Quick answers to common problems

Delphi Cookbook

50 hands-on recipes to master the power of Delphi for cross-platform and mobile development on Windows, Mac OS X, Android, and iOS

www.allitebooks.com

Delphi Cookbook

50 hands-on recipes to master the power of Delphi for cross-platform and mobile development on Windows, Mac OS X, Android, and iOS

Daniele Teti

BIRMINGHAM - MUMBAI

Delphi Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2014

Production reference: 1190914

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-958-9

www.packtpub.com

Cover image by Junaid Shah (junaidshah111@gmail.com)

www.allitebooks.com

Credits

Author
Daniele Teti

Reviewers
Eric Van Feggelen
Sherwin John
Olivier Pons
Jorge H. Rodriguez

Commissioning Editor
Sam Birch

Acquisition Editor
Richard Harvey

Content Development Editors
Balaji Naidu
Pooja Nair

Technical Editors
Mrunal Chavan
Dennis John
Edwin Moses

Copy Editors
Roshni Banerjee
Adithi Shetty

Project Coordinator
Leena Purkait

Proofreaders
Bridget Braund
Paul Hindle
Amy Johnson

Indexers
Monica Ajmera Mehta
Tejal Soni

Graphics
Valentina D'silva
Abhinash Sahu

Production Coordinators
Aparna Bhagat
Manu Joseph

Cover Work
Aparna Bhagat

About the Author

Daniele Teti is a software architect, trainer, and consultant with over 18 years of professional experience. He writes code in a number of languages but his preferred language to compile native software is Object Pascal.

Daniele is a well-known Delphi and programming expert in the developer community. He's the main developer and drives the development of some Delphi open source projects (DelphiMVCFramework; DORM, "the Delphi ORM"; Delphi Redis Client; and so on). He wrote his first program when he was 11 years old, and since then, he happily continues to write software almost every day. Apart from Delphi, he's a huge fan of design patterns, expert systems, RESTful architectures, and Android OS. When he is not busy writing software or programming (as his job or hobby), he likes to play the guitar, write songs, and do voluntary activities. Currently, he works as an R&D Director & Educational at bit Time Software (`www.bittime.it`), an Italian representative of Embarcadero Technologies (`www.embarcadero.com`). He recently became the CEO of bit Time Professionals, which is a spin-off company of bit Time Software; this company specializes in consultancy, training, and development.

Being a software architect, consultant, and teacher for many Italian and European companies, he travels very often around the world. He is the Technical Director of ITDevCon, the biggest European Delphi conference (`www.itdevcon.it`). He's also an international speaker at many technical conferences.

Daniele lives in Rome, Italy (where each photographer becomes an artist) with his beloved wife, Debora, and their little boy, Mattia.

www.allitebooks.com

About the Reviewers

Eric van Feggelen is a passionate and experienced software consultant who delivers high-quality solutions using the latest technologies available. He has about 15 years of experience as a developer and has been interested in information technology his entire life. In the past few years, he worked for major corporations, such as Microsoft and Avanade and continues to serve the Microsoft Enterprise space as a private contractor for his own company. At the time of writing this book, Eric has worked as a lead developer for a Microsoft Dynamics start-up.

In 2013, Eric reviewed *Mastering Windows 8 C++ App Development*, *Packt Publishing*.

Olivier Pons is a developer who's been building websites since 1997. He is a teacher at IngéSup (École Supérieure d'Ingénierie Informatique – `http://www.ingesup.com/` and `http://www.y-nov.com`), at the University of Sciences (IUT) in Aix-en-Provence, France. At École d'Ingénieurs des Mines de Gardanne, he teaches state-of-the-art web techniques, such as MVC fundamentals, Symfony, WordPress, PHP, HTML, CSS, jQuery/jQuery Mobile, Node.js, AngularJS, Apache, Linux basics, and advanced Vim techniques. He has already worked as a technical reviewer for *Ext JS 4 First Look*, *jQuery Hotshot*, *jQuery Mobile Web Development Essentials*, *WordPress Complete*, and *jQuery 2.0 for Designers Beginner's Guide Second Edition*. All these books were published by Packt Publishing. In 2011, he left his full-time job as a Delphi and PHP developer to concentrate on his own company, HQF Development (`http://hqf.fr`). He currently runs a number of websites including `http://www.battlesoop.fr`, `http://www.krystallopolis.fr/` (which will be released soon), `http://www.livrepizzas.fr`, `http://www.papdevis.fr`, and `http://olivierpons.fr` (his own web development blog). He also works as a consultant, teacher, and project manager, and sometimes, helps big companies as a senior / highly skilled developer.

Jorge H. Rodriguez has a background in software development and more than 20 years of experience under his belt, many of them working with Delphi.

Always on the lookout for new and exciting technologies, Jorge lives to code and spends much of his spare time reading technical books and playing online chess. During winters, he likes to go snowboarding with his only son, Camilo.

Jorge resides in Vancouver, Canada, awaiting his Colombian girlfriend, Shana. He can be contacted at `delphi.developer@shaw.ca`.

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

Preface 1

Chapter 1: Delphi Basics 7

- Introduction 7
- Changing your application's look and feel with VCL styles and no code 8
- Changing the style of your VCL application at runtime 10
- Customizing TDBGrid 14
- Using the owner's draw combos and listboxes 20
- Creating a stack of embedded forms 23
- Manipulating JSON 26
- Manipulating and transforming XML documents 32
- I/O in the twenty-first century – knowing streams 38
- Putting your VCL application in the tray 42
- Creating a Windows service 48
- Associating a file extension with your application on Windows 53

Chapter 2: Become a Delphi Language Ninja 59

- Introduction 59
- Fun with anonymous methods – using higher-order functions 59
- Writing enumerable types 64
- RTTI to the rescue – configuring your class at runtime 68
- Duck typing using RTTI 72
- Creating helpers for your classes 76
- Checking strings with regular expressions 84

Chapter 3: Going Cross Platform with FireMonkey 91

- Introduction 91
- Giving a new appearance to the standard FireMonkey controls using styles 92
- Creating a styled TListBox 98

Impressing your clients with animations 102
Using master/details with LiveBindings 105
Showing complex vector shapes using paths 116
Using FireMonkey in a VCL application 122
Chapter 4: The Thousand Faces of Multithreading 129
Introduction 129
Synchronizing shared resources with TMonitor 129
Talking with the main thread using a thread-safe queue 137
Synchronizing multiple threads using TEvent 140
Displaying a measure on a 2D graph like an oscilloscope 143
Chapter 5: Putting Delphi on the Server 147
Introduction 147
Web client JavaScript application with WebBroker on the server 148
Converting a console service application to a Windows service 157
Serializing a dataset to JSON and back 160
Serializing objects to JSON and back using RTTI 165
Sending a POST HTTP request encoding parameters 171
Implementing a RESTful interface using WebBroker 174
Controlling remote applications using UDP 190
Using App Tethering to create a companion app 197
Creating DataSnap Apache modules 203
Chapter 6: Riding the Mobile Revolution with FireMonkey 213
Introduction 213
Taking a photo, applying effects, and sharing it 214
Using listview to show and search local data 222
Do not block the main thread! 227
Using SQLite databases to handle a to-do list 234
Using a styled TListView to handle a long list of data 239
Taking a photo and location and sending it to a server continuously 247
Talking to the backend 257
Making a phone call from your app! 264
Tracking the application's life cycle 269

www.allitebooks.com

Chapter 7: Using Specific Platform Features 275
Introduction 275
Using Android SDK Java classes 276
Using iOS Objective-C SDK classes 282
Displaying PDF files in your app 286
Sending Android intents 291
Letting your phone talk – using the Android TextToSpeech engine 300
Index 305

www.allitebooks.com

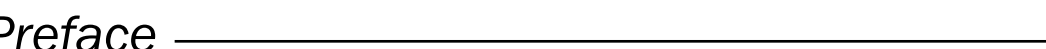
Preface

If you've been a software developer for a long time, you certainly know how useful a conversation can be with a colleague who already did something similar to what you are doing, and can discuss it as he/she may have faced the same problem. It is not possible to include all the possible situations that a developer may face in a book, but most problems are similar at least in principle. This is the reason this book is organized as a cookbook; just like how a combination of foods can be adapted and modified to be appropriate for different types of meals. A programming recipe can provide the idea to solve many different problems.

This book is an advanced-level guide that will help Delphi developers become experts in their every day job. The every day job and the quality of your deliverables is what contribute to the quality of your professional life. It does not make sense to reinvent the wheel repeatedly, especially when working with a well-established tool such as Delphi. The focus of this book is to provide readers with comprehensive and detailed examples on how effectively the Delphi software can be designed and written. All the recipes in this book are a result of years of development, training, and consultancy activities in the most different fields of the IT industry, from small systems with thousands of installations to large systems commissioned by any big company or government. It is not a magic book that will solve all your development problems (if you find it, tell me please!), but it can be a valid source of help to get a different point of view on a specific problem, or a hint on how to solve problems.

Armed with the knowledge of advanced concepts, such as high-order functions and anonymous methods, generics and enumerables, extended RTTI and duck typing, LiveBindings, multithreading, FireMonkey, mobile development, server-side development and many more, you will be pleasantly surprised as to how quickly and easily you can use Delphi to write high quality, clean, readable, maintainable, and extensible code.

I have read too many boring programming books, so I tried to maintain a relaxed and light exposition. A small applicability scenario, which describes a situation where a particular technology, approach, or design pattern can be used successfully, introduces all the recipes. The recipes are not too complex because otherwise the book may consist of thousands of pages; however, it is also not trivial because the IT books landscape is already full of trivial examples with a few direct applications. I tried to do a good trade-off and hope I succeeded.



Every time I start to read a new book, I ask myself, "Will the author have something interesting to say?", "How much will this book change my point of view on the topics it talks about?", or "Will it be worth the time to read this book?". Now that I'm on the other side of the river, I worked hard to put as much good quality content as possible in this book, which I hope will match your expectations.

On a final note, writing hundreds of pages about advanced programming is not an easy task. However, I am very pleased to have done it, and I hope you will enjoy reading it as much as I enjoyed writing it.

What this book covers

Chapter 1, *Delphi Basics*, talks about a set of general approaches that should not be ignored by any Delphi programmer. Some recipes are simple, while some are not, but all of them should be deeply understood. By the end of this chapter, you will be able to use some of the fundamental Delphi techniques related to the RTL, VCL, and OS integration.

Chapter 2, *Become a Delphi Language Ninja*, focuses on the Object Pascal language. A programming language is the way you talk to the machine, so you must be fluent and should know all the possibilities offered. This chapter talks about higher-order functions, practical utilization of the extended RTTI, regular expressions, and other things useful to augment the power of your code and lower the amount of time spent on debugging.

Chapter 3, *Going Cross Platform with FireMonkey*, is dedicated to the FireMonkey framework in general. What you will learn from this chapter can be used in many of the platforms FireMonkey supports. Moreover, you will learn about nontrivial LiveBindings utilizations.

Chapter 4, *The Thousand Faces of Multithreading*, is one of the most complex chapters. It talks about thread synchronization and the mechanisms used to obtain this synchronization, including TMonitor, thread-safe queues, and TEvent. By the end of this chapter, you will be able to create and communicate with background threads, leaving the main thread free to update your GUI (or communicate with the OS).

Chapter 5, *Putting Delphi on the Server*, focuses on how well Delphi can behave when running on a server. Some people think that Delphi is a client-only tool, but it is not true; the number of Delphi server-side systems running all over the world prove it! In this chapter, we'll show how to create powerful servers that offer services over a network. Then, in one of the recipes, we'll also implement a JavaScript client that brings the database data to the user's browser. The techniques explained in this chapter open a range of possibilities, especially in the mobile and web area.

www.allitebooks.com

Chapter 6, Riding the Mobile Revolution with FireMonkey, is dedicated to mobile development with Delphi and FireMonkey. If you are interested in mobile development, I think that this will be your favorite chapter! Mobiles are everywhere and this chapter will explain how to write software for your Android or iOS device, what are the best practices to use, how to save your data on your mobile device, how to retrieve and update remote data, and how to integrate with a mobile operating system.

Chapter 7, *Using Specific Platform Features*, shows you how to integrate your app with the underlying mobile operating systems beyond what FireMonkey offers. You will learn how to import Java and Objective-C libraries in your app and use the SDK classes from your Object Pascal code.

What you need for this book

This book talks about Delphi, so you need Delphi. Not all recipes are available in all the Delphi editions. Typically, the mobile projects can be compiled only if you have Delphi Enterprise or better (or Delphi Professional plus the mobile add-on, or RAD Studio professional, or better). All the projects are compiled and tested on Delphi XE6. Many of the recipes can also be compiled on older versions.

If you want to run the mobile app on a phone or tablet, you can use the Android emulator or iOS simulator, but it is strongly recommended that you use an actual device to see how the app really behaves. To deploy an iOS app on your device, you also need an Apple computer with Mac OS X. More information is provided in the related chapters.

Who this book is for

This book aims to help professional Delphi developers in their day-to-day jobs. This book will teach you about the newest Delphi technologies and its hidden gems. It is not a book for a newbie, but the practical approach will help you reach a new level in your Delphi skills. An experienced developer will benefit from this book because nontrivial problems are solved using the best practices. Where more than one way is available or the topics are too vast to be explained in the available pages, references are provided that allow interested readers to go deeper in that field. It is a book that'll hold on to your desk for the next few years.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `TStyleManager.StyleNames` property contains all names of the available styles."

A block of code is set as follows:

```
LogMessage('Your message goes here for SUCCESS',
         EVENTLOG_SUCCESS, 0, 1);
```

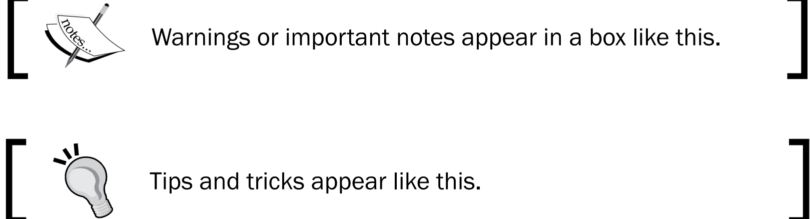
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  TStyleManager.TrySetStyle('Iceberg Classico');
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Any command-line input or output is written as follows:

```
C:\<ExeProjectPath>\WindowsService.exe /install
C:\<ExeProjectPath>\WindowsServiceOrGUI.exe /GUI
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on **Start**, wait for the confirmation, and the service should start to write its logfile."



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

www.allitebooks.com

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

www.allitebooks.com

1
Delphi Basics

In this chapter, we will cover the following recipes:

- Changing your application's look and feel with VCL styles and no code
- Changing the style of your VCL application at runtime
- Customizing TDBGrid
- Using the owner's draw combos and listboxes
- Creating a stack of embedded forms
- Manipulating JSON
- Manipulating and transforming XML documents
- I/O in the twenty-first century – knowing streams
- Putting your VCL application in the tray
- Creating a Windows service
- Associating a file extension with your application on Windows

Introduction

This chapter explains some of the day-to-day needs of a Delphi programmer. These are ready-to-use recipes that will be useful every day and have been selected ahead of a lot of others because although they may be obvious for some experienced users, they are still very useful. Even if there is no specifically database-related code, many of the recipes can also be used (or sometimes *especially* used) when you are dealing with data.

Changing your application's look and feel with VCL styles and no code

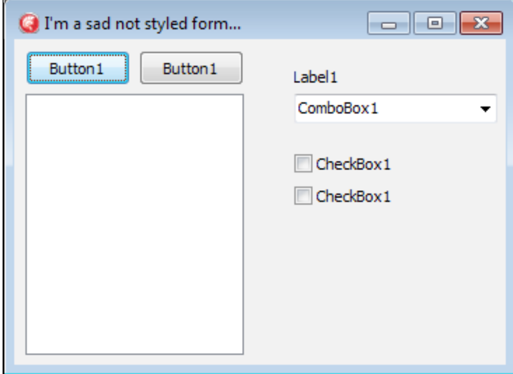
VCL styles are a major new entry in the latest versions of Delphi. They have been introduced in Delphi XE2 and are still one of the less-known features for the good old Delphi developers. However, as usual, some businessmen say *looks matter*, so the look and feel of your application could be one of the reasons to choose your product over one from a competitor. Consider that with a few mouse clicks you can apply many different styles to your application to change the look and feel of your applications. So why not give it a try?

Getting ready

VCL styles can be used to revamp an old application or to create a new one with a nonstandard GUI. VCL styles are a completely different beast to FireMonkey styles. They are both styles but with completely different approaches and behavior.

To get started with VCL styles, we'll use a new application. Let's create a new VCL application and drag-and-drop some components onto the main form (for example, two **TButton** components, one **TListBox** component, one **TComboBox** component, and a couple of **TCheckBox** components).

The following screenshot is the resultant form that runs on a Windows 7 machine:



A form without style

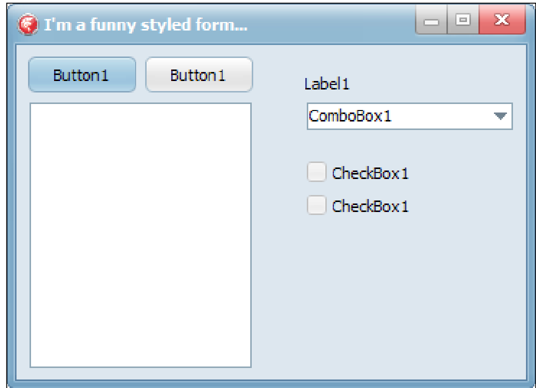
How to do it...

Now we've to apply a set of nice styles. To do this, perform the following steps:

1. Navigate to **Project** | **Options**. In the resultant dialog, go to **Application** | **Appearance** and select all the styles that we want to include in our application.

www.allitebooks.com

2. Using the **Preview** button, the IDE shows a simple demo form with some controls, and we can get an idea about the final result of our styled form. Feel free to experiment and choose the style—or set of styles—that you like. Only one style will be used at a time, but we can link the necessary resources to the executable and select the proper one at runtime.
3. After selecting all the required styles from the list, we've to select one in the combobox at the bottom of the screen. This style will be the default style for our form and will be loaded as soon as the application starts. You can delay this choice and make it at runtime using code if you prefer.
4. Click on **OK** and hit *F9* (or navigate to **Run** | **Run**) and your application is styled! The resultant form is shown in the following screenshot:



The same form as the preceding one but with the Iceberg Classico style applied

How it works...

Selecting one or more styles by navigating to **Project** | **Options** | **Application** | **Appearance** can cause the Delphi linker to link the style resource to your executable. It is possible to link many styles to your executable, but you can use only one style at time. So, how does Delphi know which style you want to use when there are more than one styles? If we check the `Project` file (the file with the `.dpr` extension) by navigating to **Project** | **View Source**, you can see where and how this little magic happens.

The following lines are the interesting part:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  TStyleManager.TrySetStyle('Iceberg Classico');
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

When we've selected the **Iceberg Classico** style as the default style, the Delphi IDE adds a line just before the creation of the main form, setting the default style for the application using the `TStyleManager.TrySetStyle` static method.

`TStyleManager` is a very important class when dealing with VCL styles. We'll see more about it in the next recipe when we'll learn how to change a style at runtime.

There's more...

Delphi and C++Builder XE6 come with 29 VCL styles available in `C:\Program Files (x86)\Embarcadero\Studio\14.0\Redist\styles\vcl\` (with a standard installation).

Moreover, it is possible to create your own styles or modify the existing ones by using the **Bitmap Style Designer** available at **Tools** | **Bitmap Style Designer** menu. The Bitmap Style Designer also provides test applications to test VCL styles.

For more details on how to create or customize a VCL style, check the following link:

`http://docwiki.embarcadero.com/RADStudio/XE6/en/Creating_a_Style_using_the_Bitmap_Style_Designer`

Changing the style of your VCL application at runtime

VCL styles are a powerful way to change the appearance of your application, but using them only as design-time tools is way too limited. One of the main features of a VCL style is the ability to change the style while an application is running.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Getting ready

Because a VCL style is simply a particular kind of binary file, we can allow our users to load their preferred style at runtime, and we can even provide new styles—publishing them on a website or sending them by an e-mail to our customers.

In this recipe, we'll be able to change the style while an application is running using a style already linked at design time or let the user choose between a set of styles deployed inside a folder.

www.allitebooks.com

How to do it...

Styles manipulation at runtime is done using the class methods of the `TStyleManager` class:

1. Create a brand new VCL application and add the `Vcl.Themes` and `Vcl.Styles` units to the main implementation form. These units are required to use VCL styles at runtime.
2. Drop on the form a **TListBox** component, two **TButton** components, and two **TOpenDialog** components. Leave the default component names.
3. Go to **Project** | **Appearance** and select eight styles of your choice from the list. Leave the **Default style** option to **Windows**.
4. The `TStyleManager.StyleNames` property contains all names of the available styles. In the `FormCreate` event handler, we have to load the already linked styles present in the executable to the listbox to let the user choose one of them. So, create a new procedure called `StylesListRefresh` with the following code and call it from the `FormCreate` event handler:

