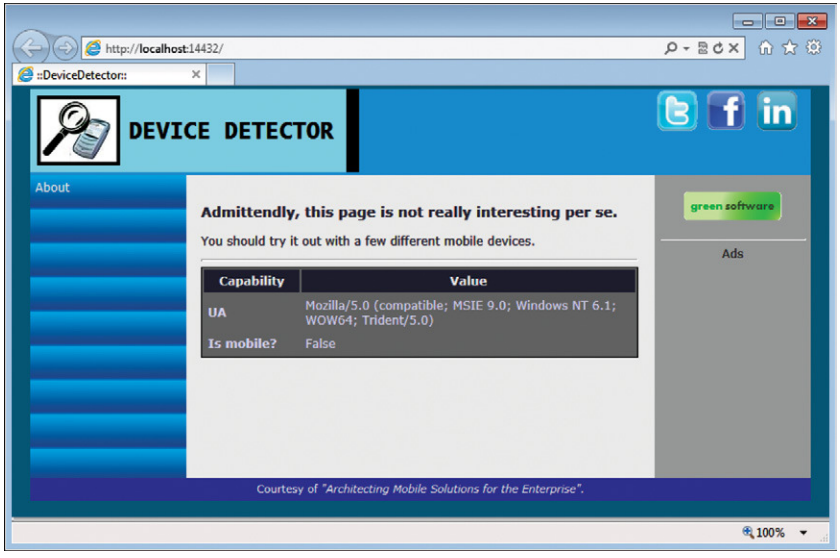



**FIGURE 4-12** The desktop version of the Device-Detector sample site.

A mobile version of this site will remove a lot of bells and whistles and mostly focus on the primary use-case: providing details about the current browser.

You should expect to see a single-column template with some graphics at the top (header) and bottom (footer). In addition, a couple of links in the main content area just connect users to the page with device details and contact (see Figure 4-13).

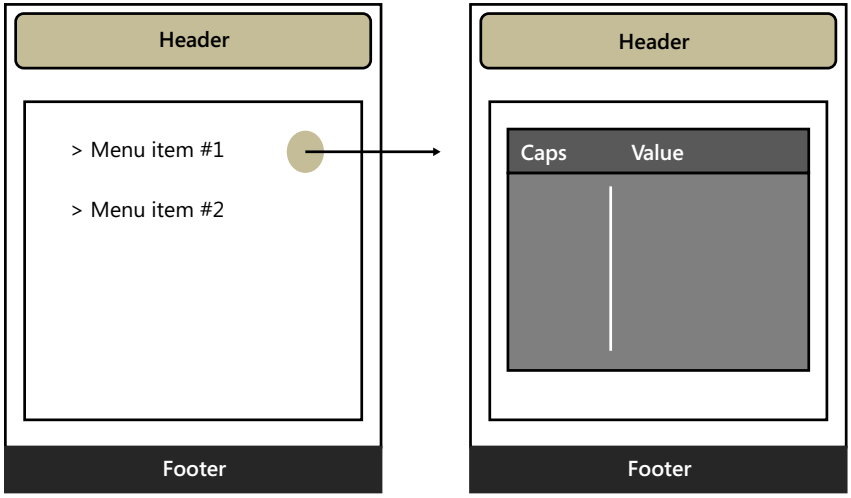



**FIGURE 4-13** The mobile layout for the Device-Detector sample site.

In the early days of software development, about 20 years ago, it was natural to design the main screen of applications (mainframe and desktop) through a main menu that accepted user selections via mouse clicks or keystrokes. Nearly all applications were designed around a main menu that collected the various use-cases of the application.

The advent of pop-up and floating menus marked the end of this approach, which clearly was limited to moderately complex applications. With mobile sites (and to some extent, also with native applications), this old-fashioned model is revamped due to the relative simplicity of the logic and the need to be direct and focused.

## Routing to Mobile Views

For a site that serves mobile content, an accurate detection of the device is key. In addition, a (possibly) automated mechanism to route requests to the right view would be welcome. Up to ASP.NET MVC 3, you have no tools to allow you to select the view and no convention-over-configuration (CoC) approach.

In ASP.NET MVC 4, instead, an enhanced infrastructure lets you give mobile views conventional names that are resolved automatically by the system. For example, in ASP.NET MVC 4, a view named *index.mobile.cshtml* will be used to serve requests for the *Index* action when coming from a generic mobile device. ASP.NET MVC 4 is expected to give you even more control over the presentation layout, as it also supports syntax like *index.iphone.cshtml* for a specific class of mobile devices.

**More Info** For more information about MVC 4, refer to *http://www.asp.net/mobile*. In addition, you might want to look at my *Programming ASP.NET MVC* book for a deep coverage of ASP.NET MVC 3.

## Configuring a Mobile-Aware View Engine

The source code that comes with this book provides a sample view engine that works well with ASP.NET MVC3—the *AmseViewEngine* class. The engine builds on top of the built-in *Razor* view engine and allows you to invoke both desktop and mobile views. If the browser is detected as a mobile browser, the engine will attempt to resolve any view named *Xxx* as a view named *Xxx.mobile*. If the mobile browser belongs to a specific class, then it is mapped instead to an *Xxx.profile* view, where *profile* indicates the name of the class.

You register the mobile-aware view engine in *global.asax,* as shown in this code:

```
public static void RegisterViewEngines(ViewEngineCollection viewEngines)
{
   viewEngines.Clear();
   viewEngines.Add(new AmseViewEngine(new AspnetMobileViewResolver()));
}

protected void Application_Start()
{
   ...
   RegisterViewEngines(ViewEngines.Engines);
}
```

The constructor of the view engine receives a user-defined class that knows the strategy to transform the originally requested view to see if it exists in the site. In particular, the *AspnetMobileViewResolver* class checks for *xxx.mobile* instead of *xxx* if it detects that the requesting browser is mobile.

View engines are a specific feature of ASP.NET MVC. In ASP.NET Web Forms, you can achieve the same result by registering an HTTP module that intercepts incoming page requests, detects whether the requesting browser is a mobile browser, and redirects to a corresponding mobile page, if any.

**Note** The *AmseViewEngine* can be extended in a fairly easy manner to look for mobile views in a distinct folder, such as Mobile. All you need to do is change the default value of the *ViewLocationFormats* and *PartialViewLocationFormats* properties to point them to your new Mobile folder. You set the properties in the view engine constructor.

## Routing to Mobile Resources

Just as any other type of view, a mobile view may rely on a bunch of external resources, such as images, style sheets, and script files. Because you control the markup of the mobile view, you can simply place all the references you need there. For example, you can link the jQuery Mobile library

only from the mobile layout; likewise, you might want to use smaller images for a mobile view or, better yet, you might want to embed images in the same view as Base64-encoded strings.

It is crucial to note, however, that mobile views may have their own set of resources. Most of the time, you don't need to differentiate mobile resources on a per-device-class basis.

## Detecting Device Capabilities

So now there is a mechanism that can redirect automatically to a view that has been specifically designed for mobile browsers. But which part of the ASP.NET run time determines whether the requesting browser is a mobile browser? And, more important, which algorithm is used?

As you may understand, this is the central point of mobile site development—you may not need detailed information in all cases about what a given device can or cannot do, but you always need to know—with extreme accuracy—at least whether the requesting browser is mobile or not.

### ASP.NET Native Detection Engine

ASP.NET has its own detection API centered on the following code:

```
HttpContext.Request.Browser.IsMobileDevice
```

The *IsMobileDevice* property returns a Boolean value and indicates whether the current request comes from a mobile device.

Without beating around the bush, this code is not really reliable. For example, it fails on a number of popular devices, such as the HTC Desire and Samsung Galaxy S smartphones (both equipped with Android); it also fails on a wide range of simpler devices, such as the Samsung GT S3370 Corby. Curiously, the native ASP.NET detection API succeeds with the BlackBerry, iPhone and iPod devices, and with Windows Phone devices. Why is this so?

The value returned by the *IsMobileDevice* property results from a partly accurate analysis of the UA string that ASP.NET performs under the hood. Essentially, ASP.NET uses the UA string as a key to match the requesting browser to one of the predefined device profiles. A *device profile* is a text file with a *.browser* extension located on the server under the Windows folder at the following path:

```
// This is the path if you have the .NET Framework 4 installed on the server
\Microsoft.NET\Framework\v4.0.30319\Config\Browsers
```

Figure 4-14 offers a preview of the typical content of this folder.

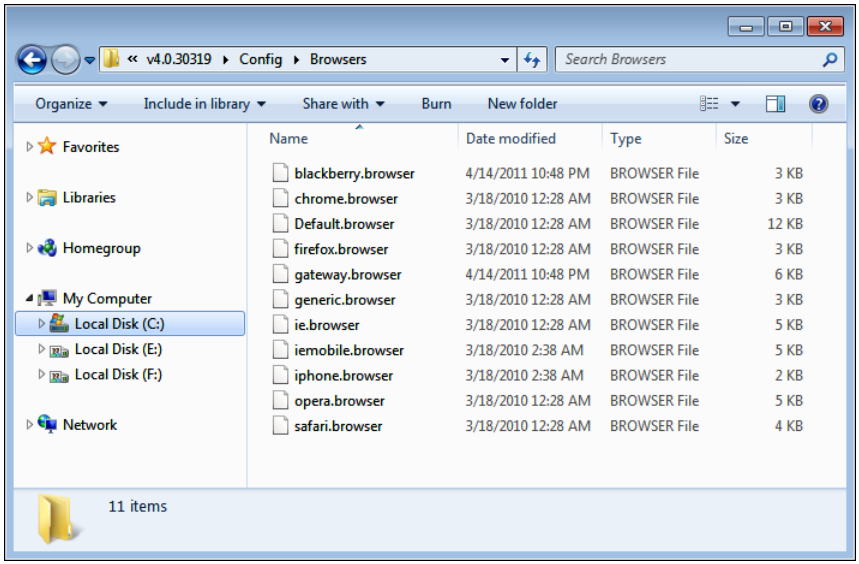



**FIGURE 4-14** The content of the Browsers folder.

Files in this folder contain some basic information about a few families of browsers. Each family contains a regular expression used to match the UA string. If a match is found, then the dictionary of browser capabilities exposed to the application is filled with the values of the properties that are known to apply to that family of devices.

This solution was devised years ago when the problem was detecting just a few desktop browsers. Matching the UA string of a mobile device—and its nearly infinite variations—and identifying the right value for a given property require a much richer and articulated database. The content of browser files can be extended and new files can be created, but it doesn't change the basic fact that something stronger is needed.

The bottom line is that if you simply rely on the *IsMobileDevice* property, you seriously risk offering a desktop site to many mobile devices. Worse yet, this especially happens with older devices that will display just a basic site, to the frustration of users. What else can you do?

## A Better Way of Detecting Mobile Devices

Detecting device capabilities is a difficult problem in mobile (not just ASP.NET mobile) because of the wide fragmentation of devices. In my opinion, the device fragmentation problem has just one *exact* solution that I'll discuss thoroughly in Chapter 6. This solution is using a DDR like the Wireless Universal Resource File (WURFL).

Most DDRs are not free for every use; and free versions of most DDRs just reduce severely the number of capabilities that they return. The bottom line is that you should be ready to spend money when it comes to DDRs; your money will be well spent.

Anyway, I'd like to illustrate quickly a couple of totally free (but possibly only approximate) solutions that you might want to consider before you go to Chapter 6 and pick up your favorite DDR framework. I'll leave it up to you whether any of these options may work for you in the real-life battlefield.

The first option entails writing your own wrapper around the *Browser.IsMobileDevice* property. You can create it as your own class, or perhaps as an extension method to the *Request* object. Regardless of these implementation details, what really matters is the logic that you use to write the code. Here's a sample detection function written as an extension method for the native ASP.NET *Request* object:

```
public static class RequestExtensions
{
    public static Boolean IsMobileDevice(this HttpRequestBase request)
    {
        var response = request.Browser.IsMobileDevice;
        if (response)
            return true;

        // If response is false, there are still good chances to have a mobile device.
        // Let's check the user-agent string for common substrings.
        var userAgent = request.UserAgent.ToLower();
        response = userAgent.Contains("opera mini") ||
                   userAgent.Contains("mobile") ||
                   userAgent.Contains("samsung") ||
                   userAgent.Contains("nokia") ||
                   userAgent.Contains("htc") ||
                   userAgent.Contains("android") ||
                   userAgent.Contains("windows phone") ||
                   userAgent.Contains("midp") ||
                   userAgent.Contains("cldc");
        return response;
    }
}
```

If the *IsMobileDevice* native property returns *true,* then you can be sure that the requesting browser is really a mobile browser. The problem is with false negatives. The simplest thing you can do is check the UA string looking for common substrings associated with mobile agents. The preceding example includes some manufacturer names and operating system names. The MIDP string refers to the Mobile Information Device Profile (MIDP), a specification that is part of the Java Platform Micro Edition (Java ME) framework. MIDP works on top of the Connected Limited Device Configuration (CLDC), which is instead a lower-level specification. In the end, both MIDP and CLDC are strings that appear often in UA strings sent by mobile devices.

Finally, you may have noticed that this list contains no reference to popular devices such as iPhone (and iPod/iPad) and BlackBerry. This is because ASP.NET 4 comes with *.browser* files for detecting both platforms (see Figure 4-14). As a result, the basic *IsMobileDevice* property works on iOS and BlackBerry devices.

A more powerful approach is based on a repository of mobile profiles that Microsoft built for internal purposes and made public through a CodePlex project: *http://mdbf.codeplex.com*. It's called the Mobile Device Browser File (MDBF). If you visit the website, however, you find out that it is a

dead project now. This means that the database for this project won't be maintained and extended any longer, although you still can download and use it. With the wave of new devices being released every month, this is clearly a problem.

To use the MDBF repository, all you need to do is copy the file in the App_Browsers/Devices folder of your website. The content of the file will be read by the ASP.NET infrastructure and used to populate the dictionary of browser capabilities.

You can extend the MDBF repository—an 18 MB XML file—to keep it up to date. Likewise, you can update *.browser* files and add new ones. Both options, however, are not compelling because they require a lot of maintenance work and research. And this is probably the reason why approximate and exact solutions exist, and exact solutions are not free.

## DDR Options

The quintessential DDR is, without a doubt, WURFL, by ScientiaMobile (*http://www.scientiamobile.com*). WURFL was created in 2002—four years before the DDR acronym was first coined on a World Wide Web Consortium (W3C) mailing list. An open-source community-based initiative, WURFL is comprised of an API and a data repository. The API maps incoming HTTP requests to a known device definition and then retrieves known capabilities for that device from the repository. The repository is updated independent of the API so that companies just refresh the repository periodically without the need to change or rebuild the application. The WURFL API is available for a variety of platforms and languages, including ASP.NET, Java, C++, Ruby, and PHP. See *http://wurfl.sourceforge.net*.

In the summer of 2011, the WURFL owners have moved the project to a different licensing model, which is stricter in many ways. While the API is technically still open-source, the Affero GPL v3 license now requires that users completely open-source the proprietary code linked to the WURFl API on their servers. This requirement is typically not compatible with the requirements of commercial entities. Therefore, to avoid open-source provisions, companies can buy a commercial license for WURFL API and data from ScientiaMobile. Note that the WURFL repository is distributed with a proprietary license that prevents you from copying the WURFL data and using it with third-party APIs.

Another interesting DDR specifically aimed at the ASP.NET platform is 51Degrees. See *http://51degrees.codeplex.com*. 51Degrees relied originally on WURFL as the source for device information, but the change in the licensing model of WURFL forced to adopt a different and proprietary repository. Recently, 51Degrees has been relaunched with a new vocabulary (i.e., set of property names) and new data. 51Degrees is a purely commercial initiative, but it offers a free version as a teaser for the platform. The free version is limited to a DDR with just four properties: *isMobile*, *ScreenPixelWidth*, *ScreenPixelHeight,* and *LayoutEngine,* which is simply the browser engine.

Other players in the DDR world are DetectRight (*http://www.detectright.com*) and DeviceAtlas (*http://www.deviceatlas.com*). MobileAware (*http://www.mobileaware.com*) and NetBiscuits (*http://www.netbiscuits.com*) are also names worth mentioning, although they do not offer pluggable DDRs as part of their main business model. In other words, the DDR is just one component of a more elaborate product.

**Note** Search engines may point you to a few other device detection initiatives, mostly created as an aspect of WURFL and, for this exact reason, subject to legal dispute. Accuracy and level of service, however, don't currently compare to any of the DDRs discussed here.

## CSS Media Queries

Lateral thinking is about solving problems using an innovative approach and unconventional reasoning. The lateral thinking about device detection seems to be CSS Media Queries and responsive (or adaptive) web design. Is detecting devices hard? Don't do that, then; just take a bunch of basic properties (e.g., screen size) and let the page adapt and reflow accordingly.

The magic potion that enables responsive web design is CSS Media Queries. Introduced with CSS 3, media queries simplify the design of sites that might be consumed through devices of different screen sizes ranging from 24 inches on a desktop monitor to 3 inches on most smartphones. Media queries are not specifically a technology for mobile development, but the flexibility of this feature makes it really compelling to use to serve different devices with a single codebase.

The idea, in fact, is that you just create one site with a single set of functions and then apply different CSS styles to it by loading a different style sheet for different media. The great improvement brought by CSS 3 is that a medium (such as a screen) now can be restricted to all devices that match given rules. Here's an example of media queries:

```
<link type="text/css"
      rel="stylesheet"
      href="downlevel.css"
      media="only screen and (max-device-width: 320px)">
```

Placed in a HTML page (or view), this markup links the Downlevel.css file only if the page is viewed through a browser with a width of 320 pixels or less. Note that there's no explicit check on the type of browser, whether mobile or desktop: all that matters is the real width of the screen. (Needless to say, with a screen width of 320 pixels, it can only be a mobile phone or handheld device.) The *only* keyword should be added for the sole purpose of hiding the statement from browsers that don't support media queries. These browsers, in fact, don't understand the media type and go right ahead. The full documentation about media queries can be found at *http://www.w3.org/TR/css3-mediaqueries*.

What's the problem with media queries?

It is a common idea these days that by simply adding media queries to a site, you make it ready for mobile clients. CSS media queries help making the page content more mobile-friendly, but they don't affect other critical areas, such as the number of HTTP requests per page, whether DOM manipulation and AJAX are supported, or if a touchscreen is available.

Media queries can check out only a limited number of browser properties—namely, those listed in the W3C standard: device width and height, orientation, aspect ratio, color depth, and resolution to name the most frequently used ones. Most properties support the *min-* and *max-* prefix to help you write more precise queries. Being a CSS feature, media queries only can hide elements that are

too big or low-prioritiy to display on a small screen. You still pay the costs of downloading or keeping these elements in memory. You can use some JavaScript in the pages to download or configure images programmatically. In this way, heavy elements can be managed in a more optimized manner.

In addition, media queries require a browser that supports CSS3. So they work on most smartphones, but not, for example, on Windows Phone 7.0. An all-browser solution for media queries is available through a jQuery plug-in that you can get at *http://www.protofunc.com/scripts/jquery/mediaqueries*. However, there's no guarantee that the mobile browser where you may be using this plug-in can really run jQuery.

### Browser Capabilities

Detecting whether the requesting browser runs on a mobile device is only the first step toward delivering an adequate experience to any mobile users. Once you have identified a device correctly as a mobile device, you should try to detect the capabilities of the device. For example, you might want to know the version of the operating system, its real screen width and height, whether it supports AJAX, if it can perform some DOM manipulation, and if CSS is supported. In addition, you might want to optimize the user interface in case the device has a touchscreen or is really a tablet.

Finally, have you ever tried to visit a site with an older phone? If you have, then you know what I mean. First, the phone will likely have no support for WiFI, so it will connect over the mobile network. The slow connection will take a lot of time to see how many different connections are made to download images, scripts, and auxiliary files. You might always want to merge CSS and minify scripts, but what about images? Sprites are a possible solution, but they require CSS support from the device. Inline images (namely, images embedded in the page as Base64 strings) are another route to explore.

You need to know these and possibly more details about the specific device. Some properties can be tested programmatically with a bit of JavaScript. The following code, for example, shows how to check programmatically whether AJAX is supported:

```
var xhr = window.ActiveXObject ? new ActiveXObject("Microsoft.XMLHTTP") : new XMLHttpRequest();
if (xhr === null)
    alert("No support for Ajax");
```

Many other properties can't just be tested programmatically. For instance, how would you detect programmatically if a device understands inline images, has a touchscreen, or is a tablet? For these and other types of capabilities, you need a repository of information that is updated weekly, if not more frequently. If delivering a great user experience on a variety of mobile devices is your goal, then you need the appropriate tool. And you probably need to pay for it.

## Putting the Site Up

At the end of the day, a mobile site is just a website that has been designed according to a different set of guidelines. Once you know whether the device is mobile and what its capabilities are, you can proceed safely with the actual design of the site—layout, style, and markup.

## Adjusting the Layout

In a mobile site, you might want to use mostly a single-column layout and move navigation and search functions to the top and bottom of the page. Finally, you might want to leave the user free to scroll vertically to locate what is relevant but doesn't fit on the physical page. Beyond these basic rules, the design of a mobile site is all about finding the most friendly and creative way of presenting your content. Here's the layout file for the mobile version of our site.

Note that the listing uses the ASP.NET MVC *Razor* syntax to describe the view. The *Razor* syntax mixes plain HTML with executable expressions. Executable expressions are prefixed with the @ symbol. In particular, in the following example, the *ViewBag* expression refers to a collection through which the page passes values to the view. A good step-by-step tutorial to *Razor* can be found at *http://goo.gl/9eTEm*.

```
<html>
    <head>
        <meta name="viewport"
              content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
                       user-scalable=no" />
        <title>@ViewBag.Title</title>
        <link href="@Url.Content("~/Content/Styles/Site.css", mobile:true)"
              rel="stylesheet"
              type="text/css" />
    </head>
    <body>
        <div id="header">
            <img src="@Url.Content("~/Content/Images/logo.png", mobile:true)"
                 class="image" alt="" />
        </div>
        <div id="content">
            <h2>Know Thy Devices</h2>
            <p>
                Find out details of the devices that visit your site.
                This demo also shows a sample mobile template.
            </p>
        </div>
        <div class="actual-body">
            @RenderBody()
        </div>
        <div id="footer">
            <p>Architecting Mobile Solutions for the Enterprise</p>
        </div>
    </body>
</html>
```

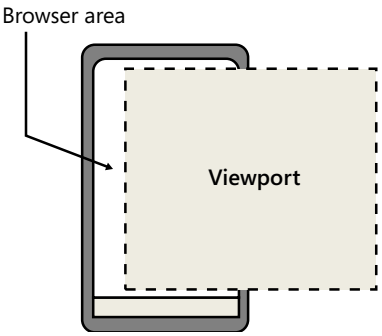
Note that this file is named Layout.mobile.cshtml and is resolved in _Viewstart.cshtml, using an enhanced version of the native *UrlHelper* object in ASP.NET MVC:

```
@using Mobi.Framework.ViewEngines;
@{
    Layout = Url.Content("~/Views/Shared/_Layout.cshtml", mobile:true);
}
```

As mentioned, the layout file implements a single-column view and points to external resources using our custom *Url.Content* method as a resource switcher.

Let's find out more about the *viewport* <*meta*> tag.

Most mobile browsers can be assumed to have a rendering area that's much larger than the physical width of the device. This virtual rendering area is called the *viewport*. The real size of the internal viewport is browser-specific. However, for most smartphones, it is around 900 pixels. Having such a large viewport allows browsers to host nearly any webpage, leaving users free to pan and zoom to view content, as in the following illustration:




This behavior may perhaps be desirable (or at least not too disturbing) when you have a high-resolution smartphone; but what if your users host the site within a 240 × 320 device? It's like looking through a keyhole. To gain control over mobile browsers' viewports, you add an explicit viewport <*meta*> tag and instruct the browser about it as follows:

```
<meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
```

In this example, you tell the browser to define a viewport that is the same width as the actual device. Furthermore, you specify that the page isn't initially zoomed and can't be zoomed in by users. Setting the width property to the device width is fairly common, but you can indicate an explicit number of pixels. Figure 4-15 shows how the same page looks on an older device with and without the viewport tag.

## Adjusting the Style

A mobile site deserves its own CSS file where you define a bunch of new styles required by the specific user interface. In addition, the CSS file will probably need to override some of the styles shared with the desktop site (if any).

For example, you might want to remove background images and replace them with solid colors. Background images are a great example to show how CSS media queries can hardly be the perfect fit when you need to provide multiple views for devices other than smartphones. With media queries, you only have the device width to distinguish devices. However, when it comes to devices less than 300 pixels wide, you can still find different capabilities. Some relatively powerful devices fall in this category that have touchscreen and good HTML capabilities. For these devices (e.g., Samsung Corby),

you can still use background images, but you cannot for any devices smaller than 300 pixels. This is to say that you can do a lot with CSS styles, but not everything. Beyond a threshold, you just need to upgrade to another solution as WURFL. (See Chapter 6 for more information.)


**FIGURE 4-15** Effects of the viewport tag.

In a mobile style file, it is important to use the padding property appropriately to ensure that clickable elements (in touch-enabled devices) are large enough to accommodate a relatively inaccurate pointing device like the human finger.

## Adjusting the HTML View

The sample site, Device Detector, while not realistic in terms of functionality, is an excellent starting point for understanding the role of browser capabilities. Figure 4-16 compares the desktop and mobile versions of Device Detector.

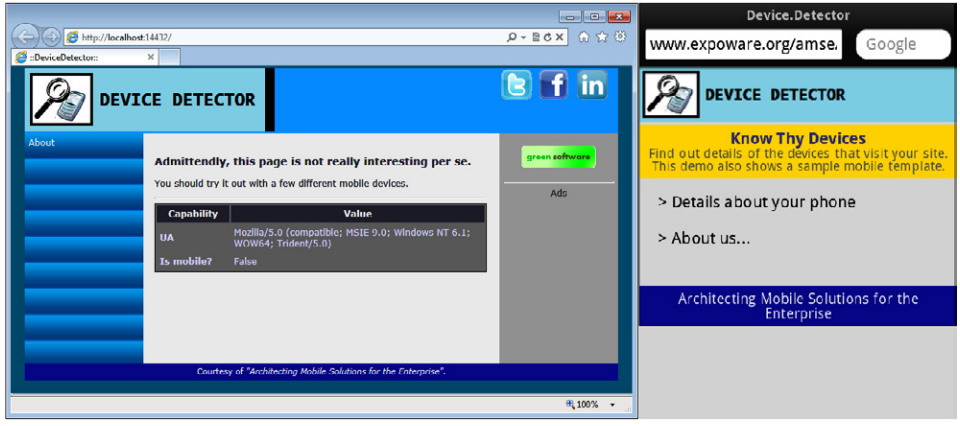



**FIGURE 4-16** Desktop and mobile site face to face.

The mobile site requires an extra step to get to the real data: you must click the Details link. The *Index* view is therefore different in our ASP.NET MVC application. The *Index.mobile.cshtml* view simply renders a static markup with a couple of links. It is interesting to see where the *Details* link points. It points to a *Details* action method on a new controller—the *DeviceController*—that is used only by the mobile subsystem:

```
public class DeviceController : Controller
{
    public ActionResult Details()
    {
        ViewBag.UserAgent = Request.UserAgent;
        ViewBag.IsMobile = Request.Browser.IsMobileDevice;
        ViewBag.SupportTables = Request.Browser.Tables;
        ViewBag.MobileDeviceInfo = String.Format("{0}, {1}",
                                                 Request.Browser.MobileDeviceModel,
                                                 Request.Browser.MobileDeviceManufacturer);
        ViewBag.PreferredImageMime = Request.Browser.PreferredImageMime;
        ViewBag.ScreenSize = String.Format("{0} x {1}",
                                           Request.Browser.ScreenPixelsWidth,
                                           Request.Browser.ScreenPixelsHeight);
        ViewBag.SupportAjax = Request.Browser.SupportsXmlHttp;
        ViewBag.DomVersion = Request.Browser.W3CDomVersion;

        // Point to the device.cshtml view
        return View();
    }
}
```

The method would return the view generated from the Details.cshtml template. However, because this method is invoked only from the mobile site, the actual view file will be Details.mobile.cshtml. However, the view engine being used here can pick up Details if it can't find Details.mobile.

The *Details* method collects some data about the requesting browser and composes them into the view. As you can see in this example, the information about browser capabilities is what ASP.NET makes available natively. I actually took the screenshots of this chapter from a site where I installed the MDBF repository—now largely outdated, but far better than the default ASP.NET browser configuration.

In particular, you can consume some of the capabilities being passed down to the view to fork your rendering code, as shown here:

```
@if (ViewBag.SupportTables)
{
    <table id="deviceTableInfo" cellpadding="4" cellspacing="2">
    ...
    </table>
}
else
{
    <ul>
        <li> ... </li>
        ...
    </ul>
}
```

Figure 4-17 shows a screenshot of the site captured from an iPod device.

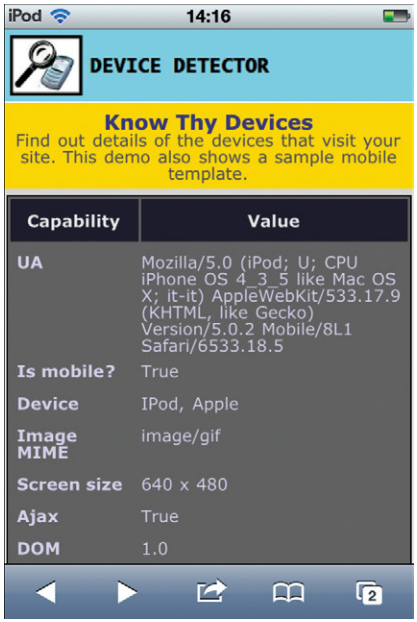



**FIGURE 4-17** The Device Detector site displayed on an iPod device.

Figure 4-18, instead, shows the same site on a low-level device, but it is still touch-based and has some decent HTML capabilities (JavaScript and CSS support).

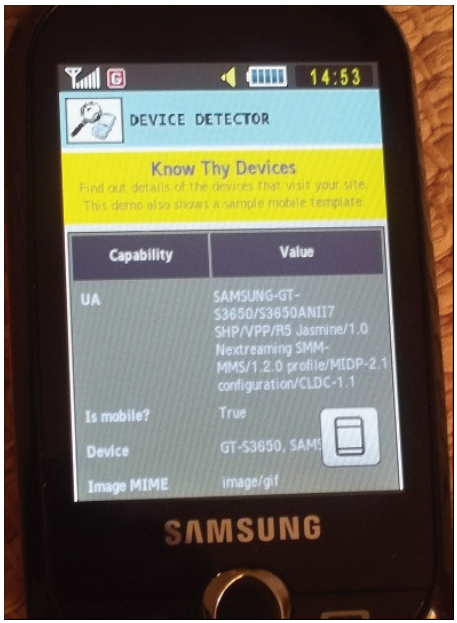



**FIGURE 4-18** The Device Detector site displayed on an Samsung Corby device.

Figures 4-17 and 4-18 don't show noticeable differences in the rendered markups. However, if you view it live, you see that the device of Figure 4-18—an older device—takes a while to render the page for at least a couple of reasons. First, it doesn't have much processing power, so any operation (JavaScript or just downloading) is slower. Second, it doesn't have WiFi support; so any download can occur only via 3G. The rendering experience is somewhat painful because it first displays the skeleton of the HTML; next, it downloads the CSS and applies colors; and finally, it gets the picture and inserts that into the layout. Tweaking HTML for older browsers is an operation that you might want to optimize on a per-page basis—if it is worth the cost.

## Summary

This chapter tried to turn into practical advice and bits and pieces of code some of the most common practices employed today to build mobile websites. The number of mobile devices is huge, but, on the other hand, you don't have to target all of them. Mobile development is about understanding what you and your customers want and need. This is a crucial point and results in an ideal selection of use-cases. Appropriate and well-described use-cases are essential for any application, but it is a bit more important for mobile applications and sites. For mobile sites, in particular, I estimate that it represents the largest share of the work.

Selection of use-cases helps to keep the whole application up-close-and-personal, establishing a relationship between code and user that is much stricter than with any other type of application.

Beyond this point, building a mobile site is a matter of optimization: minimizing requests, minimizing content being downloaded, and minimizing the user's activity. But it's also a matter of optimizing the design to find a good compromise between UI widgets and touch capabilities. The user interface is critical: common widgets like drop-down lists and check boxes, while valid to express the behavior of a view, may require a different rendering and graphical structure.

Finally, a mobile site may largely reuse code in the back end of the sister desktop site. Coding the back end is probably the simplest aspect of mobile site development. The next chapter covers in more detail two technologies that have been mentioned only briefly up to now: jQuery Mobile and HTML5.

CHAPTER 5

# HTML5 and jQuery Mobile

*I always like to look on the optimistic side of life, but I am realistic enough to know that life is a complex matter.*

*— Walt Disney*

**In this chapter:**

- jQuery Mobile Fast Facts
- HTML5 Fast Facts
- Summary

So far, this book has covered the foundation of mobile site development—a single-column layout, different views, a viewport, browser detection, and the role (and understanding) of capabilities. With these tools, you build basic applications that may be fully functional, but they aren't necessarily compelling to users.

Mobile users have high expectations in terms of user experience; they expect applications to provide an overall user interface similar to that of smartphones (e.g., iPhone): touch-based and populated by common widgets such as pick-lists and the iPhone-ish toggle switch. These widgets don't exist (yet?) as native elements of HTML and must be simulated using server-side controls that output a mix of JavaScript and markup every time. The bottom line is that creating a plain site making the best possible use of HTML, Cascading Style Sheets (CSS), and JavaScript is one thing; it is quite another to make a compelling mobile site that looks like a native application or, at the very least, behaves like one.

This chapter will introduce two alternate approaches to mobile site development. One is based on the jQuery Mobile library; the other employs the full power of HTML5. Is it just a matter of preference? Well, it should be noted up front that while with jQuery Mobile, you can build one site that works on any browser (mobile and desktop) in at least some way, a full HTML5 solution works only on browsers that support it. The list of HTML5-compliant browsers includes the latest versions of Safari, Firefox, Chrome, Opera, Konqueror, and Windows Internet Explorer, and default mobile browsers available on recent versions of major mobile platforms (iPhone, Android, and Windows Phone 7.5). So HTML5 is more widespread than one may think and is definitely worth a look. Chapter 11, "Mobile Applications with PhoneGap," will return to discussing a stand-alone HTML5-based solution as a way to build cross-platform native applications through an intermediary tool like PhoneGap.

# jQuery Mobile Fast Facts

Working on top of the popular jQuery library, jQuery Mobile is built from the ground up to be a mostly comprehensive platform for building mobile sites. In this regard, jQuery Mobile is more intrusive than jQuery. In fact, jQuery Mobile implements Progressive Enhancement from the ground up. You code your way through the library, and the library takes care of rendering the markup in the best possible way on the browser. By using jQuery Mobile, you don't necessarily have to worry about device detection and capabilities. The library guarantees that the output works also on older browsers; whether the obtained output is really what you want—well, that's quite another story.

**Important** So are libraries like jQuery Mobile really the Holy Grail of mobile site development? Do these libraries have the power to free you from the need to detect browser capabilities carefully? Like CSS media queries, libraries can shield you from dealing with the nitty-gritty capabilities of specific devices; but the power of libraries ends with the rendering of the markup and perhaps with some general-purpose forms of optimization, such as minimizing Hypertext Transfer Protocol (HTTP) requests and compressing data. They can't help much if you just need to know about capabilities that can't be reliably tested via code such as inline images, screen resolution, and touch capabilities.

## Generalities of jQuery Mobile

The main purpose of the jQuery Mobile library is enabling cross-platform, cross-device, and cross-browser programming of mobile websites. It shares the same architecture and programming model of full jQuery. The biggest benefit of using JavaScript libraries like jQuery and jQuery Mobile is that you learn a unified application programming interface (API) and it works across a large number of browsers. Consider that a significant part of the jQuery codebase deals with browser compatibilities and performs tricks to ensure that users get nearly the same experience regardless of the browser.

### Setup of the Library

The official site of jQuery Mobile is *http://www.jquerymobile.com*. You can get the latest version of jQuery Mobile from *http://code.jquery.com*. This chapter is based on jQuery Mobile version 1.0.

The library is nearly unusable without a companion CSS file. The library does a lot of work on any page and transforms it from a plain collection of DIV tags into a usable, mobile-friendly document. For this to happen, you need a slew of styles and images that have been created following strict standards. The library comes with a predefined CSS file you can use in your websites. Like many other things, themes are customizable by developers, of course.

The following code shows the files to link into any application page or just into a master page. With reference to ASP.NET MVC, here's the structure of a layout file:

```
<!DOCTYPE html>
<html>
<head>
    <title>jQuery Mobile Demos</title>
    <meta name="viewport" content="width=device-width" />
   <link rel="stylesheet"
              href="@Url.Content("~/Content/Styles/jquery.mobile-1.0.css")" />
   <link rel="stylesheet"
              href="@Url.Content("~/Content/Styles/site.css")" />
   <script type="text/javascript"
               src="@Url.Content("~/Content/Scripts/jquery-1.6.4.min.js")" />
        <script type="text/javascript"
               src="@Url.Content("~/Content/Scripts/jquery.mobile-1.0.js")" />
</head>
<body>
    @RenderBody()
</body>
</html>
```

As you can see, you need to reference the jQuery Mobile CSS file (plus any other application-specific CSS file), as well as the regular jQuery library. The regular jQuery library must be referenced before the jQuery Mobile library.

## Graded Support Matrix

The jQuery Mobile library targets a variety of browsers and divides them into three main groups named A, B, and C. The partition is based on the capabilities that browsers provide with respect to the library's needs. Support for CSS media queries is a key capability in jQuery Mobile. Browsers without media query capabilities still can be able to display jQuery Mobile pages effectively, but the final user experience is much less pleasant.

Now let's find out more about the capabilities required for browsers in the various groups.

Top-level browsers are called *A-graded* browsers. They typically support media queries, JavaScript, Ajax, CSS, and full Document Object Model (DOM) manipulation. A-graded browsers are tested regularly and represent the group of browsers that the library is basically built against. There's no guarantee, however, that 100% of the library's features are supported on just any A-graded browser.

B-graded browsers are less powerful but still capable and powerful. The most common deficiency of B-graded browsers is that they lack Ajax. The library is not optimized for such browsers, so you might expect to be able to carry out operations successfully but with a less pleasant experience. The point is that B-graded browsers are less prioritized and subsequently have less opportunity to see workarounds implemented in the core library to fix any misbehavior. If any of these browsers is critical for you, you may even decide not to use jQuery Mobile or opt for more customized pages.

Finally, C-graded browsers are browsers that don't support media queries and are limited in terms of JavaScript, DOM manipulation, and CSS. When operating through a C-graded browser, a jQuery Mobile page falls back to plain HTML and basic CSS. No DOM modification is applied to page elements. For these browsers, you should plan an alternative set of pages to improve the user experience. The most updated matrix of browsers and grades is available at *http://jquerymobile.com/gbs*.

Figure 5-1 shows the same page on an A-graded and a C-graded browser.

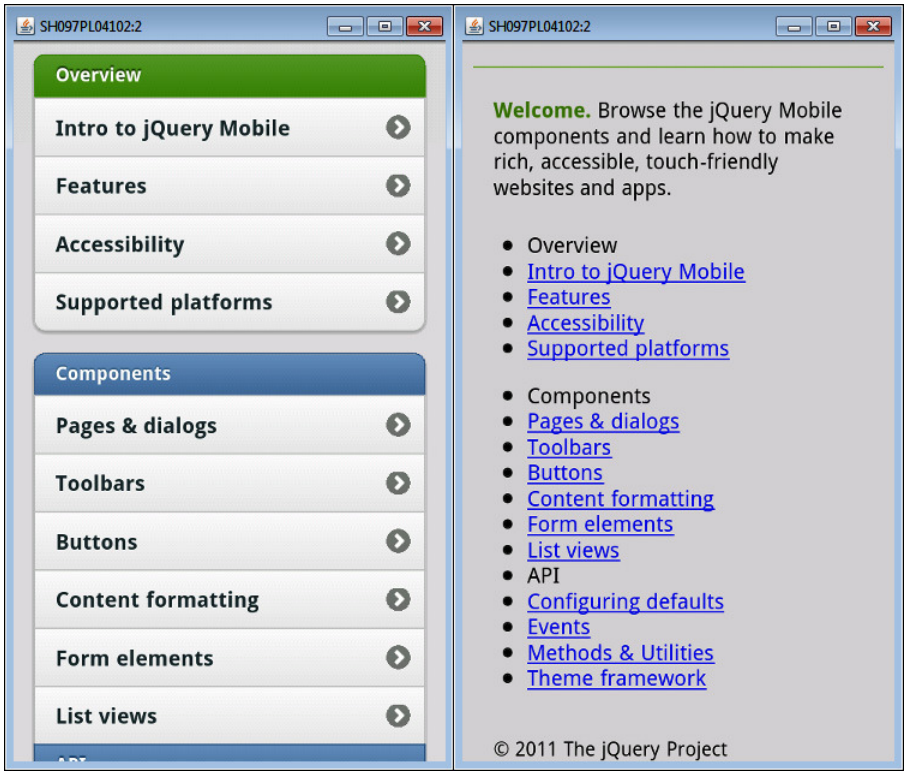



**FIGURE 5-1** Comparing A-graded (left) and C-graded browsers.

Improvements in A-graded browsers are not simply aesthetic: links are padded to make it easier for users to tap.

**Important** Note also that the plain HTML version of a page that you see on a C-graded browser may not always be as clean as in this example. To have a clear idea of what the output of a page on a C-graded browser will look like, just disable JavaScript on the browser. The bottom line is that jQuery Mobile is an excellent solution for A-graded browsers; for other browsers, it may need some work and probably different page templates.

## Themes and Styles

The jQuery Mobile library comes with a few predefined themes identified with the first letters of the alphabet: a, b, c, and so forth. Each theme consists in a number of CSS styles being uniformly applied to various HTML elements. Most of the time, you just pick up the theme you prefer, and the choice is based on colors. The library defaults to theme C.

You can apply a different theme to different parts of the page by using a *data-** attribute. (You will see more on *data-** attributes in a moment.) Themes are applied by default but can be overwritten using plain CSS commands on particular elements, as shown here with the *class* attribute:

```
<div data-role="footer" class="amse-footer center">
   ...
</div>
```

The preceding DIV element has been given the role of the footer; as such, it gets a particular style from the library depending on the current theme. However, the *class* attribute can be used to override some properties (e.g., colors, borders, and font). The *class* attribute will hardly replace settings completely; if that's your purpose, you are probably better off creating your own custom theme using the ThemeRoller tool of jQuery Mobile. (For more information about this tool, see *http://jquerymobile.com/themeroller*.)

### Data-* Attributes

In HTML5, *data-xxx* attributes are custom attributes that you can use to define the semantic of an element better. Such attributes are forced to have a name in the form of *data-xxx*, but any framework (or page) is responsible for specifying the *xxx* variable part, and, more important, for the interpretation of the attribute name. A *data-xxx* attribute always returns and accepts a string.

The jQuery Mobile library recognizes quite a few *data-xxx* attributes and uses them to decorate HTML elements and give them a special meaning. The semantic expressed by the attributes determines the graphical output.

One of the most important *data-** attribute in jQuery Mobile is the *data-role* attribute. It indicates the role played by that specific element in the context of the page. The attribute is commonly used to decorate DIV tags and make them pass as very semantic-specific components such as the header, the footer, and content.

**Note** HTML5 *data-** attributes are different from microformats. A microformat is still a special HTML markup, but it has a general scope and is not limited to the current application when not on the current page, as *data-** attributes are.

## Building Mobile Pages with jQuery Mobile

A mobile page is a relatively simple page often based on the single-column layout. In jQuery Mobile, a page results from the combination of header, footer, and content elements. However, the page itself doesn't necessarily have to be a separate file.

**Note** Any jQuery Mobile physical page starts with an HTML5-compatible *doctype* to alert HTML5-ready browsers to express their full potential. At the same time, older devices with non-HTML5 browsers will simply ignore the *doctype,* as well as ad hoc elements and *data-** attributes.

## What's a Page?

In jQuery Mobile, a page can either be a single HTML file or an internal section of an existing page. In the library jargon, the two scenarios are referred to as *single-page template* and *multipage template*. Nicely enough, the library allows you to navigate to pages regardless of their nature, whether they are physical pages (distinct HTML files) or logical sections of an existing HTML file. Here's the typical page definition:

```
<div id="homePage" data-role="page" data-theme="a" class="amse-bkgnd">
   ...
</div>
```

A page is identified by a DIV element decorated with the *data-role* attribute set to the *page* value. In the end, a jQuery Mobile page is mostly the root container of markup. The page can have its own ID and can use the *class* attribute to override style settings. You can select the theme for the entire page using the *data-theme* attribute. By default, the theme attribute accepts values like *a*, *b*, c, *d,* and *e*. You are entitled to add more themes, however. You can refer to the jQuery Mobile online documentation for the graphical details of these predefined themes.

A jQuery Mobile page is often made of a header, a footer, and content. It should be noted, however, that this is just a convention: the page can contain any valid markup. A HTML file can contain multiple elements marked as logical pages, such as the following:

```
<div id="homePage" data-role="page" data-theme="a" class="amse-bkgnd">
   ...
</div>
<div id="aboutPage" data-role="page" data-theme="b">
   ...
</div>
```

When multiple logical pages are used, you might want to use unique IDs for each of them, which enables navigation and initialization.

## Initializing Pages

Any jQuery developer knows the *ready* event very well. The event is fired to your code as soon as the page DOM is fully initialized, and the page author can complete the initialization of the page elements safely. In jQuery Mobile, you don't use the *ready* event; instead, you use the new *pageinit* event:

```
<div id="homePage" data-role="page" class="amse-bkgnd">
    <script type="text/javascript">
```

```
        $("#homePage").bind("pageinit", function () { alert("home");  });
    </script>
    ...
</div>
```

The difference is that in jQuery Mobile, Ajax is used to download requested pages and set up animations and page transitions silently. This means that the display of the page (whether a real page or a virtual page container) follows different rules than in classic jQuery.

In a nutshell, all you need to do is add a *<script>* tag within the page element and bind the DIV that represents the page with the *pageinit* event. That code is guaranteed to be invoked every time the page is loaded—whether following a link or by using an Ajax call. In *pageinit*, you typically register your jQuery plug-ins and do your initialization work (apply localized strings, set up controls, and the like). Similar events exist for page display and unload.

**Important** In mobile development, the *Back-and-Save* pattern (or AutoSave) is pretty popular in both native applications and mobile sites. The pattern suggests that you automatically save any data or selection the user makes on a form. In mobile applications, you need to capture the *return* event (as exposed by the framework you're using); in mobile sites, this is harder to do at the DOM level. The jQuery Mobile library makes it easy. Chapter 7, "Patterns of Mobile Application Development," will cover Back-and-Save and the other mobile application development patterns in more detail.

## Headers and Footers with jQuery Mobile

Two commonly used values for the *data-role* attribute are *header* and *footer*. The header bar contains the page title and a couple of optional buttons to the left and right (mimicking the iPhone template). The header role receives a special style by the framework and undergoes some default manipulation. As a developer, you can customize both the header template and the text and target of the buttons completely. Here's a sample header bar:

```
<div data-role="header">
   <h1>Demos</h1>
</div>
```

In particular, the first heading element (*H1*–*H6*) is used to title the bar; if the content is not empty, that text also becomes the page title, overriding any value assigned to the *<title>* element. The heading element that you use to title the header bar doesn't matter so long as it is an *Hx* element—the style applied is the same. If you want to give the page its own title distinct from the header text, you use the *data-title* attribute on the page container. The first link found in the header bar is automatically styled as a button and moved to the left. The second link, instead, is placed to the right. If you want just one button to the right, you add an extra *class* attribute, as shown here:

```
<div data-role="header">
   <h1> Settings </h1>
   <a href="..." data-icon="back">Back</a>
   <a href="..." data-icon="gear" class="ui-btn-right">Save</a>
</div>
```

The data-icon attribute selects one of the predefined icons in the jQuery Mobile themes; the *ui-btn-right* value moves the button to the right (in the example, it is not strictly needed because the button to the right is the second one). It is interesting to note that if you use ASP.NET MVC and HTML helpers, the preceding anchors must be rewritten as shown here:

```
@Html.ActionLink("Back", "index", "home", null, new {data_icon = "back"})
@Html.ActionLink("Save", "save", "home", null, new {data_icon="gear", @class="ui-btn-right"})
```

ASP.NET MVC will expand the underscore (_) to the dash (-) symbol automatically, and the @ symbol escapes the word *class*, which otherwise would have another meaning to the Razor compiler.

Unlike the header, the footer is not designed to simplify a given markup template. However, any link that you place in the footer area is turned into a button automatically and any markup, including form markup, is acceptable in the footer. In this way, you can place an application bar in the footer or even a drop-down list to let the user choose, say, the language. Note that jQuery Mobile manages the actual position of the footer bar. By using the *data-position* attribute, you can keep the position constant to the bottom of the page:

```
<div data-role="footer" data-position="fixed" data-id="about-us">
   ...
</div>
```

When the user follows links between pages, jQuery Mobile applies transitions to the entire page. Sometimes, you get a smoother effect by keeping the footer still. For this to happen, the footer must be fixed on both pages making the transition; and in addition, both footers must have the *data-id* attribute set to the same (unique) identifier. Figure 5-2 shows a sample page with a header, footer, and placeholder for the body.


**FIGURE 5-2** The default page template in jQuery Mobile.

To customize the header template entirely, you add a child DIV to the header block and populate it at will. Note that in this case, you lose automatic manipulation; for example, links display as plain links. You need to use the *data-role=button* attribute to make the transformation:

```
<div data-role="header">
    <div>
       <img src="@Url.Content("~/Content/Images/mobile.png")" alt="">
       <h3>Custom templates</h3>
       @Html.ActionLink("Back", "index", "home", null,
                 new {data_icon = "back", data_role="button", @class="ui-btn-right"})
     </div>
</div>
```

Figure 5-3 shows a custom header template. Note that you also add CSS gradients to the background.


**FIGURE 5-3** A custom header template.

## Lists, Lists, and Lists

As stated repeatedly up to now, the home page of a mobile site should simply provide a list of actions like the classic main menu of applications of 30 years ago. The items of the menu can be rendered in a number of ways—they can form a navigation bar, a button bar, a collection of tiles, or a plain list of good-looking links. jQuery Mobile supports nearly any scenario, but in particular, it works great with navigation bar and list views.

You enable a navigation bar by using the *navbar* role on a DIV element and adding buttons through a *UL* collection. Here's an example:

```
<div data-role="navbar">
   <ul>
     <li><a href="..." class="ui-btn-active">Results</a></li>
     <li><a href="...">Draws</a></li>
     <li><a href="...">Live scores</a></li>
   </ul>
</div>
```

This code produces the following graphic:


Conveniently enough, you can add style information to define a fixed or variable width for elements. If not, jQuery Mobile will split the width evenly between the *LI* elements.

The *listview* role provides the most common user interface on mobile devices these days—very close to iPhone and Android pick-lists. The following plain HTML code processed by jQuery Mobile produces the output shown in Figure 5-4:

```
<div data-role="content">
    <div id="mainmenu">
       <ul data-role="listview" data-inset="true" data-theme="d" data-dividertheme="e">
            <li data-role="list-divider">Content & formatting</li>
            <li><a href="@Url.Content("~/demo/customtemplate")"> Custom templates</a></li>
            <li><a href="#"> Example #2</a></li>
            <li><a href="#"> Example #3</a></li>
            <li data-role="list-divider">Forms</li>
            <li>One more</li>
            <li><a href="#"> Yet another</a></li>
       </ul>
   </div>
</div>
```

By using variations of the *UL* and *OL* elements, you can create numbered lists and nested lists. Graphical variations include the *data-inset* attribute responsible for the rounded corners and nice scaffolding of Figure 5-4 and *data-filter* to add a search bar with automatic auto-completion of the list items statically added to the page. (This auto-completion doesn't entail connecting to a remote service to download dynamic data.)

You can add icons and images to the list items. Here's the code that you use to add a small icon to the left of the list item:

```
<li><a href="..."> <img alt="" class="ui-li-icon" sr="icon.png" />Yet another demo</a></li>
```

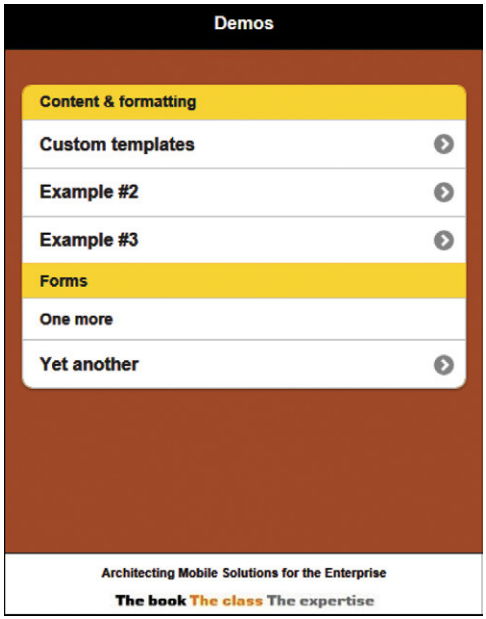



**FIGURE 5-4** A jQuery Mobile list view.

The *ui-li-icon* class ensures that the image is left-aligned; note that the image element must be a child of the anchor. Similarly, you can add text to the right of the list item. If you want the text to be rendered automatically within a bubble, you mark the text with the *ui-li-count* class; otherwise, use the *ui-li-aside* class. Here's the markup that generates Figure 5-5:

```
<ul data-role="listview">
   <li data-role="list-divider">Content & formatting</li>
   <li><a href="...">Custom templates</a>
       <span class='ui-li-count'>!!!</span></li>
   <li> <a href="..."> Example #2 </a> </li>
   <li> <a href="..."> Example #3 </a> </li>
   <li data-role="list-divider">Forms</li>
   <li>Basic demo <span class="ui-li-aside"><small>Need fix!</small></span></li>
   <li><a href="..."> <img alt="" class="ui-li-icon" sr="icon.png" />Yet another demo</a></li>
</ul>
```

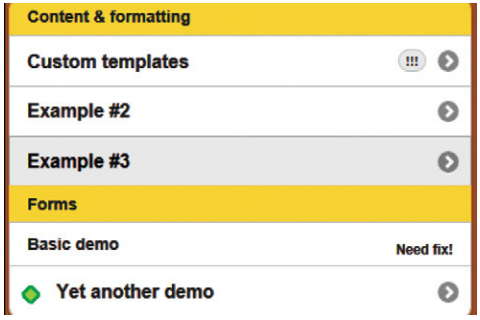



**FIGURE 5-5** List views created with jQuery Mobile.

## Fluid Layout

Although having multiple columns in the layout is hardly a great idea for a mobile site, having an easy way to place a few elements side by side would be a great feature. In jQuery Mobile, you find a very basic but effective implementation of CSS grids and fluid layouts that automatically adapt to the screen.

The library makes available two main CSS classes—*ui-grid* and *ui-block*. The former marks a DIV (or a FIELDSET) as the container grid, while the latter marks a DIV as a child element. The main CSS classes, however, need a progressive letter to indicate the number of cells and the position, respectively. The *ui-grid-a* class, for example, split each row evenly into two blocks. The first is assigned to the content referenced by *ui-block-a*; the second goes to *ui-block-b*. Here's an example:

```
<fieldset class="ui-grid-a">
   <div class="ui-block-a"> First block </div>
   <div class="ui-block-b"> Second block </div>
   <div class="ui-block-c"> Third block </div>
</fieldset>
```

In this case, there is also a third cell assigned to a grid that can't host more than two cells (it splits the row width equally in half). The net effect is that the third DIV, styled as *ui-block-c,* wraps to a second row. To keep the three child blocks on the same row, you change the grid style to *ui-grid-b*, which splits into three equal parts (see Figure 5-6). Likewise, *ui-grid-c* splits into four parts and *ui-grid-d* splits into five parts.

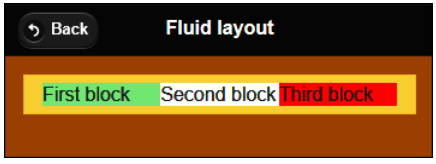



**FIGURE 5-6** Fluid layout with jQuery Mobile.

## Collapsible Panels

Collapsible panels are easy too, as the following code snippet shows:

```
<div data-role="collapsible" data-theme="a" data-collapsed="true" data-content-theme="e">
     <h3>Drill down</h3>
     <div>
        This text was initially kept hidden from view.
     </div>
</div>
```

Assigned to a container element, the *collapsible* role uses the first *Hn* child element to title the panel and makes everything else collapsible. You can use *data-collapsed* to decide whether the content is initially hidden or not. You use *data-content-theme* to style the content of the panel and *data-theme* to style the header (see Figure 5-7).

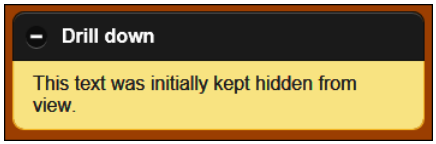
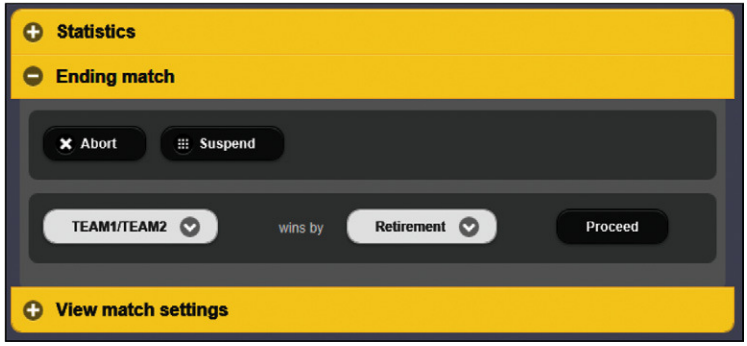



**FIGURE 5-7** Collapsible panels with jQuery Mobile.

By simply surrounding an array of collapsible panels with an outermost container, you can obtain an accordion widget, as shown here:

```
<div data-role="collapsible-set">
    <div data-role="collapsible">
      ...
    </div>
    <div data-role="collapsible">
      ...
    </div>
     ...
</div>
```

This code produces the following graphic:




By placing style attributes on the parent container, all panels are styled the same. Otherwise, you can style each panel individually.

## Working with Pages

A jQuery Mobile application is still a site, and it is made of pages or views if you use ASP.NET MVC. The dynamics of the site are mostly expressed by linking pages and having the user jump from one page to the next.

### Page Links and Transitions

As mentioned previously, in jQuery Mobile, a page is preferably designed to use Ajax. In a way, jQuery Mobile realizes the old dream of having Ajax on the site without notice and without having to use a special API to enable it. In jQuery Mobile, you use links—that's it. Under the hood, the library intercepts the call and makes it an Ajax operation if the browser supports Ajax.

Using Ajax under the covers enables the library to manipulate the DOM and to perform transitions, thus providing an experience similar to that of native applications. (The effect, however, is hardly the same because of the latency of the "real" network.) You can control the transition by using the *data-transition* attribute on the link to a page. Feasible values include *slide* (default), *pop, fade,* and *slideup*. Here's an example:

```
<a href="index.html" data-transition="pop">Home</a>
```

You also can programmatically change page, as shown here:

```
$.mobile.changePage( "about.html", { transition: "slideup"} );
```

Note that the effect is not the same on any device and test carefully. You also can set the transition for when you're back from the navigated pack. You use the *data-direction="reverse"* attribute on the link. The type of transition performed when you move back is the same.

Once downloaded, a page is added to the current DOM. Note that if you have a single-page template, then all your logical pages are being held in the same DOM. This may have two effects—memory occupation and ID conflicts. It is recommended that you implement a strict naming convention to prevent ID belonging to different pages to be the same—there's no real memory separation when the single-page template is used.

The jQuery Mobile does its best to keep memory occupation to a minimum and attempts to unload portions of the DOM of a page when the page is no longer in the foreground. When the user returns to the page, the markup is either retrieved from the browser cache or requested again. As a developer, you can change this practice by instructing jQuery Mobile to cache the DOM of a given page. Here's how to proceed:

```
<div data-role="page" data-dom-cache="true">
   ...
</div>
```

Be careful, though, as caching a large DOM may affect overall performance.

## Disabling Caching on Ajax Calls

In addition, you should note that Ajax calls are made through the regular jQuery library. When this happens, some caching occurs for performance reasons. This means that the results of some Ajax calls (including page transitions) may not be what you expect. If this is the case, you have just one option: disabling the caching feature on Ajax calls:

```
$.ajaxSetup({ cache: false });
```

You need to call the *ajaxSetup* function only once. You do that at application startup, or just before the first time you execute the Ajax operation whose results you don't want to cache.

Ajax is not used to follow page links in all cases. You can set the *data-ajax* attribute to *false* on the link element and have the jQuery Mobile library follow the link in a browser-led way and without transitions. The Ajax navigation system is also bypassed when the link specifies a nonempty *target* attribute or has the *rel* attribute set to *external*:

```
<a href="..." rel="external" />
```

The effect of using *rel* or *data-ajax* is the same, but it is recommended that you use *rel* when linking outside the current site.

**Note** When an Ajax call fails and returns a 400 or 500 status code, the jQuery Mobile library reacts to the exception by displaying an alert box. In terms of user interface, that's highly desirable, but it leaves it up to you to figure out what's happened.

## Dialog Boxes

In Ajax programming, displaying modal message boxes has never been particularly easy. In jQuery Mobile, it is insanely simple and especially natural. All you have to do is define a link to a page and state that you want that page to be opened in a dialog box, as follows:

```
<a href="@Url.Content("~/demo/dialog")" data-rel="dialog"> Dialog box </a>
```

In addition to the *data-rel* attribute, you might want to indicate a *data-transition* attribute to specify the transition effect that you want. By default, a Close button is added to the dialog box that dismisses the dialog box without taking any further action except returning to the previous page. Here's the code for the sample dialog box shown in Figure 5-8:

```
<div id="demoPage1" data-role="page" class="amse-bkgnd">
    <div data-role="header">
        <h3>Dialog box</h3>
    </div>

    <div data-role="content" data-theme="b">
        Look I'm a regular page but displayed as a dialog box!
        Do you like this feature?
        <a href="..." data-role="button">Yes</a>
        <a href="..." data-role="button">No</a>
        <a href="..." data-rel="back" data-role="button">Cancel</a>
    </div>
</div>
```

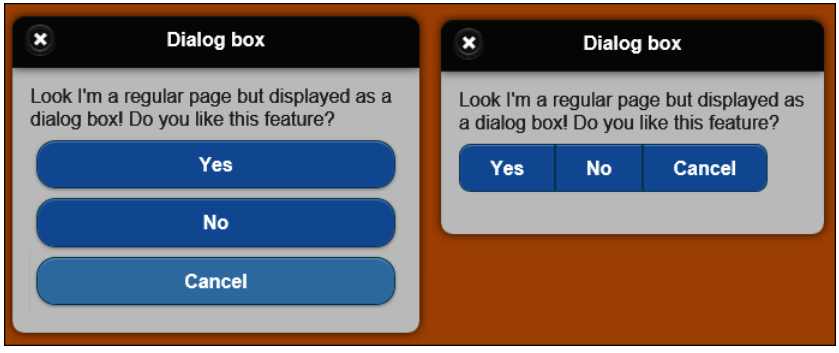



**FIGURE 5-8** A jQuery Mobile dialog box.

The caption of the dialog box is inferred by the first *Hn* element found on the page. Any link followed within the page closes the dialog box. To return to the previous page, you can click the Close button or any other button that you may have around the page that returns to the original page. You can obtain this functionality with the following code:

```
<a href="..." data-rel="back" data-role="button">Cancel</a>
```

Alternatively, you can close the dialog box programmatically by using the following JavaScript code:

```
$(".ui-dialog").dialog("close");
```

To display the three buttons in Figure 5-8 horizontally, you either use a grid or group the buttons together, as shown here:

```
<div data-role="controlgroup" data-type="horizontal">
     <a href="..." data-role="button">Yes</a>
     <a href="..." data-role="button">No</a>
     <a href="..." data-role="button">Cancel</a>
</div>
```

Finally, you can use themes to style dialog boxes in much the same way you do for regular pages.

## Input Forms

In jQuery Mobile, you treat forms as usual. A form posts to its action URL using the specified HTTP verb. Forms are managed internally via Ajax. Forms exist to collect data from users and are full of input elements. The jQuery Mobile recognizes the new input fields introduced with HTML5—*email, date, range, search, number,* and *phone,* to name just a few. The library, however, attempts to provide a unified experience across a number of browsers. This means that jQuery Mobile transforms each input into plain text and then builds some DOM, CSS, and JavaScript magic to improve the input experience through date pickers, sliders, and controlled input.

In addition, you have handy ready-to-use search text boxes, drop-down lists, and toggle-switch controls. Here's an example of the markup that you need to arrange a toggle switch and a slider (see Figure 5-9):

```
<div data-role="fieldcontain">
    <label for="slider">How is your day?</label>
    <input type="range" name="slider" id="slider" value="3" min="1" max="9"  />
</div>

<div data-role="fieldcontain">
    <label for="slider">Are you ready?</label>
    <select name="readiness" id="readiness" data-role="slider">
       <option value="no">No</option>
       <option value="yes">Yes</option>
    </select>
</div>
```

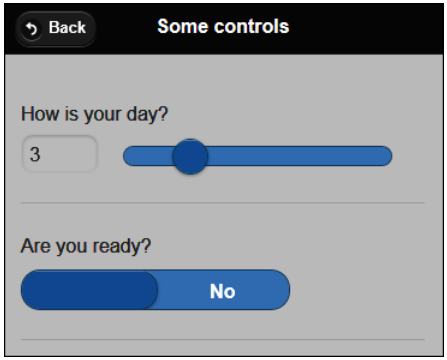
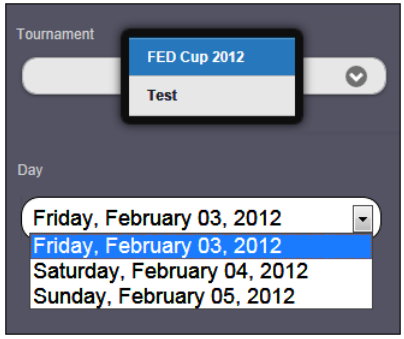



**FIGURE 5-9** A toggle-switch control.

Finally, it should be noted that for drop-down lists, you can choose between the native interface of the browser or a jQuery Mobile manipulated standard user interface that mimics that of iPhone and Android. To get the script-driven user interface, similar to native mobile operating systems, you add the *data-native-menu* attribute and set it to *false*:

```
<select name="foo" id="foo" data-native-menu="false">
   <option value="a" >One</option>
   <option value="b" >Two</option>
   <option value="c" >Three</option>
</select>
```

The following screenshot illustrates the effect of having a *SELECT* element with the *data-native-menu* attribute set to *true* and *false*, respectively.




## HTML5 Fast Facts

The HTML language has been around for about 20 years now and has gone through two main ages—the age of *static* rendering and the age of *dynamic* rendering. Static rendering was prevalent in the infancy of the web, when HTML was used only to render the content of hypertext documents with minimal templating, a few related images, and mostly links to other documents. The ratio between content and layout was approximately 90/10.

Dynamic rendering came a few years later, building on the rapid adoption of the web medium. Along the way, the ratio between content and layout slowly but steadily moved toward 50/50. An important change occurred in the late 1990s, when browser vendors made the representation of the displayed document modifiable via JavaScript. That was the early days of the web, and it added an entire new perspective to markup. HTML was then perceived as the language to provide a dynamic representation of the content of a webpage; it was more of an application delivery format than a plain document format.

HTML5 marks the beginning of the third age of the web, in which HTML advances at a brisk pace toward becoming a true and full-fledged application delivery format. HTML5 is not limited to presentation; it also provides a slew of new functionalities for web and other types of applications. The big change is that HTML5 is about client-side programming and about building applications that can run within the browser with limited (or no) interaction with the back end.

What's the purpose of an HTML5 executive summary in a mobile book? One reason is certainly the central role that HTML5 plays in mobile development. Chapter 11, "Mobile Applications with PhoneGap," will demonstrate how to use PhoneGap to package a client HTML5 solution as a native mobile application for a variety of platforms. In addition, if your focus is more on the web side of mobile development, you should know that mobile browsers (and particularly Safari for iOS) offer an excellent support for HTML5. I'd even say that in pages that target smartphones, you can use HTML5 without worry. The support is not uniform, of course, but key HTML5 features are easier to find on mobile browsers than desktop ones—at least, that's the case at the moment.

Let's find out more now.

## Semantic Markup

Compared to its predecessor (which was defined more than a decade ago), HTML5 is a significantly richer markup language. One could say that all these years have not passed in vain, as HTML5 now incorporates in its standard syntax many common practices that developers and designers employed in thousands of websites. In doing so, HTML5 issues specific rules on how to structure HTML elements and deprecates tags that were introduced in the past to style elements. The new message is: use CSS to style elements and use specific, newly introduced tags to define the structure of the document.

### Headers and Footers with HTML5

Nearly any website out there shares a common layout that includes headers and footers, as well as a navigation bar on the left side of the page. More often than not, these results are achieved by using DIV elements styled to align to the left or the right. Most pages today end up with the following template, which also was used in the samples of this book:

```
<div id="page">
   <div id="header">
      ...
   </div>
   <div id="navbar">
      <ul>
```

```
        <li> ... </li>
        <li> ... </li>
        <li> ... </li>
     </ul>
   </div>
      <div id="container">
        <div id="left-sidebar">
          ...
          <ul>
             <li> ... </li>
             <li> ... </li>
             <li> ... </li>
           </ul>
        </div>
        <div id="content">
          ...
        </div>
        <div id="right-sidebar">
          ...
        </div>
   </div>
   <div id="footer">
       ...
   </div>
</div>
```

The template includes a header, navigation bar, and footer, with a three-column layout. The preceding markup alone, however, doesn't produce the expected results. For that, you need to add ad hoc CSS styles to individual DIV elements and make them float and anchor to the left or right edge.

What's different in HTML5?

First, using HTML5 doesn't mean that you stop using CSS to transform a layout into a good-looking page. You still need to use the same bit of CSS to make the page look compelling and place segments where they belong. However, now you can describe the page in a way that is much cleaner and also easier for a designer working on a cascading style sheet to read. Essentially, with HTML5, you replace generic DIV elements with more semantically meaningful elements such as *HEADER*, *FOOTER,* and *ARTICLE*. Here's how you can rewrite the preceding template with the newest HTML tags:

```
<header> ... </header>
<nav> ... </nav>
<article>
   <aside>
      ...
   </aside>
   <section> ... </section>
   <section> ... </section>
   <section> ... </section>
   <aside>
      ...
   </aside>
</article>
<footer> ... </footer>
```

The *NAV* element logically groups links that would go on a navigation bar. The *ARTICLE* element represents the container of any content for the page and incorporates *ASIDE* elements and *SECTION* elements.

All of these are block elements and must be styled properly to form a presentable page. Other new elements complete the list of enhancements, such as *FIGURE* and *DETAILS*. The *FIGURE* element is designed to include figures with captions, whereas *DETAILS* replaces the canonical hidden DIV element that developers use to hide optional content and display it via JavaScript when desired.

## A Native Collapsible Element

The new *DETAILS* element is functionally equivalent to a DIV, but its internal content is interpreted by the browser and used to implement a collapsible panel. Here's an example of this new element:

```
<details open="true">
   <summary>Drill down</summary>
   <div id="details_inside">
         This text was initially kept hidden from view
   </div>
 </details>
```

The *open* attribute indicates whether or not you want the content to be displayed initially. The *SUMMARY* element indicates the text for the clickable placeholder, whereas the remaining content is hidden or shown on demand. At this time, Google Chrome is the only browser that supports this feature (see Figure 5-10).

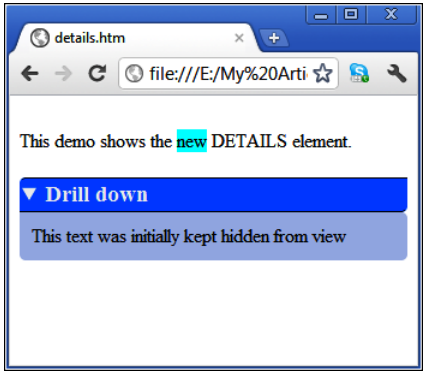



**FIGURE 5-10** The *DETAILS* element in action in Google Chrome.

The icon that you see in Figure 5-10 next to the "Drill down" caption is provided by the browser. The *DETAILS* element requires a bit of CSS to look nice. Here's the CSS used for the element in Figure 5-10:

```
<style>
   summary {
      border-top: solid 1px #000;
      border-bottom: solid 1px #000;
      font-size: 120%;
      font-weight: bold;
```

```
      background: #0055ff;
      color: #efefef;
      padding:2px;
      border-top-left-radius: 5px;
      border-bottom-right-radius: 5px;
   }
   #details_inside  {
      padding: 10px;
      background: #aabbe8;
      border-bottom-left-radius: 5px;
      border-bottom-right-radius: 5px;
      border-top-right-radius: 5px;
   }
</style>
```

It is key to note that fully HTML5-compliant browsers will provide drill-down capabilities for free. Likewise, it is interesting to remark the close similarity between the *DETAILS* element and the collapsible role in jQuery Mobile that was discussed in the "Collapsible Panels" section, earlier in this chapter. The jQuery Mobile provides excellent polyfills (i.e., replacements) for many upcoming HTML5 features.

## Adjusting HTML5 Pages for Older Browsers

The main problem with HTML5 is that only the latest versions of most popular browsers currently support it. So there are still quite a few users out there who won't be able to see your HTML5 page properly. Needless to say, the ideal solution is serving ad hoc pages to different browsers. However, in addition to the extra work of creating and maintaining multiple pages, you have the problem of detecting the browser capabilities on the server. Some websites such as *http://html5test.com/* will analyze browsers and return a list of capabilities for you to arrange different pages.

A library like Modernizr (*http://www.modernizr.com*) will help create HTML5-ready pages by letting you write plain HTML5 pages and adapting the DOM to the requesting browser via script. You can learn more about how Modernizr works at *http://www.modernizr.com/docs*.

Finally, as far as section elements are concerned, you can use a simple but effective trick. You place a child DIV element in each of the new HTML5 block elements:

```
<section>
  <div id="section">
     ...
  </div>
</section>
```

The CSS class ensures that you style the element properly on older browsers to match the HTML semantic. A browser that doesn't recognize an element typically does one of the following: it ignores the element or generates a DOM node of an unknown type. By having a child DIV that provides the expected behavior regardless of the actual browser recovery strategy, the rendering is consistent. At most, you get two nested DIV blocks that do the same thing.

### Missing Elements in HTML5

As mentioned previously, HTML5 removes a few elements of little use whose presence would only increase redundancy. The list of elements that are no longer supported most notably includes frame and font elements. (The *IFRAME* element remains, however.)

In addition, a few style elements such as *CENTER*, *U,* and *BIG* are removed. The reason is that this functionality can be achieved easily through CSS. For some reason, the current draft maintains analogous elements such as *B* and *I*. HTML5 also adds a new element—*MARK*—to highlight small portions of text, as shown here:

```
mark {
   background: cyan;
}
...
<p>This demo shows the <mark>new</mark> DETAILS element.</p>
```

Most HTML5 browsers default to yellow as the highlight color; the background color, however, can be styled easily via CSS. Figure 5-10 also demonstrates the *MARK* element.

## Web Forms and Data Entry

Currently, HTML doesn't support anything but plain text as input. There's quite a bit of difference between dates, numbers, or even email addresses, not to mention predefined values. Today, developers are responsible to prevent users from typing unwanted characters and for client validation of the entered text. The jQuery library has several plug-ins that simplify the task, but this just reinforces my point—input is a delicate matter.

### New Input Types

HTML5 comes with a bunch of new values for the attribute type of the *INPUT* element. In addition, the INPUT counts several new attributes that are mostly related to these new input types. Here are a few examples:

```
<input type="date" />
<input type="time" />
<input type="range" />
<input type="number" />
<input type="search" />
<input type="color" />
<input type="email" />
<input type="url" />
<input type="tel" />
```

What's the real effect of these new input types? The intended effect—though it's not completely standardized yet—is that browsers provide an ad hoc user interface so that users can enter a date, time, or number comfortably. Some of these new input types specifically address the need of mobile site users. In particular, the *email*, *url,* and *tel* types push mobile browsers on smartphones (e.g., iPhone, Android 2.0, Windows Phone 7.5) to adjust the input scope of the keyboard automatically. Figure 5-11 shows the effect of typing in a *tel* input field on an iPhone: the keyboard defaults to numbers and phone-related symbols.


**FIGURE 5-11** The *tel* input field on Safari for iPhone, which is similar on Android and Windows Phone.

Today, not all browsers provide the same experience, and while most of them agree about the user interface associated with the various input types, some key differences exist that may motivate developers to add script-based custom polyfills. As an example, let's consider the *date* type. As of today, Opera 11 is the only browser that edits a date by automatically opening a calendar. Other browsers (e.g., Chrome) provide buttons to navigate day by day (see Figure 5-12).

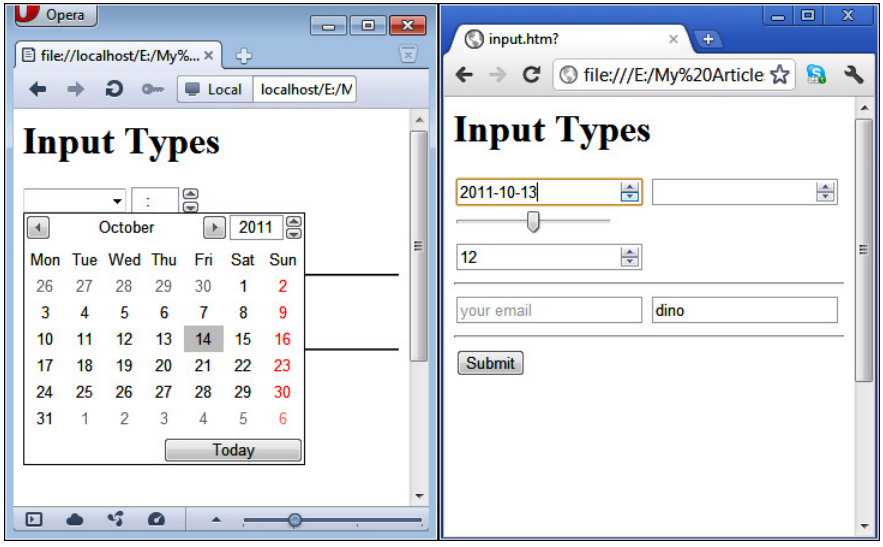



**FIGURE 5-12** The date input field as currently implemented by Opera 11 and Chrome 10.

The previous chapter discussed how the iPhone browser—which so far is unique in the mobile browser category—intelligently shows a date picker for a *date* input field. In general, mobile browsers on recent smartphones are quite accommodating to HTML5 elements. As a mobile site developer, therefore, you might want to be very careful when using new input elements for *email*, *number*, *url*, *date*, *tel*—well, just for everything!

Finally, it's worth noting the *placeholder* attribute, which implements the long-awaited ability to display a hint in a text box wherever possible. You won't get any hint text displayed in range and date/time input fields.

**Note** I decided not to put a screenshot for each input type for the sole reason that the user interface that browsers serve is constantly changing and may be quite different by the time you read this.

## Validation

Not just developers want to prevent users from entering meaningful values; they also want to make sure that what the user enters matches expectations. You can prevent users from typing letters where numbers are expected, but there's no way to prevent users from entering something into a text field that's not, say, an email address.

The current HTML5 draft, however, doesn't strictly mandate that browsers apply a sanitization algorithm on individual input fields. The result is that current browsers behave differently. Opera, for example, is quite strict in enforcing that users enter only valid values. For example, Opera stops users from entering letters into a *time* field, whereas Chrome doesn't block typing but clears the field upon exit if the content is invalid. In general, there's no guarantee of automatic sanitization of input as the user leaves the field. Instead there is an optional validation step performed when the form that that input field belongs to is submitted. This validation step is applied by default but can be disabled programmatically through the new *novalidate* attribute on the *FORM* element.

The draft defines the *required* attribute on the *INPUT* element to mark an input field as mandatory, as follows:

```
<input type="number" placeholder="length" required />
```

In addition, you have the *pattern* attribute, through which you specify a regular expression that the actual content must comply with. The following example shows a text field that allows users to type any free text but requires validation against numbers:

```
<input type="text" placeholder="only digits" pattern="[0-9]" />
```

When validation fails, both Opera and Google provide visual feedback, as shown in Figure 5-13.

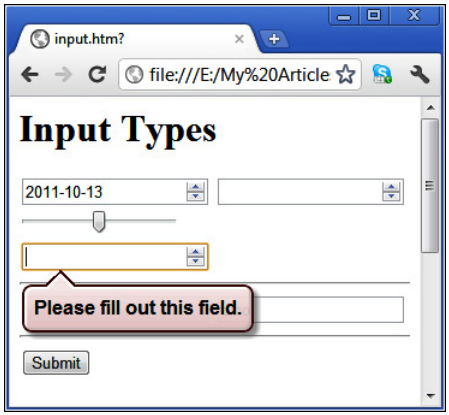



**FIGURE 5-13** HTML5 validation in action in Google Chrome.

## Predefined Entries

Another great improvement in HTML5 forms is the *DATALIST* element. The element is a specialized version of the popular *SELECT* element. It provides the same behavior, except that the drop-down list applies to a text input field. Here's an example:

```
<input list="countries" />
<datalist id="countries">
   <option value="Italy">
   <option value="Sweden">
   <option value="Denmark">
</datalist>
```

**Important** In the end, until browsers fully and uniformly support the new input fields, developers have quite a bit of work to do with JavaScript polyfills to ensure that correct data is posted to the server and that the users are informed properly about what is wrong. Many influential experts in this field today encourage people to always use HTML5 input fields, even on older browsers, saying that doing so won't break existing pages—it will only make them ready for the future and make the underlying markup more readable. This is absolutely correct; however, the challenge with input fields (and not necessarily with the entire HTML5 stack) is that it's more problematic with browsers that recognize new input types—they currently don't offer a consistent experience. It's not by chance that the jQuery Mobile people downgrade internally all input fields to *text* and then implement the HTML5 experience by hand. Unless you're using jQuery Mobile (it also works on desktop browsers), that responsibility will be on your own shoulders.

The net effect is that when the input field gets the focus, a menu displays, and the user either can enter free text or pick up one of the predefined options. At present, only Opera 11 supports this feature (see Figure 5-14).

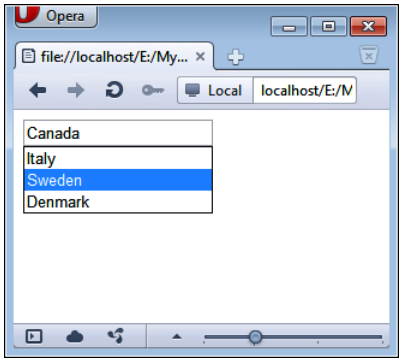



**FIGURE 5-14** The *DATALIST* element in action.

## Programmer-Friendly Features

We all know that websites need an active Internet connection to work. Whether Ajax- or browser-led, any requests that the user generates by interacting with the site is delivered at some remote endpoint and downloads some response—typically, a webpage or auxiliary resources such as images, scripts, and styles.

All browsers have some caching mechanism specifically introduced to save a few HTTP requests by resolving static resources from the local computer. This mechanism has never been standardized. Browsers use it today more as a form of internal optimization than as a way to enable users to work on the site while offline.

HTML5 comes with a couple of separate specifications for Data Storage and Offline Applications. All together, these APIs provide a framework for developers to build web applications that not only force the browser to cache resources according to a public manifest, but also enable the local code to save application data in a custom format and following application-specific rules. Looking at this from a distant perspective, one could say that data storage is a largely enhanced version of cookies, whereas offline working is a largely enhanced version of the classic browser cache.

### Local Storage

HTML5 provides a standard API that makes saving data on the user's machine more affordable and represents an effective replacement for cookies. The average size of local storage is around 5 MB—much more than a cookie.

You access the local storage through the *localStorage* property exposed by the browser's *window* object. The *localStorage* property offers a dictionary-based programming interface similar to that of cookies. You have methods to add and remove items, to count the number of items in the store, to get the value of a particular item, and to empty the store. Here's how you can save a value and retrieve it later:

```
<script type="text/javascript">
function save() {
   window.localStorage["message"] = "hello";
```

```
}
function init() {
   document.getElementById("message").innerHTML = window.localStorage["message"];
}
</script>
```

Upon loading, the page retrieves and displays data from the local storage (if any). Data is saved to the storage through a function invoked interactively. Data saved to the local storage remains on the user's machine indefinitely unless you programmatically empty it. The storage is specific for the application.

In addition to *localStorage*, HTML5 provides a *sessionStorage* object with the same programming interface but saves to the browser's memory instead. The *sessionStorage* object is emptied at the end of the current browser session. Web storage accepts primitive types (the spec is not restrictive on this point, so any JavaScript type is acceptable), but complex objects can be saved as well if serialized to the JavaScript Object Notation (JSON) format.

**More Info** Local storage is described separately from HTML5; full details about it (the specification is named *Web Data Storage*) can be found at *http://www.w3.org/tr/webstorage*.

## Offline Applications

Up until HTML5, the browser had total control over caching. HTML5 defines a public interface for pages to instruct the browser about resource caching. Developers use the manifest cache special resource to let the browser know which resources to cache and which ones to always request over the network. You reference a manifest from the root HTML element, as shown here:

```
<html manifest="manifest.cache">
```

Note that the URL to the manifest doesn't have to be a static file. It can be a dynamic page (e.g., ASPX or PHP) so long as the resource is marked with a *text/cache-manifest* Multipurpose Internet Mail Extensions (MIME) type. The content of a manifest file is a plain list of resource names with some syntax rules:

```
CACHE:
default.html
styles.css
logo.png

NETWORK:
login.aspx
/public/api

FALLBACK:
login.aspx nologin.png
```

The *CACHE* section indicates resources that always should be cached; the *NETWORK* section lists resources that always should be requested from the server; the *FALLBACK* section indicates alternate resources to use when network is not available.

**More Info** Offline applications are described separately from HTML5; full details about this topic can be found at *http://www.w3.org/tr/offline-webapps*.

## Geolocation

Another compelling functionality that you find around the HTML5 standard is Geolocation, defined as a unified API that locates and describes the geographical position (i.e., latitude and longitude) of the device that hosts the browser. The engine that HTML5 browsers are called to support is centered on the new *geolocation* object exposed by the browser's *navigator* object.

The following code demonstrates how to use the HTML5 API to take latitude and longitude and pass that information to a Google Maps object. The final result is shown in Figure 5-15.

```
<!DOCTYPE html>
<html>
<script type="text/javascript"
        src="http://maps.googleapis.com/maps/api/js?sensor=true" />
<script type="text/javascript">
    function initialize() {
        navigator.geolocation.getCurrentPosition(
               showPositionOnMap,
               function(e) {alert(e.message);},
               {enableHighAccuracy:true, timeout:10000, maximumAge:0 });
        }
    function showPositionOnMap(position) {
        var point = new google.maps.LatLng(
                           position.coords.latitude,
                           position.coords.longitude);
        var options = {
               zoom: 16,
               center: point,
               mapTypeId: google.maps.MapTypeId.ROADMAP
        };
        var map = new google.maps.Map($("#area"), options);
        var marker = new google.maps.Marker({
               position: point,
               map: map,
               title: "Dino Esposito" });
   }
</script>
<body onload="initialize()">
    <div id="area" style="width:100%; height:100%"></div>
</body>
</html>
```


**FIGURE 5-15** Geolocation in action.

Note that browsers normally require the users of pages based on geolocation to approve the sharing of location data explicitly.

**More Info** Geolocation is described separately from HTML5; full details about this topic can be found at *http://www.w3.org/TR/geolocation-API*.

## Audio and Video

One of the biggest gains of HTML5 is saying farewell (but really?) to external plug-ins such as Flash and Silverlight for just playing audio and video. HTML5 brings two new elements—*AUDIO* and *VIDEO*—that point to a URL and play any content. The browser implementation of these tags is also expected to provide a control bar for the user to pause and resume the playback. Here's how to link to an audio resource:

```
<audio poster="init.png" controls="controls">
    <source src="nicestory.wav" />
</audio>
```

The sore point of multimedia elements (mostly video) is the format of files—both file format and codecs. The HTML5 standard won't make an official ruling about codecs, so deciding the format to support will remain up to each vendor. From a developer's perspective, this is not exactly great news because it represents a breaking point. Different browsers support different formats, and you should detect the browser or provide multiple files for the browser to choose. Here's the syntax to indicate a selection of video formats:

```
<video poster="init.png" controls="controls">
   <source src="tiger.mp4" type="video/mp4" />
   <source src="tiger.webm" type="video/ogg" />
   Oops, it seems that your browser doesn't support video.
</video>
```

Note that you use the *controls* attribute to display the control bar and the *poster* attribute to specify an image to use as a splash screen until the media is ready to play.

Popular codecs are MP4, MOV, and AVI. These codecs pose licensing issues to browser vendors; for this reason, Google and Opera are advocating for the new, royalty-free WebM codec and Firefox is supporting OGG/Theora, which the Xiph.Org Foundation made royalty-free. Here's the *VIDEO* element in action in Opera.

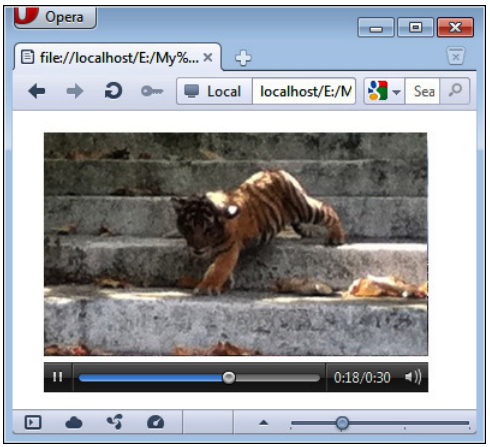



**FIGURE 5-16** The *VIDEO* element in action in Opera.

To cut a long story short, you should plan to have an MP4-encoded video for Internet Explorer and Safari and OGG/Theora for all the others. At present, this seems to be the perfect solution to avoid external plug-ins. But this is a matter that changes frequently, so look before you leap.

## Using HTML5 Today

These days, HTML5 is being talked about a lot, and it is being touted as the next big thing no matter what kind of software applications you're involved with. As is usually the case with any hot technology, there's always hype, as well as good points and strange facts.

### The Hype in HTML5

HTML5 is not here yet, and the World Wide Web Consortium (W3C) is not expected to release any recommendation for at least three more years. Until recently, and mindful of the past browser wars, one would have just dismissed the argument about using an "announced-but-yet-to-come" new version of the HTML language. One would have said something like, "Sure, it sounds great. But we'll probably look at it when most browsers are actively supporting it."

So why talk about HTML5 today, at least three years ahead of any official recommendation? HTML5 is relevant today because most browsers are already incorporating HTML5 capabilities in their latest versions.

Honestly, it's hard to say whether companies are investing in HTML5 because they see real value or because they need to be there because some of their competitors are. But, on the other hand, is this a really relevant point? If your competitor is there, at the very minimum, you need to explore in the same direction. The next version of HTML certainly will play the same prominent place in web development that the technology had played so far. In addition, the new programmer-friendly features in HTML5 make it possible to build other types of applications (for example, mobile native applications for platforms like iPhone and Android).

I don't know—and neither does anyone else—whether HTML5 and JavaScript will form the primary development platform for the years to come. But HTML5 surely is worth a closer technical look.

## HTML5 and Browsers

Only the latest versions of most browsers support HTML5, although not to the same degree. This is still largely reasonable because the draft is not finalized and maybe was not completely crystal-clear when the development of today's latest browsers started. As mentioned, the biggest problem in using HTML5 today is not so much with browsers that don't support HTML5 but with browsers that already support it.

HTML5-ready browsers are still inconsistent in how they implement certain features, such as input fields and some semantic elements. This poses a huge problem: what should you do? Should you accept the reality that you will deliver sites that don't look and behave consistently across all browsers? Or should you rather plan some extra work to make the site look consistent across all HTML5-ready browsers—at your (possibly considerable) expense? And, at that point, what benefit is HTML5 really going to give you at this time?

HTML5 brings huge benefits to web development (and other areas as well).

## The Weirdness in HTML5

HTML has always been a web technology. HTML5, instead, presents itself as a development framework in conjunction with CSS and JavaScript. Looking at the classic web, I am still baffled about using HTML5 today. It's a totally different story, however, if I look at using HTML5 outside the web.

The full set of HTML5 capabilities forms a compelling framework for mobile and tablet applications and even for developing applications to be delivered over a variety of software platforms. A tool like Adobe's PhoneGap allows writing native applications for iOS, BlackBerry, Windows Phone, and Android by using HTML5, CSS3, and recent JavaScript enhancements. In this context, you use the full power of HTML5 without being concerned about the browser's capabilities. Surprisingly enough, in fact, all default browsers available on recent smartphone operating systems are HTML5-ready. Similar full support for HTML5 development is provided in Adobe AIR and was announced by Microsoft for Windows 8. In the end, HTML5 is probably ready today for good development; but it is mostly beneficial in closed environments.

**Important** To get an up-to-date matrix of how the various mobile browsers support HTML5, you can check out *http://mobilehtml5.org*. This site is a very good attempt at tracking and understanding HTML5 compatibility on mobile and tablet browsers, and is being maintained by Maximiliano Firtman, a popular author of many books on mobile web development.

## Summary

Many developers have grown up with the idea of building applications against the most powerful platform possible but providing a way to degrade gracefully on older platforms. This has been the primary approach for web development. In mobile development, this pattern is less effective, even though the real effectiveness can be measured only in specific contexts.

In mobile development, you should follow a bottom-up approach. First, ensure that you can serve any browser; next, proceed by adding more and more features. Each enhancement step that you take represents a class of mobile devices you intend to target and for which you may have the need to create ad hoc views. This is the essence of Progressive Enhancement.

How can you achieve it?

Several libraries, such as jQuery Mobile and SenchaTouch (*http://www.senchatouch.com*), offer an ad hoc framework that guarantees that your markup works well enough on any device. Whether "well enough" is really enough for you is your decision to make. I'd say that it doesn't necessarily happen all the time. You probably don't want to serve plain HTML lists and headings to users; you want to style those pages a little. A library like Modernizr can help in that it does feature detection for you and simplifies the development of dual sites.

When it comes to managing multiple versions of the site (or simply multiple versions of the same pages within one site), you need to find a reliable way to identify devices and their capabilities. This is just where we're going with the next chapter.

CHAPTER 6

# Developing Responsive Mobile Sites

*There is no law governing all things.*

*—Giordano Bruno*

**In this chapter:**

- A Developer's Perspective of Device Detection
- Inside WURFL
- Implementation of a Multiserving Approach
- Summary

Mobile site development can be particularly easy or particularly tricky, with not much middle ground. Depending on the actual requirements, building a mobile site can be as smooth and pleasant as a walk in the park if you don't need to care about the effective capabilities of the requesting browsers. However, it can get really problematic when you need to optimize and adjust what's served to each device. The major issue of mobile site development is not really the implementation of the functionalities—a mobile site generally has fewer functions and actions per page than a desktop site. However, to ensure a great user experience, the team often has to resort to extremely clever coding and design solutions. Overall, what's painful about mobile sites is the wide range of different devices (in the order of thousands) your site may be dealing with.

Years ago, we faced a similar problem for desktop browsers, which libraries like jQuery vastly contributed to solving. Nevertheless, subtle differences between, say, different versions of Windows Internet Explorer or Safari still make it more difficult and more expensive to build a site that looks and works the same way regardless of the browser. However, the range of different capabilities that you experience in desktop browsers is much smaller than what you may find in the mobile space. The problem is the same; the dimension, however, is quite different.

The key lesson that developers and architects have learned in the past is to focus on effective capabilities rather than pointing out a generic behavior associated with a browser's brand and name. This principle, however, is simple to understand but yet hard to adopt. Chapter 4, "Building Mobile Websites," illustrated a few possible solutions, each of which works to some extent. This chapter, on the other hand, presents a different approach (and a specific framework) that currently represents

the state-of-the-art technology for device detection and capabilities. The idea is using an external repository that, constantly updated, provides up-to-date information about any known devices and their capabilities. Put another way, you trust a vendor and use its repository and related application programming interface (API) to figure out what a given device and browser can or cannot do. This chapter mainly focuses on WURFL, short for Wireless Universal Resource File—an open-source project that you can read about at *http://wurfl.sourceforge.net*.

# A Developer's Perspective of Device Detection

How can you detect browser capabilities? Without beating around the bush, detecting capabilities is possible only by querying the browser. However, the browser lives on the client, and you likely need to have this information available on the server to arrange and serve an ad hoc markup. How can you carry browser information to the server and use it within an ASP.NET, PHP, or Java environment?

In addition, browsers expose programmatically only a very small set of capabilities, and hardly through a clean and uniform API. A browser typically lets you know about the width and height of the screen, Ajax support, and some HTML capabilities. This may or may not be sufficient for you.

At present, there are two major schools of thought for solving the problem of detecting a browser's capabilities and adjusting the markup being served. One goes under the name of Responsive Web Design (RWD) and pushes a client-side solution resulting from a combination of JavaScript and CSS. RWD was introduced briefly in Chapter 4. The other solution centers on a human-managed repository of device data that can be checked on the server and help generate device-specific markup.

Let's see what it means to trust (and not trust) the browser to figure out effective capabilities and serve ad hoc pages over mobile devices.

## The Client-Side Route

Because the browser is the client side of a website, you need to build your solution from a huge amount of JavaScript logic. This means that you arrange webpages around a relatively simple HTML layout, which is then reconfigured and styled dynamically on the client. That means having some ad hoc JavaScript functions in your pages that check the browser's width and height and then pick up a different Cascading Style Sheet (CSS) style and perform some reorganization of the content.

The net effect is that an entirely different page structure may be served to different devices, with some original elements hidden or moved around.

This is the essence of the *Responsive Web Design (RWD)* philosophy.

### What's Good About RWD

The most attractive point of RWD is that you build and maintain just one site regardless of the devices that may be used to visit the site. With RWD, you have no need to adjust your solution to match any new devices—if the screen size of new devices is known, the match happens automatically, and users also will see the page layout change as they resize the browser's window.

As originally formulated by Ethan Marcotte in his book *Responsive Web Design* (*http://www.abookapart.com/products/responsive-web-design*), RWD involves the use of a variety of grids in the page layout and operates the dynamic substitution of images and blocks based on CSS media queries. Just the use of media queries makes the design quite flexible because it moves to the browser the burden of switching on the proper cascading style sheet whenever the size of the window matches one of the provided media types.

A good live example of RWD in action is the Boston Globe's website. As Figure 6-1 shows, by resizing the browser, the layout of the page changes significantly and the layout is different as well, with some elements resized and other hidden.

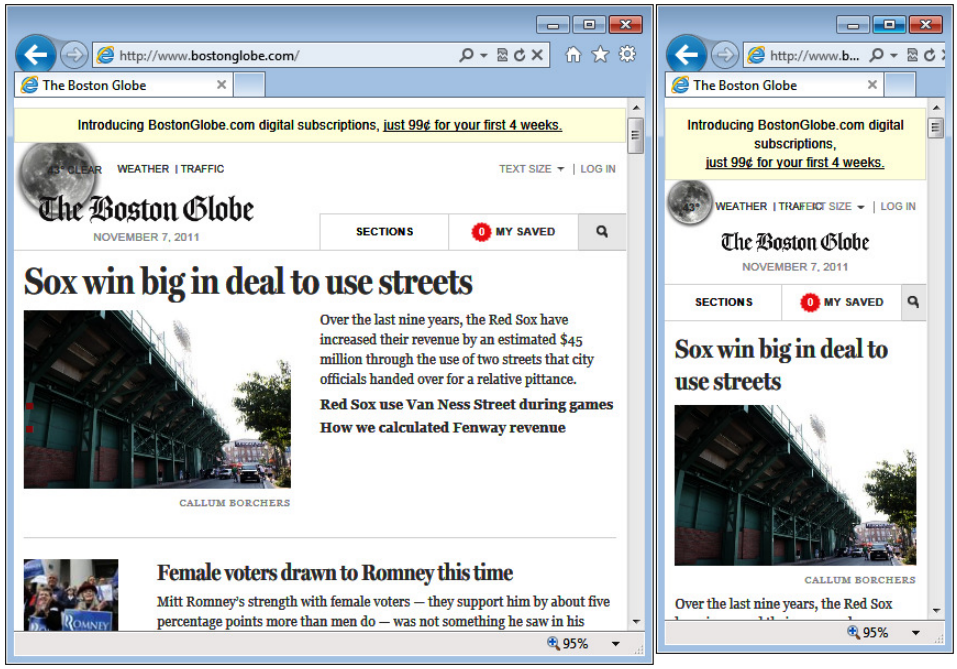



**FIGURE 6-1** A live example of an RWD-powered website.

## Technical Aspects of RWD

Technically, RWD encourages the use of liquid layouts as opposed to classic fixed-width layouts. A liquid layout is a very flexible grid in which every element is sized in percentage units, both in relationship with the parent container and with one another.

In addition to liquid grids, RWD deals with image and font resizing. Font resizing (and in some cases, just a different font) are ways to maintain a good level of cleanliness and keep the text visible and readable.

As the screen size grows or shrinks, you also might want to consider changing the size of the image beyond the simple application of percentage-based *width* and *height* styles to the *img* element. That

means employing tricks to request ad hoc images from the server. For example, a common solution is based on using a custom attribute with the *img* tag that references the full-size image, whereas the standard *src* attribute points to a low-quality image, as shown in the following code:

```
<img src="small.jpg" data-fullsrc="large.jpg" />
```

By using some ad hoc JavaScript code, you can scan the document and replace the URL of all *img* tags with whatever URL is associated with the *data-fullsrc* attribute. This is the trick employed by the code you find at *https://github.com/filamentgroup/Responsive-Images,* a site developed by the Filament Group. Upon page loading, the JavaScript helper file detects the browser width and, using a configurable break-point value (i.e., 480 pixels), determines which image to set.

You should note that in this context, the use of CSS media queries (see Chapter 4) helps significantly in automating the change of layout at different resolutions. You can read more about RWD at *http://www.alistapart.com/articles/responsive-web-design*.

## What's Bad About RWD

The first problem with RWD to consider is that it is *not* a solution specifically created for mobile sites. With its inherent flexibility, RWD can be used to arrange pages that fit to mobile devices, but this approach is not free of issues (some of which are significant). The bottom line is that RWD is a great approach for building compelling sites, and it reacts well to different browser window sizes. However, it's one thing if a user chooses to resize the desktop browser window; it is quite another if a user visits the site using a mobile browser with a fixed screen size. The difference between desktop and mobile browsers is in the underlying hardware—speed of download, reliability of connection, processing power, cache size, memory, and so on—plus some contextual issues such as the ability to serve different content to mobile users and use hardware (e.g., GPS).

In addition, a mobile site is considered to be a pretty unique solution. At best, very little in a desktop site is specifically mobile-oriented. By construction, RWD can't add mobile-specific logic to a desktop site.

Now let's review in more detail what's problematic about RWD. These aspects are reduced here to just five categories: images, nonviewable content, CSS media queries, overall design considerations, and development practices. The impact of these aspects on mobile projects is not uniform, but it is hardly null either. The depth of impact, however, largely depends on what you mean by *mobile*.

If you plan to limit your site to smartphones only, then RWD is probably not such a bad idea, so long as a single site solution fits into your business model. You probably can adjust for the limitations of an iPhone, a Windows Phone, or a high-end Android device. But what about other devices? So long as your strategy is focused on smartphones, you can ignore other lower-end devices; otherwise, you should ignore RWD and focus on the server-side route to device capability detection.

**Note** Nearly all reports agree that a large share of mobile traffic is due to smartphones. It can be estimated to be about 60 percent in general, but in some European countries, smartphone traffic (mostly on the iPhone/iPad) is well beyond 80 percent. Yet, very few companies would lightheartedly decide to disregard a section of potential customers completely. That clearly depends on the company's business core, but in general, going for a mobile-specific site is the preferred approach. And once you're there, there's no reason for taking a client-side route: it seems much more efficient to detect capabilities on the server and serve directly what's needed.

## Technical Downsides of RWD Implementation

In any website, images tend to be numerous and large. On a smartphone, it may be acceptable to stress CPU and memory to resize images on the fly. On any other device that is less powerful than a high-end smartphone (e.g., 1 Ghz CPU, 512 MB of RAM, and a dedicated GPU), it is probably much less acceptable. Therefore, as far as images are concerned, you want to employ some tricks to download smaller ones no matter what. The benefits are partly limited by the need to add some extra scripts and style sheets.

When an RWD page is resized, whether dynamically or because it is displayed on a mobile device, sometimes it displays a greatly simplified structure where, say, the layout is reduced to two columns or even just one column. Where does the extra content go? Because everything happens rigorously on the client side, any extra content is not even detached from the Document Object Model (DOM), but is simply hidden at the CSS level. Seen from a mobile perspective, this means that RWD also has the potential of downloading a large amount of content and logic that will never be displayed on the mobile device.

CSS media queries work nicely for desktop solutions, as well as for very limited mobile scenarios, those where you can exercise control over the browsers and have tested it enough to feel comfortable. CSS is about style; effective rendering is also about downloading content that has been built appropriately for a mobile device. In mobile, performance is key, especially because mobile applications are likely to be used over slow connections. Solutions employing CSS media queries to adjust a single layout achieve the goal of hiding content based on a (very) small set of parameters (screen, resolution, orientation, and color depth). CSS media queries give you no chance to optimize the layout and style based on specific capabilities such as support for WiFi, Ajax, preferred images, video, plug-ins, and more. Overall, CSS media queries tend to offer the promise of a simple solution along the lines of "design once, display everywhere." However, this is a mostly false promise that turns into a problem as soon as you widen your horizons and address more than just smartphones. Furthermore, support for CSS media queries is limited across mobile browsers, paradoxically meaning that an RWD solution based on CSS media queries covers only a minimal share of mobile browsers.

In general, a mobile site must be resourceful and designed from scratch around a simple layout filled with optimized markup that is guaranteed to work on the requesting browser. This requires deep and accurate knowledge of the device capabilities—many more capabilities than the five that you can control over media queries where media queries themselves are supported.

As a final note, consider that RWD brings a very complex page structure that requires a lot of work to render. You may not notice any significant difference on a smartphone, but it surely will show up on other devices.

### Why a jQuery-Like Approach Is Not Always Effective

While it'll never become a settled point, the browser compatibility problem for the desktop has been simplified significantly by the jQuery library. The library takes the burden of hiding a lot of the browser differences and internally forks the code as appropriate to provide the same behavior regardless of the requesting browser. This simple pattern, however, doesn't work in the mobile space.

A JavaScript library is hosted in the browser and, to do any magic, it can access only the information that the browser makes available. The browser capabilities directly available through the browser are very limited, and those which can be tested programmatically (i.e., Ajax support or support for some HTML5 API, such as local storage) are only a small subset. A lot of properties you want to check to optimize mobile download can't just be tested on the fly—you must *know* about them rather than *detect* them.

But even if you can manage to inject browser capabilities into the client page as JavaScript-accessible data, you're won't go much further. The point of mobile is minimizing Hypertext Transfer Protocol (HTTP) requests and the data being downloaded, and simplifying the markup. This can be achieved only with strict control over the requests made on the server side.

**Note** If a jQuery-like approach to mobile is not ideal, what's the point of a library like jQuery Mobile? Does it really make sense? It's a different issue. My point is that you cannot blindly expect that some JavaScript library will take some HTML and magically turn that into a mobile page. The jQuery Mobile library does a great job helping you to arrange a mobile page; ad hoc content and architecture are up to you.

## The Server-Side Route

The server-side route to mobile development is based on the idea that you first *read* about the effective browser capabilities. Next, armed with that knowledge, you intelligently decide what the most appropriate content for the request and the browser would be.

How would you find out about browser capabilities? At present, the most effective strategy seems to be to use a Device Description Repository (DDR)—namely, some sort of database that stores nearly all possible properties of nearly all devices and that is constantly updated as new devices hit the market.

**Important** Previously, I said that at present, employing a DDR seems to be the most effective strategy. Well, I'd even formulate around that the hypothesis that DDR is the *only possible* way to know about effective device capabilities. As we'll see later in this chapter, a few capabilities exist that can't be checked programmatically (e.g., whether inline images are supported). For them, therefore, you have to *know* the value reading from a reliable and credible store. In light of this, the real problem around DDR is picking up the most accurate and best-maintained DDR that offers the most favorable pricing model and a wide range of delivery options (i.e., on-premise, cloud).

## Device Recognition and Description Repositories

Very few server programming environments for building web applications offer native tools to query for some browser capabilities. A notable exception is ASP.NET, which offers the *Request.Browser* object. This object retrieves device information from a bunch of XML files deployed to the server. ASP.NET installs one XML file per supported class of browser. The key to locating a record in this repository is the browser's user agent (UA) string that gets passed with the HTTP request. If a match is found between the UA string and the files in the repository, then browser information is collected and shared with the application. In ASP.NET, the machinery is hidden from view and all that a developer has to do is check capabilities through the *Request.Browser* object.

In other environments, getting a device description repository and API is up to you. In Java, you can probably look at Spring Mobile (see *http://www.springsource.org/spring-mobile*) but with caution. Spring Mobile doesn't currently detect specific capabilities and is limited to telling you whether the requesting browser is on a mobile device. An extension to Spring Mobile is in the works and uses a DDR under the hood (specifically, WURFL).

Chapter 4 discussed that the default ASP.NET mechanism is easy to use, but its effectiveness depends exclusively on the accuracy and quality of the data in the repository being used. So what would be a good repository of device information? Does one exist that can work in a cross-platform manner?

## DDRs and Crowd-Sourcing

The number of different models of mobile devices is in the order of thousands and growing. The properties that a developer can find interesting for each device profile in a particular use-case may vary significantly, but if you take the union of all of them, you easily reach several hundred. How can you ensure that a DDR can keep this information accurate and up to date?

An extremely popular DDR is WURFL— the Wireless Universal Resource File—an open-source community-powered project started by mobile developer Luca Passani about a decade ago. At that time, the term *wireless* was used with the same meaning as we use *mobile* today. WURFL is a XML database that currently counts more than 15,000 profiles of mobile devices and matches half a million UA strings available in the wild. A profile contains over 500 properties referred to as *capabilities*. WURFL evolved into a commercial initiative in 2011.

**Important** WURFL is released under the AGPL v3 license—an open-source license approved by both the Free Software Foundation (FSF) and the Open Source Initiative (OSI). Due to the strict licensing terms of AGPL v3, however, commercial users may need to buy a commercial license. In a nutshell, to use WURFL without buying a commercial license, you need to make your full source code available as open-source, regardless of the fact that the code is hosted on your own web server. For more details, refer to *http://wurfl.sourceforge.net/faqlicensing.php*.

As discussed in Chapter 4, WURFL is not the only DDR out there, but WURFL can be considered the de facto standard. Among other things, WURFL is employed in the mobile platform of Facebook and Google.

# Inside WURFL

A DDR is the component that works like an oracle and tells you the whole truth about the mobile browser that is viewing your page so that you can decide intelligently what response to serve. Let's explore the most popular DDR available today in more depth.

## Structure of the Repository

The WURFL DDR consists of an XML file that measures about 1 MB compressed and about 15 MB expanded. You can download the file from *http://wurfl.sourceforge.net*. Note that before you download the file, you must agree explicitly to some terms and conditions. Basically, you are authorized to use the WURFL file without modification and only through one of the standard WURFL APIs as provided by ScientiaMobile.

### The Overall XML Schema

The WURFL data file consists of a flat list of <*device*> elements. Here's the overall skeleton of the database:

```
<devices>
  <device id="..." user_agent="..." fall_back="...">
     <group id="...">
        <capability name="..." value="..." />
        ...
     </group>
     ...
  <device>
  ...
</devices>
```

The *id* attribute uniquely identifies a device with a name. The *user_agent* attribute indicates a specific UA string to be matched.

The key attribute is *fall_back,* which refers by name to other <*device*> elements. The *fall_back* attribute indicates the device from which missing capabilities of the present device will be inherited. Put another way, each device section describes just the delta between the current device and its parent device. All devices refer directly or indirectly to a root generic device, which ensures that any capabilities supported always have a default value and no exceptions will be thrown during queries. WURFL supports a number of root generic devices, one for each category of devices recognized: mobile phones, tablets, smart TVs, and possibly more in the future.

**Note** An optional attribute that you may find on some <*device*> elements is *actual_device_root,* typically associated with devices whose WURFL ID terminates with *_ver1*. This attribute is intended to point out the best profile to represent all devices in WURFL with that manufacturer and model. For example, although you may find 20 different profiles for, say, a Nokia N70, only one of those will be marked as *actual device root*. This is useful for developers who use WURFL to generate lists of devices ordered by manufacturer and model.

## Groups of Capabilities

Each device is associated with a list of capabilities. A capability is described as a name/value pair in which the value is always considered to be a string. This means that the WURFL API always will return capability values as plain strings, with no attempt to match the value to a specific type such as Boolean or Integer.

This choice has been made to prioritize extensibility and performance of the API over everything else. In fact, quite a few capabilities take values from an enumeration of values. For example, the *pointing_method* capability indicates how links are activated on the device. Possible values are *stylus*, *joystick*, *touchscreen*, *clickwheel*, or the empty string. All these options could be expressed comfortably as an *enum* type in the Java and .NET languages. However, in this case, any extension to the data file to add a new possible pointing method also would require a change to the API, which potentially could break existing applications.

To keep things manageable, capabilities are split into groups. Table 6-1 lists the currently recognized groups. Groups, however, have no role in the API in the sense that you don't need group information to retrieve the value of a capability.

**TABLE 6-1** The Most Relevant Groups of Browser and Device Capabilities in WURFL

| Group | Description |
|---|---|
| ajax | Despite the name, this group defines capabilities that also go beyond plain Ajax programming. It tells you whether Ajax is supported, but also whether DOM and CSS manipulation and geolocation are allowed. |
| bearer | Capabilities regarding networking aspects, such as support for radio, WiFi, virtual private networking (VPN), and the maximum reachable bandwidth. |

| Group | Description |
|---|---|
| chtml_ui | Capabilities related to Compact HTML markup. |
| chips | Capabilities related to features available through extra chips installed on the device such as FM radio and a near-field-communication (NFC) facility. |
| css | Capabilities related to CSS features, such as sprites, borders, rounded corners, and gradients. |
| display | Capabilities related to screen size (both pixels and millimeters) and orientation. |
| flash_lite | Capabilities related to built-in support for Flash application types and versions. |
| html_ui | Capabilities related to content served with the HTML MIME type. The group includes properties about viewports, HTML5 canvas, inline images, and preferred document type definition (DTD). |
| image_format | Boolean capabilities related to the support of a few image formats. |
| j2me | Capabilities telling developers which Java features are available for midlets in the Java Micro Edition (Java ME) run-time location, screen size, sockets, images, multimedia and more. |
| markup | Boolean capabilities related to a variety of markup types being supported, including Extensible Hypertext Markup Language (XHTML), Wireless Markup Language (WML), and HTML. |
| mms | Capabilities that are relevant for MMS, such as images supported, videos, and maximum frame rate. |
| object_download | Capabilities related to downloadable objects, such as video clips, images, wallpapers, screensavers, and ringtones. |
| pdf | Capabilities related to native support for PDF content. |
| playback | Capabilities related to supported video formats and codecs for content downloaded from websites. |
| product_info | Capabilities related to the device, such as brand and model name; whether it is a mobile device, phone, or tablet; operating system; keyboard; or browser. |
| rss | Capabilities related to native support for Really Simple Syndication (RSS) feeds. |
| security | Capabilities related to Hypertext Transfer Protocol Secure (HTTPS) support and International Mobile Equipment Identity (IMEI) visibility. |
| sound_format | Boolean capabilities related to a variety of different sound formats. |
| smarttv | Capabilities that are relevant for smart TVs. |
| sms | Capabilities that are relevant for Short Message Service (SMS), EMS (rich-text SMS), and ringtones, including those specific to vendors like Nokia and Panasonic. |
| storage | Capabilities related to the size of the pages that the device can manage. |
| streaming | Capabilities related to supported video formats and codecs for content streamed from websites. |
| transcoding | Capabilities aimed at identifying the request as coming from a transcoder or a proxy—a piece of software that may act as a gateway and hide real device information. These capabilities are offered in case you need to handle such requests in a special way. |
| wap_push | Capabilities aimed at detecting effective Wireless Application Protocol (WAP) features. |
| xhtml_ui | Capabilities related to XHTML markup. |

In addition to these groups, WURFL has a group of deprecated capabilities that may be removed from the repository at any time. Details about the deprecated capabilities can be found at *http://wurfl.sourceforge.net/help_doc.php#deprecated*. For the most part, these are capabilities that now are expressed more properly by other properties in one of the aforementioned groups.

This listing shows an excerpt illustrating the CSS capabilities of the generic device—the root of WURFL devices:

```
<group id="css">
   <capability name="css_gradient" value="none" />
   <capability name="css_border_image" value="none" />
   <capability name="css_rounded_corners" value="none" />
   <capability name="css_spriting" value="false" />
   <capability name="css_supports_width_as_percentage" value="true" />
</group>
```

The following example, instead, shows the same group for a generic Android device:

```
<group id="css">
    <capability name="css_border_image" value="webkit"/>
    <capability name="css_rounded_corners" value="webkit"/>
    <capability name="css_spriting" value="true"/>
    <capability name="css_supports_width_as_percentage" value="true"/>
</group>
```

As you can see, some properties are overridden.

## The WURFL Patch Files

The WURFL repository comes with two or more files (arbitrarily named): one is the XML file that represents the repository itself (usually named Wurfl.xml); the others are patch files that are usually named after the pattern *xxx_patch.xml*. Patch files are optional.

**Note** The WURFL API is based on a configuration module through which you point to the repository and optional patch files. In ASP.NET, you can do that either programmatically through a fluent interface or via a custom section in the Web.config file.

A patch file allows you to make changes to some capabilities within the default repository without physically tweaking the original file (which would break the license anyway, even if you did it on your legally acquired copy). In other words, a patch file is the provided way to override some content within the WURFL database. If any patch is found when the WURFL file is parsed, its content is imported to build a modified version of the repository. Here's an excerpt from a patch file that adds support for Firefox 10 (in case it is not supported in the latest update of the repository):

```
<device user_agent="Firefox" fall_back="generic_web_browser" id="firefox">
     <group id="product_info">
          <capability name="brand_name" value="firefox" />
     </group>
</device>
```

```
<device user_agent="Mozilla/5.0 (Windows NT 5.1; rv:10.0) Gecko/20100101 Firefox/10.0"
        fall_back="firefox" id="firefox_10_0">
    <group id="product_info">
         <capability name="model_name" value="10.0"/>
    </group>
</device>
```

Why would you want to use a patch file?

Overall, the primary reason for using a patch file is that you have your own good reasons to assign certain capabilities a different value. For example, suppose you tailor-made a website for tablet devices. Next, you run across a particular device whose screen is large enough to accommodate the tablet user interface that you designed. Unfortunately, though, WURFL continues to consider that particular device as something other than a tablet. You then create a new patch file (or edit an existing one) and override the *is_tablet* capability, only for that particular UA. Here's an example:

```
<device user_agent="your nice tablet device that WURFL doesn't consider a tablet"
        fall_back="generic_mobile" id="mytablet">
     <group id="product_info">
          <capability name="is_tablet" value="true" />
     </group>
</device>
```

Another scenario where patch files are useful is when you need a capability that is not natively supported in WURFL, either because it is too specific for your application or because nobody ever thought of it before. Finally, a patch file can come to the rescue when some wrong data is found to exist in the original WURFL database. For more information and examples of patch files, visit *http://wurfl.sourceforge.net/patchfile.php*.

## Top 20 WURFL Capabilities

Let's take a closer look at some of the WURFL capabilities to get a precise idea of the level of control over the content being served that you can gain through WURFL. I picked my favorite 20 capabilities; but the choice is arbitrary—it doesn't mean that the remaining 500 or so capabilities are less important. Full documentation about capabilities can be found at *http://wurfl.sourceforge.net/help_doc.php*.

For the sole purpose of this section, I organized these 20 top capabilities on a per-scenario basis.

### Identifying the Current Device

Table 6-2 lists some very handy capabilities that describe the device being used to carry the current request. The table shows the name of the capability, its WURFL group, its description, and possible values for it.

**TABLE 6-2** Device-Related Capabilities

| Capability | WURFL Group | Value | Description |
|---|---|---|---|
| *is_wireless_device* | product_info | *true/false* | The device is wireless. |
| *is_tablet* | product_info | *true/false* | The device is a tablet. |
| *is_smarttv* | smarttv | *true/false* | The device is a smart TV. |
| *device_os*<br>*device_os_version* | product_info | *string* | The name and version of the current device (i.e., Android 2.2). |
| *resolution_width*<br>*resolution_height* | display | *integer* | The screen width and height in pixels. |
| *max_image_width* | display | *integer* | The maximum width, in pixels, of images as they can be viewed on the device. |
| *can_assign_phone_number* | product_info | *true/false* | The device can be associated with a phone number. Used to distinguish devices using a SIM only to browse the web. |
| *pointing_method* | product_info | *joystick, stylus, touchscreen, clickwheel, ""* | The method used to select links. Note that the empty string indicates classic four-way navigation on devices with top, left, right, and bottom buttons to navigate links. |
| *brand_name*<br>*model_name*<br>*marketing_name* | product_info | *string* | The brand (i.e., HTC), model name (i.e., HTC A8181), and even marketing name of the device (i.e., HTC Desire). |

Some of these properties allow you to catalog the device very precisely. For example, you can check whether the incoming request comes from a browser hosted on a wireless device. The *is_wireless_device* capability returns true for any UA string matched to mobile devices such as cell phones, PDAs, and tablets (but not laptops and smart TVs like AppleTV). If all you need is to detect a mobile device, this property is all you need. For a more detailed analysis, you can also check *is_tablet,* which returns true on iPads; and *can_assign_phone_number,* which returns true on mobile phones (which can have a phone number assigned) but not on, say, iPods. Another similar capability is *has_cellular_radio* (in the *bearer* group): this capability indicates whether the device can mount a SIM for whatever reason. You can have a SIM on an iPad but not, say, on an iPod Touch.

If you need to distinguish iOs from Android or Windows Phone devices, you can use the *device_os* and *device_os_version* capabilities. If you then need to know the exact device (manufacturer and product name) go with *model_name* and *brand_name*.

The actual size of the screen is returned by *resolution_width* and *resolution_height*. Finally, information about touch capabilities are is returned by the *pointing_method* capability when the value equals *touchscreen*.

**Note** One might reasonably wonder why *is_smarttv* has its own group and isn't grouped with other analogous *is_xxx* capabilities. Smart TVs are a new market with tremendous potential, and in the near future as more and more TVs will include a web browser site, developers may face the need to optimize for smart TVs as well. In WURFL, an ad hoc group is already in place.

## Serving Browser-Specific Content

Table 6-3 lists a few capabilities that can help you fine-tune the markup that you serve to the browser. WURFL is full of capabilities for fine-tuning the markup being served. The three capabilities below are representative of scenarios in which you will want to use different markup templates on the server to generate the view.

**TABLE 6-3** Capabilities for Serving Ad Hoc Content

| Capability | WURFL Group | Value | Description |
|---|---|---|---|
| *viewport_supported* | html_ui | *true/false* | The browser supports the *viewport* meta tag. |
| *image_inlining* | html_ui | *true/false* | The browser can display images embedded via the data Uniform Resource Identifier (URI) scheme. |
| *full_flash_support* | flash_lite | *true/false* | The browser fully supports Flash. |
| *cookie_support* | xhtml_ui | *true/false* | The browser supports cookies. |
| *preferred_markup* | markup | *string* | The preferred type of markup to serve to the browser. (See later in this section for more information on this topic.) |
| *png,*<br>*jpg,*<br>*gif,*<br>*tiff,*<br>*greyscale* | image_format | *true/false* | The browser can display images of a given type. |

Some mobile browsers assume they can render every page, so they shrink the actual page to the actual screen size and let users zoom in and out to view a section of the page in a convenient manner. The HTML *viewport* meta attribute has been introduced to enable the developer to indicate which size the virtual screen—the viewport— actually should have. The *viewport* meta tag is not standard, though, and it is safer if you check before you emit it. The *viewport_supported* capability just tells you so.

Browsers treat images as separate resources and trigger an additional request to download them (if not cached locally). For mobile devices, HTTP requests carry a much higher cost than they do for desktop browsers, so any techniques are welcomed that reduce the number of HTTP requests necessary to finalize a page. A common technique consists of embedding small images as Base64-encoded text within the HTML page, as demonstrated in Chapter 4.

The *image_inlining* capability allows you to know in advance whether the requesting browser will be able to show correctly an image embedded in this way. If you fail this check, though, the worst problem you can run into is that the image is replaced by the browser's specific placeholder for missing images.

The *flash_lite* group has modeled Flash-related capabilities on mobile devices. Between 2007 and 2009, Adobe has promoted a Flash profile called *Flashlite* on mobile devices in Japan, South Korea, the United States, and regions enabled for Global System for Mobile Communications (GSM). Nokia has been one of the main proponents of *Flashlite* on their Series 60 devices for a long time. The different *Flashlite* capabilities indicate whether the player can be used to deliver, for example, screensavers and wallpapers. The *flash_lite* group also contains the *full_flash_support* capability, which tells you whether a given device can display the Flash content that we have come to love (or hate) on the web.

Lately, and especially in relationship to Adobe's recent moves, the focus on *Flashlite* for mobile has diminished greatly. With Adobe focusing on HTML5 development, it is hard to imagine new mobile development being done using Flash. However, there's still older code around, so this WURFL group still may play a role.

In mobile web, there are two types of markup languages, which are only apparently similar:

- **XHTML MP** This is a mobile-optimized markup format that browsers find extremely fast to parse and render. In addition, by simply seeing the Multipurpose Internet Mail Extensions (MIME) type, any browser can reasonably perceive that the page is a mobile page. Unfortunately, the markup language is not as powerful as plain HTML, and support for DOM manipulation, CSS, and JavaScript is not really advanced—at least not in a cross-browser way.
- **HTML/viewport** This is plain HTML markup with the addition of the *viewport* meta tag. HTML/viewport was essentially introduced by iPhone and Safari for mobile. As the MIME type is the same as for a full webpage, Apple added the *viewport* meta tag as a clue to the browser about the page being mobile.

In WURFL, the *preferred_markup* capability indicates which type of markup is ideal for a given browser. If the capability returns *html_wi_oma_xhtmlmp_1_0,* then you should serve XHTML MP markup. If the returned value is *html_web_4_0*, then you'd better go with plain HTML and use the *viewport* meta tag to mark the page as mobile.

Knowing in advance that the page is mobile helps the browser to arrange an optimal rendering avoiding shrunk pages and the need of zooming in to interact effectively.

## Understanding JavaScript Capabilities

Table 6-4 lists some of the capabilities related to the JavaScript support available in the device browser. This is one of the trickiest points to consider when it comes to serving content to mobile devices. If you mostly focus on smartphones or relatively recent high-end phones, you can assume JavaScript support from the browser. Maybe you won't find advanced HTML5 capabilities in the browser, but the

core work of applying CSS and manipulating the DOM after Ajax calls is always possible. In general, though, the more you enlarge your audience, the more you might want to check these basic aspects.

**TABLE 6-4** JavaScript Capabilities

| Capability | WURFL Group | Value | Description |
|---|---|---|---|
| *ajax_xhr_type* | ajax | *none, standard, msxml2, legacy_microsoft* | An object that enables Ajax in the browser. |
| *ajax_support_javascript* | ajax | *true/false* | The browser supports JavaScript. |
| *ajax_manipulate_dom* | ajax | *true/false* | The browser allows to use JavaScript to apply dynamic changes to the DOM. |
| *wifi* | bearer | *true/false* | The browser supports WiFi connectivity. |

The *ajax_support_javascript* capability returns *true* if a minimum set of features are known to be available on the device. The minimum set of features includes the ability to display message boxes via functions like *alert* and *confirm*, access form elements and modify values dynamically, change document location, and start timers. Ajax capabilities are measured against the browser object used to trigger out-of-band calls. The *ajax_xhr_type* capability takes values like *msxml2* or *legacy_microsoft* if Ajax is possible via ActiveX components like *Microsoft.MsXml2* and *Microsoft.XmlHttp*. The value is standard if Ajax is implemented via the browser's native *XmlHttpRequest* object.

Ajax capabilities are also doubly linked to WiFi support because a WiFi connection makes it easier and faster to place a network call. You check whether the device has *wifi* capabilities through the *wifi* Boolean capability. Note that WURFL only tells you that *wifi* is supported on the phone. It doesn't tell you whether a WiFi connection is currently active or not. This comes as no surprise because WURFL is a static data repository that only recognizes capabilities.

## HTML5-Related Capabilities

WURFL also contains a bunch of capabilities that are related to the set of technologies that are grouped under the HTML5 umbrella. Table 6-5 lists some of them.

**TABLE 6-5** HTML5 Capabilities

| Capability | WURFL Group | Value | Description |
|---|---|---|---|
| *canvas_support* | html_ui | *true/false* | The browser supports canvases. |
| *css_gradient* | css | *true/false* | The browser can render CSS3 gradients. |
| *ajax_preferred_geoloc_api* | ajax | *none, gears, w3c_api* | The browser supports geolocation either through Google Gears or the W3C API. |

The *canvas_support* capability and the *css_gradient* capability are both Boolean properties that indicate whether the browser can display a canvas and can render a CSS gradient as specified in CSS3. Finally, the *ajax_preferred_geoloc_api* capability indicates whether the browser supports geolocation and, if so, through which API. There are two options: Google Gears (now deprecated) and the W3C API, which is the current standard.

## Using WURFL from ASP.NET

Now that we know enough about WURFL capabilities, it would be nice to see what's required to use them in a real website. For this purpose, I'll update the sample device detector website I built for Chapter 4 and make it powered by WURFL for the device detection logic.

In Chapter 4, I presented an ASP.NET MVC application and used a custom algorithm to detect mobile devices and then relied on the Mobile Device Browser File (MDBF) or native ASP.NET DDR solutions for some additional capabilities. I didn't need to use any special API—it was plain ASP.NET programming. To incorporate WURFL, instead, you need to become familiar with its standard API. I'm presenting the WURFL API in the context of an ASP.NET MVC website, but the steps and the logic of the API are the same for ASP.NET Web Forms, Java, and PHP as well.

### Introducing the WURFL API

To enable the WURFL API, you reference the packages for the specific platform (assemblies in .NET) and add some bootstrap code to initialize the library and load the WURFL data. Past the onetime initialization, you have a WURFL manager object that you might want to cache for serving successive requests.

The WURFL manager object is the primary intermediary between your code and the WURFL data. This object controls the device repositories and selects the right device for any provided UA string. You use the manager to read about specific capabilities. The manager object caches the WURFL data internally and exposes it through an in-memory dictionary. As a developer, you are only responsible for caching the manager in any way that suits you—local ASP.NET cache or any sort of distributed cache you may be using, such as AppFabric Caching Services. Any access to capabilities happens in a constant time because it is as complex and accessing a dictionary entry.

The manager returns a specific device object based on the UA string or the entire request object including additional HTTP headers. Note that currently the .NET API for WURFL is compiled against the Microsoft .NET Framework 2.0. On one end, this ensures maximum compatibility; on the other hand, however, it doesn't allow you to use base types for intrinsic objects added later—*HttpRequestBase,* for example. You must be ready to cast types to have things compile successfully.

### Adding WURFL to the Device Detector Website

The simplest way to add WURFL to an ASP.NET project is via NuGet. As Figure 6-2 shows, by typing **WURFL API** in the Search box of Microsoft Visual Studio, you immediately get the reference and can enable your project to support WURFL in a click or two.

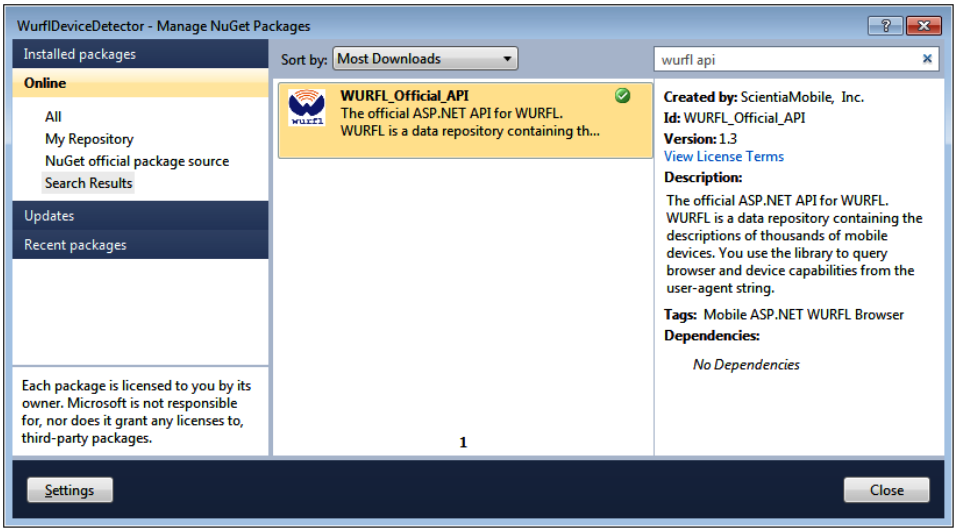



**FIGURE 6-2** Adding WURFL to a Visual Studio project via NuGet.

Figure 6-3 shows how the NuGet package modifies your project. It adds an *App_Data* folder and copies the WURFL repository into it. Note that the NuGet package gives you the latest public release of the repository available at the time that the package was uploaded. If you have a commercial license, you may have access to more recent versions of the repository.

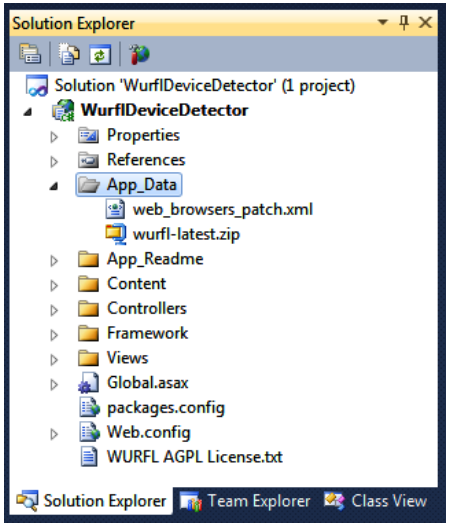



**FIGURE 6-3** The ASP.NET project as modified by the WURFL NuGet package.

The next steps consist of tweaking the code in Global.asax to bootstrap the WURFL library and adding actual code that deals with capabilities.

## Loading WURFL Data

The following code shows the minimum that you need to have in your *Application_Start* handler. Note that the *RegisterRoutes* method is related to the ASP.NET MVC plumbing.

```
protected void Application_Start()
{
    RegisterRoutes(RouteTable.Routes);
    RegisterViewEngines(ViewEngines.Engines);
    Wurfl.Initialize();
}
```

*RegisterViewEngines* is a method discussed in Chapter 3 that adds the ability to fork the views rendered for mobile browsers automatically. The method enables you to have an *index.mobile.cshtml* view along with an *index.cshtml* view and have the right one picked up based on whether the requesting browser is mobile or not. Note that the code in Chapter 4 for switching automatically to mobile views now should be updated to perform the device detection via WURFL.

The *Wurfl* class feature in the listing is a helper class created to minimize the code to write to set up WURFL in a project. Here's the full source code:

```
public class Wurfl
{
    /// <summary>
    /// Internal name used to cache the manager
    /// </summary>
    private const String WurflManagerCacheKey = "__WurflManager_v1_3";

    /// <summary>
    /// Gets the currently cached instance of the WURFL manager object.
    /// </summary>
    public static IWURFLManager Manager
    {
        // We're assuming that the Cache object was correctly populated in global.asax at
        // application startup.
        get { return HttpContext.Current.Cache[WurflManagerCacheKey] as IWURFLManager; }
    }

    /// <summary>
    /// To be called once in the app, loads the WURFL database in memory.
    /// </summary>
    public static void Initialize()
    {
        const String wurflDataFilePath = "~/App_Data/wurfl-latest.zip";
        const String wurflPatchFilePath = "~/App_Data/web_browsers_patch.xml";

        var wurflDataFile = HttpContext.Current.Server.MapPath(wurflDataFilePath);
        var wurflPatchFile = HttpContext.Current.Server.MapPath(wurflPatchFilePath);
        var configurer = new InMemoryConfigurer()
                .MainFile(wurflDataFile)
                .PatchFile(wurflPatchFile);
        var manager = WURFLManagerBuilder.Build(configurer);
        HttpContext.Current.Cache[WurflManagerCacheKey] = manager;
    }
}
```

Note that patch files are used in this example just for clarity. You should use patch files only if you really have something to fix in the repository.

To initialize WURFL, you indicate the location of the data. The configurer is the object that knows how to retrieve and load data files for you. The *InMemoryConfigurer* accepts file names (data file and patch file) as plain strings; on the other hand, *ApplicationConfigurer* reads them from the Web.config file. To use the *ApplicationConfigurer* class, you need the following section of code in the configuration:

```
<wurfl>
   <mainFile path="~/App_Data/wurfl.latest.zip" />
   <patches>
     <patch path="~/App_Data/my_patch.xml" />
     ...
  </patches>
</wurfl>
```

Once the configurer is in place, you build the WURFL manager object and then cache it for any later use. The previous listing also features a public property—*Manager*—to retrieve quickly a reference to the manager when you need to get device information.

**Note** The WURFL API supports both ZIP and GZ compression formats, so you can store compressed files on your web server as well. The uncompressed size is currently around 15 MB, while the compressed size is around 1 MB.

## From the UA to a Virtual Device

WURFL figures out device capabilities by sniffing the UA string. Internally, the UA string is processed to remove noise that may have been added along the way by gateways and proxies and to identify the pieces of information that really matter. Each UA goes through a pipeline of special components called *matchers* that just attempt to match the normalized UA to keywords that they have been instructed to deal with. For example, the WURFL library contains an Android matcher that knows how to recognize all UAs coming from such devices. At the end of the matching phase, you have a *Device* object ready for queries. Here's the code that you need to get a *Device* object:

```
var deviceInfo = Wurfl.Manager.GetDeviceForRequest(Request.UserAgent);
```

You can call this code from anywhere in your pages where you have access to the ASP.NET HTTP context. In an ASP.NET MVC application, you can call this code from within a controller method. In an ASP.NET Web Forms page, you can call this code in any page event—for example, *Page_Load*.

The *GetDeviceForRequest* method accepts the UA string as well as the ASP.NET *HttpRequest* object. Because the WURFL library is compiled against the .NET Framework 2.0, you cannot use the *HttpRequestBase* object which, in newer versions of ASP.NET, is just what *HttpContext.Request* returns.

Once you have a *Device* object, you can start querying for capabilities.

## Querying for Device Capabilities

Let's see how to use WURFL to check whether the requesting browser is hosted in a mobile device. As mentioned, this is the crucial test that helps our ASP.NET MVC application (via the custom view engine) to switch automatically to a mobile view. Here is the code for this:

```
// Check whether the requesting browser is hosted in a mobile device
var ismobile = deviceInfo.GetCapability("is_wireless_device").ToBool();
```

The *GetCapability* method on the *Device* object takes a string representing the capability name and returns a string. In general, the value of a WURFL capability maps to any of the following:

- *True/false* Boolean strings
- Integers
- Plain strings
- Special strings that correspond to entries in an enumerated type

By design, WURFL simply returns strings; any mapping to more specific types is entirely on your own. However, this also gives you the opportunity of shaping things the way you like. You also should create a bunch of extension methods for strings to convert Boolean and integer strings quickly to actual Boolean and Integer values. The *ToBool* method shown previously may be coded as shown here:

```
public static class StringExtensions
{
    public static Boolean ToBool(this String theString, Boolean defaultValue = false)
    {
        Boolean value;
        var success = Boolean.TryParse(theString.ToLower(), out value);
        return success ? value : defaultValue;
    }
}
```

That's all you need to know about WURFL programming. What remains is figuring out the best way to use capabilities to arrange your views.

The following listing illustrates how to detect device capabilities in a controller method and pass the information down to the next view for actual display:

```
public ActionResult Details()
{
    var deviceInfo = Wurfl.Manager.GetDeviceForRequest(Request.UserAgent);

    ViewBag.UserAgent = deviceInfo.UserAgent;
    ViewBag.OS = String.Format("{0} {1}",
             deviceInfo.GetCapability("device_os"),
             deviceInfo.GetCapability("device_os_version"));
    ViewBag.Browser = String.Format("{0} {1} {2}",
             deviceInfo.GetCapability("brand_name"),
             deviceInfo.GetCapability("marketing_name"),
             deviceInfo.GetCapability("model_name"));
    ViewBag.IsMobile = deviceInfo.GetCapability("is_wireless_device").ToBool();
```

```
    ViewBag.IsTablet = deviceInfo.GetCapability("is_tablet").ToBool();
    ViewBag.SupportTables = deviceInfo.GetCapability("xhtml_table_support").ToBool();
    ViewBag.PointingMethod = deviceInfo.GetCapability("pointing_method");
    ViewBag.ScreenSize = String.Format("{0} x {1}",
             deviceInfo.GetCapability("resolution_width"),
             deviceInfo.GetCapability("resolution_height"));
    ViewBag.SupportAjax = !deviceInfo.GetCapability("ajax_xhr_type").NoneOrEmpty();
    ViewBag.PreferredMarkup = deviceInfo.GetCapability("preferred_markup");
    return View();
}
```

Figure 6-4 shows the site in action on an Android HTC Desire device.

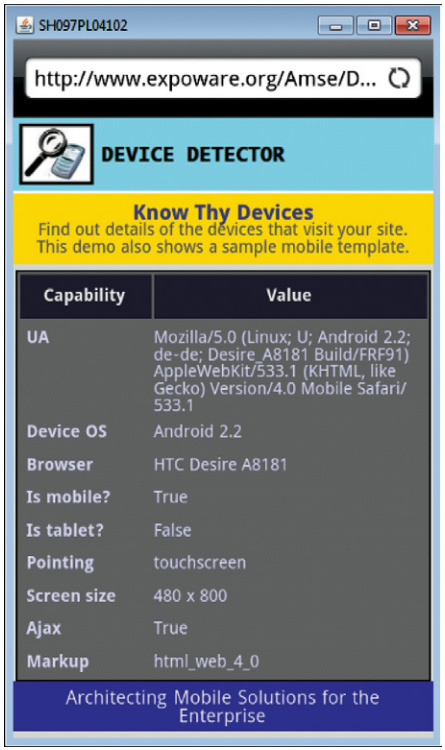



**FIGURE 6-4** Detecting device capabilities with WURFL.

The sample application is available online at *http://www.expoware.org/mobile*. I invite you to visit the site with your mobile device to experience live the power of WURFL.

## Implementing a Multiserving Approach

A DDR platform gives you the power to grab a lot of information about the device and the browser that is placing the HTTP request. In this way, you can make a decision about which content to serve—whether rich HTML5 markup, Ajax-intensive script, or plain HTML, with more or less graphics and multimedia content.

This section first summarizes the key aspects of a mobile site that you want to optimize for each request and then proceeds to present an example of multiserving—deciding which view to serve based on detected capabilities.

## Key Aspects of Mobile Views

Developers of a mobile site that aims at reaching the widest possible audience should consider carefully the implementation of a few different views for each page. Note that this isn't just about a desktop site that morphs into a mobile site. This is about a parent site—whether mobile or desktop—that serves multiple versions of the same logical page, one per each device profile that it intends to support.

Device-specific versions of the same logical page differ for a number of aspects beyond markup: image transcoding, viewport, and various ancillary forms of optimization.

**Note** By saying "the same logical page," I definitely hint at the same Index.cshtml page, which may look different when viewed on an iPhone versus a cheap and old-fashioned Nokia device. However, this only represents the most common scenario. It may not simply be a matter of reorganizing markup and images; sometimes it also may be a matter of logic and data being processed by the various views of the same page. In ASP.NET MVC jargon, you may need to face the need to serve not just different view templates, but also different view models for the same action.

### Optimizing the Content Rendered

While creating a view for a device profile, you also might need to check other rendering capabilities to compress the markup and decrease data traffic.

This entails looking at whether or not CSS files can be merged and if the device supports CSS sprites. Also, gradients can be rendered as CSS instructions instead of using background images. Image inlining is another great example of little tweaks that you may apply to the markup to speed up the download and rendering time. Video, sound, Flash content, and cookies may be subject to further checks to avoid an unpleasant experience for some of your users.

### Resizing Images

Even on powerful smartphones, reducing the size and number of images is never a bad thing. There's nothing fancy about resizing an image, but it would be great if the task could be automated to some extent. An approach you might want to consider is creating your own HTTP handler that serves as a centralized dispatcher of image files and properly resizes and converts them based on the capabilities of the device.

Details of the implementation are up to you; for example, you can opt for dynamic resize and caching or you can let the handler pick one of a few predefined images. In general, this is a very compelling service that commercial frameworks could offer to mobile developers.

**Note** When talking images and their sizes, it is not a secondary point to recall that a pixel is not always a pixel in mobile environments due to *dots-per-inch* (DPI) issues. Currently, there are pixel ratios of 1, 1.5, 2, and even 2.25. This is relevant info for images, icons, and other graphics on the site.

### Viewport Control

In any mobile page today, it is common to set the viewport. As mentioned, the viewport indicates—to the browsers that support it—the effective size to use for the browser window. In other words, a viewport is a request for the browser to not use any virtual window to host the content, but to stick to a specific width, which can be either relative or absolute. Here's a common way to set the viewport in an HTML page:

```
<meta name="viewport"
      content="width=device-width; initial-scale=1.0; maximum-scale=1.0; user-scalable=0;" />
```

It is key to remember that not all browsers support the viewport, especially those on older devices. If you're defining a view for very basic devices, make sure you check the viewport capability as well. In WURFL, it is expressed through the *viewport_supported* capability and a few others in the *html_ui* group.

## Creating Device Profiles

To implement a multiserving approach, you need well-defined rules to partition the potentially large number of different devices, a DDR platform to help in the implementation of such rules, and some code artifacts to keep device detection and page routing smooth and effective.

### Managing Segmentation

There's not just one way to partition the full range of devices into classes. Each application can have good reasons to define its own set of classes. This may depend on the results of some analysis that showed how potential users tend to reach the site; or it may be inspired heavily by characteristics of the application itself. Flexibility is key, and each team knows very well what would be the ideal segmentation for client devices.

Rules indicate conditions that a device should meet to be admitted into a class and be served a certain type of markup. So you first define your classes and grossly define which devices fall into each; next, you define more formal rules and implement them against a DDR.

A simple but effective example of device segmentation can be the following: one class reserved to iPhone and iPod Touch, one for all other smartphones, and perhaps one more for tablets. All

remaining devices will get a plain old XHTML page. Nothing prevents you from adding more or different profiles to best address the needs of, say, Windows Phone users. In general, however, the number of different device profiles you end up with never should exceed five. Beyond this threshold, the effort may be excessive; and if you really need to distinguish, say, iPhone users from Android users, you're probably better off going with a native application.

**Note** Before you decide to create distinct native applications for a few platforms, you also can try skinning the site differently for iOS (and perhaps Android) using ad hoc CSS files. WURFL helps in detecting CSS support and OS information for the device.

### Rules for a Device Profile

Each device profile is identified by a bunch of rules. Rules are expressed in terms of capability values. For example, one rule can be expressed as "all devices that support touch." Another can be "all devices larger than 240 pixels."

Not all the capabilities for WURFL—one of the largest repositories of device data—discussed so far in this chapter are necessarily good for creating rules to classify devices. General parameters to consider are the width of the screen, touch and WiFi support, Ajax and JavaScript, the level of HTML, and CSS support. Some of these capabilities may be inferred by looking at the operating system; others should be checked directly. For example, if the operating system is, say, iPhone OS or Android (2.0 or later), you can assume excellent HTML capabilities that are nearly at the same level as a desktop browser.

In any case, to implement rules and segmentation, you need to employ a reliable and accurate DDR platform.

## Device Profiles in Action

As an example, let's consider a scenario in which we have two profiles—smartphones and plain mobile devices. Even though the number of profiles is fairly small, one of the profiles is quite hard to define. In fact, what's your definition of a smartphone?

### The Smartphone Profile

Any attempt of providing a formal definition of a smartphone risks being largely arbitrary. Yet, smartphones are a category of devices that produce the most mobile traffic and will be used more and more. In terms of software and capabilities, a smartphone can be defined as follows:

- A device that has touch capabilities
- A device that is at least 300 pixels wide
- A device that runs the iPhone OS or Android 2.0

This definition probably shows the way to go but, admittedly, it is arbitrary and incomplete, especially as far as the list of operating systems is concerned. You might want to add Windows Phone 7.5, RIM OS 6, and some version of the Symbian OS to the list.

In our example, devices with the aforementioned capabilities will make it to the smartphone class. Any other mobile devices will make it to another class that we use for all other devices. Everything else is not mobile and will be treated as desktop. The following code shows a possible algorithm to detect a smartphone. The algorithm is based on WURFL:

```
private static Boolean IsSmartphone(IDevice device)
{
    // Must be wireless
    if (!device.IsWireless())
        return false;

    // Must be touch
    if (!device.IsTouch())
        return false;

    // Must be 320px wide or more
    if (device.Width() < 300)
        return false;

    // Must be Android 2+ or iPhone OS
    if (device.HasOs("android", new Version(2, 0)))
        return true;
    if (device.HasOs("iphone os"))
        return true;

    return false;
}
```

Note that the implementation uses C# extension methods to enhance code readability. The *IDevice* WURFL type doesn't have any of the members of the code snippet. All of them are plain wrappers around the *GetCapability* member of the *IDevice* type. Here's the implementation of *IsTouch*:

```
public static Boolean IsTouch(this IDevice device)
{
    return device.GetCapability("pointing_method").Equals("touchscreen");
}
```

Even more interesting is the implementation of *HasOs,* which uses the .NET Framework Version class to work around WURFL strings expressing a version number:

```
public static Boolean HasOs(this IDevice device, String os)
{
    return HasOs(device, os, new Version(0, 0));
}

public static Boolean HasOs(this IDevice device, String os, Version version)
{
    // Check OS
    var deviceOs = device.GetCapability("device_os");
    if (!deviceOs.Equals(os, StringComparison.InvariantCultureIgnoreCase))
        return false;
```

```
    // Check OS version
    var deviceOsVersion = device.GetCapability("device_os_version");
    Version detectedVersion;
    var success = Version.TryParse(deviceOsVersion, out detectedVersion);
    if (!success)
        return false;
    return detectedVersion.CompareTo(version) >= 0;
}

public static Boolean HasOs(this IDevice device, String os, String version)
{
     // Check OS
     var deviceOs = device.GetCapability("device_os");
     if (!deviceOs.Equals(os, StringComparison.InvariantCultureIgnoreCase))
          return false;

     // Check OS version
     var deviceOsVersion = device.GetCapability("device_os_version");
     return deviceOsVersion.Equals(version, StringComparison.InvariantCultureIgnoreCase);
}
```

In WURFL, the version of an operating system also can be expressed as a string (e.g., Symbian Anna). The final overload for the *HasOs* method just covers this case.

In an analogous manner, you can define additional *IsXxx* functions to define other classes. For example, *IsMobile* can be as simple as shown here:

```
private static Boolean IsMobile(IDevice device)
{
    // Must be wireless
    return device.IsWireless();
}
```

The *IsWireless* extension method just checks the *is_wireless_device* capability of WURFL.

**Important** When it comes to defining class profiles to partition the entire set of mobile devices that you face, be ready to deal with priorities too. It can likely happen that a UA string matches multiple rules. Which one would you pick up? Think of tablets. They're mobile; they're probably smartphones (according to the previous definition), and they can display the full website easily. Either you provide true partitions (e.g., negate the *is_tablet* WURFL capability for members of the smartphone group to distinguish from tablets) or define a priority rule as well.

## A DDR-Based ASP.NET Routing System

Implementing segmentation rules is clearly a developer's responsibility. This means that during the processing of the request, right before ordering a HTML template to render the markup, the code goes through a list of conditional statements and looks at known capabilities that the DDR of choice associates with the current UA and other HTTP headers. At the end of the workflow, the code knows the name of the HTML template that best fits the requesting device.

Trying to be more specific, where exactly would these conditional statements go? You have basically two options: create the markup on the fly (not especially easy to do with ASP.NET Web Forms using default controls) or redirect the request to a mobile landing page. In ASP.NET Web Forms, this can be achieved by installing an HTTP module that intercepts incoming requests for ASP.NET pages, analyzes the UA, and rewrites the URL to a page that best serves the device. Here's some code to illustrate the point:

```
public class WurflHttpModule : IHttpModule
{
    public void Init(HttpApplication app)
    {
        app.BeginRequest += OnBeginRequest;
    }

    private void OnBeginRequest(Object source, EventArgs e)
    {
        var app = (HttpApplication) source;
        var context = application.Context;
        var page = ApplyRoutingWorkflowForMobile(context.Request);
        context.RewritePath(page);
    }

    private String ApplyRoutingWorkflowForMobile(HttpRequest request)
    {
        var originalPath = HttpContext.Current.Request.Path.ToLower();

        // Get device information from the DDR (WURFL in this case)
        var device = Wurfl.Manager.GetDeviceForRequest(request);
        if (device.IsSmartphone())
            return GetLandingPageFor(originalPath, "smartphone");
        ...
        return originalPath;
    }
}
```

The *GetLandingPageFor* method contains the application-specific logic that determines the page to serve for the application-specific profile named *smartphone*. It goes without saying that the logic in *ApplyRoutingWorkflowForMobile* can be customized flexibly.

## A DDR-Based ASP.NET View Engine

As was shown in Chapter 4 and in the previous example in this chapter, when you use ASP.NET MVC, things are simpler because of the neat separation existing in the framework between the processing logic of the page and the rendering of the view. The previous approach, in fact, poses some development issues in the sense that it assumes that each possible mobile page is a stand-alone page with its own processing and rendering logic. In simple scenarios when the data being represented is the same and all that changes is the markup, you are exposed to the risk of code duplication.

In ASP.NET MVC, controllers process the request and generate the data for the view; the view is then picked up as a parameter. The net effect is that you just need to fork the code that, in the same regular request processing, selects the next view. Simple scenarios are greatly simplified and even can be automated by using some convention-over-configuration. For example, you may decide that the

default version of the site's home page (whether mobile or desktop) is provided by *index.cshtml*. Ad hoc views resulting from multiserving follow the naming convention of *index.xxx.cshtml,* where *xxx* is the name that you assign to your device profiles. For example, if you detect a smartphone, you then serve a view called *index.smartphone.cshtml*. If such a view is not defined, you fall back to the default view, *index.cshtml*.

I'd code this algorithm via a DDR-powered custom view engine written along the following skeleton:

```
public class AmseRazorViewEngine : RazorViewEngine
{
    private readonly ViewResolverBase _resolver;

    public AmseRazorViewEngine() : this(new DefaultWurflViewResolver())
    {
    }
    public AmseRazorViewEngine(ViewResolverBase resolver)
    {
        _resolver = resolver;
    }

    protected override IView CreateView(ControllerContext controllerContext,
              String viewPath, String masterPath)
    {
        var resolvedViewPath = viewPath;
        var resolvedMasterPath = masterPath;

        if (!String.IsNullOrEmpty(viewPath))
            resolvedViewPath = _resolver.GetName(controllerContext.RequestContext, viewPath);
        if (!String.IsNullOrEmpty(masterPath))
            resolvedMasterPath = _resolver.GetName(controllerContext.RequestContext,
            masterPath);

        return base.CreateView(controllerContext, resolvedViewPath, resolvedMasterPath);
    }

    ...
}
```

The *ViewResolverBase* class represents the base class for components that transform the originally requested view name into a different name according to some logic—in this specific example, the device capabilities as detected by WURFL. Here's an excerpt from the WURFL resolver:

```
public override String GetName(RequestContext context, String viewName)
{
     var mobileView = TransformViewName(context.HttpContext.Request.UserAgent, viewName);
     return VirtualFileExists(context, mobileView) ? mobileView : viewName;
}
private String TransformViewName(String userAgent, String view)
{
     var device = Wurfl.Manager.GetDeviceForRequest(userAgent);
     if (device.IsSmartphone())
         return ModifyViewNameFor("smartphone");    // Some helper code here
```

```
    ...
    return view;
}
```

All in all, it is always a good thing to have quick-and-easy solutions for everything. Not that everything is easy in software, and quick-and-easy solutions are such just because they make assumptions and restrictions. So long as you know the underlying mechanics and which assumptions and restrictions are applied, you can enjoy quicker development without losing the flexibility that allows you to take tougher challenges.

So in the end, we have a website with three flavors: desktop, smartphone, and plain mobile. Figures 6-5 and 6-6 show the site on a smartphone (at least according to the previous and entirely arbitrary definition) and a plain mobile device. For the purpose of illustration, I'm considering Windows Phone 7 to be a plain device, not a smartphone. This has been done only for the sake of the demo!

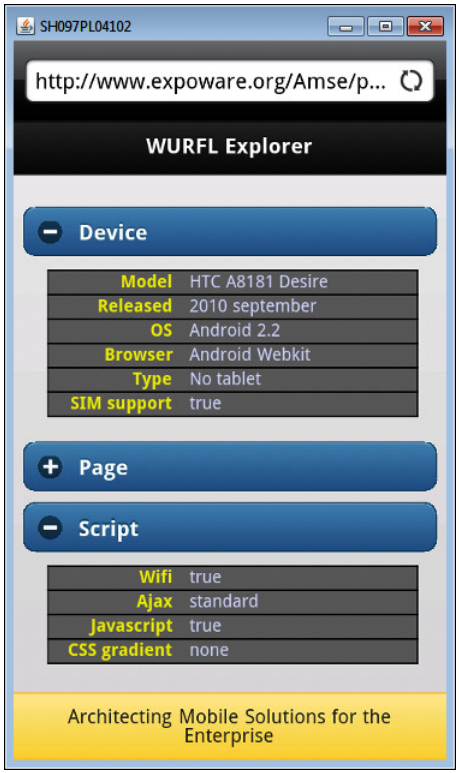



**FIGURE 6-5** The WURFL Peek site on a device in the *smartphone* profile.

You can try the WURFL Peek site yourself by using your own device to visit *http://www.expoware.org/mobile*.

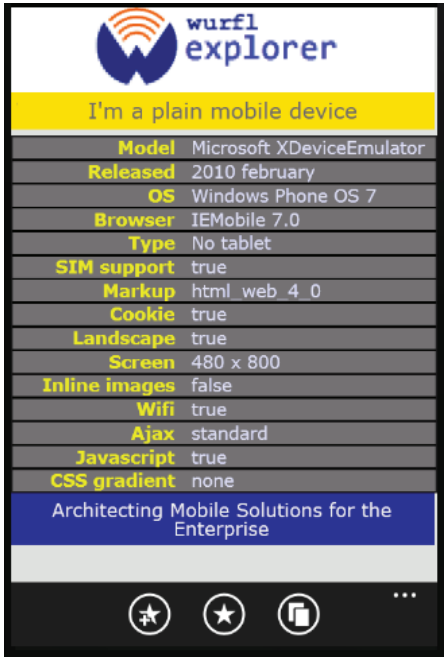



**FIGURE 6-6** The WURFL Peek site on a device in the *mobile* profile.

## The Role of jQuery Mobile

As you may have figured out from the visual themes, the smartphone view is extensively based on jQuery Mobile. I provided an introduction to jQuery Mobile in Chapter 5, "HTML5 and jQuery Mobile." The library does a great job of letting you create effective mobile user interfaces and also offers automatic fallback in case of older browsers. The library does use a lot of script and CSS facilities that—while common overall on most mobile browsers—may still produce a nonoptimal rendering on older browsers. That's precisely the point I want to make here.

The jQuery Mobile library bases its decision about whether a required implementation may work on the requesting browser solely on the information that it gets directly from the browser or that it can detect programmatically. As we've seen, the number of these capabilities can be measured in the order of tens; a server-side DDR is at least one order of magnitude more powerful.

Not all mobile solutions need a DDR, but what a DDR can do in the context of multiserving is much more than what jQuery Mobile (and other JavaScript libraries) can do.

Finally, consider that you have limited control over how fallback happens. The free plain HTML user interface that you get in this case may not suit you. And more, the jQuery Mobile library is fairly small in size (24 KB of minified/gzipped script, plus 7 KB of minified/gzipped CSS, plus another 10 KB of images), but it also requires the full jQuery library, which takes up another 30 KB (minified and gzipped). All this content also will be downloaded (but probably cached) for a trivial HTML page anyway. My suggestion is that you consider jQuery Mobile for rich devices, and use your own HTML pages for falling back.