```
procedure TMainForm.StylesListRefresh;
var
  stylename: string;
begin
  ListBox1.Clear;
  // retrieve all the styles linked in the executable
  for stylename in TStyleManager.StyleNames do
  begin
    ListBox1.Items.Add(stylename);
  end;
end;
```

5. In the `Button1Click` event handler, we've to set the current style according to the one selected from `ListBox1` using the following code:

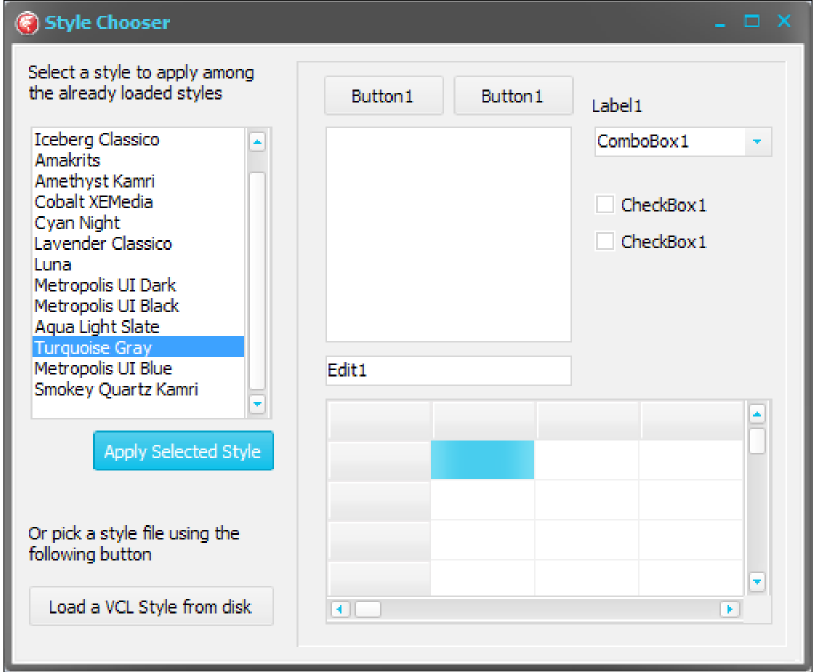
```
  TStyleManager.SetStyle(ListBox1.Items[ListBox1.ItemIndex]);
```

6. The `Button2Click` event handler should allow the user to select a style from disk. So, we have to create a folder named `styles` at level of our executable and copy a few `.vsf` files from the default style directory which is `C:\Program Files (x86)\Embarcadero\Studio\14.0\Redist\styles\vcl\` in RAD Studio XE6.
7. After copying the files, write the following code under the `Button2Click` event handler. This code allows the user to chose a style file directly from the disk. Then you can select one of the loaded styles from the listbox and click on **Button1** to apply it to the application. The code is as follows:

```
if OpenDialog1.Execute then
begin
  if TStyleManager.IsValidStyle(OpenDialog1.FileName) then
```

```
    begin
      //load the style file
      TStyleManager.LoadFromFile(OpenDialog1.FileName);
      //refresh the list with the currently available styles
      StylesListRefresh;
      ShowMessage('New VCL Style has been loaded');
    end
    else
      ShowMessage('The file is not a valid VCL Style!');
    end;
  end;
```

8. Just to have an idea of how the different controls appear with the selected style, drag-and-drop some controls to the right-hand side of the form. The following screenshot shows an application with some styles loaded, some at design time and some from the disk. Hit *F9* (or go to **Run** | **Run**) and play with your application using and loading styles from the disk.



The Style Chooser form with a Turquoise Gray style loaded

www.allitebooks.com

How it works...

The `TStyleManager` class has all the methods we need:

- Inspect the loaded styles with `TStyleManager.StyleNames`
- Apply an already loaded style to the running application using `TStyleManager.SetStyle('StyleName')`
- Check if a file is a valid style with `TStyleManager.IsValidStyle('StylePathFileName')`
- Load a style file from disk using `TStyleManager.LoadFromFile('StylePathFileName')`

After loading new styles from the disk, these new styles are completely similar to the styles linked to the executable during the compile and link phases and can be used in the same way.

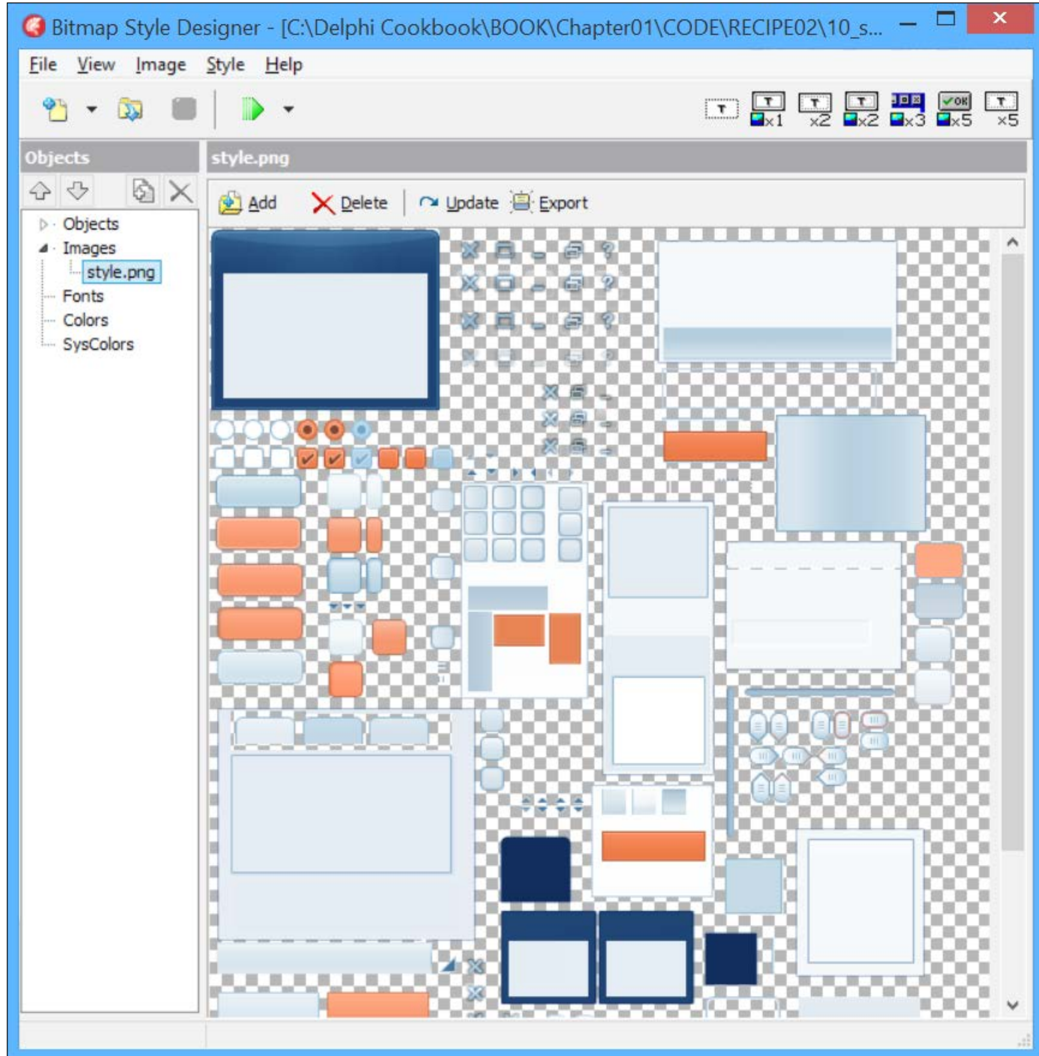
There's more...

Other things to consider are third-party controls. If your application uses third-party controls, take care with their style support. If your external components do not support styles, you will end up with some controls styled (the original included in Delphi) and some not (your external third-party controls)!

By navigating to **Tools** | **Bitmap Style Designer** and using a custom VCL style, we can also perform the following actions:

- Change the application's colors (for example, `ButtonNormal`, `ButtonPressed`, `ButtonFocused`, `ButtonHot`, and so on)
- Override the system's colors (for example, `clCaptionText`, `clBtnFace`, `clActiveCaption`, and so on)
- Change the font color and font name for particular controls (for example, `ButtonTextNormal`, `ButtonTextPressed`, `ButtonTextFocused`, `ButtonTextHot`, and so on)

The following screenshot shows the **Bitmap Style Designer** window while working on a custom style:



The Bitmap Style Designer while it is working on a custom style

Customizing TDBGrid

The adage *a picture is worth a thousand words* refers to the notion that a complex idea can be conveyed with just a single still image. Sometimes, even a simple concept is easier to understand and nicer to see if it is represented by images. In this recipe, we'll see how to customize the `TDBGrid` object to visualize graphical representation of data.

www.allitebooks.com

Getting ready

Many VCL controls are able to delegate their drawing, or part of it, to user code. This means that we can use simple event handlers to draw standard components in different ways. It is not always simple, but `TDBGrid` is customizable in a really easy way. Let's say that we have a class of musicians that have to pass a set of exams. We want to show the percentage of musicians who have already passed exams with a progress bar, and if the percent is higher than 50 percent, there should also be a check in another column.

How to do it...

We'll use a special in-memory table from the `FireDAC` library. `FireDAC` is a new data access library from Embarcadero, which is included in RAD Studio since Version XE5. If some of the code seems unclear at the moment, consider the in-memory table as a normal `TDataSet` descendant. However, at the end of this section, there are some links to the FireDAC documentation, and I strongly suggest you to read them if you still don't know FireDAC.
To customize `TDBGrid`, perform the following steps:

1. Create a brand new VCL application and drop on the form the `TFDMemTable`, `TDBGrid`, `TDataSource`, and `TDBNavigator` component. Connect all these components in the usual way (`TDBGrid`->`TDataSource`->`TFDMemTable`). Set the `TDBGrid` font size to `24`. This will create more space in the cell for our graphical representation.
2. Using the `TFDMemTable` fields editor, add the following fields and then activate the dataset setting by setting its `Active` property to `True`:

Field name	Field datatype	Field type
`FullName`	String (size `50`)	Data
`TotalExams`	Integer	Data
`PassedExams`	Integer	Data
`PercPassedExams`	Float	Calculated
`MoreThan50Percent`	Boolean	Calculated

3. In a real application, we should load real data from some sort of database. But for now, we'll use some custom data generated in code. We have to load this data into the dataset with the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FDMemTable1.InsertRecord(
         ['Ludwig van Beethoven',30,10]);
  FDMemTable1.InsertRecord(
```

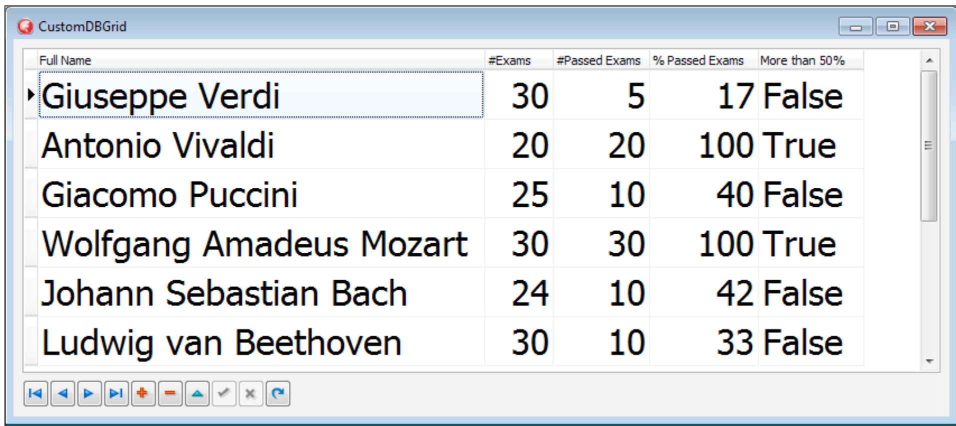
```
          ['Johann Sebastian Bach',24,10]);
  FDMemTable1.InsertRecord(
          ['Wolfgang Amadeus Mozart',30,30]);
  FDMemTable1.InsertRecord(
          ['Giacomo Puccini',25,10]);
  FDMemTable1.InsertRecord(
          ['Antonio Vivaldi',20,20]);
  FDMemTable1.InsertRecord(
          ['Giuseppe Verdi',30,5]);
end;
```

4. Do you remember? We've two calculated fields that need to be filled in some way. Create the `OnCalcFields` event handler on the `TFDMemTable` component and fill it with the following code:

```
procedure TMainForm.FDMemTable1CalcFields(
   DataSet: TDataSet);
var
  p: Integer;
  t: Integer;
begin
  p := FDMemTable1.FieldByName('PassedExams').AsInteger;
  t := FDMemTable1.FieldByName('TotalExams').AsInteger;
  if t = 0 then
  begin
    FDMemTable1.FieldByName('PercPassedExams').AsFloat := 0
  end
  else
  begin
    FDMemTable1.
         FieldByName('PercPassedExams').
         AsFloat := p / t * 100;
  end;

FDMemTable1.FieldByName('MoreThan50Percent').AsBoolean :=
FDMemTable1.
   FieldByName('PercPassedExams').AsFloat > 50;
end;
```


www.allitebooks.com

5. Run the application by hitting *F9* (or navigating to **Run** | **Run**) and you will get the following screenshot:



A normal form with some data

6. This is useful, but a bit boring. Let's start our customization. Close the application and return to Delphi IDE.

7. Go to the `TDBGrid` properties and set `DefaultDrawing` to `false`.

   Go to the `TDBGrid` event and create an event handler for `OnDrawColumnCell`. All the customization code goes in this event.

   Include the `Vcl.GraphUtil` unit and write the following code in the `DBGrid1DrawColumnCell` event:

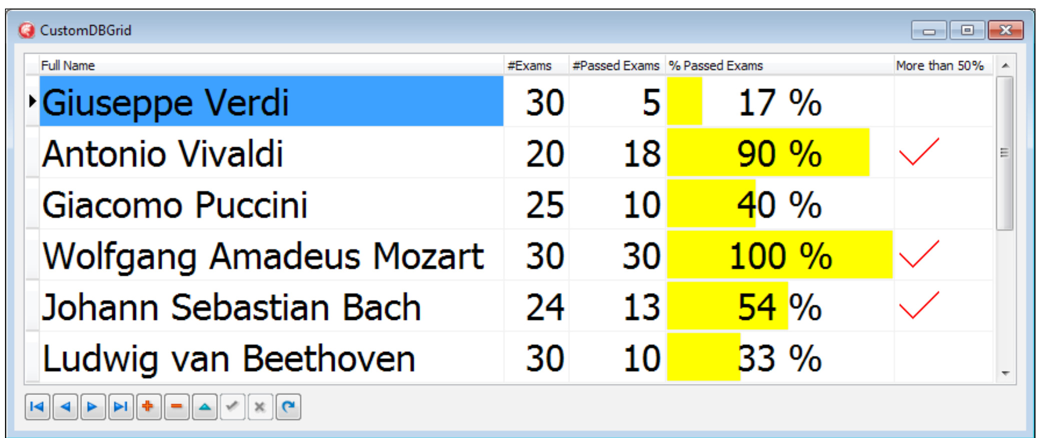
```
procedure TMainForm.DBGrid1DrawColumnCell(Sender: TObject; const
  Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
var
  R: TRect;
  Grid: TDBGrid;
  S: string;
  WPerc: Extended;
  SSize: TSize;
  SavedPenColor: Integer;
  SavedBrushColor: Integer;
  SavedPenStyle: TPenStyle;
  SavedBrushStyle: TBrushStyle;
begin
  Grid := TDBGrid(Sender);
  if [gdSelected, gdFocused] * State <> [] then
    Grid.Canvas.Brush.Color := clHighlight;
```

```
if Column.Field.FieldKind = fkCalculated then
begin
  R := Rect;
  SavedPenColor := Grid.Canvas.Pen.Color;
  SavedBrushColor := Grid.Canvas.Brush.Color;
  SavedPenStyle := Grid.Canvas.Pen.Style;
  SavedBrushStyle := Grid.Canvas.Brush.Style;
end;

if Column.FieldName.Equals('PercPassedExams') then
begin
  S := FormatFloat('##0', Column.Field.AsFloat) + ' %';
  Grid.Canvas.Brush.Style := bsSolid;
  Grid.Canvas.FillRect(R);
  WPerc := Column.Field.AsFloat / 100 * R.Width;
  Grid.Canvas.Font.Size := Grid.Font.Size - 1;
  Grid.Canvas.Font.Color := clWhite;
  Grid.Canvas.Brush.Color := clYellow;
  Grid.Canvas.RoundRect(R.Left, R.Top,
       Trunc(R.Left + WPerc), R.Bottom, 2, 2);
  InflateRect(R, -1, -1);
  Grid.Canvas.Pen.Style := psClear;
  Grid.Canvas.Font.Color := clBlack;
  Grid.Canvas.Brush.Style := bsClear;
  SSize := Grid.Canvas.TextExtent(S);
  Grid.Canvas.TextOut(
       R.Left + ((R.Width div 2) - (SSize.cx div 2)),
       R.Top + ((R.Height div 2) - (SSize.cy div 2)),
       S);
end
else if Column.FieldName.Equals('MoreThan50Percent') then
begin
  Grid.Canvas.Brush.Style := bsSolid;
  Grid.Canvas.Pen.Style := psClear;
  Grid.Canvas.FillRect(R);
  if Column.Field.AsBoolean then
  begin
    InflateRect(R, -4, -4);
    Grid.Canvas.Pen.Color := clRed;
    Grid.Canvas.Pen.Style := psSolid;
    DrawCheck(Grid.Canvas,
      TPoint.Create(R.Left, R.Top + R.Height div 2),
        R.Height div 3);
  end;
```


www.allitebooks.com

```
      end
      else
        Grid.DefaultDrawColumnCell(Rect, DataCol,
             Column, State);

      if Column.Field.FieldKind = fkCalculated then
      begin
        Grid.Canvas.Pen.Color := SavedPenColor;
        Grid.Canvas.Brush.Color := SavedBrushColor;
        Grid.Canvas.Pen.Style := SavedPenStyle;
        Grid.Canvas.Brush.Style := SavedBrushStyle;
      end;
    end;
```

8. That's all, folks! Hit *F9* (or navigate to **Run** | **Run**) and we now have a nicer grid with more direct information about our data:



The same grid with a bit of customization

How it works...

By setting the `DBGrid` property `DefaultDrawing` to `false`, we told the grid that we want to manually draw all the data into every cell. The `OnDrawColumnCell` event allows us to actually draw using the standard Delphi code. For each cell we are about to draw, the event handler is called with a list of useful parameters to know which cell we're about to draw and what data we have to read considering the column currently drawn. In this case, we want to draw only the calculated columns in a customized way. This is not a rule, but this can be done to manipulate all cells. We can draw any cell in the way we like. For the cells where we don't want to do custom drawing, a simple `DefaultDrawColumnCell` call method passing the same parameters we got from the event and the VCL code will draw the current cell as usual.

Among the event parameters, there is `Rect` (of the `TRect` type) that represents the specific area we're about to draw, there is `Column` (of the `TColumn` type) that is a reference to the current column of the grid, and there is `State` (of the `TGridDrawState` type) that is a set of the grid cell states (for example, `Selected`, `Focused`, `HotTrack`, and so on). If our drawing code ignores the `State` parameter, all the cells will be drawn in the same way and users cannot see which cell or row is selected.

The event handler uses a sets intersection to know whether the current cell should be drawn as a selected or focused cell:

```
if [gdSelected, gdFocused] * State <> [] then
  Grid.Canvas.Brush.Color := clHighlight;
```

Remember that if your dataset has 100 records and 20 fields, the `OnDrawColumnCell` method will potentially be called 2,000 times! So the event code must be fast, otherwise the application will become less responsive.

There's more...

Owner drawing is a really large topic and can be simple or tremendously complex involving much `Canvas` related code. However, often the kind of drawing you need will be relatively similar. So, if you need checks, arrows, color gradients, and so on, check the procedures in the `Vcl.GraphUtil` unit. Otherwise, if you need images, you could use a `TImageList` class to hold all the images needed by your grid.

The good news is that the drawing code can be reused by different kind of controls, so try to organize your code in a way that allows code reutilization avoiding direct dependencies to the form where the control is.

The code in the drawing events should not contain business logic or presentation logic. If you need presentation logic, put it in a separate and testable function or class.

Using the owner's draw combos and listboxes

Many things are organized in a list. Lists are useful when you have to show items or when your user has to choose among a set of possible options. Usually, standard lists are flat, but sometimes you need to transmit more information in addition to a list of items. Let's think about when you go to choose a font in an advanced text editor such as Microsoft Word or OpenOffice.org. Having the name of the font drawn in the font style itself helps users to make a faster and more reasoned choice. In this recipe, we'll see how to make listboxes more useful. The code is perfectly valid also for a `TComboBox`.

www.allitebooks.com

Getting ready

As we saw in the *Customizing TDBGrid* recipe, many VCL controls are able to delegate their drawing, or part of it, to user code. This means that we can use simple event handlers to draw standard components in different ways. Let's say that we have a list of products in our store and we have to set discounts on these products. As there are many products, we want to make it simple so that our users can make a fast selection between the available discount percentages using a color code.

How to do it...

1. Create a brand new VCL application and drop on the form a `TListBox` component. Set the following properties:

Property	Value
`Style`	`lbOwnerDrawFixed`
`Font.Size`	`14`

2. In the listbox `Items` property, add seven levels of discount. For example, you can use the following: no discount, 10 percent discount, 20 percent discount, 30 percent discount, 40 percent discount, 50 percent discount, and 70 percent discount.
3. Then, drop a `TImageList` component on the form and set the following properties:

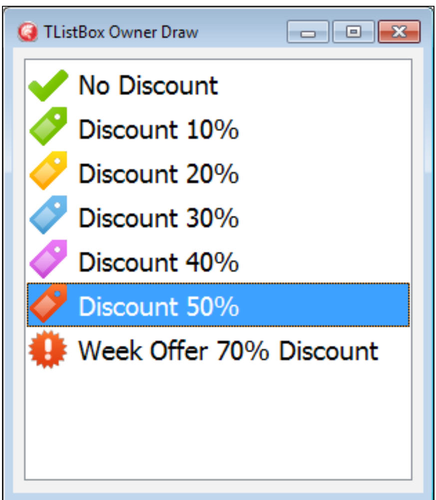
Property	Value
`ColorDepth`	`cd32Bit`
`DrawingStyle`	`dsTransparent`
`Width`	`32`
`Height`	`32`

4. The `TImageList` component is our image repository and will be used to draw an image by index. Load seven PNG images (of 32 x 32 size) into `TImageList`. You can find some nice PNG icons in the recipe's project folder (`ICONS\PNG\32`).
5. Create an `OnDrawItem` event handler for the `TListBox` component and write the following code:

```
procedure TCustomListControlsForm.ListBox1DrawItem(
   Control: TWinControl; Index: Integer;
   Rect: TRect; State: TOwnerDrawState);
var
  LBox: TListBox;
  R: TRect;
  S: string;
```

```
  TextTopPos, TextLeftPos, TextHeight: Integer;
const
  IMAGE_TEXT_SPACE = 5;
begin
  LBox := Control as TListBox;
  R := Rect;
  LBox.Canvas.FillRect(R);
  ImageList1.Draw(LBox.Canvas, R.Left, R.Top, Index);
  S := LBox.Items[Index];
  TextHeight := LBox.Canvas.TextHeight(S);
  TextLeftPos := R.Left +
   ImageList1.Width + IMAGE_TEXT_SPACE;
  TextTopPos := R.Top + R.Height div 2 - TextHeight div 2;
  LBox.Canvas.TextOut(TextLeftPos, TextTopPos, S);
end;
```

6. Run the application by hitting *F9* (or navigate to **Run** | **Run**) and you will see the following screenshot:



Our listbox with some custom icons read from TImageList

How it works...

The `TListBox.OnDrawItem` event handler allows us to customize the drawing of the listbox. In this recipe, we used a `TImageList` component as the image repository for the listbox. Using the `Index` parameter, we read the correspondent image in the image list and drawn on the `Canvas` listbox. After this, all the other code is related to the alignment of image and text inside the listbox row.

www.allitebooks.com

Remember that this event handler will be called for each item in the list, so the code must be fast and should not do too much slow `Canvas` writing. Otherwise, all your GUI will be unresponsive. If you want to create complex graphics on the fly in the event, I strongly suggest you to prepare your images the first time you draw the item and then put them in a sort of cache memory (`TObjectList<TBitmap>` is enough).

There's more...

While you are in the `OnDrawITem` function, you can do whatever you want with the `TListBox Canvas`. Moreover, the `State` parameter (of the `TOwnerDrawState` type) tells you in which states the listbox item is (for example, `Selected`, `Focused`, `HotTrack`, and so on), so you can use different kind of drawings depending on the item's state. You can check the *Customizing TDBGrid* recipe to know about the `TDBGrid` owner drawing for an example of the `State` parameter.

If you want to make your code aware of the selected VCL style, changing the color used according to it, you can use `StyleServices.GetStyleColor()`, `StyleServices.GetStyleFontColor()`, and `StyleServices.GetSystemColor()` into the `Vcl.Themes` unit.

The icons used in this recipe are from the Icojam website (`http://www.icojam.com`). The specific set used is available at `http://www.icojam.com/blog/?p=259`.

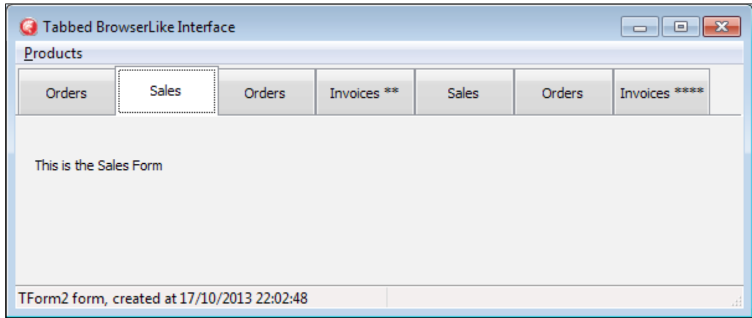
Creating a stack of embedded forms

Every modern browser has a tabbed interface. Also, many other kinds of multiple views software have this kind of interface. Why? Because it's very useful. While you are reading one page, you can rapidly check another page, and then still come back to the first one at the same point you left some seconds ago. You don't have to redo a search or redo a lot of clicks to just go back to that particular point. You simply have switched from one window to another and back to the first. I see too many business applications that are composed by a bounce of dialog windows. Every form is called with the `TForm.ShowModal` method. So, the user has to navigate into your application one form at time. This is simpler to handle for the programmer, but it's less user-friendly for your customers. However, providing a switchable interface to your customer is not that difficult. In this recipe, we'll see a complete example on how to do it.

Getting ready

This recipe is a bit more complex than the previous recipes, so I'll not explain all the code but only the fundamental parts. You can find the complete code in the book's code repository (`Chapter1\RECIPE05`).

Let's say we want to create a tabbed interface for our software that is used to manage product orders, sales, and invoices. All the forms must be usable at the same time without having to close the previous one. Before we begin, the following screenshot is what we want to create:

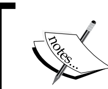


The main form containing seven embedded child forms

How to do it...

The project is composed by a bounce of forms. The main form has a `TTabControl` component that allows switching between the active forms. All embedded forms inherit from `EmbeddableForm`. The most important is the `Show` method shown as follows:

```
procedure TEmbeddableForm.Show(AParent: TPanel);
begin
  Parent := AParent;
  BorderStyle := bsNone;
  BorderIcons := [];
  Align := alClient;
  Show;
end;
```



Note that all the forms apart from the main form have been removed from the **Auto-Create Form** list (**Project** | **Options** | **Forms**).

All the other forms descend from the `EmbeddableForm` method and are added to the `TTabControl` component on the main form with a line of code similar to the following:

```
procedure TMainForm.MenuOrdersClick(Sender: TObject);
begin
  AddForm(TForm1.Create(self));
end;
```


www.allitebooks.com

The MainForm AddForm method is in charge of adding an actual instance of a form into the tabs, keeping a reference of it. The following code shows you how this is done:

```
//Add a form to the stack
procedure TMainForm.AddForm(
  AEmbeddableForm: TEmbeddableForm);
begin
  AEmbeddableForm.Show(Panel1);
  //each tab show the caption of the containing form and
  //hold the reference to it
  TabControl1.Tabs.AddObject(
         AEmbeddableForm.Caption, AEmbeddableForm);
  ResizeTabsWidth;
  ShowForm(AEmbeddableForm);
end;
```

Other methods are in charge of bringing an already created form to the front when a user clicks on the related tab and then to close a form when the related tab is removed (check the `ShowForm` and `WMEmbeddedFormClose` methods).

How it works...

There is a bit of code, but the concepts are simple:

- When we need to create a new form, add it in the `TabControl1.Tabs` property. The caption of the form is the caption of the tab, and the object is the form itself. This is what the `AddForm` method does with the following line:

  ```
  TabControl1.Tabs.AddObject(
     AEmbeddableForm.Caption, AEmbeddableForm);
  ```

- When a user clicks on a tab, we have to find the associated form cycling through the `TabControl1.Tabs.Objects` list and bring it to the front.
- When a form asks for closing (sending a `WM_EMBEDDED_CLOSE` message), we have to set the `ParentWantClose` property and then call the `Close` method of the correspondent form.
- If a user wants to close a form by closing the correspondent tab (in the recipe code, there is a `TPopMenu` component connected to the `TabControl` component, which is used to close a form with a right-click), we have to call the `Close` method on the correspondent form.
- Every form frees itself in the `OnClose` event handler. This is done once for all in the `TEmbeddableForm.CloseForm` event handler using the `caFree` action.

There's more...

Embedding a form into another `TWinControl` is not difficult and allows you to create flexible GUIs without using `TPageControl` and frames. For the end user, this multitabbed GUI is probably more familiar because all the modern browsers use it, and probably your user already knows how to use a browser with different pages or screens opened. From the developer point of view, the multitabbed interface allows for much better programming patterns and practices. This technique can also be used for other scenarios where you have to embed one screen into another.

More flexible (and complex) solutions can be created involving the use of Observers, but in simple cases, this recipe's solution based on Windows Messaging is enough.

More information about the Observer design pattern can be found at `http://sourcemaking.com/design_patterns/observer/delphi`.

Another interesting solution (that does not rely on Windows Messaging and so is also cross platform) may be based on the `System.Messaging.TMessageManager` class. More info about `TMessageManager` can be found at `http://docwiki.embarcadero.com/Libraries/XE6/en/System.Messaging.TMessageManager`.

The code in this recipe can be used with every component that uses `TStringList` to show items (`TListBox`, `TComboBox`, and so on) and can be adapted easily for other scenarios.

In the recipe code, you'll also find a nice way to show status messages generated by the embedded forms and a centralized way to show application hints in the status bar.

Manipulating JSON

JavaScript Object Notation (**JSON**) is a lightweight data-interchange format. As the reference site (`http://www.json.org`) says:

> *It is easy for humans to read and write. It is easy for machines to parse and generate.*

It is based on a subset of the JavaScript programming language, but it is not limited to JavaScript in any way. Indeed, JSON is a text format that is completely language agnostic. These properties make JSON an ideal data-interchange language for many utilizations. In recent years, JSON has superseded XML in many applications, especially on data exchange and in general when the data size matters, because of its intrinsic conciseness and simplicity.

www.allitebooks.com

Getting ready

JSON provides the following five datatypes: string, number, object, array, Boolean, and null.

This simplicity is a plus when you have to read a JSON string into some kind of language-specific structures, because every modern language supports JSON datatypes as simple types, HashMap (in case of JSON object), or List (in case of JSON array). So, it makes sense that a data format that is interchangeable with programming languages is also based on these types and structures.

Since Version 2009, Delphi provides built-in support for JSON. The `System.JSON.pas` unit contains all JSON types with a nice object-oriented interface. In this recipe, we'll see how to generate, modify, and parse a JSON string.

How to do it...