**Note** As far as I can tell, no frameworks today support automatic, convention-based routing of pages to mobile devices. A notable exception is ASP.NET MVC 4. The framework supports the *mobile* suffix to identify the mobile display mode for a view requested by a mobile device (e.g., *index.mobile.cshtml*). ASP.NET MVC 4 also allows you to define dynamic rules to create custom display modes. ASP.NET MVC 4 doesn't prescribe a DDR—it only offers the largely insufficient native ASP.NET browser capabilities provider—but WURFL can be used easily to create custom rules and custom display modes such as *index.smartphone.cshtml*.

### RWD Plus Server Side Components (RESS)

So far in this book, I have advocated the idea that content for a mobile site can be arranged entirely on the client or entirely on the server. Arranging a mobile presentation on the client requires a lot of CSS and some JavaScript (clever) tricks—and more important, it works by hiding, resizing, and repositioning downloaded content. This is RWD, and, among other things, it relies on browsers to support CSS media queries.

Next, I discussed the server-side approach, where you rely on a DDR to tell you as much as possible about the known capabilities of the requesting browser. Based on that, you intelligently serve the best content. A server-side approach is—in my opinion—more powerful as it gives you the great chance to download only relevant content and also organized in the most appropriate layout.

There's also a kind of middle way, though, that's based on a combined use of RWD on the client and a server-side DDR. Luke Wroblewski called this approach RESS (short for REsponsive design plus Server Side components). You might want to read his post at *http://www.lukew.com/ff/entry.asp?1392*.

At its core, an RESS solution is based on a single set of master pages written for RWD. Therefore, these pages can resize properly and lay out their content to a different structure based on the screen size. At the same time, the content of these pages is not the same regardless of the browser. Such pages contain areas whose actual content is determined by looking at the browser capabilities. The benefit is that you distribute the logic for adapting the layout between server and client; and on the client, this logic is mostly native browser logic. In addition, you have a single application for desktop and mobile browsers and a single set of links to deal with.

RESS is a new field, and it's not completely clear whether it is a brilliant new idea or simply yet another attempt to build effective mobile pages. An excellent and pragmatic perspective of RESS and responsive sites can be found at *http://www.slideshare.net/yiibu/pragmatic-responsive-design*.

## Summary

We all want sites to be capable of adapting to the actual browser. This means desktop browsers with a restricted window, as well as browsers whose rendering area is smaller. All mobile devices fall in the second group.

A hot question being debated these days is whether you can afford to create a single website with such a smart design that can handle browsers of any size effectively. This is the promise of RWD (as a design approach) and also the promise of CSS media queries (as a more concrete technology).

With mobile sites, it is essential to minimize the amount of data being transferred. This includes CSS, script, and image files. Most of the time, RWD performs its magic by simply hiding DOM elements. This means that the memory footprint of the page remains quite large, which is another problem in itself given the limited processing power of mobile devices. Furthermore, as it happens, some resources are downloaded, but not to be used! While some tricks can be applied to download smaller images, that often comes at the extra cost of downloading additional scripts and CSS style sheets. Downloading extra stuff that is not even used is a luxury that nobody can afford in the mobile space.

So if a single "liquid" design may not be affordable for mobile scenarios, what would be a valid alternative? You use a server-side DDR to match the requesting device to a known profile and figure out effective capabilities. Based on that information, you then intelligently serve the most appropriate markup and create as many levels of segmentation as you think you need.

WURFL is the most popular DDR today. It brings a catalogue of mobile device profiles and a cross-platform API that spans ASP.NET, PHP, and Java. WURFL counts in its database over 15,000 unique device profiles (corresponding to many more real devices) and 500+ capabilities.

WURFL and RWD share the same goal, but RWD is limited to a few design-related capabilities (device width/height, screen width/height, color depth, resolution, and orientation). WURFL has more than 500 capabilities, most of which are also design-related. In mobile solutions, you need to understand the capabilities of the device and decide what to do. It's not simply a matter of changing the layout; it's often also a matter of logic and data.

PART III

# Mobile Applications

CHAPTER 7

# Patterns of Mobile Application Development

*Life is what happens while you are busy making other plans.*

*—John Lennon*

**In this chapter:**

- Mobile Applications Are Different
- Patterns for Interaction
- Patterns for Presentation
- Behavioral Patterns
- Summary

In this chapter, we begin a tour of the exciting world of mobile applications. A mobile application is not a website optimized for mobile devices. It is, instead, a relatively small and compact application natively created for the operating system available on the device. The application is often a self-contained application installed on the device, possibly interacting with sensors and local hardware and connecting to the outside world. Mobile applications are one of the key reasons for today's growing smartphone market, and it is probably the most considered option when a company starts thinking of establishing a mobile presence.

Mobile applications are written for a specific version of a specific operating system. Nearly any major vendor of mobile devices has its own operating system and related software development kit (SDK). Writing mobile applications for the iPhone/iPad platform is not the same as writing for the Android platform, and writing for Windows Phone is different than writing for BlackBerry, Symbian, or Bada. The list of mobile platforms is not endless, but it definitely counts a few items that architects and developers can't just ignore.

It turns out that writing a mobile application is a development experience that may vary significantly depending on the platform of choice and the skills that one may have. For example, if your team has a strong Java background, then writing for Android and BlackBerry is not a huge problem. The same can be said if you come from years of .NET development and face the task of building Windows Phone applications.

All mobile applications, however, share a common set of characteristics that make them fairly unique. This chapter discusses a specific set of design patterns and ad hoc coding practices that may apply to mobile applications.

**Note** Not all the patterns presented in the rest of this chapter necessarily apply to all mobile applications. However, it's also unlikely that *none* of the following patterns apply, either. More important, these patterns apply on a per-function basis rather than on per-platform basis.

# Mobile Applications Are Different

Mobile applications are relatively simple applications that perform a specific task for the user. Such applications must be installed on the device to perform their tasks, sometimes interacting with the device hardware, including camera, storage, and global positioning satellite (GPS) sensor. Mobile applications also can be used to build user-specific workflows by orchestrating pieces of custom logic and native components such as the Short Message Service (SMS) subsystem, the default email program, and the preinstalled browser.

Mobile applications are still software applications, so any well-known practices of good design and development apply. However, mobile applications run on mobile devices, and such devices are significantly different from laptops and desktop computers.

## Critical Aspects of Mobile Software

Mobile software is software designed to run only on very special devices. This apparently simple fact imposes a number of constraints on developers and raises a few issues for which new patterns and practices are required.

### The Interaction Model

A key difference between desktop and mobile applications is that users of modern mobile applications likely will use a finger to point to any available content. Using a finger to select a menu item is quite natural and a pleasant experience for the user, but not if the clickable item is too small.

A finger, in fact, will never be as precise as a mouse pointer or a stylus can be. What about typing text, then? Whether the device has a hardware keyboard or relies on some software emulation, typing on mobile devices is definitely problematic. If the keyboard is provided via software, then it ends up hiding a large part of the existing user interface. Sometimes it happens that the keyboard covers some of the buttons that users may tap to confirm the operation. All of this is annoying and should be addressed at the design level by keeping usability practices in mind.

## The Presentation Model

Touch-sensitive screens force you to increase the size of any clickable content—primarily buttons and links. Unfortunately, you cannot simply make each button and link larger—the overall size of the screen of a mobile device is much smaller than the screen of a laptop.

The combined effect of these two factors leads to a complete rethinking of the user interface of a mobile application. A mobile application is designed around a few very specific use-cases, each of which finds its trigger on the home screen. In the implementation of each use-case, then the number of steps that require user interaction should be kept to a minimum.

Presentation of the data is also subject to new rules. Scrolling is highly encouraged, and it is done vertically for the most part. However, forms of horizontal scrolling are coming up, especially in Windows Phone with the panorama view (see Chapter 10, "Developing for Windows Phone"). This is a major and notable difference with desktop and web applications. A widely agreed guideline says that horizontal scrolling should be avoided altogether on webpages and minimized in desktop user interfaces. In addition, in mobile, the horizontal swipe gesture is an accepted way to flip through distinct but related blocks of content. While a swipe is just a gesture to introduce new but related content, overall, it transmits the idea that you can navigate through content by flipping through pages. As a developer, swiping is an excellent way to save valuable screen space while not sacrificing a sense of continuity in the presentation experience.

## The Behavior of the Application

Screen real estate is not the only resource that is limited on a mobile device. Processing power—namely, the power of the CPU and the associated graphics processing unit (GPU)—is not comparable to what is typical in desktop computing; therefore, it is used for memory and local storage. In particular, some devices also prevent (or at least complicate) the act of mounting an extra SD memory card onto the device.

This means that memory consumption should be kept under strict control, and optimizing algorithms is more important than ever. Furthermore, for its inherent nature, a mobile device is interruptible and is utilized by users in a way that requires a strong multitasking logic. Multitasking, however, conflicts with a lack of processing power and memory. The net effect is that all applications should be created from the ground up around the idea that they can be put in a rest state and even removed from memory when the system reclaims their resources.

Finally, mobile applications run on battery power and are subject to the quirks of connectivity. Because the user might be traveling when she deals with the device, the connectivity can come and go at any time, and it also might change its quality as the user moves from a WiFi area to 3G roaming.

The actual behavior of mobile applications, therefore, depends on a variety of issues that most developers never faced before.

### Security Concerns for Mobile Software

In the past 15 years of web development, architects and developers have learned a lot about threats to the security of data and applications and how to combat them. In particular, we've learned that it is essential to use long passwords made of both capital and lowercase letters, numbers, and symbols.

Try typing such a strong password on a mobile device. You have to switch multiple times between different keyboard layouts; in the end, all users would opt for a simpler password, and the application developers would opt for number-only PINs. All this is effective, but it reduces the level of security.

Also, consider that a device is much easier to steal than a laptop and is also far easier to lose. In addition, a mobile device often is lent to other people, if only for something simple and quick like a phone call. All these common behaviors increase the risk of attacks and data loss.

The challenge is finding a good mix of measures that keep security at a high enough level without making users' lives significantly harder. Mobile security, however, is a whole new field of research in much the same way as web security was in the 1990s.

## New Patterns and Practices

Mobile applications pose new challenges and require a new set of patterns and practices. Such design patterns are necessary to drive the development of mobile applications and keep them effective and aligned with the needs and expectations of people.

Mobile design patterns apply regardless of the actual operating system and development platform. Three main areas of programming are affected by patterns because of their relevance: application life cycle, storage and connectivity.

### Application Life Cycle

In a desktop scenario, the user commonly starts and stops applications. The termination of a running program, therefore, is a deterministic event. A mobile operating system is a totally different environment for programs. In a mobile environment, a user is only allowed to start an application—the system will be the one to stop the application. Borrowing from Java- and C#-specific terminology, I would even say that in a mobile environment, all running applications are at some point "garbage-collected" by the operating system scheduler and terminated as appropriate.

This sort of application garbage collector will operate on application instances that have been started once but are no longer interacting with the user in the foreground. These application instances are referred to as *background applications*. What background applications are allowed to do may vary across different mobile operating systems like iOS, Android, and Windows Phone. In general, while mobile operating systems seem to run multiple applications at the same time, that's mostly a trick played to keep users happy. In a mobile scenario, truly unlimited multitasking (where multiple applications are alive and kicking at the same time) is just not affordable. Limited memory and excessive pressure on the CPU would run the battery down, reducing the power for foreground applications as well.

On mobile devices, applications swing between three possible states: actively running, paused with some pending work, or paused waiting to be resumed or unloaded. The various mobile operating systems differ for the actual implementation of these states, but the bottom line is that any mobile application is designed to a different expectation than classic desktop and web applications (even web applications run on a mobile device).

## Tools for Data Storage

Mobile applications are fundamentally conceived to be stand-alone applications. This certainly doesn't mean that mobile applications are totally disconnected from one another; instead, it means that mobile applications require special tricks and application programming interfaces (APIs) to share data. In all mobile platforms, applications are allowed to persist data locally to the device. This can usually be done in a number of ways—via custom files, local databases, or system-managed data repositories such as the *Settings* bundle facility of iOS, *ApplicationSettings* in Windows Phone, *SharedPreferences* in Android, and *PersistentStore* object in BlackBerry.

Custom files give you the maximum flexibility because they mostly offer a stream-based programming interface and can receive any serializable in-memory object. However, mobile applications sometimes are assigned a specific section of the file system, and their persistent data is well isolated from the others. In Android and BlackBerry, the developers can control the visibility of the files and can access an external SD card programmatically. In Windows Phone and iOS, sharing persistent data between applications requires tricks. In these platforms, however, it seems that the way to share persistent content is the cloud—the iCloud platform for iOS and SkyDrive for Windows Phone.

Local databases are still changing and evolving. A popular choice seems to be SQLite (see *http://sqlite.org* for general information), a transactional SQL database engine that requires little or no administration and lends itself very well to be hosted in devices as an application-specific local database. SQLite is a widely usable, cross-platform solution. In Windows Phone, you also have the option of Microsoft SQL Server Compact Edition (SQL CE) and the Sync Framework for synchronizing local and remote databases. SQL Anywhere from Sybase is an excellent and award-winning commercial product for syncing up mobile and remote data on a variety of platforms, including iPhone, Android, BlackBerry, and Windows Mobile.

Just the ability to synchronize device data with the enterprise server is a key factor for mobile applications. It is especially crucial for line-of-business mobile applications that employ a local cache when the network is down and smoothly sync up with the remote server whenever the network becomes available again. For this specific problem, another class of solutions may appear soon on the mobile horizon—mobile NoSQL solutions.

Put this way, the connection between NoSQL and mobile may be hard to find, as a mobile application can hardly be considered a data-intensive application. Nonetheless, NoSQL is a boldly emerging approach in the mobile space. A NoSQL mobile platform is merely the mobile edition—hosted right in the device—of a NoSQL database. This allows you to store plain objects locally regardless of table schemas and relying on typical NoSQL query operations for retrieving saved objects. In a NoSQL scenario, queries are expressed in terms of *map/reduce* operations, where the

*map* operation selects the item of interest (e.g., the WHERE clause of a classic query) and the *reduce* operation receives a mapped object and performs any required work on it. In addition, you'll have a server NoSQL component living in your datacenter that you can sync up with.

**Note** The term *NoSQL* refers to a relatively new class of storage systems that do not use SQL as the query language and may not require fixed schemas in their repositories. NoSQL systems also differ from classic relational systems for the lack of JOIN operations and incomplete support for atomicity, consistency, isolation, and durability (ACID)—the pillars of the traditional, SQL-based relational model. Whereas typical relational solutions work best in frequent but small read/write transactions and large batch and mostly read-only transactions, NoSQL solutions can operate efficiently in scenarios where heavy read/write transactions are required. Their improved performance is due to the release of ACID constraints and fixed table schemas.

A good example of mobile NoSQL is CouchBase Mobile, which is the mobile edition of the increasingly popular CouchDB database. From a developer's perspective, CouchBase Mobile consists of an embedded web server and a JavaScript interpreter to process queries. In CouchDB, map/reduce operations are coded in JavaScript. Currently, CouchBase Mobile is available for Android and iOS.

## Connectivity

In general, not all mobile applications need connectivity. Games and personal utilities (e.g., a to-do list) may work locally forever without any problems. But when you wed mobile to business, connectivity becomes a crucial point. As mentioned, connectivity is frequently associated with data synchronization and occasionally connected functionalities. However, from a developer's standpoint, connectivity triggers a number of potential headaches that can be rooted in one common cause: mobile connectivity may not be reliable.

*Mobile* just means using a device while on the road, which by itself sufficiently explains why connectivity should not be assumed to be reliable. You can lose connectivity as you walk through a subway station, or it can be flaky if you are in a crowded shopping mall a few days before the holidays.

The bottom line is that the user interface and network-dependent operations should be designed and implemented according to specific patterns that take into account a bit of troubleshooting and try repeatedly before succeeding or giving up. If it's critical for the user interface, connectivity should be checked constantly—preferably via a background service. In this way, you can guarantee that any network state change is detected and user interface (UI) elements can be enabled or disabled in a timely fashion. Similarly, being connected and starting a network operation is no guarantee of success—the network can go down at any time. Clearly, a WiFi connection is more reliable and faster than a 3G connection on average, but far from making things simpler, this aspect adds yet another parameter—type of connectivity—to consider for the implementation of network tasks.

Let's explore now a few recommended patterns for design and implementation of features in mobile applications.

**Important** This chapter and the following patterns are devised primarily with mobile native applications in mind (e.g., Android, iPhone, and Windows Phone). However, some of the next considerations and suggestions for their generality apply to mobile web applications as well.

## Patterns for Interaction

The patterns in this section are related to the way in which the user interacts with the application and what the application should (or should not) do to make the user's life easier and offer a better user experience.

**Note** When it comes to patterns, names are extremely important. I'd even say that names are as important as the content they express. One of the major benefits of using patterns is that you can explain your idea of implementation by using common names. And those names are unambiguously associated with a triad given by a strategy, a list of actors, and a description of the role played by each actor in the strategy. Mobile patterns are a new field, so most of the names used here are new (and in some cases, even a bit arbitrary). In the near future, there hopefully will be more stabilization on both names and strategies.

### The Back-and-Save Pattern

We all see that youngsters are incredibly quick to type on the hard and soft keyboards of mobile phones and devices. They are this quick because they're used to tapping and—not a trivial point—because their fingers are tinier. Even when your application is created for a relatively young audience, a feature like word auto-completion (which saves users the effort of typing an entire word or phrase) is always welcome. This is great help in web and desktop applications, but it an even greater help in mobile environments, where typing on a keyboard is definitely a much harder task for everybody.

#### Formulating the Pattern

The core idea behind this pattern is to take any reasonable measures to minimize the typing effort of the user. This results from the combined effects of a few practices, one of which is automatically saving whatever the user has typed, regardless of how she leaves the current screen. The principle of the Back-and-Save pattern is here:

*Save the content of input screens when the user leaves (or is forced to leave) the screen.*

This pattern goes hand in hand with the Auto-Save pattern, which can be expressed as shown here:

*Save the content of input screens periodically.*

This is the same feature that you can enable in Microsoft Word that silently saves your current document after every given number of minutes (see Figure 7-1).


**FIGURE 7-1** The auto-save feature in Word.

Both Back-and-Save and Auto-Save express the same core idea—don't request an explicit command (i.e., a tap) from the user to save the data just entered. What's been just typed is a nontrivial effort the user made—don't throw it away because the user leaves the screen by hitting the device's Back or Search button.

Consider that sometimes the user taps Back because she simply doesn't see other ways of leaving the screen: losing typed data is never a pleasure for her (especially if it's taken great effort to type it). Or consider another typical situation: the application crashes. If a user still can retrieve input data from the previous session, he probably will be less angry with the application. Likewise, consider that especially in input forms, the limited screen real estate of devices can make it hard to squeeze out some extra space for a pair of Save/Clear buttons that keeps the overall user interface clean and clear. The convention, therefore, is to reverse the classic pattern of web and desktop applications—save regardless and clear/undo on demand, and only when that represents a valuable feature for the form. Figure 7-2 presents the Settings screen for the browser application on an Android smartphone. As you can see, there's no hint of a Save or an Undo button, and the visible state of the screen is what has been saved or will be saved upon exit.

## Implementation of the Pattern

The actual implementation of this pattern is subject to the specific facilities that you find on a given platform. You can use a timer to save at a given interval; you can detect when the screen is being unloaded and do your saving work. Finally, you also can opt for a save-as-you-go approach and permanently save data as the state of any given widgets in the user interface is altered. As an example, consider that the Android screen in Figure 7-2 is coded using a built-in component—the *PreferenceActivity* class—that detects changes in visual controls and automatically saves for you.

Figure 7-3, instead, shows a feature of a free iPhone application (Postino) that takes photos and lets you send them as physical postcards. At some point, you can add a message to the postcard and even draw your signature. There's no need to do anything to save; but shaking the phone (like an Etch-a-Sketch) will clear any sign you've drawn. The Windows Phone version of Postino has a button on the application bar to let users clear the signature.

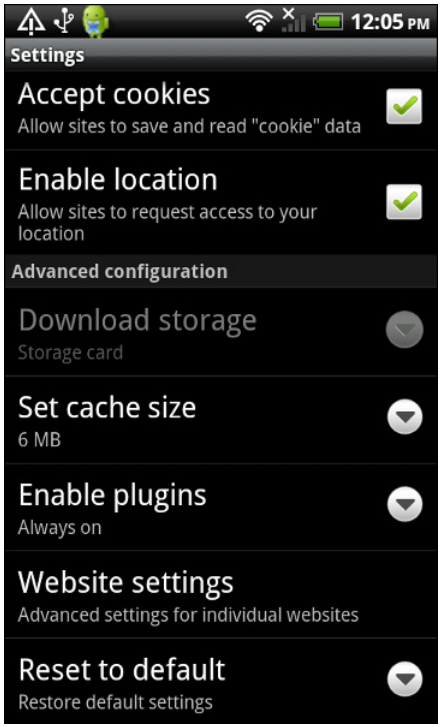



**FIGURE 7-2** Settings of an Android browser application: no buttons to save or undo.


**FIGURE 7-3** Back-and-Save in action in Postino for iPhone (left) and Postino for Windows Phone with an explicit command for undo.

**Note** Postino is available for iPhone, Android, and Windows Phone. For more information, visit *http://www.postinoapp.com*.

### Considerations for Mobile Data Entry

If the idea of a complete turnaround still sounds a bit harsh to you after years of programming with explicit save/undo commands, then consider that having a Save Confirmation dialog box has been treated as one of the big problems with current software by David Platt in his popular book *Why Software Sucks* (Addison-Wesley, 2007). In his book, David focuses on the user experience and uses a great analogy: A clerk will never ask you to confirm the groceries you've just put on the register belt—it's self-evident. Slightly rephrasing David's concepts, in mobile applications, the data entry experience should follow the paper-and-pen analogy: what's written stays there unless you actively decide to remove it. Shaking the device to undo/redo is a fairly common practice.

## The Guess-Don't-Ask Pattern

In addition to automatic saving, the data entry experience can and should be improved in any way. This entails, for example, making extensive use of hints and watermarks on text boxes, and in general, any smart form of interaction that can save space on the screen and help minimizing errors.

### Formulating the Pattern

Guess-Don't-Ask has more of a general principle than a classic pattern. It expresses the idea that the application should try to limit data entry as much as possible. If there's something that you, as a developer, can do to save your users a click or some typing, then by all means do that. The principle of the Guess-Don't-Ask pattern is below:

> *Use any available resources to make intelligent guesses and save users the largest possible bit of interaction.*

This pattern is all about offering suggestions to users and guessing what they would most likely do in a given scenario. Applications of this pattern range from configuring auto-completion in text boxes to providing tips on how to perform common tasks.

Another fairly common situation is filling in a form with address information. You might want to use the GPS service to reverse a location to an address via the Bing or Google Maps service. In this way, the address fields of the form can be filled automatically with a reasonable default value. For the *country* field, you also can try to guess it from the system locale or (perhaps more reliably) the keyboard layout.

Guessing some input data that the user is going to enter (and, likewise, doing whatever is in your power to predict user actions) bring a key benefit—these features give the perception that the software is taking care of the user. It is much more common than you may think that mobile

applications to be downloaded, used once, and then uninstalled or never used anymore. A little bit of guessing can really make the difference between a good application that is successful and a good application that is just downloaded, but without enthusiasm.

## Implementation of the Pattern

As mentioned, Guess-Don't-Ask is more a driving vector than an algorithm to implement at some specific point. It turns out that its implementation details are largely specific for each application and mostly left to the creativity and feeling of single developers. However, here's a list of features that represent good guesses at various stages of the application.

Input forms are definitely an excellent place where you can exercise your guessing ability to minimize input effort. You can prefill fields with geographical information and greatly simplify data entry by enabling word completion, facilitating voice-based input, and choosing the best-fitting input scope. Figure 7-4 shows the user interface of a classic grocery list application for Android that allows you to add items by speaking.

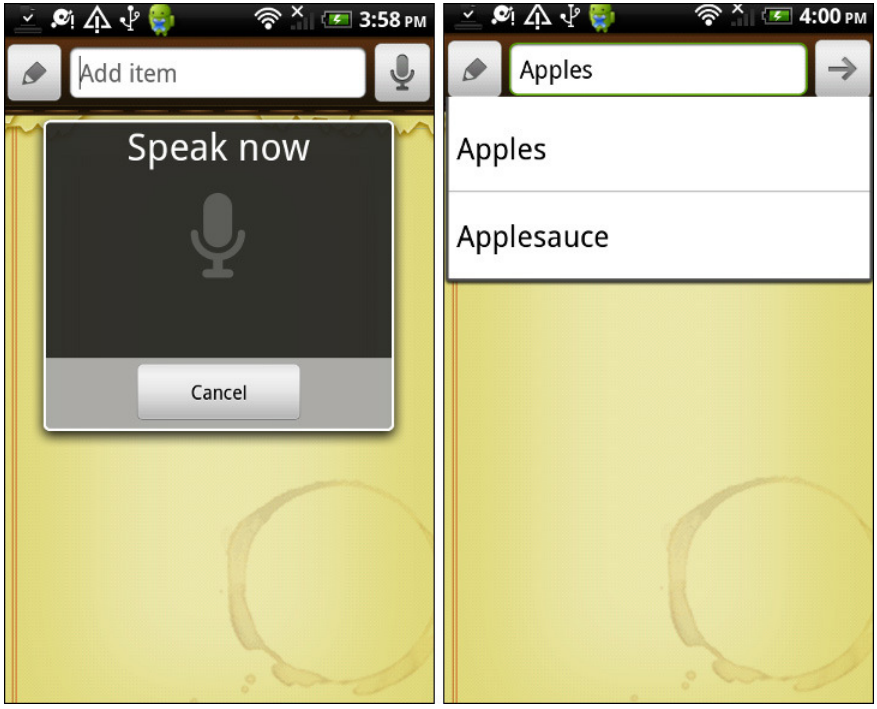



**FIGURE 7-4** A voice-enabled input form.

## Making Typing Easier

When it comes to typing, the input scope plays a fundamental role. The input scope defines the type of data being entered and is used to define the layout of the onscreen keyboard. On mobile devices, the onscreen (soft) keyboard for space constraints is necessarily limited to letters, numbers, or symbols.

However, depending on the specific data type that is expected in a text box, having a given character in the layout can save users a lot of switching and speed up the data entry. For example, Figure 7-5 shows the keyboard layout used by the default browser applications in Windows Phone and Android.


**FIGURE 7-5** Different keyboard layouts in the Windows Phone and Android browser applications.

As a developer, you can configure the keyboard layout programmatically to a large extent. It's not that you have to create a new layout from scratch every time; however, a well-made choice among the natively supported layouts can make a good deal of difference. Figure 7-6 shows a Windows Phone screenshot that is expected to contain a description. It may not look that much different from Figure 7-5, but in action, it looks very different. First, it starts with uppercase letters. Second, it allows the user to add a smile with a single click. Finally, it supports word completion (but this can't be guessed from the figure).

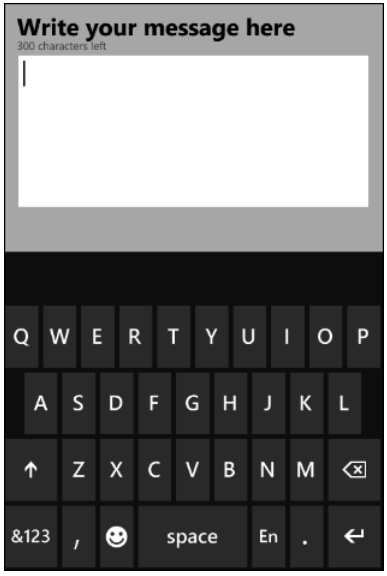



**FIGURE 7-6** A more specific keyboard for entering a description text in Windows Phone.

### If You Can't Guess, Then Remember

Guessing requires logic—sometimes smart logic. However, there's a much simpler yet effective way to save users some data entry—remembering last entries and preferences. Again, how you take advantage of this pattern largely depends on the form that you're writing. For example, Postino for iPhone remembers the address and description that the user entered for the latest postcard and uses those values as the default for the next one.

It should be noted that you don't want to use this strategy every time. It makes sense for Postino, because it is reasonable to expect that the user wants to send more postcards to people in the same city, or with a similar if not identical text. The same strategy, instead, might offer little value in other situations where the likelihood that the user has to clear everything before typing could be higher.

Sometimes, arranging statistics about how the user is dealing with the application can help make more precise guesses about what's appropriate to improve the user experience. As a general rule, however, saying that keeping track of the last action/selection and use that as a default choice is, on average, a rewarding approach.

Finally, if the data to enter can be taken from other storage places within the device (e.g., name or address of a contact, email, and phone number), then there's really no reason to do otherwise.

## The A-la-Carte-Menu Pattern

Older readers may recall that there was a time before the advent of Windows and pop-up menus that all applications (mostly MS-DOS applications) started with a relatively small list of choices, such as press 1 for this, press 2 for that, and so forth. One may argue that it was a totally different time and that applications were much simpler. That's right—but today's mobile applications are incomparably simpler than many desktop and web applications. More important, mobile applications (as well as mobile websites) are built around a few specific use-cases. And you should be able to make it clear to the user from the beginning which options are available.

### Formulating the Pattern

The core idea of the A-la-Carte-Menu pattern is to make any action quick and direct for users at any time. Users are almost never sitting comfortably when they use the application—they may be walking, waving for a taxi, stuck on a car or train, or maybe even eating or drinking. At any rate, immediacy is key in the mobile space.  The pattern can be summarized as:

> *At any time, it should be clear for the user which action to take and how many options she has.*

Options always should be limited, and any functions should be no more than two or three clicks away.

### Implementation of the Pattern

An effective implementation of this pattern deeply affects the usability and design of the application. It is essential that you go with well-selected and detailed use-cases. In general, a mobile application is a simple front end—the difference between great and not-so-great applications is mostly in the selection of use-cases.

This clarity of thought should be evident since the home screen of the application. Figure 7-7 provides two examples of what it means to list available choices in a crystal-clear way. The screenshot on the left is the home page of a mobile web application (*http://m.opentable.com*); the other is Facebook for Android.

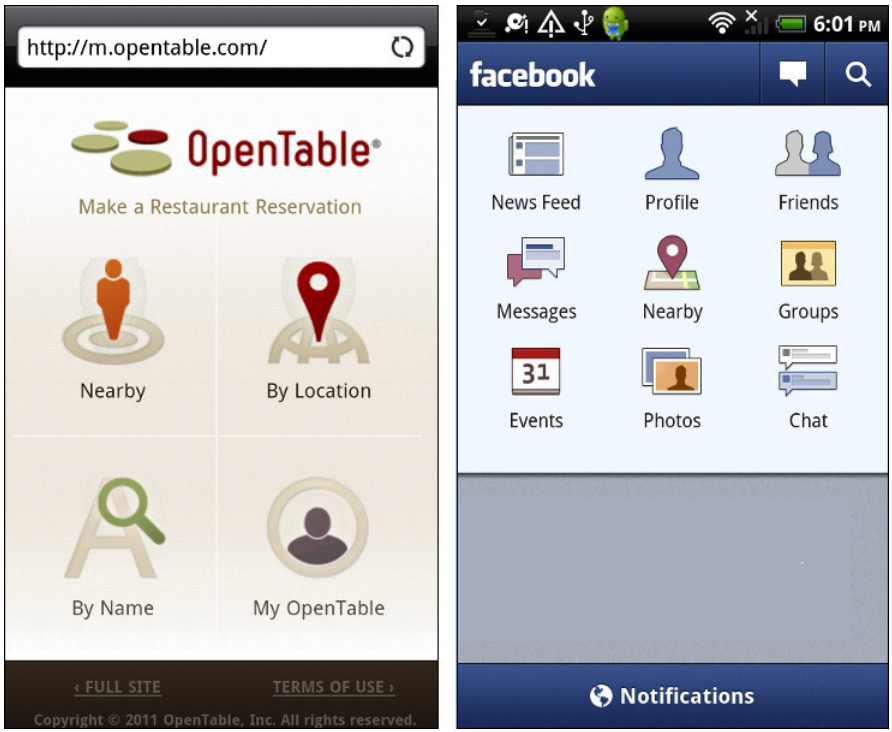



**FIGURE 7-7** Two examples of the A-la-Carte-Menu pattern.

The A-la-Carte-Menu pattern is often truly necessary because of touch devices—your fingers are often too big to select links in webpages or certain buttons in mobile applications.

In addition, this pattern also relates to the idea of hiding rather than disabling controls that are not going to be used by the user in a given context. This not only saves space but also contributes to keeping the user interface clean and clear.

## The Sink-or-Async Pattern

It is widely agreed that it is a good idea for developers to opt for asynchronous implementations of potentially long operations. This is a universal principle of software design—it's not specifically targeted to mobile software. However, in mobile applications, using synchronous code for operations subject to network latency turns out to be significant ballast. Some operating systems may even kill your application if it freezes the user interface for more than a given number of milliseconds.

## Formulating the Pattern

An operation coded asynchronously doesn't block the UI thread and keeps the application much more responsive. The Sink-or-Async pattern can be summarized as shown here:

> *Implement asynchronously any operations expected to perform for longer than a bunch of milliseconds.*

Expressed in this way, the pattern may sound a bit too tough; however, it addresses a very sensitive point of most mobile applications. Imagine a user that, inadvertently or not, starts a Hypertext Transfer Protocol (HTTP) request to a web service. If not coded in an asynchronous way, the user interface is blocked, and the user can't hit Back to cancel the operation. This is frustrating to any user.

## Implementation of the Pattern

Generally speaking, asynchronous operations can be coded in one of two ways: by having a worker thread make a synchronous request or by using the asynchronous API that the SDK of choice makes natively available.

In terms of functionality, both approaches work so long as you have the possibility of spawning your own background threads in the operating system and the programming framework allows choosing between synchronous and asynchronous API. For example, in Windows Phone and Windows 8 Metro, you only have an asynchronous API—there's no choice.

However, the bottom line is that loading or posting information asynchronously allows you finer control over the user interface, which remains responsive. Going asynchronously requires a bit more attention to refreshing the user interface with the results of the operation. It's simply a matter of using the proper API.

## Chaining Async Network Operations

The use-case you're working on may require that two or more network calls are chained together, and the next doesn't start until the previous one terminates. This is never a problem if you can code each network operation synchronously. In this case, you just execute operations sequentially:

```
var feed = GetSomeFeed(url1);
var content = ProcessData(url2, feed);
```

If you only have an async API available, how can you chain two or more calls? That's precisely the case in Windows Phone, where you have just no API to execute a network call synchronously. You start with the following code for the first call:

```
var request1 = (HttpWebRequest) HttpWebRequest.Create(url1);
var result1 = request1.BeginGetResponse(ProcessResponse1);
```

The *ProcessResponse1* callback is invoked when the response has been obtained. If you need to use part of the response to prepare a second call, you can only prepare the call within the body of the *ProcessResponse1* callback, as follows:

```
private static void ProcessResponse1(IAsyncResult asyncResult)
```

```
{
    // Get response of the first call
   var response = (HttpWebResponse) request.EndGetResponse(asyncResult);
   // Start a new async call from here
   :
}
```

It turns out that you probably can design the code to maintain a high level of readability with only two or maybe three sequential calls; with more calls, it becomes a very intricate mess.

The problem here is not asynchrony—it's readability. You can have sequential calls, each of which executes asynchronously, but how would you code them in a readable way? An elegant way out would be to use coroutines. A *coroutine* is a method that defines multiple entry and exit points and that can maintain its state between invocations. In the end, the basic idea is keeping the overall workflow in a loop and starting the various steps as the results they need become available. The following pseudocode explains the idea:

```
while (!Task1_IsCompleted || !Task2_IsCompleted || !Task3_IsCompleted)
{
   if (!Task1_IsPending && !Task1_IsCompleted)
       ExecuteTask1(...);
   if (!Task2_IsPending && !Task2_IsCompleted)
       ExecuteTask2(...);
   if (!Task3_IsPending && !Task3_IsCompleted)
       ExecuteTask3(...);
}
```

The various properties used for controlling the loop are wrapped as members of a class and set by *ExecuteTask*N procedures as appropriate. You write operations as distinct async tasks, and the surrounding routine ensures that the overall workflow runs steps in the right sequence.

C# offers only the basics for building coroutines such as the *yield* statement and iterators. An excellent introduction to coroutines and a C# implementation is provided at the following website by Jeremy Likness: *http://goo.gl/HPe9G*.

In .NET 4.5 and C# 5.0, this problem will be solved brilliantly with the introduction of some syntactic sugar. C# 5.0 has two new keywords specifically designed for the purpose: *async* and *await,* used as follows:

```
private async void startButton_Click(Object sender, RoutedEventArgs e)
{
    // First step
    var response1 = await ExecuteTask1();

    // Second step
    var response2 = await ExecuteTask2(response1);

    // Third step
    var response3 = await ExecuteTask2(response2);
    :
}
```

The *await* keyword instructs the compiler to generate code that executes the method (i.e., *ExecuteTask*N) asynchronously. In addition, the control flow won't move to the next instruction until the current operation has completed. In the end, it looks like it is plain synchronous programming, but it is merely a sequence of async operations.

## The Logon-and-Forget Pattern

When you land on a webpage that requires authentication, you typically have a chance to ask the system to keep you logged on for a number of days. Having verified your credentials, the system emits an authentication cookie that the browser will use for future sessions until it expires. Figure 7-8 shows the logon window of the Twitter website.

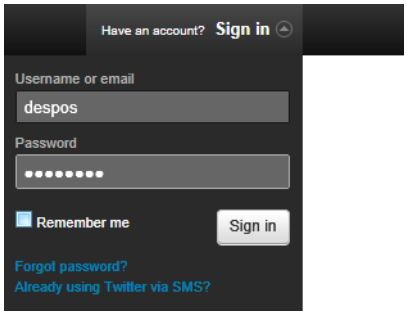



**FIGURE 7-8** The logon box on the Twitter website.

Mobile applications should do the same, and possibly store the credentials for an even longer time. Depending on the application, it may be acceptable to authenticate the user once, store his credentials, and keep on working until the user explicitly signs out.

### Formulating the Pattern

The Logon-and-Forget pattern is common in any mobile application that is bound to a remote web service. A large share of mobile applications fall into this category because the remote web service may simply be the back end of the application.

The idea behind this pattern is to save the user the burden of typing credentials over and over again. It's analogous to the single sign-on idea for web applications, except that it is a more viable point in mobile because of the harder input experience. This pattern can be expressed as follows:

> *You should ask for credentials once, store them safely, and transparently authenticate the user in every session.*

The user should log on once, and then she should be able to use the application even if she forgets the credentials and login duties.

## Implementation of the Pattern

An application that implements the Logon-and-Forget pattern starts by offering a classic logon screen where the user can enter the user name and password. As mentioned, the classic Remember Me check box we've clicked a million times is redundant here. The pattern is all about assuming that such a setting is always on. Figure 7-9 shows the logon screen of the Facebook application for Android.

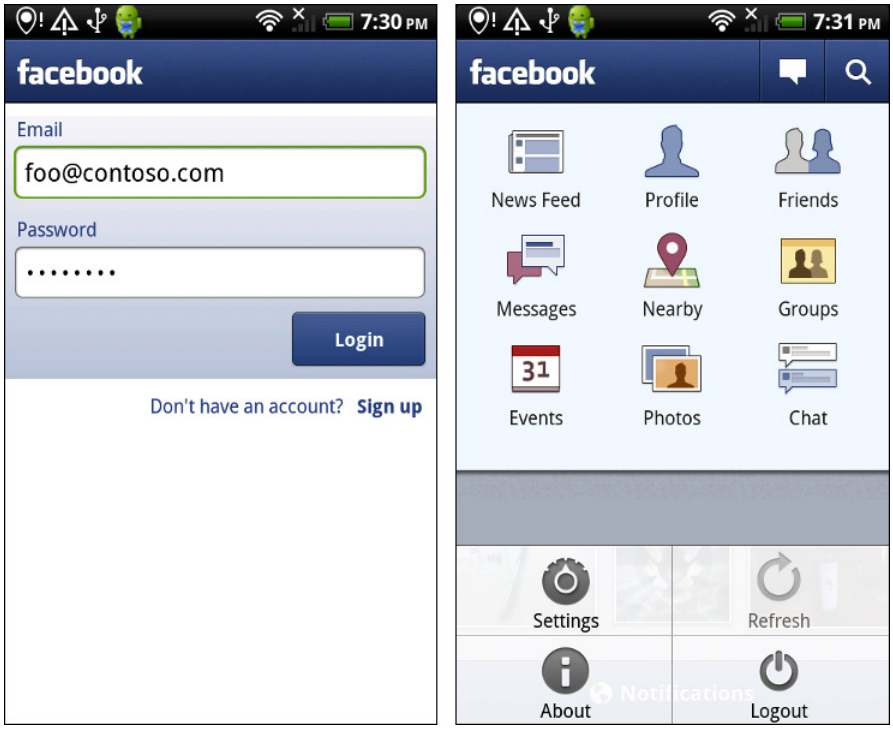



**FIGURE 7-9** The Logon-and-Forget pattern as implemented by Facebook for Android.

If the authentication succeeds, the application stores whatever data it needs to retain to make successive calls to the remote server. The shape and color of this data depends on the server application and its authentication protocol. If authentication is based on a homemade protocol and basically consists in sending an HTTP request with credentials embedded in the body of the packet then all the application needs to store safely are user name, password, PIN, token, and whatever the server may require. If authentication is based on a protocol such as OpenID or OAuth (e.g., Twitter and Facebook), then you need to store the access token that the server returns after a successful authentication. For Twitter and Facebook, for example, that token is the magic key that will enable the application to access the server-side user account and post and read messages, photos, and more.

How long should the credentials stay in place? A common practice is to keep credentials until the user explicitly decides to log out, as in Figure 7-9. When the user logs out, credentials and any other user-specific data is cleared. It is also possible for the application to ask users periodically to renew authentication. Personally, I consider this pattern annoying to the user. Log on once and forget about credentials is definitely the way to go.

### Security Considerations

Web and mobile security looks like a balancing act between ease of use and risks. Even more so than a laptop, a mobile device can be lost or stolen, and another person can pass himself off as you and access all the information in the phone, such as email, personal data, pictures, and contacts. With the Logon-and-Forget pattern implemented, this person also gains access to the information (and possibly the secrets) that the application stores. This fact can't be denied.

If your application can reveal important secrets if it's used by unauthorized users, then the Logon-and-Forget pattern is either not implemented or at least left as an option under the user's responsibility. In the latter case, the scenario will turn exactly into a weblike scenario.

As for storing credentials safely, you might want to use one of the system's repositories. In one way or another, these repositories are safe enough. In iOS, you use the Keychain repository, which signs data stored with the application key. In Windows Phone, you can use isolated storage to store data in a file—even a clear text file—with the guarantee that no other application can access that data programmatically. However, isolated storage has a known location in the file system, so there's always the risk that if the device is stolen, someone can navigate to the files and read clear text. In Android, you find a similar situation. If you use a private system repository, then your data cannot be accessed programmatically, which means that neither other programs nor the user via installed programs can access private files. However, low-level tools—mostly created for developers—always can be used to explore the depths of the file system.

In the end, if your application stores really critical data, you might want to use ciphering or encryption. There's no guarantee that your files can't be read. And to be on the safe side, write applications that save application-specific files to a private store that isn't programmatically accessible by others.

**Important** For Android devices, encrypting or hiding sensitive data is always recommended because if the phone is rooted and applications are allowed to access every fold of it, then an external application can navigate and open just every file.

## Patterns for Presentation

The patterns in this section are related to the way in which the application should present data to the user—some basic dos and don'ts of the user interface.

### The Babel-Tower Pattern

Localization has always been a very important aspect of software applications, but the reality is that only big companies can face the costs of a serious localization effort. On the other hand, very few custom applications really need a deep and well-done localization that covers more than two or three languages and span over the full spectrum of localization aspects.

In fact, there are many aspects of localization: there's text to translate, graphics to adjust, and maybe layouts to restructure. For relatively simple and small applications like mobile applications, however, the translation aspect dwarfs everything else. Therefore, localization in this context is mostly about displaying messages in the user's native language.

## Internationalization vs. Localization

Two similar terms are sometimes used when talking about multilingual software. They are internationalization (abbreviated as *i18n*) and localization (abbreviated as *l10n*). *Internationalization* refers to changing the internal architecture of software so that it can handle multiple languages and regional settings. On the other hand, *localization* refers to the specific action of adding support for a specific language to already internationalized software.

Should you really care about internationalizing and then localizing your mobile applications? For how many languages? To what level of accuracy?

**Note** The apparently weird shortcuts used to refer to *internationalization* and *localization* stem from the number of letters found between initials—18 in the word *internationalization* and 10 in the word *localization*.

## Formulating the Pattern

In the Bible, the Tower of Babel was built by humans to reach heaven. In return for this presumptuous attempt, God punished humans by making their languages mutually incomprehensible. Today, the term *Babel Tower* indicates a confusion of voices and languages.

The Babel-Tower pattern is inspired by the recognition that a typical mobile application is small enough to be localized effectively with a modest investment into any business-relevant language that you can identify. So the answer to the core question "Should I care about localization?" is something along the lines of "Sure, why not!" The pattern can be summarized as follows:

> *Avoid hard-coded and fixed layout text and design your application to support the dynamic injection of properly translated text.*

A best-selling point of mobile applications is being highly comfortable for users. Offering the application in a user's own language is a double-edged sword. On one end, it would make the application more enjoyable and highly rated; on the other hand, if the localization is poorly done, the benefit immediately turns into a drawback. For this reason, it is crucial that you give your application an international structure from the beginning and that you design localization so that text and other culture-specific resources can be injected easily and ideally without needing to wait for a new release of the application.

## Implementation of the Pattern

As mentioned, localization is the act of adapting an application to a specific culture, whereas internationalization comes first and consists of designing the code in such a way that localization is possible. An internationalized application, therefore, supports:

- Replacing text over all the screens
- Replacing graphics
- Replacing screen layouts
- Replacing specific portions of the application logic

These features are listed in the order in which I would personally address them. The first and most important step toward internationalization is enabling text replacement. This step is common to all applications; other steps may not apply to every application.

Mobile operating systems are multilingual by design, meaning that once you make the application available in multiple languages, the device will do the rest, picking up the right text automatically. Mobile development platforms typically offer native tools to make text replaceable. In iOS as well as in Android, BlackBerry, and Windows Phone, you add resource files to the project that contain localizable text and optionally reference other auxiliary resources, such as graphics and screen layouts. Using resource files is the first step; referencing localized text in the user interface is the second. Each platform has its own API. Here's an example of what you use in iOS to reference a localized string:

```
NSLocalizedString(@"HiThere", @"Welcome message displayed to logged users");
```

Instead of the plain string, you use a pointer to an entry in the resource file. @HiThere is the ID that identifies the real text in the resource file selected for the current language. The second string in the *NSLocalizedString* is merely a comment aimed at providing context information to translators and developers. In other platforms, you find nearly identical patterns and classes. And, overall, this is not really different from what you may have done for years in web and desktop applications. So where's the challenge?

Resource files are embedded in the binaries being packaged and installed onto the device. When you release the application, you can add as many localized resource files for as many languages you like. If you receive feedback about the low quality of the translation of one of the languages, or if you want to add a new language to the list, all you can do is prepare a new release.

The Babel-Tower pattern suggests that you consider using an external service to provide your application dynamically with translated text. Upon startup, the application connects to the server and downloads the text it needs in the (single) language it needs. The text is cached locally and updated only when the application receives notification that an updated translation is available. You can either design the application to check for updates automatically upon startup or arrange a push notification service.

This approach is beneficial in many ways. First, it shrinks your time to market, especially when you intend to support several languages from the beginning. You can be published initially with only a decent quality translation and improve it in a few weeks. The user receives frequent notifications of updates, notices the difference, and may even be happier because she sees that you're taking care of her. The translation work is done remotely, meaning that you can buy the services of professional translators or even set up a crowd-sourced mechanism that has users participate in the translation efforts and vote for the best translation for a given text.

The downside of the Babel-Tower pattern is that you need a framework that makes it work. You need your own API to download available text; you need an API to receive notification of changes; and you need an API to emit text and other resources in the application. Furthermore, you also need an API for the back end to enable people or vendors to contribute translations.

**Note** In a recent multiplatform project of mine (consisting of multiple distinct applications for a few mobile platforms), I developed the embryo of such a system. I limited it to a Representational State Transfer (REST) service returning text and a device-specific library to extract text strings from cached data. That's not really hard to do and involves code that is highly reusable for any given platform. A work in progress that implements the full Babel-Tower pattern is *http://Tiyla.com*.

## Further Considerations

While most international websites are limited to just a few languages, international mobile applications are often offered in more than 10 languages. If you measure the amount of work, translating a mobile application is easier than translating a full-blown website. Other factors, however, make mobile translations more challenging—for example, space constraints. English is the most used language for applications, but the English language is one of the most compact languages. Space is critical in mobile devices, so a user interface optimized for English text may experience overflows when the text is translated to another language. This is a common scenario for a variety of European languages, such as German, but it could be even more dramatic if you look at Chinese, which features relatively short words but much larger fonts.

Text used in mobile applications must be much more concise than on equivalent desktop solutions. This is because of space constraints, but also because of the hurrying and cognitive load that go with reading on a small screen. Subsequently, the user interface of mobile applications is often padded with specific terms, abbreviations, symbols, and shortcuts of many kinds. Stated simply, there's a context behind most mobile screens, and this context must be well understood by translators. The quality of translation is therefore a critical factor. That's why a professional translation service is an option (if you have the budget), but it has to be a very good one. A crowd-sourcing platform for software translation sounds like a great compromise between cost and quality.

## The Do-as-Romans-Do Pattern

In the 1990s, desktop applications conquered the world with their graphical user interfaces (GUIs); in the 2010s, mobile applications are going to do the same, thanks to their natural user interfaces (NUIs). A child of the multitouch capacity of devices, a NUI is made of a number of gestures, such as swiping the screen to scroll and pan, pinching/stretching to zoom, tapping to click, tapping and holding to open context menus, and shaking the device to undo and redo.

Mobile software platforms offer a native set of widgets and shared services that characterize the look-and-feel of native applications. When you write your own application for a given mobile operating system, you can't help but stay as close as possible to the native look-and-feel. As that old proverb says: when in Rome, do as the Romans do.

### Formulating the Pattern

The pattern can be summarized as follows:

> *It is compelling for users, and possibly also advantageous for developers, to abide by the look-and-feel and capabilities of the host operating system.*

Applying the pattern is compelling for users because your application will behave like others, so common gestures will produce the same effect and common actions never will leave users baffled and confused. In one word, the overall user experience is improved. For developers, it is beneficial, too, because they can use a number of existing services and widgets. This means a lot of work saved and more time to focus on what really makes your application unique and compelling.

### Implementation of the Pattern

The Do-as-Romans-Do pattern is another example of a pattern that has more of a guiding principle than a concrete and detailed strategy to solve a common problem. Just for this reason, though, it is probably even more important to follow.

Steps to implement the pattern boil down to keeping the overall user interface as close as possible to the standards of the platform. This certainly means using native controls and common solutions for data navigation and presentation. For example, most Android devices have a pair of Menu and Search buttons. The intended role of these buttons is providing a Menu button and a Search box based on the current state and context of the application. This means that for Android users, it is fairly common to tap the Menu button when they wish to perform an action (e.g., quit a game or change settings), but they can't see any button or widget on the current screen. Likewise, the Search button is what Android users tend to tap when looking for specific items. Figure 7-10 shows how a sample water polo scoring application uses the Menu feature.

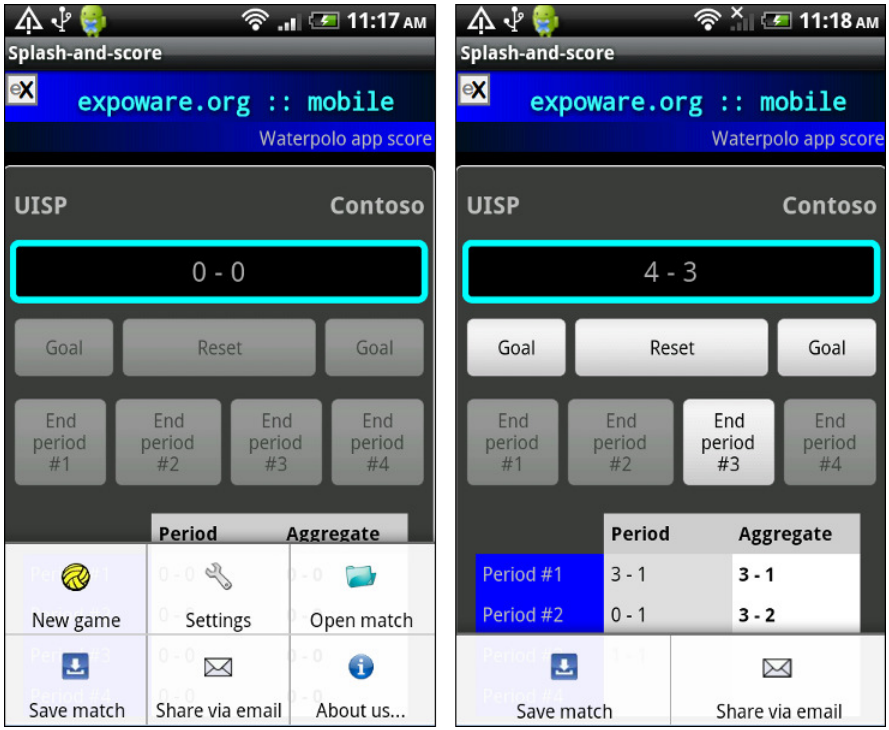



**FIGURE 7-10** Different context menus based on the state of the application.

When the match has not started yet, the menu offers to open a previously saved match, settings, and information about the author. When the match is in progress, the menu is more compact and just offers to save and suspend the current match or share the current score via email.

In Windows Phone, instead, you still have a Search button, but that button is not shown to third-party developers. This means that developers are responsible for any in-app search feature.

This is not to say that one platform is preferable over another; it is only to remark that each platform is different and has its own specific guidelines that third-party applications must comply with for the sake of users.

Other examples of the Do-as-Romans-Do pattern can be found in the overall approach to application navigation and data presentation. For example, a Windows Phone application doesn't use tab strips but rather goes for a pivot or panorama layout. Likewise, a Windows Phone application has an application bar similar to iPhone applications, whereas Android applications use the hardware Menu and Search buttons.

## The List-and-Scroll Pattern

For years, we've been told that endless lists of items weren't a good thing in both web and desktop applications. Displaying too many items on a single webpage would make the page slower to download; displaying too many items within a desktop application would force the user to deal with

the scroll bar, which may not be fun. (Admittedly, though, mouse scroll wheels greatly alleviated this issue.) So developers looked into pagination, and pagination toolbars became a common presence on any webpage with more than 20 items to display.

In mobile applications, pagination happens less frequently and through a totally different user interface. Displaying a long list of items in a mobile user interface is not necessarily a big deal for a couple of surprising reasons—scrolling is a very natural gesture on handheld touch devices, and it is also quite natural on larger tablets such as iPads. Native mobile applications rely on a richer software infrastructure than most websites, so they can use data caching and REST services to return data-only responses instead of HTML-based responses. The payloads are much smaller, and the download time is not necessarily an issue. Furthermore, due to the Sink-or-Async pattern, any download happens asynchronously, which makes the user experience much smoother anyway.

## Formulating the Pattern

Sometimes a long list of items should be displayed; in mobile applications, you don't need to invent new ways of displaying a list. You just pick up the list widgets (or solutions) that the platform makes available and stick to them. The core idea behind the List-and-Scroll pattern, therefore, is the following:

> *Don't be afraid of using (vertical) lists in your mobile application, even long lists that contain more than 100 items to scroll.*

List-oriented widgets are available in the various mobile SDKs to simplify coding, but also to reinforce the idea that listing is great to use in the mobile space. Moreover, mobile guidelines are also rethinking the role of horizontal scrolling, which was banned from web and desktop applications for being patently annoying. The reason of such as a turnaround is always the same—scrolling both vertically and horizontally is a natural gesture on touch devices, especially handheld devices like a phone.

## Implementation of the Pattern

While building lists is a common task in mobile applications, the level of automation offered by the various platforms is not the same. In Windows Phone, for example, it couldn't be easier: all you do is add a *ListBox* control to the screen and use data templates, and then you specify the layout for individual rows and perform data binding. The *ListBox* control in Windows Phone is a self-contained, reusable component that needs only a bit of configuration; it doesn't need you to program it unless you intend to alter its predefined behavior.

In Android, the approach to building lists is nearly identical, and so are the tools that you find in the framework. What's different, though, is the level of automation. In Android, in fact, you use the *ListView* widget in the parent screen but need to provide it with an adapter object, which gets data and does the binding job. This code has to be written every time, whereas it is mostly buried in the framework in Windows Phone. In iOS, the situation is analogous to Android.

Vertical lists are the simplest approach and quite an effective one to display lists of items. However, lists that are too long (containing over 100 items) may be boring to scroll for the user, who needs

a few swipes to reach the bottom. Pagination, therefore, still has a role in mobile development—it serves to keep the initial load of the list to a reasonable threshold. A common practice is putting a fake item at the bottom of the list that causes more items to appear. When hit, the fake item triggers an event that adds more items to the list. Alternatively, you can download and show the next page of items automatically as the user scrolls down until he reaches the bottom of the list. This form of pagination is different from web pagination, as the number of data kept in memory is not constant but grows with the pages visited.

## Horizontal Scrolling

Horizontal scrolling is definitely an option you should consider in mobile user interfaces. It may not suit every scenario, but it can help when you want to display more information in a limited screen space. When the text being displayed is too long to fit in a screen area, and if it is a monolithic text that can be broken up into pieces, then it's OK to wrap the text in an auto-scrollable horizontal panel. The user just needs to pan the text left and right to read it all: I consider it to be the mobile replacement for tooltips. Figure 7-11 shows this pattern in action in Factor Master—a free Windows Phone application for having a bit of fun with prime numbers.

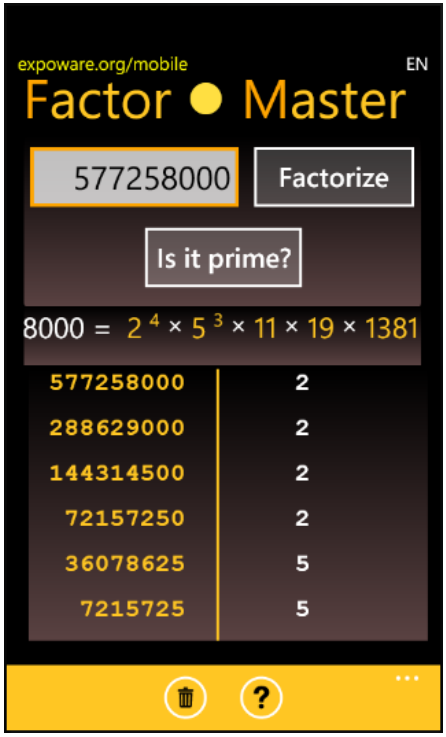



**FIGURE 7-11** An example of panning text: the expression resulting from factorization can't be broken up, so it can be slid horizontally to the right with a finger movement.

Toolbars with too many items can sometimes be implemented in the same way, to scroll horizontally. All users see is a horizontal bar to swipe to find the desired element—mostly an item represented with some graphics (see Figure 7-12).

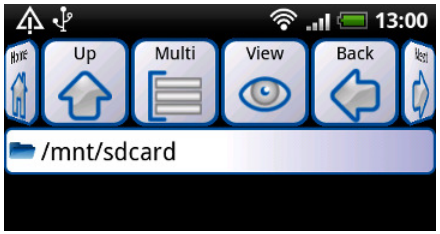



**FIGURE 7-12** A horizontally scrolling toolbar from Astro, a popular Android file system manager.

# Behavioral Patterns

The patterns in this section are related to specific behavior that the mobile application should implement while performing tasks that alter the state or connect to the outside world.

## The Predictive Fetch Pattern

Predictive Fetch is a design pattern historically bound to Ajax. With Ajax, developers gained the ability to place out-of-band calls to the web server and get raw data instead of full-blown HTML pages. In this way, developers can fetch data ahead of using it and be ready to serve the next user requests instantly. Predictive Fetch is ultimately a guess that you make based on evidence such as statistics that prove what actions your users commonly take at a given point.

If a user spends more than a minute watching a certain video, then you quietly download abstracts of other videos that you have stored that relate to the current one. There's no guarantee that the user will choose to see another video (let alone one of those that you proposed), but if he does, at least you've eliminated any delay between two logically related and successive actions.

### Formulating the Pattern

Over the web, the Predictive Fetch pattern is employed to increase the responsiveness of the application and improve the user experience. In the mobile space, the original purpose of the pattern is entirely preserved and even augmented. The pattern can be summarized as follows:

> *If you depend on network connectivity, download data that is likely to be used later and make sure you have enough data stored at any time to survive a lack of connectivity.*

This description contains the entire classic Predictive Fetch pattern from Ajax, plus a new perspective. In mobile, connectivity is not guaranteed to exist and be reliable. You cannot afford to download data on demand—you must try constantly to download useful information in advance and cache it for later—essentially saving for a rainy day.

### Implementation of the Pattern

An effective implementation of the Predictive Fetch pattern requires a deep understanding of how users play with the application. For web applications, you can rely on logs and for more specific analysis on Google Analytics. For mobile applications, you can rely on Google Analytics for Mobile, which enables tracking application usage in much the same way as in a website. All you do is call an API for each event or screen view that you're interested in. (See *http://code.google.com/mobile/analytics* for more about this.)

Armed with this information, you know what users are likely to do at a given stage and can code appropriately to get ahold of the data you may need next. This is the classic implementation of Predictive Fetch, and while it definitely relates to mobile applications, it is not specifically targeted to mobile.

In mobile, you have another more compelling reason to exercise your skills at prediction. As there's no guarantee that you will have connectivity when you need it, downloading and caching data for later use will permit to survive lack of connectivity for hours, if not days. I call this aspect of mobile applications the *sliding download*. As an example, consider an application that gives weather forecasts for the day. Most similar applications work even if you're disconnected. In the past, I have been disconnected for three days, and still I got regular forecasts. The trick is easy to unveil—such applications download forecasts for a week or so in the reasonable hope that you won't stay disconnected for more than an entire week. As soon as the application has a chance to connect, it will download data for another week, and so forth.

Figure 7-13 shows a screenshot of the Postino application (for Windows Phone), which informs the user about the currently available number of stamps. This number is downloaded from the server and cached locally. When the user navigates to the page, the application attempts to download a fresher report. If that fails, the cached number is displayed. This pattern allows the application to give a continuous feel to the user regardless of connectivity problems, and overall, it transmits a positive message—the application understands me and is taking care of me.

## The Memento-Mori Pattern

The expression "Memento mori" is a Latin phrase that translates as "Remember you will die." It is commonly believed that the words were uttered by the slave of a Roman general during the celebration of a successful military campaign. In the context of mobile development, Memento Mori is a reminder of the mortal condition of any applications. In fact, in a mobile environment, the operating system—not the user—decides when an application has to stop.

### Formulating the Pattern

All mobile platforms allow only one application to control the device at a time. When the user starts a new application, the one previously in the foreground goes to the background, ready to be resumed if the user returns to it or quickly starts a new instance. Applications in the background are considered to be paused applications and can be resumed instantly when the user navigates back to them. (The details of how this happens vary with the operating system, but the core story remains the same.) Furthermore, the operating system reserves the right to stop background applications at any time if that helps to free much needed resources.


**FIGURE 7-13** An example of the Predictive Fetch pattern.

The primary purpose of the Memento Mori pattern is alerting applications to pay due attention to notifications received from the operating system. Applications are notified when they are no longer in the foreground and are given only a few seconds of guaranteed lifetime to save any relevant state. The pattern is summarized like this:

> *Applications always should save their relevant state when the operating system forces them into the background.*

Every application is then responsible for determining which data needs to be saved and how. In general, you might want to save the application state directly to a permanent store. Operating systems, however, may offer some intermediate in-memory repository to speed up the resurrection time. For instance, this is what Windows Phone does.

**Note** I deliberately used the term *pause* to indicate the state of applications in the background. These applications still may be allowed to do some work (that doesn't in any way affect the user interface) and may be abruptly terminated if idle. After being paused, applications won't receive any further notification before being terminated.

### Implementation of the Pattern

Implementing the Memento Mori pattern is a matter of implementing the Memento pattern from object-oriented programming. The Memento pattern is about object serialization and refers to the object's ability to save its public state to a stream so that a brand new instance with the same state can be created later.

All programming frameworks provide good native support for object serialization and, more often than not, they also offer a quick shortcut through system-managed dictionaries that are easier to populate and are fast enough. In iOS, you can either save the application's state to a class that conforms to the *NSCoding* protocol, or you can create a property list (plist). An iOS plist is basically a dictionary serialized to an XML file. In Android and in Windows Phone, you have native object serializer classes that do most of the work for you. The hardest part of working with the *Memento Mori* pattern, however, is providing a formal definition of your application's state.

More often than not, and regardless of the operating system, what the user really gets when a previously paused application resumes is a fresh new instance of the application that has been passed any state saved. Saving state appropriately and accurately is vital for any mobile application.

## The As-Soon-As-Possible Pattern

Occasionally connected applications are very common in the mobile space. They are applications that require reliable connectivity to work properly but, at the same time, they sometimes are used in places where connectivity is unavailable. This means that for such applications, it is critical to devise operations to work both with and without a functioning network.

### Formulating the Pattern

The As-Soon-As-Possible pattern just aims at emphasizing the importance of always having a plan B when planning critical and durable operations from within a mobile application. It is encapsulated in the following statements:

> *Remote operations that are critical for the application should be implemented in a protected manner and reiterated a few times before failing. In case of failure, however, the operation should be recorded and played back as connectivity returns.*

The expression "as soon as possible" just indicates that the message that the user sends to the application when commanding an operation is "Perform this task as soon as possible and any way you can, at your earliest convenience." The failure is intended to be temporary, and there should be at least the guarantee that the data the application refers to can be recovered later.

### Implementation of the Pattern

The As-Soon-As-Possible pattern is strictly related to network operations such as the POST of data to a remote server. You can implement the pattern in two ways that I like to call Black-or-White and Grayscale.

The Black-or-White algorithm indicates that you attempt to perform the operation, and if it fails, you notify the user and add the operation to an internal queue. As soon as the network comes back, all the operations in the queue are performed. An operation is removed from the queue only when it completes successfully.

The Grayscale algorithm indicates that you break down the data to post in small pieces (a few kilobytes in size each) and attempt to send them separately. You might want to vary the size of the packets depending on the network connection: smaller pieces if over 3G, and larger pieces if over WiFi. An operation is completed only when all of its pieces have been sent successfully. If the network fails to execute a given step, you queue that step and proceed to the next step only when it has completed successfully.

The Grayscale algorithm requires collaboration from the server, which typically will return a token on the first request. The token must be sent with all successive pieces and will be used as the key to link all the pieces of the same original operation.

## Detecting Network Changes

Even once established, a connection may go down at any time. The most likely reason is that the user is moving around and can go out of range in a matter of minutes. So never trusting the network state is one of the golden rules of mobile (native) applications.

In general, you might want to always have a set of helper functions to check quickly if the network is available—or better yet, you might want to set up a background service that notifies of network changes.

**Note** The term *background service* here refers to a platform-specific component that acts as a listener and receives system notifications about a variety of events. You create this component and register it with the system. Such a component is managed by the system and turns out to have a minimal impact on battery life.

For detecting network changes, the ideal is to have a background service detect changes and update the user interface asynchronously and to have your own helper function that you actively invoke to check connectivity before embarking on a user-commanded network operation.

In Android, you create a background service for detecting network changes using a *receiver;* namely, an application extension that captures system events of a certain type fired by other applications as well as system events. Figure 7-14 shows a simple (global) notification receiver that is limited to showing pop-up messages. More likely, an application would have an embedded receiver that performs code, refreshes the user interface, or both.

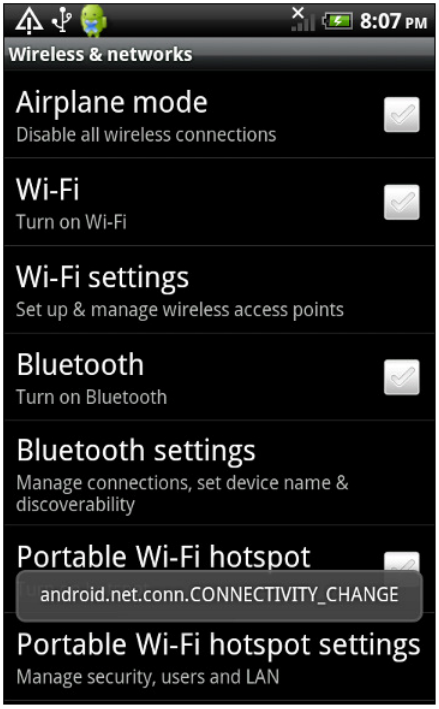



**FIGURE 7-14** An Android receiver in action.

In iOS, you use the *SystemConfiguration* framework, and more specifically the *SCNetworkReachabilityRef* interface, to monitor the network state of a device. In Windows Phone, you have a couple of classes that claim they can tell you about the state of the network. They are *DeviceNetworkInformation,* for Windows Phone 7.5, and *NetworkInterface,* which works in all versions of Windows Phone. Both classes have methods that return a Boolean value if connectivity is available. Using these methods, though, is not entirely reliable because both methods may fail to tell you if you have access to the Internet. They seem to be reliable only in telling you whether a connection is available, but not if you are actively using it. In my tests, I've found the following trick much more reliable:

```
// Works in all versions of Windows Phone
private static Boolean IsConnected()
{
    var networkType = NetworkInterface.NetworkInterfaceType;
    return networkType != NetworkInterfaceType.None;
}
```

We'll return to the topics of network state and connectivity for the various platforms in the next chapters dedicated to specific platforms.

**Note** Being able to check programmatically the state of the network, including the type of connectivity you're having, may require a special permission in your application manifest file.

## Summary

Writing a mobile application, regardless of the target platform and the details of the SDK to use, is not the same as writing a desktop or web application. Overall, it is an easier task because a mobile application has a far lower number of use-cases to plan for than a web or desktop application. However, a mobile application poses a number of challenges that the developer must be ready to face. This chapter discussed a dozen design patterns that attempt to provide guidance for the most common of these challenges (at least, the most common ones that I have encountered so far).

CHAPTER 8

# Developing for iOS

*Before everything else, getting ready is the secret of success.*

*—Henry Ford*

**In this chapter:**

- Getting Ready for iOS Development
- Programming with Objective-C
- Programming with MonoTouch
- Deploying iOS Applications

These days, many chief technology officers (CTOs) are struggling to define the ideal strategy to integrate mobile development into their platforms. One reason that developing mobile solutions is problematic is that even today, the word *mobile* often just means "iPhone and iPad" to many executives.

Mobile software has many facets, the most compelling of which is probably an iPhone app. We have executives who are impressed by a cool iPhone app they've seen and now want to replicate it. We have program managers impressed by the amount of work (and money) it would take to provide a comprehensive mobile solution that reaches a wide audience share. And then we have mobile users. Mobile users just use their devices; they enjoy apps and sites so long as they are easy to use, are compelling in some way, and address a specific need—perhaps a non-business need, but still a need.

Mobile users don't have explicit demands, but they are extremely selective and not particularly forgiving. In the mobile space, users are no longer the victims of developers' manias and obsessions. In mobile development, the user is king, and applications are made to please the user.

In this context, iOS—the mobile operating system run by iPhone, iPod Touch, and iPad devices—contributed significantly toward establishing some de facto standards for mobile technology, usability, and application capabilities. Today, a mobile solution sometimes consists solely of a mobile site. However, every time someone goes beyond the level of a mobile site, then an iOS native application is built.

# Getting Ready for iOS Development

This chapter attempts to cover the key aspects of application development for the iOS operating system. Because this book is not intended to teach you iOS programming in detail, this chapter (as well as the upcoming chapters on Android and Windows Phone) won't delve into the depths of the Apple software development kit (SDK); however, hopefully it will provide a clear guide to the available options and a full overview of what's required and where you can start when planning a real-world development project that goes beyond the basics.

## A Brand New Platform for (So Many) Developers

At first, approaching the task of building iOS applications can be a very annoying—or very exciting—experience for developers who have never worked with Mac computers and Apple programming tools before. I know numerous people in both camps, and I can say that building iOS applications is special in a way that other programming is not.

Especially if you, like me, have a strong .NET and Microsoft Visual Studio background, then you should be prepared to face a whole new world and a brand-new development platform. In general, when you, as a .NET developer, embrace full-range mobile development, nothing will be the same anymore. The first thing you have to change is your computer!

### Getting a Mac Computer

You can't do any iOS development without first getting a computer equipped with a Mac operating system. In the Apple world, you will find two operating systems: Mac OS X and iOS. The former runs on laptop and desktop computers, and the latter runs on mobile devices. As obvious as it may seem, iOS derives from Mac OS X and serves multiple Apple devices, including Apple TV. Both iOS and Mac OS X share the same set of core components.

Mac hardware is not known to be cheap; however, you don't need to buy the most expensive Mac computer to do some good iPhone development. The cheapest Mac computers (Macbook or MacMini) are easily good enough to run the Mac integrated development environment (IDE) and the iPhone emulator.

You need a Mac computer to compile iPhone applications because those applications rely on libraries that simply don't exist in Windows and Linux.

**Note** Technically speaking, you can run some versions of Mac OS X on a Windows-equipped machine and save yourself the cost of buying a Mac computer. Unfortunately, this practice is not legal; it's a violation of the Apple OS license. Legally, you may install the OS only on Apple hardware. The reverse is not true; you *can* legally install a version of Windows on a Mac computer.

## Getting Familiar with the IDE

Any development for the Mac platform, including iPhone and iPad, requires Xcode. Xcode is Apple's toolkit for developers. It includes an IDE, a design tool (Interface Builder), a compiler, and the iOS emulator, as well as a set of tools for instrumentation and performance analysis. You can get Xcode either from the Mac store or directly from *http://developer.apple.com/xcode*. Currently, Xcode 4 is a free download for all members of the iOS and Mac developer programs.

Figure 8-1 offers a view of the Xcode environment.

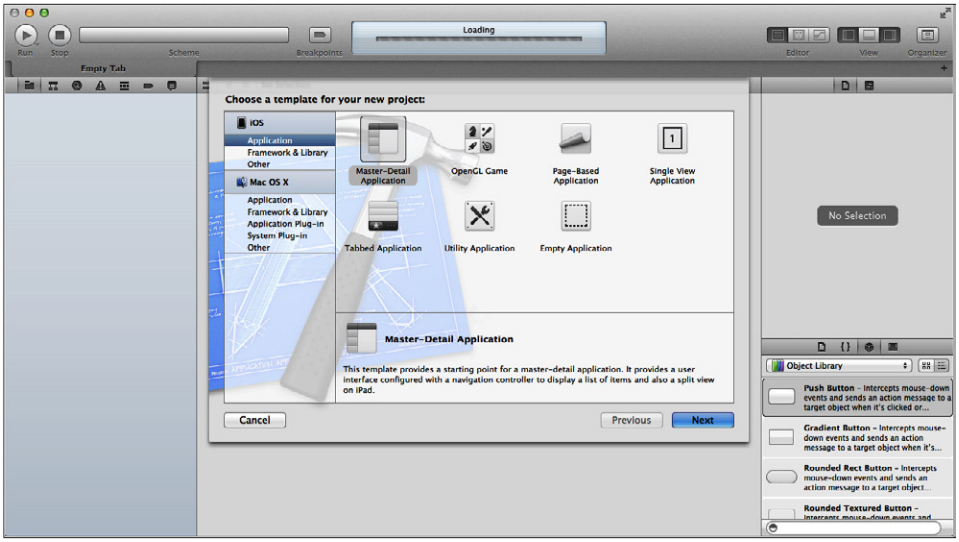



**FIGURE 8-1** The Xcode environment.

The other essential download for iPhone development is the iOS SDK. Normally, you are given the option of installing the SDK while installing Xcode.

The iOS programming platform is based on the set of Cocoa Touch frameworks: the iOS version of the Mac OS X Cocoa application programming interface (API). Cocoa Touch includes nearly everything that you need to create iOS programs: audio, video, graphics, networking, local and cloud storage, control over hardware, and access to built-in applications such as the address book, phone features, and maps.

## Joining a Developer Program

To download tools and start practicing with iOS code, you need to enroll in one of the Apple developer programs. To start with, you can simply create a free account and use that to download Xcode and the iOS SDK. This configuration enables you to write code and compile iOS programs at will—but you can't deploy any of them to any physical devices. If you want to go beyond the boundaries of the iOS simulator and experience the thrill of running your app on a real phone, as in Figure 8-2, then you have to join a paid program.

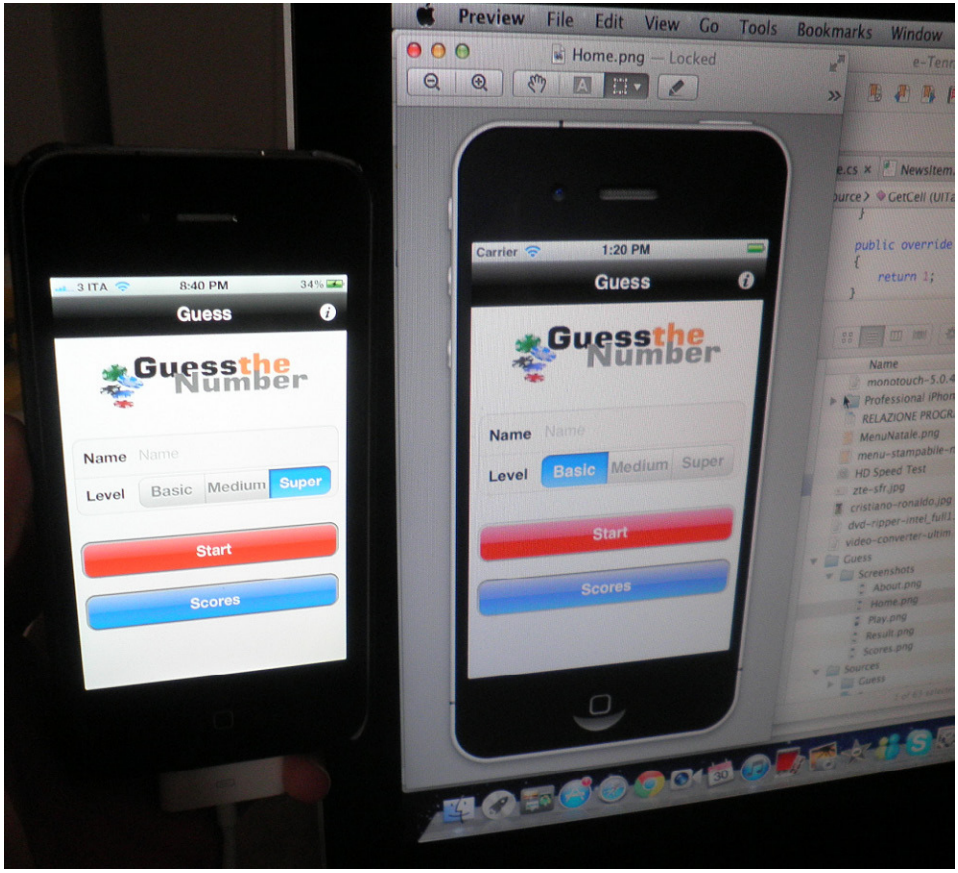



**FIGURE 8-2** From the iOS simulator to a real iPhone device.

To become an official iOS developer, you need to pay a fee of $99/year. Paying the fee unlocks your account and enables your Xcode environment to deploy any applications to a registered test device as an option. (It should be noted that there's a limitation on the number of developer applications that you can install on a test device.)

You register an iOS device (i.e., iPhone, iPad, or iPod Touch) as a test device from Xcode. Subsequently, all you need to do is keep the device connected to the computer running Xcode. The environment detects the device status. If a connected device is not currently registered as a test device, Xcode will ask you if you want to promote the device to the rank of a test device.

The annual fee also enables developers to submit finished applications to the App Store for certification and publishing. The App Store is the only channel a developer has to distribute applications. In an enterprise scenario—at the cost of a different license—you can distribute applications through your intranet (or over the web) to devices that hold the enterprise certificate. For more information, visit *http://developer.apple.com*.