1. Create a new VCL application and drop three **TButton** and a **TMemo**. Align all the buttons as a toolbar at the top of the form and the memo to all the remaining form client area.
2. From left- to right-hand side, name the buttons as `btnGenerateJSON`, `btnModifyJSON`, and `btnParseJSON`.
3. We'll use static data as our data source. A simple matrix is enough for this recipe. Just after the start of the `implementation` section of the unit, write the following code:

```
type
  TCarInfo = (
   Manufacturer = 1,
   Name = 2,
   Currency = 3,
   Price = 4);

var
  Cars: array [1 .. 4] of
         array [Manufacturer .. Price] of string = (
          ('Ferrari','360 Modena','EUR', '250000'),
          ('Ford', 'Mustang', 'USD', '80000'),
          ('Lamborghini', 'Countach', 'EUR','300000'),
          ('Chevrolet', 'Corvette', 'USD', '100000')
          );
```

4. The `TMemo` component is used to show our JSON and our data. To keep things clear, create on the form a public property called `JSON` and map its `setter` and `getter` to the `Memo1.Lines.Text` property. Use the following code for this:

```
//…other form methods declaration
private
  procedure SetJSON(const Value: String);
  function GetJSON: String;
public
  property JSON: String read GetJSON write SetJSON;
end;

//…then in the implementation section
function TMainForm.GetJSON: String;
begin
  Result := Memo1.Lines.Text;
end;

procedure TMainForm.SetJSON(const Value: String);
begin
  Memo1.Lines.Text := Value;
end;
```

5. Now, create event handlers for each button and write the following code. Pay attention to the event names. The code is as follows:

```
procedure TMainForm.btnGenerateJSONClick(Sender: TObject);
var
  i: Integer;
  JSONCars: TJSONArray;
  Car, Price: TJSONObject;
begin
  JSONCars := TJSONArray.Create;
  try
    for i := Low(Cars) to High(Cars) do
    begin
      Car := TJSONObject.Create;
      JSONCars.AddElement(Car);
      Car.AddPair('manufacturer',
          Cars[i][TCarInfo.Manufacturer]);
      Car.AddPair('name', Cars[i][TCarInfo.Name]);
      Price := TJSONObject.Create;
      Car.AddPair('price', Price);
      Price.AddPair('value',
         TJSONNumber.Create(
```


www.allitebooks.com

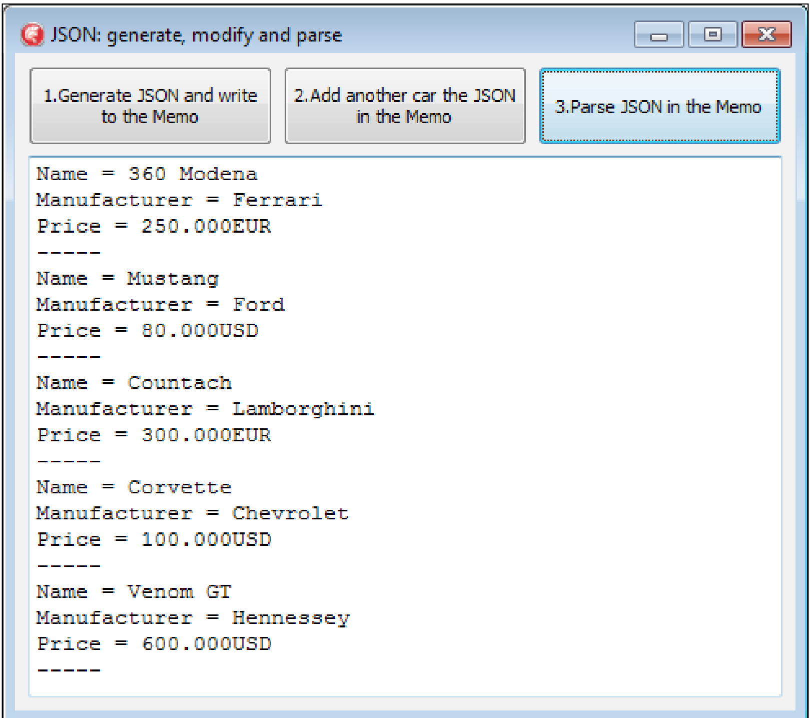
```
                Cars[i][TCarInfo.Price].ToInteger));
        Price.AddPair('currency',
            Cars[i][TCarInfo.Currency]);
      end;
      JSON := JSONCars.ToString;
    finally
      JSONCars.Free;
    end;
  end;

  procedure TMainForm.btnModifyJSONClick(Sender: TObject);
  var
    JSONCars: TJSONArray;
    Car, Price: TJSONObject;
  begin
    JSONCars := TJSONObject.ParseJSONValue(JSON)
                      as TJSONArray;
    try
      Car := TJSONObject.Create;
      JSONCars.AddElement(Car);
      Car.AddPair('manufacturer', 'Hennessey');
      Car.AddPair('name', 'Venom GT');
      Price := TJSONObject.Create;
      Car.AddPair('price', Price);
      Price.AddPair('value', TJSONNumber.Create(600000));
      Price.AddPair('currency', 'USD');
      JSON := JSONCars.ToString;
    finally
      JSONCars.Free;
    end;
  end;

  procedure TMainForm.btnParseJSONClick(Sender: TObject);
  var
    JSONCars: TJSONArray;
    i: Integer;
    Car, JSONPrice: TJSONObject;
    CarPrice: Double;
    s, CarName, CarManufacturer, CarCurrencyType: string;
  begin
    s := '';
    JSONCars := TJSONObject.ParseJSONValue(JSON)
           as TJSONArray;
```

```
    if not Assigned(JSONCars) then
      raise Exception.Create('Not a valid JSON');
    try
      for i := 0 to JSONCars.Size - 1 do
      begin
        Car := JSONCars.Get(i) as TJSONObject;
        CarName := Car.Get('name').JsonValue.Value;
        CarManufacturer := Car.Get('manufacturer')
           .JsonValue.Value;
        JSONPrice := Car.Get('price')
           .JsonValue as TJSONObject;
        CarPrice := (JSONPrice.Get('value').JsonValue
                        as TJSONNumber).AsDouble;
        CarCurrencyType := JSONPrice.Get('currency')
                              .JsonValue.Value;
        s := s + Format(
          'Name = %s' + sLineBreak +
          'Manufacturer = %s' + sLineBreak +
          'Price = %.0n%s' + sLineBreak +
          '-----' + sLineBreak,
        [CarName, CarManufacturer,
        CarPrice, CarCurrencyType]);
      end;
      JSON := s;
    finally
      JSONCars.Free;
    end;
  end;
```

6. Run the application by hitting *F9* (or navigate to **Run** | **Run**).
7. Click on the **btnGenerateJSON** button, and you should see a JSON array and some JSON objects inside in the memo.
8. Click on the **btnModifyJSON** button and you should see one more JSON object inside the outer JSON array in the memo.
9. Click on the last button and you should see the same data as before, but in a normal text representation.

www.allitebooks.com

10. After the third click, you should see something like the following screenshot:



Text representation of the JSON data generated and modified

There's more...

Although not the fastest or the most standard compliant on the market (at the time of writing), it is important to know the JSON Delphi parser because other Delphi technologies such as DataSnap use it. Luckily, there are a lot of alternative JSON parsers for Delphi if you find you have trouble with the standard ones.

Other notable JSON parsers are as follows:

- Superobject (`https://code.google.com/p/superobject/`)
- The JSON Delphi library (`http://sourceforge.net/projects/lkjson/`)
- The one included in the Delphi Web Script library (`https://code.google.com/p/dwscript/`)

If your main concern is speed, then check the Delphi Web Script or the superobject parsers.

There are also a lot of serialization libraries that use JSON as a serialization format. In general, every parser has its own way to serialize an object to JSON. Find your favorite. For example, in the *Serializing objects to JSON and back using RTTI* recipe in *Chapter 5*, *Putting Delphi on the Server*, you will see an open source library containing a set of serialization helpers using the default Delphi JSON parser.

However, JSON is not the right tool for every interchange or data representation job. XML has been creating other technologies that can help if you need to search, transform, and validate your data in a declarative way. In JSON land, there is no such level of standardization apart from the format itself. However, over the years, there is an effort to include at least the XML Schema counterpart in JSON, and you can find more details at `http://json-schema.org/`.

Manipulating and transforming XML documents

XML stands for eXtensible Markup Language (`http://en.wikipedia.org/wiki/XML`) and is designed to represent, transport, and store hierarchical data in trees of nodes. You can use XML to communicate with different systems to store configuration files, complex entities, and so on. All of these use a standard and powerful format. Delphi has had good support for XML for more than a decade now.

Getting ready

All the basic XML-related activities can be summarized with the following points:

- Generating XML data
- Parsing XML data
- Parsing XML data and modifying it

In this recipe, we will see how to do all these activities.

How to do it...

1. Create a new VCL application and drop three **TButton** and a **TMemo**. Align all the buttons as a toolbar at the top of the form and the memo to the remaining form client area.
2. From left- to right-hand side, name the buttons as `btnGenerateXML`, `btnModifyXML`, and `btnParseXML`.
3. The real work on the XML will be done by the `TXMLDocument` component. So, drop one instance of the form and set its `DOMVendor` property to `ADOM XML v4`.

www.allitebooks.com

4. We'll use static data as our data source. A simple matrix is enough for this recipe. Just after the `implementation` section of the unit, write the following code:

```
type
  TCarInfo = (
   Manufacturer = 1,
   Name = 2,
   Currency = 3,
   Price = 4);

var
  Cars: array [1 .. 4] of
         array [Manufacturer .. Price] of string = (
          (
            'Ferrari','360 Modena','EUR', '250,000'
           ),
          (
            'Ford', 'Mustang', 'USD', '80,000'
          ),
          (
            'Lamborghini', 'Countach', 'EUR','300,000'
           ),
           (
             'Chevrolet', 'Corvette', 'USD', '100,000'
            )
          );
```

5. We will use a `TMemo` component to display the XML and the data. To keep things clear, create on the form a public property called `XML` and map its `setter` and `getter` methods to the `Memo1.Lines.Text` property. Use the following code:

```
//…other form methods declaration
private
  procedure SetXML(const Value: String);
  function GetXML: String;
public
  property Xml: String read GetXML write SetXML;
end;

//…then in the implementation section
function TMainForm.GetXML: String;
begin
  Result := Memo1.Lines.Text;
end;

procedure TMainForm.SetXML(const Value: String);
begin
  Memo1.Lines.Text := Value;
end;
```

6. Now, create event handlers for each button. For the **btnGenerateXML** button, write the following code:

```
procedure TMainForm.btnGenerateXMLClick(Sender: TObject);
var
  RootNode, Car, CarPrice: IXMLNode;
  i: Integer;
  s: String;
begin
  XMLDocument1.Active := True;
  try
    XMLDocument1.Version := '1.0';
    RootNode := XMLDocument1.AddChild('cars');
    for i := Low(Cars) to High(Cars) do
    begin
      Car := XMLDocument1.CreateNode('car');
      Car.AddChild('manufacturer').Text :=
          Cars[i][TCarInfo.Manufacturer];
      Car.AddChild('name').Text :=
          Cars[i][TCarInfo.Name];
      CarPrice := Car.AddChild('price');
      CarPrice.Attributes['currency'] :=
         Cars[i][TCarInfo.Currency];
      CarPrice.Text := Cars[i][TCarInfo.Price];
      RootNode.ChildNodes.Add(Car);
    end;
    XMLDocument1.SaveToXML(s);
    Xml := s;
  finally
    XMLDocument1.Active := False;
  end;
end;
```

7. Now we've to write the code to change the XML. In the `btnModifyXML` click event handler, write the following code:

```
procedure TMainForm.btnModifyXMLClick(Sender: TObject);
var
  Car, CarPrice: IXMLNode;
  s: string;
begin
  XMLDocument1.LoadFromXML(Xml);
  try
    Xml := '';
    Car := XMLDocument1.CreateNode('car');
    Car.AddChild('manufacturer').Text := 'Hennessey';
```


www.allitebooks.com

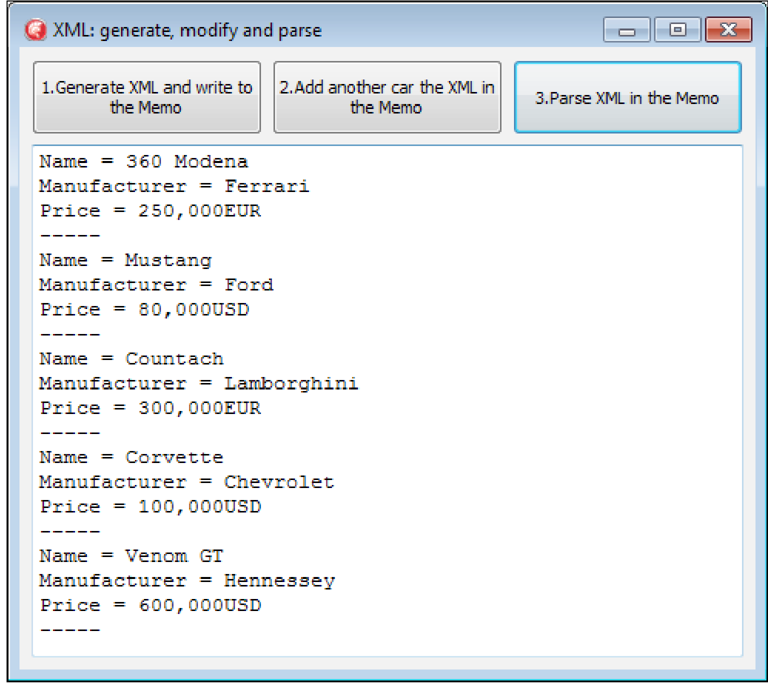
```
      Car.AddChild('name').Text := 'Venom GT';
      CarPrice := Car.AddChild('price');
      CarPrice.Attributes['currency'] := 'USD';
      CarPrice.Text := '600,000';
      XMLDocument1.DocumentElement.ChildNodes.Add(Car);
      XMLDocument1.SaveToXML(s);
      Xml := s;
    finally
      XMLDocument1.Active := False;
    end;
  end;
```

8. Write the following code under the `btnParseXML` click event handler:

```
procedure TMainForm.btnParseXMLClick(Sender: TObject);
var
  CarsList: IDOMNodeList;
  CurrNode: IDOMNode;
  childidx, i: Integer;
  CarName, CarManufacturer, CarPrice, CarCurrencyType: string;
begin
  XMLDocument1.LoadFromXML(Xml);
  try
    Xml := '';
    CarsList := XMLDocument1.
         DOMDocument.getElementsByTagName('car');
    for i := 0 to CarsList.length - 1 do
    begin
      CarName := '';  CarManufacturer := '';
      CarPrice := '';  CarCurrencyType := '';
      for childidx := 0 to
         CarsList[i].ChildNodes.length - 1 do
      begin
        CurrNode := CarsList[i].ChildNodes[childidx];
        if CurrNode.nodeName.Equals('name') then
          CarName := CurrNode.firstChild.nodeValue;
        if CurrNode.nodeName.Equals('manufacturer') then
          CarManufacturer := CurrNode.firstChild.nodeValue;
        if CurrNode.nodeName.Equals('price') then
        begin
          CarPrice := CurrNode.firstChild.nodeValue;
          CarCurrencyType :=
            CurrNode.Attributes.
            getNamedItem('currency').nodeValue;
        end;
```

```
          end;
          Xml := Xml +
            'Name = ' + CarName + sLineBreak +
            'Manufacturer = ' + CarManufacturer + sLineBreak +
            'Price = ' +
                    CarPrice + CarCurrencyType + sLineBreak +
            '-----' + sLineBreak;
        end;
      finally
        XMLDocument1.Active := False;
      end;
    end;
```

9. Run the application by hitting *F9* (or navigate to **Run** | **Run**).
10. Click on the **btnGenerateXML** button and you should see some XML data in the memo.
11. Click on the **btnModifyXML** button and you should see some more XML in the memo.
12. Click on the last button and you should see the same data as before, but in normal text representation.
13. After the third click, you should see something like the following screenshot:

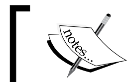


Text representation of the XML data generated and modified

www.allitebooks.com

How it works...

- The first button generates the XML representation of the data in our matrix. We've used some car information as sample data.



Note that the prices of the cars are not real.

- To create an XML, there are three fundamental `TXMLDocument` methods:
  - `XMLNode := XMLDocument1.CreateNode('node');`
  - `XMLNode.AddChild('childnode');`
  - `XMLNode.Attributes['attrname'] := 'attrvalue';`
- There are other very useful methods but these are the basics of XML generation.
- The **btnModifyXML** button loaded the XML into the memo and appended some other data (another car) to the list. Then, it updated the memo with the new updated XML. The following are the most important lines to note:

```
//Create a node without adding it to the DOM
Car := XMLDocument1.CreateNode('car');

//fill Car XMLNode… and finally add it to the DOM
//as child of the root node
XMLDocument1.DocumentElement.ChildNodes.Add(Car);
```

- The code under the `btnParseXMLClick` event handler allows you to read the data in the XML tree as simple text.

There's more...

There are many things to say about XML ecospace. There are XML engines that provide facilities to search data in an XML tree (XPath), validate an XML using another XML (XML Schema or DTD), transform an XML into another kind of format using another XML (XSLT), and for many others uses (`http://en.wikipedia.org/wiki/List_of_XML_markup_languages`). The good thing is that, just like XML itself, the `DOM` object is also standardized, so every library that is compliant to the standard has the same methods, from Delphi to JavaScript and from Python to C#.

`TXMLDocument` allows you to select the `DOMVendor` implementation. By default, there are three implementations available:

- **MSXML**: This is from Microsoft and implemented as a `COM` object. This supports XML transformations and is available only on Windows (so no Android, iOS, or Mac OS X).
- **ADOM XML**: This is an open source Delphi implementation and does not support transformations. This is available on all the supported Delphi platforms, so if you plan to write XML handling code on a mobile or Mac, this is the way to go.
- **XSLT**: This allows you to transform an XML into something else, using another XML as a stylesheet. The following function loads an XML and an XSLT from two string variables. Then, use the XSLT document to transform the XML document. The following code shows the details:

```
function Transform(XMLData: string; XSLT: string): WideString;
var
  XML: IXMLDocument;
  XSL: IXMLDocument;
begin
  XML := LoadXMLData(XMLData);
  XSL := LoadXMLData(XSLT);
  XML.DocumentElement.TransformNode(XSL.DocumentElement, Result)
end;
```

  This function doesn't know about the output format because it is defined by the XSLT document. The result could be an XML, an HTML, a CSV, or a plain text, or whatever the XSLT defines, the code doesn't change.

  XSLT can be really useful—go to `http://www.w3schools.com/xsl/xsl_languages.asp` for further details about the language.

I/O in the twenty-first century – knowing streams

Many of the I/O related activities handle streams of data. A stream is a sequence of data elements made available over time. According to Wikipedia:

> *A stream can be thought of as a conveyor belt that allows items to be processed one at a time rather than in large batches.*

At the lowest level, all the streams are bytes, but using a high-level interface could obviously help the programmer to handle his data. This is the reason why a stream object usually has methods such as `read`, `seek`, `write`, and many more, just to make the handling of byte stream a bit simpler.

In this recipe, we'll see some streams utilization examples.

www.allitebooks.com

Getting ready

In the good old Pascal days, there was a set of functions to handle the I/O (`AssignFile`, `Reset`, `Rewrite`, `CloseFile`, and so on), now we've a bounce of classes. All Delphi streams inherit from `TStream` and can be used as an internal stream of one of the adapter classes (as adapter, I mean an implementation of the Adapter or Wrapper design pattern from the famous *Gang of Four*, *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Professional*, book about design patterns).

There are 10 fundamental types of streams:

Class	Use
`System.Classes.TBinaryWriter`	This is a writer for binary data
`System.Classes.TStreamWriter`	This is a writer for characters to stream
`System.Classes.TStringWriter`	This is a writer for a string
`System.Classes.TTextWriter`	This is a writer for sequence of characters; it is an abstract class
`System.Classes.TWriter`	This writes component data to an associated stream
`System.Classes.TReader`	This reads component data from an associated stream
`System.Classes.TStreamReader`	This is a reader for a stream of characters
`System.Classes.TStringReader`	This is a reader for a string
`System.Classes.TTextReader`	This is a reader for sequence of characters; it is an abstract class
`System.Classes.TBinaryReader`	This is a reader for binary data

You can check the complete list and their intended use directly on the Embarcadero website at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Streams,_Reader_and_Writers`.

As Joel Spolsky (`http://www.joelonsoftware.com/articles/Unicode.html`) says, "You can no longer pretend that plaintext is ASCII", so while we write streams, we've to pay attention to which encoding our text has and which encoding our counterpart is waiting for. One of the most frequent necessities is to efficiently read and write a text file using the correct encoding.

> *"The Single Most Important Fact About Encodings"*
>
> *It does not make sense to have a string without knowing what encoding it uses. You can no longer stick your head in the sand and pretend that "plain" text is ASCII.*
>
> *--Joel Spolsky*

The point Joel is making is that the content of a string doesn't know about the type of character encoding it uses.

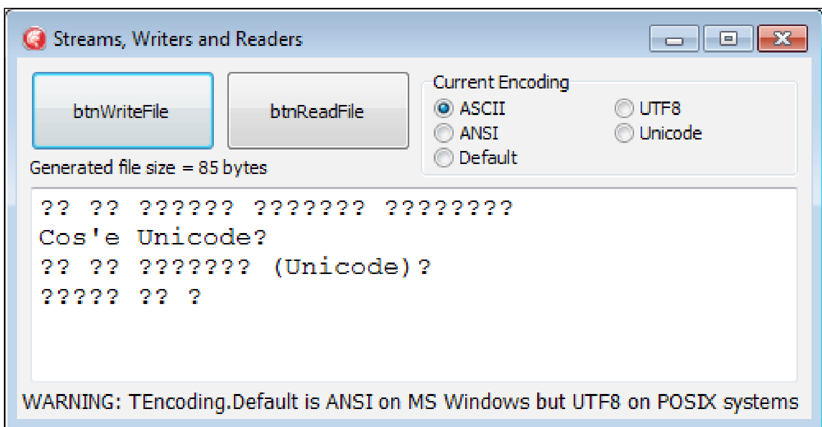
When you think about file handling, ask yourself: could this file become 10 MB? And 100 MB? 1 GB? How will my program behave in that case? Handling a file one line at time and not loading all the files contents in memory is usually a good insurance for these cases. A stream of data is a good way to do this kind of thing. In this recipe, we'll see the practical utilization of streams, stream writers, and streams readers.

How to do it...

The project is not complex, all the interesting stuff happens in the `btnWriteFile` and `btnReadFile` files.

To write the file, we use `TStreamWriter`. The `TStreamWriter` class (as its counterpart `TStreamReader`) is a wrapper for a `TStream` descendent and adds some useful high-level methods to write to the stream. There are a lot of overloaded methods (`Write/WriteLine`) to allow an easy writing to the underlying stream. However, you can access the underlying stream using the `BaseStream` property of the wrapper. Just after writing the file, the memo reloads the file using the same encoding used to write it and shows it. This is only a fast check for this recipe, you don't need the `TMemo` component at all in your real project. The `btnReadFile` file simply opens the file using a stream and passes the stream to a `TStreamReader` that, using the right encoding, reads the file one line at time.

Now, let's do some checks. Run the program and with the encoding set to **ASCII**, click on **btnWriteFile**. The memo will show garbage text, as shown in the following screenshot. This is because we are using the wrong encoding for the data we are writing in the file.



Garbage text written to the file using the wrong encoding. No one line of text is equal to the original one. It is necessary to know the encoding for the text before writing and reading it.

www.allitebooks.com

Now select **UTF8** from the RadioGroup and retry. Clicking on **btnWriteFile**, you will see the correct text in the memo. Try to change the **Current Encoding** setting using **ASCII** and click on **btnReadFile**. You will still get garbage text. Why? Because the file has been read with the wrong encoding. You have to know the encoding before safely reading the file contents. To read the text that we wrote, we have to use the very same encoding. Play with the other encodings to see different behaviors.

There's more...

Streams are very powerful and their uniform interface helps us to write portable and generic code. With the help of streams and polymorphism, we can write code that uses a `TStream` component to do some work without knowing which kind of stream it is!

Also, a lesser known possibility, if you ever write a program that needs to access to the good-old `STD_INPUT`, `STD_OUTPUT`, or `STD_ERROR`, you can use `THandleStream` to wrap these system handles to a nice `TStream` interface with the following code:

```
program StdInputOutputError;
//the following directive instructs the compiler to create a
//console application and not a GUI one, which is the default.
{$APPTYPE CONSOLE}
uses
  System.Classes, // required for Stream classes
  Winapi.Windows; // required to have access to the STD_* handles
var
  StdInput: TStreamReader;
  StdOutput, StrError: TStreamWriter;
begin
  StdInput := TStreamReader.Create(
               THandleStream.Create(STD_INPUT_HANDLE));
  StdInput.OwnStream;
  StdOutput := TStreamWriter.Create(
                THandleStream.Create(STD_OUTPUT_HANDLE));
  StdOutput.OwnStream;
  StdError := TStreamWriter.Create(
               THandleStream.Create(STD_ERROR_HANDLE));
  StdError.OwnStream;
  { HERE WE CAN USE OURS STREAMS }
  // Let's copy a line of text from STD_IN to STD_OUT
  StdOutput.writeln(StdInput.ReadLine);
  { END - HERE WE CAN USE OURS STREAMS }
  StdError.Free;
  StdOutput.Free;
  StdInput.Free;
end.
```

Moreover, when you work with file-related streams, the `TFile` class (contained in `System.IOUtils.pas`) is very useful, and it has some helper methods to write shorter and more readable code.

Putting your VCL application in the tray

Some applications are designed to be always in the Windows tray bar. For almost all their running time, the user knows where that particular application is in the tray. Think about antivirus, custom audio processors, and video management tools provided by hardware vendors and many other things. Instead, some other applications need to go in the tray only when a long operation is running and the user should otherwise attend in front of a boring *please wait* animation. In these cases, users will be very happy if our application is not blocked and lets them do some other things. Then, a not intrusive notification will bring up an alert if some thing interesting happens. Think about heavy queries, statistics, heavy report generation, file upload or download, or huge data import or export. Think for a second: what if Google Chrome showed one modal dialog with a message **Please wait, while this 2 GB file is downloading...** stopping you to navigate to other pages? Crazy! Many applications could potentially behave like this.

In such cases, the users knows that they have to wait, but the application should be so "polite" as to let them do other things. Usually, programmers think that their software is the only reason the user bought a computer. Very often, this is not the case. So, let's find a way to do the right thing at the right moment.

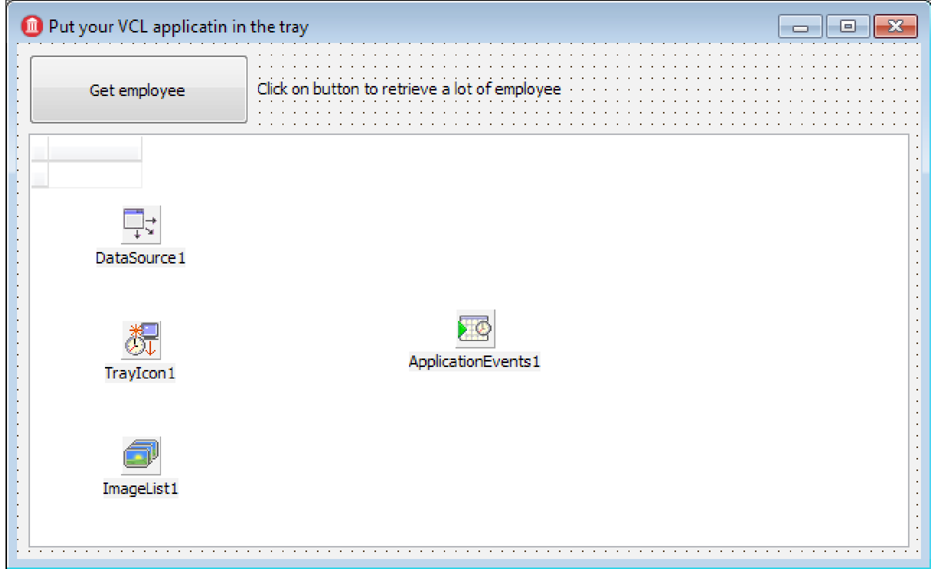
Getting ready

This recipe is about creating a good Windows citizen application. Let's say our application allows us to execute a huge search in a database. When the user starts this long operation, the application UI remains usable. During the request execution, the user can decide to wait in front of the form or minimize it to the taskbar. If the user minimizes the application window, it also goes on the tray bar and when the operation finishes and it will alert the user with a nonintrusive message.

How to do it...

1. Create a new VCL application and drop on it a **TButton**, a **TLabel**, a **TTrayIcon**, a **TApplicationEvents**, a **TImagelist**, a **TDataSource**, and a **TDBGrid** component. Connect the **TDBGrid** to the **TDataSource**. Leave the default component names (I'll refer to the components using their default names). Use the disposition and the captions to make the form similar to the following screenshot:

www.allitebooks.com



The form and its components as they should look

2. In the `implementation` section of the unit, add the following units:
   - `AnonThread`: Add this unit to the project (this is located under `C:\Users\Public\Documents\Embarcadero\Studio\14.0\Samples\Object Pascal\RTL\CrossPlatform Utils` on my machine). You can avoid adding this unit in the project and add the path to the IDE library path by navigating to **Tools** | **Options** and then clicking on **Delphi Options** | **Library**.
   - `RandomUtilsU`: Add this unit to the project (this is located under the `Commons` folder of the recipes).
   - `FireDAC.Comp.Client`: Add this unit in the `implementation uses` section of the form.

3. We'll start with the code that will actually do the heavy work. In the `Button1.OnClick` method, put this code:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  I: Integer;
  ds: TDataSet;
begin
  Button1.Enabled := False;

  if Assigned(DataSource1.DataSet) then
  begin
    ds := DataSource1.DataSet;
```

```
    DataSource1.DataSet := nil;
    RemoveComponent(ds);
    FreeAndNil(ds);
  end;

  Label1.Caption := 'Data retrieving... may take a while';
  TAnonymousThread<TFDMemTable>.Create(
    function: TFDMemTable
    var
      MemTable: TFDMemTable;
      I: Integer;
    begin
      Result := nil;
      MemTable := TFDMemTable.Create(nil);
      try
        MemTable.FieldDefs.Add('EmpNo', ftInteger);
        MemTable.FieldDefs.Add('FirstName', ftString, 30);
        MemTable.FieldDefs.Add('LastName', ftString, 30);
        MemTable.FieldDefs.Add('DOB', ftDate);
        MemTable.CreateDataSet;
        for I := 1 to 400 do
        begin
          MemTable.AppendRecord([
            1000 + Random(9000),
            GetRndFirstName,
            GetRndLastName,
            EncodeDate(1970, 1, 1) + Random(10000)
            ]);
        end;
        MemTable.First;
        //just mimic a slow operation
        TThread.Sleep(2*60*1000);
        Result := MemTable;
      except
        FreeAndNil(MemTable);
        raise;
      end;
    end,
    procedure(MemTable: TFDMemTable)
    begin
      InsertComponent(MemTable);
      DataSource1.DataSet := MemTable;
      Button1.Enabled := True;
      Label1.Caption :=
```


www.allitebooks.com

```
          Format('Retrieved %d employee',
          [MemTable.RecordCount]);
        ShowSuccessBalloon(Label1.Caption);
      end,
      procedure(Ex: Exception)
      begin
        Button1.Enabled := True;
        Label1.Caption := Format('%s (%s)',
                  [Ex.Message, Ex.ClassName]);
        ShowErrorBalloon(Label1.Caption);
      end);
end;
```

4. Now, create the following event handler for the `Tray1.OnBalloonClick` method and connect it to the `Tra1.OnDoubleClick` event handler:

```
procedure TMainForm.TrayIcon1BalloonClick(Sender: TObject);
begin
  TrayIcon1.Visible := False;
  WindowState := wsNormal;
  SetWindowPos(Handle, HWND_TOPMOST, 0, 0, 0, 0,
         SWP_NOSIZE or SWP_NOMOVE);
  SetWindowPos(Handle, HWND_NOTOPMOST, 0, 0, 0, 0,
         SWP_NOSIZE or SWP_NOMOVE);
end;
```

5. In the next step, the two raw `SetWindowPos` calls will be less obscure, believe me.
6. Now, to keep things clear, we need the following two procedures. Create them as private methods of the form:

```
procedure TMainForm.ShowErrorBalloon(const Mess: String);
begin
  if TrayIcon1.Visible then
  begin
    TrayIcon1.IconIndex := 2;
    TrayIcon1.BalloonFlags := bfError;
    TrayIcon1.BalloonTitle := 'Errors occurred';
    TrayIcon1.BalloonHint := Label1.Caption;
    TrayIcon1.ShowBalloonHint;
  end;
end;

procedure TMainForm.ShowSuccessBalloon(const Mess: String);
begin
  if TrayIcon1.Visible then
  begin
```

```
      TrayIcon1.IconIndex := 0;
      TrayIcon1.BalloonFlags := bfInfo;
      TrayIcon1.BalloonTitle := 'Request terminated';
      TrayIcon1.BalloonHint := Label1.Caption;
      TrayIcon1.ShowBalloonHint;
    end;
  end;
```

7. Create one last event handler for the `ApplicationEvents1.OnMinimize` method:

```
procedure TMainForm.ApplicationEvents1Minimize(
   Sender: TObject);
begin
  TrayIcon1.Visible := True;
  TrayIcon1.BalloonTitle := 'Employee Manager';
  TrayIcon1.BalloonHint :=
    'Employee Manager is still running in the tray.' +
      sLineBreak +
      'Reactivate it with a double click on the tray icon';
  TrayIcon1.BalloonFlags := bfInfo;
  TrayIcon1.ShowBalloonHint;
  TrayIcon1.IconIndex := 0;
end;
```

8. Run the application by hitting *F9* (or navigate to **Run** | **Run**).
9. Click on the **Get Employee** button and then minimize the application (note that as the GUI is responsive, you can resize, minimize, and maximize the form).
10. An icon is shown in the tray and shows a message about what the application is doing.
11. As soon as the data has been retrieved, a **Request terminated** message will pop up. Click on the balloon. The application will come to the front and you will see the data in the **TDBGrid**.
12. Try to repeat the procedure without minimizing the window. All is working as usual (this time without the tray messages) and the GUI is responsive.

How it works...

This recipe is a bit articulated. Let's start from the beginning.

The actual code that executes the request uses a nice helper class provided by Embarcadero in the `Samples` folder of RADStudio (not officially supported, it is just an official sample). The `TAnonymousThread<T>` constructor is a class that simplifies the process of starting a thread and when the thread ends, this class updates the UI with data retrieved by the thread.

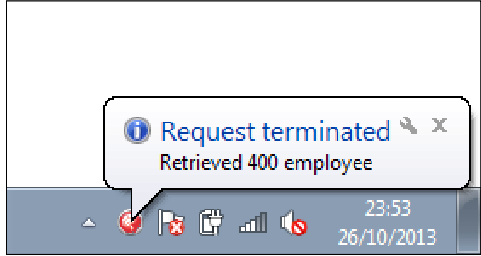
www.allitebooks.com

The `TAnonymousThread<T>` constructor (there are other overloads, but this the most used) expects three anonymous methods:

- `function: T`: This function is executed in the background thread context created internally (so you should avoid accessing the UI). Its `Result` value will be used after the thread execution.
- `procedure (Value: T)`: This procedure is called after the thread is executed. Its `input` parameter is the result value of the first function. This procedure is executed in the context of the main thread, so it can update the UI safely. It is not called in the case of an exception raised by the first function.
- `procedure (E: Exception)`: This procedure is called in the case of an exception during the thread execution and is executed in the context of the main thread, so it can update the UI safely. It is not called if there isn't an exception during thread execution.

The background thread (the first function passed to the `TAnonymousThread<T>` constructor) creates a memory table using the `TFDMemTable` component (we will talk about this component in the FireDAC-related section) and then that object is passed to the second anonymous method that adds it to the form's components using the `InsertComponent()` method and binds it to the **DBGrid** causing the data visualization.

When the data is ready in the grid, a call to the `ShowSuccessBalloon()` function shows a balloon message in the tray area, informing users that their data is finally available. If the user clicks on the balloon (or double-clicks on the tray icon), the application is restored. The balloon message is shown in the following screenshot:



The balloon message when the data are ready in DBGrid

If the user clicks on the balloon, the form is restored. However, since Windows XP (with some variation in subsequent versions), the system restricts which processes can set the foreground window. An application cannot force a window to the foreground while the user is working with another window. The calls to `SetWindowPos` are needed to bring the form to the front.

In the included code, there is also another version of the recipe (`20_VCLAppFlashNotification`) that uses the most recent flash on the taskbar to alert the user. Consider this approach when you want to implement an application that, when minimized, has to alert the user in some way. The tray area may become rapidly crowded with icons. So consider to flash your icons in the taskbar instead.

The other code is required to correctly handle the memory ownership of the `TFDMemTable` instance.

There's more...

The use of a tray icon is a well-known pattern in Windows development. However, the concept of *I'll go into the background for a while, if you want, and I'll show you the notification as soon something happens* is used very often on Android, iOS, and Mac OS X. In fact, some part of this recipe code is reusable also on Mac OS X, iOS, and Android. Obviously, using the right system to alert the user when the background thread finishes (for example, on a mobile platform) execution should use the notification bar. The thread handling of this recipe works on every platform supported by Delphi.

Creating a Windows service

Some kind of application needs to be running H24. Usually, these are network servers or data transfer / monitoring applications. In these cases, you probably start with a normal GUI or console application; however, when the systems are to be used in production, you face a lot of problems related to the Windows session termination, reboots, user rights, and other issues related to the server environment.

Getting ready

The way to go, in the previous scenario, is to develop a Windows service. In this recipe, we'll see how to write a good Windows service scaffold and this can be the skeleton for many other services, so feel free to use this code as a template to create all services that you will need.

How to do it...

The project has been created starting from the default project template accessible from **File** | **New** | **Other** | **Delphi Projects** | **Service Application** and then has been integrated with a set of functionalities to make it real.

All the low-level interfacing with **Windows Service Manager** is done by the `TService` class. In the `ServiceU.pas` component, there is the actual descendant of `TService` that represents the Windows service we are implementing. Its event handlers are used to communicate with the operating system.

www.allitebooks.com

Usually, a service needs to respond to the Windows `ServiceController` commands independently of what it is doing, so we need a background thread to do the actual work, while the `TService.OnExecute` event should not do any real work (this is not a must, but usually is the way to go). The unit named `WorkerThreadU.pas` contains the thread and the main service needed to hold a reference to the instance of this thread.

The background thread starts when the service is started (the `OnStart` event) and stops when the service is stopped (the `OnStop` event). The `OnExecute` event waits and handles the `ServiceController` commands but doesn't do any actual functional work. This is done using the `ServiceThread.ProcessRequests(false);` event in a `while` loop.

Usually, the `OnExecute` event handler looks like the following:

```
procedure TSampleService.ServiceExecute(Sender: TService);
begin
  while not Terminated do
  begin
    ServiceThread.ProcessRequests(false);
    TThread.Sleep(1000);
  end;
end;
```

The waiting time of `1000` milliseconds is not a must, but consider that the wait time should be not too high because the service needs to be responsive to the Windows service controller messages, and not too low because, otherwise, the thread context switch may waste resources.

The background thread writes a line in a logfile once a second. While it is in a `Paused` state, the service stops writing. When the service continues, the thread will restart writing the log line. In the service event handlers, there is a logic to implement this change of state:

```
procedure TSampleService.ServiceContinue(Sender: TService;
   var Continued: Boolean);
begin
   FWorkerThread.Continue;
   Continued := True;
end;

procedure TSampleService.ServicePause(Sender: TService;
   var Paused: Boolean);
begin
   FWorkerThread.Pause;
   Paused := True;
end;
```

In the thread, there is the actual logic to implement the `Paused` state and in this case, it is fairly simple; we've to pause the writing of the logfile.

Here's an extract:

```
Log := TStreamWriter.Create(
           TFileStream.Create(LogFileName,
                   fmCreate or fmShareDenyWrite));
try
  while not Terminated do
  begin
    if not FPaused then
    begin
      Log.WriteLine('Message from thread: ' + TimeToStr(now));
    end;
    TThread.Sleep(1000);
  end;
finally
  Log.Free;
end;
```

The `boolean` instance variable `FPaused` can be considered as a thread safe for this use.

Delphi services don't have a default description under **Windows Service Manager**. If we want to give a description, we have to write a specific key in the Windows registry. Usually, this is done in the `AfterInstall` event. In our service, write the following code in the `AfterInstall` event handler:

```
procedure TSampleService.ServiceAfterInstall(
   Sender: TService);
var
   Reg: TRegistry; //declared in System.Win.Registry;
begin
   Reg := TRegistry.Create(KEY_READ or KEY_WRITE);
  try
       Reg.RootKey := HKEY_LOCAL_MACHINE;
       if Reg.OpenKey(
         '\SYSTEM\CurrentControlSet\Services\' + name,
         False {do not create if not exists}) then
    begin
         Reg.WriteString('Description',
        'My Fantastic Windows Service');
      Reg.CloseKey;
    end;
  finally
       Reg.Free;
  end;
end;
```

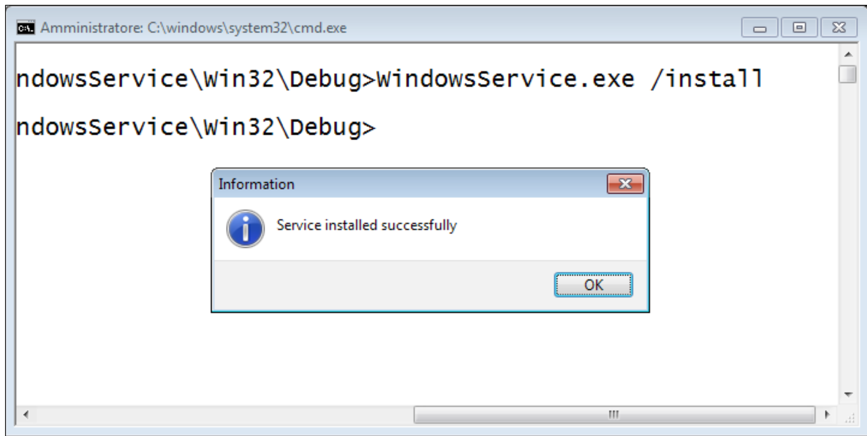

www.allitebooks.com

It is not necessary to delete this key in the AfterUnInstall event because Windows deletes all the keys related to the service (under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<MyServiceName>) when it is actually uninstalled.

Let's try an installation. Build the project, open the Windows command prompt, and go to the folder where the project has been built and run this command:

```
C:\<ExeProjectPath>\WindowsService.exe /install
```

If all is okay, you should see this message:



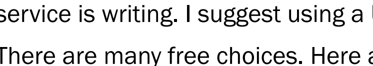
The service installation is okay

Now, you can check in the **Windows Services Console**. You should find the service installed. Click on **Start**, wait for the confirmation, and the service should start to write its logfile.

Play with **Pause**, **Continue**, and check the file activity.

Some text editors could have problem with opening the logfile while the service is writing. I suggest using a Unix tail clone for Windows.

There are many free choices. Here are some links:



- http://sourceforge.net/projects/tailforwin32/
- http://ophilipp.free.fr/op_tail.htm
- http://www.baremetalsoft.com/baretail/

There's more...

Windows services are very powerful. Using the abstractions that Delphi provides, you can also create an application that can act as a normal GUI application or as a Windows service after reading a parameter on the command line.

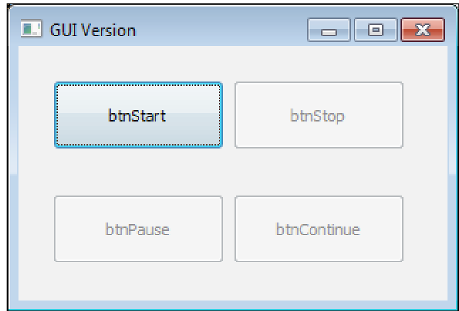
In the recipe folder, there is another recipe called `20_WindowsServiceOrGUI`.

This application can be used as a normal Windows service using the normal command-line switches used so far, but if launched with `/GUI`, it acts as a GUI application and can use the same application code (not `TService`). In our example, the GUI version uses the same worker thread as the service version. This can be very useful for debugging purposes.

Run the application with the following command:

```
C:\<ExeProjectPath>\WindowsServiceOrGUI.exe /GUI
```

You will get a GUI version of the service, as shown in the following screenshot:



The GUI version of the Windows service

Using the TService.LogMessage method

If something happens during the execution of your service which you want to log and you want to log into the system logger, you can use the `LogMessage` method to save a message, which can be viewed later using Windows built-in event viewer.

You can call the `LogMessage` method using appropriate logging type:

```
LogMessage('Your message goes here for SUCCESS',
          EVENTLOG_SUCCESS, 0, 1);
```

If you check the event in **Event Viewer**, you will find a lot of garbage text that complains about the lack of description for the event.

www.allitebooks.com

If you really want to use **Event Viewer** to view your log messages (when I can, I use a text logfile and don't care about **Event Viewer**, but there are scenarios where **Event Viewer** log is needed), you have to use the **Microsoft Message Compiler**.

The Microsoft Message Compiler is a tool able to compile a file of messages in a set of RC files. Then those files must be compiled by a resource compiler and linked to your executable.

More information about message compiler and steps needed to provide the needed description for the log event can be found at the following link:

`http://www.codeproject.com/Articles/4166/Using-MC-exe-message-resources-and-the-NT-event-lo`

Associating a file extension with your application on Windows

In some cases, your fantastic application needs to be opened with just a *double-click* on a file with an extension associated with it. This is the case with MS Word, MS Excel, and many other well-known pieces of software. If you have a file generated with a program, double-click on the file and the program that generated the file will bring up pointing to that file. So, if you click on a `mywordfile.docx` file, MS Word will be opened and the `mywordfile.docx` file's content will be shown. This is what we'd like to do in this recipe. The association can be useful also when you have multiple configurations for a program. Double-click on the `ConfigurationXYZ.myext` file and the program will start using that configuration.

Getting ready

The hard work is done by the operating system itself. We have to instruct Windows to provide the following information:

- The file extension to associate
- The description of file type (this will be shown by Windows Explorer describing the file type)
- The default icon for the file type (in this recipe, we'll use the application icon itself, but it is not mandatory)
- The application that we want to associate

Let's start!

How to do it...

1. Create a new VCL application and drop two **TButton** components and a **TMemo** component. Align all the buttons as a toolbar at the top of the form and the memo to all the remaining form client area.
2. The button on the left-hand side will be used to register a file type while the button on the right-hand side will be used to unregister the association (cleaning the registry).
3. We have to handle some MS Windows-specific features, so we need some Windows-related units. Under the `implementation` section of the unit, write this use clause:

```
uses System.Win.registry, Winapi.shlobj, System.IOUtils;
```

4. In the `implementation` section, we need two procedures to do the real work; so just after the `uses` clause, put this code:

```
procedure UnregisterFileType(
   FileExt: String;
   OnlyForCurrentUser: boolean = true);
var
  R: TRegistry;
begin
  R := TRegistry.Create;
  try
    if OnlyForCurrentUser then
      R.RootKey := HKEY_CURRENT_USER
    else
      R.RootKey := HKEY_LOCAL_MACHINE;

    R.DeleteKey('\Software\Classes\.' + FileExt);
    R.DeleteKey('\Software\Classes\' + FileExt + 'File');
  finally
    R.Free;
  end;
  SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0);
end;

procedure RegisterFileType(
   FileExt: String;
   FileTypeDescription: String;
   ICONResourceFileFullPath: String;
   ApplicationFullPath: String;
   OnlyForCurrentUser: boolean = true);
var
  R: TRegistry;
begin
```


www.allitebooks.com

```
R := TRegistry.Create;
try
  if OnlyForCurrentUser then
    R.RootKey := HKEY_CURRENT_USER
  else
    R.RootKey := HKEY_LOCAL_MACHINE;

  if R.OpenKey('\Software\Classes\.' + FileExt,
       true) then begin
    R.WriteString('', FileExt + 'File');
    if R.OpenKey('\Software\Classes\' + FileExt + 'File',
             true) then begin
      R.WriteString('', FileTypeDescription);
      if R.OpenKey('\Software\Classes\' +
       FileExt + 'File\DefaultIcon', true) then
      begin
        R.WriteString('', ICONResourceFileFullPath);
        if R.OpenKey('\Software\Classes\' +
             FileExt + 'File\shell\open\command',
              true) then
          R.WriteString('',
             ApplicationFullPath + ' "%1"');
      end;
    end;
  end;
finally
  R.Free;
end;
SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, 0, 0);
end;
```

5. These two procedures allows us to register (and unregister) a file type considering only the current user or all the machine users. Note that if you want to register the association for every user, write your data to the following location:

   `HKEY_LOCAL_MACHINE\Software\Classes`

6. If you want to register the association for the current user only, write your data to the following location:

   `HKEY_CURRENT_USER\Software\Classes`

7. On the newest Windows versions, you need admin rights to register a file type for all the machine users. The last line of the procedures tells Explorer (the MS Windows graphic interface) to refresh its setting to reflect the changes made to the file associations. As a result, for instance, the Explorer file list views will be updated.

8. We've almost finished. Change the left-hand side button name to `btnRegister`, the right-hand side button name to `btnUnRegister`, and put the following code on their `onclick` event handlers:

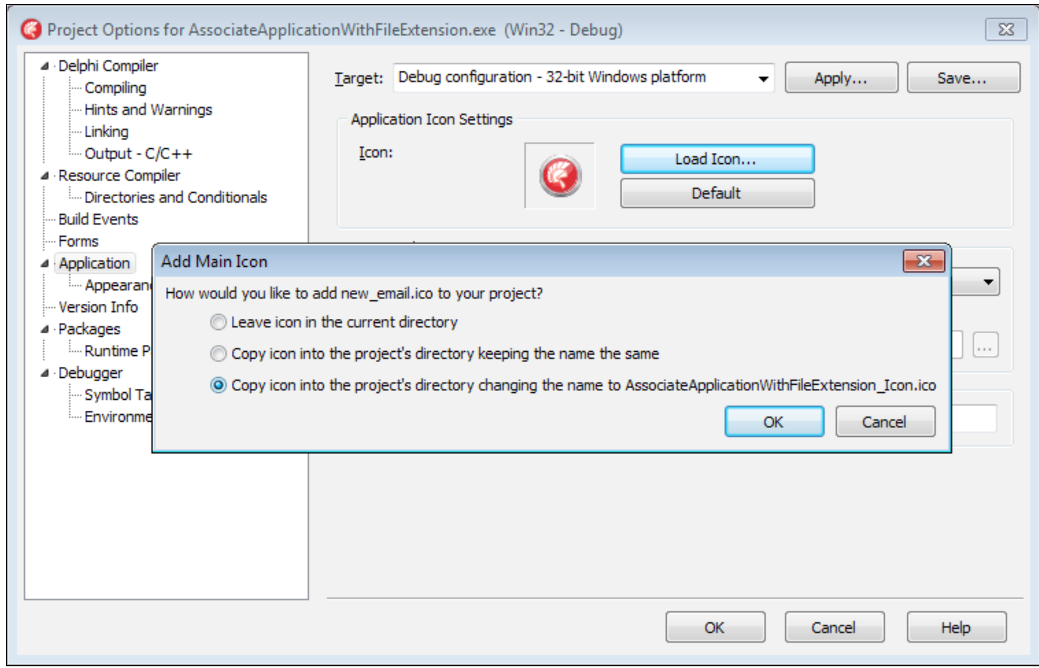
```
procedure TMainForm.btnRegisterClick(Sender: TObject);
begin
  RegisterFileType(
    'secret',
    'This file is a secret',
    Application.ExeName,
    Application.ExeName,
    true);
  ShowMessage('File type registered');
end;

procedure TMainForm.btnUnRegisterClick(Sender: TObject);
begin
  UnregisterFileType('secret', true);
  ShowMessage('File type unregistered');
end;
```

9. Now, when our application is invoked with a double-click, we'll get the file name as a parameter. It is possible to read a parameter passed by Windows Explorer (or command line) using the `ParamStr(1)` function. Create a `FormCreate` event handler using the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  if TFile.Exists(ParamStr(1)) then
    Memo1.Lines.LoadFromFile(ParamStr(1))
  else
  begin
    Memo1.Lines.Text := 'No valid secret file type';
  end;
end;
```

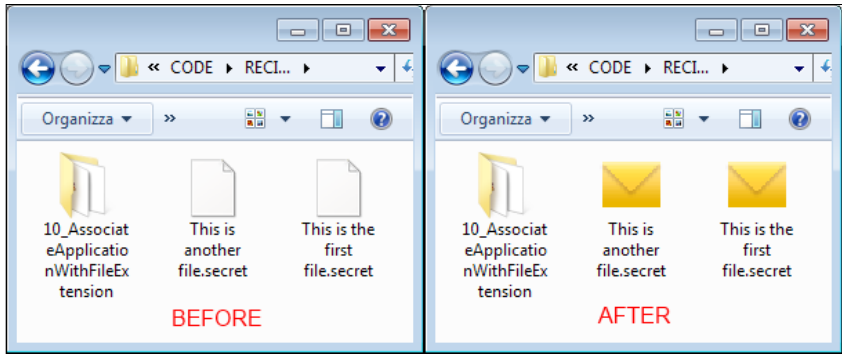
10. Now the application should be complete. However, a nice integration with the operating system requires a nice icon as well. In the code, the associated file will get the same icon as the main program, so let's change our default icon by navigating to **Project** | **Options** | **Application** and choose a nice icon. Click on the **Load Icon** button, choose an ICO file, and then select the third item from the resultant dialog:

www.allitebooks.com



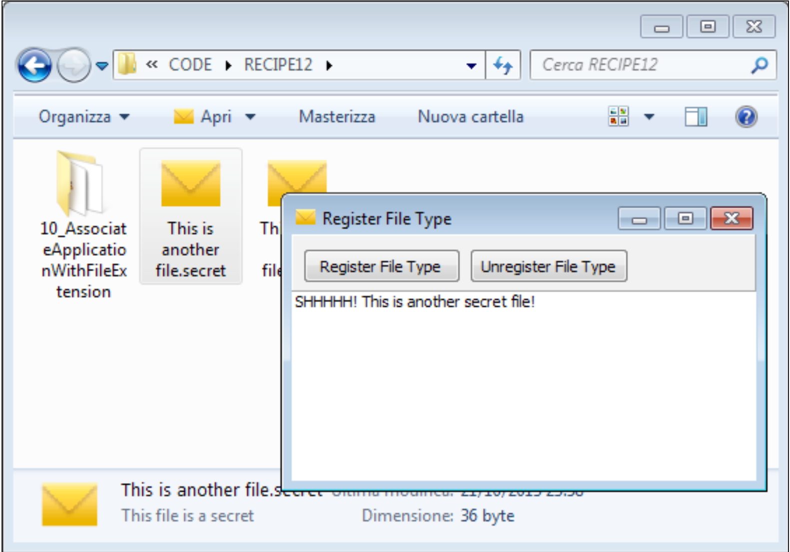
Changing the default application icon for our application

11. Now, create some text files with our registered `.secret` extension.
12. These files will appear with the default Windows icon, but in some seconds, they will have a brand new icon.
13. Run the application by hitting *F9* (or navigate to **Run** | **Run**).
14. Click on the **btnRegister** button and close the application. Now the files get a new icon, as shown in the following screenshot:



The files in Windows Explorer before and after having registered the .secret extension

15. Now, with the application not running, double-click on the `.secret` file. Our program will be started by Windows itself, using the information stored in the registry about the `.secret` file, and we'll get this form (the text shown in the memo is the text contained into the file):



Our application, launched by the operating system, while showing the content of the file

There's more...