**Important** As mentioned, membership dues are paid yearly. Membership allows you to publish applications to the App Store for the duration of your membership. Also, note that applications that you publish while a member remain available in the store only for the duration of your membership. Therefore, if you don't renew your membership, your published applications will be delisted.

## iPhone vs. iPod Touch vs. iPad

The iOS world is populated by at least three main types of device: iPhone, iPod Touch, and iPad. Today, all of them can be equipped with the same version of iOS so long as the hardware of the device is aligned with the requirements of the operating system. For example, this means that you can't upgrade an iPhone 3G released back in 2008, or an equally old iPod Touch to the latest iOS5.

That begs a question: Should you address these types of device differently and consider creating distinct applications? To answer this, let's first take a closer look at the differences between these devices.

Very few differences exist between an iPod Touch and an iPhone device, and all of them are linked to the availability of specific hardware subsystems. For example, the iPod Touch lacks the radio and cellular systems found in the iPhone, so it can't be used to make phone calls. Likewise, the iPod Touch can't host a SIM card, and therefore it can't connect to the Internet over a 3G network. An iPod Touch also may lack a gyroscope sensor. For most applications, these differences don't matter. In the end, unless you aim to build an application that relies on, say, phone calls and text messages extensively, there's usually no reason to fork development between iPhone and iPod Touch. Most of the time, it suffices that you disable any user interface (UI) elements that trigger functions not available on the iPod Touch.

Regardless of the (few) hardware differences, iPhone and iPod Touch share the same screen size. That's important, because screen size alone is a major factor that may cause you to fork development. Screen size is also what makes the iPad (and tablets in general) special and often worthy of a specific version of your software. Like the iPod Touch, the iPad lacks a radio system and can't be used to make phone calls or send text messages. Unlike the iPod Touch, though, an iPad may have the 3G subsystem. In addition, the first version of the iPad lacked a camera.

To sum up, an application for the iOS platform is usually optimized for the iPhone device and works unchanged on iPod Touch devices. Applications written for iPhone work on iPad devices as well, but the larger real estate screen of iPad devices may suggest you fork and develop around a different set of use-cases. When running on iPad, an iPhone application either will run in a smaller area of the screen or will be stretched to cover the full screen. Note, however, that you also can create dual-purpose applications that incorporate an iPad-specific layout. In this case, the system picks the proper layout automatically.

## Choosing the Development Strategy

An iOS application is internally based on the set of Cocoa Touch frameworks. Calls to the various components are glued together by Objective-C statements. Objective-C, therefore, is the primary programming language of the iOS platform.

Development of iOS applications started in 2008, when Apple opened the App Store to applications contributed by third-party developers. For the first couple of years, using Objective-C to perform direct access to the Cocoa Touch frameworks was the only possible way to write iOS applications. Even today, the vast majority of the 500,000+ iOS applications are written in Objective-C.

Objective-C, though, is no longer the only option that developers can use to build iOS applications. A few other higher-level frameworks have been built on top of Cocoa Touch that developers can program using other languages that they may already know or be more comfortable using.

### Using Objective-C

Without beating around the bush, Objective-C is neither a popular language nor a language that you can learn on the fly. Compared to popular object-oriented languages such as Java and C#, Objective-C may look a bit old-fashioned and even weird. Objective-C is essentially the ANSI C language extended with some object-oriented features mostly inspired by Smalltalk syntax.

**Note** The previous statements about Objective-C are likely biased by my personal background, preference, and attitude. So don't give this opinion more value than it deserves—and, especially, don't take it personally if you disagree!

Although Objective-C ultimately supports classes and object orientation, it does so through a syntax that looks a bit unfamiliar to C++ developers and even less familiar to C# and Java developers. Note also that you can develop for iPhone using C or C++. This is just what many game developers do, largely for portability reasons.

Garbage collection is the programming aspect that marks a key difference between Objective-C and C#/Java. Not all versions of the Objective-C compiler available for iOS applications support garbage collection—that is, the ability to pass responsibility for releasing object instances at the right time to a system-provided automatic collector. As a result, iOS developers may be entirely responsible for managing the lifetime of objects. The object lifetime can be managed manually by the developer—as in classic C-style programming—or in a sort of automatic way by marking objects so that the compiler will make inferences about their expected lifetimes.

**Note** Xcode 4.2 introduced Automatic Reference Counting (ARC), which greatly simplified the memory management of Objective-C objects.

Beyond garbage collection, the syntax for defining classes and invoking objects' members is fairly unique in Objective-C. For example, the definition of a class is distinct from the implementation, and

invoking methods looks more like sending a message than placing a direct method call. Furthermore, symbols like + and – are used to mark a method as static or instance. In the end, writing Objective-C code is certainly not an impossible task, as the huge number of applications available clearly demonstrates. At the same time, the costs of writing iOS applications don't end with learning Objective-C. You also need to become familiar with the Cocoa Touch frameworks to build user interfaces and invoke helper services. Finally, you need some familiarity with the Xcode environment and with facilities such as Interface Builder.

The bottom line is that most developers can become productive on Objective-C regardless of their background and skills. However, developers with a strong Java, C#, or even Microsoft Visual Basic background may find it easier to approach iOS development from another angle. Enter the MonoTouch project.

## Using MonoTouch and C#

MonoTouch is an ad hoc framework developed by Xamarin that lets you write iOS applications using a subset of the Microsoft .NET Framework and the C# language. Unlike Objective-C and the direct iOS SDK access, MonoTouch is a commercial product; to use it, you must buy a license. A single developer license will cost you about $399. (You can find more information about MonoTouch at *http://www.xamarin.com*.)

Using MonoTouch doesn't save you from having to get a Mac computer for development, or from downloading and using Xcode. In the end, MonoTouch is simply a framework that offers a .NET facade on top of some of the Cocoa Touch frameworks. You call classes with the same interface used for equivalent .NET objects, which have functionality mapped to iOS-specific frameworks. MonoTouch requires the Mono framework—the cross-platform .NET Framework—as well as the iOS SDK. To write code, you use the MonoDevelop tool and rely on Xcode (and, optionally, Interface Builder) for any UI work.

Overall, MonoTouch makes it possible to write iOS applications using a more modern and feature-rich programming language such as C#. In doing so, you can employ your .NET Framework skills and even reuse some C# code you may have. One key point is that the only code you can share with .NET projects is back-end code. As far as the UI code is concerned, MonoTouch allows you to write event handling and presentation code following .NET patterns; but overall, you're still using the Cocoa Touch philosophy and moving around the same data as in an Objective-C application.

**Important** On the reusability side, it is worth noting that MonoTouch comes with a twin framework—MonoDroid—that wraps the Android SDK and allows you to program Android applications using C# and skills from the .NET Framework. More importantly, if you're planning to port an existing iOS application to Android (or vice versa) if you use MonoTouch and MonoDroid, you can easily find yourself reusing 90 percent of the back-end code. Also porting to and from Windows Phone is much easier than one may think at first. When starting a journey to find the Holy Grail of cross-platform mobile development, MonoTouch, MonoDroid, and the C# language on the background are a really great start. There will be more about MonoTouch later in this chapter, and Chapter 9, "Developing for Android," will discuss MonoDroid briefly.

## Using the PhoneGap Framework

Adobe's PhoneGap is a framework that transforms a client-side web application into a native application for a variety of mobile operating systems, including the iOS platform. As a developer, you write a classic client-side web application using HTML5, Cascading Style Sheets (CSS), and JavaScript. You write and test the application on your favorite platform, using your favorite tools. For example, you can write a client-side web application using Visual Studio 2010 or perhaps WebStorm by JetBrains or even a plain text editor such as Notepad++ (see *http://www.notepad-plus-plus.org*) or Sublime Text 2 (see *http://www.sublimetext.com*). Both Notepad++ and Sublime Text 2 have versions for Windows and Mac OS.

A client-side web application is a web application made up of static HTML pages. By "static HTML page," I just mean a webpage that is not downloaded from any remote web server and doesn't rely on a server-side technology such as ASP.NET or PHP. The HTML page then can be made as dynamic as you like by using JavaScript and Ajax calls to remote endpoints.

You build the user interface using HTML5, and make it compelling with CSS. HTML5 is essential because it incorporates a feature called "local storage," which gives you the ability to save data locally. Your client application also may link to the PhoneGap JavaScript library to gain access to additional features and device-specific hardware such as the camera and accelerometer.

You test the application on the development machine (e.g., a Windows machine with Visual Studio) using an HTML5-compliant browser such as Chrome, Safari, or Windows Internet Explorer 10. If you aim to target primarily iOS or Android, then you need to test on desktop editions of Safari and Chrome, respectively.

After the client web application is complete, you create an ad hoc project for the platform (using Xcode on iOS) and incorporate the PhoneGap framework for the platform and the web source files. You then build and deploy the project as usual on the platform. The web application is packaged as a shell of native code, which internally uses a full-screen web view to load the locally stored HTML pages. Exposed as JavaScript functions, the PhoneGap JavaScript library acts as a bridge between client pages and native device features.

Building a PhoneGap application requires HTML5 and web skills, plus some familiarity with the PhoneGap framework, but you don't need any specific iOS skills to build successful applications. Moreover, you can reuse essentially the same web codebase to package applications for a variety of mobile platforms, including iOS, Android, Windows Phone, and BlackBerry. Chapter 11, "Mobile Applications with PhoneGap," covers PhoneGap development in more detail.

## Other Options

Whle Objective-C, MonoTouch, and PhoneGap are the three most popular options for building iOS applications, other options exist. In particular, you might want to look into Appcelerator's Titanium (*http://www.appcelerator.com/titanium*).

Titanium Mobile is a framework similar to PhoneGap in that it allows you to use JavaScript and can build native applications for a variety of mobile platforms. But in other ways, the two frameworks

are quite different. You develop a Titanium mobile application using JavaScript and the Titanium framework. As it turns out, no HTML or CSS is required to define the user interface. Instead, you build views and add logic only through the Titanium framework. The classes in the framework then are expanded, at compile time, into a mobile-native counterpart on the target platform. Put another way, the Titanium infrastructure reads your JavaScript and translates it into native iOS or Android code.

To build Titanium applications, you need only the ad hoc tools provided with the default package. You don't have to create a different project for each platform. Another difference is that PhoneGap currently supports a larger number of mobile platforms. There's no license fee associated with Appcelerator's Titanium; instead, you pay only if you request support (see *http://www.appcelerator.com/products/plans-pricing*).

Finally, yet another option for iOS applications that's worth at least a brief mention is the possibility of using tools that package existing applications written against a given UI framework into an iOS application. The same pattern that you find behind the PhoneGap initiative can be found in the Adobe's Creative Suite 5.5, which also includes a packager for iOS and Android. Vendors of vertical solutions [e.g., business intelligence, Customer Relationship Management (CRM) solutions, and databases] may offer similar features, which just include an iOS packager on top of apps developed against their platforms.

In the end, you have many options to produce iOS applications. This chapter focuses on the two most popular ones: Objective-C and MonoTouch.

# Programming with Objective-C

Let's start with a quick look at the fundamental programming language of the iOS platform. The building blocks of the iOS development platform are the Objective-C programming language; the Cocoa Touch framework, which provides a set of built-in libraries to code against; and a run-time environment.

The primary role of the run-time environment is processing input from the compiler and dynamically performing operations on objects, such as creating new instances and invoking methods on existing instances. In particular, a basic operation of an object-oriented language, such as invoking a method on an object in Objective-C, is resolved through the run-time system making the whole thing look more like sending a message to the object than invoking a public and fixed endpoint on the instance.

## A Quick Look at Objective-C

As mentioned, Objective-C is the plain C language padded with some object-oriented extensions. Overall, the syntax looks quite different from C++ or Java, and this is just one of the aspects of Objective-C that scares developers at first. I assume here a basic knowledge of the C language and just focus on what extends it.

## Defining a Class

Unlike most popular object-oriented languages (with the notable exception of C++), Objective-C requires that you split the definition of a class into two parts that we can call as interface and implementation. The two parts are usually saved to distinct files. The interface defines the class blueprint, including public properties and methods and, optionally, the parent class. The implementation actually provides code for the declared members. Overall, in Objective-C, the process of defining a class is similar to the Java or C# process of defining and implementing an interface.

**Note** Saving interface and implementation to distinct files is not mandatory, though it is highly recommended for the sake of clarity and design. In general, however, you can have interface and implementation in the same file. Likewise, the same file can contain multiple class definitions. Naming also can be arbitrary, but you usually name interface and implementation files after the class name and give them *.h* and *.m* extensions, respectively.

According to common programming practices, to define a *UserAccount* class, you create two files: Useraccount.h and Useraccount.m. The *.h* extension that you use for interfaces is related to header files of the C language:

```
@interface UserAccount : MyLibraryRootObject
   // Declare properties and methods here
@end
```

In Objective-C, the *@interface* and *@end* directives wrap up any interface declaration. Here's an example that includes fields, properties, and methods:

```
#import <Foundation/Foundation.h>
@interface UserAccount : NSObject {
   BOOL isActive;     // This is equivalent to fields in Java/C#
}
@property (strong, nonatomic) NSString *name;
@property (strong, nonatomic) NSString *password;
- (BOOL) Login;
- (BOOL) LoginAndStayLogged:(BOOL)someBoolValue;
- (BOOL) LoginWithOptions:(BOOL)stayLogged
                          throwOnError:(BOOL)someBoolValue
                          message:(NSString *)someText;
- (void) Test; // void and parameterless
@end
```

*NSObject* and *NSString* are foundation types of Cocoa; you need to import the Foundation.h header file to find their definition. The *#import* directive is equivalent to *using* in C# and Java.

The *@property* directive defines a property that is expected to have a getter/setter method. When you declare a property, you can indicate a bunch of attributes by enclosing them in round brackets. For example, *strong* indicates that the property is owned by the object and is not simply a weak reference to some external object. The *nonatomic* attribute indicates that the default getter will simply return the value without dealing with locks in a multithreaded environment. In a class, fields must be enclosed in a pair of curly brackets.

In a method declaration, return types and parameter types are wrapped in round brackets with the return type that precedes the method name. Parameters follow the method name prefixed by a colon and the type name. The name of the first parameter is implicit; the names of additional parameters are preferably explicit. It is common that the role of the first argument is figured out by the name of the method. This is shown in the previous code, where *throwOnError* and *message* indicate the second and third argument for the *LoginWithOptions* method.

Note that the following syntax, where additional parameters are unnamed, would compile as well. However, this approach is discouraged because it would result in poorly readable code when calling the method. I'll return to this point momentarily when discussing object messaging:

```
- (BOOL) LoginWithOptions:(BOOL)stayLogged :(BOOL)someBoolValue :(NSString *)someText;
```

A method can take a variable number of arguments. In this case, it will have the following prototype:

```
- (NSString *) formatString: (NSString *) pattern, ...;
```

The minus symbol qualifies the method as an instance method; the plus symbol qualifies the method as a static method. No minus and plus just indicates a plain C function.

In Xcode, right after typing the preceding code, you get a warning about missing the implementation of the *name* and *password* properties (see Figure 8-3).

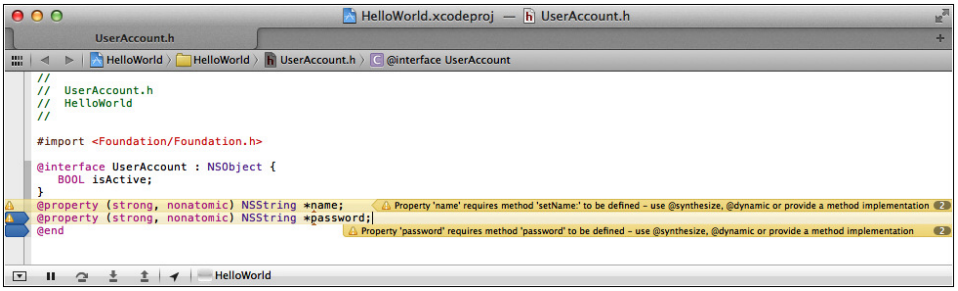



**FIGURE 8-3** Defining a class in Xcode.

Let's switch to the implementation file.

**Note** Objective-C allows you to omit the return type or the type of a parameter. In this case, the compiler assumes it to be the type actually returned by methods. You indicate this type unresolved at compile time as the *id* type. The *id* type is similar to the *dynamic* type that you have in C#. It is worth noting that the *dynamic* type in C# also relies on a run-time system that evaluates the expression dynamically.

## Namespaces and Naming Conventions

Objective-C doesn't have namespaces. For this reason, you are encouraged to add a unique prefix to the name of your classes to avoid collisions. Prefixes are usually a short sequence of uppercase letters (two or three letters) representing the initials of your library or company. If you derive a custom class from a standard Cocoa class, then you should name the new class by simply replacing the Cocoa prefix (NS) with your own.

The name of Cocoa classes begins with *NS* for historical reasons. Cocoa, in fact, is based on the NeXTSTEP development framework, and *NS* are just the corresponding initials.

As for naming conventions, the fundamental point to keep in mind is that method and property names are based on camelCasing; and type information in variable names is not recommended, but it may be tolerated if the variable is not of a basic type such as *string* or *number*.

## Implementing a Class

To implement a class, you start by importing the definition of that class from a previously created *.h* file. The template of an implementation file looks like the code shown here:

```
#import "UserAccount.h"

@implementation UserAccount
   // Actual implementation of class members
@end
```

Properties need a getter/setter method, depending on when they've been declared. You can name and implement these methods as you like. If you do so, though, you must point to methods in the header file via the *getter* and *setter* property attributes, as follows:

```
@property (getter = getValueForIsActive, setter = setValueForIsActive) BOOL isActive;
```

Default names for getters and setters are supported, too. They are *Xxx* and *setXxx,* where *Xxx* is the property name. Objective-C, in fact, tends not to use the *get* prefix on simple accessors. You also can use the *@synthesize* directive if you want the compiler to create getter/setter methods automatically for you. This is equivalent to the following C# code:

```
public Boolean IsActive {get; set;}
```

Here's how to use the *@synthesize* directive and define the body of methods:

```
@implementation UserAccount
@synthesize name;
@synthesize password;
- (BOOL) login {
   return TRUE;
}
- (BOOL) loginAndStayLogged:(BOOL)someBoolValue {
   return TRUE;
}
```

```
- (BOOL) loginWithOptions:(BOOL)stayLogged
                          throwOnError:(BOOL)someBoolValue
                          message:(NSString *)someText {
   return TRUE;
}
- (void) test {
   return;
}
@end
```

In the implementation of methods, you use *self* to refer to the current object (i.e., *this* in C# and Java) and use *super* to refer to the parent class (i.e., *base* in C#). The equivalent for *null*, instead, is *nil*.

## Object Messaging

As mentioned, in Objective-C, you "send a message" to an object rather than invoking a method on an object. The syntax required takes the following form:

```
[receiver message]
```

In the code snippet, *receiver* denotes an object, whereas *message* denotes the name of the method to execute. If any parameters are involved, they are chained to the message. It should be noted that the method to execute is expressed as a string and is resolved dynamically by the run-time environment. Here's the code that you need to create a new instance of the *UserAccount* class and attempt a login:

```
NSString *message = @"Login failed";
UserAccount *account = [[UserAccount alloc] init];
[account loginWithOptions:TRUE throwOnError:TRUE message:message];
```

To instantiate a new object, you first allocate enough memory for it, and then you initialize the object. The *alloc* and *init* methods are defined on the *NSObject* class and are common to any objects that you happen to use in an iOS application.

When calling a method with multiple parameters, the good practice of using named parameters shines:

```
[account loginWithOptions:TRUE throwOnError:TRUE message:message];
```

Rewritten in C#, the preceding code looks like this:

```
UserAccount account = new UserAccount();
account.loginWithOptions(true, true, message);        // Not highly readable code indeed
```

With named parameters, in C#, it also can be rewritten as follows:

```
UserAccount account = new UserAccount();
account.LoginWithOptions(true, throwOnError:true, message:message);    // Much more readable
```

In C#, you don't need a special declaration to enable named parameters and the name of the formal parameter matches the parameter name; in Objective-C, formal parameters and parameter names are distinct. The syntax for methods that accept just one parameter is simpler:

```
[account LoginAndStayLogged:TRUE];
```

Technically speaking, the method name in a message selects the method to execute. For this reason, messages often are referred to as *selectors*. Selectors also include parameter names for methods that accept parameters.

**Important** In Objective-C, an attempt to invoke a method on a *null* object doesn't result in an exception. It simply returns a *null* value.

It is also worth noting that in Objective-C, you can read the value of a property defined on an object using both the square-bracket syntax and the Java/C# common syntax based on the dot:

```
// id account = [[UserAccount alloc] init];  // Don't work with the dot syntax!
UserAccount *account = [[UserAccount alloc] init];
id user = [account name];
id pswd = account.password;
```

Both expressions compile and work well. Note, though, that you can use the dot syntax only on object instances of a known type. The previous code would give you a compile error if you define the *account* object of type *id*.

## Protocols

Objective-C uses the term *interface* to refer to the blueprint of a class being created. In other object-oriented languages, there's no such element: creating the class means specifying its structure and implementation. The closest you get to the concept of an Objective-C interface in programming languages like C# and Java is with *abstract* classes.

In Java and C#, an interface is something different; it primarily serves the purpose of grouping together members that other classes may expect to find in some objects. What you call an *interface* in Java and C# is known as a *protocol* in Objective-C. Protocols are not heavily used in Cocoa Touch and are just an optional programming tool that you may or may not choose to use. Here's an example where optional and required methods are also present:

```
@protocol UserAccountProtocol
- (BOOL) LoginWithOptions:(BOOL)stayLogged
                          throwOnError:(BOOL)someBoolValue
                          message:(NSString *)someText;
@optional
- (NSDate *) GetExpirationDate;
@required
- (BOOL)Login;
@end
```

A class may implement one or more protocols. Protocols are listed in the class header following the *super* class. Note that the verbiage that the Objective-C documentation uses for "implementing an interface" is "adopting a protocol":

```
@interface UserAccount : NSObject < UserAccountProtocol >
...
@end
```

Objective-C provides statements to check whether an object conforms to a given protocol.

## Categories

In any object-oriented language, you can derive a new class from an existing class and add new methods and properties or replace overridable members. To do so, you just need the header of the class and a compiled version of its implementation. Normally, you can't extend an existing class without having its source code.

In C#, though, you are allowed to add new methods to an existing class via extension methods. An extension method is a plain static method written with a slightly different syntax. The extra syntax elements allow the compiler to resolve apparently wrong calls made to the base type by redirecting the call to the extension object. For example, if you define an extension method *ToInt* on the *String* type, then you are actually telling the compiler to accept calls made to *ToInt* from within *String* objects. These apparently wrong calls—you actually have no *ToInt* method on the *String* type—will be resolved by calling the *ToInt* method where it is really defined. In the end, extension methods are some syntactic sugar to make developers' life a bit easier and to increase code readability.

In Objective-C, you achieve a similar capability through *categories*. A category is a separate file that adds new methods (not new properties) to a given class. The implementation of these methods gains full visibility over the class internal members, including private members:

```
#import "UserAccount.h"

@interface UserAccount ( ChangePasswordCategory )
  // Declarations of new methods that manage change of the password
@end
```

You implement methods that fit into a category in a separate implementation file.

You can have as many categories as you wish for a given class. The only restriction is that each category has a unique name and declares (and defines) a different set of methods. At run time, native and category methods are undistinguishable and can be used to achieve the same capabilities. Likewise, category methods are listed as part of the blueprint of the extended class and therefore will be inherited by derived classes.

You mostly use categories to extend native types (*NSString*, *NSObject,* and the like) with additional helper methods.

## Exception Handling

Exception handling in Objective-C looks nearly the same as in Java and C#. It is based on the classic four statements—*try, catch, throw,* and *finally.* In Objective-C, these statements take the form of directives, as shown here:

```
@try {
   ...
}
@catch (UserAccountException *uae) {
```

```
    ...
}
@catch (NSException *ne) {
    ...
}
@catch (id ue) {
    ...
}
@finally {
    ...
}
```

You can catch different types of exceptions, and you should list them from the most specific to the most generic. In the *@catch* blocks, you can just recover from the exception or you can rethrow the same exception. You also can swallow the original exception and throw another exception that better suits your needs.

The *@finally* directive runs regardless of whether the operation attempted in the *@try* block completed successfully or threw an exception.

You can throw exceptions from anywhere in your code. It just suffices that you get hold of an instance of an exception object and pass it to the *@throw* directive, as shown here:

```
id uae = [NSException exceptionWithName: @"UserAccountException"
                     reason: @"Your credentials are invalid"
                     userInfo: nil]
@throw uae;
```

You can create your own exception types by deriving a new class from *NSException*.

**Note** In the preceding code snippet, a static method is called on the *NSException* class to create a named exception. The method is called *exceptionWithName,* and, as you can see, it refers quite clearly to its first parameter. The remaining parameters are referred by name. This is a good programming practice in Objective-C.

## Memory Management

In Objective-C, memory management is explicit, whereas in other languages such as C# and Java, it is mostly hidden from developers. Nearly all concepts discussed here apply to other languages as well, except that the compiler and the virtual machine will make it kind of transparent.

The key fact to notice about objects that you create in Objective-C is that they are subject to *reference-counting*. Reference-counting means that any individual object is associated with a number that indicates how many owners it has. An owner is, in this context, just another object that is currently holding a reference. When the reference count for the object goes down to zero, then the system deallocates the object.

To be precise, an object is a plain reference to a memory location where its data is stored. The memory location is associated with a reference count (also known as a *retain count*); when the count

is zero, then the memory becomes available for other objects, but it is not automatically cleared. Here's an example of creating an object:

```
UserAccount *account = [UserAccount alloc] init];
```

The code creates a *UserAccount* object. The newly created object has a reference count of 1. You create an object using a method whose name begins with *alloc*, *new*, or *copy*.

Every time you pass the object to a method of another object, then the reference count of the passed object is increased by 1. The receiving object is then responsible for decreasing the reference count once it has finished with it.

According to Cocoa Touch ownership rules, you are responsible for the reference count of objects that you create explicitly and objects that you receive. Being responsible for the reference count means releasing your ownership as soon as possible. This decreases the reference count and contributes to ensuring that the memory location of the object will be reused eventually. You release an object using the *release* message:

```
UserAccount *account = [UserAccount alloc] init];
...
[account release]
account = nil;
```

Setting the released object to *nil* may avoid future problems. In fact, invoking a method on a *nil* object results in a no-operation; invoking a deallocated object may crash the application.

Auto-releasing is another aspect of memory management in iOS. Sometimes you need to create an object and then return it—for example, as the return value of a method. In this case, you send the *autorelease* message to the object, which basically means that the object will receive a release call at some point in the future. Auto-release is a form of deferred release. Here's how to use auto-release in code:

```
- (NSArray *) getRegisteredUsers {
    NSArray *users = [NSArray new];
    // ...
    return [users autorelease];
}
```

Objects that received the *autorelease* message go in the auto-release pool and typically are processed at every iteration of the system's run loop. Like old-fashioned Windows applications, iOS applications are based on a message loop. Every loop picks up a message from the queue, creates an auto-release pool, dispatches the message, and processes the auto-release pool, removing objects with a reference count of zero or with a pending auto-release.

What if you need to increase the reference count of an object—such as an auto-release object? You just send the *retain* message. The *retain* message is useful to last for the lifetime of an object beyond the current scope.

All Cocoa Touch objects have a method called *dealloc*, which the run-time environment invokes automatically when the object's reference count drops to zero. This is the last chance that an object has to release resources it holds, such as instance variables.

Finally, in iOS 5, Objective-C supports ARC, which moves the burden of memory management to the compiler and aligns Objective-C to other languages, such as C# and Java. With ARC enabled, the following code doesn't produce any memory leaks:

```
NSObject *yourObject = [[NSObject alloc] init];
// Use the object but don't call release
```

This is because the compiler will detect increments of the reference count not followed by a proper decrement algorithmically and adds a decrement call automatically. Once compiled, the preceding code looks like this:

```
NSObject *yourObject = [[NSObject alloc] init];
// Use the object but don't call release
[yourObject release];
```

ARC is a simple setting in Xcode that must be enabled on a project. However, using ARC in a project binds you to a few policies. If you're new to Objective-C, using ARC is a no-brainer, and policies will just sound like programming features. If you're familiar with Objective-C and memory management, ARC may change some of your habits. Refer to the Apple documentation for more details.

## The *HelloWorld* Program

Armed with some basic knowledge of Objective-C, let's see what it takes to create a basic application with Xcode. As a first step, I'll just take one of the sample iOS projects that Xcode creates for you and dissect its source code.

### Application Startup

In Figure 8-1, you saw the default project templates that Xcode has to offer. After you pick up one of those, you are asked to name your project and, more important, to select the target platform. The options are iPhone (which would also work on iPod Touch devices), iPad, and Universal, as shown in Figure 8-4.

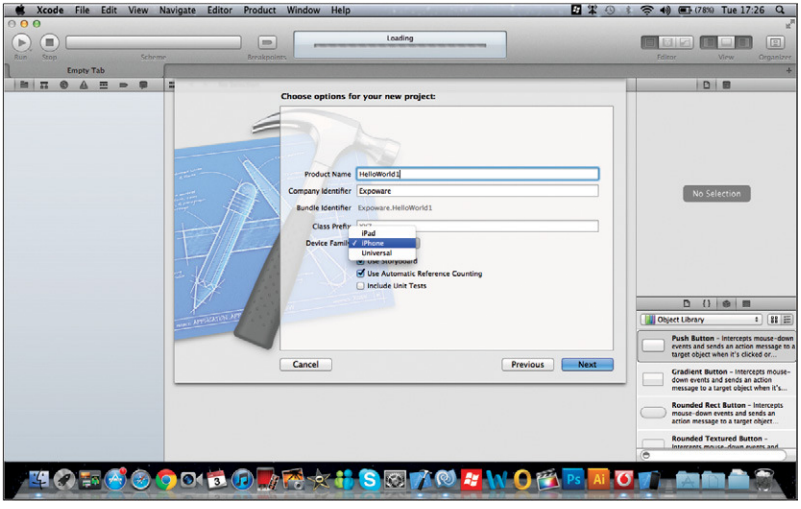



**FIGURE 8-4** Choosing options for the project.

Any iOS program is launched from a *main* starter method that's usually located in the Main.m file. Here's the typical implementation of the *main* method:

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}
```

The role and signature of the *main* method is the same as in every C program. In particular, the *argc* parameter indicates the number of command-line arguments, whereas *argv* is a an array of strings where the first element contains the command line as a string and remaining elements are command-line arguments.

The *main* method creates the top-level auto-release pool and then starts the application with a call to *UIApplicationMain*. The *@autoreleasepool* directive tells the compiler that the following block is an auto-release block, meaning that any objects allocated within are subject to ARC and will be released when the pool is drained. Note that *@autoreleasepool* is syntactic sugar for some code that would explicitly create the pool like this:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

The *UIApplicationMain* method creates a singleton *UIApplication* object. The *UIApplication* object is ultimately responsible for loading the user interface and starting the application.

In iOS applications, the *UIApplication* object routes incoming user events and action messages sent to controls. It also maintains a list of all the windows currently open in the application. A window is represented by a *UIWindow* object and displays a view which is, in turn, represented by a *UIView* object. Most applications count just one window and a number of views inside the window.

As you can see, *UIApplicationMain* also takes a third and a fourth argument. The third argument (*nil* in the previous snippet) refers to the main window class to create for the application. Passing *nil* indicates that you will build the window starting from the content of an XIB file.

**Note** If you read through the documentation, you will run into NIB and XIB files in the same context. Originally, files with UI definitions were binary files known as NIB files, created and edited exclusively via Interface Builder. XIB files are a more recent introduction and provide the same content, but in a human-readable XML format. XIB files are compiled into NIB files when the application is built.

The fourth argument refers to the name of the app-delegate class to use. If *nil*, then the name is assumed to be *AppDelegate,* and a class with this name is assumed to be in the project In the following code snippet, the app-delegate name is indicated explicitly as the name of the class *AppDelegate*:

```
return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
```

A delegate is an object that gets notified when the object to which it is connected reaches certain events or states. An app-delegate is therefore the object that receives notifications when the *UIApplication* object reaches certain states such as "finished launching" or "will terminate" or receives a memory warning.

## The App-Delegate Object

The app-delegate is a class implemented by default in a couple of *.h* and *.m* files. Here's the public interface of a standard app-delegate for a single-view iOS application:

```
#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) ViewController *viewController;
@end
```

The *AppDelegate* object adopts the *UIApplicationDelegate* protocol. The protocol lists all the messages that a *UIApplication* object can receive in its lifetime. The most important of these messages is *application*:*didFinishLaunchingWithOptions*. This message arrives when the application has finished starting, and it represents your last chance to do some work before the user interface is displayed to the user. Here's the implementation of a sample app-delegate:

```
#import "AppDelegate.h"
#import "ViewController.h"

@implementation AppDelegate
@synthesize window;
@synthesize viewController;

// Message application:didFinishLaunchingWithOptions
- (BOOL)application:(UIApplication *)application
                    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Create a window object that covers the entire screen
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    // Create a view from NIB and adds its controller (for iPhone and iPad)
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone)
    {
       self.viewController = [[ViewController alloc] initWithNibName:@"UI_iPhone" bundle:nil];
    }
    else
    {
        self.viewController = [[ViewController alloc] initWithNibName:@"UI_iPad" bundle:nil];
    }

    // Set the view controller
    self.window.rootViewController = self.viewController;

    // Show the window
    [self.window makeKeyAndVisible];
    return YES;
}
```

When the *application:didFinishLaunchingWithOptions* message arrives, the least you can do is display the main window of the application. This is accomplished by sending the *makeKeyAndVisible* message to the application itself.

Other interesting events you find in the *UIApplicationDelegate* protocol are *applicationWillResignActive* and *applicationDidEnterBackground*. Both events have the same signature:

```
- (void)applicationWillResignActive:(UIApplication *)application
{
   ...
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
   ...
}
```

The former event is fired when the application is about to move from an active to an inactive state. The latter event fires when the application is pushed to the background. In this case, it is your responsibility to release shared resources and save any relevant state information for later, when the application will be resumed.

**Note** Compared to the signature of other messages in the *UIApplicationDelegate* protocol, the *application:didFinishLaunchingWithOptions* message may seem a bit weird. In the hope of making the Objective-C syntax more digestible to C# and Java developers, here's the C# signature of the *application:didFinishLaunchingWithOptions* message:

```
bool application(UIApplication *application, NSDictionary *launchOptions)
```

This is how MonoTouch exposes the message to application developers.

## Dissecting the Project

In an iOS project, you find a bunch of XIB files. XIB files are created when the user interface behind the iPhone application is saved. XIB files are analogous to form designer files in .NET. As the XIB file provides the graphical user interface for a view, a view-controller object provides its behavior (see Figure 8-5).

Expressed as a pair of *.h* and *.m* files, a view-controller class governs the behavior of a view and handles things like initialization and touch events. You typically have a view-controller class for each XIB file.

In iOS, a view is different from a window. A view is simply a rectangular area hosted in a window. The view displays content such as text, drawing, animation, and, of course, controls. You typically have a single window, but multiple views displayed one at a time. In iOS, you have various specialized views such as table, web, alert, and navigation.

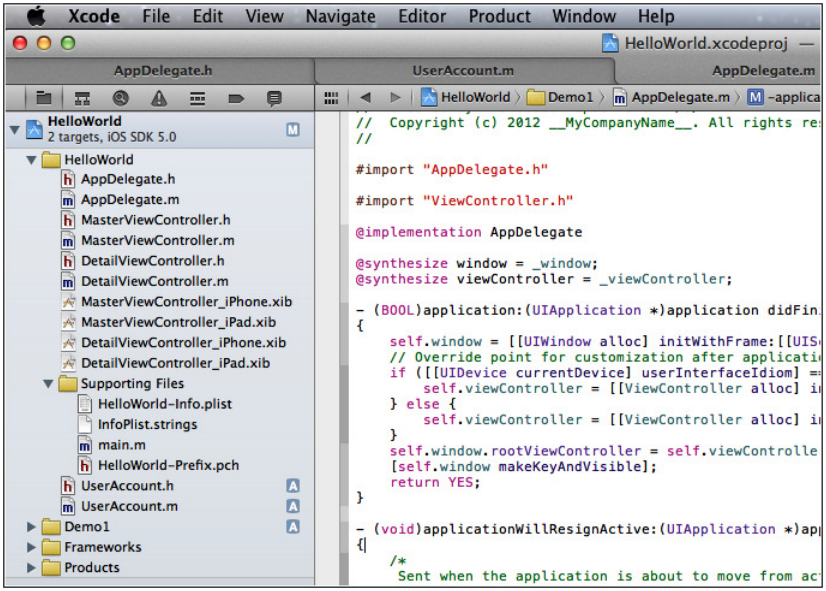



**FIGURE 8-5** A sample iOS project.

An iOS project includes a PLIST file that acts as the manifest of your application. Other support files that you may have are lists of localized strings and images. All auxiliary resources are packaged together in a *bundle* when you build your application. In the iOS jargon, a *bundle* is a directory that contains executable and auxiliary files. You control name, version, and other aspects of the bundle from the PLIST file of the project. (In Visual Studio, this is similar to the project's Properties page.)

## The View-Controller Object

Let's have a look at the internals of a simple view-controller object. The header of a simple single-view iOS project looks like the following:

```
#import <UIKit/UIKit.h>
@interface ViewController : UIViewController
@end
```

The following code is an excerpt from the implementation of the view-controller:

```
#import "ViewController.h"

@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)viewDidUnload
```

```
{
    [super viewDidUnload];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
}

- (void)viewWillDisappear:(BOOL)animated
{
   [super viewWillDisappear:animated];
}

- (void)viewDidDisappear:(BOOL)animated
{
   [super viewDidDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
    } else {
        return YES;
    }
}
@end
```

The most relevant methods are *viewDidLoad* and *viewDidUnload,* where you complete the initialization for display and release subviews, respectively.

## A Look at a Table-Specific View-Controller

Let's briefly consider the structure of a more sophisticated view-controller, such as the one you would use in a master/detail application. The default Master/Detail project template that you get from Xcode creates one view-controller for the master view and one for the details view. Here's the interface for the controller of the master view:

```
@interface MasterViewController : UITableViewController
@property (strong, nonatomic) DetailViewController *detailViewController;
@end
```

As you can see, in this case, the base class of the view-controller is more specialized than *UIViewController*. The *UITableViewController* class provides the basic behavior expected for the view-controller of a table-based view. The following code snippet illustrates what's different in a table view-controller:

```
@implementation MasterViewController

// Customize the number of sections in the table view.
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

// Customize the number of rows in a section.
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return 1;
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
                            cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
        if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
            cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
        }
    }

    // Configure the cell
    cell.textLabel.text = NSLocalizedString(@"Detail", @"Detail");
    return cell;
}
@end
```

A table view is split into sections and rows. The first two messages for the table view indicate how many sections you're going to have and how many rows will be in each section. The third message—the *cellForRowAtIndexPath* message—prepares the cell for the given item to show in the table.

**Note** As you may have figured out, programming for iOS requires at least a working knowledge of some design patterns; the Model-View-Controller (MVC) is one of these. In the end, it stresses the idea that the view and the logic behind it should cooperate but remain distinct blocks of code. The same is true for any data being worked on in the view—the model.

## Examining a Sample Application

Guess is a sample application that you get as companion code with this book. (For more information on accessing companion material to this book, see the section "Code Samples" in the Introduction.) It implements a simple game: guessing a number in a given range. The application consists of a Home view, where you enter the player's name, and a Play view, where you enter a number and get a response. A Scores view and an About view complete the application. Figure 8-6 shows the Home view and the Play view of the application.


**FIGURE 8-6** The Guess application in action.

### The App-Delegate

The Guess application is based on a navigation application template where a bunch of views are linked to one another. The following code shows the header of the app-delegate for the application:

```
#import "Player.h"

@interface GuessAppDelegate : UIResponder <UIApplicationDelegate>
@property (nonatomic, strong) UIWindow *window;
@property (nonatomic, strong) UINavigationController *navigationController;
@property (nonatomic, strong) Player *player;
@end
```

As you can see, it exposes a window object and a navigation controller. In addition, the app-delegate has access to a *Player* object that represents the central object of the domain.

The implementation of the app-delegate just provides a handler for the *didFinishLaunchingWithOptions* message. The message handler creates the main screen and sets the navigation controller. The navigation controller receives as an argument the controller of the main view—the *HomeViewController* object. Here's a code snippet:

```
- (BOOL)application:(UIApplication *)application
                    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    HomeViewController *homeViewController = [[HomeViewController alloc]
                                              initWithStyle:UITableViewStyleGrouped];
    self.navigationController = [[UINavigationController alloc]
                                 initWithRootViewController:homeViewController];
    self.navigationController.navigationBar.barStyle = UIBarStyleBlack;
    [self.window makeKeyAndVisible];
    return YES;
}
```

The final lines of the preceding code just pick up the graphical style of the navigation bar and display the main window.

## The *Player* Class

The *Player* class incorporates the state and parameters of the game. The header is shown here:

```
@interface Player : NSObject<NSCoding>
@property (nonatomic, copy) NSString *name;
@property (nonatomic, assign) NSUInteger level;
@property (nonatomic, assign) NSUInteger score;
@property (nonatomic, strong) NSDate *scoreDate;

- (id)initWithName:(NSString *)name level:(NSUInteger)level;
@end
```

The *Player* class inherits from *NSObject* and conforms to the *NSCoding* protocol. The *NSCoding* protocol marks the class as serializable. The properties on the class indicate the name of the player, as well as the level of the game, the current score, and the date.

The class features one method that initializes the game starting from the name of the player and level. Now let's look at the implementation of the *initWithName* method:

```
- (id)initWithName:(NSString *)name level:(NSUInteger)level
{
   _name = [name copy];   // get a copy of the string object
   _level = level;
   _score = 0;
   return self;
}
```

The *Player* class also contains a comparer method that will be used later in the Scores view to sort the results:

```
- (NSComparisonResult)compare:(Player *)player
{
   if (self.score > player.score)
   {
        return NSOrderedAscending;
   }
   else
   {
        return NSOrderedDescending;
   }
}
```

Finally, let's have a look at class serialization. In iOS, serialization is referred to as *encoding*, whereas the process of deserializing an archived object instance is called *decoding*. The method *encodeWithCoder* demonstrates serialization, whereas the method *initWithCoder* creates an instance of the class from a decoder:

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
   [aCoder encodeObject:_name forKey:kNameKey];
   [aCoder encodeInteger:_level forKey:kLevelKey];
   [aCoder encodeInteger:_score forKey:kScoreKey];
   [aCoder encodeObject:_scoreDate forKey:kScoreDateKey];
}

- (id)initWithCoder:(NSCoder *)aDecoder
{
   _name = [aDecoder decodeObjectForKey:kNameKey];
   _level = [aDecoder decodeIntegerForKey:kLevelKey];
   _score = [aDecoder decodeIntegerForKey:kScoreKey];
   _scoreDate = [aDecoder decodeObjectForKey:kScoreDateKey];
   return self;
}
```

Names like *kNameKey* and *kLevelKey* are constants that refer to the entry in the serialization stream where the value of the corresponding property will be saved or read. In Objective-C, you define a constant as you would in plain C:

```
#define kNameKey             @"name"
#define kLevelKey            @"level"
```

This code uses member names prefixed by an underscore. Where do these member names come from? The implementation of the *Player* class contains the following line:

```
@synthesize name = _name, level = _level, score = _score, scoreDate = _scoreDate;
```

The *@synthesize* directive instructs the compiler to provide a default getter/setter method for the property. The *_xxx* expression, where *xxx* refers to the name of the property, just indicates the name of the field to use in the getter and setter to read and write the actual value.

## The Home View

In iOS development, you can define the view using a visual designer (i.e., Interface Builder) or you can create the entire view programmatically by placing UI components in the specific location that you want. Overall, using Interface Builder is quick, but it's not as immediate and straightforward as it would be to do the same thing in Visual Studio. In any case, creating views programmatically is not much less unusual in iOS development than in .NET development. The home view of Guess has been created programmatically. I'll discuss the use of Interface Builder later in this chapter.

The following code presents the view-controller of the main view. It is a *UITableViewController,* meaning that the user interface is structured as a table:

```
@interface HomeViewController : UITableViewController<UITextFieldDelegate>
{
   @private
   UITextField *_nameTextField;
   UISegmentedControl *_levelSegmentedControl;
}
@end
```

The interface lists a couple of private members for the text box used to enter the player name and the radio buttons to choose the level (a *UISegmentedControl* component in iOS). Initialization of visual elements and positioning take place in the implementation of the *HomeViewController* class.

The user interface of the main view features the navigation bar, a table with input elements, and a couple of buttons to start the game and see the scores. All these elements are created in the *loadView* method of the *HomeViewController* class. Here's the code that sets up the Start button that you saw in Figure 8-6:

```
- (void)loadView
{
     // Invoke the method on the base class
   [super loadView];

     // Sets the background color of the view
   self.view.backgroundColor = [UIColor whiteColor];

     // Sets up the Start button
          UIImage *redButtonImage = [[UIImage imageNamed:@"redButton"]
                                stretchableImageWithLeftCapWidth:kButtonImageLeftCap
                                topCapHeight:0];
   UIButton *startButton = [UIButton buttonWithType:UIButtonTypeCustom];
   [startButton setTitle:@"Start"
                 forState:UIControlStateNormal];
   [startButton setBackgroundImage:redButtonImage
                 forState:UIControlStateNormal];
   [startButton addTarget:self
                 action:@selector(showPlay)
                 forControlEvents:UIControlEventTouchUpInside];
   startButton.titleLabel.font = [UIFont boldSystemFontOfSize:kFontSize];
   startButton.frame = CGRectMake(kButtonPaddingX,
                                  CGRectGetMaxY(self.tableView.frame),
```

```
                                    self.view.frame.size.width - (kButtonPaddingX * 2.0f),
                                    redButtonImage.size.height);

    // Other code here
    ...
}
```

The button consists of a background image and a caption. The image is stretched to cover the entire area of the button. The event touch-up ends up invoking the *showPlay* method. The *showPlay* method is defined as follows:

```
- (void)showPlay
{
   if (_nameTextField.text && ![_nameTextField.text isEqualToString:@""])
   {
       currentAppDelegate.player = [[Player alloc]
                                    initWithName:_nameTextField.text
                                    level:_levelSegmentedControl.selectedSegmentIndex];
         UIViewController *playViewController = [[PlayViewController alloc] init];
         [self.navigationController pushViewController:playViewController animated:YES];
   }
   else
   {
          UIAlertView *alertView = [[UIAlertView alloc]
                                    initWithTitle:@"Error"
                                    message:@"Please insert a player name"
                                    delegate:nil
                                    cancelButtonTitle:@"OK"
                                    otherButtonTitles:nil];
          [alertView show];
   }
}
```

If the text box is left empty, the application displays a message box. Otherwise, it creates a new instance of *Player* using the specified name and level and stores it in the global app-delegate. Next, the *showPlay* method sets up a view-controller for the Play view and pushes it to the navigation controller requesting some animation. The net effect is that when the user taps the Start button, the application moves to the rightmost view of Figure 8-6—the Play view, which is discussed next.

## The Play View

In the sample project, the Play view is created from an XIB file. You create and edit XIB files using Interface Builder, one of the add-on tools of Xcode. Interface Builder is similar to Visual Studio designers in that it offers drag facilities and WYSIWYG editors to create views (see Figure 8-7).

All graphical designers have the problem of matching their visual components to object references. Each platform seems to have a slightly different solution to this problem. As we'll see in the next chapter dedicated to Android programming, each visual component is given a unique ID. When creating the view, you programmatically figure out the exact name of the class associated to the ID, get an instance, and assign that to a local variable. In Windows Phone, the Visual Studio designer does all the work for you. All you do is assign components a unique ID and the tool gives you ready-made object references through view properties that have the same name as the ID.

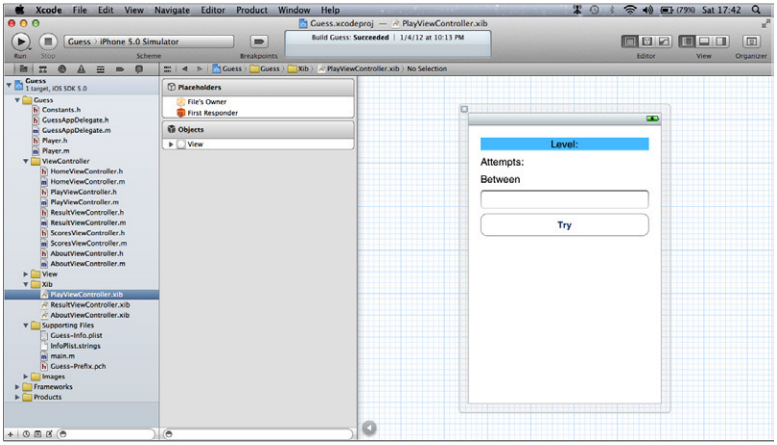



**FIGURE 8-7** Interface Builder in action.

In iOS and Interface Builder, things are even more different. You don't assign unique IDs to the components that you drop onto a view surface; you create outlets instead. An *outlet* is an artifact that hides plain object references. You need an outlet to gain programmatic access to a visual element. For example, if you want to set the text of a label after a button click, you need an outlet to the label.

Once you add a new component to the view, you open up the Inspector editor pane. The Inspector has an area for referencing new outlets. You first select the element to reference and then drag from the outlet area directly to the source code of the view-controller header file, as in Figure 8-8.

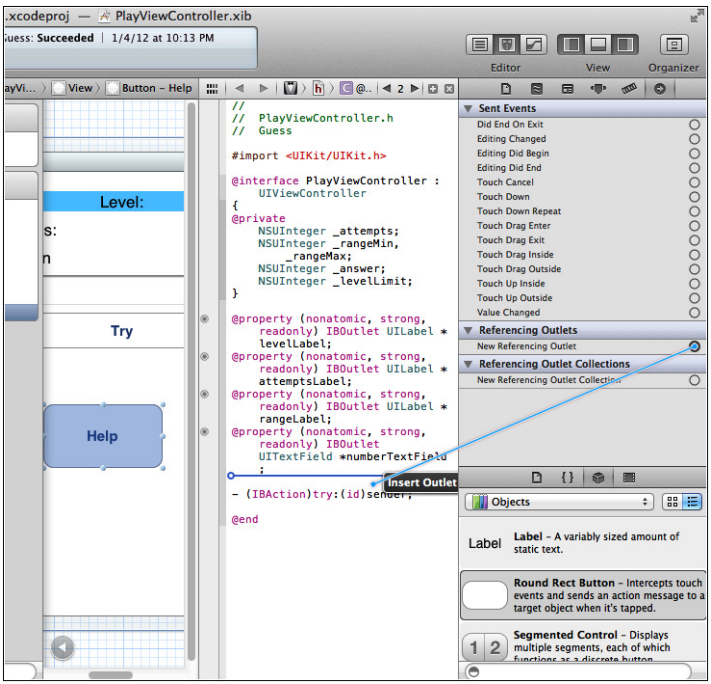



**FIGURE 8-8** Inserting a new outlet.

When you drop the element, a small window will pop up, asking you to name the outlet and finalize the operation. This is illustrated in Figure 8-9.

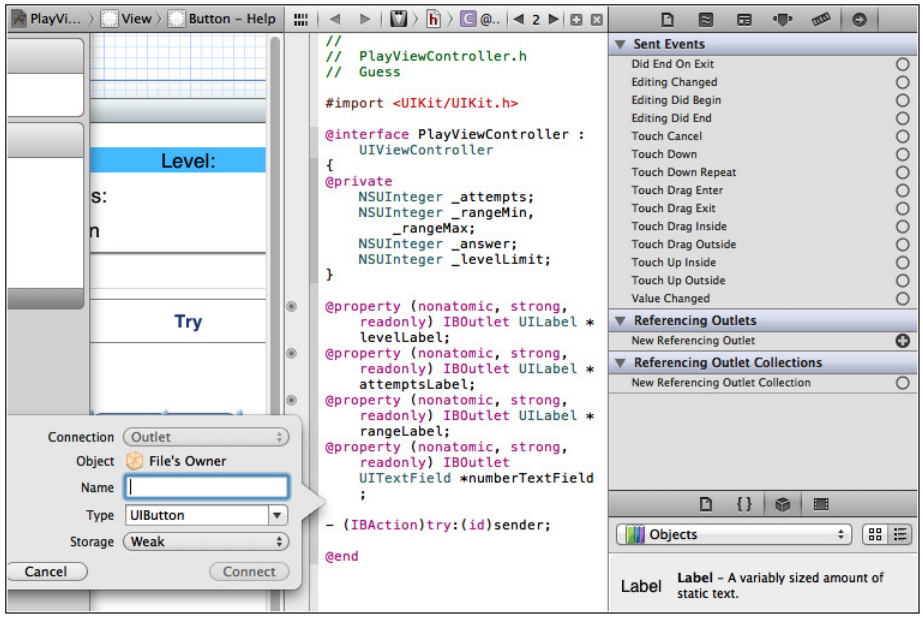



**FIGURE 8-9** Finalizing the creation of an outlet.

Creating an outlet edits the header file. Here's the header file of the view-controller class for the Play view:

```
@interface PlayViewController : UIViewController
{
@private
   NSUInteger _attempts;
   NSUInteger _rangeMin, _rangeMax;
   NSUInteger _answer;
   NSUInteger _levelLimit;
}
@property (nonatomic, strong, readonly) IBOutlet UILabel *levelLabel;
@property (nonatomic, strong, readonly) IBOutlet UILabel *attemptsLabel;
@property (nonatomic, strong, readonly) IBOutlet UILabel *rangeLabel;
@property (nonatomic, strong, readonly) IBOutlet UITextField *numberTextField;
- (IBAction)try:(id)sender;
@end
```

The header file has outlets for a few labels and one text box. It doesn't have any outlet for the button. What about adding event handlers to a button? In iOS, you add event handlers via *actions*.

The process of adding an action is similar to adding outlets except that you start dragging from another area in the Inspector pane. When you select a UI component, the Sent Events area of the Inspector pane is populated with the list of related events. All you do is drag from the selected

event to the source view of the header. You name the action with the name of the method in the implementation that will handle the event. In the previous code snippet, *try* indicates the handler for the touch-up (i.e., tap/click) event. Here is its implementation:

```
- (IBAction)try:(id)sender
{
            if (![_numberTextField.text isEqualToString:@""])
   {
          // Read and process the value the user has entered
          NSUInteger tryValue = _numberTextField.text.integerValue;
          if (tryValue > _answer && tryValue <= _rangeMax)
          {
              _rangeMax = tryValue;
          }
          else if (tryValue < _answer && tryValue >= _rangeMin)
          {
              _rangeMin = tryValue;
          }
          else if (tryValue == _answer)
          {
              // Calculate the score
              NSUInteger score = _levelLimit / _attempts;

              // Jump to the success view
              ResultViewController *resultViewController = [[ResultViewController alloc]
                           initWithNibName:@"ResultViewController" bundle:nil];

              // Update the view with the score
              resultViewController.result = score;
              [self.navigationController pushViewController:resultViewController animated:YES];
          }

          // Refresh the UI for another attempt
          _numberTextField.text = nil;
          ++_attempts;
          [self updateAttempt];
     }
}
```

When the user has guessed the hidden number, the code jumps to another view controlled by the *ResultViewController* class (see Figure 8-10).

**Note** In both the header and implementation files, there's no place where you can see the *try* method explicitly linked to a touch-up event. This binding is maintained in the source code of the XIB file—a plain XML file. However, the binding isn't clearly visible, even if you go to the effort of opening the XIB file. In fact, UI elements and events are identified in Interface Builder by number. In Interface Builder, the touch-up event has no human-readable name, but it is identified with the number 19.

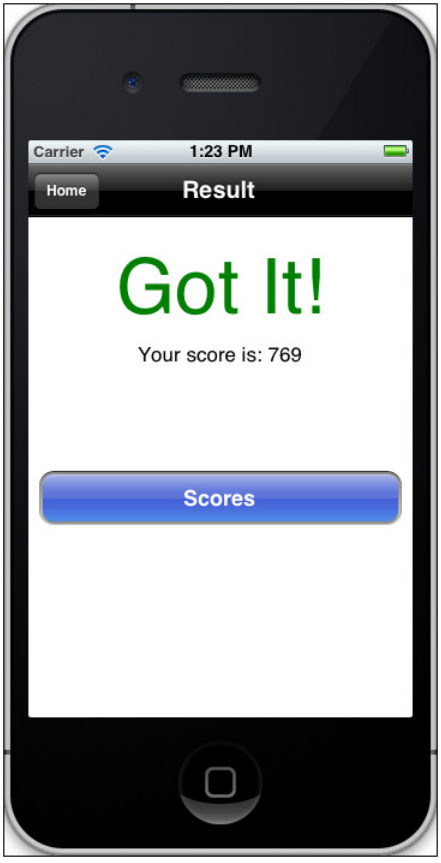



**FIGURE 8-10** The Result view.

**Important** In summary, there are two approaches to writing code that deals with UI components and their events. You can create UI components programmatically (e.g., using buttons, labels, text fields, and toolbars) by instantiating and placing them at a given position, or you can use Interface Builder and do the same work in a WYSIWYG manner. If you take the latter approach, outlets and actions are your way to gain programmatic control over the UI components. If you opt for the former approach, you create object references and can configure them (i.e., adding actions) at will.

## The Scores View

In the previous listing, although briefly commented, the following line didn't get the emphasis it actually needs:

```
// Assign the score of the current player to the Result view and ...
resultViewController.result = score;
```

The setter of the *result* property on the view-controller of the Result view takes care of saving the score in some persistent storage. In iOS (and, in general, in all mobile platforms), you have a few options to persist data. One is certainly based on a relational database. For iOS and Android, this option relies on the services of the SQLite database. Another option entails creating custom files. A third option is probably the simplest—using a system-provided persistent dictionary. Mostly created to let an application save a user's preferences, such a dictionary actually can be used to store any serializable data that can be identified with a unique key. In iOS, this dictionary is the *NSUserDefaults* class. The sample application uses the dictionary to create a single entry with the Players list and the best scores. The key is "Players," and the content is an array of (serializable) *Player* objects. The following code shows how to save a score to the dictionary:

```
// Gain access to the NSUserDefaults dictionary
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

// Read the value for the entry kPlayersArrayKey (a constant for "Players")
NSData *playersData = [userDefaults objectForKey:kPlayersArrayKey];

// Deserialize any content found to an array of Player objects
NSMutableArray *playersArray = nil;
if (playersData)
{
    playersArray = [NSKeyedUnarchiver unarchiveObjectWithData:playersData];
}
if (!playersArray)
{
    // Allocate a 1-element array
    playersArray = [[NSMutableArray alloc] initWithCapacity:1];
}

// Add the current player to the array
[playersArray addObject:currentAppDelegate.player];

// Add the array back to the dictionary
playersData = [NSKeyedArchiver archivedDataWithRootObject:playersArray];
[userDefaults setObject:playersData forKey:kPlayersArrayKey];
[userDefaults synchronize];
```

The Scores view reads data back from the dictionary and prepares a table of results. In doing so, it uses a *UITableViewController* that is very common in iOS applications. A table view is a collection of cells that is analogous to a single-column HTML table. Cells in a table view can have a default or custom template. The sample application uses a custom cell. Let's examine the source code of the *ScoresViewController* class. Here's the header file:

```
@interface ScoresViewController : UITableViewController
{
@private
   NSArray *playersArray;
}
@end
```

The implementation follows:

```
@implementation ScoresViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self)
   {
        self.title = @"Scores";
        NSData *playersData = [[NSUserDefaults standardUserDefaults]
                                    objectForKey:kPlayersArrayKey];
        if (playersData)
        {
            playersArray = [(NSArray *)[NSKeyedUnarchiver unarchiveObjectWithData:playersData]
                                      sortedArrayUsingSelector:@selector(compare:)];
        }
     }
     return self;
}
// Customize the number of sections in the table view
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
   return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
   return playersArray.count;
}

// Customize the appearance of table view cells
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
                              :(NSIndexPath *)indexPath
{
   static NSString *cellID = @"cellID";
   ScoresTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cellID];
   if (cell == nil)
   {
        cell = [[ScoresTableViewCell alloc]
                initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:cellID];
   }

   cell.selectionStyle = UITableViewCellSelectionStyleNone;
   cell.accessoryType = UITableViewCellAccessoryNone;
   cell.player = [playersArray objectAtIndex:indexPath.row];
   return cell;
}
@end
```

A table has a number of sections, and each section has a number of rows. A section is just a group of rows within the table with its own title. In this example, the table has one section that is as long as the list of players in the deserialized array.

A new cell is created on demand within the handler for the message *cellForRowAtIndexPath* on the *UITableViewController* class. You create an instance of the cell object that you want, configure it with the data you need, and return.

The cell of the sample application is made of three pieces of information: player name, date, and score. You start by creating a class as follows:

```
@interface ScoresTableViewCell : UITableViewCell
{
   UILabel *_nameLabel;
   UILabel *_scoreLabel;
   UILabel *_dateLabel;
}
@property (nonatomic, strong) Player *player;
@end
```

When initialized, the cell object receives a *Player* object. In the setter of the *Player* property, the three private members are set to the content of the *Player* object.

In iOS, a table cell can have some predefined position (and style). For example, the *textLabel* property on the parent *UITableViewCell* class refers to the title of the cell and is rendered in bold. On the other hand, the *detailTextLabel* property refers to a detail line and is rendered in gray and with smaller text. The sample code shown here first saves references to these members into local variables and then proceeds with some graphical work aimed at creating a custom area to the right edge for the score:

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier
{
    self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
    if (self)
   {
        // Get references to writable parts of the cell
        _nameLabel = self.textLabel;
        _dateLabel = self.detailTextLabel;

        // Create the area for the score (rightmost)
        CGRect scoreLabelFrame = CGRectMake(
                          self.frame.size.width - kScoreLabelWidth - kScoreLabelPadding,
                          0.0f,
                          kScoreLabelWidth,
                          self.frame.size.height);
        // Place a UILabel component in the area and add it to the cell view
        _scoreLabel = [[UILabel alloc] initWithFrame:scoreLabelFrame];
        _scoreLabel.textAlignment = UITextAlignmentRight;
        [self addSubview:_scoreLabel];
    }
    return self;
}
```

The cell layout is populated when the code assigns a value to the *Player* property of the custom *ScoreTableViewCell*. Here's the setter of the *Player* property:

```
- (void)setPlayer:(Player *)player
{
   _player = player;
   _nameLabel.text = _player.name;
   _scoreLabel.text = [NSString stringWithFormat:@"%d", _player.score];
   _dateLabel.text = [NSDateFormatter
                                  localizedStringFromDate:_player.scoreDate
                                  dateStyle:NSDateFormatterShortStyle
                                  timeStyle:NSDateFormatterShortStyle];
}
```

Figure 8-11 shows the final result.


**FIGURE 8-11** The Scores view.

## Other Programming Topics

This brief analysis of the Guess application has demonstrated the most common aspects of iOS and mobile programming. A couple of relevant APIs have been left out: accessing the network and performing common tasks, such as sending an email.

## Accessing the Network

The following code shows how to prepare a call to a given URL. You set the Hypertext Transfer Protocol (HTTP) address through the *NSURL* class and use the *NSMutableURLRequest* class to send it. Here's the initialization code:

```
NSURL *url = [NSURL URLWithString:@"http://yourserver.com/app"];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
```

To add content to the body of the HTTP request, you prepare a string like the following and stuff it into the *request* object:

```
NSString *params = [[NSString alloc] initWithFormat:@"foo=bar&key=value"];
[request setHTTPMethod:@"POST"];
[request setHTTPBody:[params dataUsingEncoding:NSUTF8StringEncoding]];
```

Finally, you fire the request using a call to the *NSURLConnection* object:

```
[[NSURLConnection alloc] initWithRequest:request
                         delegate:self];
```

To capture the response, it suffices that you use the following method:

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
```

The response that you get is of type *NSMutableData*. You can then transform this object into an array of bytes or a string.

## Common Tasks

In iOS and in other mobile operating systems, you have ready-made classes to perform a bunch of common tasks, such as sending an email, a text message, adding a contact, or placing a phone call. For example, in iOS, you use the *MFMailComposeViewController* class to display the standard email dialog box to the user. The class offers you methods to set recipients, subject, body, and even attachments:

```
MFMailComposeViewController *picker = [[MFMailComposeViewController alloc] init];
picker.mailComposeDelegate = self;
[picker setSubject:@"Hi there!"];

// Recipients
NSArray *toRecipients = [NSArray arrayWithObject:@"you@someserver.com"];
NSArray *ccRecipients = [NSArray arrayWithObjects:@"friend@someserver.com];
[picker setToRecipients:toRecipients];
[picker setCcRecipients:ccRecipients];

// Attach a file
NSString *path = [[NSBundle bundle] pathForResource:@"notes" ofType:@"txt"];
NSData *data = [NSData dataWithContentsOfFile:path];
[picker addAttachmentData:data mimeType:@"text" fileName:@"notes"];

// Set the body of the message
NSString *body = @"My two cents.";
```

```
[picker setMessageBody:body];
[self presentModalViewController:picker animated:YES];
[picker release];
```

The user, however, is allowed to send the email as is or manually edit any field. When done, the email is submitted to the iOS system mail application for actual delivery.

It should be noted that the device may not be configured to send emails programmatically. If not, you always can directly invoke the system mail application, as shown here:

```
NSString *recipients = @"mailto:you@someserver.com?cc=friend@someserver.com&subject=Hi there!";
NSString *body = @"&body=My two cents.";
NSString *email = [NSString stringWithFormat:@"%@%@", recipients, body];
email = [email stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:email]];
```

The key difference is that in the first case, you integrate a system dialog box within your application. In the second case, your application is pushed to the background and the default Mail application is launched.

In a similar way, you can deal with Short Message Service (SMS) messages. In this case, the class to use is *MFMessageComposeViewController*. Note, however, that you won't be able to send SMS in a fully automated way without the user explicitly tapping the Send button.

## Navigation and Controllers

We briefly touched on navigation while discussing the sample application. Let's review what it takes to move to a different view. Applications typically have a navigation controller that is set upon launching, as follows:

```
self.navigationController = [[UINavigationController alloc]
```

The navigation controller keeps track of the various views displayed and organizes them into a stack. Every view that you navigate to since the application first displayed is added to the stack. Navigating to a new screen just adds a new view to the stack; navigating back just pops the topmost view from the stack. The default UI navigation controller takes care of the functionality of the back button that brings you back to the previous screen in the stack.

The following code shows how to display a new view:

```
ResultViewController *resultViewController = [[ResultViewController alloc]
                          initWithNibName:@"ResultViewController" bundle:nil];

[self.navigationController pushViewController:resultViewController animated:YES];
```

You first get a new instance of the controller for the next view, and then you send the *pushViewController* message to the navigation controller that identifies the controller of the new view as the argument. As an additional argument, you can indicate an animation parameter. Animated transitions are an important part of the user experience, so you might want animation turned on.

By default, when you push a new controller on the stack, a back button appears with the caption set to the title of the previous view. The *UINavigationController* class also has a method you can use to jump to the home page. To navigate to the home page, you need to add your own button and attach it to an action, as shown here:

```
(void)backToHome
{
   [self.navigationController popToRootViewControllerAnimated:YES];
}
```

Another popular type of controller is the *UITabBarController*. It renders a tab bar interface that is fairly convenient when you want to provide different views about the same data, or perhaps keep the functions well separated. The *UITabBarController* manages the tab view user interface and the controllers responsible for the various child views. It's like having a double level of control: Each tabbed view has its own controller, and all tabs are managed by the *UITabBarController* and displayed as the user taps.  Here's how to create a tab bar:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
       tabBarController = [[UITabBarController alloc] init];
       FirstViewController* tab1 = [[FirstViewController alloc] init];
       SecondViewController* tab2 = [[SecondViewController alloc] init];
       NSArray* controllers = [NSArray arrayWithObjects:tab1, tab2, nil];
       tabBarController.viewControllers = controllers;
       window.rootViewController = tabBarController;
}
```

For each tab, you provide a *UITabBarItem* object that indicates text and image to display in the user interface. You associate a *UITabBarItem* object with the view-controller of a child tab at any time before the tab bar is displayed.

## Programming with MonoTouch

A large number of developers who need to write iPhone applications have a strong background in Java or .NET. In both cases, learning Objective-C, although not impossible, is definitely a time-consuming (and expensive) task. What other choice do you have?

MonoTouch is a commercial framework that allows you to use the C# language to write iOS applications. Not only can you use a truly object-oriented, modern, and elegant language to write iPhone and iPad applications, but also you can use a subset of the .NET Framework for common tasks.

The MonoTouch framework is built on top of the Mono framework and is sold by Xamarin (*http://www.xamarin.com*), the same company that built Mono—a cross-platform compiler and language run time that brings .NET well outside the Windows platform. Mono packages, in fact, exist for a few Linux distributions, Mac OSX, and Solaris.

## The .NET Framework on iOS

Although MonoTouch is not the only alternative to using Objective-C and Xcode for writing iOS applications, it is probably the most flexible one. With MonoTouch, you have nearly full coverage of the iOS SDK features, plus the ability to invoke native Objective-C or C/C++ compiled code. This means that you can incorporate Objective-C code in your projects, as well as .NET assemblies produced by Visual Studio. Let's find out more about the internal architecture of MonoTouch and then proceed with a simple example.

### From Mono to MonoTouch

You typically write a MonoTouch application on a Mac computer using the C# language. It's not for the faint-hearted, but you could even write your source code with Visual Studio. In any case, you need a Mac to compile your sources for iOS and test on the emulator.

In your code, you call classes whose naming and programming interfaces are the same as the classes in the .NET Framework. In reality, though, you have no .NET Framework assemblies deployed in any form to your Mac. The classes you call belong to the Mono software platform.

Mono is a collection of software tools aimed at making the .NET Framework available on a variety of alternate platforms—Mac OSX is just one of these. The Mono framework comprises a bunch of tools for each platform, including a C# compiler, a run-time environment, a version of the Base Class Library (BCL) compatible with the original .NET Framework, and the Mono Class Library (MCL), which includes classes for capabilities that aren't in the .NET Framework but still are needed on non-Windows platforms.

The Mono run time is particularly important for iOS applications. The run time contains subsystems for threading, interoperability, and garbage collection. It also contains the Just-in-Time (JIT) compiler that turns intermediate code to binary code on the fly. Unfortunately, Apple doesn't accept any code that requires JIT compilation in the App Store, the only place to distribute your iOS applications.

For this reason, the Mono run time includes an Ahead-of-Time (AOT) compiler that runs past the standard C# compiler and reduces your application to native iOS code. The AOT compiler also enables you to use generics, a feature that, according to .NET, would require JIT compilation.

In summary, you write your C# code, and then the combination of the Mono and AOT compilers turns it into the .NET Intermediate Language (IL) first and then directly into ARM assembly code, the low-level language for iOS devices.

**Note** Processors used by Apple devices—iPhone, iPod Touch, and iPad—are A4 and A5 processors developed internally by Apple but compatible with the ARM architecture. This same architecture will be supported by Microsoft on Windows 8.

## Pillars of MonoTouch Applications

MonoTouch applications are compiled against a subset of the BCL of .NET Framework 4. This specific profile required by MonoTouch applications makes it impossible to use existing .NET assemblies without prior recompilation. The MonoTouch profile is roughly equivalent to the Silverlight 4.0 profile of the Windows-based .NET Framework.

So your code may target classes like *System.Net.WebClient* and use the usual .NET interface of such classes. Under the hood, though, this and other classes are bound to iOS native classes and features. For this reason, you also need the iOS SDK installed to write MonoTouch applications. As mentioned, the level of coverage of iOS features that you'll find in MonoTouch is nearly 100 percent.

In addition to the iOS edition of BCL classes, MonoTouch provides a bunch of classes that are specific to iOS applications. The application model, in fact, is the same as if you use Objective-C. This means that you still will have a *UIApplication* object around events like *FinishedLaunching*. The user interface will be based on UIKit elements such as *UILabel, UIButton,* and *UISegmentedControl*. The name of properties and constants are preserved or slightly adapted. In the end, using MonoTouch is like programming for classic .NET, but using a set of new namespaces and assemblies.

## Analysis of a Simple MonoTouch Project

You write a MonoTouch application using MonoDevelop, a cross-platform IDE that runs on Linux and Windows as well as Mac. For developers coming from the .NET world (or even the Java world), MonoDevelop is a more familiar tool than Xcode—or at least that has been the experience of my co-workers and myself. Figure 8-12 offers a view of the MonoDevelop IDE for iOS applications.

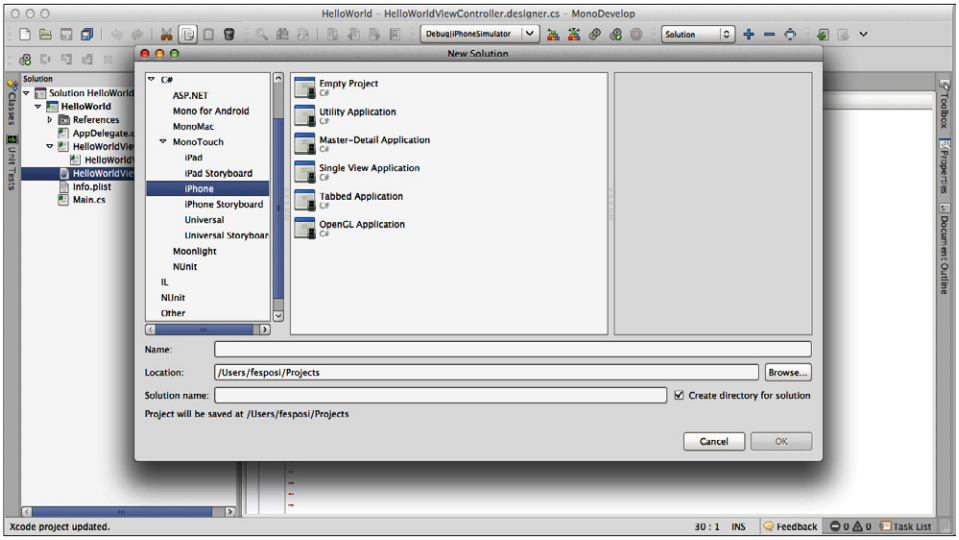



**FIGURE 8-12** The MonoDevelop IDE.

After creating a HelloWorld application, your MonoDevelop project contains a list of files that is nearly identical to an Objective-C Xcode project, as shown in Table 8-1.

**TABLE 8-1** Files in a Simple MonoTouch Project

| File | Description |
|---|---|
| Main.cs | Application starter and main entry point. |
| AppDelegate.cs | The delegate for the application responsible for starting the user interface, as well as listening (and optionally responding) to application events from iOS. |
| HelloWorldViewController.cs | Controller of the main view of the application. In this case, HelloWorldViewController is the only screen of the application. |
| HelloWorldViewController.designer.cs | Designer class for the view; it contains references to visual elements declared in the XIB file. This is a plain .NET partial class. |
| HelloWorldViewController.xib | Standard XIB file defining the UI of a view. This file will be edited, opening an instance of Interface Builder. |

In particular, the Main.cs file starts the application, as shown here:

```
using System;
using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace HelloWorld
{
   public class Application
   {
        // This is the main entry point of the application.
        static void Main (String[] args)
      {
            // Feel free to indicate another AppDelegate class name
            UIApplication.Main (args, null, "AppDelegate");
      }
   }
}
```

The *AppDelegate* class has a slightly more compact structure than it does in Xcode:

```
[Register ("AppDelegate")]
public partial class AppDelegate : UIApplicationDelegate
{
   UIWindow window;
   HelloWorldViewController viewController;

     public override bool FinishedLaunching (UIApplication app, NSDictionary options)
   {
         window = new UIWindow (UIScreen.MainScreen.Bounds);
         viewController = new HelloWorldViewController ("HelloWorldViewController", null);
         window.RootViewController = viewController;
         window.MakeKeyAndVisible();
         return true;
   }
}
```

The *Register* attribute is used to register a MonoTouch class explicitly with the Objective-C run time, using a short name. The *FinishedLaunching* overridden method performs any tasks that are

required to prepare the view. In this method, the application's window is created to cover the entire screen and the view-controller instance is created and attached to the window. Finally, the window is displayed and given input focus. It should be noted that this method is given a fixed amount of time to return. If it stalls for more than a few seconds, iOS will stop the process.

The designer class has a structure that is very similar to analogous .NET classes that you may know from ASP.NET or Windows Forms programming:

```
[Register ("HelloWorldViewController")]
partial class HelloWorldViewController
{
   [Outlet]
   MonoTouch.UIKit.UITextField quantityTxt { get; set; }

   [Outlet]
   MonoTouch.UIKit.UITextField percentageTxt { get; set; }

   [Outlet]
   MonoTouch.UIKit.UILabel resultLabel { get; set; }

   [Outlet]
   MonoTouch.UIKit.UIButton calcButton { get; set; }

   [Outlet]
   MonoTouch.UIKit.UIButton clearButton { get; set; }
}
```

The designer file is created automatically by MonoDevelop and stores any outlets and actions that are created in Interface Builder. In the previous code, you see references to a bunch of visual elements that produce the view shown in Figure 8-13.

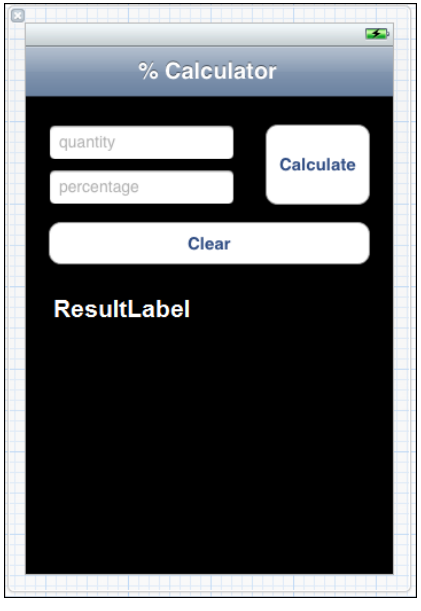



**FIGURE 8-13** A sample view for a simple percentage calculator.

As the XIB file is edited in Interface Builder, you add outlets and actions as described in the section "Using Objective-C," earlier in this chapter.

Note that in MonoTouch, you also can handle actions as events using the .NET Framework syntax, except that you must use iOS names. Here's the code that programmatically defines an action for the *click/tap* event on a button:

```
calcButton.TouchUpInside += delegate { ... };
```

In event handlers, you can access BCL classes and employ your existing .NET skills.

### Reusing Existing .NET Code

Code reusability with .NET also reaches significant levels. It should be noted that you can't just drop in existing assemblies that you may have created for .NET projects. Any assemblies that you import must be compiled for the specific MonoTouch .NET profile. This means that reusability applies, but at the source level and so long as referenced classes exist in the target profile.

In addition, when evaluating MonoTouch, consider that MonoTouch is just the iOS leg of a single product that lets you address the Android platform as well. MonoTouch, in fact, is paired with MonoDroid, which features the same idea and patterns, but applied to the Android platform. This is another great advantage, as your .NET code now may be ported to a significant extent across .NET, Windows Phone, iOS, and Android.

There's no magic here, so don't expect miracles; you can simply count on a reliable and effective cross-platform solution.

## Examining a Sample Application

InstantScore is a small application that, just for its overall simplicity, demonstrates the power of the mobile paradigm. Mobile applications don't have to be overly complex to become useful to customers. The idea behind, and the selection of, use-cases is what really determines the success of an application. Without a good idea that addresses a specific need, you will hardly have a successful mobile application.

The sample application examined in this section connects to a web service and reports the status and score of a few tennis matches. The first version of this application was written in less than a couple of hours by a junior .NET developer using MonoTouch for the first (or maybe second) time. The application allowed a few people to stay constantly informed about ongoing matches during a tennis tournament and helped them to estimate how long they could relax before their next shift on a given court.

This application was never submitted to the App Store; we simply managed to deploy it as a beta application to a couple of specific devices. (I'll say more about the deployment of iOS applications at the end of the chapter.)

## A Master/Detail View

The master/detail pattern is fairly common in mobile views. You provide a scrollable list of items, the user picks one, and a new screen replaces the old one. The *AppDelegate* of the sample application defines the controller of the main view as a standard *UINavigationController* class:

```
public override bool FinishedLaunching (UIApplication app, NSDictionary options)
{
   window = new UIWindow (UIScreen.MainScreen.Bounds);
   var controller = new RootViewController();
   navigationController = new UINavigationController (controller);
   window.RootViewController = navigationController;

   // Make the window visible
   window.MakeKeyAndVisible ();
   return true;
}
```

The root view-controller derives from the *UITableViewController* class. It sets the application title and connects to the web service to get the list of matches. The whole process is coordinated from the *ViewDidLoad* method:

```
public partial class RootViewController : UITableViewController
{
   public RootViewController () : base ("RootViewController", null)
   {
        Title = "Live";
   }

   public override void ViewDidLoad ()
   {
         base.ViewDidLoad ();

         // Download matches
         var list = InstantScoreHelpers.DownloadCurrentMatches();

         // Bind the list of matches to the table view
         TableView.Source = new MatchesDataSource (this, list);
   }
     ...
}
```

It is interesting to notice that in this previous code, you can arrange for the download to take place synchronously or asynchronously. In Windows Phone and Silverlight, on the other hand, it can happen only asynchronously:

```
// Download content in a synchronous manner
public static IList<MatchItem> DownloadCurrentMatches()
{
   var webClient = new WebClient();
   var content = webClient.DownloadString(new  Uri("http://..."));
   return Parse(content);
}
```

The *MatchItem* class just defines the match entity being handled, as follows:

```
public class MatchItem
{
   public String Icon {get; set;}
   public String Text {get; set;}
   public String Score {get; set;}
}
```

As is, the code crashes if there's no Internet connectivity or the host is not reachable for any reason. In MonoTouch, you use the *NetworkReachability* class to find information about the reachability of a given host:

```
var nr = new NetworkReachability(host);
```

To get a binary answer, you need a couple of other lines of code. One of the MonoTouch SDK examples, though, provides a helper class—the *Reachability* class—that you can invoke in a very natural way in case you don't need to distinguish between WiFi and 3G connectivity:

```
if(!Reachability.IsHostReachable(host))
{
    // Skip
}
```

Once you've got the list of matches, the next step consists of populating the table. Figure 8-14 shows the layout of the main view.

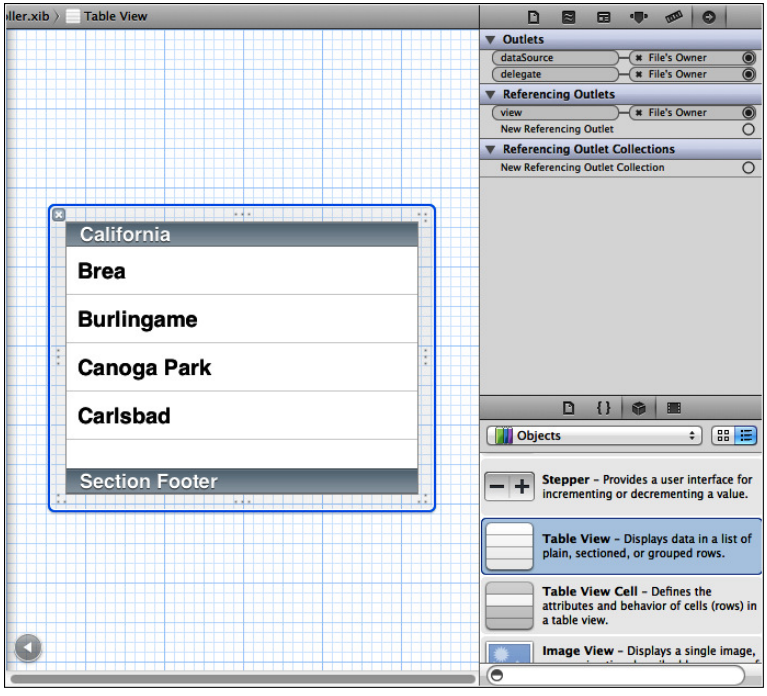



**FIGURE 8-14** The main view of the sample application.

## Populating the Table View

Any controller that derives from *UITableViewController* has a *TableView* property of type *UITableView*. You need to pass a couple of objects to this table view: the data source to populate the table and a delegate to handle user activity. The *TableView* object has two distinct properties for this—*Source* and *Delegate*—that you can set separately:

```
TableView.Source = new MyDataSource();
TableView.Delegate = new MyTableDelegate();
```

The data source will derive from *UITableViewDataSource,* whereas the delegate object derives from *UITableViewDelegate*.

You also can set the *Source* property with an instance of a class that derives from *UITableViewSource*. This base class combines the capabilities of a delegate and a data source. Here's an example:

```
public class MatchesDataSource : UITableViewSource
{
   RootViewController _controller;
   IList<MatchItem> _matchList;

   public MatchesDataSource (RootViewController controller, IList<MatchItem> currentMatches)
   {
         _controller = controller;
         _matchList = currentMatches;
   }

   public override int NumberOfSections (UITableView tableView)
   {
         return 1;
   }

   public override int RowsInSection (UITableView tableview, int section)
   {
         return _matchList.Count;
   }

   public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
   {
         string cellIdentifier = "Cell";
         var cell = tableView.DequeueReusableCell (cellIdentifier);
         if (cell == null) {
               cell = new UITableViewCell (UITableViewCellStyle.Subtitle, cellIdentifier);
               cell.Accessory = UITableViewCellAccessory.DisclosureIndicator;
         }

         // Configure the cell
         cell.TextLabel.Text = _matchList[indexPath.Row].Text;
         cell.DetailTextLabel.Text = _matchList[indexPath.Row].Score;
         cell.ImageView.Image = UIImage.FromFile(_matchList[indexPath.Row].Icon);
         return cell;
   }
```

```
   public override void RowSelected (UITableView tableView, NSIndexPath indexPath)
   {
         // More code here
         ...
   }
}
```

The data source class overrides *RowsInSection* and *NumberOfSections* to set the length and structure of the table. Another method that is mandatory to use in a data source class is *GetCell*. As the name suggests, *GetCell* returns the cell object to add to the table during rendering.

In the implementation of a table view, you usually don't create one cell for each data item. Before you create a cell, you ask the table view to get one cell of the specified type. The cell type is referred to as the cell identifier:

```
string cellIdentifier = "Cell";
var cell = tableView.DequeueReusableCell (cellIdentifier);
```

If the table view can't return a valid cell object, then you create a new one:

```
if (cell == null) {
   cell = new UITableViewCell (UITableViewCellStyle.Subtitle, cellIdentifier);

   // The cell will have an arrow (>) at the end indicating that there's
   // another view associated with the row. Other accessories are checkmark (if
   // the row is selected) and + symbol (if a new row can be added to the table).
   cell.Accessory = UITableViewCellAccessory.DisclosureIndicator;
}
```

The table view maintains a pool of cell objects and groups them using the cell identifier. You don't need to use different identifiers if your table is populated by cells of the same type. If you need to display cells of different types (e.g., cells with different layouts), then you might want to assign an identifier to each type and have the table view create multiple pools of cell objects. The table view returns a plain cell object, and it is your responsibility to configure it with the content of the current data item. That is done as follows:

```
cell.TextLabel.Text = _matchList[indexPath.Row].Text;
cell.DetailTextLabel.Text = _matchList[indexPath.Row].Score;
cell.ImageView.Image = UIImage.FromFile(_matchList[indexPath.Row].Icon);
```

The index of the row is expressed as an *NSIndexPath* object, which represents the path to a specific element in a hierarchy of nested collections. You need to invoke the *Row* property to get the real index if you have a single collection of elements.

Finally, the content of the table view can be modified by adding, deleting, and reordering rows. This can be done either programmatically, with the user interface providing clues on how to do that, or through built-in row accessories that use some capabilities of the table view object. Another interesting facility that you find in the table view object is the ability to refresh its content by reloading data. You use the *reloadData* method to do this.

**Important** The working of the *UITableView* object, discussed here in the context of MonoTouch, is exactly the same one you would experience coding while in Objective-C.

## The Detail View

The *RowSelected* method contains the logic that triggers when the user selects a given table row. If you intend to navigate to another view, the first thing to do is create a controller for the detail view, as follows:

```
public override void RowSelected (UITableView tableView, NSIndexPath indexPath)
{
    // Prepare the next view
    var DetailViewController = new DetailViewController ();
    ...

    // Move to the next view
    controller.NavigationController.PushViewController (DetailViewController, true);
}
```

If you're having a detail view, then you also likely have some data to pass from the master view to the detail view. In the sample application, the detail view contains references to the selected match. Here's an excerpt from the *DetailViewController* class:

```
public class DetailViewController : UIViewController
{
   MatchItem _currentMatch;

   public void SetDetailItem (MatchItem match)
   {
       if (currentMatch != match) {
             currentMatch = match;
             ConfigureView ();
       }
   }

   void ConfigureView ()
   {
       if (currentMatch!= null) {
           Title = _currentMatch.Text;
           ...
       }
   }
}
```

You call *SetDetailItem* from within *RowSelected* to pass information from the master view to the detail view:

```
public override void RowSelected (UITableView tableView, NSIndexPath indexPath)
{
    // Prepare the next view
    var detailViewController = new DetailViewController ();

    // Pass data
    detailViewController.SetDetailItem(_matchList[indexPath.Row]);

    // Move to the next view
    controller.NavigationController.PushViewController (DetailViewController, true);
}
```

Figure 8-15 shows the resulting master and detail views.


**FIGURE 8-15** Two views of the sample application.

## Objective-C or MonoTouch: Your Take?

MonoTouch provides wrappers for Objective-C objects and entirely relies on Cocoa Touch and native iOS frameworks under the hood. Is there any value in using MonoTouch over Objective-C?

Admittedly, this is a highly debatable point in which a few facts can be observed but conclusions remain quite personal. A key fact is that you need to understand the logic of the iOS user interface to build MonoTouch applications. MonoTouch may shield you from some—but not all—of the frameworks that form Cocoa Touch. Further, when you use MonoTouch (or MonoDroid), the code that you reuse, or the code where you can use your .NET skills, is generally the back-end code and any helper code (which isn't really a bad thing anyway). The logic behind the presentation layer remains specific to the platform of choice.

At the very end of the day, picking up Objective-C/Cocoa Touch versus C#/MonoTouch is a choice based on the programming language and the framework that you like most. If you are a .NET person, you immediately feel at home with MonoTouch. But if you already managed to learn Objective-C, regardless of your background, you may find little value in moving to MonoTouch.

Furthermore, a MonoTouch application is about 10 times larger than an iOS application built with the native Apple SDK. It's a matter of a few megabytes versus a few hundred kilobytes. So long as your application doesn't exceed the limit that Apple allows for downloading applications over a 3G connection (currently set to 20 MB), the size is not typically a big concern for iOS applications, where most of the time you have at least a 16 GB on-board memory card for iPhone and iPad and a 8 GB card for iPod Touch devices. When you consider MonoDroid—the MonoTouch's twin framework for Android—to write Android applications, the size becomes much more critical, given the wide range of devices with different hardware capabilities that you encounter.

If you haven't come to a clear decision yet, then these additional aspects may help you. (I don't believe that developers with a strong orientation towards either Objective-C or MonoTouch will change their mind because of the following points, though.)

- The Objective-C community is much richer than MonoTouch's. This means that you will find a lot more examples, components, tutorials, and various contributions to support you. This isn't surprising (the iOS SDK has been around for longer), nor does it mean you have insufficient documentation from Xamarin and won't have a community of MonoTouch users to help you.
- You can write code switching between MonoDevelop and Xcode, which still is used for UI tasks.

What's my personal take for iOS development? It's easy: I feel physical pain when I'm exposed to some Objective-C syntax for longer than five minutes or so. Having learned enough about Cocoa Touch, I do use MonoTouch with ease for the presentation layer and enjoy the pleasure of using .NET-like classes for logic, network, and storage chores.

# Deploying iOS Applications

Applications written for iOS are packaged into *application bundles* and downloaded to the device. An application bundle contains any code and resources associated with an executable program. The bundle is like a directory with a fixed structure that stores executables and resources in specific locations. Your IDE of choice—whether Xcode or MonoDevelop—is responsible for producing an application bundle. The Info.plist file in the project is your way to configure naming, structure, and other parameters of the application bundle.

The bundle gets deployed to a device and executed. The details of how this apparently simple deployment process takes place are a bit tricky. First and foremost, you can't deploy a freshly compiled bundle on just any device. The bottom line is that there are two ways for a bundle to make it to a physical iOS device: the bundle is available from the App Store and can be installed on any iOS device or the bundle is created for a specific set of devices and can be installed on those devices (a device is identified via a unique ID).

Let's find out more about testing and distributing the application.

## Testing the Application

So you finally wrote a few lines of code that may not contain any errors. You proudly click the Run button, and the IDE compiles your code and makes it ready to run. What happens next? The freshly created bundle needs be installed on some device—either physical or virtual.

### Joining an Apple Development Program

All mobile SDKs come with a tool that almost can simulate the behavior of the real device, and iOS is no exception. Regardless of the IDE and frameworks that you use to write your iOS application, you always rely on the Apple iOS simulator to test your work. The first option that you have to run your application is just deploying to a virtual device, the iOS simulator.

So long as you only test on the simulator, you don't have to pay anything to anybody. For example, Xcode deploys compiled code to the simulator virtual machine for free. The same goes for MonoTouch.

To deploy on a physical device instead, the first thing that you need to do is enroll in one of Apple's paid development programs. For more information, see *http://developer.apple.com/programs/start/ios*. Ad hoc programs are also available for companies.

If you use MonoTouch, you also need to get a MonoTouch license; the trial version of MonoTouch doesn't allow you to deploy to a physical device.

### Getting Your Development Certificate

The first time that you run Xcode after becoming a registered developer, Xcode creates a development certificate for you and stores it in your Mac computer's keychain. A development certificate assigns you a private key and identifies you as an iOS developer. Any further use of this development certificate is transparent to you.

At the end of each compile session, Xcode uses the private key from your development certificate to sign the application bundle. If the certificate is somehow invalid, the build fails.

Being a registered (and paying) developer is not enough to enjoy the thrill of seeing your application running on an iPhone, however. As a general rule, in the Apple world, deployment of applications is strictly controlled. You can't install applications directly on just any device. By "directly," I mean that you can't copy a signed application bundle yourself to just any iOS device. Deployment must go through Xcode, iTunes, or direct download from the App Store.

For Xcode to install a freshly compiled bundle to a device, another certificate is required that ties together a developer identity (your development certificate), an application ID (determined by the project), and one or more specific devices. This is known as the *provisioning profile*.

For an iOS to run on a device, at least one provisioning profile is required. How can you get one? And how can you identify a device?

## Registering a Device

Xcode can manage the provisioning profile for you and make it entirely transparent, so long as you are using a device registered for development. If you have a "development" device connected to the computer, all that's required to test the application live on that device is hit the Run button. Under the hood, Xcode will manage certificates and provisioning profiles for you.

A "development" device is an iOS device whose unique identifier (UDID) has been registered with Apple and associated with your developer account. You can do that from Xcode by connecting the device to the computer and using the Organizer submenu of the Window menu.

As an alternative, you can register a device manually by connecting to the iOS Provisioning Portal at *http://developer.apple.com/ios/manage/overview/index.action*. Once connected to the portal, you can manage your devices. Figure 8-16 shows the page of the portal where you can see the details of a particular device and the list of profiles it is associated with. A device, in fact, can be associated to multiple provisioning profiles; similarly, a provisioning profile can include multiple devices.

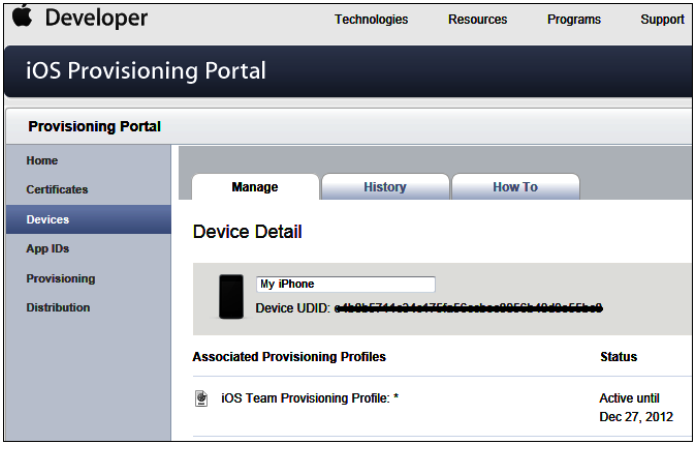



**FIGURE 8-16** The iOS Provisioning Portal.

If you choose to register a device manually within the portal, then you are required to enter the UDID of the device. The UDID is a 40-character string that uniquely identifies an iOS device much as a serial number does. The UDID is an Apple-only number, and it is different from the International Mobile Equipment Identity (IMEI) number used by the GSM network to identify devices. You can read the UDID of a phone by connecting it to the computer and launching Xcode or iTunes. You also can read the UDID directly from the device by navigating to the Settings page.

**Note** You are allowed to register up to 100 non-unique development devices per year of enrollment. Any UDID that you add counts toward that threshold, even if it is a duplicate or you deleted another one. In other words, if you remove a previously added device, you won't get back the entry until your year of enrollment expires and is renewed. Moreover, if you add the same device again, it consumes one more entry. You'll be credited 100 new entries the first day of any new enrollment. For example, if you enrolled on January 1, you'll receive 100 new entries on January 2 of the next year (provided that you renew, of course).

## Getting the Provisioning Profile

Once the developer certificate is installed and you have a registered device connected to the computer, you simply build the application and tell Xcode that you want to test on the attached device. In this case, Xcode gets a development provisioning profile automatically.

As an alternative, you can connect to the iOS Provisioning Portal, create the profile manually, and download it to the connected device. Once created, the development provisioning profile is installed on the device and needs no further work for successive builds.

## Enabling Beta Testing

Beta testing indicates a scenario in which you want to distribute your application to a relatively large number of users. Stepping into the beta testing stage is surprisingly simple: all you need is an ad hoc distribution provisioning profile that enables a given application to run on a bunch of selected devices.

A distribution provisioning profile is different from a development provisioning profile in that it requires a different type of certificate—a distribution certificate versus a development certificate—in addition to an app ID and a list of enabled devices. A distribution certificate enables the owner of the account to publish applications for one year. The certificate can be revoked and re-created at any time from the provisioning portal.

There are two types of app IDs: wildcard and explicit. A profile with a wildcard app ID can be used to install and start any applications that don't have an explicit App ID set in the bundle. A profile with an explicit app ID restricts to install only the specified application. The following is the default App ID which sets up for a wildcard provisioning profile:

```
ABCDE12345.*
```

The first part of the App ID is the bundle seed ID automatically generated for a given developer account. The second part—the asterisk—indicates the wildcard for the application name. You typically use this App ID for most of your Xcode development, but resort to an explicit App ID when you want to distribute your application for beta testing. The following string indicates an explicit App ID:

```
ABCDE12345.MyApp
```

Before creating a distribution provisioning profile, you register the App ID with your account. Next, you pick up a distribution certificate and select all the devices the application is enabled to run on. All devices must be registered as development devices using their UDID.

There's no distinction between the devices that you use for development and the devices of your beta testers: both must be registered, and both contribute to reach the yearly threshold of 100 devices. You probably need more accounts if you reach the maximum number of devices per account. Figure 8-17 shows the page on the iOS Provisioning Portal that allows you to create an ad hoc distribution provisioning profile.

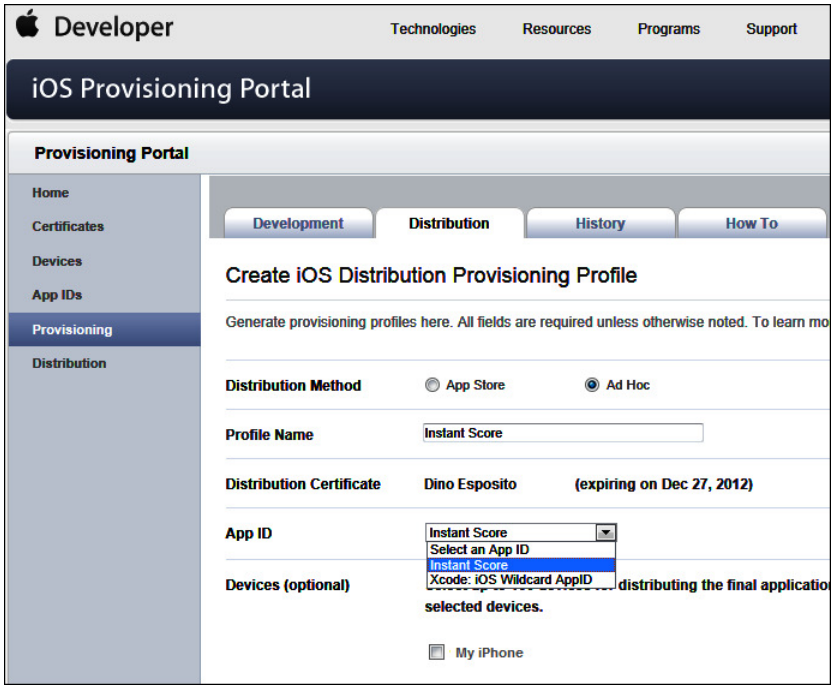



**FIGURE 8-17** Creating a new distribution provisioning profile.

Sharing beta builds with testers may be your next problem. Once you have selected the beta profile in Xcode, and Xcode has successfully compiled the application against that profile, you deliver the resulting package (an *.ipa* file) to testers via email or in some other way. The IPA package contains both the executable and profile. Testers can install the IPA file on their devices using iTunes.

### Over-the-Air Beta Testing

To reduce the burden of beta testing an application, you can use the TestFlight service, which is available at *http://testflightapp.com*. As a developer, you sign up with the service and create a team of testers for a given application. A team of testers consists of a collection of UDID strings. Once you have a distributable IPA that works against those testers, you upload the bits to the site and distribute. The site then will make your application available for download to testers.

As a tester, you sign up to the TestFlight service and register your device(s) with the site. The site offers a number of ways for you to show up and be invited to test an application. In any case, if you accept the task of testing an application—you need to be invited by the developer; you can't pick up an application to test yourself—you'll receive via email a link to a copy of the application that will run just on your device.

TestFlight is a very effective way to share your beta builds with a restricted number of users and, maybe more important, is an effective way to find beta testers. The site shortens your testing phase and allows you to address and solicit feedback more quickly.

## Distributing the Application

A developer account gives you the right to deploy applications on no more than 100 devices. Subsequently, the only possible way to sell or just distribute your application is to upload it to the Apple marketplace—the App Store.

**Note** The App Store channel is the only possible way to install applications on an iOS device unless you *jailbreak* the device. Jailbreaking consists of installing a custom kernel that removes the limitations imposed by Apple and allows users to download applications outside the official App Store. Once you jailbreak your device, you gain root access to it and can manage it directly without using iTunes for everything. A jailbroken device can connect to the App Store and functions just like a regular device. On July 26, 2010, the U.S. Library of Congress affirmed that the practice of jailbreaking is not a violation of copyright law. This statement, though, must be reviewed and reiterated every three years.

### The App Store

The App Store is the official Apple online platform for distribution of iPad, iPhone, and iPod Touch applications. Users access the App Store via iTunes, the web, or directly from their devices through native applications.

The idea behind the App Store is to provide a single place where users can look for applications and developers can publish applications. With some modification, the idea of a centralized application store has been adopted by other mobile platforms such as Android and Windows Phone. An official application store is also slated for debut with the Windows 8 platform.

Publishing an application to the App Store is a multistep process. First, you package the application for App Store distribution. Next, you submit the application for approval. Finally, once the application has been certified and signed by Apple, it goes to the App Store and becomes available to the masses. At this point, it can be installed on any iOS device that it was compiled for.

To submit an application to the App Store, you need a distribution provisioning profile that's valid for the App Store. You achieve this by doing what is shown in Figure 8-17, except that you switch to the App Store distribution method. Once you hold the distribution profile for the App Store, you compile your application against that, submit it, and wait for approval.

If your application behaves well, doesn't crash, and is respectful of the guidelines, it will be published. The process to get a response from Apple usually takes a few days, but the wait time can vary. However, it should never take more than two weeks to get a response. It should be noted that you undergo the same process—submit, wait, and maybe retry—when you publish an update as well. This happens regardless of the quality of the update, whether it is a full new release with new features or just a quick bug fix.

You can publish free applications or sell them for a minimum price of $0.99. Apple offers free hosting and marketing but deducts 30 percent of your revenues at the source. For free applications, though, the App Store is totally free.

**Note** Any applications published to the App Store remain there so long as your developer account is in good standing. If you fail to renew your membership, Apple will remove your applications from the App Store. In addition, Apple won't accept any new submissions until you join again. However, downloaded applications will continue working on the devices.

## In-House Deployment

In-house deployment is useful when you build the application for internal purposes and want to distribute it only to your employees and/or contractors. The whole process is nearly the same as with beta testing, except for the type of license required and certificates involved. In particular, to enable in-house deployment, you need a Developer Enterprise Program license.

The Developer Enterprise Program license enables you to distribute the application to up to 500 devices. With a basic ad hoc distribution, you can reach up to 100 devices. Note that the distribution certificate expires after one year. However, applications published to the App Store or distributed privately via in-house deployment don't expire automatically.

To compare the Developer Program and Developer Enterprise Program, refer to *http://developer.apple.com/programs/start/ios*.

**Important** To publish applications to the App Store, you need a developer license, acquired as an individual or a company. If you enroll in the Developer Enterprise Program, you get other benefits (i.e., in-house deployment), but you will not be able to publish applications to the public App Store.

## Summary

When it comes to mobile application development, iOS is the first platform to take into account. How would you write iOS applications? Most of the existing applications have been written using the iOS SDK and the Objective-C language. Objective-C is an extended version of the C language. It offers a richer syntax that allows objects and messages to be handled in a sort of object-oriented approach. Although there are no difficult new concepts to learn in Objective-C, its syntax looks rather weird, at least compared to the elegance of Java and C#. To cut a long story short, if you're OK with the language syntax and find yourself making progress with it, by all means, do your iOS development with the native iOS SDK and related frameworks.

What other options do you have?

This chapter presented MonoTouch, a framework that offers a C# compiler and a set of .NET-like classes. If you are escaping from Objective-C, using MonoTouch is like being in heaven. Regardless of whether you use MonoTouch or Objective-C, this chapter touched on some common scenarios that you encounter when writing and deploying iOS applications.

The next chapters tackle the other two major mobile platforms—Android and Windows Phone—and finish with a review of an HTML5-based framework for building mobile applications, called PhoneGap.

CHAPTER 9

# Developing for Android

*He who stops being better stops being good.*

*—Oliver Cromwell*

**In this chapter:**

- Getting Ready for Android Development
- Programming with the Android SDK
- Summary

Like it or not, one of the key factors to the success of the iOS is certainly that it's a closed environment. There are just a few hardware configurations and one operating system. Both aspects are strictly controlled by Apple, and carriers have no control over the operating system. As a user, you receive notification of system updates directly from Apple; as a developer, you have just one counterpart to talk to (and sometimes to quarrel with).

More specifically, this means that as a developer, you don't face the problem of adapting your user interface and presentation logic to different resolutions and different hardware—or, at least, there are only three options to deal with: iPhone, iPod Touch, and iPad. Most of the time, this all comes down to giving some applications (not even all applications) a different user interface on iPad so they can take advantage of the larger screen. When you do so, however, you're really starting another project—one with a high level of code reuse. Within the same application, you rarely need to accommodate different user interface and hardware configurations.

In iOS, you still may distinguish a few different versions of the operating system, and subsequently a few varying levels of the application programming interface (API). This means that an application written for iOS 4 may not run on a fairly old iPod Touch, but frankly, there are few differences beyond this point. The iOS panorama is definitely not fragmented.

In Android, you face a completely different situation.

Google started Android as an open-source project and, unlike Apple, Google doesn't mandate a unique and non-modifiable version of the operating system and hardware configuration. As a result, the range of different Android devices that your application may need to run on is quite wide. This makes Android development challenging, and potentially quite expensive.

# Getting Ready for Android Development

As mentioned in Chapter 8, "Developing for iOS," the primary goal of these platform-specific chapters is not to teach programming for them in depth. Instead, at the end of this chapter, you should have a clear idea of what it takes to write Android applications—but you probably will have a long way to go before you can reasonably call yourself an expert Android developer.

## Development Tools and Challenges

Let's review the first basic steps required to create your first Android application. You need to download a couple of software development kits (SDKs) and pick up an integrated development environment (IDE) where you write and compile the source code. By using an IDE, you save yourself the burden of dealing with command-line tools when compiling, debugging, and deploying applications to the emulator or to the physical device. If you prefer to use the command line, however, you will find all that you need in the Android SDK.

Finally, it is worth noting that you can do Android development on a variety of computers—you can use a Mac, or a Linux or Windows development machine.

### Becoming an Android Developer

The primary and native language of Android development is Java. As you'll see later in this chapter, you can use C# or even JavaScript to write Android applications, but in such cases, your calls are bridged to the components of the Android SDK, which is a Java framework.

**Important** Just to be clear, you can download everything that you need to write Android applications for free, without having to create an account anywhere. Deployment on a physical device is always free if you use SDK tools or do it manually. However, you may need to buy a license if you use a commercial IDE.

You don't need to register anywhere as an official Android developer to download and start using the SDKs and embedded tools. However, you do need an account, and you also must pay a small one-time fee (currently $25) if you want to distribute your applications through Google Play (which was formerly known as Android Market).

**Warning** While discussing Android application deployment, one thing you might want to be aware of is that sometimes (and mostly on Windows) your device may need an extra driver or some software that enables synchronization between the computer and the device. Don't be too surprised if that happens to be true in your case, too. In particular, I faced that with a couple of HTC devices. No big deal, though—in the end, it usually takes only a few moments for Windows to locate the driver automatically. In the worst case, finding it may require a quick search.

## Configuring the Environment

To do any Android development, you also need to install the Java Development Kit (JDK), which includes both the Java Runtime Engine (JRE) and the Android SDK. You can download the JRE from *http://www.oracle.com/technetwork/java/javase/downloads/index.html*, and the Android SDK from *http://developer.android.com/sdk*.

You can download the Android SDK for Windows, Mac OS, and Linux. If you use Windows, then an installer downloads the JDK for you if it's not already installed. Whether you use Eclipse or IntelliJ IDEA (from JetBrains) as the IDE, you also must add the path to the Android SDK to the preferences settings of the IDE; otherwise, no target SDK shows up when you try to create a new project, and the IDE will give you no chance to start writing code. On Windows, the Android SDK usually installs in the following folder:

```
C:\Program Files\Android\Android-Sdk
```

The SDK that you install at first is only a starter package. So the final step in setting up your environment is to download some other essential SDK components. You do that through one of the tools included in the starter package—the AVD Manager (see Figure 9-1).

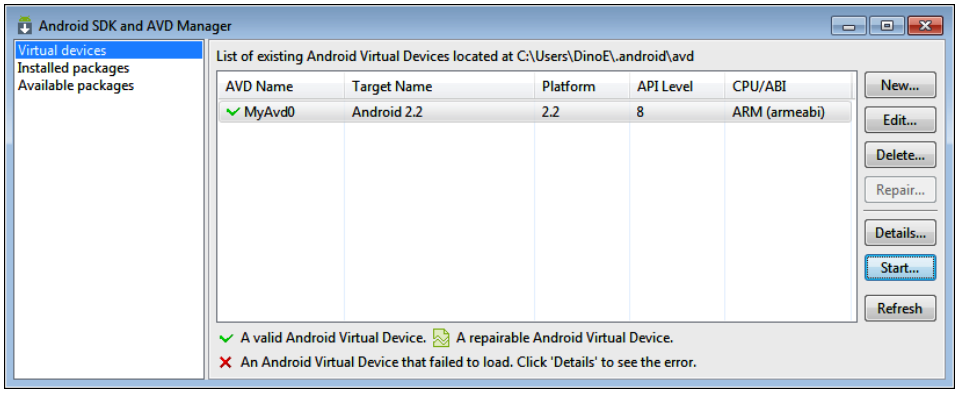



**FIGURE 9-1** The AVD Manager.

The AVD Manager has a graphical interface through which you can select new or updated components to install. Through this tool, you also can create and configure Android virtual devices if you intend to test your code on the emulator.

## Picking Up Your Favorite IDE

A fairly common choice among Android developers is to use Eclipse as the IDE. You can download Eclipse from *http://www.eclipse.org/downloads*. After downloading Eclipse, you also might want to ensure that you have the Android-specific plug-in. After the plug-in is fully configured, you find Android-specific project templates available when you start a new project. Figure 9-2 offers a view of the Android project template in the Eclipse environment.

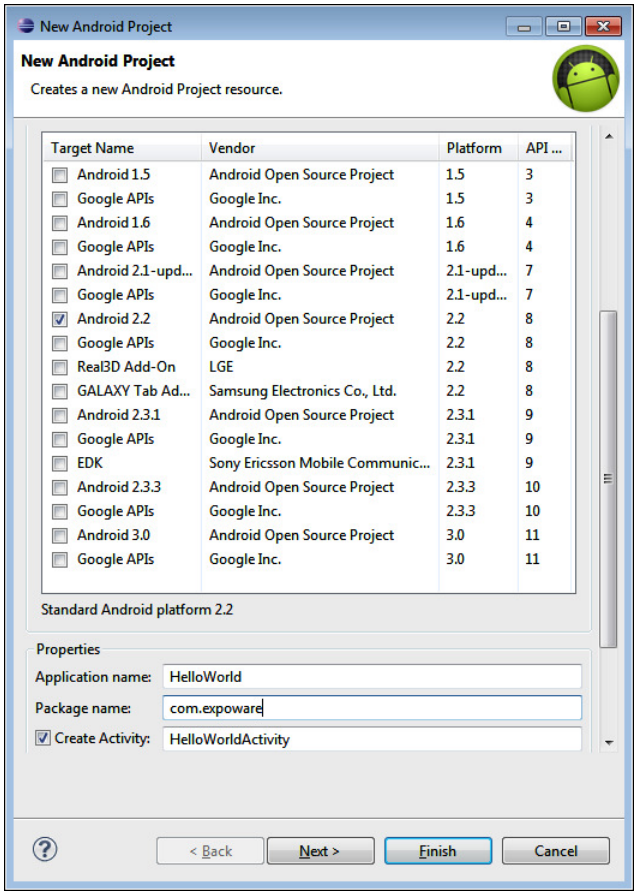



**FIGURE 9-2** The Eclipse environment for Android.

Another interesting IDE option for Android developers is IntelliJ IDEA. IntelliJ IDEA is a commercial product; however, it comes with a free Community Edition that provides great support for Android development. Figure 9-3 offers a view of the IntelliJ IDEA environment.

Both products are proficient at helping you write Java code. Both, however, provide minimal support for WYSIWYG scenarios (i.e., preview) and for user interface (UI) design. Personally, I use IntelliJ IDEA for my Android development.

## Choosing the Development Strategy

Defining a development strategy that targets the Android platform is fairly easy: you just use the Java language and the Android SDK. Other options exist, but choosing this native strategy doesn't pose any particular issues, such as having to learn a new language that may look old-fashioned or even weird.

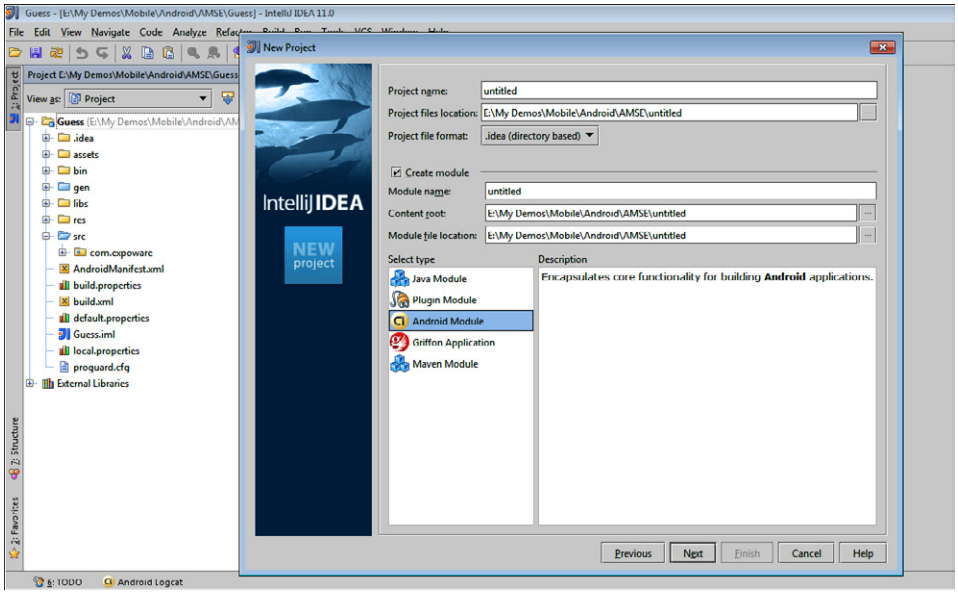



**FIGURE 9-3** The IntelliJ IDEA environment.

Java is an easy language to come to grips with. Even if you, like me, never have used it seriously before, it likely will take you only a few moments to become somewhat familiar with it. The code that you need to make an Android application work doesn't necessarily require implementation of patterns and use of inversion of control tools. If you know object-oriented principles and have written programs with C#, C++, or even Microsoft Visual Basic .NET, you will easily grasp the parts of Java that you need for Android.

## Using Java and the Android SDK

Java is a fully object-oriented language that is, in many respects, similar to C#. Java lacks some of the syntactic sugar that the C# compiler offers, such as extension methods, the *dynamic* type, and the *var* facility. However, a solid knowledge of object-oriented techniques provides plenty of background for good Android programming.

There are many reasons for choosing Java as the primary language for Android programming. First and foremost, Java is a known and widely used language with a rich ecosystem of programming tools. Compared to C/C++ (and to some extent, Objective-C), Java doesn't use pointers explicitly, so it dramatically reduces the chances of memory leaks and accidental overwrites.

Before smartphones came to hold the spotlight, Java was already in use in several mobile scenarios thanks to the Java Micro Edition (Java ME)—a Java platform designed for embedded systems and mobile devices. Although Java ME has been overtaken in the news by the newer smartphone operating systems, it still holds a large share of the market and powers many popular devices, such as Nokia's Series 40. So Java is already a familiar language for many early mobile developers.

Another excellent reason for picking up Java is that it runs in a virtual machine. This is beneficial in two ways. First, the virtual machine shields developers from the underlying hardware. Second, it runs applications in a sandbox, which prevents one poorly written (or malicious) application from bringing down the entire system.

Having said that, you should be aware that although Android uses the Java syntax, it's not a true Java platform. In fact, that specific aspect is at the origin of a legal battle between Google and Oracle, which is the current owner of the Java language. You write applications using the Java language, but your source code does not get compiled to the bytecode of the Java Virtual Machine. Instead, your Android Java source code gets compiled to a *custom* virtual machine that uses a completely different bytecode. This Google version is called the Dalvik virtual machine.

**Note** At this point, it's worth taking a brief look at the previous chapter to investigate why Apple decided to go with Objective-C for native iOS development. Objective-C was the language of choice for NeXT, the platform that Steve Jobs created after leaving Apple in the 1990s. Later, NeXT and Apple merged. Subsequently, the Mac OS operating system was created from NeXTSTEP—the NeXT operating system. Objective-C was the language of choice for NeXTSTEP, so it became the language of choice of Mac OS initially, and of iOS more recently.

## Using MonoDroid and C#

Java is not the only option available for writing Android applications. You also can use the C# language and a subset of the Microsoft .NET Framework to target Android devices. To do that, you need MonoDroid—a MonoTouch twin product developed by Xamarin.

MonoTouch is a commercial product that requires you to buy a license. A single developer license will cost you about $399 (you can get more information at *http://www.xamarin.com*).

You can use MonoDroid on a Windows machine and write code using Xamarin's MonoDevelop IDE, or even by using an extension to Microsoft Visual Studio.

A MonoDroid application executes within an instance of the Mono virtual machine. The Mono virtual machine lives side by side with Google's Dalvik virtual machine on all devices. To access functionality, developers use a set of classes that look like classes in the .NET Framework, except that they bind to the Android API under the hood. The build process for a MonoDroid application passes through four steps:

1. Processing resources from Android resource files to .NET-compatible resource files
2. Creating the .NET code
3. Processing of the .NET code to create Java wrappers
4. Final packaging of the Android executable

As with MonoTouch on iOS, MonoDroid can reuse a lot of existing .NET code, but mostly in the back-end area. The presentation layer of MonoDroid applications closely reflects the philosophy and behavior of Android *activities*. You must know that if you're going to write MonoDroid applications effectively.

## Using the PhoneGap Framework

In the panorama of mobile programming tools, Adobe's PhoneGap is a fairly unique framework that can turn a bunch of client-side webpages into a native application for a variety of mobile operating systems, including the Android platform.

As a developer, you write a classic client-side web application using HTML5, Cascading Style Sheets (CSS), and JavaScript. You can write and test the application on your favorite development platform using your favorite tools. For example, you can use Visual Studio 2010, or WebStorm (from JetBrains), or any other text editor.

**Note** On a personal note, although I see a lot of value in using MonoTouch to target iOS, I don't honestly see the same value (or at least not necessarily) for Android. For me, the primary reason to consider MonoTouch is to avoid Objective-C because I would prefer a more classic object-oriented language. As mentioned in Chapter 8, though, you still need to learn Cocoa Touch to program for MonoTouch. And for Android, you still need to understand how the Android user interface works. So it boils down to a Java vs. C# trade-off, including tools and ecosystem. Even with my strong .NET background, I find the Java offering quite compelling and well worth the price. As an important side note, by the way, Java-based Android applications are far smaller than MonoDroid applications. Android has a large base of devices, and not all of them come with a large SD card. Storage is a problem, so having small applications is much more important on Android than on iOS.

It is important to note that the core input to the PhoneGap framework is a collection of static HTML pages that may possibly use JavaScript and Ajax calls to download data from remote endpoints. You build the user interface using HTML5 and make it compelling through CSS. HTML5 is essential because it incorporates a local storage feature, thus giving you the ability to save data locally. The client application also may link to the PhoneGap JavaScript library to gain access to additional features and device specific hardware, such as a camera or accelerometer.

Using HTML5 is highly recommended because it also gives you free goodies, such as native date pickers (if the browser supports that) and access to local storage. The best HTML5 browsers are currently mobile browsers on the mobile platforms that PhoneGap specifically addresses.

You test the application on the development machine (for example, a Windows machine with Visual Studio) using an HTML5-compliant browser such as Chrome, Safari, or Windows Internet Explorer 10. If you aim to target Android primarily, then you should test against desktop editions of Chrome.

Once the client web application is complete, you use the PhoneGap framework for Android. You create an ad hoc project and build and deploy it. The original set of webpages is packaged as a

browser-based native application with access to the native device features that you claimed via the PhoneGap JavaScript library.

Building a PhoneGap application doesn't require many Android skills, though you still need an Android project, an Android IDE, and the various SDKs. I'll cover PhoneGap development in more detail for iOS and Android in Chapter 11, "Mobile Applications with PhoneGap."

## Other Options

As mentioned in Chapter 8 for iOS, you also have cross-platform options for development for Android. Specifically, you might want to look into Appcelerator's Titanium Mobile (see *http://www.appcelerator.com/titanium*) and Adobe's Flash Builder 4.6.

Titanium Mobile produces a native application on the target platform that is not based on the services of the default device browser. You develop an application using JavaScript and the Titanium framework. As it turns out, you don't actually use HTML and CSS to define the user interface; instead, you build views and add logic solely through the Titanium framework. The classes that you use in the framework then are expanded, at compile time, into a mobile native counterpart on the target platform that you choose.

To build Titanium Mobile applications, you need the ad hoc tools provided with the default package. No license fee is associated with Appcelerator's Titanium unless you want support, in which case you can buy an enterprise license.

Flash Builder can produce applications for both iOS and Android. It uses the Adobe's AIR engine. You write applications using ActionScript for the logic and mXML for expressing the user interface. The programming environment is both stunning and effective, with a lot of facilities, and would be an excellent and natural choice for developers who already have good Flash skills (see Figure 9-4).

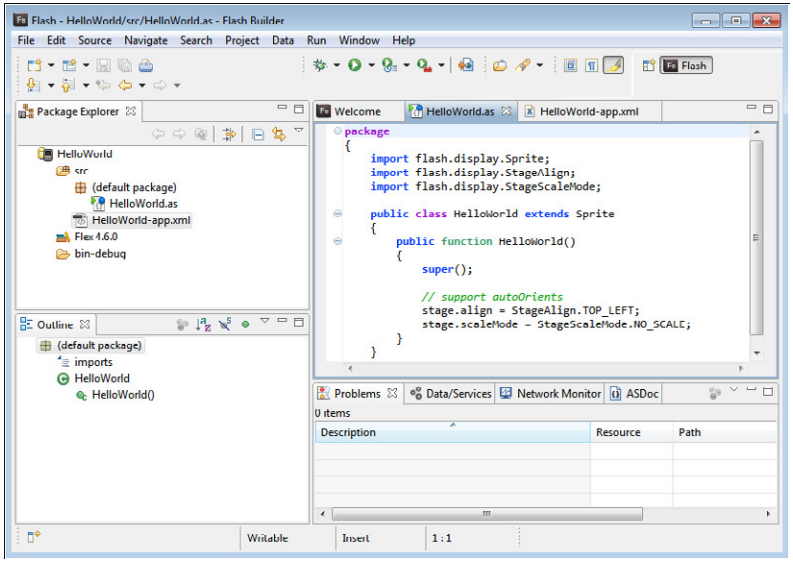



**FIGURE 9-4** The Flash Builder IDE in action.

Even more impressive is how you target different platforms in Flash Builder. Figure 9-5 shows the Run Configurations dialog box, where all you need to do is choose the target platform—either iOS or Android.

**Note** You also can use Flash Builder to compile applications for the BlackBerry platform. You choose from the list of supported platforms when you create the project.

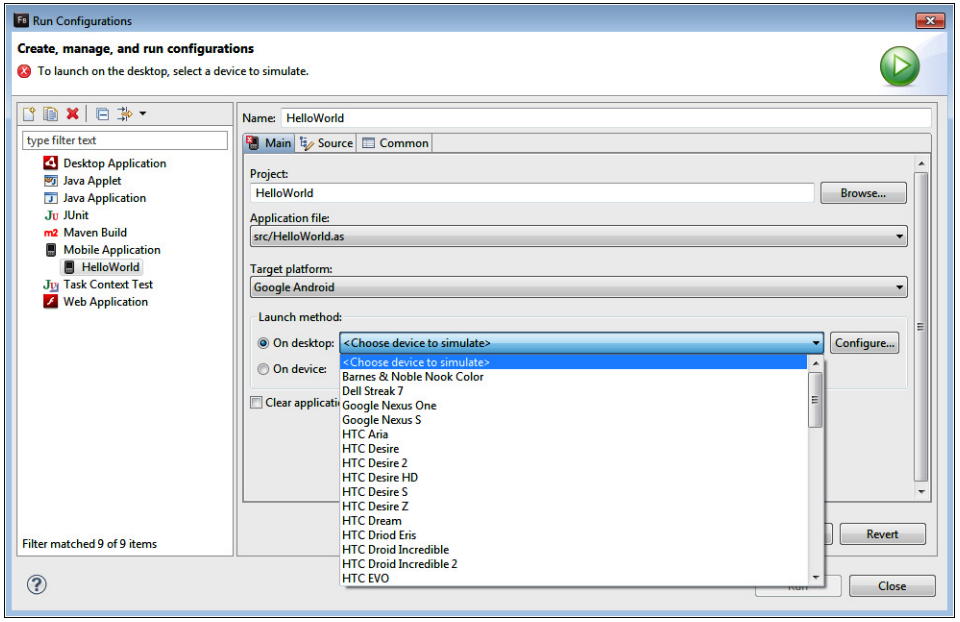



**FIGURE 9-5** Testing applications with Flash Builder.

## The Android Jungle

Carriers, as well as companies and individuals, can tweak and recompile the Android operating system and adapt it to any particular hardware and needs they have. The original goal of the Android project was to make available a totally open mobile operating system and API. That decision was probably made with the best possible intentions; however, I'm not sure that the net effect of that decision has been completely positive.

### Firmware, Carriers, Manufacturers, and You

On a mobile device, the operating system is loaded into a read-only memory area (known as ROM for short). Because this system code stays firmly in memory and is not subject to dynamic modification, it is also often referred to as *firmware*.

**Note** Terms like *operating system* and *firmware* are nearly synonymous in the mobile context. On personal computers, however, firmware is often integrated with hardware components and controls the behavior of that piece of hardware. Also, firmware on personal computers, more often than not, is loaded dynamically via device drivers. On much simpler mobile devices, the firmware is the operating system, which includes both hardware controllers and basic applications.

The firmware that you find on Apple devices is truly the stock firmware released by the operating system vendor (Apple itself, in this case). On Windows Phone devices, you may find some customization applied by carriers and manufacturers, which consists mostly of adding additional applications, perhaps written by accessing an internal API. For example, only manufacturers can produce Windows Phone applications that have custom tile sizes.

On Apple and Microsoft Windows Phone devices, however, you have no chance to customize the firmware. But for Android, things are different.

Carriers and manufacturers can modify the Android firmware as originally released by Google. In doing so, they often enhance the user interface. HTC does this when it adds HTC Sense on its devices. They also may add proprietary applications and even lock the device so it will operate only on a given carrier or in a specific region. In addition, individual developers can replace the factory firmware with their handcrafted firmware.

**Note** Installing custom firmware may not be a good idea unless you know what you're doing. Unfortunately, many users and developers do this to avoid having to rely on carriers and manufacturers to push updates (which often happens slowly, if at all) for any given device model.

## API Levels

When Google releases a new update to the Android operating system, carriers and manufacturers are supposed to propagate it down to users. Any Android release requires updates to the custom firmware; this means carriers and manufacturers have to create the updates and go through a testing cycle. However, whether they do so is not simply a matter of adding a bit of delay to the update process. Manufacturers in particular may have different goals than the operating system producer (Google). So it is not uncommon to find that only a few older models get updated to the latest version of the operating system—and even those, in general, propagate quite slowly. In the long run, the gap between the level of API available in the latest version and the already distributed API level available in older, not-updated versions can get quite large. This not only generates confusion and disappointment for users, but is a total disaster for developers.

Each release of Android is characterized by an API level that grows over the previous version. Fortunately, higher releases are backward-compatible. For instance, if you target API Level 8, you can be sure that your code will run on devices with a version of the operating system that offers API Level 10.

So what should you do? Should you always choose the lowest common denominator approach? If so, what is the current bottom API level?

As an example, consider that the latest version of the Android system is currently 4.0 (API Level 15). However, a very large share of devices run on Android 2.*x,* and even Android 1.6 is still fairly popular. Android 2.2 (code-named Froyo) runs API Level 8 (see *http://developer.android.com/guide/appendix/api-levels.html)*.

**Note** Today, a reasonable bottom level for Android, if you really want to make sure you don't leave out any devices, is version 1.6. Version 2.1 or even version 2.2 is acceptable as well, if you're willing to abandon the oldest devices.

## Different Screen Sizes

In addition to API levels, the toughest issue you face when writing Android applications is dealing with different screen sizes. Android 1.6 was the first version of the operating system to ship with native support for multiple screens. And Android 3.2 is the first version with an additional and specific API for controlling the layout of views specifically for tablets.

Support for multiple screens means that the SDK gives you programming tools to shield your code from different resolutions and different pixel densities. Some work on your own is required, though.

In particular, you should avoid using hard-coded pixel values and use relative sizing—typically, you use ad hoc values for pseudocommands like *fill-parent* and *wrap-content*. In addition, you might want to detect orientation changes and swap layouts as appropriate. For simplicity, Android classifies common screen sizes into four groups: small, normal, large, and extra large. In your project, you should ideally provide different layouts for each of these screen sizes. You are not forced to support them all; however, if you don't, you should declare in the application's manifest file which screen sizes you support. Note that if you declare a list of supported screen sizes, then your application won't execute on devices that don't support any of those supported sizes.

In addition to screen sizes, you need to consider pixel density. Pixel density is defined as the number of pixels in a given area of a screen of a given size. Android defines four densities: low, medium, high, and extra high. You are expected to provide drawable content (images) for each density level to ensure that bitmaps don't stretch or shrink when rendered.

Finally, Android provides a way for developers to indicate measurements using device-independent units that are processed and transformed into real pixels when the code runs on a given device. You use the *dp* unit for distances and the *sp* unit for font sizes.

**Note** Sometimes you may encounter code that uses the *dip* unit. That unit is exactly the same as *dp*, and the compiler treats them in the same way. The *dp* unit is usually preferred because it is more consistent with the *sp* used for fonts.

# Programming with the Android SDK

Let's start with a quick look at the skeleton of an Android application. You need to acquire the basic fundamentals of Android programming (views, user interface, and event handling), and get used to the SDK.

## Anatomy of an Application

This chapter will use IntelliJ IDEA Community Edition as the editor. The project structure, however, is nearly identical if you use Eclipse.

**Note** IntelliJ comes with a classic Windows installer, so configuring the environment is easy. You only need to indicate where you installed the Java and Android SDKs.

### Dissecting the Project

A typical Java project consists of quite a few folders, each with a specific content and meaning. The most important folders are Src, Lib, and Res.

As you can see in Figure 9-6, the Src folder contains the source code of the project, including Java classes such as activity classes (specific to Android), helper classes, enumerated types, and custom types. You can group some of these classes in packages by simply creating child directories and adding classes. Note that a Java package is a close relative to a C# namespace.

The Lib folder is the repository of externally linked libraries. For example, if you're writing an application that uses Twitter, then you probably want to use an external library (e.g., Twitter4j) that saves you from the burden of dealing directly with the OAuth protocol. An external Java library is usually a *.jar* file.

The Res folder is the central repository of all your graphical resources. The Res folder has a number of subfolders, each containing a different type of resources. For example, the Layout subfolder contains the XML files that describe the application views, whereas the Drawable folders store bitmaps of different sizes. The system will intelligently select which one to use based on the actual pixel density of the device.

**Note** Android employs a sort of convention-over-configuration (CoC) approach in the naming of folders. For example, Layout-land indicates the folder where landscape views are stored. Likewise, Drawable-hdpi indicates bitmaps for high-density devices, and Drawable-land is where you store bitmaps for landscape views.

## The Manifest File

In Android, every application needs to have a manifest file located in the project root folder. An Android manifest is an XML file named AndroidManifest.xml (note that the name of the file can't be changed).

The manifest contains selected information about the application. The system looks into the manifest file before it installs or runs the application. As the first item in the manifest, you set the package name and version of the application. The package name is comparable to the assembly name of .NET applications. It usually takes the form of *com.company.application,* where *company* is a word that identifies the author and *application* is a word that identifies the application. However, there's no strict requirement that you name your Android applications (and Java packages in general) that way.

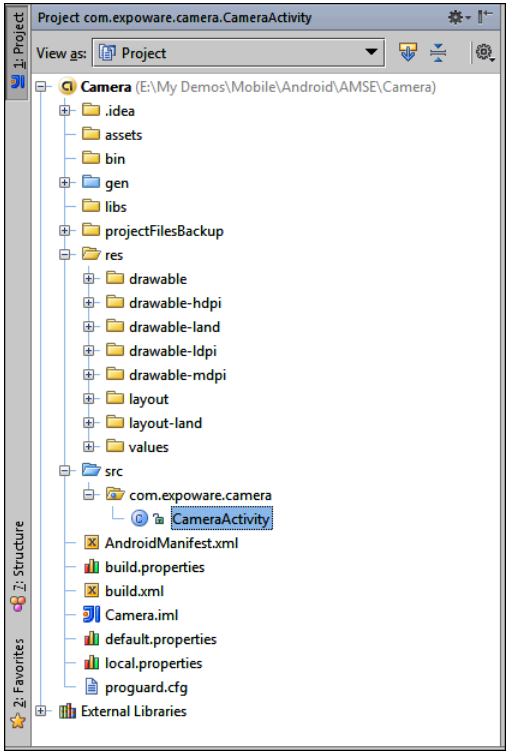



**FIGURE 9-6** A sample Android project in Intellij IDEA.

**Note** In Java, a widely used naming convention (yet, still simply a convention) is that applications are named after one's domain name. For example, if your company's domain is *contoso.com,* then your Java/Android packages should be named *com.contoso.XXX*—the reverse order of the elements of a domain name. Likewise, if your company has established a domain such as *contoso.org,* then the suggested name would be *org.contoso.XXX*. The convention has the sole purpose of ensuring that unique names are always picked up.

Here's how a minimally working manifest file looks:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.expoware.camera"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="My app"
                 android:icon="@drawable/app_logo">
        <activity android:name="MyAppActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

In the manifest, you set the public name and icon of the application and provide information about the starter component. An Android application is made of one or more activity classes. An activity is a close relative to a UI view controller in iOS or a UI page in Windows Phone. The name *activity* just refers to a component that allows the user to do something on that screen. All the activities that an application plans to use must be declared as child elements of the *application* node. Each activity element indicates the name of the class behind the activity and optional information, such as the required screen orientation and launch mode.

The *intent-filter* node specifies special intents (if any) of the activity. Intents are essentially actions the activity may perform on some data. Some predefined actions are VIEW to view a screen and CALL to initiate a phone call. You indicate supported actions through the *action* element, whereas the *category* element just adds more information about how the action will be executed. For more information on intents and advanced aspects of Android programming, the best reference is the Android SDK documentation at *http://developer.android.com*.

One piece of information that you often need to add to a manifest file is the list of permissions the application must have to work properly. Note that in Android, you need to specify a permission for nearly every significant form of coding. Want to save a file? You need to add permission. Need to access the Internet? You also need permission.

Finally, in the manifest file, you may need to indicate supported screens, the minimum level of the Android API that you support, and the non-default Android packages that you need to link to (such as maps).

**Important** Android is certainly not the only platform that requires permissions. However, in Android, the manifest that project wizards create contains no permissions by default, so as a developer, you are forced to explicitly add all permissions that you require. In contrast, in Windows Phone development, the manifest created by the default project template already declares a variety of permissions, so you almost never need to deal with them. Note also that *permissions* in this context refers to notifying users of potentially intrusive actions that your application will take. Some of these actions (such as placing a phone call or accessing the Internet) may cost users money or take up space on the phone, so the policy is that applications must let users know what they need and get their approval beforehand. Permissions, in fact, are listed before a user installs the application from the market, so users can decide whether to go ahead with the install. No such facility exists if the user manually installs the application outside the market. In any case, any attempt to execute code without having declared related permissions simply fails.

## Application Startup

In Android, you don't need any special starter code as you do in iOS. You just need to declare in the manifest the main entry point into the application. An entry point is an activity class with a particular intent filter declaration like the following:

```
<activity android:name="MyAppActivity"
          android:label="My application">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

In Android, an intent is a message being sent to target components such as an activity or a background service. An intent is a data structure that carries information about the action to perform and is represented by an instance of the *Intent* class. An intent filter, instead, is the list of intents that a target component (e.g., an activity) can handle.

An Android entry point must be able to support the *MAIN* action and the *LAUNCHER* category. *MAIN* is a standard Android action that is sent to start a component. *LAUNCHER* is an attribute that causes the entry point to be listed in the application launcher pad with its own icon and label.

## Android Application Components

An Android application results from the combination of up to four different types of components. The most commonly used are activities and services; other components are broadcast receivers and content providers.

An *activity* is expected to provide a screen for the user to interact with. Typically, an activity hosts a view, which can be an application-specific view or perhaps a system-provided view. An activity is associated with a window and commonly (but not necessarily) covers the entire screen. An application needs to have at least one main activity to start and can have a number of other activities—roughly one per screen. Activities are organized in a stack and when a new activity starts it goes to the top of the stack. The user can then navigate through the stack using the Back button. Activities receive notification from the system about relevant facts during their lifetime. For example, activities are notified when they are pushed to the background and restored.

A *service* is a UI-less component and runs in the background typically engaged in the performance of long-running tasks such as uploading or downloading data and application-specific calculations. In general, a service is used to perform any tasks that are asynchronous with the respect to the main user interface. A service is typically started by an activity.

A broadcast receiver listens to special messages being broadcast by the system or individual applications. An interesting example of a broadcast message is a network-state that is sent out when you get or lose connectivity. You get similar messages when your connectivity changes from WiFi to 3G, when the battery is dangerously low, or when a Short Message Service (SMS) item is received. Broadcast is not limited to the system; applications can broadcast, too. A receiver is often UI-less but can display notifications on the status bar. Also, a receiver is usually quite a simple component and relays work to other services or activities.

Finally, a *content provider* is a component that manages shared data and optionally exposes query and update capabilities for other components to invoke. An example is the provider for data about the contacts you have stored in the phone.

## The "Hello, World" Program

At this point, let's have a look at what a minimal "Hello, World" style program may look like in Android. It is centered on an activity like the following:

```
public class MyAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.myapp);
    }
}
```

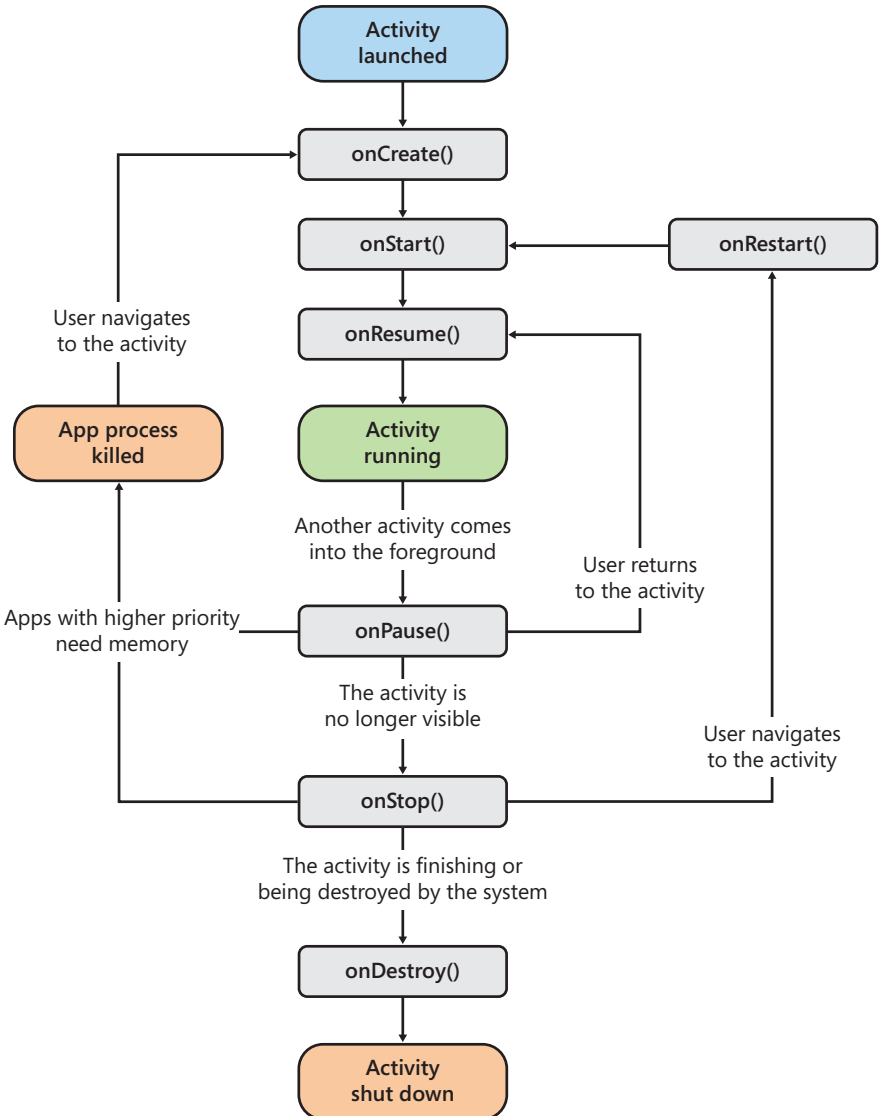
The base class *Activity* offers a virtual method named *onCreate* that—at the very minimum—you override to give the activity a view to show. As you can see, the *onCreate* method may receive data from the system in the form of a *Bundle* object—a dictionary-like object.

The Bundle parameter of *onCreate* is always *null* unless the activity itself stored some data in it overriding the *onSaveInstanceState* method. In particular, the *onSaveInstanceState* method is invoked just before the activity is destroyed. An activity is destroyed and immediately re-created when, for example, the screen orientation changes. In this way, you can manage to restore the application to its

previous state. The *Bundle* parameter is also non-null when the activity is pushed to the background and then gets terminated by system to reclaim its resources. On the next start (i.e., the user navigates back) a new instance is created with the saved state information.

The other thing to note about the previous code is the *setContentView* method. As the name suggests, the method sets a view. The view is identified by ID. The *setContentView* method receives the identifier of the view. Let's learn more about how to identify application resources.

Here's a diagram that shows the life cycle of an activity:



## Application Resources

In Android, identifiers are expressed as constants exposed by ad hoc classes. Android offers several classes, each one grouping a specific type of resource. For example, *R.layout* is a class that contains identifiers for layout resources, and *R.id* is a class that contains identifiers for UI widgets, such as buttons or text boxes. All *R.xxx* classes in the Android SDK contain a bunch of predefined constants.

In general, any expression of the type *R.layout.XXX* is resolved automatically if you have a file called *XXX.xml* in the Res/Layout folder. Similar rules exist for other *R* classes. For example, *R.id.xxx* is resolved to an identifier *xxx* if you have a UI widget with that ID in the current activity view. A layout file describes the content of a view. We'll return to layout files in a moment. For now, though, let's focus on the *android:id* attribute:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent">
     <TextView android:id="@+id/description" />
     <Button android:id="@+id/buttonOK"
             android:text="Click me" />
</LinearLayout>
```

As mentioned, the Android SDK comes with an *R.id* class that exposes a few predefined constants; you reference predefined constants using the expression *@id/xxx*. You use the expression *@+id/xxx* to force the creation of an ID named *xxx* in the *R.id* class that gets compiled with your source code.

In Android, you get references to UI widgets by calling the method *findViewById*. Getting these references is necessary if you intend to interact with that widget:

```
Button ok = (Button) findViewById(R.id.buttonOK);
```

If you want to handle events on a widget you need a listener. A listener is an instance of a framework class that implements an event-specific interface. For the click event, you need something like this:

```
okButtonClickListener =  new View.OnClickListener() {
      public void onClick(View v) {
         okButtonClick((Button) v);
      }
};
```

Any activity class then will have a private member for each event handler and a private member for any UI widget you need to interact with. The following listing illustrates a sample application that presents a button. If clicked, the button updates the content of a label:

```
public class MyAppActivity extends Activity
{
    Button ok;
    TextView description;

    private View.OnClickListener okListener = new View.OnClickListener() {
        public void onClick(View v) {
            UpdateLabel();
```

```
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        initializeWidgets();
        addHandlers();
    }

    private void initializeWidgets()
    {
        ok = (Button) findViewById(R.id.buttonOK);
        description = (TextView) findViewById(R.id.description);
    }

    private void addHandlers()
    {
        ok.setOnClickListener(okClickListener);
    }

    private void UpdateLabel()
    {
        description.setText("Clicked");
    }
}
```

## Defining the User Interface

The user interface of an Android application consists of XML files, through which you define the layout of a view. A view is made of widgets such as buttons, spinners, text boxes, and check boxes. In addition, you have other resources, such as menus, strings, and graphical shapes, to deal with.

### Defining a Layout

In Android, the term *layout* means the structure of any content displayed in a view. You can define the content being displayed in a view either programmatically or declaratively. In the former case, you build the visual tree by creating instances of widgets and adding them to the root element of the view. In the latter case, you provide an XML-based description of the expected user interface and have the activity to parse it and figure out the widgets to create.

The following code shows how to complete the user interface of an activity, adding a programmatically created widget to an XML-defined layout:

```
public void onCreate(Bundle savedInstanceState)
{
   super.onCreate(savedInstanceState);

   // Create the original view from a layout file (res/layout/main.xml)
   setContentView(R.layout.main);
```

```
    // Create pieces to be added dynamically
    TextView text = new TextView(this);
    text.setText("Added programmatically");

    // Add content dynamically to the view of the current activity
    addContentView(text, new LinearLayout.LayoutParams(
             LinearLayout.LayoutParams.FILL_PARENT,
             LinearLayout.LayoutParams.WRAP_CONTENT));
    ...
}
```

You still can call *addContentView* even if you didn't call *setContentView* beforehand. All you get in this case is a relatively empty template in which all that appears is what you explicitly added through code.

Note, though, that in case you're creating a layout from scratch you should start with a layout object such as *LinearLayout*. You're not allowed to have a *TextView* or a *Button* as the root of the view. To add widgets programmatically at a specific point of an existing hierarchy, you must locate the parent object and then add the view to its subtree, as shown here:

```
// Get the root container
LinearLayout linear = (LinearLayout) findViewById(R.id.yourLayoutId);
linear.setOrientation(LinearLayout.VERTICAL);
text = new TextView(this);
text.setText("Some text here");

// Add the object to the layout
linear.addView(text);
```

You need a layout file for any activities you create. An exception to this rule is when you derive your activity from one that comes with a predefined layout, such as *ListActivity* or *PreferenceActivity*.

## Layout Types

In Android, you have four main types of layouts: *FrameLayout*, *LinearLayout*, *TableLayout,* and *RelativeLayout*.

An extremely simple container, *FrameLayout* is basically a placeholder for a single object. Ideally, you use it to reserve a blank space in your main layout where some dynamic content will be displayed. The content of the *FrameLayout* is a single object, but with any level of nesting you like. All child objects of a *FrameLayout* are always rendered from the upper-left corner of the screen.

*LinearLayout* is one of the most frequently used layouts. It works by stacking its child elements either horizontally or vertically. An interesting capability of the *LinearLayout* object is support for weight. *Weight* is an integer meant to indicate the importance of a child element and is set to 0 by default. A higher weight value provides a hint for the activity, which will try to stretch weighed elements to a larger size.

*TableLayout* looks like an HTML table, but it doesn't feature most of the advanced capabilities of HTML tables. In particular, it doesn't support graphics such as borders or padding and only supports rows. Columns are calculated based on the maximum number of cells required by a single row. Cells in a row are not allowed to span over multiple columns.

*RelativeLayout* is a container in which contained elements indicate their preferred position relative to the parent. Likewise, a child object can specify its position relative to another element identified by ID. As an example, you can align two objects with the right edge or place one under the other or to the right or left of another.

When defining a layout, you must provide at least width and height information. The code here shows an example:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  ...
</LinearLayout>
```

The orientation setting is just as important to set but at least it has a default value—horizontal. If you miss *layout_width* and *layout_height,* you get a run-time error. You use special values like *fill_parent* to instruct the widget to extend horizontally as much as possible and *wrap_content* to have it take up just the real estate it needs. Both values help a lot in creating Android user interfaces that survive multiple screens.

**Note** The XML schema used by Android layouts has several points in common with the Extensible Application Markup Language (XAML) schema used by Windows Presentation Foundation (WPF) applications and Windows Phone applications. In particular, the *LinearLayout* component is similar to XAML's *StackPanel*, whereas *TableLayout* is a close relative to *Grid*.

## Change of Orientation

When the device is rotated to a portrait or landscape view, you might want to switch the layout accordingly. Android looks for layout and resources in an orientation-specific folder that is characterized by a *–land* or *–port* suffix. If the application is in portrait mode and there's no folder like Layout-port or Drawable-port, then Android automatically falls back to the Layout (and Drawable) folders.

Keep in mind that change of orientation means that the application will be destroyed and then immediately restarted. Note that in iOS, a change of orientation doesn't result in the activity to be destroyed and re-created. It is your responsibility to guarantee that no relevant state is lost in the process. You can use the *onSaveInstanceState* method on the activity class to save your state before the orientation change:

```
@Override
public void onSaveInstanceState(Bundle outState)
{
    // Save current total score
    outState.putInt("Goals1", homeGoals);
    outState.putInt("Goals2", visitorsGoals);
```

```
    outState.putString("Home", homeTeam);
    outState.putString("Visitors", visitorsTeam);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ...
    // Restore state (if any)
    if (savedInstanceState != null)
    {
        ...
    }
}
```

The state is saved to a *Bundle* object and can be restored when *onCreate* is invoked again upon restart.

## Style and Themes

Multiple graphic attributes can be grouped together into a style. The overall developer experience is analogous to styles as you may know them from XAML. Here's how you style an element:

```
<Button style="@style/CoolButton" android:text="I'm a cool button" />
```

In this case, *CoolButton* is a style defined in a XML file located in the Res/Values folder of the project. The name of the XML file doesn't matter. (It is usually called Styles.xml, though.)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <style name="CoolButton">
     <item name="android:layout_width">fill_parent</item>
     <item name="android:layout_height">wrap_content</item>
     <item name="android:background">#00ff00</item>
   </style>
</resources>
```

A style can be inherited from an existing style. If the parent style is a system style, you use the parent attribute on the style element, as shown here:

```
<style name="CoolButton" parent="@android:style/BaseButton">
   ...
</style>
```

If you want to inherit from one of your styles, then you use a slightly different syntax, which is similar to the CSS syntax. The name of the style results from the concatenation of the parent style with the derived style. For example, you can have *CoolButton.Shadowed,* where both are user-defined styles defined within the same application.

An Android style can be used to style individual elements as well as an entire application. If you want to style an entire application, then you use the theme attribute to reference the style by name in the <*application*> tag within the Android manifest file.

## Graphical Shapes

Solid colors are the simplest option to paint the background of layout widgets. You are not limited to solid colors, however. You use a *shape* to define a combination of graphical entities to render as the background of layouts. The following code shows a shape that simply contains a gradient. Let's assume that the code is taken from a file named Apptitlebar.xml located in the appropriate Drawable-xxx folder under /Res/Layout:

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">
    <gradient android:startColor="#001020"
              android:endColor="#0000ff"
              android:angle="180" />
</shape>
```

In this particular example, the shape is a rectangle and contains a bluish gradient. To give an area of the screen this background, you proceed as follows:

```
<LinearLayout android:id="@+id/titlebarRow"
              android:orientation="horizontal"
              android:background="@drawable/apptitlebar"
              android:layout_width="fill_parent">
   ...
</LinearLayout>
```

The *background* attribute references a *drawable* resource named *apptitlebar*. The name of the resource must match the name of a file in one of the Drawable-xxx folders.

Here's an example of a slightly more sophisticated shape with a rounded rectangle, padded, with a border and a gradient in the background:

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">
    <stroke android:color="#ffffff"
            android:width="1px" />
    <padding android:left="5dp"
             android:top="5dp"
             android:right="5dp"
             android:bottom="5dp" />
    <corners android:radius="5dp" />
</shape>
```

When it comes to WYSIWYG capabilities, IDE tools for Android—whether Eclipse or IntelliJ—pale in comparison to Visual Studio tooling. Recently, Eclipse integrated a simple graphical tool that lets you use dragging to compose widgets. At present, IntelliJ provides a viewer, but not a designer (see Figure 9-7).

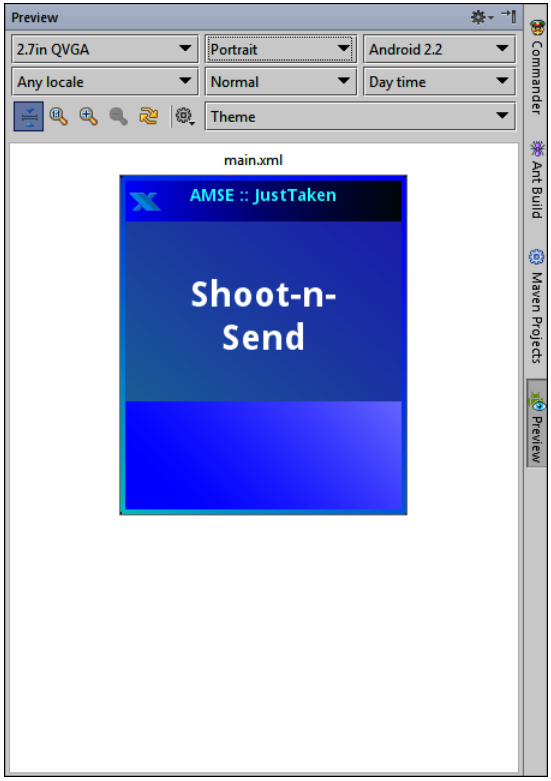



**FIGURE 9-7** WYSIWYG support in IntelliJ IDEA.

## Adding an Options Menu

Android devices feature a Menu button that, when touched by users, automatically opens an Options menu that applies to the current context.

The Options menu refers to a feature has similarities with both the main menu and the context menu of desktop applications. In fact, the Options menu is usually hidden and shows up only when the user touches the Menu button. The Options menu always displays at the bottom of the screen, and it is dismissed right after a selection is made.

As a user, you touch the Menu button when you're looking for a list of possible actions to take at a given stage of the application. Just for this reason, the list of options presented by the menu may be significantly different at different times. You can handle these differences in either of two ways. For example, you can create a single menu and add or remove items and disable and enable items on a per-display basis; or you can reset and repopulate the menu before each display. You typically choose the latter option if the two menus to display are significantly different.

The menu is an application resource and should be defined as an XML file located in the Res/Menu folder. The name of the XML file is unimportant, but it determines the name of the menu that you'll be referencing later.

Here's a concrete example of the content that a menu resource may have:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menuSave"
          android:icon="@drawable/save"
          android:title="Save" />
    <item android:id="@+id/menuShare"
          android:icon="@drawable/share"
          android:title="Share" />
</menu>
```

All items are grouped under the <*menu*> node, and each item is characterized by a <*item*> node. Each item contains at least an ID, a title, and an icon. You can reference titles and icons from the resources of the application using resource IDs. The previous example defines two menu items—to save and share the current state of the application.

To create submenus, you simply create nested <*menu*> subtrees. Creating a menu from a resource is the most common approach, but you can also do that programmatically by first creating a *Menu* object and then filling it up with *MenuItem* objects.

To deal with menus in an Android activity, you must know about a couple of *overridable* methods of the Activity class. They are *onCreateOptionsMenu* and *onPrepareOptionsMenu,* and both take a *Menu* object and return a Boolean value:

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
}

@Override
public boolean onPrepareOptionsMenu(Menu menu)
{
}
```

The *onCreateOptionsMenu* is invoked only once, the first time the *Menu* button is touched in the application's instance. (Note that this behavior is slightly different in Android 3.0 and later versions.) If your menu is relatively static and never changes, or it limits its changes to enabling/disabling items, then you may want to initialize it only once and move adjustments to the implementation of *onPrepareOptionsMenu*.

If your application's menu might need to be significantly restructures (depending on the context), then you're better off creating it from scratch each time the *Menu* button is touched. In this case, you save yourself overriding *onCreateOptionsMenu* and concentrate on *onPrepareOptionsMenu*. In the example listed here, you see an implementation of *onPrepareOptionsMenu* that chooses between two different menus. The method *onPrepareOptionsMenu* is invoked every time the menu is about to display. Both methods receive the menu object as an argument—the menu object is created by system as soon as the user taps the Menu button:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu)
```

```
{
    // Clear the current menu
    menu.clear();

    // Determine the menu to load based on the context
    int menuResourceId = IsMatchPlaying() ?R.menu.duringmatch : R.menu.general;
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(menuResourceId, menu);
    return true;
}
```

The method first clears the current menu and then repopulates it based on the known state of the application. Note that you have menu methods to override on a per-activity basis (see Figure 9-8).


**FIGURE 9-8** The Options menu in action.

The final aspect of menus to consider is how you handle the user's clicking. As you'll see in a moment, this particular aspect of Android programming is vintage and calls back to memory the Windows SDK programming style of the 1990s:

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
   // Handle item selection
   switch (item.getItemId())
```

```
    {
        case R.id.menuNewGame:
            StartNewGame();
            return true;
        case R.id.menuSettings:
            showSettings();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

All you need to do is override the *onOptionsItemSelected* method, grab the unique ID of the selected item, and arrange a *switch* statement to decide on the next step.

**Note** Speaking of menus, another really interesting concept is the menu group. A menu group is a logical way to group menu items so that you can enable and disable them in a single shot. To create a group menu, you use the <*group*> element to group multiple menu items. Each group is given a unique ID. Group menus have no impact on the user interface.

## Localization

Localizing the text of an Android application requires that you keep all the strings the application uses in a Strings.xml file located in the Res/Values folder. Here's a snippet of one of these files:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Waterpolo Instant Score</string>
    <string name="ui_refresh">Refresh</string>
</resources>
```

For localization to work smoothly, you need to reference entries in this file from anywhere in your code. Here's how you would set the caption of a button to a localizable string in a layout file:

```
<Button android:id="@+id/buttonRefresh"
        style="@style/button “
        android:text="@string/ui_refresh">
</Button>
```

You use the *@string* expression to prefix the string ID. In code, you use the *getString* method as defined on the *Activity* class.