One application can register many file types. In some cases, I've used this technique to register some specific desktop database files to my application (FirebirdSQL Embedded database files or SQLite database files). So a double-click on such database file (registered with an application-specific extension) was actually a connection to that database.

www.allitebooks.com

2
Become a Delphi Language Ninja

In this chapter, we will cover the following recipes:

- Fun with anonymous methods – using higher-order functions
- Writing enumerable types
- RTTI to the rescue – configuring your class at runtime
- Duck typing using RTTI
- Creating helpers for your classes
- Checking strings with regular expressions

Introduction

This chapter explains some of the not-so-obvious features of the language and the RTL that every Delphi programmer should know. There are ready-to-use recipes that will be useful every day and have been selected over many others.

Fun with anonymous methods – using higher-order functions

Since Version 2009, the Delphi language (or better, its Object Pascal dialect) supports anonymous methods. What's an anonymous method? Not surprisingly, an anonymous method is a procedure or a function that does not have an associated name.

An anonymous method treats a block of code just like a *value* so that it can be assigned to a variable or used as a parameter to a method or returned by a function as its result value. In addition, an anonymous method can refer to variables and bind values to the variables in the context scope in which the anonymous method is defined. Anonymous methods are similar to closures defined in other languages such as JavaScript or C#. An anonymous method is declared as a reference to a method:

```
type
  TFuncOfString = reference to function(S: String): String;
```

Anonymous methods (or anonymous functions) are convenient to pass as an argument to a higher-order function. What's a higher-order function?

Wikipedia gives the following explanation (`http://en.wikipedia.org/wiki/Higher-order_function`):

> *In mathematics and computer science, a higher-order function (also functional form, functional, or functor) is a function that does at least one of the following:*
>
> *- Takes one or more functions as an input*
>
> *- Outputs a function*
>
> *All other functions are first-order functions.*

Getting ready

In this recipe, you'll see how to use Delphi's anonymous methods with some of the more popular and useful higher-order functions:

- **Map**: This is available in many functional programming languages. This takes as arguments a `func` function and a list of elements `list`, and returns a new list with `func` applied to each element of `list`.
- **Reduce**: This is also known as `Fold`. This requires a combining function, a starting point of a data structure, and possibly some default values to be used under certain conditions. The `reduce` function proceeds to combine elements of the data structure using the injected function.

  This is used to do operations on a set of values to get only one result (or a smaller set of values) that represent the *reduction* of that initial data. For example, the values 1, 2, and 3 can be reduced to the single value 6 using criteria of `SUM`.
- **Filter**: This requires a data structure and a filter condition. This returns all the elements in the structure that match the filter condition.

www.allitebooks.com

How to do it...

For the `HigherOrderFunctions.dproj` project, the actual high-order functions are implemented in the `HigherOrderFunctionsU.pas` unit as generic class functions as shown here:

```
type
  HigherOrder = class sealed
    class function Map<T>(InputArray: TArray<T>;
      MapFunction: TFunc<T, T>): TArray<T>;
    class function Reduce<T: record>(InputArray: TArray<T>;
      ReduceFunction: TFunc<T, T, T>; InitValue: T): T;
    class function Filter<T>(InputArray: TArray<T>;
      FilterFunction: TFunc<T, boolean>): TArray<T>;
  end;
```

Let's analyze each of these functions.

The `Map` function requires a list of `T` parameters as its input data structure and an anonymous method that accepts and returns the same type of data `T`. For each element of the input data structure, the `MapFunction` is called and another list of data is built to contain all its results.

This is the body of the `Map` function.

```
class function HigherOrder.Map<T>(InputArray: TArray<T>;
    MapFunction: TFunc<T, T>): TArray<T>;
var
  I: Integer;
begin
  SetLength(Result, length(InputArray));
  for I := 0 to length(InputArray) - 1 do
    Result[I] := MapFunction(InputArray[I]);
end;
```

The main form uses the `Map` function in the following way:

```
procedure TMainForm.btnMapCapitalizeClick(Sender: TObject);
var
  InputData, OutputData: TArray<string>;
begin
  //let's generate some sample data
  InputData := GetStringArrayOfData;

  //call the map function on an array of string
  OutputData := HigherOrder.Map<string>(
    InputData,
```

```
    function(Item: String): String
    begin
      //this is the "map" criteria that will be applied to each
      //item to capitalize the first word in the item
      Result := String(Item.Chars[0]).ToUpper + Item.Substring(1);
    end);

  //fill the related listbox with the results
  FillList(OutputData, lbMap.Items);
end;
```

The `Reduce` function requires a list of `T` as its input data structure and an anonymous method that accepts two parameters of type `T` and returns a value of type `T`. It can also be passed a `DefaultFor` each element of the input data structure, the `ReduceFunction` is called passing the intermediate result calculated so far and the current element of the list. After the last call, the result is returned to the caller function.

This is the body of the `Map` function:

```
class function HigherOrder.Reduce<T>(
  InputArray: TArray<T>;
  ReduceFunction: TFunc<T, T, T>; InitValue: T): T;
var
  I: T;
begin
  Result := InitValue;
  for I in InputArray do
  begin
    Result := ReduceFunction(Result, I);
  end;
end;
```

The main form uses the `Reduce` function in the following way:

```
procedure TMainForm.btnReduceSumClick(Sender: TObject);
var
  InputData: TArray<Integer>;
  OutputData: Integer;
begin
  InputData := GetIntArrayOfData;
  //sum the input data using as starting value 0
  OutputData := HigherOrder.Reduce<Integer>(InputData,
    function(Item1, Item2: Integer): Integer
    begin
      Result := Item1 + Item2;
```


www.allitebooks.com

```
    end, 0);
  lbReduce.Items.Add('SUM: ' + OutputData.ToString);
end;
```

The last implemented function is `Filter`. The `Filter` function requires a list of `T` as its input data structure and an anonymous method accepts a single parameter of type `T` and returns a Boolean value. This anonymous method represents the filter criteria that will be applied to the input data. For each element of the input data structure, the `FilterFunction` is called and if it returns `true`, then the current element will be in the returning list, but not otherwise. After the last call, the filtered list is returned to the caller function.

Here is the body of the `Filter` function:

```
class function HigherOrder.Filter<T>(InputArray: TArray<T>;
    FilterFunction: TFunc<T, boolean>): TArray<T>;
var
  I: Integer;
  List: TList<T>;
begin
  List := TList<T>.Create;
  try
    for I := 0 to length(InputArray) - 1 do
      if FilterFunction(InputArray[I]) then
        List.Add(InputArray[I]);
    Result := List.ToArray;
  finally
    List.Free;
  end;
end;
```

The main form uses the `Filter` function to filter only the even numbers. The code is as follows:

```
procedure TMainForm.btnFilterEvenClick(Sender: TObject);
var
  InputData, OutputData: TArray<Integer>;
begin
  InputData := GetIntArrayOfData;
  OutputData := HigherOrder.Filter<Integer>(InputData,
    function(Item: Integer): boolean
    begin
      Result := Item mod 2 = 0; //gets only the even numbers
    end);
  FillList(OutputData, lbFilter.Items);
end;
```

In the recipe's code, there are other utilization samples related to higher-order functions.

There's more...

Higher-order functions are a vast and interesting topic, so in this recipe we only scratched the surface. One of the main concepts is the abstraction of the internal loop over the data structure. Consider this: abstracting the concept of looping, you can implement looping any way you want, including implementing it in a way that scales nicely with extra hardware. A good sample of what can be done using functional programming is the parallel extension of the good `OmniThreadLibrary` (a nice library to simplify multithreading programming) written by Primož Gabrijelčič (`http://www.thedelphigeek.com/`). This is a simple code sample that executes a parallel function for defining the single iteration with an anonymous method and runs it using multiple threads:

```
Parallel.ForEach(1, 100000).Execute(
    procedure (Const elem: integer)
    begin
      //check if the current element is
      //a prime number (can be slow)
      if IsPrime(elem) then
        MyOutputList.Add(elem);
    end);
```

Writing enumerable types

When the `for..in` loop was introduced in Delphi 2005, the concept of enumerable types was also introduced into the Delphi language.

As you know, there are some built-in enumerable types. However, you can create your own enumerable types using a very simple pattern.

To make your container enumerable, implement a single method called `GetEnumerator`, that must return a reference to an object, interface, or record, that implements the following three methods and one property (in the sample, the element to enumerate is `TFoo`):

```
function GetCurrent: TFoo;
function MoveNext: Boolean;
property Current: TFoo read GetCurrent;
```

There are a lot of samples related to standard enumerable types, so in this recipe you'll look at some not-so-common utilizations.

www.allitebooks.com

Getting ready

In this recipe, you'll see a file enumerable function as it exists in other, mostly dynamic, languages. The goal is to enumerate all the rows in a text file without actual opening, reading and closing the file, as shown in the following code:

```
var
  row: String;
begin
  for row in EachRows('..\..\myfile.txt') do
    WriteLn(row);
end;
```

Nice, isn't it? Let's start...

How to do it...

We have to create an enumerable function result. The function simply returns the actual enumerable type. This type is not freed automatically by the compiler so you've to use a value type or an interfaced type. For the sake of simplicity, let's code to return a record type:

```
function EachRows(const AFileName: String): TFileEnumerable;
begin
  Result := TFileEnumerable.Create(AFileName);
end;
```

The `TFileEnumerable` type is defined as follows:

```
type
  TFileEnumerable = record
  private
    FFileName: string;
  public
    constructor Create(AFileName: String);
    function GetEnumerator: TEnumerator<String>;
  end;
. . .
constructor TFileEnumerable.Create(AFileName: String);
begin
  FFileName := AFileName;
end;

function TFileEnumerable.GetEnumerator: TEnumerator<String>;
begin
  Result := TFileEnumerator.Create(FFileName);
end;
```

No logic here; this record is required only because you need a type that has a `GetEnumerator` method defined. This method is called automatically by the compiler when the type is used on the right side of the `for..in` loop.

An interesting thing happens in the `TFileEnumerator` type, the actual enumerator, declared in the implementation section of the unit. Remember, this object is automatically freed by the compiler because it is the return of the `GetEnumerator` call:

```
type
  TFileEnumerator = class(TEnumerator<String>)
  private
    FCurrent: String;
    FFile: TStreamReader;
  protected
    constructor Create(AFileName: String);
    destructor Destroy; override;
    function DoGetCurrent: String; override;
    function DoMoveNext: Boolean; override;
  end;

{ TFileEnumerator }

constructor TFileEnumerator.Create(AFileName: String);
begin
  inherited Create;
  FFile := TFile.OpenText(AFileName);
end;

destructor TFileEnumerator.Destroy;
begin
  FFile.Free;
  inherited;
end;

function TFileEnumerator.DoGetCurrent: String;
begin
  Result := FCurrent;
end;

function TFileEnumerator.DoMoveNext: Boolean;
begin
  Result := not FFile.EndOfStream;
  if Result then
    FCurrent := FFile.ReadLine;

end;
```


www.allitebooks.com

The enumerator inherits from `TEnumerator<String>` because each row of the file is represented as a string. This class also gives a mechanism to implement the required methods.

The `DoGetCurrent` (called internally by the `TEnumerator<T>.GetCurrent` method) returns the current line.

The `DoMoveNext` method (called internally by the `TEnumerator<T>.MoveNext` method) returns true or false if there are more lines to read in the file or not. Remember that this method is called before the first call to the `GetCurrent` method. After the first call to the `DoMoveNext` method, `FCurrent` is properly set to the first row of the file.

The compiler generates a piece of code similar to the following pseudo code:

```
it = typetoenumerate.GetEnumerator;
while it.MoveNext do
begin
  S := it.Current;
  //do something useful with string S
end
it.free;
```

There's more...

Enumerable types are really powerful and help you to write less, and less error prone, code. There are some shortcuts to iterate over in-place data without even creating an actual container.

If you have a bounce or integers or if you want to create a not homogenous for loop over some kind of data type, you can use the new `TArray<T>` type as shown here:

```
for i in TArray<Integer>.Create(2, 4, 8, 16) do
  WriteLn(i);
//write 2 4 8 16
```

`TArray<T>` is a generic type, so the same works also for strings:

```
for s in TArray<String>.Create('Hello','Delphi','World') do
  WriteLn(s);
```

It can also be used for **Plain Old Delphi Object** (**PODO**) or controls:

```
for btn in TArray<TButton>.Create(btn1, btn31,btn2) do
  btn.Enabled := false
```

See also

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Declarations_and_Statements#Iteration_Over_Containers_Using_For_statements`: This Embarcadero documentation will provide a detailed introduction to enumerable types

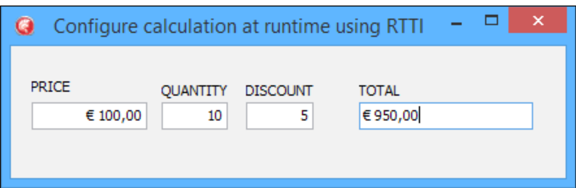
RTTI to the rescue – configuring your class at runtime

Since Delphi 2010, the Delphi RTTI has been greatly expanded. Now it is comparable to what is called Reflection in other languages such as C# or Java. A much-improved RTTI can dramatically change the way you write, or even think about, your code and your architecture. Now, it is possible to write highly flexible code without too much effort.

Getting ready

What we want to do in this recipe is dynamically create a class looking for it by name among the classes that have been linked in the executable (or loaded from dynamic packages). The goal is to change the behavior of the program using only an external file without relying on a lot of parameters and complex configuration code; just create the right class. Wonderful!

Let's say you've developed a program to do orders. Your program allows only one-line orders, so you cannot buy different things in the same orders (this is a sample, man!). The form is shown in this screenshot:



The main form

There is a dataset field connected to each of the `TDBEdit` in the form. The **TOTAL** field is a calculated field and its value is calculated in the `OnCalcFields` dataset.

The calculation is simple: *total = price * quantity * (1 – discount / 100)*

The customer is happy and you are happy as well.

Now, a new big customer, the City Mall, want a customization, "If the total is greater than $1000, apply another 10 percent discount." Ok, you can create the customized version easily. So far so good, but now you have two different versions to maintain.

www.allitebooks.com

Now, another customer, the Country Road Shop says, "If there are more than 10 pieces, the discount must be at least 50 percent." Another customer, Spark Industries, specifies, "Only at the weekend, all the calculated prices will be cut by 50 percent."

Argh! Four customers and four different version of your software to maintain because of customizations! You get the point; at the beginning things are simple, but when you start to customize something, complexity (and bugs) can arise. Let's fix this problem in this recipe.

How to do it...

The simple customization is easy. However, the difficulty comes in when you have to handle which customization you have to choose among those available. You can define some sort of parameters, sure, but your code will get a lot of `if` just to understand which calculation to apply. And, even worse, a change in one of your criteria could break something in another. Bad approach!

We can configure our software without `if` statements using RTTI. In this recipe, all the calculus engines are implemented in four different classes in four different units (you can also define all the criteria in only one unit, but it is not mandatory).