```
String caption = getString(R.string.ui_refresh);
```

Strings for the default language of the application should go in the Strings.xml file located in the Res/Values folder. For any localized version you may have, you create a Res/Values-*xx* folder, where *xx* are the two letter initials of the language. For example, it would be Res/Values-it for Italian and Res/Values-es for Spanish. The system will pick up the string ID from the correct container based on the current locale.

If a string is not found in the current localized set of strings, then the system will fall back to the value stored in the default container.

## Examining a Sample Application

Let's view what it takes to build the same Android application from Chapter 8. As you may recall, Guess is the sample multiplatform application that you get as companion code for this book. Guess implements a simple game: guessing a number in a given range. Figure 9-9 shows the Main screen and the Game screen of the application.

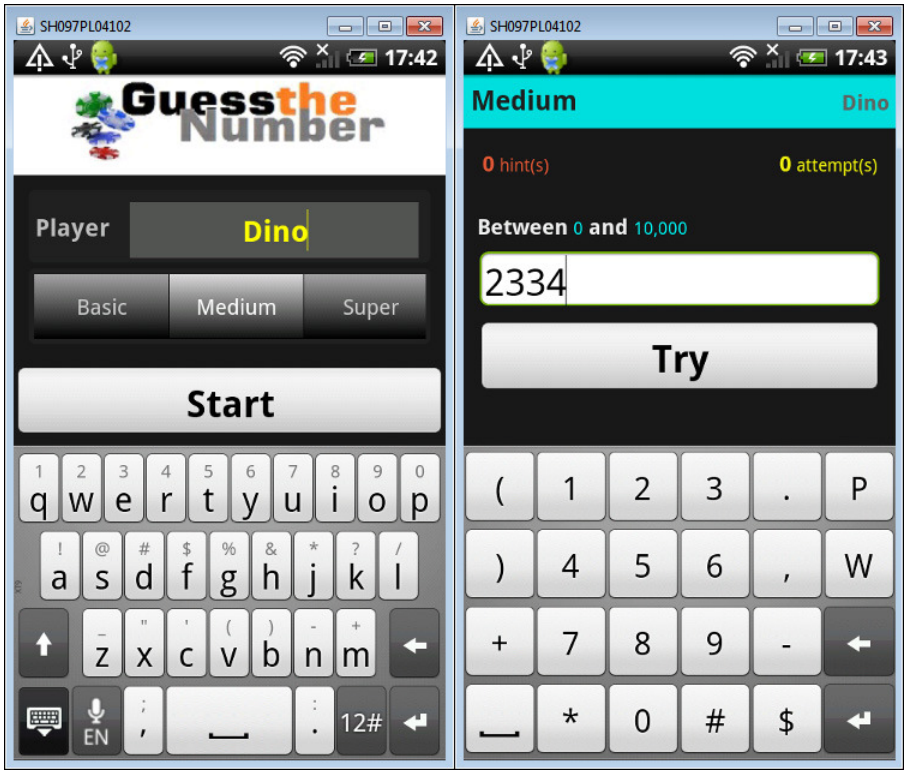



**FIGURE 9-9** The Guess application in action.

The application consists of a Home view, where you enter the player's name, and a Play view, where you enter a number and get a response. A Scores view and an About view complete the application.

**Note** To obtain a numeric-only keyboard in Android, you should set the attribute *android:inputType* to *phone* and explicitly assign a value to the *android:digits* attribute, such as "0123456789."

## The Application Manifest

The Guess application is based on four activities—the Home screen (the application's main screen), the Game screen, the score list, and the About screen. Navigation between screens occurs when the user taps a button or a menu item.

The application manifest file is shown here:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.expoware"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.VIBRATE"/>
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:name="GuessActivity"
                  android:windowSoftInputMode="stateVisible|adjustPan"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".GameActivity"
                  android:windowSoftInputMode="stateVisible|adjustPan" />
        <activity android:name=".AboutActivity" />
        <activity android:name=".ScoreListActivity" />
    </application>
</manifest>
```

In addition to declaring the aforementioned activities (one of which is the launcher), the manifest declares that the application may be using the phone vibrator. Beside this, the other remarkable point of the manifest is the use of the *windowSoftInputMode* attribute on the activities (*GuessActivity* and *GameActivity*) that show some interactive user interface.

## The Home View

The user interface of the home screen (see Figure 9-9) results from the vertical composition of four blocks: the title bar, player bar, level bar, and command bar. Here's the full layout as saved in the project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/home"
              android:orientation="vertical"
              style="@style/home"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent">
    <LinearLayout android:id="@+id/titleBarArea"
                  style="@style/titleBarArea"
                  android:layout_width="fill_parent"
                  android:layout_height="75dp">
        ...
    </LinearLayout>
    <RelativeLayout android:id="@+id/playerBar"
```

```
                    style="@style/gameConfigArea"
                    android:orientation="horizontal"
                    android:layout_width="fill_parent"
                    android:layout_height="wrap_content">
        ...
    </RelativeLayout>
    <RelativeLayout android:id="@+id/levelBar"
                    style="@style/gameConfigArea"
                    android:orientation="horizontal"
                    android:layout_width="fill_parent"
                    android:layout_height="wrap_content">
           ...
    </RelativeLayout>
    <RelativeLayout android:id="@+id/commandBar"
                    style="@style/commandArea"
                    android:layout_width="fill_parent"
                    android:layout_height="wrap_content">
            ...
    </RelativeLayout>
</LinearLayout>
```

The use of a *RelativeLayout* component makes it possible to place widgets relative to others in the same block. For example, in Figure 9-9, you see the label "Player" side by side with a text box. Here's the markup that you may use to place them side by side without using absolute measurements:

```
<TextView android:id="@+id/playerNameLabel"
          style="@style/playerNameLabel"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/playerLabel" />
<EditText android:id="@+id/playerNameEdit"
          style="@style/playerNameEdit"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content" />
```

All the graphical details for both widgets are actually stored in the style elements, as shown here:

```
<style name="playerNameLabel">
        <item name="android:layout_marginTop">10dp</item>
        <item name="android:gravity">center_horizontal|center_vertical</item>
        <item name="android:textStyle">bold</item>
        <item name="android:textSize">18dp</item>
</style>
<style name="playerNameEdit">
        <item name="android:layout_marginTop">5dp</item>
        <item name="android:layout_marginRight">5dp</item>
        <item name="android:layout_marginLeft">15dp</item>
        <item name="android:layout_toRightOf">@+id/playerLabel</item>
        <item name="android:gravity">center_horizontal</item>
        <item name="android:padding">5dp</item>
        <item name="android:inputType">textPersonName|textCapWords</item>
</style>
```

It is interesting to note the combined effect of *layout_marginLeft* and *layout_toRightOf*. The latter places the text box to the right of the label (identified by ID), whereas the former sets a margin between the two.

Finally, keep in mind that as a developer, you always should strive to make input as easy as possible for users. Loading the most convenient keyboard layout is the primary aspect of this effort. In this example, the text box where the user is expected to type the player's name can have the *inputType* attribute configured as *textPersonName* and *textCapWords*. This guarantees that the first letter is always uppercase and that the user gets proper suggestions related to person names (see Figure 9-10).

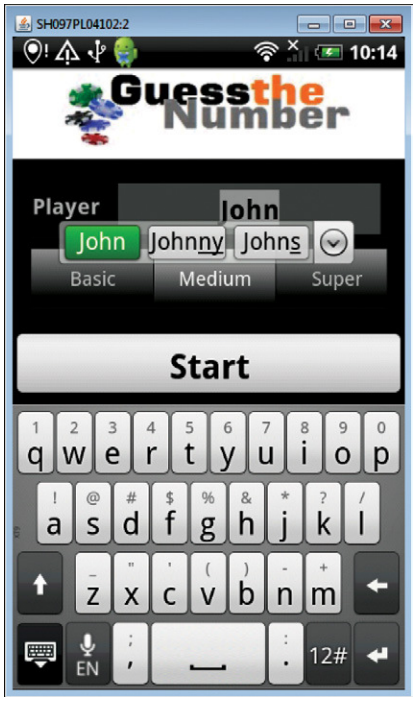



**FIGURE 9-10** Context-sensitive auto-completion in a text box.

The Chapter 8 example used a segmented button control to let users select a level. Segmented buttons are a special flavor of radio button that is typical of iOS. In Android 2.*x*, you don't have such a compelling component, but you can achieve the same effect using plain radio buttons:

```
<RadioGroup android:id="@+id/levelGroup"
            android:orientation="horizontal"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">
    <RadioButton android:id="@+id/level_basic"
                 android:layout_width="100dp"
                 android:layout_height="wrap_content"
                 android:text="Basic" />
     ...
</RadioGroup>
```

The difference is merely graphical. Segmented buttons exist natively in more recent levels of the Android API (i.e., version 4.0) but you can find snippets of some fallback implementations easily. For the UI shown in Figure 9-10, I started from the code discussed here: *http://goo.gl/a5Zp9*.

**Note** In addition to radio buttons, you could have considered an Android *spinner*—in other words, a drop-down list. The reason why I ruled spinners out is that it would require two taps to make a selection: the first tap to open the list and the second tap to make the selection. A spinner is good when the list of options is longer than just two or three elements. If the choices can be comfortably listed in the width of a mobile device, you should opt for radio buttons because they allow a quicker selection for the user.

When you start the application and the Home screen shows up, the (soft) keyboard is already visible and the input focus is on the text box. There are a couple of things you can do here to improve the user's experience. First, you can manage to ensure that the keyboard is visible so that the user can start typing right away. Second, you can remember name and level so that if the same player returns, she doesn't have to type her name again. On the other hand, isn't the device a (mostly) personal item?

In general, popping up the (soft) keyboard may be problematic because it may cover a significant part of the user interface. Without beating around the bush, this is a problem that you must address on a per-case basis. However, Android makes some tools available. For example, you can set the *windowSoftInputMode* attribute on the activity to control how the keyboard is displayed. In Guess, the attribute is set to a couple of values: *stateVisible* and *adjustPan*. The former entails that the keyboard is always displayed when the user navigates forward to the activity, but not when the user navigates back to the activity. (The *stateAlwaysVisible* value will keep it visible in all cases instead.)

As the (soft) keyboard pops up at the bottom of the screen, the rest of the user interface may be hidden from view. By default, Android scrolls the current window so that the view with input focus is visible along with the keyboard. This approach is called *panning*. In alternative to panning, there's resizing where (if possible) the window is resized to make room for the keyboard. You set resizing by using the *adjustResize* attribute.

## The Play View

When the user taps the Start button, the application saves the player name and level as default settings for future games and launches the Game activity. This new activity needs to know about player name and level to adjust its user interface and scoring system. Here's how you push another view to screen passing some arguments:

```
public void StartGame()
{
    // Figure out selected level and player
    GameLevel selectedLevel = getSelectedLevel();
    String player = PlayerNameEditRef.getText().toString();

    // Save current status for later
```

```
    PreferenceHelper.SaveGame(this, new GameInfo(player, selectedLevel));

    // Start the new activity and pass data explicitly
    Intent i = new Intent(this, GameActivity.class);
    i.putExtra(Constants.PlayerNameKey, player);
    i.putExtra(Constants.GameLevelKey, selectedLevel.ordinal());
    startActivity(i);
}
```

The new activity sets up its user interface from the layout file and then tweaks it with passed data:

```
String playerName = getIntent().getStringExtra(Constants.PlayerNameKey);
GameLevel level = GameLevel.Basic;
int index = getIntent().getIntExtra(Constants.GameLevelKey, -1);
if(index >= 0 && index < GameLevel.values().length)
   level = GameLevel.values()[index];
Game currentGame = new Game(playerName, level);
```

The methods *getIntExtra* and *getStringExtra* are used to extract data associated with the starter intent. Next, the Game activity sets up an internal class—the *Game* class—that holds the logic of the game.

The user types a number from a numeric keyboard (see Figure 9-9) and taps the Try button. When this happens, a method is invoked on the *Game* class. If the typed value is outside the current range of values where the number to guess lies, then a toast message is displayed and the device vibrates:

```
public void NewAttempt(int number)
{
   boolean isValid = currentGame.isValidAttempt(number);
   if (!isValid)
   {
       // Toast
       Toast theToast = Toast.makeText(this,
               getString(R.string.InvalidAttempt),
               Toast.LENGTH_SHORT);
       theToast.show();

       // Vibrate
       Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
       v.vibrate(300);     // milliseconds

       return;
   }
   ...
}
```

If the attempt is not successful, the activity refreshes the status of the game, updating the number of attempts. Android allows you to use an HTML-like syntax to compose text where you use mixed styles. Here's an example that renders the string "Between X and Y" using lighter weight and different color for the boundaries of the range:

```
String rangeTemplate =
          "<b>%s </b> <small><font color='#00ffff'>%,d</font></small>" +
```

```
          "<b> %s </b> <small><font color='#00ffff'>%,d</font></small>";
String rangeMarkup = String.format(rangeTemplate,
          getString(R.string.betweenLabel), currentGame.Lowest,
          getString(R.string.andLabel), currentGame.Highest);
StatusLabelRef.setText(Html.fromHtml(rangeMarkup));
```

The static method *fromHtml* takes a HTML-like string and produces an Android-compatible markup that can be displayed on a view. In this example, we're assigning an HTML-like output to a plain *TextView* widget.

The Game activity features a menu with two options—*Hint* to get a hint about a number you can try and *Quit* if you wish to surrender and stop the game. This interface is shown in Figure 9-11.

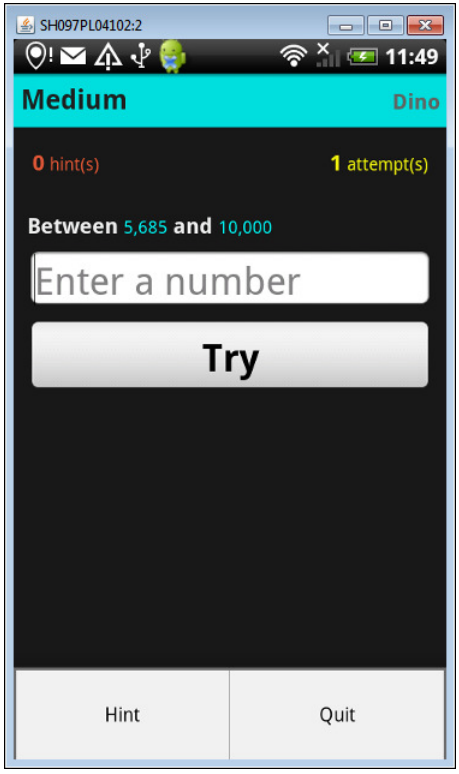



**FIGURE 9-11** A context-sensitive menu displayed during the game.

When you tap the Menu button to display the menu, the keyboard disappears, and you need to give explicit focus to the text box to have it come back and continue the game. If you choose to quit the game, a message box is displayed that looks like Figure 9-12.

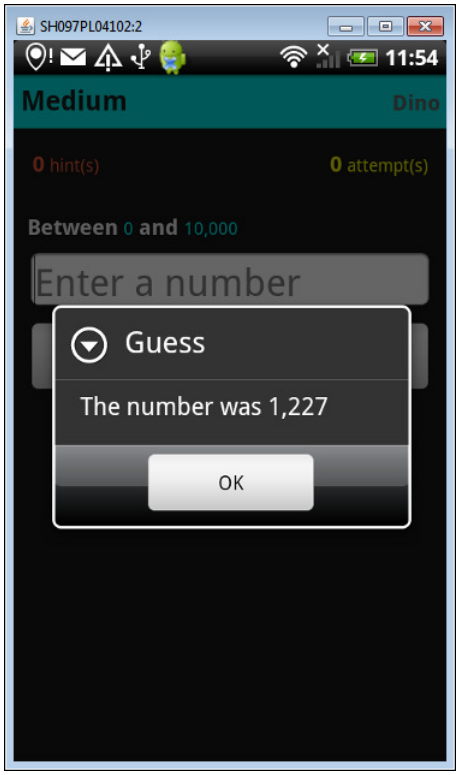



**FIGURE 9-12** A message box displayed when the game is over.

In Android, you can create a single global instance of the *AlertDialog* class and use it to display any messages throughout the activity. In the sample application, I have an initializer class for each activity where I do all preparation work, including saving references to UI widgets, attaching event handlers, and preparing ready-to-use objects for pop-up and toast messages:

```
public class GameActivityInitializer
{
    private static AlertDialog theDialogBox;

    public static void MessageBox(Activity parent, String message)
    {
        theDialogBox.setButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                dialog.dismiss();
            }
        });
        theDialogBox.setMessage(message);
        theDialogBox.setTitle(parent.getString(R.string.app_name));
        theDialogBox.show();
    }

    public static void PrepareWidgets(GameActivity parent)
    {
        final GameActivity theParent = parent;
```

```
        if (theParent == null)
            return;

        // More code here
        ...

        // Prepare the dialog box object
        AlertDialog.Builder builder = new AlertDialog.Builder(parent);
        theDialogBox = new AlertDialog.Builder(parent).create();
    }
}
```

Note that you can't use any of the aforementioned HTML facilities in a plain alert dialog box. So if you want to display a pop-up dialog box with colors and widgets, you have to create a custom dialog box. This is precisely what you want to display when the user finally guesses the number (see Figure 9-13).

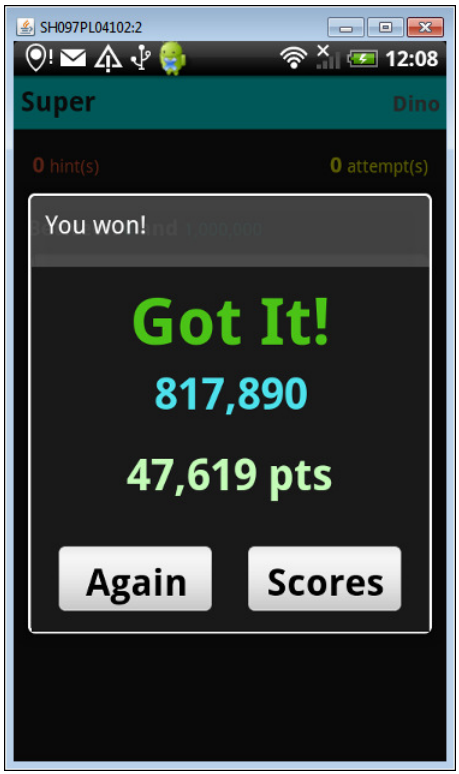



**FIGURE 9-13** The dialog box for the winner.

You need to do a number of things when the user wins the game. You might want to hide the keyboard, display the dialog box from Figure 9-13, and reset the game status. Here's the code to do all this:

```
InputMethodManager im = (InputMethodManager) getSystemService(Context.INPUT_METHOD_SERVICE);
im.hideSoftInputFromWindow(
        this.getCurrentFocus().getWindowToken(),
```

```
        InputMethodManager.HIDE_NOT_ALWAYS);

// You won!
YouWonDialog dialog = new YouWonDialog(this, new WinDialogEventListener(currentGame));
dialog.show();

// Quit game and fix underlying UI
SaveGame();
currentGame.quit();
UpdateGameStatus();
```

The dialog box is the most interesting aspect of this part of the application. Let's find out more. Here's an excerpt from the *YouWonDialog* custom class:

```
public class YouWonDialog extends Dialog implements View.OnClickListener
{
    public static String ScoredPointsKey = "Points";
    public static String SecretNumberKey = "Number";

    private Button startButton, scoresButton;
    private TextView pointsScored, secretNumber;
    private IDialogListener _onYouWonDialogListener;
    private Context currentContext;

    public YouWonDialog(Context context, IDialogListener listener)
    {
        super(context);
        _onYouWonDialogListener = listener;
        currentContext = context;

        setContentView(R.layout.winner);
        setTitle(R.string.youWon);

        againButton = (Button) findViewById(R.id.againButton);
        scoresButton = (Button) findViewById(R.id.scoresButton);
        pointsScored = (TextView) findViewById(R.id.scoreLabel);
        secretNumber = (TextView) findViewById(R.id.secretNumber);
        startButton.setOnClickListener(this);
        scoresButton.setOnClickListener(this);
    }
    ...
}
```

The dialog box is based on a layout file set in the class constructor. The class also receives an object of type *IDialogListener,* which is a custom interface used to abstract the way in which the dialog box communicates with the rest of the application. More often than not, dialog boxes need to receive data to display and return the processed data. The interface streamlines the process:

```
public interface IDialogListener
{
    public void Completed(Bundle bundle);
    public Bundle Initialize();
}
```

The dialog box receives initialization data from *Initialize* and returns values (if any) through *Completed*. The caller—in this case the *Game* activity—will pass a dialog listener object to the *YouWonDialog* class.

The two buttons—Scores and Again—share the same handler but behave in a slightly different way:

```
public void onClick(View v)
{
   if (v == againButton)
   {
      Bundle bundle = new Bundle();   // In case you need to return values from the dialog
      _onWinDialogListener.Completed(bundle);
      dismiss();
   }

   if (v == scoresButton)
   {
      Intent i = new Intent(currentContext, ScoreListActivity.class);
      currentContext.startActivity(i);
      dismiss();
   }
}
```

The Again button just dismisses the dialog box and returns in Play mode; instead, the Scores button navigates to the Scores view.

## The Scores View

The Scores view essentially consists of a list of data items. In similar situations, you can derive the activity class from a special system-provided class—the *ListActivity* class. The *ListActivity* class is designed to display a list of items taken from a data source, such as an array. The *ListActivity* class extends the base *Activity* class by adding ad hoc methods to handle item selection and data binding.

*ListActivity* is one of the Android classes to ship with a default layout—a full-screen list view. This means that if you simply pass data, then you still can have your data displayed through a default layout. You are welcome to change the default layout using the *setContentView* method. However, it is mandatory that any custom layout that you provide contains a *ListView* object with the *id* attribute set to *@android:id/list*. You don't need the + symbol here because *list* is a system-defined ID and doesn't have to be added dynamically to the *R* class. Nicely enough, you also can specify a custom view for when the list view is empty. This empty view must have the *id* attribute set to *@android:id/empty*. Here's the layout for our scores view. It contains the default *ListView* plus a text header:

```
<LinearLayout android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:background="#393939">
    <TextView android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:gravity="center_horizontal"
              android:text="@string/topScorer">
    </TextView>
```

```
    <ListView android:layout_height="fill_parent"
              android:layout_width="fill_parent"
              android:layout_marginTop="20dp"
              android:id="@android:id/list">
    </ListView>
</LinearLayout>
```

Here's the skeleton of a sample list activity that will display the best scores of the game:

```
public class ScoreListActivity extends ListActivity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.scoreslist);

        // Load saved scores
        LoadScores();
    }

    private void LoadScores()
    {
        ArrayList<ScoreInfo> scores = ReadScoresFromHistory();
        Bind(scores);
    }

    private ArrayList<ScoreInfo> ReadScoresFromHistory()
    {
        GuessHistory history = PreferenceHelper.LoadHistory(this);

        // Sort results
        Collections.sort(history.Scores, new ScoreComparator());
        return history.Scores;
    }

    private void Bind(ArrayList<ScoreInfo> scores)
    {
        ArrayAdapter<ScoreInfo> adapter = new ScoresAdapter(this, scores);
        setListAdapter(adapter);
    }
    ...
}
```

Let's assume for now that we know how to load and save data to a permanent local storage. (I'll return to Android data persistence in a moment.) The saved data is an array of *ScoreInfo* objects:

```
public class ScoreInfo implements Serializable
{
    public int Points;
    public String DateOfScore;
    public String Name;
```

The class must be serializable because we intend to save a list of instances to the file system so that the best scores can be tracked and the list populated. In the preceding code, we read from the file system and copy scores to some sortable array; next we invoke data binding.

When the Scores view is being displayed, we load an array of *ScoreInfo* objects from the storage and manage to display it. When a new game is finished, the new score is added to the list and persisted. Once we hold the array of scores, you arrange some data binding logic to display it.

In Android, data binding requires the services of an adapter object. The adapter knows about the *ListView*, data items, and the template of individual cells. The adapter is the repository of your data binding logic. The adapter holds the data to display and knows about the list and its layout. To populate a list, all you need to do is assign a proper adapter to the list. The *setListAdapter* method is built into the *ListActivity* class just to bind the default list to the given adapter.

Here's the code for a score adapter that uses a custom cell template named *listitem*. You define a custom cell template as a layout file:

```
public class ScoresAdapter extends ArrayAdapter<ScoreInfo>
{
    private final Activity context;
    private final ArrayList<ScoreInfo> scores;

    public ScoresAdapter(Activity context, ArrayList<ScoreInfo> currentScores)
    {
        super(context, R.layout.listitem, currentScores);
        this.context = context;
        this.scores = currentScores;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent)
    {
        // Some code here
    }

    }
```

The *getView* method is the heart of an adapter. The method is invoked for each cell that fits into the currently visible part of the list view. The implementation shown here used the *view-holder* pattern that is very similar to what was shown in Chapter 8 for iOS.

The *getView* method receives a *View* object that is one of the cells the system will render. The *position* argument tells you whether you're currently rendering the first or, say, the fifth visible cell in the list. If the cell view object is null, you create it using a system inflater (a sort of factory for layout elements) and the layout of choice; otherwise, you retrieve a valid cell object via the view-holder pattern:

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
{
    ViewHolder holder;
    View listItemView;

    listItemView = convertView;
```

```
    if (convertView == null)
    {
        LayoutInflater inflater = context.getLayoutInflater();
        listItemView = inflater.inflate(R.layout.listitem, null, true);
        holder = new ViewHolder();
        holder.ScoreDate = (TextView) listItemView.findViewById(R.id.scoreDate);
        holder.ScorePlayer = (TextView) listItemView.findViewById(R.id.scorePlayer);
        holder.ScorePoints = (TextView) listItemView.findViewById(R.id.scorePoints);
        listItemView.setTag(holder);
    }
    else
    {
        holder = (ViewHolder) convertView.getTag();
    }

    // Set the text/icon to display
    String pts = String.format("%,d", scores.get(position).Points);
    holder.ScorePoints.setText(pts);
    holder.ScoreDate.setText(scores.get(position).DateOfScore);
    holder.ScorePlayer.setText(scores.get(position).Name);
    return listItemView;
}

static class ViewHolder
{
    public TextView ScorePoints;
    public TextView ScorePlayer;
    public TextView ScoreDate;
}
```

You define a *ViewHolder* internal class and give it as many members as there are bindable widgets in the cell template. When the cell is null, you store in a new *ViewHolder* instance references to bindable widgets and then cache the view holder instance within the cell instance using the *setTag* method. At the end of the rendering, the *ListView* has a cell object for each of displayed cells—probably less than the bindable items.

On the next display (or during scrolling), it may happen that the *ListView* has to render, say, the first cell. This time, though, the content may be different, but not the references to the widgets in the cell template. The cached *ViewHolder* instance saves you from repeatedly calling into the expensive *findViewById* method. Once you've got your *ViewHolder,* you simply set its properties to the actual value—as *ViewHolder* members reference widgets any updates is reflected to the actual cell. Figure 9-14 shows the final results.

**Note** The *ViewHolder* class is defined as static to save keeping a reference to the outer class and to avoid access to any members of the containing class.


**FIGURE 9-14** The Scores view.

## Other Programming Topics

The sample Guess application touches on a variety of aspects of mobile and Android programming. The list of hot topics, though, doesn't end here. In this section, we'll discuss storage, network access, and common tasks such as sending an email or a SMS.

### Permanent Data Storage

Android offers three different places to save data permanently: in a dictionary of preferences, file streams, and SQLite database tables.

You have a dictionary model in the *SharedPreferences* object, which is appropriate for a small amount of partially related data, such as player name and game level in the previous example. For data that don't fit the dictionary model, you might want to try streams or relational SQLite tables. Let's find out more about shared preferences.

To save simple stand-alone values such as settings or preferences, you need an instance of the *SharedPreferences* object plus an ad hoc editor object if you are writing to it:

```
SharedPreferences storage = getSharedPreferences(filename, MODE_PRIVATE);
```

Shared preferences are saved to an XML file with the given name. The visibility of the file depends on the second argument that you specify. *MODE_PRIVATE* indicates that only the current application has access to the file. Other options are *MODE_WORLD_READABLE* and *MODE_WORLD_WRITEABLE,* which would make the content visible to every application for reading, writing, or both. To save data, you also need an editor. Here's how to proceed:

```
SharedPreferences storage = getSharedPreferences(filename, MODE_PRIVATE);
SharedPreferences.Editor editor = storage.edit();
editor.putString("Player", playerName);
editor.putInt("Level", gameLevel);
editor.commit();
```

To write data to the system dictionary, you use an API that is nearly identical to that of the *Bundle* class. Methods like *putInt* and *putString* take a key and a value and update the internal dictionary. When you commit the editor, data is written to an underlying XML file.

A nice feature of the preferences API is that when it comes to reading, you don't have to worry about exceptions. There's no need to check whether the XML file really exists. You just try to read and provide default values should the read fail for any reason, as follows:

```
SharedPreferences storage = getSharedPreferences(filename, MODE_PRIVATE);
String player = storage.getString("Player", "");
int level = storage.getInt("Level", GameLevel.Basic);
```

The *SharedPreferences* object supports a bunch of types natively. In addition to strings, you can read and write primitive data types such as *int, long, float,* and *Boolean*.

## Accessing the Network

In a mobile application, checking the state and the type of the network before embarking in Internet operations is crucial. In some cases, you also might want to apply different algorithms or take different countermeasures if the user is connected via a WiFi network or a mobile network. Large uploads of documents—for example, a synchronization procedure between client and server—should be planned carefully: the more it takes to upload, the higher the risks that the network will go down. Handling these nasty situations is up to you. Checking the network state and type is not a big deal. It only requires that you become familiar with a couple of Android API functions.

In Android, programmatic access to the Internet is subject to permissions. In particular, you need to declare in the application manifest that the application is going to check the network state and access the Internet. Why is it also required to declare the intention of simply checking the network state?

In Android, you check the network state via the *ConnectivityManager* class—a system class that doesn't simply exist to answer your inquiries but it also monitors all possible connections (including WiFi, UMTS, and GPRS) and their changes. The class may broadcast messages about network changes and has the ability to try to fail over to another type of network when connectivity is lost. In a nutshell, the *ConnectivityManager* class allows applications to access any available information about any available networks. As you can see, it is a rather critical class and Android designers figured that

you must declare its use so that users can be notified of that when they install the application. Here's the code you need to enter in the application's manifest file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.expoware"
          android:versionCode="1"
          android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <application android:label="Guess"
                 android:icon="@drawable/icon">
       ...
    </application>
</manifest>
```

The *ConnectivityManager* class offers two key methods—*getNetworkInfo* and *getActiveNetworkInfo*. Both methods return a *NetworkInfo* object through which you can examine the state of the network and make your further decisions. The former method checks just the specified type of network (WiFi, GPRS, and the like); the latter, instead, checks the currently active network, if any. Here's how to use the class to check if connectivity is available:

```
public class HttpHelper {
    public static Boolean IsInternetAvailable(Context context)
    {
        ConnectivityManager cm;
        cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo ni = cm.getActiveNetworkInfo();
        if (ni != null && ni.isAvailable() && ni.isConnected())
            return true;
        return false;
    }
}
```

If you invoke the *getSystemService* method from within an *Activity* class, then you don't need to obtain a *Context* object first. Otherwise, the simplest thing to do is just pass the context as an argument to any helper class.

## Placing HTTP Calls

In Android, downloading data via the Hypertext Transfer Protocol (HTTP) is quite simple; on the other hand, processing downloaded data into usable data objects is more bothersome than, say, in .NET. The Android API lacks some of the facilities that make downloading data from the web easy in Windows Phone. Here's the code required to arrange a *GET* call:

```
String url = "...";
String responseText = "";
try {
    DefaultHttpClient client = new DefaultHttpClient();
    HttpGet request = new HttpGet();
    request.setURI(new URI(url));
    HttpResponse response = client.execute(request);
    InputStream in = response.getEntity().getContent();
```

```
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    responseText = reader.readLine();
    reader.close();
}
catch(URISyntaxException uriSyntaxException)
{
}
catch(IOException ioException)
{
}
```

As a first step, you get a new instance of the *DefaultHttpClient* class. This class governs the execution of the request, grabs request headers and body, and downloads a response packet. The request itself is packaged in an ad hoc class that depends on the HTTP verb to use: you have *HttpGet* for a GET command and *HttpPost* for a POST command. You also have analogous classes for other HTTP verbs such as *HttpDelete*, *HttpPut,* and *HttpHead*.

On the request object, you can set headers and the URL to call. To execute a request, you call the *Execute* method on the *DefaultHttpClient* object. This method works synchronously and returns an *HttpResponse* object. The response object always wraps a final response with a valid HTTP status code.

**Note** You can wrap the call in a new thread to avoid blocking the user interface. Alternatively, you can use your own class derived from AsyncTask and perform any background work while still being able to update the user interface. For more information on AsyncTask, you can refer to the documentation at *http://developer.android.com/reference/android/os/AsyncTask.html*.

In synchronous operations, you need to write some ad hoc code to extract usable data from the response stream. You first access the content being returned as a stream:

```
InputStream in = response.getEntity().getContent();
```

Next, you get a reader to read the content of the stream. If you want to read piecemeal, you can stick to a basic reader. If you want to read lines of text, you may use the *BufferedReader* class:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(in));
responseText = reader.readLine();
```

If the response is on a single line—a fairly common scenario—then the *BufferedReader* class helps with its *readLine* method. If you want to read the entire content regardless of the number of lines, you have to implement a loop yourself.

To post data to a web server, you change the class that you use for the request: instead of *HttpGet*, you use *HttpPost*. Beyond that, placing a *POST* call differs from a *GET* call for the extra work that it takes to write name/value pairs in the body of the request packet. Here's the code:

```
DefaultHttpClient client = new DefaultHttpClient();
HttpPost request = new HttpPost(url);
try {
```

```
    List<NameValuePair> data = new List<NameValuePair>();
    data.add(new BasicNameValuePair("key", "value"));
    ...
    request.setEntity(new UrlEncodedFormEntity(data));
    HttpResponse response = client.execute(request);
}
catch()
{
  ...
}
```

You first create a name/value pair to populate the body of the POST request and then attach the dictionary to the request.

## Broadcasters

The *IsInternetAvailable* method we discussed works great if called at the beginning of a network operation. In mobile, much more than in a desktop scenario, the connectivity can come and go at any time. From the application perspective, the connectivity change is an asynchronous event, whereas the *IsInternetAvailable* method operates synchronously. To detect asynchronous events like the change of the network state, you define a broadcast receiver and set it to listen to connectivity change events. This entails writing a class that inherits from *BroadcastReceiver*, as shown here:

```
public class NetworkStateReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent) {
        // Your reaction here
    }
}
```

The canonical behavior of a broadcast receiver is expressed by the *onReceive* method. The method provides you with the context of the Android application that you can use to retrieve global objects.

Writing the receiver class is only the first step. The next is registering the receiver. A receiver can be registered in either of two ways—statically in the application's manifest or programmatically in the application's main activity. The two methods are not equivalent and produce different run-time conditions. Let's tackle static registration first.

You add a *<receiver>* section in the manifest file and indicate one or more intent filters to specify which events you're interested in. Here's the manifest file for a sample application that reacts in some way to network state changes:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.expoware.hidoing"
          android:versionCode="1"
          android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

```
    <application android:label="Hi-Doing" android:icon="@drawable/icon">
        <activity android:name="HiDoingActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="NetworkStateReceiver">
            <intent-filter>
                <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

The name attribute of the <*action*> element just contains one of the predefined messages: *CONNECTIVITY_CHANGE* is just the system-provided message that identifies broadcast messages about changes in the connectivity state.

With this configuration in place, the *onReceive* method of the *NetworkStateReceiver* class will be invoked every time your device gets connected or disconnected to the network—regardless of whether it is WiFi or 3G.

Statically registered receivers are entirely managed by the system that is responsible for their life cycle. In a way, these receivers are global and referenced in a system table so long as the host application is installed on the device. When the event the receiver registered for occurs, the system always will call the receiver, regardless of whether the process that hosts the receiver is running or not.

It turns out that static registration is good for special flavors of Android application that are designed to run as services—always in the background, mostly UI-less and guaranteed to kick in under certain conditions. If you want to capture the network state change event within the boundaries of the running application—and use it to update the user interface—then you must opt for dynamic registration.

Dynamic registration occurs with a few lines of code that you execute upon application startup and resume. In addition, when the application is paused, you also need to unregister the receiver. Let's see what it takes to handle two related events—network access and change of connectivity type. You begin by adding a couple of private members to the main activity class:

```
private BroadcastReceiver networkStateReceiver;
private IntentFilter connectivityChangeFilter;
```

You initialize these members upon application startup (typically in *onCreate*), as shown here:

```
connectivityChangeFilter = new IntentFilter("android.net.conn.CONNECTIVITY_CHANGE");
networkStateReceiver = new NetworkStateReceiver();
registerReceiver(networkStateReceiver, connectivityChangeFilter);
```

You can call *registerReceiver* multiple times to add multiple intent filters. You repeat the same code also in *onResume*—the method invoked on the activity when an Android application resumes from

the background. To stay on the safe side, you also unregister receivers from within *onPause*—namely, when the Android application makes it to the background:

```
@Override
public void onResume()
{
    super.onResume();
    if (networkStateReceiver != null && connectivityChangeFilter != null)
         registerReceiver(networkStateReceiver, connectivityChangeFilter);
}
@Override
public void onPause()
{
    super.onPause();
    if (networkStateReceiver != null)
        unregisterReceiver(networkStateReceiver);
}
```

With this code in place, the receiver applies only to your application and is not invoked when some of the monitored events occur but the host application is not running. The life cycle of the receiver object terminates right after calling the *onReceive* method.

A receiver that is tightly bound to a single application likely will need to update the user interface when the event occurs. For example, you may want to enable or disable a few UI buttons when the connectivity is down.

You add a custom constructor to the receiver class and pass it any references to any objects that it may need to call. The code shown here passes the *Activity* reference through the constructor; the *Activity* reference is cached and used when it is time to refresh the user interface:

```
public class NetworkStateReceiver extends BroadcastReceiver
{
    HiDoingActivity _main;
    public NetworkStateReceiver(HiDoingActivity main)
    {
        _main = main;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        _main.RefreshView();
    }
}
```

The code shown so far detects only whether the application is (or is not) connected to the Internet. If in addition, you want to monitor when the connection type changes (i.e., from WiFi to 3G and vice versa), then you need to add a second intent filter, as shown here:

```
connectivityChangeFilter = new IntentFilter("android.net.conn.CONNECTIVITY_CHANGE");
wifiChangeFilter = new IntentFilter("android.net.wifi.STATE_CHANGE");
networkStateReceiver = new NetworkStateReceiver();
registerReceiver(networkStateReceiver, connectivityChangeFilter);
registerReceiver(networkStateReceiver, wifiChangeFilter);
```

Next, in *onReceive,* you can get a reference to the connectivity manager and check the type of the network you're connected to:

```
@Override
public void onReceive(Context context, Intent intent)
{
    // Get the connectivity manager
    ConnectivityManager cm;
    cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);

    // Get network information
    NetworkInfo wifi =  cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
    NetworkInfo mobile = cm.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);

    // Do your checks
    if (wifi.isAvailable())
    {
        // Handle wifi scenario
    }
    if (mobile.isAvailable())
    {
        // Handle 3G scenario
    }
}
```

Compared to web and desktop applications, the state of connectivity in mobile is much more unstable. It may go down or change type in a matter of seconds. Therefore, it is crucial for applications to be ready to detect changes and react properly.

**Note** When you opt for statically registered receivers, you can attach and detach them as you like, per screen or per application, depending on the use that you're planning.

## Common Tasks

Nearly any graphical user interface (GUI) platform has some reusable pieces of code that applications can use to implement common tasks in the same guise as the system. In Windows, you have common dialog boxes to select and save a file or pick up a color or a font. Mobile platforms are no exception. In mobile, reusable system code is applets to send an email, take a picture, select a contact, send a text message, and more.

In Android, you use *intents* for executing a variety of system tasks, such as sending a text message, placing a phone call, or, as we'll see in a moment, capturing a picture and sending it via email.

You have two ways for working with the device camera in Android. You can choose to operate at a very low level and use the camera API, or you can rely on the services of the built-in application. In the latter case, you need to get an instance of the corresponding intent and just pass it to *startActivity*:

```
Intent camera = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
startActivity(camera);
```

Notice that you don't need to declare any special capabilities in the application's manifest if you simply use the camera intent. Instead, if you use the API directly, then it is required that you add the following permission (and feature) to the manifest:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
```

**Important** The reason that the camera intent is not subject to permissions is that the task is an interactive one that can't just start and complete without explicit user consent. Permissions serve the purpose of warning users about things that the software may be doing that can cost something to users. Having the user take a picture is an explicit and interactive action, so no permissions are required. The same applies to emails and text messages: no permissions are required if you bring up the system application. A permission is required if you intend to send emails or SMS messages silently.

All in all, just calling the default camera application is kind of pointless; you need to establish a bit of interaction to make things more interesting. At the very minimum, you want to instruct the application to save the taken picture to a particular location and with a given name:

```
String capturedImage = Environment.getExternalStorageDirectory().getAbsolutePath() + picName;
Uri uri = Uri.fromFile(new File(capturedImage));
camera.putExtra(MediaStore.EXTRA_OUTPUT, uri);
```

The *putExtra* method on the *Intent* class adds the specified Uniform Resource Identifier (URI) as the output file name. The URI is just a value added to the data collection of the intent; the code associated with it, though, adds a role to the data and contains instructions for a particular intent. You can choose to save the file to the SD card (as in the example) or the local memory. In both cases, if you use directories, you should ensure that the entire tree exists.

What if you want to display a preview and maybe send an email later? In this case, you need to claim a notification for when the camera intent has done its job—that is, after the user has accepted the picture through the standard *Done/Cancel* pair of buttons.

When you need to receive a response from a started activity, you no longer use the method *startActivity* but opt for *startActivityForResult* instead:

```
startActivityForResult(camera, 435);     // 435 is your request code
```

The numeric parameter in the call is arbitrary and indicates a request code. In other words, because the same handler will be invoked for any results from any launched intent, you use that code to understand which action is required on your own. The request code is a way for the handler to select notifications and ignore those that just don't apply. The notification comes through a protected virtual method of the activity that starts the intent—the *onActivityResult* method:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent)
{
    if (requestCode != 0)      // Use 0 or whatever request code you set
        return;
    if (resultCode != -1)      // User cancelled photo capture
        return;
    ...
}
```

The *resultCode* parameter indicates whether the user accepted or rejected the photo. It is –1 if you clicked on *Done;* it is 0 otherwise.

If you want to send the picture that you've just taken via email, you need to invoke another intent:

```
Intent email = new Intent(android.content.Intent.ACTION_SEND);
startActivity(Intent.createChooser(email, "Send mail..."));
```

The *createChooser* method is responsible for the additional step of choosing the medium to share the picture through (see Figure 9-15).

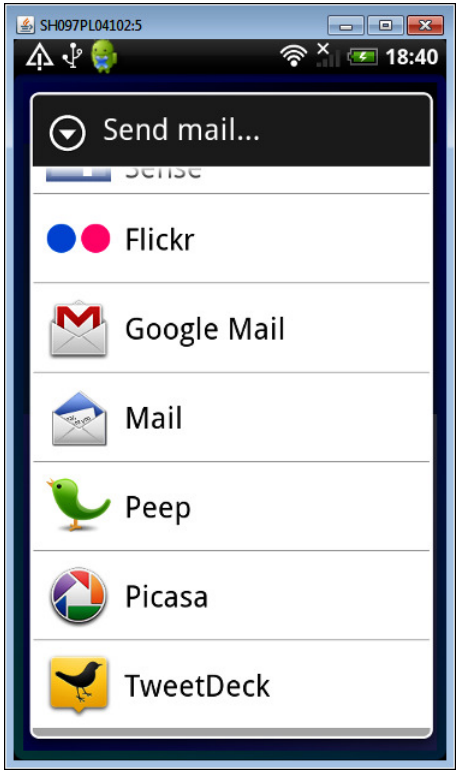



**FIGURE 9-15** Choose the application to send the email.

You may configure the email by adding attachments and recipients. Here's some code to do that:

```
email.setType("image/jpeg");
email.putExtra(android.content.Intent.EXTRA_EMAIL, new String[] {"some@contoso.com" });
email.putExtra(android.content.Intent.EXTRA_SUBJECT, "One for you");
email.putExtra(android.content.Intent.EXTRA_TEXT, "Thought it was a nice pic!");
email.putExtra(Intent.EXTRA_STREAM, Uri.parse("file://" + capturedImage));
```

The *setType* method indicates the type of the attachment. The various calls to the *putExtra* method set specific pieces of information for the intent: email address, subject line, and text. The attachment is appended as a stream, and the reference to the file is obtained via a stream (see Figure 9-16).

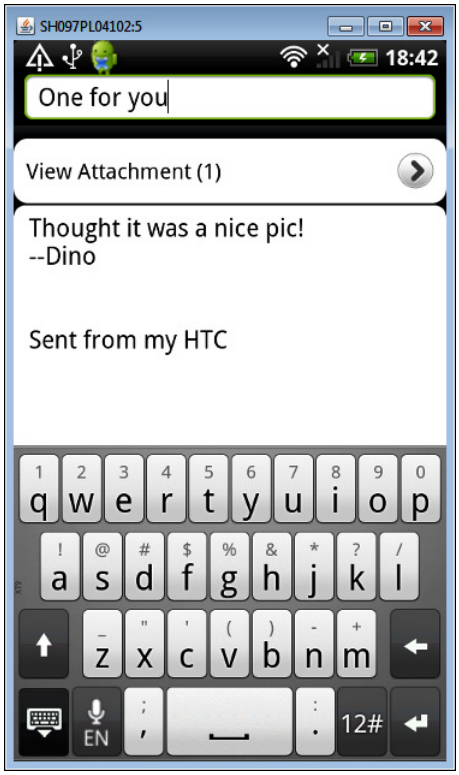



**FIGURE 9-16** Sending an email via code.

Let's find out more about what it takes to test and distribute the application. Curiously, things are somewhat reversed in Android compared to iOS. Testing is much more problematic in Android than in iOS—but distribution is much easier on Android.

## Testing the Application

Testing your Android code is quite a comfortable task due to the rich support for debugging and tooling that you find in both Eclipse and IntelliJ. The Android emulator offers some nice features, such as the ability to set different screen resolutions. It is, however, quite slow, and adding more RAM is the

only thing that seems to make it run a bit faster. One of the reasons for such poor performance is that this is an emulator, not a simulator like the iPhone tool. The Android emulator attempts to emulate the ARM processor, which makes it slower; on iOS, the iPhone tool is a simulator, doesn't attempt to mimic the ARM processor, and is considerably faster.

This is, however, only one side of the coin. You can't say you tested your Android application enough if you haven't tested it on a variety of devices. And that's precisely the sore point.

## Enabling Devices

Unlike iOS, you have no system-enforced restrictions on installed applications in Android. So long as you have an Android executable (an APK file), you can install and run it on any compatible device. It's only the user—not the system—that can prevent this. It may be different on some devices, but in general, any flavor of Android offers a way to accept or deny install of applications coming from unofficial application markets. If you don't have a setting, as in Figure 9-17, there's no chance that your executable can be installed and run.

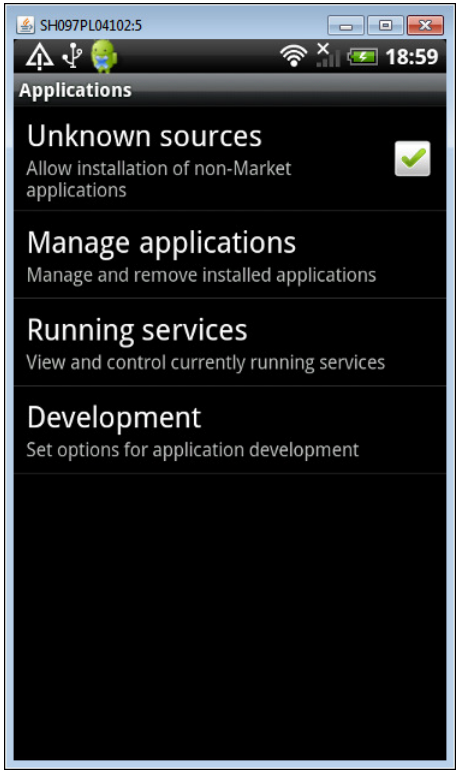



**FIGURE 9-17** Application settings for installing applications from unknown sources.

If this may be an issue for distribution, though, it isn't for testing. As a developer, you likely have a bunch of test devices where you can install just about everything.

### Selecting Test Devices

Testing on devices is far easier in iOS (and Windows Phone) than it is in Android. The root of the problem is what was addressed earlier in the chapter, about the Android jungle.

To test an Android application thoroughly—even an Android application that targets a relatively low API level (Level 8, for example)—you need to have at least three or four devices available. Having only one device that qualifies as a smartphone may not be sufficient; and in addition, you have tablets and a myriad of cheaper, smaller phones to take into account.

The Resource page on the Android website (*http://developer.android.com/resources*) provides a couple of great links with statistics about the most popular versions of the Android OS and most common screen sizes. Based on that, you can form a clearer idea about how to position your application in the Android jungle. At present, an application written for Android 2.1 allows you to reach 93 percent of devices, and one written for Android 2.2 handles more than 80 percent. This says that you probably can focus your testing efforts on devices that support just one operating system. A similar statistics exists for screen sizes. At present, normal size and high or medium dot-per-inch (DPI) values cover more than 80 percent of devices.

Once you have identified the characteristics of abstract devices, how would you get physical devices? It is desirable that you get devices from a variety of vendors, too—HTC, Motorola, or Samsung, for instance.

It's not going to be easy; neither is it cheap, necessarily. Perhaps for this reason, some companies are coming up with testing services on a variety of devices. One example of such a company is Apkudo (*http://www.apkudo.com*). And finally, don't forget that you still can use the Android emulator to test applications on configurations that you can reproduce with real devices. Although the emulator does not allow you to test specific hardware, at least it will give you feedback on how your application may look on different screen sizes and orientations.

**Note** Let me also add that not just any Android applications need the same testing effort. The right strategy depends on the specific application and scenario.

## Distributing the Application

As mentioned, distribution of applications in the Android world is free and not restricted in any way. You can place the executable on a website or distribution list and have users receive and install the application with ease. Google also offers Google Play as a place to showcase your applications where users can search and install the software they like. Google Play is not the only market for Android applications: Amazon Appstore, GetJar, Opera Mobile App Store, and AndAppStore are a few alternatives.

### Google Play

To publish applications to Google Play, you need to register as a Google Play developer and pay a one-time fee of $25. Once registered, you have access to a bunch of tools for monitoring downloads and updating publishing options.

Your application can be published for free or for a fee. In the latter case, you also need to set up a special merchant account. The same is required if your application uses in-app billing.

Note that Google Play allows you to publish multiple executables under the same application name. This addresses the scenario in which your application targets multiple screen sizes and operating system versions. If you upload multiple executables, then Google Play will pick up the most appropriate executable based on the characteristics of the requesting device.

### Signing the Application

You can't publish an application if its executable isn't signed digitally. As a developer, you must obtain a certificate that will be used to identify you as the author of the application.

There's no need to buy a certificate from an authorized certificate authority (CA); a self-issued certificate is acceptable. The Android SDK provides a tool to generate a certificate—the *keytool* application. Once you have the certificate, you need to compile the application for release and sign the resulting executable. There's a command-line tool for signing—the *jarsigner* tool—but often this task is integrated in the IDE of choice. Figure 9-18 shows the dialog box that you get from IntelliJ.

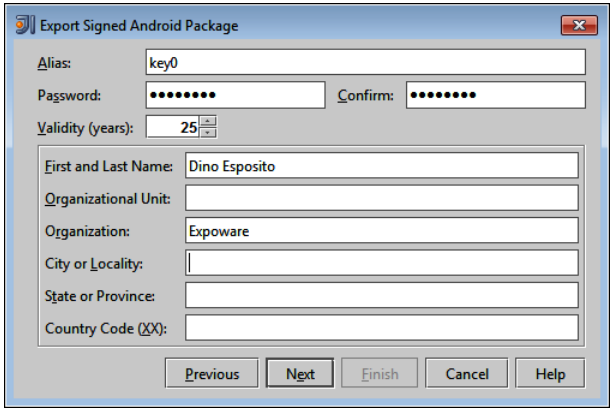



**FIGURE 9-18** The wizard to sign an Android executable in IntelliJ.

## Summary

If you look at most statistics, Android is the most popular platform. But if you look at how most companies address mobile, Android is hardly the most popular platform. Developing Android applications may be very challenging and even expensive due to the wide range of devices and the many slightly and subtly different Android-based firmwares all around.

It all depends on your definition of Android—devices, operating system versions, and screens. The more defined your boundaries, the easier it is to develop for Android. Due to fragmentation, you sometimes end up releasing software, taking the risk that it won't run perfectly on just any device that runs the operating system.

If you focus on software aspects instead, developing for Android is not hard even if you don't have Java skills. Moving to native Android development from .NET, for example, is relatively easy. If you have C++ or object-orientation skills, you can manage to write Android code quite easily.

Using Java and native SDKs is not the only option for Android applications; you can take the C# route with MonoDroid or the hybrid route using PhoneGap, as well as a cross-platform framework such as Titanium or Flash Builder. Overall, however, I consider using Java and native SDKs the primary choice: it's clearly the most compact and performing option, and it is a largely affordable option from a development perspective.

CHAPTER 10

# Developing for Windows Phone

*Genius is 1 percent inspiration and 99 percent perspiration.*

*—Thomas A. Edison*

**In this chapter:**

- Getting Ready for Windows Phone Development
- Programming with the Silverlight Framework
- Deploying Windows Phone Applications
- Summary

Windows Phone, the successor to Windows Mobile, is Microsoft's newest operating system for smartphones. Similar to Apple (and unlike Google), Microsoft exercises strict control over the hardware platform on which Windows Phone applications can run. This makes it easier to write applications because you don't have innumerable resolutions and varying hardware capabilities (such as odd sensors or cards) to deal with.

The Windows Phone application programming interface (API) and development tools are high-quality, and they don't require any significant learning curve—at least, not from developers already familiar with the Microsoft .NET Framework paradigm and related languages, such as C#. For developers approaching Windows Phone with a C++ or Java background, the path is also smoother because of the quantity of relevant tutorials and articles available from both the vendor and from the community.

**Note** Overall, Windows Phone development is no more than a mild challenge for anyone with a working knowledge of an object-oriented language and for everyone who's a quick learner. Noticeably, the list of Windows Phone developers includes quite a few eighth-grade students from around the world. I know this very well, because my son (who is currently 14 years old) is one of them. If you're curious, you can check out a number of applications by typing **expoware.org** on the Windows Phone Marketplace.

If you already have a .NET Framework background, you're almost done. You truly can plan to deliver applications without having to deal with most of the usual mobile gap—learning new languages, becoming familiar with new frameworks, and perhaps dealing with new operating systems, tools, and computers.

Even if you're not coming to Windows Phone with a .NET background, you'll feel at home quite soon anyway.

## Development Tools and Challenges

Getting ready for Windows Phone development requires a single download and a unified setup process. You can get Windows Phone–specific tooling as an extension to a professional version of Microsoft Visual Studio or by acquiring a specific edition of the free Visual Studio Express platform. At the end of the setup program, you're all set; there's no need to tweak system variables or install additional tools. You will have the Visual Studio code editor, a simple embedded graphic designer, a debugger, and the Windows Phone emulator for testing applications. You can also rely on Microsoft Expression Blend (a separate product and download) for an even richer and interactive development and designer experience.

### Becoming a Windows Phone Developer

You can download the tools for free, which means you can practice with the framework at no cost. At this stage, though, you will be allowed to test your applications only via the emulator. To feel the thrill of running your applications on a real device, however, you have to be a registered Windows Phone developer.

You become an official Windows Phone developer by enrolling in the Microsoft developer program for Windows Phone. Having an account will cost you about $99 per year but will enable you to publish applications for free or for a fee on the Microsoft Windows Phone Marketplace at *http://www.windowsphone.com*. Your entry point in the world of Windows Phone development is the App Hub, located at *http://create.msdn.com*.

### The Visual Studio Environment

The primary language for Windows Phone development is C#, but you can use Microsoft Visual Basic as well. You start by creating a new project from the dialog box shown in Figure 10-1.

As you can see from Figure 10-1, a Windows Phone application is essentially a special flavor of Silverlight. Therefore, any Silverlight development skills that you may have enable you to be a highly productive Windows Phone developer right away—or almost, anyhow.

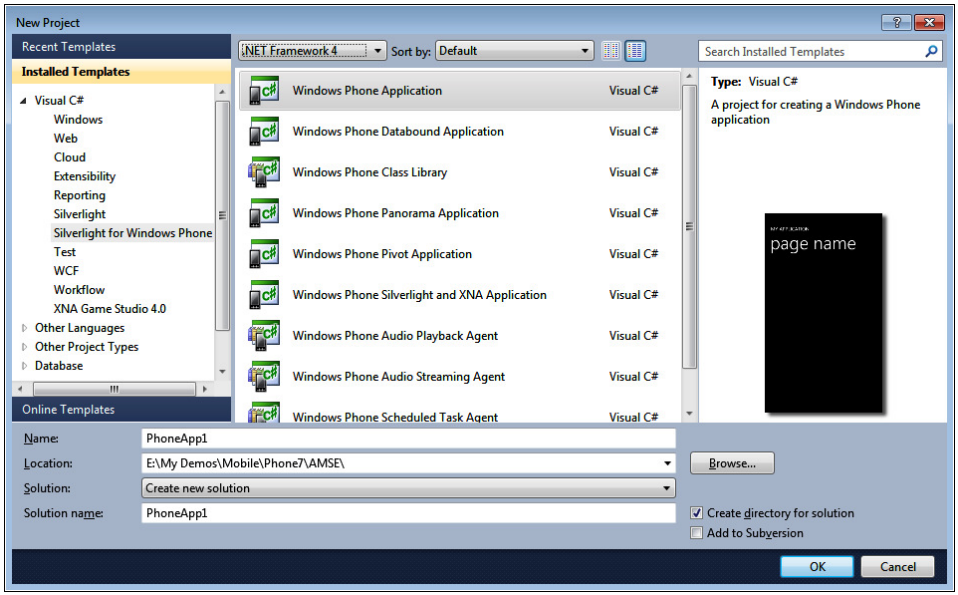



**FIGURE 10-1** The Windows Phone programming environment in Visual Studio 2010.

**Note** If you spent the last five years in a cave, lost in a rainforest, or simply busy with some overwhelming project, Silverlight is a lightweight spin-off of the .NET Framework that originally was created to host binary applications on webpages in a totally safe way. In a sense, Silverlight is the counterpart to Adobe Flash. Silverlight applications can run in a browser plug-in and have access to a subset of the full .NET Framework. That same framework has been reworked to run on a virtual machine hosted in a Windows Phone–equipped device.

## As Easy as Possible, but No Easier

From a merely technical perspective, writing Windows Phone applications is not difficult for the majority of developers. The technical hurdles can be nearly zero if you're already a .NET, Windows Presentation Foundation (WPF), or Silverlight developer, but they're also quite low if you're a Java or C++ developer. The combination of C# as the programming language and tools cuts off a large part of the gap. If you have other types of background experience, the learning curve may be a bit steeper, but typically not what I would see as a *significant* issue.

Instead, I see another potential issue with Windows Phone development: sometimes it can be too easy to use.

Having an easy-to-use platform is not an excuse for writing bad applications that disregard established patterns and principles of software design and that downplay usability and user experience.

**Important** The real challenge for a mobile application is not in getting users to download and install it, but in being used. The true value of a mobile application lies in the idea, implementation, and graphical surroundings. When that experience is poor, the app is a failure. Similarly, an application that's a pleasure to use but is not backed by a strong use-case is destined for oblivion. The same idea—implemented differently—can lead to a radically different experience. I'll return to this point later when discussing the sample application—Guess-the-Number. At present, I've been able to find at least five analogous apps on the Windows Phone Marketplace (and as many on the Google Play Store, formerly known as Android Market). None of these applications are a joy to use (to say the least) and they fail at most of the patterns presented in Chapter 7, "Patterns of Mobile Application Development." A great and easy-to-use platform poses an additional challenge to developers—it's so easy that you may be led to release your creation too early.

## Choosing the Development Strategy

Defining the development strategy to target the Windows Phone platform is fairly easy. I would even say it has only a plan A: Use the C# (or Visual Basic) language and the specific segment of the .NET Framework that is bound to the project.

What about other options?

As you'll see in Chapter 11, "Developing with PhoneGap," you can use the PhoneGap software development kit (SDK) to turn a bunch of static HTML5 and JavaScript pages into a Windows Phone application. Personally, I would consider this option viable only in situations such as when you are writing the same application for multiple platforms at the same time. For example, suppose that your customer needs an application for iOS, Android, Windows Phone, and BlackBerry to be available on a given and fixed date (perhaps a public event such as a conference or a sports tournament). In this scenario, PhoneGap can be a real time-saver. For everything else, writing native Windows Phone applications is usually a fast-enough process that you don't need to look for alternative solutions.

**Important** By "fast enough," I simply mean that you are unlikely to face delays due to the need for becoming familiar with the language or the SDK. Rest assured that a complex application that faces a dozen of use-cases still takes longer than an application with only a couple of screens.

### Silverlight-Based Applications

For Windows Phone development, you can choose between two distinct but not entirely mutually exclusive frameworks: the Silverlight framework and the XNA framework.

A simple but effective argument to differentiate the framework is the following: you use the Silverlight framework for business applications and the XNA framework for game-oriented applications.

Such a rule is not carved in stone, however. You can choose one as your primary framework, yet still be able to import the functionality you need from the other. For example, you can write games using the Silverlight framework but integrate your Silverlight-based application with XNA-only capabilities, such as write access to the media library of the device.

The primary difference between the Silverlight and XNA frameworks is the execution model. A Silverlight-based application runs as most .NET applications do: it is essentially an event-driven application that looks for events exposed by visual controls. In contrast, an XNA-based application is driven by a loop that constantly refreshes the screen, bringing new content into play as determined by other concurrently running application components.

The basic idea underlying a Silverlight application is that you have bunch of pages, each of which represents a screen. A screen is made up of visual controls arranged in a given layout. As the user interacts with these controls, various events are raised and handled in code, starting from the code-behind class of the screen. (You'll see later that the code-behind class is just the starting point for the processing logic of a screen.) The Silverlight framework has a strong API for data binding and user interface management. Support for graphics is good for simple two-dimensional (2-D) shapes, but minimal for three-dimensional (3-D) projections.

## XNA Applications

The Silverlight framework is the natural choice for Windows Phone applications *except* when you're planning to build a game that includes animation and advanced graphics—precisely the areas in which the XNA framework excels. On the other hand, the acronym *XNA* originally stems from the name *Xbox New Architecture* and is the Microsoft set-top box platform for game development. It's worth noting that you can write highly graphical Windows Phone applications using XNA Game Studio 4.0, which was released in the fall of 2010, along with the Windows Phone development tools.

Gaming and advanced 2-D and 3-D graphics represent the major strength of the XNA framework. XNA-based applications don't have anything anywhere near the idea of controls or data-binding facilities. You still can do all of these things, of course, but the implementation costs rest entirely on your shoulders.

## HTML-Based Applications

As mentioned earlier, you also can use the latest version of Adobe's PhoneGap framework to build Windows Phone applications. In the panorama of mobile programming tools, Adobe's PhoneGap is an increasingly popular framework built around a simple but effective idea: turning a bunch of client-side webpages into a native application by embedding web resources [HTML, JavaScript, or Cascading Style Sheets (CSS)] into a rather scanty native application centered on a web view component.

Any logic that you need should be coded in JavaScript and embedded within HTML pages. You can use Ajax to make external calls, and you might want to use the capability of HTML5 to store data locally. PhoneGap provides bridges to some native capabilities (mostly sensors) and is extensible with any piece of native code that you need to populate further the ecosystem of objects that you have access to from within the browser.

The performance of PhoneGap applications is strictly dependent on the capabilities of the browser and its support for HTML5. Using HTML5 is not mandatory, but it is highly recommended due to the access it offers to specific capabilities, such as local storage. You need at least Windows Phone 7.5 to run a PhoneGap-based application.

I'll cover PhoneGap development in the next chapter.

## The Windows Phone—Way Ahead

Windows Phone was released in the fall of 2010 and—I'll say it—it was good enough for a start, but was not a full-fledged development platform. A development platform is much more than a useful editor, compiler, and emulator. It requires a complete API (with background agents, total support for common tasks, feature-complete user interface (UI) components, and sync capabilities) and it also needs an effective infrastructure for beta testing and companywide, private appstores. A capable HTML5 browser would help, too.

In 2011, Windows Phone 7.5 filled some of these holes. In particular, the API becomes richer and the Marketplace Beta made its debut, along with a valid HTML5 browser based on Windows Internet Explorer 9.

**Note** The browser embedded in Windows Phone 7.0 was based on a hybrid engine between Internet Explorer 7 and Internet Explorer 8. According to the capabilities tracked by device description repositories (DDRs), from the perspective of mobile web, Windows Phone 7 is hardly considered a high-end device. (See Chapters 3 and 6 for details on device segmentation and multiserving.) In Project Liike—Microsoft's Patterns & Practices project for mobile web—Windows Phone 7 falls in the medium-end group of devices, whereas Windows Phone 7.5 falls in the high-end group.

The roadmap features another minor release planned for 2012, code-named Tango, and a major release planned for 2013, code-named Apollo. The latter is expected to bring significant improvements in the attempt to reduce the gap with iOS and Android. Expected features include app-to-app communication and integration, an enterprise-level marketplace, expandable storage via SD cards, more control over the camera, deep SkyDrive integration, and native BitLocker encryption.

From a developer's standpoint, the first Windows Phone was mostly about a new user interface and user experience—the popular Metro interface, destined to land on the Windows territory, too, with Windows 8. With Mango, Microsoft just completed the work that a first good release of a mobile platform would have required.

From now on, it's a battle to improve the platform, to make it run on the largest possible number (and types) of devices and conquer the masses.

### The Metro Interface

The focus of the first release of Windows Phone was the user interface and user experience and Metro was its prophet. *Metro* is the blanket term used to indicate a collection of design principles that have inspired Microsoft in the creation of the new interface of Windows Phone and more. In fact, Metro is also behind the new user interface and platform restructuring being operated for Windows 8.

According to Wikipedia, Metro is a "design language" inspired by "design principles of classic Swiss graphics" that emphasize cleanliness and readability. If Metro is mainly a design language, then a Metro application is a Windows (Phone) application whose user interface and user experience are inspired by the Metro principles.

How is this going to change the life of developers? Metro is only the recommended User Interface/User Experience (UI/UX) of Windows Phone and, maybe more important, of the next Windows version.

As far as mobile Windows Phone applications are concerned, you build them using native controls. These controls are styled after Metro, but this is a mere detail for you. If you're creating custom controls, you might want to ensure that they fit well with the Metro principles. If you're creating a mobile site, you might want to make a masterly move and switch to a Metro user interface when the site is viewed on a Windows Phone device.

More than Metro in itself, the recommended guidelines for the platform and common patterns and practices (such as those we discussed in Chapter 7) are relevant to developers.

## Programming with the Silverlight Framework

Let's start with a quick look at the skeleton of a Windows Phone application written using the Silverlight framework. The rest of this chapter will be based on the assumption that a Windows Phone application is always written against the Silverlight framework.

### Anatomy of an Application

Windows Phone 7 is essentially Silverlight 3, whereas Windows Phone 7.5 is a close relative to Silverlight 4.

What do terms like *essentially* and *close relative* actually mean?

The Silverlight framework for Windows Phone is not identical to the Silverlight framework that you use in web applications because there's been some adaptation that made sense for a phone device. Specifically, Windows Phone applications are considered trusted applications and are not sandboxed. Therefore, you can connect to any URL you like, well outside the same-origin policy that you experience within the browser. Similarly, the management of isolated storage is different—there are no limits to the amount of data you can save, so long as there's storage available. At the same time, access to contacts and the user's document is subject to explicit user approval.

Visual Studio offers different templates to start off with a Windows Phone application. It's all about the number of features that should be initially generated in the source code and the overall layout of the user interface. The anatomy of a Windows Phone application is the same, regardless of the layout. Let's start with a plain application project.

## Dissecting the Project

A minimal Windows Phone project consists of just a few files—a manifest file, an application definition file, a main page file, and a handful of graphical resources (see Figure 10-2).

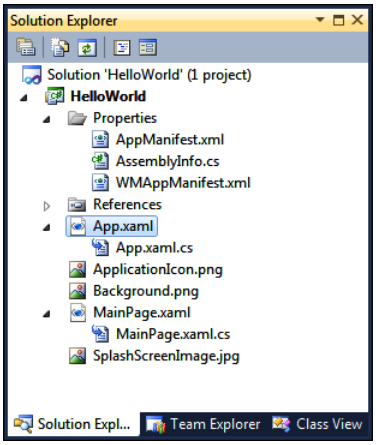



**FIGURE 10-2** A sample Windows Phone project.

As you grow your application, you may feel the need to add more project folders. However, there's no convention-over-configuration (CoC) schema being applied here.

I usually group screens under a Views folder and, if I'm using the Model-View-ViewModel (MVVM) pattern (more on this pattern later), I tend to create a ViewModels folder as well. Likewise, I tend to group auxiliary resources (e.g., images, sounds, and literals) under a single folder. At any rate, there's no reason why my choices are better than yours; we're talking about arbitrary organization of the project files.

**Important** There are some restrictions on the location of files. The splash screen bitmap (SplashScreenImage.jpg), as well as the application icon (ApplicationIcon.png) and tile icon (Background.png), are required to stay in the root of the project. In addition, the splash screen bitmap can't even be renamed (it can be disabled, though, as we'll see later in this chapter). Instead, you can rename application and tile icons easily by editing the manifest file.

## The Manifest File

Public information about a Windows Phone application results from the content of three files in the Properties folder of the project. The AssemblyInfo.cs file is a common file in any .NET project. The role and content of this file are not different in Windows Phone. You use this file to specify copyright and versioning information. In Windows Phone, it is required that you set the neutral language of the application, as in Figure 10-3. Leaving the neutral language field unspecified doesn't affect the functionality of the application, but it may prevent the executable from being uploaded to the marketplace.

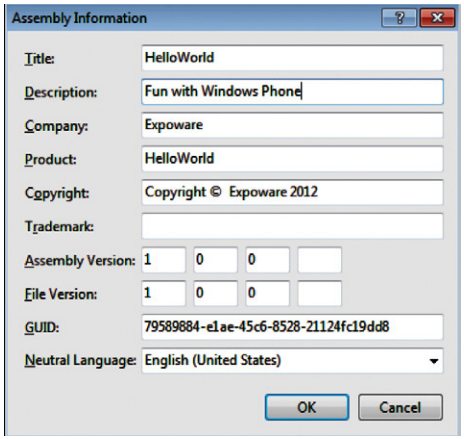



**FIGURE 10-3** Using the project's Property page to set the neutral language.

You can set the neutral language programmatically as well, by adding the following line to the AssemblyInfo.cs file:

```
[assembly: NeutralResourcesLanguageAttribute("en-US")]
```

Note that the Assemblyinfo.cs file can be opened and edited in the code editor as well. The dialog box of Figure 10-3 is just an additional facility. In the Properties folder, you find two similar-looking manifest files, but only one is the real Windows Phone manifest file, analogous to manifest files we've met in iOS and Android development. The AppManifest.xml file exists only for compatibility with the Silverlight framework. It is the same file that you may know from classic Silverlight programming. In Windows Phone, it's essentially an empty file at development time:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Deployment.Parts>
    </Deployment.Parts>
</Deployment>
```

The *Deployment.Parts* element is expected to present the list of binary files (assemblies) that form the Silverlight package. This manifest file is completed at compile time. Here's how it looks like once the HelloWorld executable—a XAP package—is ready:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
            EntryPointAssembly="HelloWorld"
            EntryPointType="HelloWorld.App"
            RuntimeVersion="3.0.40624.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="HelloWorld" Source="HelloWorld.dll" />
  </Deployment.Parts>
</Deployment>
```

The classic mobile application manifest file is WMAppManifest.xml, which is the file that you should edit to tweak information about your application. This is the file that the operating system looks into before it installs or runs the application. You don't usually touch this file unless you want to rename icons or move or rename the main screen page.

Here's what a typical Windows Phone manifest file looks like:

```
<Deployment xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
            AppPlatformVersion="7.0">
  <App xmlns=""
       ProductID="{e0ce571d-cbcc-4639-94b1-a057d0186830}"
       Title="HelloWorld"
       RuntimeType="Silverlight"
       Version="1.0.0.0"
       Genre="apps.normal"
       Author="Expoware" HelloWorld app"
       Publisher="Expoware">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
    <Capabilities>
      ...
    </Capabilities>
    <Tasks>
      <DefaultTask  Name ="_default" NavigationPage="MainPage.xaml"/>
    </Tasks>
    <Tokens>
      <PrimaryToken TokenID="HelloWorldToken" TaskName="_default">
        <TemplateType5>
          <BackgroundImageURI IsRelative="true"
                              IsResource="false">Background.png</BackgroundImageURI>
          <Count>0</Count>
          <Title>HelloWorld</Title>
        </TemplateType5>
      </PrimaryToken>
    </Tokens>
  </App>
</Deployment>
```

In the manifest, you set the public name and icon of the application, as well as providing information about the main screen. In the Tasks section, you can indicate a relative path for the main screen page. You can't indicate a relative path for application icon and tile icon.

One of the main purposes of the manifest file is presenting the operating system with the capabilities that the application needs—network access, identity, media library, sensors, contacts, phone dialer, and so forth. In this regard, Windows Phone as an operating system fully honors capabilities and wouldn't let you use something (for example, the camera) if you don't declare that

you're going to use it. You list capabilities in the Capabilities section of the manifest file, as shown here:

```
<Capabilities>
      <Capability Name="ID_CAP_APPOINTMENTS"/>
      <Capability Name="ID_CAP_CAMERA"/>
      <Capability Name="ID_CAP_CONTACTS"/>
      <Capability Name="ID_CAP_IDENTITY_DEVICE"/>
      <Capability Name="ID_CAP_IDENTITY_USER"/>
      <Capability Name="ID_CAP_LOCATION"/>
      <Capability Name="ID_CAP_MEDIALIB"/>
      <Capability Name="ID_CAP_NETWORKING"/>
      <Capability Name="ID_CAP_PHONEDIALER"/>
      <Capability Name="ID_CAP_PUSH_NOTIFICATION"/>
      <Capability Name="ID_CAP_SENSORS"/>
</Capabilities>
```

It is your responsibility as a savvy developer to declare only the capabilities that you really use. You can use the Marketplace Test Kit or the Windows Phone Capability Detection Tool to detect the application capabilities. These tools are installed with the Windows Phone package.

The most important point, however, is something else. When you submit an application to the marketplace, the capabilities of the application are detected programmatically by inspecting the binary code. The capabilities list that you provided is then overwritten with the actual list of required capabilities. In light of this, you can simply write your code and access any API that you need, blissfully ignoring the issue of capabilities. There are two notable exceptions. If the original list compiled in the executable doesn't include the capability for Internet access (ID_CAP_NETWORKING), that won't be added silently. Second, the capability for the forward-facing camera will not be modified during the submission process: it's maintained if found in the original manifest, but it's not added otherwise.

**Note** The automatic treatment of capabilities is certainly an aspect that speeds up application development a bit. So I think it can be a good thing for most developers. At the same time, it doesn't help building an awareness of capabilities and their purpose. I suggest that you always declare in your manifest file all the capabilities that the application needs to work effectively.

## Application Startup

A Windows Phone project is made of an App.xaml file with the related App.xaml.cs code-behind class. The Extensible Application Markup Language (XAML) file contains references to global resources (i.e., styles, colors, animations, and templates) for the various pages, whereas the C# class gathers handlers for global events such as unhandled exceptions, navigation failures, and life-cycle events. Here's the markup for the App.xaml file:

```
<Application
    x:Class="HelloWorld.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone">

    <Application.Resources>
         <!--Application Resources-->
    </Application.Resources>

    <Application.ApplicationLifetimeObjects>
        <shell:PhoneApplicationService
            Launching="Application_Launching" Closing="Application_Closing"
            Activated="Application_Activated" Deactivated="Application_Deactivated"/>
    </Application.ApplicationLifetimeObjects>
</Application>
```

The App.xaml file indicates the starter object—HelloWorld.App in the example—and provides an implementation for it. This class is responsible for the launch of the application instance:

```
public partial class App : Application
{
    public PhoneApplicationFrame RootFrame { get; private set; }
    public App()
    {
        UnhandledException += Application_UnhandledException;
        InitializeComponent();
        InitializePhoneApplication();
    }

    // Life cycle events
    ...

    // Phone initialization
    private void InitializePhoneApplication()
    {
        if (phoneApplicationInitialized)
            return;
        RootFrame = new PhoneApplicationFrame();
        RootFrame.Navigated += CompleteInitializePhoneApplication;
        RootFrame.NavigationFailed += RootFrame_NavigationFailed;
        phoneApplicationInitialized = true
    }

    private void CompleteInitializePhoneApplication(Object sender, NavigationEventArgs e)
    {
        // Set the root visual so that the application can display its UI
        if (RootVisual != RootFrame)
            RootVisual = RootFrame;

        // Remove this handler since it is no longer needed
        RootFrame.Navigated -= CompleteInitializePhoneApplication;
    }
}
```

The *InitializeComponent* method performs the Silverlight initialization—a Windows Phone application is a Silverlight application anyway—whereas *InitializePhoneApplication* performs additional tasks that are specific to the Windows Phone platform. The initialization of the Windows

Phone application consists of setting up the root frame and replacing the splash screen. Let's see how this takes place in the sample application.

## The Application Frame

One of the first steps in the launch of a Windows Phone application is the creation of a new instance of the *PhoneApplicationFrame* class. This class is the top-level container, and only one instance of *PhoneApplicationFrame* exists for the entire application. The frame class holds all the pages and maintains an internal stack where the visited pages are tracked for the sake of Back button functionality.

The application frame is not made visible as early as it is created. Instead, the root frame is bound to a completion event—the *Navigated* event. In the handler of the *Navigated* event, the root frame object is finally rendered by assigning it to the *RootVisual* property of the *Application* class. The *PhoneApplicationFrame* class reads the Uniform Resource Indicator (URI) of the main application page from the manifest and navigates to it. When navigation is complete—namely, when the main page is fully loaded—the *Navigated* event fires, the *RootVisual* property is set, the splash screen is dismissed, and the application's main page is served to the user.

The pages of a Windows Phone application are required to be instances of a class that inherits from *PhoneApplicationPage*.

## Application Life Cycle

In Windows Phone, only one application at a time can be in the foreground. When the user launches a new application or when an asynchronous event takes place [e.g., an incoming phone call, a Short Message Service (SMS), or just the screen gets engaged], the current application is pushed to the background. What happens at this point depends on the version of the operating system that you're running.

In Windows Phone 7, an application that goes to the background is a dead application. However, the system gives a chance to the current page and the application to copy any relevant state to global repositories. In particular, the page will save to an in-memory dictionary—the State dictionary—whereas the application will save to the isolated storage.

If the user navigates back to the application, all she gets is a brand-new instance of the application. The system, however, passes any saved state to the new instance so that the application can present itself to the user as the continuation of the previous session, rather than as a brand-new session.

An application that knows how to save its state and restore from a passed state intelligently is said to be an application that supports *tombstoning*.

In Windows Phone 7.5, multitasking is supported and the overall behavior can compare to some extent to what we've seen for iOS as well as Android. Note, though, that Android can have real background services, whereas iOS has real background services only for a few scenarios such as voice over IP (VOIP) and global positioning satellite (GPS)–based applications. Beyond that, iOS just gives an application a few minutes to complete a task after it is closed.

Figure 10-4 presents the possible states of an application and transition events.

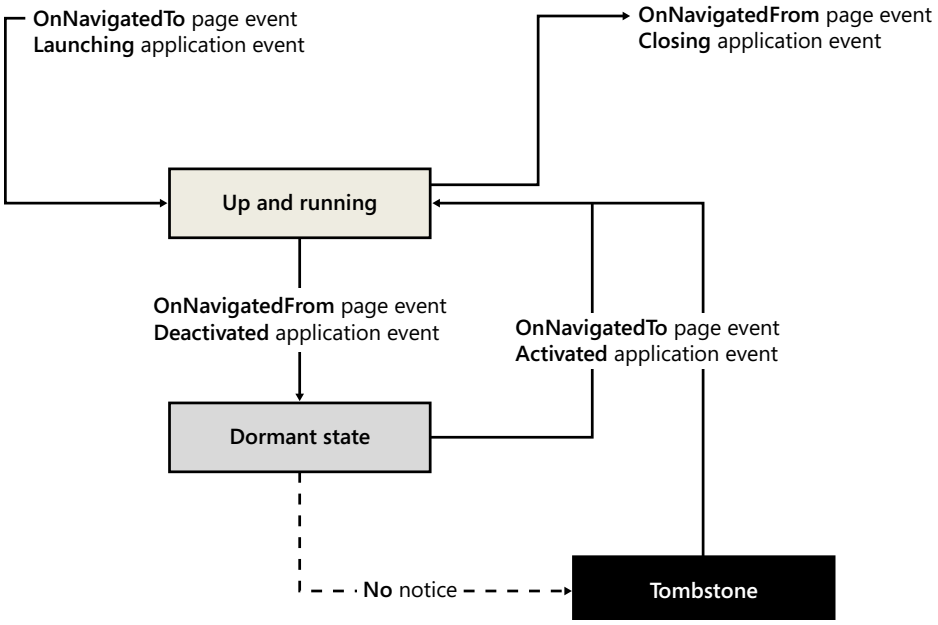



**FIGURE 10-4** The Windows Phone application life cycle.

When an application is first launched, then the *Launching* application event is fired. You can handle this event from within the App.xaml file. You normally don't handle this event. Also, the main page receives the *OnNavigatedTo* event to perform its own initialization. Initialization also can be performed in the class constructor or the *Loaded* event—it is up to you. When the user exits the application (typically hitting the Back button from the home screen or the Home button from anywhere), the application instance receives a *Closing* event and the displayed page receives a *OnNavigatedFrom* event. In the *Closing* event, you might want to save any preferences or data that you like to reuse in the next session.

As the *Memento-Mori* pattern indicates (see Chapter 7 for details on this pattern), applications are actually garbage-collected by the system. So an application that is up and running may be pushed to the background at any time—for example, when the user starts a search or a phone call is received. In this case, the current page receives an *OnNavigatedFrom* event and the global *Deactivated* event is fired. The application might want to save state at this stage because it never knows what will happen to it in the next few moments. Once deactivated, the application is put in a dormant state. The application remains in this state until it is resumed or until free memory is required. When resumed, the application receives an *Activated* global event, where you might want to restore the user interface related to any previously saved data. If the dormant application instance was sacrificed for memory instead, the user receives a brand-new instance, as if she were dealing with tombstoned instances in Windows Phone 7.

Dormant applications are live applications; when they're back to the foreground, there's no need to restore any state. In Figure 10-4, though, you see that the same events are fired when the

application is restored from tombstoning and from the dormant state. In Windows Phone 7.5, a new property has been added to the *ActivatedEventArgs* class that informs you whether you should restore any state or not. The property is a Boolean property named *IsApplicationInstancePreserved*.

## Defining the User Interface

Let's explore in more detail the primary aspects of the Windows Phone user interface. In this section, we'll discuss graphical resources, layout, views, and styles.

### Icons and the Splash Screen

A Windows Phone application is made of a few standard graphic files—the splash screen bitmap, the application icon, and the tile icon. It is required that these files have a specific size and, in some cases, also a specific name.

The application icon is displayed in the launch pad within a square box painted with the currently selected accent color. (In Windows Phone, the accent color is the color used for relevant parts of the user interface, such as the launch menu and tiles.) The expected size of the icon is 62 × 62 pixels.

Tiles are just one of the features that make Windows Phone unique. Tiles are square boxes displayed (and in some cases, animated) in the startup screen of the phone. Tiles are pictures that measure 172 × 172. For a nice effect, you might want to keep tile icons black and white on a transparent background (see Figure 10-5).


**FIGURE 10-5** Tiles and application icons in Windows Phone.

The accent color is used as the background of tiles and icons.

The splash screen is automatic in Windows Phone. So long as you have a graphic file named SplashScreenImage.jpg in the root folder of the project you're done. The graphic file should be 480 × 800. You can't rename or move this file and this file, as well as other application graphic files, should be associated with a Content build action in Visual Studio.

Not all applications need a splash screen. According to Microsoft's guidelines, you need one if your application will take longer than one second to load. In addition, you generally need a splash screen if the user usually has the time to read what's in it. If it appears and disappears too quickly, you probably don't need it.

To disable the splash screen, you just remove the splash screen file from the project. Note, though, that some Windows Phone applications might load quickly the first time but slow down on successive invocations. This is commonly due to extra work done upon loading to process saved data and preferences.

**Note** The built-in splash screen mechanism is not programmable but can be replaced altogether. Reasons to replace it are adding some animation or progress bar or ensuring that it stays up for a minimum time. In both cases, you remove the static bitmap and bring up programmatically a full-screen pop-up that updates its markup step by step and is optionally controlled by a timer. When done, you dismiss the pop-up and display the actual main page.

## Pivot and Panorama Layouts

Windows Phone offers two special types of application-level layouts—pivot and panorama views. A pivot application offers an experience similar to a tabstrip. A pivot is made of a collection of pages loaded individually. The pivot view offers natively the ability to navigate horizontally among the pages using touch gestures.

Panorama applications are made of a single page that contains multiple panes. Also, in this case, you get free horizontal scrolling and a nice rendering that displays in the current view a small portion of the next pane as a hint to the user that more is available.

Pivot and panorama layouts are just one possibility, and their adoption should be the result of a user experience analysis rather than a random choice. Pivot and panorama layouts should be considered mutually exclusive. In general, panorama layouts are more expensive in terms of memory as the content of all panes is kept in memory at any time. For this reason, you should not have more than three or four panes. With a pivot view, you don't have the same pressure on memory, but usability suggests that you keep the number of pages under control. Figure 10-6 shows an example of a pivot and panorama layout.


**FIGURE 10-6** Pivot and panorama views.

## Defining a Custom Layout

The visual interface of Windows Phone pages is expressed using XAML as in Silverlight applications. XAML describes the application user interface in much the same way HTML describes the structure of a document.

A XAML user interface is rooted in a container element that lays out its child elements by stacking them horizontally (or vertically), using a row/column grid or direct *x,y* coordinates. Elements are identified by name using the *x:Name* attribute. Table 10-1 lists the container elements that you can use in XAML.

**TABLE 10-1** Container Elements in XAML

| Container Element | Description |
|---|---|
| *Border* | Incorporates a single child element and renders a border all around. |
| *Canvas* | Contains any number of child elements located at specific relative coordinates. The order of child elements is unimportant. |

| Container Element | Description |
|---|---|
| *Grid* | Lays out elements in a matrix of any number of rows and columns. Each row/column can be sized properly. Child elements must be assigned explicitly to a particular cell. The grid is a pure layout element and doesn't allow to style cells (i.e., no way to paint the background or a border). |
| *StackPanel* | Stacks any number of child elements either vertically or horizontally. Elements are rendered as they appear in the markup. You can use margins to control relative distances. |
| *UserControl* | Incorporates a single child element and wraps it up as a new custom element to be reused in other containers. |

*Pivot* and *Panorama* are special controls that act as container elements. Here's how to define a pivot with two pages:

```
<controls:Pivot x:Name="guessPivot" Title="Guess" Foreground="#fff">
     <controls:PivotItem Header="Game">
         <Home:HomeScreen x:Name="homeScreen" />
     </controls:PivotItem>
     <controls:PivotItem Header="Scores">
         <Scores:ScoreScreen x:Name="scoreScreen" />
     </controls:PivotItem>
</controls:Pivot>
```

It should be noted that pivot items are not real pages that you reference by URI. Instead, they are chunks of XAML that you either define inline or reference through a user control, as in the preceding code snippet. A user control is saved as a XAML file, but it has references via the name of its code-behind class.

The following code snippet shows the incipit of a Windows Phone page. Namespaces in root elements are important information for the parser to understand tag names:

```
<phone:PhoneApplicationPage
    x:Class="Guess.Views.GuessMainPage"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:controls="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
    xmlns:Home="clr-namespace:Guess.Views.Home"
    xmlns:Scores="clr-namespace:Guess.Views.Scores">
   ...
</phone:PhoneApplicationPage>
```

The *Class* attribute indicates the name of the class that backs up the page—the code-behind class. Namespace declarations are bound to namespaces in the current assembly or in the specified assembly. For example, the *Home:HomeScreen* element is resolved looking for a class named *Guess.Views.Home.HomeScreen*.

## Style and Designer Tools

Each XAML element sports a long list of graphical and behavioral attributes, including templates and animations. Although you could set these properties inline in the XAML, doing so would result in code that is hard for humans to read.

To define the user interface of Windows Phone applications, you can use Expression Blend, a tool that comes with the Windows Phone 7 SDK (see Figure 10-7).

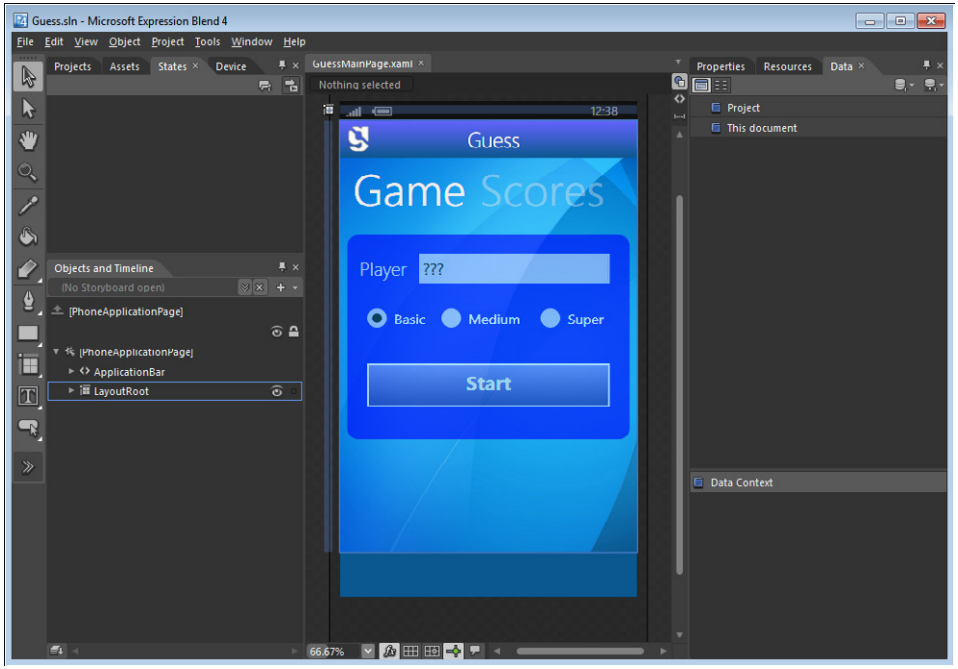



**FIGURE 10-7** Expression Blend in action on the sample application.

Nicely enough, Expression Blend can open and manipulate the same project files that you use in Visual Studio. In Expression Blend, you drag elements to a drawing surface, set properties and attributes, and have the tool generate XAML for you. Switching from a rich and intuitive design tool to Visual Studio is quick and effective. With Expression Blend, you hardly feel the need to tweak the XAML manually. Because you manage XAML with a smart tool, you may not care about readability of the XAML.

Whether you use Expression Blend or not, however, you should plan to manipulate graphical properties more effectively via styles. Here's a sample style that gives a gradient blue background to buttons:

```
<Style x:Key="guessButtonStyle" TargetType="Button">
      <Setter Property="FontSize" Value="32" />
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="Foreground" Value="#ededed" />
      <Setter Property="Background" Value="{StaticResource buttonBkgndBrush}" />
</Style>
```

In particular, any button styled with the *guessButtonStyle* properties will have been given foreground color and font settings:

```
<Button x:Name="startButton" Style="{StaticResource guessButtonStyle}" />
```

What about the background? The value of the *Background* property is defined through a static resource. A static resource is a XAML resource (brush, color, or pen) that is global to the application or to a given container (e.g., a user control). You define static resources once and reuse them frequently; static resources are ultimately a performance trick. Here's a gradient background brush:

```
<LinearGradientBrush x:Key="buttonBkgndBrush" EndPoint="0,1">
    <GradientStop Color="#7777fd" Offset="0" />
    <GradientStop Color="#2222aa" Offset="1" />
</LinearGradientBrush>
```

To make a resource static, you just define it in the Resources section of a container element. Here's an example for application-wide resources:

```
<!--  This code belongs to app.xaml -->
<Application.Resources>
    <guessFx:LocalizedStrings x:Key="LocalizedStrings" />

    <!--Colors-->
    <Color x:Key="appbarBackground">#0b5890</Color>
    <Color x:Key="appbarForeground">#ffffff</Color>
    ...

    <!--Brushes-->
    <SolidColorBrush x:Key="alertBrush" Color="#ff9933" />
    ...

    <!--Animations-->
    ...

    <!--Styles-->
    ...
</ Application.Resources>
```

For performance reasons, it is also advisable that you avoid using colors as constants throughout the XAML files and code. In XAML, to paint the background of any elements, you need a brush, not a color. Properties that let you set the background through a color string are actually doing the work of creating a brush dynamically for you. As you can see, there's no reuse and a white brush (for instance) is created several times in an application instance. With a global white brush (or even color definition), you create static references that can be invoked from anywhere in the application. Also, literals can be attached as resources to XAML attributes. The primary goal for using string resources benefit, however, is not performance but ease of localization. (We'll return to localization in a moment.)

## Dark and Light Themes

Windows Phone supports two predefined themes—dark and light—and leaves the user free of choosing one. The dark theme is the default, but the user can change it via the Settings page, as in Figure 10-8.

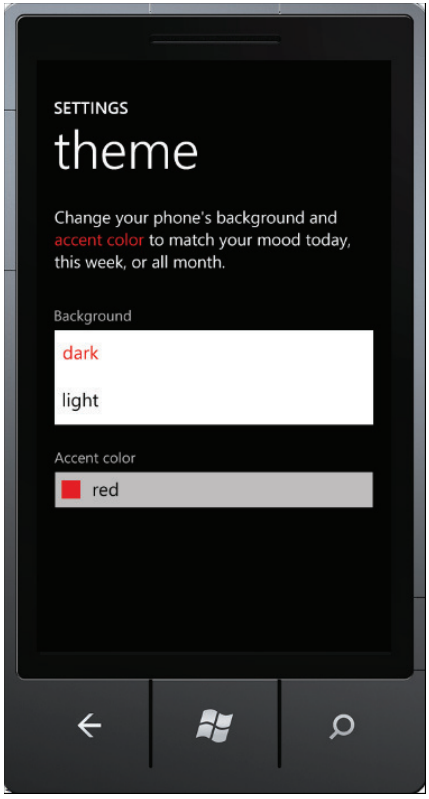



**FIGURE 10-8** Choosing the theme for the device.

More important, applications are not allowed to force the phone to a particular theme. Applications, instead, should be smart enough to adapt to the current theme (and accent color, if needed).

It is required that applications display well and clearly in both themes. An application that presents a confusing user interface under either theme will not be accepted in the marketplace.

Text boxes and input controls are the most problematic aspects of supporting both themes. Depending on the graphics of the application, you may not be able to work out a unique user interface that works well in both scenarios. In this case, you might find it easier to detect the current theme and adjust your visual settings programmatically. Here's some sample code:

```
var light = (Visibility) Application.Current.Resources["PhoneLightThemeVisibility"];
if (light == Visibility.Visible)
{
    // Light theme
    ...
}
else
{
    // Dark theme
    ...
}
```

As you can see, the current theme is exposed to developers as an application-wide resource. Starting from here, you can arrange your own helper methods to expose dark or light theme as simple Boolean properties.

## The Application Bar

Windows Phone devices have no menu button, but the API offers the *application bar* component as a way to provide the application with an easy-to-access toolbar for common tasks. You can place up to four buttons in the application bar, each with an icon and a text hint. In addition, you can add an application bar menu where you convey additional items, as well as those that you can't render easily with a small icon.

The application can be styled to a good extent, but it is not the same as other Windows Phone controls. You can set foreground and background colors, as well as a level of opacity. You can't use gradients and, maybe more important, you can't use data binding to populate the bar.

This means that to make text and icons in the application bar localizable and context-sensitive, you should resort to some C# code. You usually define the application bar in the XAML for a Windows Phone page, as shown here:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True"
                          Opacity="0.5"
                          ForegroundColor="{StaticResource appbarForeground}"
                          BackgroundColor="{StaticResource appbarBackground}"
                          IsMenuEnabled="False">
        <!-- Buttons should be listed here -->
        ...
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

If the list of buttons doesn't change with the state of the application and is not subject to localization issues, then you can just add buttons in the *ApplicationBar* element:

```
<shell:ApplicationBarIconButton IconUri="/Images/appbar_hint.png" Text="Hint"/>
```

More often than not, though, you might want to create your own application-specific helpers to assign each XAML page its own application bar. In doing so, you set button hints to localized text:

```
public static void SetupForOngoingGameScreen(GameScreen page)
{
    page.ApplicationBar.IsVisible = true;
    page.ApplicationBar.Buttons.Clear();
    page.ApplicationBar.MenuItems.Clear();

    // Add buttons
    AppendButtonInternal(page, "/images/appbar/appbar_hint.png", Literals.AppBarHint, page.
Hint);
    AppendButtonInternal(page, "/images/appbar/appbar_quit.png", Literals.AppBarQuit, page.
Quit);  }
private void AppendButtonInternal(PhoneApplicationPage page,
                                  String iconFile, String caption, EventHandler handler = null)
```

```
{
    // Define the button
    var button = new ApplicationBarIconButton()
    {
        IconUri = new Uri(iconFile, UriKind.Relative),
        Text = caption
    };

    // Add to the application bar
    page.ApplicationBar.Buttons.Add(button);

    // Attach click handler
    if (handler != null)
        button.Click += handler;
}
```

The application bar in Figure 10-9 has two buttons, one with an icon and one with text, and a level of opacity of 0.5.


**FIGURE 10-9** The application bar in action.

As mentioned, you can't have more than four icons on the bar. For everything else, you can add an embedded menu:

```
<shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
     <shell:ApplicationBarIconButton IconUri="/Images/appbar_button1.png" Text="Button 1"/>
     <shell:ApplicationBarIconButton IconUri="/Images/appbar_button2.png" Text="Button 2"/>
     <shell:ApplicationBar.MenuItems>
         <shell:ApplicationBarMenuItem Text="MenuItem 1"/>
         <shell:ApplicationBarMenuItem Text="MenuItem 2"/>
     </shell:ApplicationBar.MenuItems>
</shell:ApplicationBar>
```

The menu shows up when the user taps the dots item (as in Figure 10-9) or flicks up the application bar area. The application bar is subject to some built-in animation and graphical adjustments. You might want to choose colors and icons carefully and test the application with both dark and light theme to ensure that no poor effect comes up that may preclude approval of the application. For details about icons, refer to the "User Experience Design Guidelines for Windows Phone" paper available at *http://goo.gl/h7vCo*.

**Important** The application bar should be considered a native part of the Windows Phone platform; this means that if you need some global system menu in your application, the application bar is the recommended way to achieve it.

## Localization of Text

In Windows localization of text, the use of localization in the user interface is fairly straightforward. Localizing everything else, instead, is mostly up to you. The good news, however, is that the need to localize more than text is not usually urgent.

The procedure to localize text is the same as in any .NET application: you add a resource file to the project and identify any literal with a unique key. Visual Studio will generate a class from the resource file automatically so that you can reference literals as constants. Having a file named Literals.resx, as in Figure 10-10, with the access modifier flag set to Public (watch out, the default is Internal), enables you to refer strings in C# code using the following straightforward syntax:

```
AppendButtonInternal(page, "/images/appbar/appbar_about.png", Literals.AppBarAbout, page.About);
```

The expression *Literals.XXX* returns the string associated with the *XXX* name in the Literals.resx file. To add resources for another language, you simply add a new .resx file following a special naming convention: Literals.*xx-yy*.resx, where *xx-yy* represents the identifier of the culture. To add Italian text, for example, you create Literals.it-IT.resx. Note that the .resx file without culture information in the name is associated with the neutral language set for the application.

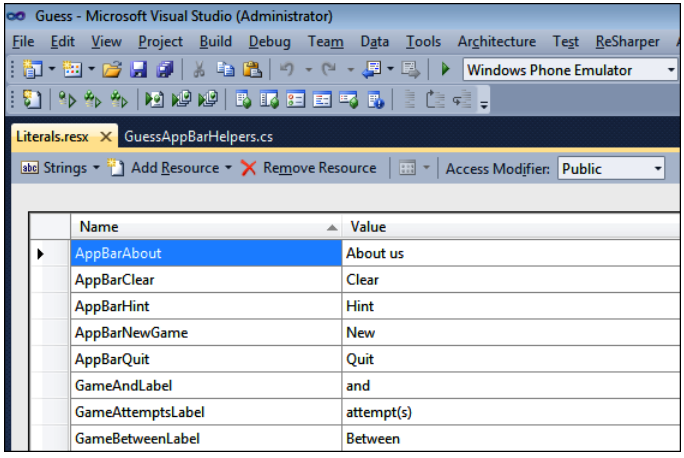



**FIGURE 10-10** The resource editor in Visual Studio.

To reference the same literals from within XAML markup, some extra work is required. In particular, you need a helper class to expose resources to XAML elements. Here's an example that you can import verbatim in your applications:

```
public class LocalizedStrings
{
   private static readonly Literals Resources = new Literals();
   public Literals Strings
   {
      get { return Resources; }
   }
}
```

The name *LocalizedStrings* is arbitrary; the *Literals* class, instead, is bound to the RESX resource file you created in the project. All that you do is exposing, through a public read-only property, an instance of the auto-generated resource class. The next step is linking the helper class to the application resources so that it becomes visible to the XAML parser. You create a global application resource in App.xaml, as shown here:

```
<Application
    x:Class="Guess.App"
    ...
    xmlns:guessFx="clr-namespace:Guess.Utils.Resources">
    <Application.Resources>
        <guessFx:LocalizedStrings x:Key="MyStrings" />
        ...
    </Application.Resources>
    ...
</Application>
```

You can give any arbitrary name to the *x:Key* attribute, but need to correctly reference the namespace where the *LocalizedStrings* class is located. Finally, you bind resource literals to XAML elements as static resources. The following example shows how to display a localizable label:

```
<TextBlock
      Text="{Binding Path=Strings.HomePlayerLabel, Source={StaticResource LocalizedStrings}}" />
```

For most .NET applications, this work would suffice, but in Windows Phone, you need to take one more step. In particular, you need to explicitly indicate which languages are supported—having culture-specific resource files is necessary but not sufficient.

The annoying thing is that you have no dedicated user interface to do that. So you must open the *.csproj* file in a text editor and manually tweak the content of the <*SupportedCultures*> element (see Figure 10-11). The CSPROJ file is an XML file that contains information about the project. By default, the file contains the following markup:

```
<Project ToolsVersion="4.0" DefaultTargets="Build"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
     ...
     <SupportedCultures>
     </SupportedCultures>
```

```
  </PropertyGroup>
</Project>
```

To add cultures (beyond the neutral language), you insert a semicolon-separated list of culture identifiers, as shown here:

```
<SupportedCultures>it-IT;es-ES</SupportedCultures>
```

Once you save everything and deploy, the system finally will be able to pick up resources for the current culture intelligently. And, nicely enough, the application will change language as the user changes the preferred language in the phone settings.

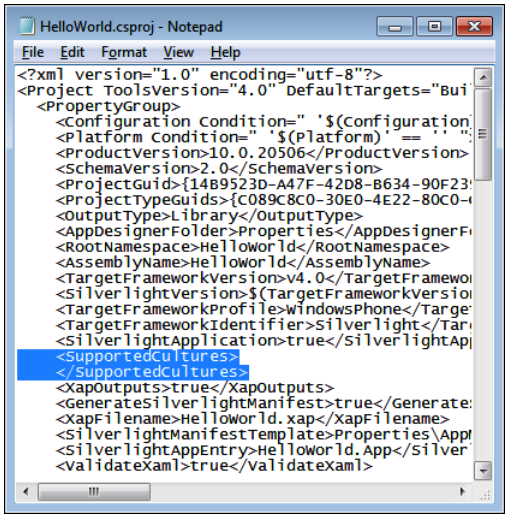



**FIGURE 10-11** Editing the CSPROJ file to add supported cultures.

## The MVVM Pattern

In the 1980s, the introduction of the Model-View-Controller (MVC) pattern for the design of applications was a milestone. The primary goal of MVC is splitting the application in distinct pieces—the model, the view, and the controller. The model refers to the data being worked on in the view and, to a large extent, the state of the application. The view refers to the generation of any graphical elements displayed to the user and captures and handles any user gestures. The controller maps user gestures to actions on the model and selects the next view. These three actors are often referred to as the *MVC triad*.

Over the years, the MVC pattern evolved into a slightly different pattern such as the Model-View-Presenter (MVP) pattern. In MVP, the basic idea is the same, but the model is exposed out of the view through an interface that the presenter commands. MVP today offers a powerful combination of separation of concerns, code cleanliness, and readability.

The MVP pattern is quite generic and can be adapted to nearly any applications—web, desktop, mobile—on a variety of software platforms. In the Microsoft ecosystem, the advent of the XAML

technology replaced the classic MVP pattern with an idiomatic version that uses some capabilities of the XAML technology. This pattern is known as *Model-View-ViewModel,* or MVVM for short.

## Generalities of the Pattern

To better understand the mechanics of the MVVM pattern, we should start from its ancestor, the MVP pattern. The two patterns share the same underlying idea—MVVM is much more technology-oriented, however, and therefore can be referenced as an idiomatic version of MVP. Figure 10-12 presents the diagram of the MVP pattern.

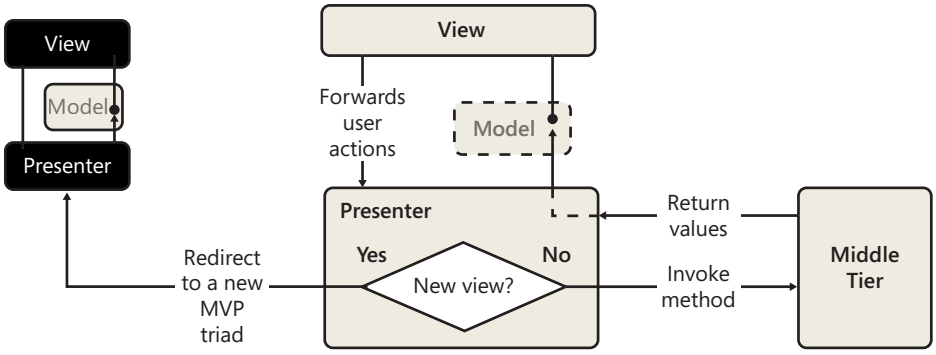



**FIGURE 10-12** The MVP pattern.

According to the pattern, each screen is articulated in two main elements. The view contains interface elements and handles gestures. The presenter performs actions to serve requests captured by the view. The presenter may invoke back-end services to produce expected responses.

Where's the model?

In the context of the pattern, the model is defined as the data being worked on in the view. In this regard, it has very little to do with the data and domain model of the application. In the context of the pattern, the model is populated by the presenter and is passed to the view for rendering.

The difference between classic MVP and idiomatic MVVM is all in how the model is implemented. In classic MVP, for each view, you abstract the model to an interface and implement that interface in the view. As in Figure 10-12, the presenter holds a reference to the view and uses the exposed interface to pass data to display. The model is not implemented explicitly anywhere, but it is defined implicitly in the interface shared between the view and the presenter.

More specifically, the presenter computes data interacting with the back end and is then responsible for binding that data to the view. Data binding occurs under the control of the presenter and must be coded explicitly by the developer.

In MVVM, you use the XAML data-binding infrastructure for display. Subsequently, the model exists as a distinct entity to be linked to the view via XAML elements. Figure 10-13 shows the MVVM diagram for Windows Phone and other XAML-based applications (e.g., WPF and Silverlight). The presenter becomes the View-Model. The presenter exposes public properties for the XAML-based

view to bind to. The presenter receives commands from the view and performs actions against the business logic layer of the application.

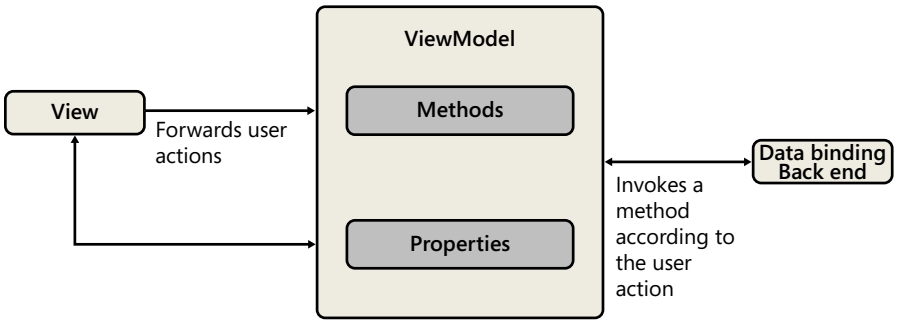



**FIGURE 10-13** The MVMM pattern.

**Important** Personally, I consider MVVM a misleading name. *MVVM* is a name that some team at Microsoft started using at some point to reinforce a design approach that was already known as Presentation Model. Presentation Model and MVVM specialize the MVP pattern by merging Model and Presenter so that a direct binding connection can exist between the view and the presentation model. I consider MVVM a misleading name because it contains the word *model* twice. You don't have two models—you have just one model that describes the data being bound to the view. This model is incorporated in a component that also exposes actions for the view to invoke.

## Design of the View-Model Class

MVVM is considered a best programming practice for XAML-based applications. It is not a must, however. In general, a pattern does not add value per se; the value of using patterns is in what they may produce: increased readability, testability, and extensibility (in a word, quality) of the code. In Windows Phone, you can write great code with or without MVVM; but if you follow the MVVM approach, you have a sure step to great code.

To implement MVVM in a Windows Phone, you have just three basic rules to follow. First, associate each screen (be it a page, a user control, or a pop-up) with an ad hoc presenter class; second, make the presenter class available to the XAML elements. Third, use the XAML data-binding syntax to link portions of the user interface with public properties on the presenter class. And, by the way, once you're there, stop using the word *presenter* and replace it with *view-model*. Figure 10-14 describes the architecture of MVVM in the context of a XAML-based project.

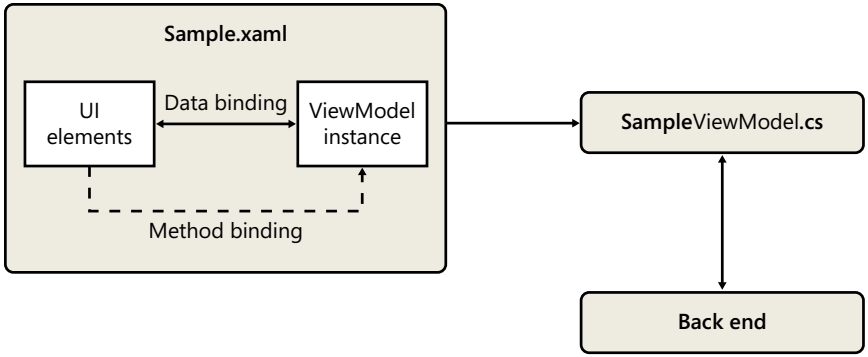



**FIGURE 10-14** MVVM in a XAML-based project.

The *view-model* class has methods and properties that serve the need of the view.

Let's consider a view like that of Figure 10-15, where the user has to enter the name of the player and the level of the game. Finally, the user can click a button to start the game. (This is the home screen of the sample application that will be explored more in depth in a moment.)

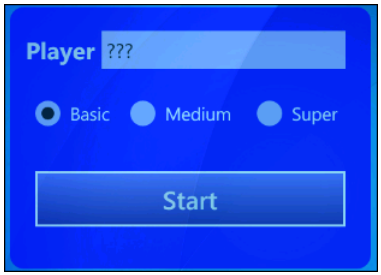



**FIGURE 10-15** A sample view.

The *view-model* class for this view may be as follows:

```
public class HomeViewModel : INotifyPropertyChanged
{
    public String PlayerName { get; set; }
    public GameLevel Level { get; set; }
    public Boolean IsBasic
    {
       get { return Level == GameLevel.Basic; }
       set {
             Level = GameLevel.Basic;
             OnPropertyChanged("IsBasic");
       }
    }
    public Boolean IsMedium
    {
       get { return Level == GameLevel.Medium; }
       set {
             Level = GameLevel.Medium;
             OnPropertyChanged("IsMedium");
```

```
        }
    }
    public Boolean IsSuper
    {
        get { return Level == GameLevel.Super; }
        set {
              Level = GameLevel.Super;
              OnPropertyChanged("IsSuper");
        }
    }

    public void Start()
    {
        if (String.IsNullOrEmpty(PlayerName))
        {
            MessageBox.Show(Literals.GlobalMissingPlayerName);
            return;
        }

        App.MainPage.NavigationService.Navigate(
             new Uri("/views/game/gamescreen.xaml", UriKind.Relative));
    }

    // More code
    ...
}
```

The class has a property—*PlayerName*—for the name of the player, and a property—*Level*—that indicates the level of game. The player name is bound to the text box in the user interface; what about the radio buttons used to set the level? How would you bind them?

A possibility is adding view-specific properties such as *IsBasic*. The property returns a Boolean value when read: *true* if the current level is *Basic; false* otherwise. When set, the property would just set *Level* to the specific value the radio button represents.

The *Start* method instead triggers the behavior expected for when the user taps the Start button—in this case, navigating to the actual gaming screen.

## The Data-Binding Engine

The second step of implementing MVVM requires that we make the *view-model* class available to the XAML elements. This can be done either declaratively by adding a new resource to the XAML file or programmatically from within the code-behind class of the view. Here's the code to do that:

```
public partial class HomeScreen : UserControl
{
    private readonly HomeViewModel _viewModel;
    public HomeScreen()
    {
        InitializeComponent();
        startButton.Click += startButton_Click;
        _viewModel = new HomeViewModel();
        DataContext = _viewModel;
    }
```

```
    void startButton_Click(Object sender, RoutedEventArgs e)
    {
        _viewModel.Start();
    }
}
```

Any XAML element has a *DataContext* property that represents the source of data-binding expressions used within the boundaries of the element. In this case, we're defining the view-model class as the source of data-binding expressions used in the XAML of the entire screen. Furthermore, the *Start* method of the *view-model* class is bound to the Start button in the view.

The third step of MVVM is using the XAML-binding syntax to connect visual elements and public properties of the view-model. Here's how to set the player name and game level:

```
<TextBox x:Name="playerName"
                Text="{Binding Path=PlayerName, Mode=TwoWay}"
                InputScope="PersonalSurname" />
<RadioButton x:Name="levelBasic" GroupName="Level"
             Content="{Binding Path=Strings.HomeLevelBasic,
                               Source={StaticResource LocalizedStrings}}"
             IsChecked="{Binding Path=IsBasic, Mode=TwoWay}" />
...
```

The *Binding* keyword sets a binding between the visual element and the value of the bound expression. If the two-way mode is used, then changes entered by the user to the user interface are automatically propagated back to the linked view-model property. For the bidirectional binding to work, the *view-model* class also needs to implement the *INotifyPropertyChanged* interface. In this way, the view-model can notify binding clients that a property value has changed. Note the following code:

```
public class HomeViewModel : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(String property)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(property));
        }
    }
}
```

You can download the companion code for this chapter for a full implementation of the MVVM pattern in the context of a full application.

## Examining a Sample Application

Let's look at what it takes to build the same application from Chapter 8, "Developing for iOS," and Chapter 9, "Developing for Android," for Windows Phone. As you may recall, Guess implements the old game of guessing a secret number. It supports various levels, each of which corresponds to a

larger interval for the number to guess. Best scores achieved with the phone are then stored and displayed in a separate screen. Figure 10-16 shows the main screen and the game screen of the application.

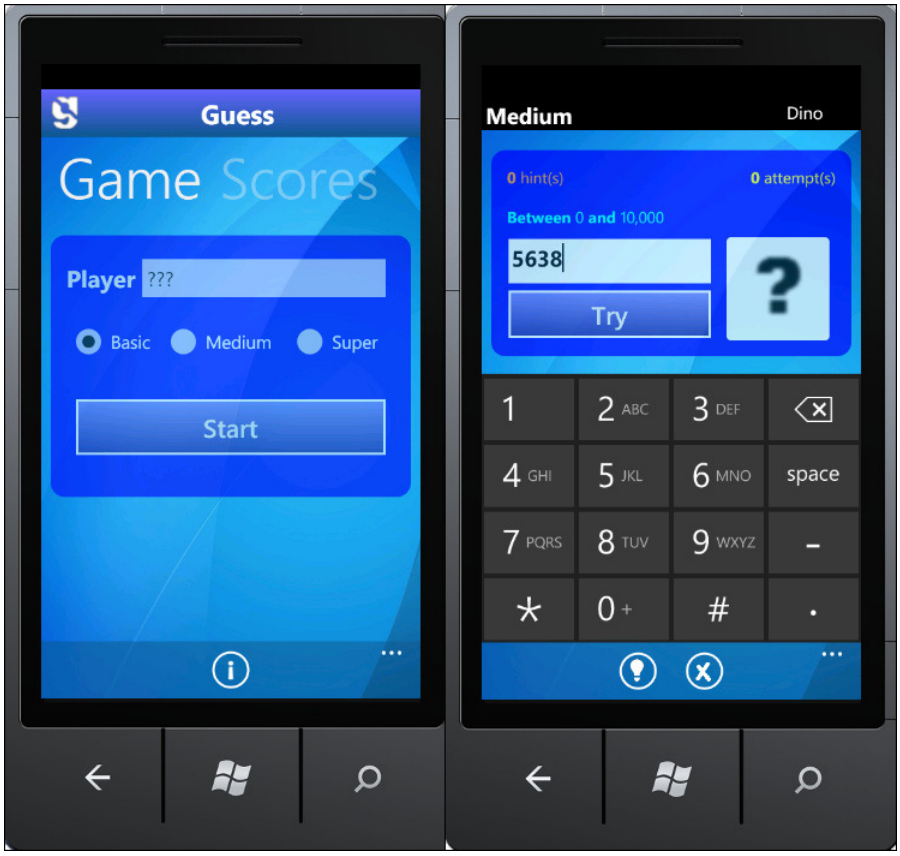



**FIGURE 10-16** The Guess application in action.

As you can see, Guess for Windows Phone is a pivot application. It consists of a Home view, where you enter the player's name, and a Play view, where you enter a number and get a response. A Scores view and an About view complete the application.

## The Home View

Guess is a pivot application, but it uses a custom template for the pivot. The custom template just adds a personalized header with a gradient and a small icon. You define a custom style for the pivot in App.xaml, as shown here:

```
<Style x:Key="guessPivot" TargetType="controls:Pivot">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="controls:Pivot">
                <Grid HorizontalAlignment="{TemplateBinding HorizontalAlignment}"
```

```
                    VerticalAlignment="{TemplateBinding VerticalAlignment}">
                    <Grid.RowDefinitions>
                        <RowDefinition Height="Auto"/>
                        <RowDefinition Height="Auto"/>
                        <RowDefinition Height="*"/>
                    </Grid.RowDefinitions>
                    <Grid Grid.Row="0" Background="{StaticResource captionBrush}">
                       <!-- New caption bar -->
                    <Grid>
                    <controlsPrimitives:PivotHeadersControl x:Name="HeadersListElement"
                                        Grid.Row="1"/>
                    <ItemsPresenter x:Name="PivotItemPresenter"
                                    Margin="{TemplateBinding Padding}"
                                    Grid.Row="2"/>
            </ControlTemplate>
        </Setter.Value>
</Style>
```

By default, the pivot is made of three rows—the title, the header with scrollable items, and the content. All we did was replace the topmost area to make it bigger and styled differently.

Each pivot child view is represented as a user control. The pivot is composed in the main application page. In the constructor of the main application page, you also register a handler for the selection-change event of the pivot, as follows:

```
private void Pivot_SelectionChanged(Object sender, SelectionChangedEventArgs e)
{
   var pivot = sender as Pivot;
   if (pivot == null)
       return;

   var index = pivot.SelectedIndex;
   if (index >= 0)
   {
      // Configure the application bar for the pivot item
      GuessAppBarHelpers.SetupFor(this, index);
   }
}
```

The handler is mainly responsible for adjusting the application bar when a new pivot item is selected. In the main page, you also implement the Back-and-Save pattern (see Chapter 7). The *OnBackKeyPress* event fires whenever the user taps the Back button while on the home page. Note that the home page is the current page when any of the pivot items is selected. A handy feature could be navigating back to the first tab when Back is hit, and exiting if Back is hit from the first tab. In any case, you might want to save any relevant state when the Back button is hit. Look at this code:

```
protected override void OnBackKeyPress(CancelEventArgs e)
{
    // Save state
    App.StateManager.Save();

    // Are we on the root pivot item?
    var pivotIndex = guessPivot.SelectedIndex;
    if (pivotIndex > 0)
```

```
    {
        GoToHomeScreen();
        e.Cancel = true;
        return;
    }

    base.OnBackKeyPress(e);
}
```

Unlike in iOS and Android, in Windows Phone you can access visual elements programmatically, by name, directly from the code-behind class. Not that the work required to set up references in other mobile platforms is not necessary in Windows Phone; more simply, this work is done automatically by Visual Studio. Here's the code that checks whether a non-empty player name has been entered:

```
public void Start()
{
    if (String.IsNullOrEmpty(PlayerName))
    {
        MessageBox.Show(Literals.GlobalMissingPlayerName);
        return;
    }
    App.MainPage.NavigationService.Navigate(
           new Uri("/views/game/gamescreen.xaml", UriKind.Relative));
}
```

To move to a different page, you use the navigation service and point it to the URI of the XAML file. The navigation service is exposed from the Windows Phone main page class. If you use the MVVM pattern, you may not have a reference to the page available. For this reason, you might find it helpful to define a bunch of global references to the various screens of the application:

```
public partial class App : Application
{
   public static GuessMainPage MainPage;
   ...
}
```

Finally, keep in mind that as a developer, you always should strive to make input as easy as possible for users. Picking up the most convenient keyboard layout is the primary aspect of this effort. The text box for the player name is expected to accept a string like a person's name. In Windows Phone, you control the input scope of the keyboard through the *InputScope* property. A good selection for the input scope when a name is being accepted by the input field is *PersonalSurname,* which does its best to help you enter a person's last name; specifically, it automatically uses an uppercase letter at the beginning and another one after each space (see Figure 10-17):

```
<TextBox x:Name="playerName" InputScope="PersonalSurname" ... />
```

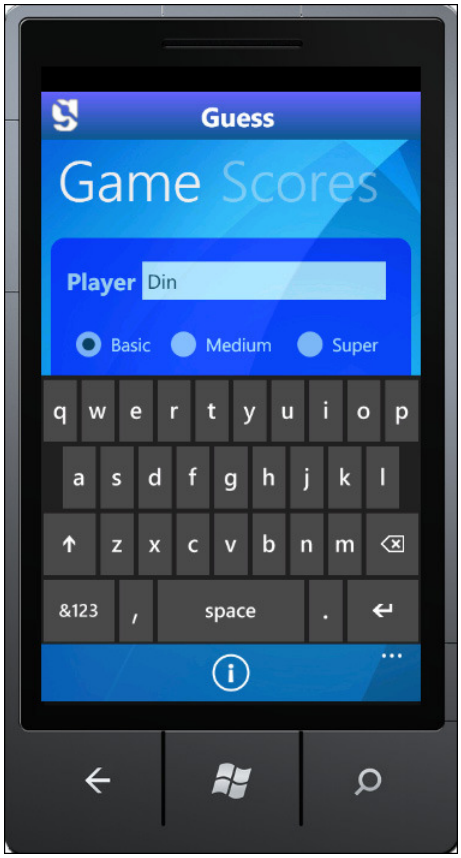



**FIGURE 10-17** Typing a name.

When designing a mobile user interface, you always should pay attention to the effects of the keyboard on the user interface. In particular, you don't want the keyboard to cover either the text box itself or a relevant button. Consider that the soft keyboard never covers the application bar. Windows Phone attempts to do a bit of work to ensure that the focused control is not covered. You should double-check that everything happens as you want it to. Failing here likely would cause users to rate your application poorly because the application becomes very hard to use.

## The Play View

When the user taps the Start button, the application saves the player name and level as default settings for future games and displays the gaming screen. The game screen just reads any data it needs from the storage; you have no need to pass data explicitly over the navigation system. (There will be more information on saving settings later in this chapter.)

The sample application consumes the logic of the game from a specific class that is responsible for generating the secret number, counts hints and attempts, updates the current range for user's feedback, and calculates the final score (see Figure 10-18).


**FIGURE 10-18** A digits-only keyboard for guessing a number.

The application guides the user to guessing the number, providing accurate feedback. If you enter a value outside the current range, the device vibrates:

```
public void NewAttempt()
{
    var number = AttemptedNumber;
    var isValid = _currentGame.IsValidAttempt(number);
    if (!isValid)
    {
        // Vibrate (300ms)
        VibrateController.Default.Start(300.Milliseconds());
        return;
    }
    var success = _currentGame.MakeAttempt(number);
    if (!success)
    {
        Update();
        return;
    }
   // You won!
   ...
}
```

The application bar—adapted to the context—offers a button to request a hint. At any time, you can quit the game and learn the secret number. You can quit the game from the application bar or by tapping the question-mark button in Figure 10-18. When you do so, a little animation reveals the secret number (see Figure 10-19).

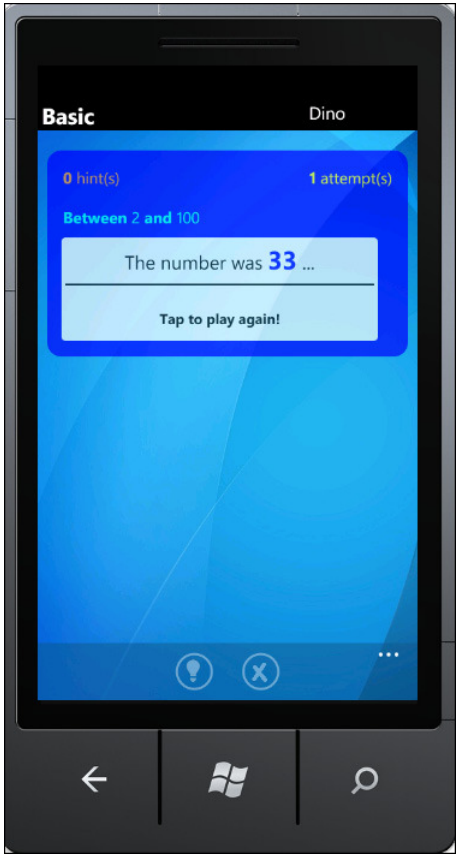



**FIGURE 10-19** An animated message box displayed when the game is over.

You define an animation as a XAML storyboard resource as follows:

```
<Storyboard x:Key="QuitPanelAnimation">
       <DoubleAnimation x:Name="Anim1"
                        From="100"
                        To="0"
                        Duration="0:0:0.10"
                        Storyboard.TargetProperty="Width"
                        AutoReverse="False" />
</Storyboard>
```

The sample animation moves the value of the *Width* property from 0 to 100 percent in a fraction of time. The animation XAML, though, doesn't say much about the holder of the *Width* property. That has to be set in code; this makes it possible for you to reuse the same animation for various targets:

```
internal void ToggleQuitPanel()
{
    // Set the target UI element being animated via the storyboard
    var storyboard = (Storyboard)Application.Current.Resources["QuitPanelAnimation"];
    Storyboard.SetTarget(storyboard.Children.ElementAt(0) as DoubleAnimation, quitPanel);

    EventHandler ehOpening = null;
    EventHandler ehClosing = null;
    ehOpening = (s, e1) =>
    {
        storyboard.Stop();
        storyboard.Completed -= ehOpening;
        quitMessage.Visibility = Visibility.Visible;
        secretNumber.Text = String.Format("{0:#,#}", _viewModel.SecretNumber);
        quitImage.Visibility = Visibility.Collapsed;
    };

    ehClosing = (s, e1) =>
    {
        storyboard.Stop();
        storyboard.Completed -= ehClosing;
        quitMessage.Visibility = Visibility.Collapsed;
        secretNumberText.Text = String.Empty;
        quitImage.Visibility = Visibility.Visible;
    };

    if (quitPanel.Margin.Left == 0)
    {
        storyboard.Completed += ehClosing;
    }
    else
    {
        storyboard.Completed += ehOpening;
    }

    // Start animation
    storyboard.Begin();
}
```

You also need to define handlers for when the animation ends. The handler will finalize graphical settings, such as displaying the secret number in an initially hidden panel.

When the user finally finds the secret number, a pop-up dialog box displays with a summary of the game:

```
// You won!
var dialog = new WinDialog(_currentGame);
dialog.Focus();
DialogManager.Show(_view, dialog);
```

*DialogManager* is a helper class that internally uses an instance of the *Popup* class. The actual content of the dialog box is provided through a user control. Here's the code that prepares and displays the pop-up box:

```
public static void Show(PhoneApplicationPage page, UserControl content)
{
    _page = page;
    if (_popupWindow == null)
        _popupWindow = new Popup();

    var colorBrush = new SolidColorBrush(Colors.Black) {Opacity = 0.75};
    var overlay = new Canvas { Background = colorBrush };
    overlay.Children.Add(content);
    _popupWindow.Child = overlay;
    _popupWindow.IsOpen = true;
}
```

The preceding code also dims the underlying screen, thus providing a full modal effect, as in Figure 10-20.

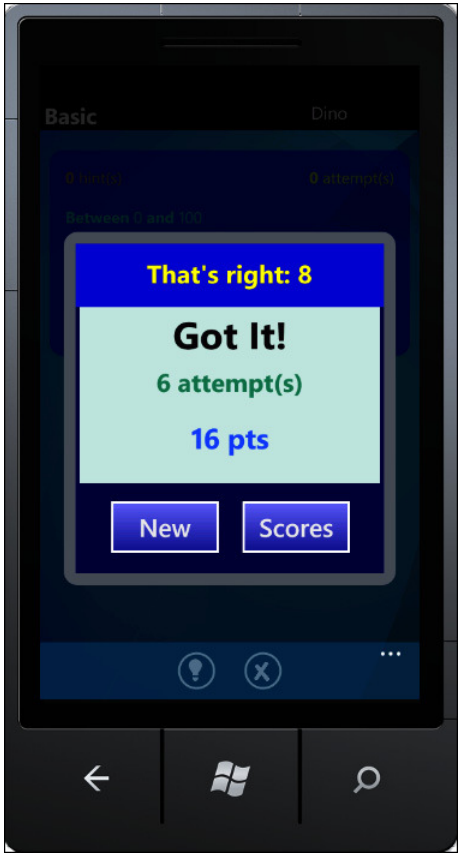



**FIGURE 10-20** The dialog box that the winner sees.

You don't need to do much in the handler of the New button in Figure 10-20: you simply reset the game and dismiss the pop-up box. It is more interesting, instead, what you do in the handler of the Scores button. You are currently on the Game page, and you want to return to the main page—the pivot page—and select a particular tab.

Simply navigating back to the main page may not be enough. You need to identify which tab you want selected. To do that, you can use query string parameters, as shown here:

```
public void ViewScores()
{
    var urlBase = String.Format("/views/guessmainpage.xaml?{0}=1", App.PivotIndexKey);
    App.MainPage.NavigationService.Navigate(new Uri(urlBase, UriKind.Relative));
}
```

You use a fixed query string parameter name and set it to 1—the index of the Scores tab in the pivot window. Whenever the user navigates to a page, Windows Phone fires the *OnNavigatedTo* event to the target page:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    var index = 0;
    if (NavigationContext.QueryString.ContainsKey(App.PivotIndexKey))
    {
        index = NavigationContext.QueryString[App.PivotIndexKey].ToInt();
    }
    if (index >0 )
        guessPivot.SelectedIndex = index;
    base.OnNavigatedTo(e);
}
```

You retrieve the query string parameter from the *QueryString* dictionary and select the proper tab.

## The Scores View

The Scores view consists essentially of a list of data items. In Windows Phone, you just arrange a view with a *ListBox* and proceed with data binding. For a nice effect, though, you might want to customize the template of the list items, as follows:

```
<ListBox x:Name="ScoreListBox"
         ItemsSource="{Binding History}">
    <ListBox.ItemTemplate>
        <DataTemplate>
           <StackPanel Orientation="Vertical">
                <Grid Width="430">
                   <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="200"></ColumnDefinition>
                        <ColumnDefinition Width="220"></ColumnDefinition>
                    </Grid.ColumnDefinitions>
                    <TextBlock Grid.Column="0"
                               Text="{Binding PlayerName}" >
                    <TextBlock TextAlignment="Right"
                               Grid.Column="1"
                               Text="{Binding Score,
                                      Converter={StaticResource NumberWithThousands}}" />
```

```
                    <TextBlock TextAlignment="Right"
                               Grid.Column="1"
                               Text="{Binding Level}" />
                    <TextBlock Grid.Column="0"
                               Text="{Binding DateOfScore,
                                      Converter={StaticResource NiceDate},
                                      ConverterParameter='ddd dd MMM yyyy, HH:mm:ss'}" />
                </Grid>
                <Rectangle Fill="#0ff" Grid.ColumnSpan="2" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

The data template is made of two columns. The leftmost column displays the player name and the date underneath. The rightmost column displays the score and the level underneath.

The data source of the list box is an array of *ScoreInfo* objects, defined as shown here:

```
public class ScoreInfo
{
    public String PlayerName { get; set; }
    public DateTime DateOfScore { get; set; }
    public GameLevel Level { get; set; }
    public Int32 Score { get; set; }
}
```

A *ScoreInfo* instance is created whenever some user wins a game. The newly created instance is then added to the application data and saved to the local storage. When the Scores view is being displayed, the list of saved scores is retrieved, sorted, and bound to the list box.

The list-box data template ends up displaying dates and numbers. What about formatting? You have two options: you can modify *ScoreInfo* to expose preformatted strings instead of specific types, or you can use XAML converters.

**Note** XAML converters make particular sense when you're using MVVM. If you're doing manual data binding, then the logic that you would put in a converter is a constituent part of your manual data-binding code.

A XAML converter is a plain class that converts values of one type to values of another type. In doing so, a converter may apply any transformation you wish. Here's an example for converting a number to a string placing separators for thousands:

```
public class ThousandsConverter : IValueConverter
{
    // From Int32 to String
    public Object Convert(Object value, Type targetType, Object parameter, CultureInfo culture)
    {
        return String.Format("{0:#,#}", value);
    }
```

```
    // From String back to Int32
    public Object ConvertBack(Object value, Type targetType, Object parameter,
                              CultureInfo culture)
    {
        Int32 number;
        return Int32.Parse((String) value);
    }
}
```

In this case, the converter is quite specific, and the formatting logic is hard-coded. You also can have more parametric converters. Here's one that converts dates to strings and that accepts the date format as a parameter:

```
public class DateConverter : IValueConverter
{
    public Object Convert(Object value, Type targetType, Object parameter, CultureInfo culture)
    {
        return parameter == null
            ? ((DateTime) value).ToString("dd MMM yyyy HH:mm:ss")
            : ((DateTime) value).ToString(parameter as String);
    }

    public Object ConvertBack(Object value, Type targetType, Object parameter,
                              CultureInfo culture)
    {
        return DateTime.Parse((String)value);
    }
}
```

The XAML data-binding syntax has specific placeholders for converters and related parameters. Here you create score items with dates and thousand separators:

```
<TextBlock Grid.Column="0"
           Text="{Binding DateOfScore,
           Converter={StaticResource NiceDate},
           ConverterParameter='ddd dd MMM yyyy, HH:mm:ss'}" />
<TextBlock TextAlignment="Right"
           Grid.Column="1"
           Text="{Binding Score, Converter={StaticResource NumberWithThousands}}" />
```

The *NiceDate* name in the code snippet seems an odd element. That's why we have one more step to accomplish: you must expose converter instances to the XAML parser by registering converter instances as resources in the local XAML file or, if necessary, at the application level. In doing so, you give instances a name such as *NiceDate*:

```
<UserControl.Resources>
    <amse:ThousandsConverter x:Key="NumberWithThousands" />
    <amse:DateConverter x:Key="NiceDate" />
</UserControl.Resources>
```

As mentioned, the list of scores is maintained as an array of *ScoreInfo* objects. The list is persisted as part of the application data. When read from storage, the scores are unordered. You can use the

LINQ syntax to sort in any way a list of objects. Here's a snippet from the *view-model* class behind the Scores screen:

```
public class ScoreViewModel : INotifyPropertyChanged
{
    public IList<ScoreInfo> History
    {
        get
        {
            return (from s in App.StateManager.Current.History
                    orderby s.Score descending
                    select s).ToList();
        }
    }
    ...
}
```

Figure 10-21 shows the final Scores screen in Guess for Windows Phone.

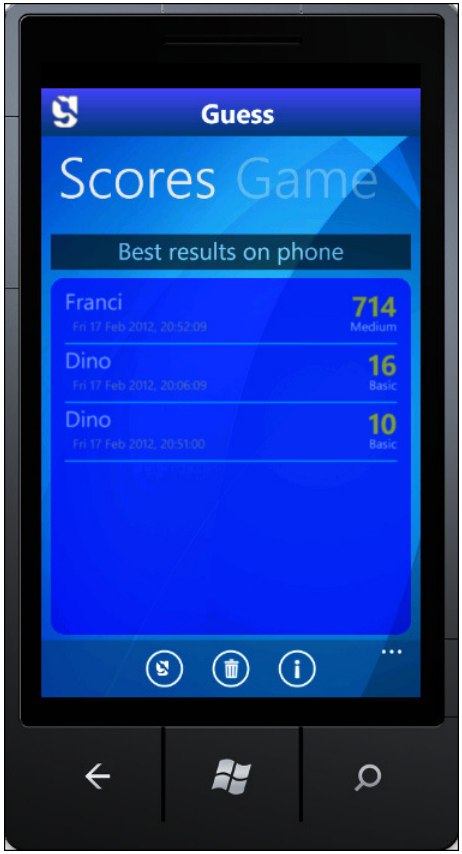



**FIGURE 10-21** The Scores view.

From the application bar, you start a new game and clear the history of the game on the phone. To start a new game, you just select the Games tab. Clearing the history is not a big deal per se; however, it poses the issue of refreshing the list box right after the click. Look at this code:

```
public void ClearScores(Object sender, EventArgs e)
{
    // Retrieves and clears the list of scores from storage
    var history = new List<ScoreInfo>(App.StateManager.Current.History);
    history.Clear();

    // Saves empty list back
    App.StateManager.Current.History = history;

    // Refresh the UI
    App.TheScoreScreen.Refresh();
}
```

For the moment, let's skip the *StateManager* object in the snippet; this topic will be covered in the section titled "Permanent Data Storage," below. Let's say it represents our access point to the phone's local storage. To refresh the user interface, you simply reset the data context of the view:

```
public void Refresh()
{
    DataContext = new ScoreViewModel();
}
```

Changing the data context has the effect of redrawing the user interface immediately.

## Other Programming Topics

The sample Guess application touches on a variety of aspects of mobile and Windows Phone programming. The list of hot topics, though, doesn't end here. Frankly, we can't call the overview of the Guess application complete until we go into the details of the storage layer. In this section, however, network access and common tasks such as sending an email or an SMS, or connecting to the marketplace, will be discussed.

### Permanent Data Storage

A mobile platform usually offers three different ways to save data permanently: Windows Phone is no exception. You have a plain simple dictionary model in the *ApplicationSettings* object, which is appropriate for a small amount of partially related data, such as player name and game level in the previous example. *ApplicationSettings* is a property of the *IsolatedStorageSettings* class. Here's how you use it:

```
IsolatedStorageSettings appSettings = IsolatedStorageSettings.ApplicationSettings;
appSettings.Add("player", "Dino");
```

You also have methods to read and delete an entry. The *ApplicationSettings* class is a string/object dictionary. It can contain any .NET object, so long as it is serializable. The beauty of *ApplicationSettings* is that is doesn't require you to (explicitly) care about saving and loading: persistence happens automatically. Application settings are stored locally on the phone.

For data that doesn't fit the dictionary model, you might want to try streams or relational tables. Let's find out more about streams. Compared to a dictionary, streams have the power of a much larger flexibility of the schema. You are not limited to a name/value schema and can define flexibly the relevant state of the application that you intend to persist. Next, you face the problem of serializing that relevant state to the phone storage.

In Guess, the relevant state is defined as follows:

```
public class GuessInternalState
{
    public GuessInternalState()
    {
        PlayerName = "???";
        Level = GameLevel.Basic;
        SecretNumber = -1;
        History = new List<ScoreInfo>();
    }
    public String PlayerName { get; set; }
    public GameLevel Level { get; set; }
    public Int32 Hints { get; set; }
    public Int32 Attempts { get; set; }
    public Int32 SecretNumber { get; set; }
    public IList<ScoreInfo> History { get; set; }
}
```

You now need an infrastructure that can save and reload this class to and from storage. It should come as no big surprise that the necessary code is quite generic and can be written in a highly reusable way. The Guess sample application comes with the following class, which saves a generic class called *T* to a given file in the isolated storage:

```
public class StorageManager<T>
{
    private readonly String _applicationDataFile;
    public StorageManager() : this("AppDataFile.dat")
    {
    }
    public StorageManager(String fileName)
    {
        _applicationDataFile = fileName;
    }

    public void Save(T data)
    {
        SaveDataToStorage(_applicationDataFile, data);
    }
    public T Load()
    {
        return LoadDataFromStorage(_applicationDataFile);
    }

    // More implementation details here
    ...
}
```

To save and load, you can use JavaScript Object Notation (JSON) serialization and the stream's API:

```
protected virtual Boolean SaveDataToStorage(String file, T data)
{
    // Iso store reference
    var isoStore = default(IsolatedStorageFile);

    // Result of the whole operation
    var result = false;

    // Create the serializer object.
    var serializer = new DataContractJsonSerializer(typeof(T));

    // Grabs the handle to the isolated storage, creates the file, and stores data.
    try
    {
        using (isoStore = IsolatedStorageFile.GetUserStoreForApplication())
        {
            using (var targetStream = isoStore.OpenFile(file, FileMode.Create))
            {
                serializer.WriteObject(targetStream, data);
                result = true;
            }
        }
    }
    catch (InvalidDataContractException)
    {
        if (isoStore != null)
            isoStore.DeleteFile(file);
    }
    catch (SerializationException)
    {
        if (isoStore != null)
            isoStore.DeleteFile(file);
    }
    catch
    {
        // Possible exceptions are: IsolatedStorageException | DirectoryNotFound
    }

    // Returns outcome of the operation
    return result;
}
```

And here's how you read the data back:

```
protected virtual T LoadDataFromStorage(String file)
{
    // Iso store reference
    var isoStore = default(IsolatedStorageFile);

    // Initialize the object to return.
    var data = default(T);

    // Create the serializer object.
    var serializer = new DataContractJsonSerializer(typeof(T));
```

```
    // Grabs handle to the isolated storage, opens the file (if existing) and reads data.
    try
    {
        using (isoStore = IsolatedStorageFile.GetUserStoreForApplication())
        {
            if (isoStore.FileExists(file))
            {
                using (var stream = isoStore.OpenFile(file, FileMode.Open))
                {
                    data = (T) serializer.ReadObject(stream);
                }
            }
        }
    }
    catch (NullReferenceException)
    {
        // code
    }
    catch (IsolatedStorageException)
    {
        if (isoStore != null)
            isoStore.DeleteFile(file);
    }

    // Returns data
    return data;
}
```

In the companion code of this book, these classes are buried in a reusable Windows Phone library. How would you manage storage from the application? To offer a simple programming interface to the developer, you can add another layer of code, as shown here:

```
public class StateManager<T> where T:class, new()
{
    private static readonly StorageManager<T> Storage = new StorageManager<T>();
    private static T _state;

    public void Save()
    {
        if (_state == null)
            return;
        Storage.Save(_state);
    }

    public void Load()
    {
        _state = Storage.Load() ?? new T();
    }

    public void Clear()
    {
        _state = new T();
        Storage.Save(_state);
    }

    public void Reset()
```

```
    {
        Clear();
        Load();
    }
    public T Current
    {
        get { return _state ?? (_state = new T()); }
    }
}
```

At this point, any user application like Guess requires a single step to set up storage (in addition to linking the library with the storage framework). You define the class that describes the relevant state (i.e., *GuessInternalState*) and provide a static entry point in the storage layer:

```
public partial class App : Application
{
    public static StateManager<GuessInternalState> StateManager =
                                new StateManager<GuessInternalState>();
    ...
}
```

To load and save the state, you use the following code:

```
// Load the state
App.StateManager.Load()

...

// Save the state
App.StateManager.Save();
```

Finally, in Windows Phone, you still have some options for storing data relationally. In Windows Phone 7.5, an application can use LINQ to SQL to store relational data in a local Microsoft SQL Server Compact edition database. The procedure requires that you start by writing all database-related classes, such as tables and data context. If you're familiar with LINQ to SQL, that means writing manually all the classes that the Data Import Wizard in Visual Studio would write for you in a LINQ to SQL .NET project.

Next, you create a new database file in the isolated storage. This has to be an empty *.sdf* file by default. You use the stream API to do this. Finally, you use the LINQ to SQL API to populate the database. Data will be stored in the isolated storage. Note that although Windows Phone 7.5 supports most LINQ to SQL features, some limitations still apply. Check it out at *http://goo.gl/oF6RS*.

**Note** In Windows Phone 7, you don't have native support for SQL Server Compact edition, but you still can rely on some other option, one of which is certainly SQLite. Note, though, that SQLite support on Windows Phone 7 is not well established, as it is on iOS and Android. You might want to resort to some Object/Relational Mapper (O/RM) facade that wraps up many of the details—an example is CoolStorage. You also might want to consider some other alternatives, such as Sterling (see *http://sterling.codeplex.com*).

## Accessing the Network

In a mobile application, the network comes and goes—and especially will change (from WiFi to GPRS and vice versa). This self-evident fact creates two key consequences for developers: the need to always check the network before participating in Internet operations, and the responsibility to ensure that the application works well even in the absence of connectivity. It is an ugly experience to launch a mobile application only to receive a cursory message box explaining that the application can't work because of an unspecified network error.

In mobile, network connectivity can be of different types: for example, it can be based on WiFi or General Packet Radio Service (GPRS). Different qualities of connectivity may make you choose a different strategy to implement the same operation. You probably can ignore the type of network if all you need to place is a GET call to grab a couple of strings. If you're planning a large upload or download, then you might want to reconsider the algorithm and implement a compensation mechanism to try again and again until the whole mass of data has been moved.

Furthermore, as network changes happen asynchronously with respect to the application, you should be able to detect changes to the network state as they happen. This is beneficial because it makes your application highly responsive as the network-related parts of the user interface turn on and off with the network state.

Windows Phone offers two slightly different APIs for version 7.0 and 7.5. The API available in Windows Phone 7.5 is more phone-oriented and exposes more detailed information. No API, though, offers a direct method that reliably answers the fundamental question: whether you can safely start a network operation. In other words, the entire Windows Phone network-checking API is based on *detected* capabilities rather than *effective* capabilities.

So the API offers the following method to check network availability:

```
// Use this in Windows Phone 7.5
if (!DeviceNetworkInformation.IsNetworkAvailable)
{
    // You can't perform a network operation
    ...
}

// Use this in Windows Phone 7.0
if (!NetworkInterface.GetIsNetworkAvailable())
{
    // You can't perform a network operation
    ...
}
```

The Boolean response that you get in both cases refers to the availability of some network of some type. It doesn't tell you whether you have effective access to the Internet. More, it may return *true* even if you are not connected to any network, but networks are available. Here's a much more effective way to check whether you can start a network operation:

```
private static Boolean IsConnected()
{
    var networkType = NetworkInterface.NetworkInterfaceType;
    return networkType != NetworkInterfaceType.None;
}
```

The network type is quite specific about the network interface—WiFi, Ethernet, mobile broadband, and more. In Windows Phone 7.5, you find simpler functions that just answer common questions:

```
DeviceNetworkInformation.IsWIFIEnabled
DeviceNetworkInformation.IsCellularDataEnabled
DeviceNetworkInformation.IsCellularDataRoamingEnabled
```

For detecting network changes, you can rely on the following global handler:

```
NetworkChange.NetworkAddressChanged += new NetworkChange_NetworkAddressChanged;
```

With the preceding line, you register your own handler for a system event that fires whenever the network address changes:

```
void NetworkChange_NetworkAddressChanged(Object sender, EventArgs e)
{
     // Detect the current network type
     var type = Microsoft.Phone.Net.NetworkInformation.NetworkInterface.NetworkInterfaceType;

     // Decide what to do ...
     switch(type)
     {
        case NetworkInterfaceType.Wireless80211:
          ...
     }
}
```

This API works in any version of Windows Phone. The latest Windows Phone 7.5 has a slightly different event handler that also specifies the type of change—connection, disconnection, or change of characteristics. Using this event handler, along with an effective strategy to detect effective connectivity, you can adjust the user interface and disable buttons that perform network tasks.

## Placing HTTP Calls

The Windows Phone programming interface for network operations is entirely asynchronous. If you feel you need a synchronous API because you have a sequential workflow of operations to implement, well, you'd better spend your time trying to work out your own chainable sequence of operations on top of existing async API rather than exploring tricks to make an async API work synchronously.

Windows Phone offers two classes for networking operations: *WebClient* and *HttpWebRequest*. The latter gives you total freedom of arranging whatever Hypertext Transfer Protocol (HTTP) call you may need. The former just offers a simplified programming interface, but it doesn't support all possible HTTP use-cases. A simple rule of thumb can be to try using *WebClient,* and if you can't find a solution quickly, then upgrade to *HttpWebRequest*. Here's an example of using *WebClient* to download a string:

```
var url = ...;
var client = new WebClient();
client.DownloadStringCompleted += client_DownloadStringCompleted;
client.DownloadStringAsync(new Uri(url, UriKind.Absolute));
```

The response is received by the specified callback function:

```
void client_DownloadStringCompleted(Object sender, DownloadStringCompletedEventArgs e)
{
    var responseAsString = e.Result;
    ...
}
```

At the end of a HTTP operation, you probably want to update the user interface. There are some threading issues to consider. With *WebClient*, the callback is invoked on the same thread—whether it is the UI main thread or a background thread. From a background thread, you can't update the user interface directly. Instead of just updating controls, as shown here:

```
labelNumber.Text = someNumber;
```

you resort to the following:

```
Dispatcher.BeginInvoke(() => { labelNumber.Text = someNumber; };
```

Cross-thread access will throw an exception.

**Note** To stay on the safe side with Internet operations, I recommend that you always wrap any operation in a *try/catch* block. This will prevent issues from incorrect detection of connectivity, but also issues due to async changes of connectivity type.

If the web request doesn't fall in the range of solutions supported by *WebClient*, you then can choose to use *HttpWebRequest*. Because of the async programming interface, arranging a call to *HttpWebRequest* is a bit annoying. The following code shows how to prepare a call:

```
var request = WebRequest.CreateHttp(url);
request.Method = "POST";
var package = new UploadPackage {Request = request, Parameters = parameters, UiWindow = ui};
request.BeginGetRequestStream(a => GetRequestStreamCallback(a, dataToPost), package);
```

The last instruction ends the preliminary phase of preparing the request. When the specified callback fires, you can start populating the body of the request. It is recommended that you wrap any data to write in the body in a helper structure, such as *UploadPackage* in this example:

```
private static void GetRequestStreamCallback(IAsyncResult asyncResult, Byte[] postData)
{
    try
    {
        var package = (UploadPackage) asyncResult.AsyncState;
        var request = (HttpWebRequest) package.Request;
        var postStream = request.EndGetRequestStream(asyncResult);
```

```
        // Write to the request stream
        postStream.Write(postData, 0, postData.Length);
        postStream.Close();

        // Start the asynchronous operation to get the response
        request.BeginGetResponse(GetResponseCallback, package);
    }
    catch (WebException webException)
    {
        ...
    }
    catch (InvalidOperationException e)
    {
        ...
    }
}
```

Finally, another callback—*GetResponseCallback*—will give you access to any response. Note that in this example, the data to write in the body of the request has been passed as an array of bytes. This is a valid approach if you have, say, an image to upload.

## Common Tasks

In Windows Phone, common tasks are grouped in two categories of API: launchers and choosers. A launcher is a common piece of the system user interface that launches a task in a fire-and-forget manner. The typical example is opening a browser window to a given URL:

```
var web = new WebBrowserTask { URL = "http://www.expoware.org/mobile" };
web.Show();
```

Another great example of a launcher is the common dialog box that appears when sending an email:

```
var body = ...;
var subject = ...;
var email = new EmailComposeTask
                {
                    To = "support@contoso.com",
                    Body = body,
                    Subject = subject
                };
email.Show();
```

Nicely enough, both the email and SMS composer dialog boxes let you set some fields to programmatically determined values.

A chooser, instead, is a dialog box that is expected to return a value to the caller application. For example, the application starts the camera so that the user can snap a photo. Once done, the control returns to the application, which is given a chance to access the photo:

```
var task = new CameraCaptureTask();
task.Completed += cameraCaptureTask_Completed;
task.Show();
```

Executing the preceding code opens the camera application so that the user can take a picture. As soon as the picture is accepted, the control reverts to the application and the *completed* callback is invoked:

```
public static void cameraCaptureTask_Completed(Object sender, PhotoResult e)
{
   if (e.TaskResult == TaskResult.Cancel)
   {
       if (e.Error != null)
           MessageBox.Show(e.Error.Message);
           return;
   }
   if (e.TaskResult == TaskResult.None)
   {
       MessageBox.Show(LiteralResources.ErrorPhotoChooserTask);
       return;
   }
   if (e.TaskResult == TaskResult.OK && e.ChosenPhoto != null)
   {
       // e.ChosenPhoto is the stream with the captured image
       ...
   }
}
```

Similarly, you can pick up an email address from the list of user contacts:

```
var emailChooserTask = new EmailAddressChooserTask();
emailChooserTask.Completed += emailChooserTask_Completed;
emailChooserTask.Show();
```

The callback receives the email of the selected contact and can process it as expected.

# Deploying Windows Phone Applications

Now, let's find out more about what it takes to test and distribute a Windows Phone application.

## Testing the Application

To test your application through the Windows Phone emulator, there are no prerequisites other than having the necessary tools up and running. To test the application on a real device, at a minimum, you need to be a registered App Hub developer. That's not enough, though, as you also need to register some test devices.

### Registering a Device

Any Windows Phone device is enabled to download any number of publicly released applications. Only a few devices, however, are entitled to run code in development. In particular, as a developer, you must register all devices that you intend to use for testing. Once your phone is registered, you can install, execute, and test unsigned (and not released) applications. A registered device is unlocked, and you can install any application on it under your responsibility.

To register your phone, you need a computer running Windows with the Zune client software installed and a currently valid App Hub developer account. You then connect the phone to the computer and run the Windows Phone Developer Registration tool, shown in Figure 10-22.


**FIGURE 10-22** Registering a development device using the Windows Phone Developer Registration tool.

To identify yourself as a registered developer, you enter your credentials to the App Hub account. It is important to note that the phone must be connected to the computer for the unlock procedure to work. Each developer account is limited to a maximum of three development devices. At any time, each Windows Phone device can't have more than 10 non-marketplace applications installed.

## The Beta Program

The Marketplace Beta is an extension of the Windows Phone Marketplace that enables you to publish an unfinished version of the application so that a few other people can download and test it. To access a beta application as a user, you don't need to unlock your phone. All you need is to find a deep link to the beta application, navigate it with the phone, and follow the instructions. The overall experience is not very different from installing a finished application from the marketplace.

As a developer, you just upload a beta version of the application, as shown in Figure 10-23.

The link to download the application is potentially accessible by everybody who learns it. However, the application will install successfully only on phones whose Windows Live ID matches one of the Live IDs in the application's package. As a developer, in fact, you also are required to list a few Windows Live IDs for users authorized to download the application. Any user whose Live ID is added to the beta submission will receive an email from the marketplace with a link for the download. You can't invite more than 100 users to join the beta. Any beta application expires a maximum of 90 days after release. As a developer, though, you can terminate the program at any time.

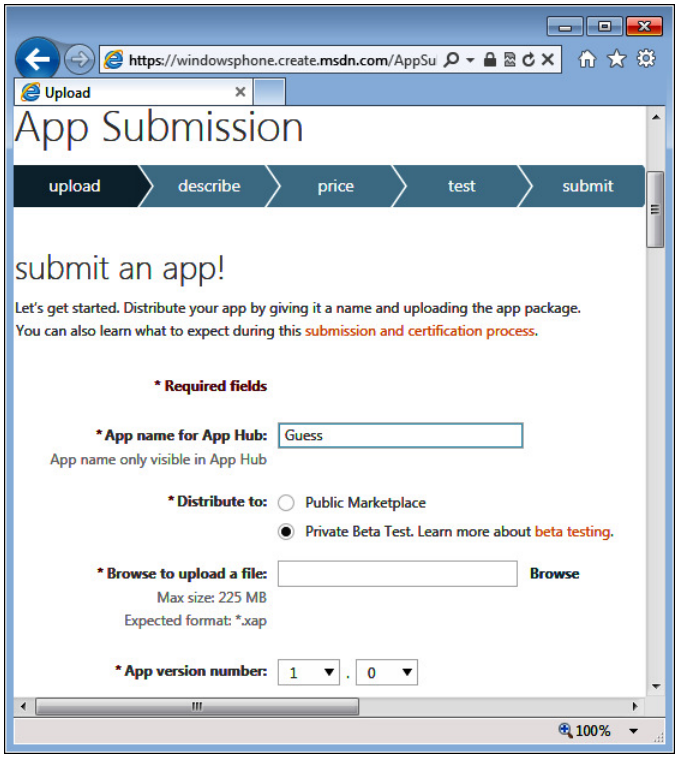



**FIGURE 10-23** Uploading a beta of the application.

Terminating the beta won't affect installed copies of the application, which will expire after 90 days. At this point, when users attempt to start the application, they will be given the option to uninstall it or send you feedback through the Marketplace client application.

**Note** Each Windows Phone is associated with a Windows Live ID to access the marketplace and download applications. If the developer fails to add a Windows Live ID to the beta submission, the user won't be able to install the application.

The Marketplace Beta works well in scenarios in which you effectively start a beta program that involves quite a few users. Once uploaded, the application is signed and wrapped up for publication (i.e., a layer is added to it that makes it expire after 90 days at most) and becomes available in a matter of hours.

It is important to notice that submissions to the Marketplace Beta do not result in instantaneous availability of the bits to the users. For this reason, the Marketplace Beta is *not ideal* in situations in which you need to release frequent updates to a single customer. In this case, it is recommended that you associate a customer's device with an App Hub account and proceed with direct downloads.

You can't update an existing beta, in fact. Instead, you have to terminate the current beta, ensure that users uninstall existing applications, and set up a new beta program that will take hours to be available.

## Distributing the Application

The only way to distribute a Windows Phone application is via the Marketplace. The structure of the Windows Phone marketplace is similar to appstores that you have for iOS applications. It is also similar to the Google Play Store, but not third-party Android stores.

### The Windows Phone Marketplace

You can publish applications to the marketplace for as little as $0.99 and for no more than $499.99. You can submit an unlimited number of paid applications and no more than 100 free applications. Microsoft will get 30 percent of the price for each copy of the application sold through the marketplace.

The marketplace will offer free hosting and advertising for all your applications. At this time, the marketplace is the only channel for distributing a Windows Phone application, except for unlocked devices.

**Important** How can you arrange a limited circulation of the application today? You might want to make the application freely available and ask users to enter a key to unlock all the potential. In doing so, you also should provide a demo key to Microsoft for due certification of the application.

### The Submission Process

The submission process requires you to upload the final XAP package of the application and provide metadata such as application title, description, search keywords, version number, and screenshots and other related graphics.

Detection of capabilities will occur at this time through code scanning. Only the detected capabilities will be added to the final manifest. Once uploaded, the application goes through the certification process, which entails both automated and manual checking of the features. If something goes wrong, you'll receive a detailed report with instructions to reproduce the failure and fix it. If all is good, you'll just receive a confirmation email and a link to the application's page on the marketplace.

It normally takes only a few days to get a first response. Application updates will go through the same route. More details about the submission process can be found at *http://goo.gl/dnzjF*.

### The Marketplace API

The Windows Phone marketplace comes with an API for a better integration with the application. For example, the following two lines of code bring up the review page of your application on the marketplace for the user to rate your application:

```
var task = new MarketplaceReviewTask();
task.Show();
```

Analogously, you can select a list of published applications and list them to your users, provided that it makes sense in the context in which you're in.

As part of the Marketplace API, the Windows Phone SDK offers support for creating trial and try-and-buy versions of a given application. In addition, you can choose to add advertising to the application via a native control in Windows Phone 7.5.

## Summary

Windows Phone is a complete rewrite of the Microsoft mobile operating system. As a platform, Windows Phone uses the Silverlight framework for development and borrows some ideas from iOS for the overall organization of the surrounding application infrastructure.

Although not all Windows Phone devices have the same screen size and hardware equipment, developing for Windows Phone devices is much easier than, say, for Android. There's nothing like the Android jungle in Windows Phone.

Compared to writing iOS and Android applications, developing applications for Windows Phone is a piece of cake. OK, because I've been a .NET guy since the early days, you may find that I'm just a little bit biased here. However, you will find that the development tools for Windows Phone development are definitely of top quality. The API is straightforward to learn if you already have a .NET and Silverlight background, and it is easy to learn even if you are relatively new to .NET programming.

Beyond this, writing a Windows Phone application poses the same challenges as any other mobile application—local storage, fake multitasking, wacky connectivity, limited resources, constrained input, and more.

It happened a couple of times already, and I expect it to happen even more in the future: hired to write a multiplatform application, I ended up using a hybrid platform (like PhoneGap) for iOS, Android, and BlackBerry, but then I turned to writing a truly native application for Windows Phone. If you have a strong .NET background, this doesn't really cost you *that* much more. On a side note, consider that Windows Phone also has a significantly characteristic user interface (the Metro style) that may make it hard to accept (for users and maybe even for Microsoft) an iPhone-like or a neutral user interface and experience.

The next chapter will tackle PhoneGap, a compelling framework for building hybrid mobile applications: half native and half web-based.

CHAPTER 11

# Developing with PhoneGap

*Ability is nothing without opportunity.*

*— Napoleon Bonaparte*

In mobile strategic consulting, there are two main scenarios. One is when your customer just needs to use the mobile media for internal purposes. The other is when the customer needs to use mobile media to reach their customers. In the former case, the challenge is to pick one platform and to build the entire stack of applications against that. A number of factors may influence the choice, such as devices in use throughout the company, the skills of the development team, and the existing infrastructure. In the latter case, the challenge is to define a strategy that reaches out to the widest possible audience at minimum cost. You reach a large audience by having a mobile website and also possibly by pushing out applications for a variety of platforms—iOS, Android, Windows Phone, BlackBerry, Symbian, Bada, and so forth.

When developing mobile applications for multiple platforms, things can get tough. Each platform is different—different software development kits (SDKs), different programming languages, different programming paradigms, different development tools, and sometimes even different operating systems and computers. Many organizations quite reasonably tend to develop the back end in house and outsource development for the various mobile platforms that they want to support. In this way, at least they save something on training and have a skilled team developing from day one. But the sore point is always the same—cost.

For example, writing the same application for three different platforms will cost you as much overall as developing three different applications. Even if the back end is the same for all the platforms, most of the cost of mobile development lies in the presentation layer, which remains highly specific (if not unique) for each platform.

Since the beginning of the mobile revolution, there's been an effort in the industry to find low(er)-cost solutions for cross-platform application development. The bad news is that no magic is possible; you can find a variety of solutions, but there's no silver bullet yet. The good news is that you have tools, and subsequently a margin, to craft your own solution that delivers multiple applications at a fraction of the cost of plain native development done from scratch.

A moment ago, I mentioned *lower-cost* solutions rather than *low-cost* solutions. Relativity reigns in cross-platform mobile development.

This final chapter discusses a prominent framework that currently is considered the primary choice for anyone looking for quick development of multiplatform mobile solutions. This framework is PhoneGap.

Originally developed by Nitobi, PhoneGap was acquired by Adobe in October 2011 and seems to be a centerpiece in Adobe's mobile strategy. PhoneGap is built around a very simple (but effective) idea: you build a local HTML5 application, and the framework packages it as a native application for a variety of mobile platforms. With PhoneGap, the majority of web developers—regardless of their server background (ASP.NET, PHP, Java, or Ruby)—can get up and running quickly, writing mobile applications using familiar technologies such as JavaScript and Cascading Style Sheets (CSS).

Does that sound like magic? Well, if you look only at the results, it may sound like magic, but just as in magic, tricks are being played behind the curtain. In this regard, PhoneGap is no exception.

**Important** As of version 1.5 of PhoneGap released in March 2011, most of the file names have been changed to use the prefix *cordova* instead of *phonegap*. Also in some posts, the name *Cordova* is used to indicate PhoneGap 1.5. I'll point out the differences later on, but I'm adding this note just to clarify that any naming related to Cordova refers to a particular version of the PhoneGap framework.

## The Myth of Cross-Platform Development

If you're a senior developer or architect—a nice way to say, if you have some silver hair—then the idea of cross-platform development probably has crossed your path a few times already, even before mobile. The fact is that using a single framework to build applications for different platforms will work only if you are willing to accept some compromises.

Given the fragmentation in mobile at the hardware level, you can't reasonably expect to pick a framework, write the same application for different platforms, and have it behave as smoothly and as fast as a native application on each platform, with the same compelling native user interface. In a way, it is like the old game of picking two options out of three. Great user experience, excellent performance, and lots of programming goodies are available to you—but you're allowed to pick only two.

Overall, I recognize two main paradigms for cross-platform development:

- The Virtual Machine approach is about writing the application using an abstraction layer that a platform-specific virtual machine will translate in the best possible way to the mechanics of the underlying platform.
- The Shell approach is about hosting a web application in a shell of native code. In this regard, the use of HTML5 and its advanced capabilities (e.g., input forms, local storage, and offline behavior) is a key factor.

The following sections explain each approach in more detail.

## The Virtual Machine Approach

In general terms, a *virtual machine* is an environment that creates a sort of abstraction layer on top of some underlying hardware. Device drivers probably were the first great example of an abstraction layer—a public and unified programming interface exposed to the operating system and internally deep knowledge of a particular hardware that was used to make the device work as expected. The Java language is also a great example of a virtual machine. The source code that you write in Java then is compiled to bytecode and processed by a virtual machine at run time. Due to this architecture, the same source code can run on any platform where a Java virtual machine exists.

This pattern just lies behind some of the frameworks for cross-platform mobile development.

### Structure of the Application

Frameworks based on the virtual machine pattern expose an abstracted application programming interface (API) that is not specific to the target system, such as iOS or Android. Such an API does expose concepts that map directly to native iOS (or Android) features, but at the API level, developers are not required to acquire and use any specific knowledge about a given platform. Figure 11-1 offers a graphical representation of the virtual machine pattern.

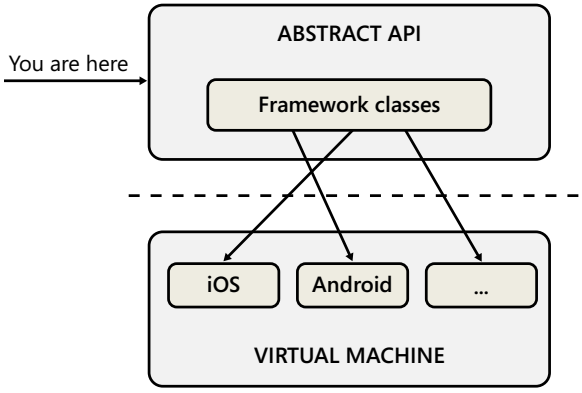



**FIGURE 11-1** Architecture of a virtual-machine solution.

In this model, you face a single (and very generic) API with which you write your application. Any calls you make to this exposed API then are mapped internally to class methods specific to a given mobile platform.

A framework based on the virtual machine pattern usually comes with an integrated development environment that lets you create an application project and then compile it for a number of different platforms.

## Titanium Mobile

Appcelerator's Titanium is a comprehensive framework for developing applications that target multiple platforms, including desktop and mobile platforms such as iOS, Android, BlackBerry, Mac OS, and Linux. The key aspect of Titanium is that you need only web skills to write cross-platform applications. Titanium Mobile is the segment of the overall SDK that targets mobile development. (See *http://www.appcelerator.com* for more information on the product and the company.)

Titanium Mobile offers an extensive API based on JavaScript. Hold on, though—it's not at all like writing an HTML page using jQuery and Ajax. First, there's no HTML; the Titanium framework consists of a set of JavaScript components that you invoke and glue together to build the desired user interface and implement the desired navigation and behavior.

This may not be considered mainstream programming, such as using HTML, CSS, and Ajax with jQuery. Still, it is probably faster than learning a brand-new language and SDK, such as Objective-C and Cocoa Touch or Java and Android SDK.

Titanium comes with its own IDE, in which you create projects and write JavaScript code. Your source code then gets packaged as a native application and deployed. The package includes the virtual machine for the selected platform as well as some bytecode created from the JavaScript source. At build time, the source code goes through a process called "cross-compiling," which produces a semi-compiled JavaScript code that is packed as binary code in the bundle, along with resources such as images. When the application runs, the bytecode is interpreted on the device and transformed to equivalent calls to native components to produce the desired effects (see Figure 11-2).

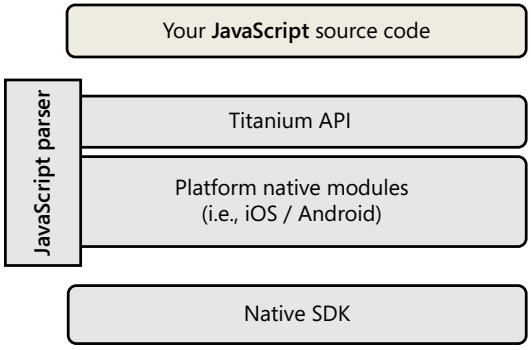



**FIGURE 11-2** The internal architecture of a Titanium application.

Because of the interpretation step, the application may not be as quick to load as an application that's entirely written against the native SDK. In addition, debugging can be problematic in interpreted environments. However, Titanium Mobile uses an Eclipse-based integrated development environment (IDE) that offers a debugger, so you can set breakpoints and inspect variables at run time comfortably. The deployment experience also is greatly simplified. To build your application for iOS, you must open the project in the Titanium Studio IDE running on a Mac. In contrast, when you open the project on, say, a Windows machine, you can build the same project for Android or BlackBerry and deploy it to any connected devices.

The Titanium API offers a set of classes to create the user interface, interact with sensors and device services (contacts, calendar, and media), and deal with persistence, localization, and networking. To add more functionality, you can create extension modules for the native platform or even fork the JavaScript API on GitHub. The source code is available at *https://github.com/appcelerator/titanium_mobile*. The following code snippet shows the classic HelloWorld application for Titanium:

```
var window = Titanium.UI.createWindow({
    title: "Hello, Titanium",
    backgroundColor: "#00f"
    color: "#fff"
});
var label = Titanium.UI.createLabel({
    color: "#ff0",
    font: {font-size: 30, font-weight: bold}
});
var button = Titanium.UI.createButton({
    title: "Say hello"
});
button.addEventListener("click", function(e) {
    label.text = "You clicked me";
});
window.add(label);
window.add(button);
window.open();
```

As you can see, it's still plain old JavaScript with a CSS and Document Object Model (DOM)–like syntax allowed to define the layout of the user interface.

**Note** The suite of Titanium products is often (and reasonably) compared to Adobe AIR because it can be used to create applications for Windows, Mac, Linux, and mobile platforms.

## Flash Builder

Adobe offers a version of Flash Builder that also allows you to compile Adobe AIR applications for a few mobile platforms: iOS and Android, plus BlackBerry PlayBook. The product—Flash Builder Premium 4.6—uses native extensions to Adobe AIR to support those three mobile platforms.

As developers, this means that you can use a rich IDE where you define the application layout using the mXML markup language and add logic through ActionScript files (see Figure 11-3).

At run time, your code is compiled against the AIR run-time environment and executed within the AIR virtual machine. You have no need to write directly to a mobile SDK; you can use existing skills to write iOS and Android applications.

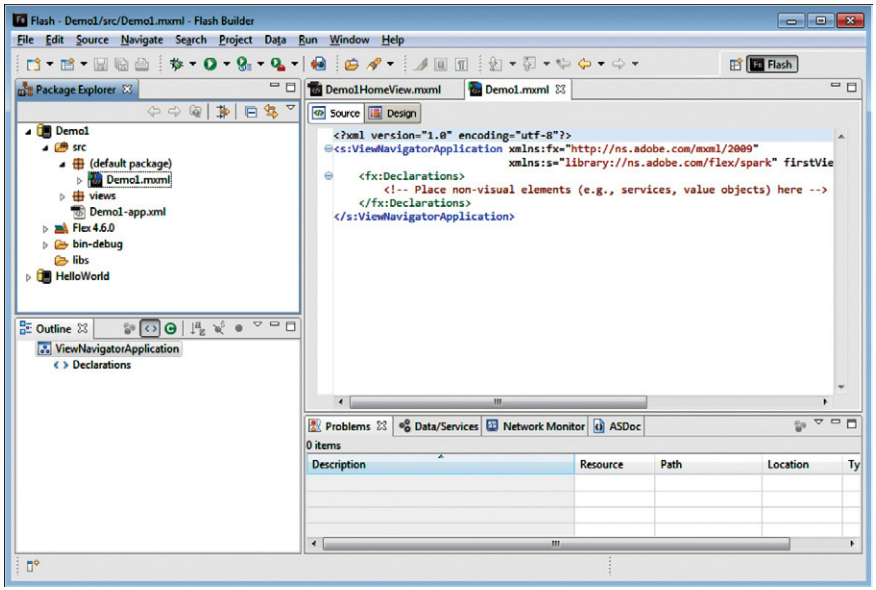



**FIGURE 11-3** The Adobe Flash Builder 4.6 environment in action.

## The *Shell* Approach

Nearly every mobile native platform offers a component to view a webpage. The idea behind the Shell approach is to integrate a web-view native component with the user interface of the application. You can mix a native user interface with web-based content to different degrees. For example, Figure 11-4 shows an application whose user interface is half native and half HTML-based. As you can see, the final effect is very nice, and there's little noticeable difference for users.

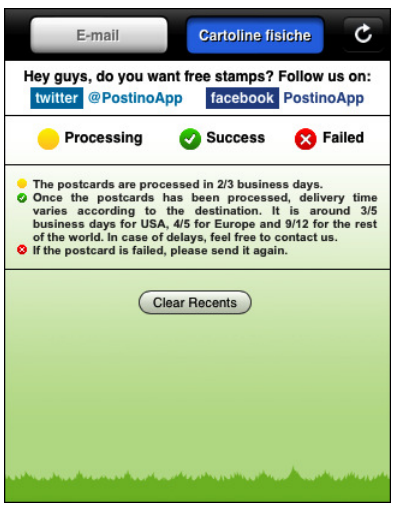



**FIGURE 11-4** A mixed user interface with native and HTML views. The navigation bar is native; the rest of the user interface is an HTML page.

You can take this approach even further and build native applications whose entire user interface is HTML-based.

What about the source of the HTML content? Does it have to be downloaded from a remote server and require an active connection? Or should it be embedded as local resources in the native application? It can actually work both ways.

The sample application shown in Figure 11-4 displays a page from a remote server accessed over Hypertext Transfer Protocol (HTTP). The application also displays a friendly user interface when connectivity is unavailable., the PhoneGap framework allows you to write native applications whose user interface is entirely HTML-based, but whose HTML resources are packed with the mobile application and referenced locally through the *file://* protocol.

## Structure of the Application

The Shell pattern requires a shell of native code that hosts a set of HTML pages. The native shell has a simple structure; it hosts an instance of the web-view component that each mobile platform provides to display web content. The web-view component is displayed in full-screen mode, so in this case, the HTML content is all that the user sees and interacts with (see Figure 11-5).

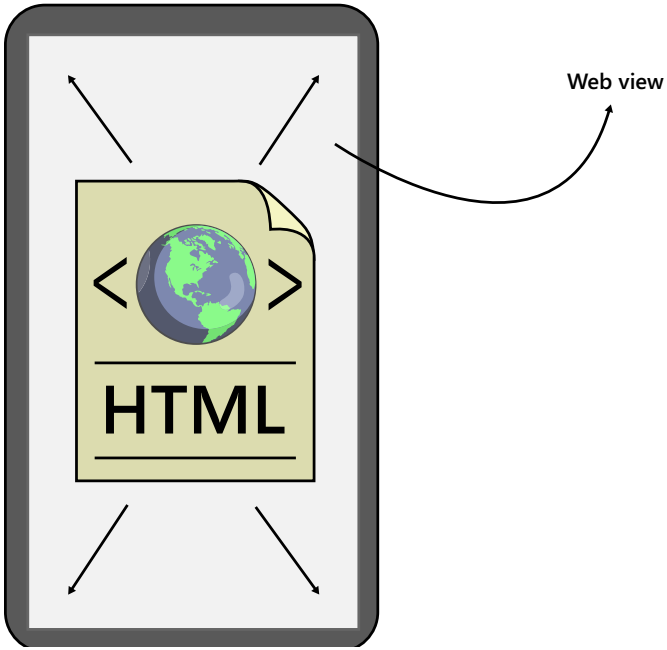



**FIGURE 11-5** The structure of a solution based on the Shell pattern.

As a developer, all (or most) of what you do is write a set of HTML pages. Navigation between pages is managed by the embedded browser (the web-view component from Figure 11-5) and controlled via device interactivity (such as clicking the Back button). The HTML-based user interface,

however, should consider offering its own navigation system, which will be uniform across the platforms. This is especially important for iOS, which runs on devices that lack a physical Back button.

## The PhoneGap Framework

PhoneGap is an open-source framework that enables developers to write mobile applications by packaging static HTML pages in a shell of native code. The shell of native code has a fairly simple structure: it is a native window with a single web view. The web view displays a set of HTML pages (and associated files) embedded as resources in the application's bundle.

A PhoneGap application *is* a native application in the sense that it must go through the canonical approval process required on some platforms (such as iOS and Windows Phone) and is distributed and installed through appstores. However, its overall performance is sometimes not as smooth and fast as a truly native application. This is because all layout rendering takes place via the web-view component. As you can see, the embedded device browser plays a key role in the performance of a PhoneGap application. For this reason, it is not surprising that the same web codebase, after being packaged as an application, may behave differently on different devices.

**Note** PhoneGap applications usually perform their best on iOS devices because of the high quality of the Safari mobile browser. On Android, performance depends on the actual device characteristics, as well as the features and capabilities of the user agent. Another platform that might be a great target for a PhoneGap application is Bada, because some Samsung devices equipped with Bada 2.0 (specifically Samsung Wave 3) ship with an excellent HTML5 browser.

A PhoneGap application binds a native library that acts as the bridge between the web-view, in-browser environment and the underlying platform. HTML pages can invoke objects from the PhoneGap JavaScript library that refer to device-specific features, such as haptic feedback, an accelerometer, a camera, and more. The PhoneGap JavaScript library calls into ad hoc objects published in the browser environment following the extensibility model of the device browser (see Figure 11-6).

**Important** I want to focus on this key point. By using PhoneGap, you gain the ability to express your user interface and logic using HTML5, CSS, and JavaScript. You don't lose the ability to access device-specific features and hardware. The most commonly used features are exposed to you automatically and comfortably via the PhoneGap JavaScript library. If you need more, or need a different API, you can create your own native shells of code and plug them into the PhoneGap framework. Furthermore, you'll likely reuse your HTML+CSS+JavaScript solution across all the mobile platforms that PhoneGap currently supports: iOS, Android, Windows Phone, BlackBerry, Symbian, and webOS.

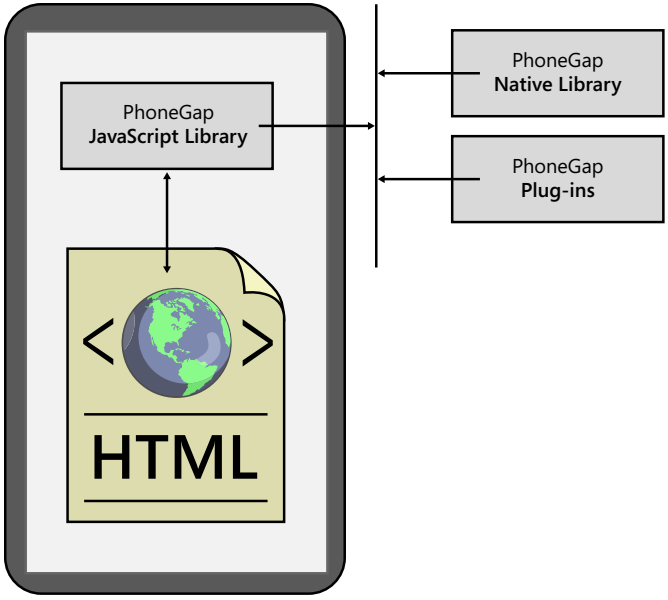



**FIGURE 11-6** The run-time environment of a PhoneGap application.

In this way, you can not only use predefined PhoneGap wrappers for most device sensors and functionality, but you also can add new JavaScript objects based on platform-specific functions that are not natively mapped by PhoneGap.

A custom extension to PhoneGap is called a *plug-in*. To create a plug-in, you need a piece of native code that is written in one of the languages supported by the platform and invokes native functions. For example, for iOS, you can write a PhoneGap plug-in by using Objective-C to call functions in Cocoa Touch. Then this native code is embedded in a class derived from a PhoneGap base class for the target platform. Finally, a bit of JavaScript code just wraps the native code. Here's an example of a PhoneGap plug-in for playing video on Android devices:

```
window.plugins.webintent.startActivity({
    action: WebIntent.ACTION_VIEW,
    url: 'http://.../someVideo.mp4'},
    function() {alert('It works.');},
    function() {alert('Failed to open the URL');}
});
```

In this particular case, you don't need any native code because the PhoneGap JavaScript library already contains the logic to start an activity. Playing a video is simply a matter of requesting the *ACTION_VIEW* standard action on a given resource. But this example illustrates how you would write and deploy a custom activity in other cases to perform exactly the tasks you need.

A PhoneGap application displays an HTML- and CSS-based user interface, and all its behavior is controlled by JavaScript routines. You can use a variety of popular JavaScript frameworks in your PhoneGap front end, including jQuery Mobile, jQuery UI, or any of the JavaScript microframeworks out there.

Sencha Touch (see *http://www.sencha.com*) is another popular JavaScript library optimized for building mobile web solutions that can be compiled then natively with PhoneGap. Sencha Touch forces you to create the user interface via a number of ad hoc JavaScript components. Basically, an application based on Sencha Touch is a single-page application centered on the following code:

```
new Ext.Application({
    launch: function() {
        new Ext.Panel({
            fullscreen: true,
            dockedItems: [{xtype:'toolbar', title:'Simple Demo'}],
            layout: 'fit',
            styleHtmlContent: true,
            html: '<h2>Hello from Sencha!</h2>'
        });
    }
});
```

The *Ext.Application* object is the main application object, and its launch method contains startup code. In particular, the startup code in this example creates a full-screen panel that displays a "Hello" message. The *Ext.Panel* object is the main container within which all the HTML magic will happen. Here, you can use Ajax calls and any form of in-memory manipulation of the page DOM. The word *touch* in the name of the library is not coincidental. It is there to remind people of the great support for touch events and animations that the library offers, along with ad hoc themes for iOS and Android.

## Handmade Hybrid Applications

PhoneGap performs the trick of taking a bunch of HTML pages and packaging them (and related resources) all into a single native application bundle. You write a single codebase composed of HTML, CSS, and JavaScript. You decide which JavaScript framework (if any) to use, and you freely use CSS and HTML markup to define the layout of the user interface. Navigation, beyond the basic forms of navigation natively offered by the host browser, is up to you.

**Important** Unlike Titanium and Flash Builder, with PhoneGap, you sometimes may have a single codebase—but not a single project. You must create a platform-specific project using Xcode, Eclipse, Microsoft Visual Studio, or whatever else is required on the platform to compile and deploy a native application. In return, PhoneGap currently supports more mobile platforms than Titanium. More often than not, in fact, you might need to tweak the user interface on a given platform because of minor issues with browsers or to reflect the native user interface of the platform better. For example, you can decide to have a tab bar at the bottom of the page in the iPhone version and at the top of the page in Android. For Windows Phone, the user interface is considerably different and may require more customization. With that said, however, you still can use PhoneGap to reuse all your JavaScript logic, changing only the CSS or some HTML markup.

As you'll see in more detail later on in the chapter, with PhoneGap applications, the major problem lies in page transitions and navigation. Compared to truly native applications, PhoneGap applications sometimes may be slower and less fluid in such transitions. The result depends on the quality of the device browser, but for exactly that reason, it's beyond your control. Therefore, you may want to consider an alternative solution that consists of having a skeleton with a native header, footer, and navigation, but where all screens are based on (local or remote) HTML, displayed through a web view (see Figure 11-7).

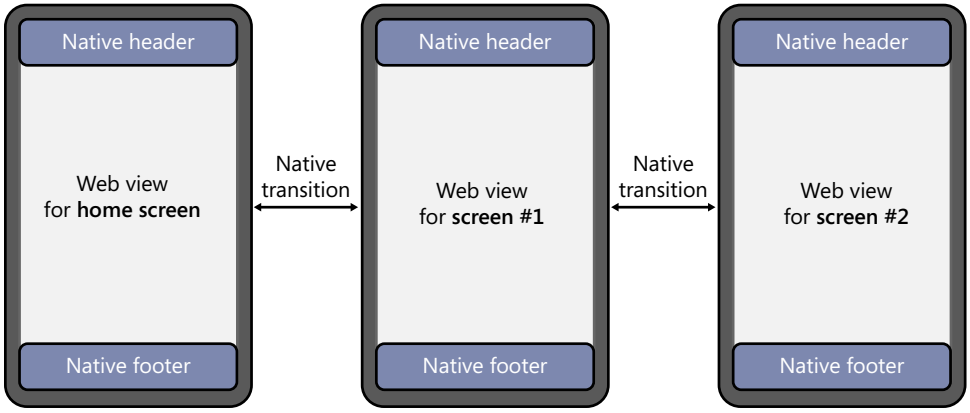



**FIGURE 11-7** The structure of a handmade hybrid solution.

This way, you can focus your HTML efforts on individual pages for individual screens and don't need to embed navigation logic in HTML. This may be beneficial in two ways. First, it offers a better experience to users because transitions are smoother and use the same animation as truly native applications. Second, it allows you to build an overall experience that is more consistent with native applications than you can achieve with PhoneGap solutions alone. For example, you could give your application a menu in Android, an application bar in Windows Phone, and a canonical Back button in iOS without artifacts and simulation.

At the same time, such a solution is not free of issues. It still requires that you create and maintain one project per platform. Compared to a pure PhoneGap scenario, the project is only a bit more sophisticated; you can decide how much more sophisticated you want it to be. In addition, because you use the web view only for displaying HTML, it is hard (but not impossible) to access device hardware and trigger native code from the HTML. One approach worth exploring entails intercepting links within the web view and handling them from within native code.

Overall, considering that most tools offer to create the bare skeleton of application that you need through a wizard (this is true with Xcode and Visual Studio, for example), this hybrid approach can be a low-cost way of developing multiplatform applications that are essentially based on simple content presentation.

**Important** When you host a web-view component in a mobile application page, you may run into some security issues related to linking cross-domain URLs. As a general rule, make every effort to ensure that the page displayed always posts to URLs on the origin site and doesn't link via anchors to any external resource. In a way, this is an even stricter policy than the usual cross-domain policies that you face in general web programming, such as making Ajax calls from within a client page. It is key to notice that web views hosted via the PhoneGap framework are not subject to cross-domain restrictions. There'll be more on this later in this chapter in the section titled "Ajax and Cross-Domain Issues."

# Building an HTML5 Solution

Let's see what it takes to write a full PhoneGap application. The first step requires that you create a set of HTML pages that serve the expected user interface and behavior of the desired application. As most mobile browsers support HTML5 very well, you can fully use markup enhancements brought by HTML5, such as input elements and a new API such as local storage. The logic is based on JavaScript and may use any library you feel comfortable with. In this example, I'll be using a bit of jQuery Mobile (and the core jQuery library as a dependency).

## JavaScript Ad Hoc Patterns

In highly dynamic HTML pages, the amount of JavaScript code may be significant. This means that without strong discipline, you end up with kludges and spaghetti code. My suggested approach is splitting the JavaScript code into four main areas: presentation, localization, application state, and application behavior.

The goal of these sections is preparing the ground for the sample Guess application that was developed for iOS, Android, and Windows Phone in the previous chapters. The application is a simple game of guessing a secret number.

### The JavaScript Presentation Layer

It's fairly common in JavaScript—but also fairly naive—to mix code that updates the user interface with code that orchestrates actions following a user's requests. On the other hand, JavaScript was created to be a simple-to-use tool in the hands of non-developers. And original creators, in a way, made a point of having a language for nearly everybody (like a scripting language) but powerful enough to mimic what at the time was reckoned to be *the* language to code for the web—Java.

For many years, JavaScript has been a language that could deliver true power in proper hands, but also messy code in most hands. Today, JavaScript skills are—more often than not—specified as a strong requirement in web-related job applications. When a language gains popularity, it tends to be used for increasingly complex tasks. At some point, more discipline becomes a necessity.

A golden rule of JavaScript programming is to have a global object that contains all your functions and object declarations. You might want to have a line similar to the following on top of all your JavaScript files:

```
var GLOBALS = GLOBALS || {};
```

The name *GLOBALS* is clearly arbitrary, but the way to instantiate it is not. In JavaScript, any global variable becomes an entry in the global system namespace. An application that has multiple global variables inevitably pollutes the global namespace. The whole topic of polluting (or not) the JavaScript namespace is, in my humble opinion, less relevant than it may seem for most developers. You definitely should care about it if you write a library or an extension that others may use in their own application-specific JavaScript code, but it's quite another animal when you write JavaScript for yourself.

The global namespace is rooted in the browser's *window* object, so it never trespasses the boundaries of the browser's session. So what is referred to as a *global namespace* is, in the end, the global namespace for the browser's session. That's probably not a big concern if you're only writing script code for a webpage. But good habits are never bad to practice; good habits are relatively cheap and keep your code cleaner and easier to maintain.

As declared above, *GLOBALS* ensures that an entry with that name is created in the global namespace if it doesn't exist already. If the entry already exists, then a reference to it is associated with the name *GLOBALS* in the scope of the current script file.

For each screen of the application, a presenter JavaScript object is created, as shown here:

```
var GLOBALS = GLOBALS || {};
GLOBALS.HomePresenter = function () {
    var that = {};

    // Populate the UI with localized text and attach event handlers.
    that.init = function () {
       ...
       $("#buttonStart").val(GLOBALS.Literals.HomeStartButton);
       $("#buttonStart").click(that._onStart);
    }

    // Handle click event on Start button
    that._onStart = function () {
       ...
       // Push next screen
       $.mobile.changePage("/Views/game.html", {});
    }

    // Return a new instance
    return that;
}
GLOBALS.GamePresenter = function () {
    ...
}
GLOBALS.WinnerPresenter = function () {
    ...
}
```

A presenter class encapsulates all the required knowledge about element IDs and uses jQuery and jQuery Mobile (or any other helper library that you may use) for building and manipulating the user interface. Any interaction between the application logic and user interface takes place through the members of the presenter object.

**Note** Using presenter objects also makes it easy to have unobtrusive JavaScript code that doesn't stuff HTML elements with script and style references and default values.

Each HTML screen needs some initialization. Everything can be summarized in the following code:

```
<div id="homepage" data-role="page">
     <script type="text/javascript">
         $("#homepage").bind("pageinit", function () { GLOBALS.appInitialize() });
         $("#homepage").bind("pageshow", function () { GLOBALS.pageInitialize("home") });
     </script>
</div>
```

Each page—in this example, jQuery Mobile pages are used (as explained in Chapter 5, "HTML5 and jQuery Mobile")—binds to the *pageinit* event for onetime initialization, and it binds a handler to *pageshow* for each per-display initialization:

```
// App startup
GLOBALS.appInitialize = function () {
    var languageSuffix = GLOBALS.getLanguage();
    GLOBALS.loadLanguage(languageSuffix);
    GLOBALS.loadPresenters();
    GLOBALS.initializeState();
}

// Page startup
GLOBALS.pageInitialize = function (screen) {
    if (screen == "home")
        GLOBALS.Presenters.Home.init();
    if (screen == "game")
        GLOBALS.Presenters.Game.init();
    if (screen == "winner")
        GLOBALS.Presenters.Winner.init();
}
```

Presenters are instantiated once when the page loads for the first time. References to presenters are stored as members of the *GLOBALS* object for use later during the display of the page. Here's the code that instantiates the presenters:

```
GLOBALS.loadPresenters = function () {
    GLOBALS.Presenters.Home = new GLOBALS.HomePresenter();
    GLOBALS.Presenters.Game = new GLOBALS.GamePresenter();
    GLOBALS.Presenters.Winner = new GLOBALS.WinnerPresenter();
}
```

In the code that starts up a HTML client application, you also will find some logic that deals with localization and state management. Let's find out more.

## The JavaScript Localization Layer

There are only a few ways to deal with localization in JavaScript. In websites, you often take care of localization on the server side by either checking the browser's locale or the server's thread locale—depending on the needs—and then loading the appropriate set of resources: literals, images, views, and so forth.

For a client-side HTML page that serves as the foundation of a mobile application, you probably don't need anything more than a way to pick up a different set of strings based on the browser's locale. You can work out your own solution for that, even though some globalization plug-ins exist for jQuery.

Right off the bat, you need to figure out the language of the browser. For a mobile phone, the language of the browser corresponds to the language of the device. (The same is not necessarily true for a laptop where browsers normally offer to set the language independently from the operating system.) The following code detects the browser language:

```
GLOBALS.getLanguage = function () {
    var language = navigator.language;
    if (typeof navigator.language == "undefined")
        language = navigator.systemLanguage; // IE only

    // Some browsers return XX; some return XX-XX: cut to first 2 letters to be on the safe side
    language = language.substring(0, 2).toUpperCase();
    return language;
}
```

The code uses a different approach for Windows Internet Explorer, as opposed to other browsers, and returns a string containing the language settings. It also truncates the string to the first two letters of the culture, which may be returned as (for example) *en* or *en-us*. Then you can use the language string to select the appropriate dictionary of string literals, as shown here:

```
GLOBALS.loadLanguage = function (lang) {
    GLOBALS.Literals = GLOBALS.AvailableLiterals[lang].value;
}
```

*AvailableLiterals* is defined as follows for an application that supports English and Italian:

```
GLOBALS.AvailableLiterals = {
    EN: { value: GLOBALS.Literals_En },
    IT: { value: GLOBALS.Literals_It }
}
```

Finally, literals are defined in distinct dictionaries, one per supported language:

```
GLOBALS.Literals_En = {
    Separator: ",",
    HomePlayerLabel: "Player",
    ...
}
GLOBALS.Literals_It = {
    Separator: ".",
```

```
    HomePlayerLabel: "Giocatore",
    ...
}
```

At any location in the user interface that requires a localized string, you use an entry from the *Literals* dictionary:

```
$("#inputPlayerLabel").text(GLOBALS.Literals.HomePlayerLabel);
```

The preceding code, excerpted from a presenter object, just sets a label in one of the application pages.

## The JavaScript Application State

Any mobile application needs to save its own state for at least two reasons: to give a sense of continuity to the user and to save some typing. This means that you should figure out the shape of the application state quite early in the development process. You then load the application state at the startup of the home page with the following code:

```
GLOBALS.initializeState = function () {
    GLOBALS.Current = new GuessState();
}
```

In this example, *GLOBALS.Current* can be accessed from any page. It returns the current state of the application. *GuessState* is a JavaScript object whose structure closely resembles the structure of the application; it has a member for each significant piece of information that the application needs to store persistently. Here's an example:

```
var GuessState = function () {
    var that = {},
        DefaultPlayerName = "???",
        NoSecretNumber = -999;

    that.init = function () {
        var loadedFromState = this.load();
        if (!loadedFromState) {
            that.Player = DefaultPlayerName;
            that.setLevel(GameLevels.Basic);
        }

        that.SecretNumber = NoSecretNumber;
        that.Attempts = 0;
        that.Hints = 0;
        that.History = [];
    }

    // More code here
    ...

    // Return a new instance
    return that;
}
```

Because the sample code is the HTML version of the Guess game discussed in previous chapters, you shouldn't be surprised to find properties such as *Attempts*, *Hints*, *SecretNumber*, and *History* in the application's state.

To display the number of attempts that the player has made in the game session, you use the following expression:

```
alert(GLOBALS.Current.Attempts);
```

The state of the application must be saved to some storage and retrieved from there. This is where an HTML5-ready browser comes in handy, with its support for local storage.

**Note** Browsers hosted on devices running operating systems such as Android 2.0 and later, any iOS, Windows Phone 7.5, and all the devices identified as smartphone operating systems in Chapter 6, "Developing Responsive Mobile Sites," typically offer good support for HTML5, which makes it possible to wrap up HTML5 applications as native applications, as you'll see later in this chapter.

In HTML5, you use the *localStorage* object to save data as plain text or in a JSON format. Here's the code that saves the *GuessState* object to the local storage of the browser:

```
var GuessState = function () {
    var that = {},
    ...
    that.save = function () {
        var serialized = JSON.stringify(GLOBALS.Current);
        window.localStorage[GLOBALS.GuessStateEntry] = serialized;
    }
}
```

It should be noted that *JSON.stringify*—a utility to turn a JavaScript object into a JSON string—cannot be assumed to be available on just any browser. However, it is safe to assume that it is supported in browsers on devices running advanced mobile operating system versions.

When the *GuessState* object is initialized, it should read values from the local storage. This is what the *load* function does:

```
var GuessState = function () {
    var that = {},
    ...
    that.load = function () {
        var serialized = window.localStorage[GLOBALS.GuessStateEntry];
        if (serialized == null || typeof serialized === "undefined")
            return false;
        var obj = JSON.parse(serialized);
        this.Player = obj.Player;
        this.SecretNumber = obj.SecretNumber;
        ...
    }
}
```

It is key to note that assigning a deserialized object directly to *GLOBALS.Current* may make any methods that you may have currently defined on *GLOBALS.Current* disappear. So long as you have a state object defined as an *anemic object* (just properties, no methods), deserializing to *GLOBALS.Current* directly works just fine. Otherwise, because serialization persists only properties, you deserialize to a temporary variable and then manually copy values to the existing instance of *GLOBALS.Current*. In this case, that's precisely what is required, because this example has *save* and *load* methods defined on the state object itself.

### The JavaScript Application Behavior

The logic of the application depends on yet another JavaScript object, which also is defined as a member of the *GLOBALS* root container. Object containers are often referred to as *namespaces* in JavaScript.

For the specific case of this Guess application, the application behavior consists of methods that implement the logic of the game. You need a method to check whether a given number is a successful or an invalid attempt; you'll also have methods that perform such actions as generating a new number, resetting counters as a new game is started, quitting the game, providing a hint, and calculating the score.

These methods have no awareness of the details of the user interface. They are fully isolated from other layers and therefore are fully testable. The actions that these methods implement is orchestrated by code in the page presenters.

## The Sample Application

In past chapters, you've seen the same application—Guess—coded for a variety of platforms: iOS, Android, and Windows Phone. This chapter concludes the demonstration by writing the same application with HTML5 and JavaScript, and then using PhoneGap to turn it into a native application.

### Screens and Navigation

When you build an HTML5 application—especially an HTML5 application that will be transformed into a native mobile application, you should decide up front whether to design for a Single-Page Interface (SPI) or an interface with multiple pages. As discussed in Chapter 4, "Building Mobile Websites," an SPI application is based on a single HTML page whose sections are managed dynamically so that it serves the appropriate visuals to the user at the appropriate times.

An SPI application has essentially no true navigation between pages. The URL remains the same; which means that when mobile users tap the Back button, they are taken back to the previous application by default. There's a lot you can do to prevent this from happening. For example, you can implement your own strategy for linking and history navigation. The point is that controlling navigation is up to you.

A library such as jQuery Mobile takes care of much of this burden and also allows you to have an SPI application with embedded pages. Chapter 5 discussed this; in that chapter, you saw how jQuery Mobile allows you to treat a DIV element with a particular *data-role* attribute as a plain page.

This sample application uses different pages. It moves from one page to another using the *changePage* function of jQuery Mobile, as well as plain links.

Figure 11-8 provides a view of the three screens of the sample Guess application. The following code generates the main screen of the application:

```
<div id="inputPanel">
    <div>
       <table class="inputForm">
            <tr>
                <td class="input.tdLabel"><label id="inputPlayerLabel"></label></td>
                <td class="input.tdInput"><input type="text" id="inputPlayer" data-role="none" >
                </td>
            </tr>
            <tr>
            <td class="input.tdLabel"><label id="inputLevelLabel"></label></td>
            <td class="input.tdInput">
                <select id="inputLevel" data-native-menu="false" data-mini="true">
                    <option id="Basic" value="0">Basic</option>
                    <option id="Medium" value="1">Medium</option>
                    <option id="Super" value="2">Super</option>
                </select>
            </td>
        </tr>
        </table>
    </div>
    <div>
        <input type="button" id="buttonStart" data-role="none" />
    </div>
</div>
```

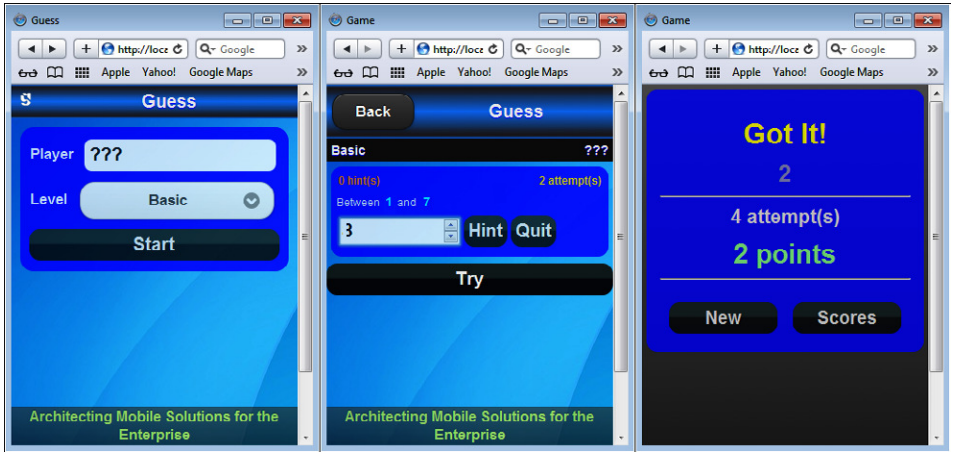



**FIGURE 11-8** The Guess application as it appears in Safari for Windows.

The user interface is plain HTML5 and CSS. The drop-down list with levels is rendered in a way that resembles native mobile widgets using the jQuery Mobile transformations. In this regard, using the *data-native-menu* attribute is key. In contrast, the *data-mini* attribute just attempts to keep the visuals as tiny as possible.

## Styling the Screen

To add color and layout information in a HTML5 user interface, you use CSS. Browsers aligned with the HTML5 standard offer a great support for CSS, but not in a truly uniform manner.

Most basic features of CSS3 (i.e., rounded corners, text effects, and backgrounds) are supported across browsers. However, don't be too surprised to see that some embellishment effects, such as gradients, aren't supported in all browsers once you port the HTML application to a given platform. Overall, I see this as a minor problem. Currently, I experienced trouble with gradients on Windows Phone 7 and 7.5. These issues will disappear as the new version of the operating system with a brand new browser arrives. However, for HTML5 and CSS3, coverage of what you can expect in mobile browsers can be found by visiting the Mobile HTML5 site (*http://mobilehtml5.org/*).

Because CSS is a key technology in enabling you to skin pages, you might want to put some effort into attaching CSS styles to page elements in an unobtrusive way. As much as possible, you should avoid placing style information inline. The following code should be considered as bad practice and avoided (though we all know that dirty tricks are sometimes necessary):

```
<div style="background-color: #fff; width: 70%; ...">
  ...
</div>
```

It is largely preferable to use CSS classes, as shown here:

```
<div class="inputPanel>
   ...
</div>
```

A CSS class is simply a way to group a bunch of CSS settings, as here:

```
/*
   This defines inputPanel as a CSS class and these settings are applied to
   all elements styled with class="inputPanel" in the markup.
 */
.inputPanel
{
    margin: 10px;
    padding: 10px;
    border-radius: 15px;
    background-color: #00f;
    opacity: 0.8;
}
```

You should use a CSS class when there are good chances to reuse it across pages and elements. To style individual elements, you don't use the class attribute at the element level, but in the CSS file, you point settings directly to an element with a given ID. This code shows how to do that:

```
/*
   This defines settings that are automatically applied to any elements
   in pages with an ID of inputPanel. There's no need to add anything to page
   elements beyond the ID attribute. Styling will happen automatically.
 */
#inputPanel
{
    margin: 10px;
    padding: 10px;
    border-radius: 15px;
    background-color: #00f;
    opacity: 0.8;
}
```

Proper use of CSS techniques helps a lot to keep the code clean and readable and, more important, makes it easy to edit and skin further.

## Testing Logic and Markup

An HTML5 application that must be transformed into a web-based mobile application is hard to test outside emulators and real devices. However, you can achieve a good approximation of the outcome by testing the HTML on a variety of desktop browsers—Chrome, Safari, and Internet Explorer 9 in particular.

Although such testing is only the first step, if something goes wrong or doesn't show up well here, it's unlikely that it will work smoothly on the device. I find that this Chrome, Safari, and Internet Explorer 9 browser combination is effective for gaining an idea of how the markup will behave on Android, iOS, and Windows Phone platforms, respectively.

Working on desktop browsers (with a properly resized window) allows you to fix major snags and move to the next step—integrating logic and markup with PhoneGap scripts and in a platform-specific PhoneGap project—comfortably.

**Note** I don't use emulators much for testing HTML5 markup that is expected to go into a PhoneGap application. I find these techniques much more effective when you are developing a mobile site. It's quite possible that you may have to tweak the markup of a PhoneGap application for specific device browsers, and that using emulators doesn't add much value—but of course, that's just my experience. Emulators tend to be slow (i.e., Android), and desktop browsers often are equipped with some handy plug-ins that help when debugging. To test native feature of the phone (such as contacts, the camera, and so on), you need to use a device emulator or a real device.

As you'll see in a moment, to create a PhoneGap application, you need to create a native project for each of the platforms that you intend to address. You may have to tweak the original HTML that you tested on desktop browsers or emulators, and you may even need some changes to your code.

The wide spectrum of Android devices is problematic for PhoneGap applications. You may find Android 2.*x* running on small devices with a 320 × 240 resolution, as well as larger smartphones that

have a 800 × 480 or higher screen resolution. As described in Chapter 9, "Developing for Android," the Android SDK suggests that you address this problem by sizing visual elements in device-independent pixels. You don't have such facilities in HTML—though you can probably use percentages to get close (at the cost of making the markup and CSS considerably more complex to handle). If you're targeting Android from a mobile site, you can use a device description repository (DDR) like Wireless Universal Resource FiLe (WURFL; see Chapter 6 for more about this) to decide which markup to serve to which class of devices. In this way, you can treat small- and large-screen Android devices differently.

Nothing of the kind is possible when you use PhoneGap. The same application will run unchanged on any Android device, regardless of the screen size. For this reason, I recommend you test the markup of Android applications thoroughly, particularly on small screens, early in the development process so that you can apply any needed changes to the overall layout. Figure 11-9 shows the game screen of Guess, as it appears compiled as a native Android application on a low-level device running Android 2.2, with a screen resolution of 240 × 320.

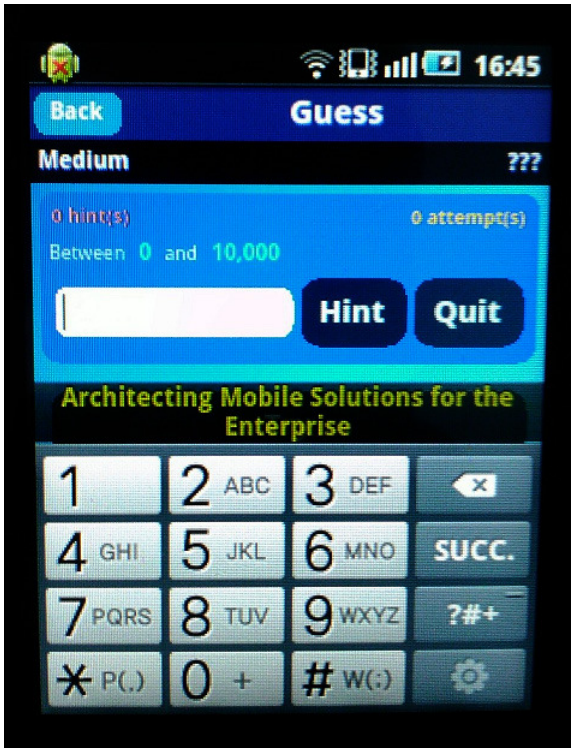



**FIGURE 11-9** The Guess PhoneGap application running on a small Android device.

As you can see, when a user is entering a number, the soft keyboard appears in the foreground, but ends up covering the Try button—which is the primary button the user might want to tap at that stage. So then the user is forced to tap the device's Back button to hide the keyboard and then tap again to set the focus and enter a new number. That's not really a pleasant experience. This might suggest you should swap the position of the Try button and the Hint/Quit pair of buttons.

Before we look into porting the HTML5 application to PhoneGap, it's worth spending a bit of time reviewing some Ajax points, because Ajax is a technology that you likely will use in your HTML5 solution to take data from remote sources.

## Ajax and Cross-Domain Issues

When it comes to client HTML pages, the Same Origin Policy (SOP) applies. Simply put, SOP refers to the default settings of most browsers, which simply refuse to allow cross-domain calls via JavaScript and Ajax. So long as you place Ajax calls to URLs located in the same domain as the originating page, everything works just fine. Otherwise, the browser applies its security measures and denies the call.

There are various ways to work around this issue. For example, you can set up a proxy service in your web server—the same server that served the current page to the browser—and use that as a router for reaching external sites outside the domain. So long as you can exercise some control over the target site, you can use JSON with Padding (JSONP).

JSONP assumes that the server accepts an extra string parameter on the call and treats that parameter as the name of a JavaScript function that will receive the JSON response. For example, suppose that the following URL returns a JSON string:

```
http://someserver/method
```

For completeness, let's assume that the JSON string has the following format, where *xxx* may be a randomly generated number:

```
{ Number:xxx }
```

Suppose also that the web application behind the URL supports JSONP and defines *jsCallback* as the extra parameter name for clients to request JSON paddings. The name *jsCallback* is arbitrarily defined by a web application that intends to support JSONP callers.

A caller will then invoke the following URL:

```
http://someserver/method?jsCallback=myLocalFuncToProcessJson
```

The response that JSONP-enabled web application returns is the following string:

```
myLocalFuncToProcessJson('{Number:xxx')
```

The *myLocalFuncToProcessJson* function is defined locally on the caller page and will be invoked automatically to process the response. Clearly, the name of the *myLocalFuncToProcessJson* function is also arbitrarily defined by the page author.

The JSONP protocol takes advantage of the fact that the *SCRIPT* tag is not subject to cross-domain restrictions, so the following markup has the effect of downloading a script expression that invokes a local function. Because browsers process script right after downloading, the JSON response padded in a function call is processed immediately:

```
<script src="http://someserver/method?jsCallback=myLocalFuncToProcessJson" />
```

Some JavaScript libraries (specifically jQuery) have their own wrappers that create the above SCRIPT tag on the fly, making it invisible to developers. The following code shows what it takes to place a JSONP call with jQuery:

```
var url = "...";
$.getJSON(url + "?js=?", null,
         function (data) { showData(data) });
```

The *url* variable is completed with a query string parameter referring to the agreed parameter that carries the name of the wrapper JavaScript function. In jQuery, however, you are not required to indicate the name of the local wrapper function. You just use the *?* symbol as a placeholder for the name of a function that jQuery generates on the fly. This dynamically generated function will be calling into the inline function that you provide with the call. The net effect is that the downloaded JSON data is passed to the unnamed function that you indicate in the call.

**Note** Cross-Origin Resource Sharing (CORS) is the name of a W3C draft that discusses an official way to make cross-domain calls opt in for servers instead of leaving the decision to browsers. Currently, the latest versions of most popular browsers have some form of support for CORS. For more information, see *http://www.w3.org/TR/cors/*.

What about cross-domain issues and HTML pages hosted in a PhoneGap application? Native mobile applications, like desktop applications and server-side web applications, are not subject to cross-domain restrictions. But what about a PhoneGap application that is a native application using a web view to display its content?

The good news is that there are ways to bypass cross-domain restrictions in PhoneGap compiled applications. This means that you have a way to use Ajax calls from script code freely to call any remote URLs. According to the PhoneGap wiki, this is possible because HTML pages are invoked internally using the *file://* protocol instead of *http://,* and for this reason, the restriction doesn't apply. There are, however, a few things to note.

First and foremost, you still need to claim that you're using the Internet for your mobile project. This must be done for each platform in accordance with what was discussed in Chapter 8, "Developing for iOS," for iOS; Chapter 9, "Developing for Android," for Android; and Chapter 10, "Developing for Windows Phone," for Windows Phone. As PhoneGap supports additional platforms, too, you might want to check out details for each of them.

Most mobile browsers automatically cache the response received for any Ajax calls. If you own the server, then you might want to try setting a *no-cache* header in the response to prevent this. Otherwise, adding a random number to each request is an effective trick. See the following code:

```
// Adding a variable number to a URL
var url = "http://www.expoware.org/services/random/get?t=" + new Date().getMilliseconds();
```

This example used the number of milliseconds; you can use a randomly generated number for more reliably random results. By the way, you can use the URL shown here to my personal website

for some tests. This URL returns a random number between 0 and 1,000. You should be able to make calls to this URL from any HTML5 page compiled to a native PhoneGap-based application.

**Important** Cross-domain calls work only when you test the application from the real device. To test from the desktop computer, you can try a couple of ways. One entails using Safari for Windows on the root file of your site accessed via the *file://* protocol. Safari is unique in that it allows you to place Ajax calls from a *file://* loaded page. Another way to do this is by using a tool like Sleight, which operates as a web server that is local to the directory of HTML pages in such a way that it intercepts resources being invoked from pages in the directory and proxies requests to a configured remote server for all resources that aren't located locally. Developed by Andrew Lunny, Sleight is a Node.js application that complements the PhoneGap platform. You can get it from *http://github.com/alunny/sleight*.

If you're using jQuery Mobile, then you should enable the framework to place cross-domain calls. By default, jQuery Mobile doesn't place cross-domain calls unless you change the value of the *cors* setting in the *mobileinit* event as follows:

```
<script type="text/javascript">
    $(document).bind("mobileinit", function () {
        $.support.cors = true;
    });
</script>
<script type="text/javascript" src="Content/Scripts/jquery.mobile.min.js"></script>
```

Note that you should bind the *mobileinit* event before you link the jQuery Mobile library. This should be done for every page.

In recent versions of PhoneGap, a feature called *white-listing* has been enabled for some platforms—notably iOS and Android, but not Windows Phone yet. *White-listing* refers to holding a list of websites that your application is enabled to call via cross-domain calls. If the white-list is configured and a site is not listed, then PhoneGap will deny the request. In iOS, the list of sites is found in the Phonegap.plist file. In Android, it is saved as a Phonegap.xml file in the Res/xml project folder.

# Integrating with PhoneGap

Once you have a reliably working website made of local resources—HTML pages, CSS style sheets, images, and JavaScript files—you are ready to make it run natively on a variety of platforms that include, but not limited to, iOS, Android, and Windows Phone 7.5.

## Supported Platforms

As of version 1.5 of PhoneGap, the list of supported platforms is as detailed in Table 11-1 (arranged in descending order by importance). For an updated support matrix, add a reference to *http://phonegap.com/about/features*.

**TABLE 11-1** Platforms Supported by PhoneGap 1.5

| Platform | Basic Requirements |
|---|---|
| iOS | You must have a computer running Mac OS X Snow Leopard (10.6) and must be a registered Apple developer. The resulting code works on devices capable of supporting iOS 3.0 and newer versions. With the exception of Compass on iPhone 3G, all PhoneGap device-related features are supported. |
| Android | You can write Android applications using PhoneGap for Android 1.6 and later. All PhoneGap device-related features are supported on Android. |
| Windows Phone | You can target Windows Phone 7.5 and must be a registered Windows Phone developer. All PhoneGap device-related features are supported on Windows Phone. |
| BlackBerry | Even though you can use PhoneGap to create applications for BlackBerry 4.6 and 4.7, it is recommended that you start with BlackBerry 5, which is where support for most device-related features has been added. PhoneGap doesn't offer BlackBerry support for Compass and Media. |
| webOS | No support for compass, file, media, and contacts. |
| Bada | A recently added platform, it doesn't yet offer full support for the features that might be available on the phone. |
| Symbian | No support for compass, file, and media. |

As PhoneGap is merely a way to package a web application into a native shell of code, you need to be ready to manage a different project for each platform you intend to support.

## Building a PhoneGap Project

To create a PhoneGap application for any given platform, you need a native project and native SDKs and programming tools. For example, this means that for iOS development, you must have a Mac computer, be a registered Apple developer, and have Xcode and the iOS SDK installed on the computer.

Once you have a native empty project up and running, you take the next step of adding the web source files and connecting all the wires. Although the overall procedure is relatively seamless, don't think that you will never face issues.

### Getting Ready for PhoneGap Applications

Your original set of HTML pages may need some tweaks to run as a PhoneGap application.

Frankly, it's hard to say whether these changes that you may need depend on the device web-view component, the PhoneGap framework, the JavaScript framework that you may use, your JavaScript code, or perhaps a combination of all these things. Fact is, don't be surprised if you need to adjust your web application before it runs just fine as a native PhoneGap-powered application. The following is a list of changes that I experienced using jQuery Mobile as a JavaScript framework.

In particular, paths to auxiliary resources and pages should be relative and not have trailing slashes. Also, it wouldn't hurt if you can ensure that case-sensitivity in names is respected. In addition, if you're using jQuery Mobile, then you may need to make some manual changes here and there to

ensure that the CSS works as expected. A common snag for me is setting a background image for a page. So far, I haven't been able to set one, except through some inline style information:

```
div id="homepage" data-role="page" style="background:url('background.jpg')">
```

The image file should be available in the root folder of the web resources.

Also, be aware that some further changes may be required to your JavaScript code to address possible subtle differences between browser engines—this may not happen every time, but it is something you can expect to see.

Finally, you may add to your script code any additional code that has to do with device capabilities—contacts, vibration, media, accelerometer, camera, and geolocation, for example—as well as handlers for device-specific events and navigation. PhoneGap, in fact, offers its own API to deal with back events. This is code that you need to deal with and test live on a device using the first platform you compile for. Once you have come to a successful configuration of JavaScript, HTML, and CSS, you can hope that it can be ported to other platforms in nearly no time.

Let's see a few examples.

## Building a iOS Application

Installing PhoneGap for iOS creates a new project template right in the Xcode environment, as shown in Figure 11-10.


**FIGURE 11-10** PhoneGap projects in Xcode.

All you do is pick up a PhoneGap project, and then a sample application is created for you in Xcode. After you compile the application for the first time, a Www folder is created in the project. Next, you copy your web application over it. Finally, you are ready to test your new iOS application. The Www folder should act as the root of the site. PhoneGap defaults to the Index.html file as

the entry point to the application. However, a HTML5 application becomes native because in the PhoneGap project (on every platform), there's some startup code that loads a web view into the main screen. If you like to use a different page to open the application initially, you just change the URL being passed to the web view.

As mentioned, the iOS environment is where PhoneGap applications do their best. The steps for creating PhoneGap applications for iOS are really just the ones that have been discussed. For more information on iOS compilation and packaging, refer to Chapter 8.

Figure 11-11 shows the Guess application in action. Overall, the speed and responsiveness is really good.


**FIGURE 11-11** The PhoneGap Guess application running on iOS.

## Building an Android Application

To set up an Android PhoneGap project, the first step to accomplish is linking the PhoneGap JAR file—the Java PhoneGap library—to the project, which you must do manually. You can use both Eclipse and IntelliJ IDEA for this. Then you create a new Www folder under the Assets project folder and fill it with your web resources. The Www folder is the root of the web application that PhoneGap will render natively to Android.

Next, after adding the JavaScript file (Cordova.js in version 1.5) to the web resources, and after linking it from pages, you ensure that you have a file named Plugins.xml located under the Res/xml

project folder. If it doesn't yet exist, create an Xml subfolder. You can get this file from the sample Android project that comes with the PhoneGap package. Here's the default content of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<plugins>
    <plugin name="App" value="org.apache.cordova.App"/>
    <plugin name="Geolocation" value="org.apache.cordova.GeoBroker"/>
    <plugin name="Device" value="org.apache.cordova.Device"/>
    <plugin name="Accelerometer" value="org.apache.cordova.AccelListener"/>
    <plugin name="Compass" value="org.apache.cordova.CompassListener"/>
    <plugin name="Media" value="org.apache.cordova.AudioHandler"/>
    <plugin name="Camera" value="org.apache.cordova.CameraLauncher"/>
    <plugin name="Contacts" value="org.apache.cordova.ContactManager"/>
    <plugin name="File" value="org.apache.cordova.FileUtils"/>
    <plugin name="Network Status" value="org.apache.cordova.NetworkManager"/>
    <plugin name="Notification" value="org.apache.cordova.Notification"/>
    <plugin name="Storage" value="org.apache.cordova.Storage"/>
    <plugin name="Temperature" value="org.apache.cordova.TempListener"/>
    <plugin name="FileTransfer" value="org.apache.cordova.FileTransfer"/>
    <plugin name="Capture" value="org.apache.cordova.Capture"/>
    <plugin name="Battery" value="org.apache.cordova.BatteryListener"/>
</plugins>
```

The Plugins.xml file contains the list of extensions being used to enrich the JavaScript environment. If you create your custom extensions of native code to add a new feature or replace an existing one, you register your extension within this file.

Finally, you should edit the source code of the root activity as follows:

```
public class MyActivity extends DroidGap
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        super.loadUrl("file:///android_asset/www/index.html");
    }
}
```

In particular, you should inherit the activity from *DroidGap* and make it call the *loadUrl* method to load the root file of your local web application. You don't usually need additional activities because all your logic is expected to be in the HTML pages.

Note that the Android emulator may sometimes be too slow, and it may cause a timeout. To avoid that, you can focus your testing on a real device. But there's an alternative, which is accomplished by the following code:

```
super.loadUrlTimeoutValue(milliseconds);
```

In fact, you can even increase the timeout of the *loadUrl* method programmatically. Figure 11-12 shows the sample application on an advanced Android smartphone.

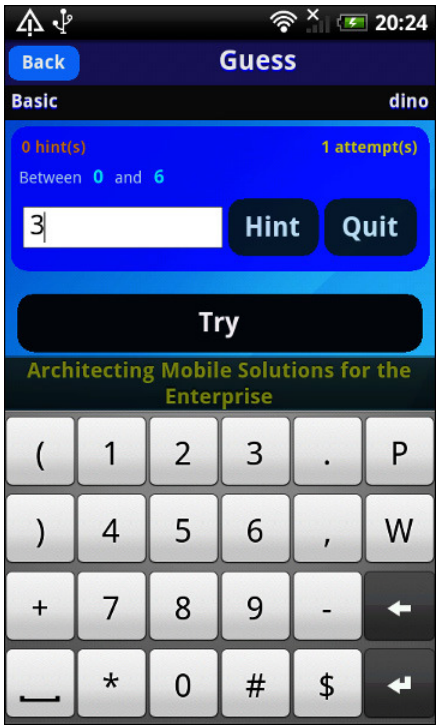



**FIGURE 11-12** The Guess web application packaged as a native Android application.

## Building a Windows Phone Application

PhoneGap offers a relatively easier experience when used within Visual Studio to create applications for Windows Phone. When you download the package, you get a ZIP file to install manually in the Visual Studio project templates. In particular, you need to copy the project template file (in PhoneGap 1.5, it's called Cordova-1.5.0-Starter.zip) in the following folder:

```
C:\Users\[Name]\My Documents\Visual Studio 2010\Templates\ProjectTemplates\Silverlight for
Windows Phone
```

As a result, you get a brand-new item in the list of Visual Studio projects, as shown in Figure 11-13.

There's no need for you to change anything in the project—you build it and it gets you a "Hello-World" style application. The next step is simply replacing the content of the Www folder with the files of your local web application.

Creating a PhoneGap application for Windows Phone 7.5 requires some changes and may generate a few additional issues than with iOS and Android. A common snag is that some resources—mostly images—appear to be missing when you start the application. To fix this, ensure that all resource files have a build action of *Content* and are copied to the output directory if they are newer than existing files. You control these settings from the Properties box in Solution Explorer.

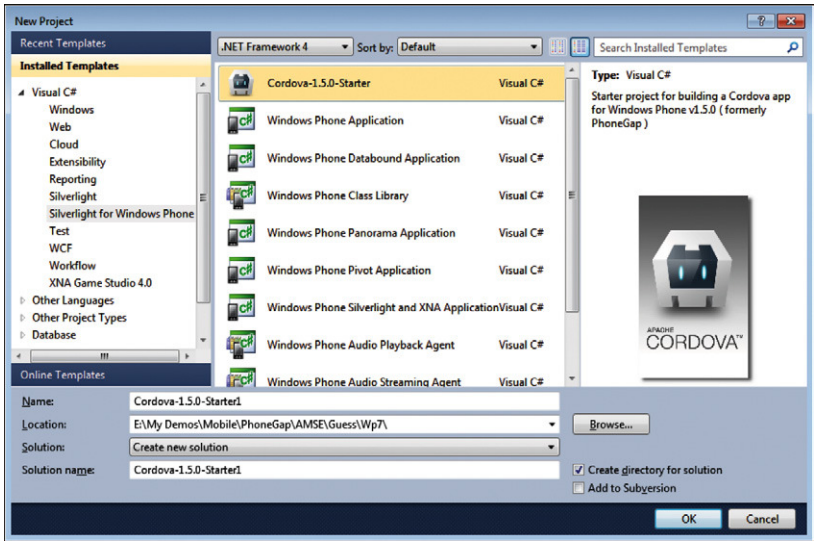



**FIGURE 11-13** Creating a Windows Phone project with PhoneGap 1.5.

Finally, the web-view component in Windows Phone 7.5 doesn't support the JavaScript *alert* function. To avoid replacing all occurrences of *alert* in your JavaScript source code, you might want to use the following code in just one global place:

```
GLOBALS.appInitialize = function () {
   ...
   window.alert = navigator.notification.alert;
}
```

In my sample code, this line is in the function that I use to initialize the application. Figure 11-14 shows the sample application compiled natively for Windows Phone 7.5.

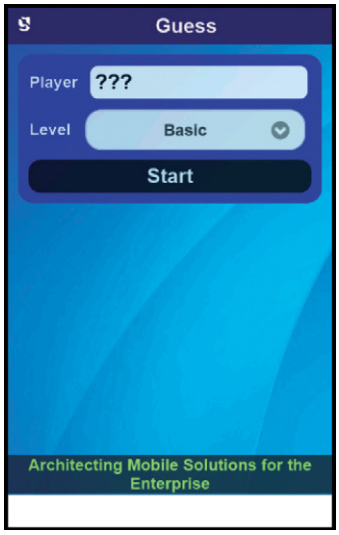



**FIGURE 11-14** PhoneGap Guess for Windows Phone.

## Final Considerations

Developing for PhoneGap is easy and difficult at the same time. It is easy because most of what you do consists of writing an HTML5 application. Because web skills are fairly common these days, there's likely no need to learn new languages and gain new skills. Compared to learning new languages, such as Objective-C, and mastering new SDKs, it seems faster and more comfortable. Is this really true?

### The Common Denominator

As PhoneGap uses HTML5 and web development, it can't specifically address the native features of any platform. Using the Menu button in Android or the Back button in Windows Phone requires tricks and extensions to the basic HTML source code. Support for the Back button is particularly problematic.

On platforms that recognize it (e.g., Android and Windows Phone), the Back button is designed to let the user navigate through the screens of the application. The Back button and related navigation is managed by the operating system, which exposes a proper API to applications. When the user interface is HTML-based, the user interface is hosted in a web-view component; in a web world, the Back button is used to navigate through hosted pages. When Ajax is used to provide smoother and faster navigation, no new URL is necessarily added to the history, so tapping the Back button once may exit the entire application.

History and navigation is managed by libraries such jQuery Mobile, and PhoneGap offers facilities; as a developer, you have the responsibility of testing and debugging. It may be a bit annoying, and it certainly adds more time to the development process.

### The Skin Factor

PhoneGap doesn't just unify the behavior and user experience of an application under various platforms; it also unifies the user interface. Because the user interface of a PhoneGap is based on CSS, by simply replacing the CSS, you can skin the same HTML pages differently. Some popular JavaScript libraries—specifically jQuery Mobile—offer a native iPhone-like user interface that you may not want to have on Android or Windows Phone.

Your design team (or CSS that you can obtain somewhere on the Internet) can probably create an ad hoc stylesheet that you just replace in the pages when creating the project for a given platform. This is a first step that dispels the dream of having a single codebase when you use PhoneGap. There's more, however.

The user interface intended as styles of controls and text is just one aspect of a mobile platform. A different CSS may help, but to make a PhoneGap application look like a native one, more changes must sometimes be made to the HTML layout—and sometimes to the JavaScript code as well.

### PhoneGap Is Not the Silver Bullet

PhoneGap is an excellent choice when creating applications with a completely customized user interface, such as games or applications that, for their purpose, require a handcrafted user interface.

Prototyping the user interface with HTML is certainly quicker than using native tools on most platforms—with perhaps the exception of Windows Phone and Microsoft Expression Blend. The results, however, is hardly as fast as a truly native application. Speed is the first aspect of PhoneGap applications that users may notice. On the other hand, there's not much that you can do to make a PhoneGap application run faster. The speed of PhoneGap applications depends for the most part on the browser, and the browser depends on the specific device. Especially on Android, this may be a problem.

As mentioned earlier, if you're writing a mobile site, you can use server-side capability detection to optimize the markup being served. With native applications, this is much harder to implement. It's not that having alternate markup in HTML5 is not possible, but it makes PhoneGap development much harder.

In the end, developing for PhoneGap is only apparently quick. If you have to produce applications for multiple platforms, using PhoneGap may save time on the whole. Overall, however, it's like that old adage—nine women can't deliver a baby in one month. Similarly, PhoneGap can't deliver applications for three different platforms in a fraction of the time.

With PhoneGap, you have to maintain multiple projects, deal with subtle browser and web-view differences (e.g., there is no *alert* function in Windows Phone) and, more important, debugging is hard. Often, debugging ends up being a series of calls to message boxes.

The native PhoneGap application hasn't really got much to test, so the crucial point is to test the HTML5 application effectively. As mentioned earlier, my favorite option is using a desktop browser and any facilities that you may have on the desktop, such as a Visual Studio–integrated debugger, Firebug, and similar tools. When the application is doing well on HTML5-compliant desktop browsers, and you have tested it with phone-specific features that require an emulator, you can build native applications. From now on, debugging on the device message boxes is probably the only way to go at the moment.

Documentation is good, but keep in mind that the business model behind PhoneGap is centered on training. PhoneGap is free; training and support is what generates revenues for Adobe. Keep this in mind before you draw the line and make a decision.

## When to Use PhoneGap

So does this mean that PhoneGap is not worth the cost? Well, having PhoneGap is much better than not. However, I believe that the right way to look at PhoneGap is to consider it as an interesting option, not a silver bullet.

Using HTML5 and script to arrange an iOS application under PhoneGap is an approach that works very well. I believe this is due to a couple of reasons: excellence of Safari Mobile as a browser (overall speed and support for HTML5) and relatively uniform hardware capabilities of devices. In addition, jQuery Mobile and other JavaScript libraries tend to offer an iOS-oriented user interface natively which, of course, is good for developers of iOS applications.

Android development is problematic overall, regardless of the frameworks that you use. The fragmentation of devices makes the choice of using HTML harder to fine-tune for optimal results than native or mobile website development. On average, the performance of the Android browser is acceptable, but often not as good as on iOS. Proper handling of the Back button is a major issue when it comes to PhoneGap for Android.

If you only have a small amount of .NET background, I wouldn't consider PhoneGap an option for building Windows Phone applications. Moreover, even if you never used .NET, I'd also suggest you take a close look at native tools before you opt for PhoneGap Windows Phone development. It may get better in the future, but my gut feeling is that the combination between PhoneGap, the Windows Phone web-view component, and JavaScript libraries for nice page transitions and touch gestures is a sort of alchemy. In the end, building an effective Windows Phone application that runs and loads slower than a native application will cost you nearly the same in terms of development time than a truly native application.

What about other platforms such as BlackBerry, Bada, and Symbian? If these are just a few more applications that you add to complete the suite of products, you probably can consider using PhoneGap for relatively quick development and to provide an adequate native user experience. However, if Symbian, Bada, or BlackBerry applications are critical in your mobile strategy, then I would recommend that you carefully consider going with truly native solutions before you opt for PhoneGap.

**Note** It may be worth stating, as a final point, that when I started on this book project in the fall of 2011, I had very positive feelings about PhoneGap. Then a serious project came in, and after having a good deal of trouble finding the right mix of JavaScript, PhoneGap, and HTML5, we opted for hybrid native applications—native skeletons and web views to display server-side HTML. Certainly, I'm not saying that this approach replaces PhoneGap; but, on the other hand, each project is different. Having more available tools means more choices—but sometimes also more chances to make the wrong decision!

## Summary

Probably the biggest source of trouble for mobile developers is building the presentation layer. A platform-specific user interface must be built for any operating system. This entails learning about the intended role of hardware buttons (e.g., the Search button is intended differently in Android and Windows Phone); it entails using and programming native widgets, and giving the application a navigation system that is coherent with the platform's guidelines.

Each platform has its own set of tools for building the user interface. You have designers in iOS and Windows Phone; a designer also comes with Eclipse for Android. The power of designers is not the same, though. And building a complex user interface is far easier and quicker for Windows Phone than for iOS, Android, or BlackBerry.

Not all apps follow the official UI guidelines of their target platform; some applications, even very successful ones, create their own customized user interfaces (custom navigation bars, tabs, buttons, lists, etc.). In this case, the argument of having a native application for having the native user interface is not valid; and actually, a unique HTML-based custom user interface can make sense. It should be clear that this approach may not work for all applications—quite the opposite, in fact. But when taking this route is acceptable, then PhoneGap helps immensely—possible performances issues aside, of course.

With HTML5, you can focus your UI efforts on the building of a unique experience for a variety of platforms. HTML5, CSS, and JavaScript are popular technologies, but they are also fairly easy for newcomers to get. PhoneGap uses these skills and adds a shell of native code around an HTML codebase to package a native application that runs in the device browser.

It's effective, and often much quicker, to develop than a native solution. But it's not free of issues as far as the final user experience is concerned. Any PhoneGap application will be evaluated on its own merits when it comes to being approved for the Apple App Store or Windows Marketplace, regardless of the underlying technology used to write it. However, this just transfers the responsibility of building a great application to you. Sounds good, but isn't this just what developers are paid for? Yes, but make sure that you pick up the right tool for the task.

# Index

## Symbols



## A

## B

# C

## D

# E



# F

## G



## H

## I

## J

## K



## L

# M

# N

## O

## P

# Q



# R

## S

# T

# U

# V



# W

# X

## Z

# About the Author

A longtime trainer and top-notch architect, Dino Esposito is the author of many popular books for Microsoft Press that have helped the professional growth of thousands of .NET developers. His latest books are *Programming ASP.NET 4* and *Programming ASP.NET MVC3,* which have been translated into a variety of languages. Every month, at least five different magazines and websites throughout the world publish Dino's articles, which cover topics ranging from web development to software design practices, and from mobile development to ASP.NET Model-View-Controller (MVC) and social network development.

An ASP.NET Most Valuable Professional (MVP), Dino is available for onsite consulting and training on web and mobile development and software practices. When traveling, Dino is often the guest star of user-group meetings in Europe. If you run a user group, feel free to get in touch.

In the rest of his everyday working life, Dino is the CTO of Crionet (*http://www.crionet.com*), a fast-growing company providing software and mobile services to professional sports, especially tennis. Dino led a team that created a range of mobile apps for Android, iOS, Windows Phone, and BlackBerry, such as the official app for the Rome ATP Masters 1000 tournament. Dino also contributed to the popular (and multiplatform) Postino app (*http://www.postinoapp.com*) for sharing real postcards from mobile pictures, and writes for the Mopapp technical blog (*http://www.mopapp.com*).

Dino speaks regularly at industry conferences all over the world, including Microsoft TechEd, DevConnections, and premiere European events such as DevWeek, Software Architect, and BASTA. He is fairly active on social media; you can follow Dino on Twitter as @despos, and read his mobile blog at *http://www.mopapp.com/blog*. The blog focuses on a wide range of mobile-related topics, including native app planning and development, sales monitoring, patterns and strategies for the various platforms, mobile site development, responsive Web design, smart TV programming, HTML5, and appstore interactions.

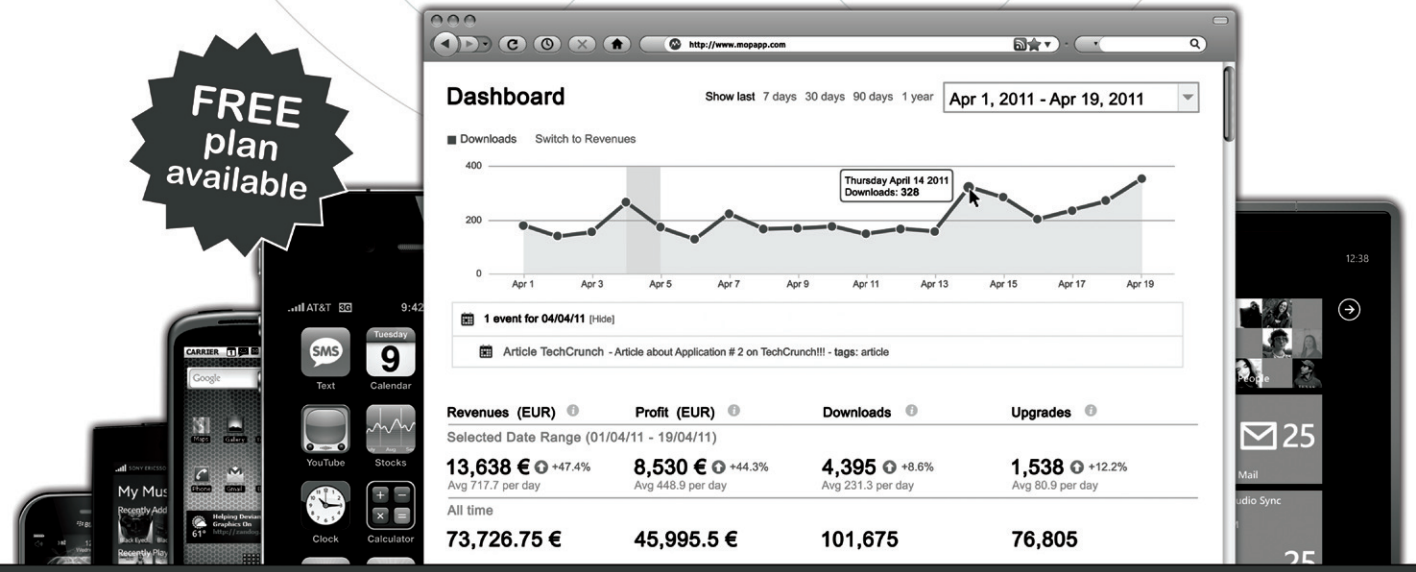
Finally, Dino makes every reasonable effort to become a better domain expert in tennis. This means watching tennis live and on TV, and planning new applications—but especially playing tennis on dusty clay courts at CT Monterotondo, in Monterotondo, Italy.

# mopapp

**The only enterprise-level solution for mobile apps analytics**

- import & export sales and data through Mopapp API
- licensing option: match your company's look & feel and use your own sub-domain
- tailor-made import of legacy data
- track unlimited apps and in-app items
- automatic integration with all major app stores
  - Handango
  - WP7 Marketplace
  - Amazon Appstore
  - Samsung Apps
  - GetJar
  - Barnes & Noble
  - MobiHand
  - iTunes App Store
  - RIM App World
  - Google Play
- lots of reports with advanced filtering and grouping
- monitor user reviews translated to your own language
- automatic integration with all major ad networks
- export to Excel and PDF
- subaccounts & report sharing

FREE plan available



# What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

**microsoft.com/learning/booksurvey**

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press