In the following table, there is a summary of the customers and the customizations implemented:

Customer	Unit/class name	Calculation criteria
Default (no customization)	`CalculationCustomerDefaultU` `TCalculationCustomerDefault`	`Result := (Price * Quantity) * (1 - Discount / 100);`

City Mall	`CalculationCustomer_CityMall`
	`TCalculationCustomer_CityMall`
		`Result := (Price * Quantity) * (1 - Discount / 100);`
		`if` `Result > 1000` **`then`**
		`Result := Result * 0.90;`

Country Road Shop	`CalculationCustomer_CountryRoad`
	`TCalculationCustomer_CountryRoad`
		`if` `Quantity > 10` **`then`**
		`if` `Discount < 50`
		`then`
		`Discount := 50;`
		`Result := (Price * Quantity) * (1 - Discount / 100);`

Customer	Unit/class name	Calculation criteria

Spark Industries	`CalculationCustomer_Spark`
	`TCalculationCustomer_Spark`
		`Result := (Price * Quantity) * (1 - Discount / 100);`
		`if DayOfTheWeek(Date) in [1, 7] then`
		`Result :=`
		`Result * 0.50;`

When the program starts, it looks for a configuration file. In the first line of the file, there is a fully qualified class name (`UnitName.ClassName`) that implements the needed calculus criteria. That string is used to create the related class and the instance will be used to calculate the total price when needed. The interesting code is as follows:

```
procedure TMainForm.LoadCalculationEngine;
var
  TheClassName: string;
  CalcEngineType: TRttiType;
const
  CONFIG_FILENAME = '..\..\calculation.config.txt';
begin
  if not TFile.Exists(CONFIG_FILENAME) then
    TheClassName := 'CalculationCustomerDefaultU.' +
    'TCalculationCustomerDefault'
  else
    TheClassName := TFile.ReadAllLines(CONFIG_FILENAME)[0];

  CalcEngineType := FCTX.FindType(TheClassName.Trim);
  if not assigned(CalcEngineType) then
    raise Exception.CreateFmt('Class %s not found',
    [TheClassName]);
  if not CalcEngineType.GetMethod('Create').IsConstructor then
    raise Exception.CreateFmt('Cannot find Create in %s',
    [TheClassName]);

  FCalcEngineObj := CalcEngineType.GetMethod('Create').
  Invoke(CalcEngineType.AsInstance.MetaclassType, []).AsObject;
  FCalcEngineMethod := CalcEngineType.GetMethod('GetTotal');
  Label5.Caption := 'Current Calc Engine: ' + TheClassName;
end;
```


www.allitebooks.com

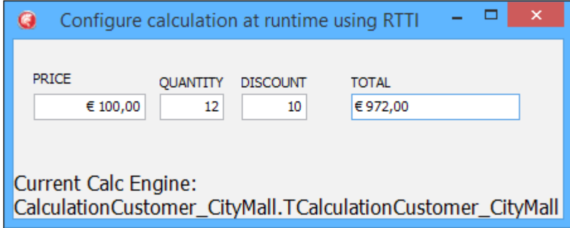
`FCalcEngineObj` is a `TObject` reference that holds your actual calculation engine, while `FCalcEngineMethod` is an RTTI object that keeps reference to the method to call when the calculus is needed.

Now in the dataset `OnCalcFields` event handler, there is this code:

```
procedure TMainForm.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
ClientDataSet1TOTAL.Value :=
  FCalcEngineMethod.Invoke(FCalcEngineObj,
      [ClientDataSet1PRICE.Value,
      ClientDataSet1QUANTITY.Value,
      ClientDataSet1DISCOUNT.Value]).AsCurrency;
end;
```

Run the program and check which calculus engine is loaded. Then stop the program, open the configuration file, and write another `QualifiedClassName` choosing from all those available. Run the program. As you can see, the correct engine is selected and the customization is applied without changing the working code.

On writing the `CalculationCustomer_Spark.TCalculationCustomer_Spark` class in the file, you will get the following behavior:



The main form using the customized calculus engine specified in the configuration file

There's more...

RTTI is a really vast topic. There are endless possibilities to use it in smart ways.

Remember, however, that if the Delphi linker sees that your class is not used in the actual code (because it is used only in the RTTI calls), it could eliminate the class from the executable. So, to be sure that your class will be included in the final executable, write a (even useless) line of code referring to the class. In this recipe, I've included a line of code similar to the following one in every initialization section of the different calculus classes:

```
//. . . other code before

initialization
```

```
//Linker will not remove the class from the final executable
//because now it is used somewhere

TCalculationCustomer_CityMall.ClassName;

end.
```

See also

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Working_with_RTTI_Index`: This documentation from Embarcadero gives more information about extended RTTI

Duck typing using RTTI

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*
>
> *- James Whitcomb Riley*

Clear, isn't it? What may not be so clear is that this approach can be used also in computer programming. Yes, even without an actual duck!

Getting started

Referring to Duck typing, Wikipedia gives the following explanation (`http://en.wikipedia.org/wiki/Duck_typing`):

In computer programming with object-oriented programming languages, duck typing is a style of typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.

How can all these concepts be used in everyday programming? This is the question that this recipe aims to answer.

Let's say that you have a form and you want to inform the user that something bad happened, changing all the *colorable* components to `clRed`. I don't know what the property `Color` means for any control that has that property, I only want to set all the properties named `Color` to `clRed`. How can you achieve this? The naive approach could be to cycle the `Components` property, check whether the current control is a control that I know that has a `Color` property, and then cast that control reference to an actual `TEdit` (or `TComboBox`, `TListBox`, or whatever) reference and change the `Color` property to `clRed`. However, what if tomorrow you need to color another kind of control as well? Or you have to change the `Color` property on `TPanels` but the `Font.Color` property on `TEdits`? You get the point, I think.

www.allitebooks.com

Using the naive approach can raise the complexity of your code. A programmer should hate complexity. More complexity means more time to handle and more time means more money to spend. As usual, the KISS approach is the best one: Keep It Simple, Stupid!

How to do it...

The code in this recipe allows you to write code like the following snippets.

In this snippet, the `Color` property of all controls in the form will be set to `clRed`. I don't know which kind of controls there are on the form, but if they have a property named `Color`, that property will be set to `clRed`:

```
Duck.Apply(Self, 'Color', clRed);
```

In this snippet, the `Caption` property of the controls in the array; if it exists, will be set to `'Hello There'`:

```
Duck.Apply(
  TArray<TObject>.Create(Button1, Button2, Edit1),
  'Caption',
  'Hello There');
```

The following code disables all the `TDataSource` on the form, preventing data editing:

```
Duck.Apply(Self, 'Enabled', False,
  function(Item: TObject): boolean
  begin
    Result := Item is TDataSource;
  end);
```

The following code sets the font name to `Courier New` for some controls:

```
Duck.Apply(TArray<TObject>.Create(Edit1, Edit2, Button2),
    'Font.Name', 'Courier New');
```

This code works for every kind of control. If you change the `TButton` in `TSpeedButton`, it continues to work. If you change a `TListBox` with a `TComboBox`, the code still works. The concept is simple, if you have a property *X* then I'll set that property independent of the actual object type.

Let's see the code that actually does the job.

The main `Duck` class is a mere method container (this is because it doesn't start with the usual T, so it is not a real type) declared as shown in the following code:

```
type
  Duck = class sealed
    class procedure Apply(ArrayOf: TArray<TObject>;
```

```
        PropName: string; PropValue: TValue;
      AcceptFunction: TFunc<TObject, boolean> = nil); overload;
      class procedure Apply(AContainer: TComponent;
        PropName: string; PropValue: TValue;
        AcceptFunction: TFunc<TObject, boolean> = nil); overload;
    end;
```

Methods are very similar and the second one adds some helper to work with `TComponents`; the real job is done by the first one:

```
class procedure Duck.Apply(ArrayOf: TArray<TObject>;
PropName: string; PropValue: TValue;
   AcceptFunction: TFunc<TObject, boolean>);
var
  CTX: TRttiContext;
  Item, PropObj: TObject;
  RttiType: TRttiType;
  Prop: TRttiProperty;
  PropertyPath: TArray<string>;
  i: Integer;
begin
  CTX := TRttiContext.Create;
  try
    for Item in ArrayOf do
    begin
      if (not Assigned(AcceptFunction)) or
        (AcceptFunction(Item)) then
      begin
        RttiType := CTX.GetType(Item.ClassType);
        if Assigned(RttiType) then
        begin
          PropertyPath := PropName.Split(['.']);
          Prop := RttiType.GetProperty(PropertyPath[0]);
          if not Assigned(Prop) then
            Continue;
          PropObj := Item;
          if Prop.GetValue(PropObj).isObject then
          begin
            PropObj := Prop.GetValue(Item).AsObject;
            for i := 1 to Length(PropertyPath) - 1 do
            begin
              RttiType := CTX.GetType(PropObj.ClassType);
              Prop := RttiType.GetProperty(PropertyPath[i]);
              if not Assigned(Prop) then
                break;
```


www.allitebooks.com

```
                if Prop.GetValue(PropObj).isObject then
                  PropObj := Prop.GetValue(PropObj).AsObject
                else
                  break;
              end;
            end;
            if Assigned(Prop) and (Prop.IsWritable) then
              Prop.SetValue(PropObj, PropValue);
          end;
        end;
      end;
    finally
      CTX.Free;
    end;
  end;
```

This is not very simple, I know, but you can see all the pieces we've already talked about. Obviously, we use RTTI to get names and set values of the properties.

The main loop cycles over the array parameter and asks the `AcceptFunction` whether the object must be inspected or not. `AcceptFunction` is optional, so the value can be nil. In this case, all the objects are inspected. To allow syntax such as `Font.Name`, there is a small parser that splits the strings and walks through each *piece* to check whether there is a property with that name. If the last piece (or the only one) is found, then check whether that property is writable and if it is writable, set the property to the passed value. In this way, you can write code that walks through a complex object graph with a simple syntax:

```
Duck.Apply(TArray<TObject>.Create(
DataSource1, DataSource2, Button2), 'DataSet.Active', true);
```

There's more...

Duck typing is a very broad topic and allows you to do wonderful things with a few lines of code. In this recipe's code, there is a bonus project called `DuckTypingUsingRTTIExtended.dproj` that contains an advanced version of the base recipe. It uses a fluent interface and allows you to select the components that you want to change, and defines what type of change to do on those components; something similar to the following code snippets:

Set all the `Caption` property of the components on the form to `On All Captions`:

```
Duck(Self).All.SetProperty('Caption').ToValue('On All Captions');
```

Set all the `Text` properties to `'Hello There'` for the components with the name starting with `'Edit'` using an anonymous method as filter to select the components:

```
Duck(Self).Where(
function(C: TComponent): boolean
   begin
     Result := String(C.Name).StartsWith('Edit');
end)
.SetProperty('Text')
.ToValue('Hello There');
```

Set the `Color` property to `clRed` for all the `TEdit` components on the form. Use an anonymous method to define what to do on the components:

```
Duck(Self).Where(TEdit).Map(
procedure(C: TComponent)
   begin
     TEdit(C).Color := clRed;
end);
```

In the bonus project, there are more examples. Feel free to experiment and expand it.

Creating helpers for your classes

As you know (and if you don't know, you can read the documentation about it), a class helper is a type that can be associated to a class. When a class helper is associated with another class, all the methods and properties defined in the helper are also available in the other class and in its descendants. Helpers are a way to extend a class without using inheritance. However, it is not the same thing as inheritance. In other words, if the `TFooHelper` helper is in the same scope as `TFoo`, the compiler's resolution scope then becomes the original type (`TFoo`), plus the helper (`TFooHelper`). So, if the `TFoo` helper defines the `DoSomething` method and the helper of `TFoo` defines `DoAnotherThing`, when `TFoo` is used in the same scope as the `TFooHelper`, then the `TFoo` instances, and all its descendants, have also the `DoAnotherThing` method.

Getting ready

In this recipe, you'll see how to use class helpers to add iterators (or, a sort of) to the `TDataSet` class, so that any other `TDataSet` descendants—even from another vendor—automatically can support this kind of iterator. Moreover, you'll also add a `SaveToCSV` method so that any `TDataSet` can be saved in CSV with only one line of code.

www.allitebooks.com

How to do it...

For the DataSetClassHelpers.dproj project, let's start to talk about the simpler helper: the SaveToCSV method.

The current compiler implementation of class helpers allows only one helper active at a time. So if you need to add two or more helpers at the same time, you have to merge all the methods and properties in a single helper class. Your TDataSet helper is contained in the DataSetHelpersU.pas unit and is defined as follows:

```
TDataSetHelper = class helper for TDataSet
public
  procedure SaveToCSVFile(AFileName: String);
  function GetEnumerator: TDataSetEnumerator;
end;
```

To use this helper with your TDataSet instances, you have to add the DataSetHelpersU unit in the uses clause of the unit where you want to use the helper. The helper adds the following features to all the TDataSet descendants:

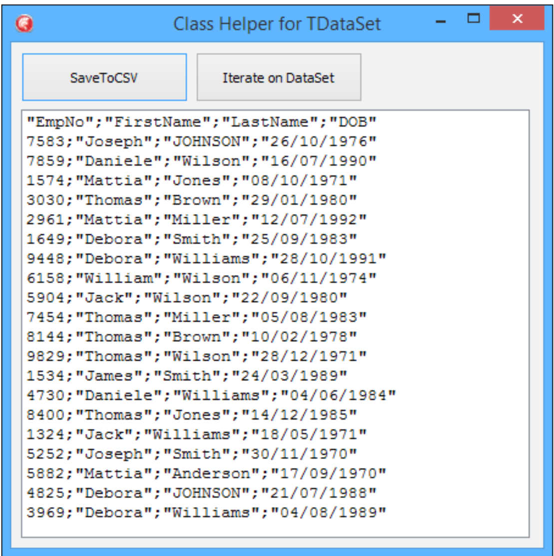
Method name	Description
SaveToCSV	This allows any dataset to be saved as a CSV file. The first row contains all the fieldnames. All the string values are correctly quoted while the numeric values aren't. The resultant CSV file is compatible with MS Excel and can be opened directly with it.
GetEnumerator	This enables the dataset to be used as enumerable type in the for..in loops. This removes the necessity to cycle the dataset using the usual while loop (so you cannot forget the DataSet.Next call at the end of the loop). The dataset is correctly cycled from the current position to the end, and for each record the for loop is executed. The enumerator item type is a wrapper type called TDSIterator able to access the individual values of the current record using a simplified interface.

To have an idea about what the helpers can do, check the following code:

```
//all the interface section before

implementation

uses
  DataSetHelpersU; //add the TDataSet helper to the compiler scope

procedure TClassHelpersForm.btnSaveToCSVClick(Sender: TObject);
begin
//use the SaveToCSVFile helper method
  FDMemTable1.SaveToCSVFile('mydata.csv');
  ListBox1.Items.LoadFromFile('mydata.csv');
end;

procedure TClassHelpersForm.btnIterateClick(Sender: TObject);
var
  it: TDSIterator; //this is the enumerator item type
begin
  //setup the ListBox with some nice headers
  ListBox1.Clear;
  ListBox1.Items.Add(
    Format('%-10s %-10s %8s',
    ['FirstName', 'LastName', 'EmpNo']));
  ListBox1.Items.Add(StringOfChar('-', 30));

  //iterate the dataset in a for..in loop using the helper
  for it in FDMemTable1 do
  begin
    ListBox1.Items.Add(
      Format('%-10s %-10s %8d',
      [
        it.Value['FirstName'].AsString,  //using the default
        it.S['LastName'],  //using the S[fieldname] for strings
        it.I['EmpNo']      //using the I[fieldname] for integers
      ]));
  end;
end;
```

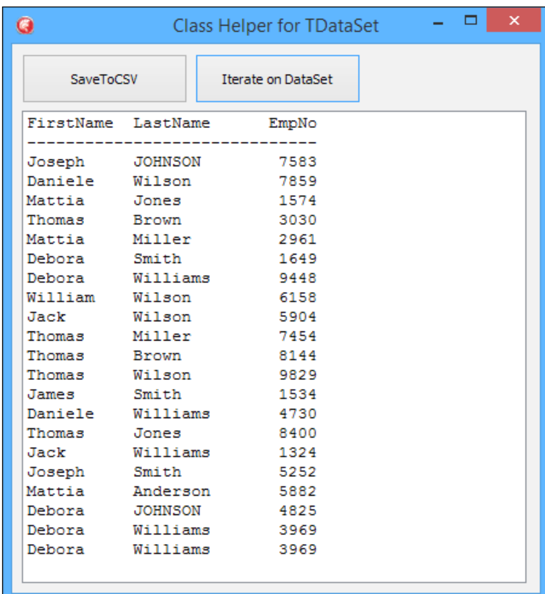

www.allitebooks.com

Useful, isn't it? The following screenshot shows the the status of the demo application after the **SaveToCSV** button was clicked. The demo application is seen as running.



The form after the SaveToCSV button is clicked; the generated CSV is reloaded in the listbox to show its contents

The following screenshot shows the output of the dataset iteration using the helper:



The form after the Iterate on DataSet button is clicked; the iteration is used to show dataset data in the listbox.

Let's see the implementation details.

The `SaveToCSV` method has been implemented as shown here:

```
procedure TDataSetHelper.SaveToCSVFile(AFileName: String);
var
  Fields: TArray<string>;
  CSVWriter: TStreamWriter;
  I: Integer;
  CurrPos: TArray<Byte>;
begin
  //save the current dataset position
  CurrPos := Self.Bookmark;

  Self.DisableControls;
  try
  //create a TStreamWriter to write the CSV file
  CSVWriter := TStreamWriter.Create(AFileName);
  try
    SetLength(Fields, Self.Fields.Count);
    for I := 0 to Self.Fields.Count - 1 do
    begin
      Fields[I] := Self.Fields[I].FieldName.QuotedString('"');
    end;

    //Write the headers line joining the fieldnames with a ";"
    CSVWriter.WriteLine(String.Join(';', Fields));

    //Cycle the dataset
    while not Self.Eof do
    begin
      for I := 0 to Self.Fields.Count - 1 do
      begin
        //DoubleQuote the string values
        case Self.Fields[I].DataType of
          ftInteger, ftWord, ftSmallint, ftShortInt,
       ftLargeint, ftBoolean, ftFloat, ftSingle:
            begin
              CSVWriter.Write(Self.Fields[I].AsString);
            end;
        else
          CSVWriter.Write(
            Self.Fields[I].AsString.QuotedString('"'));
        end;
```


www.allitebooks.com

```
        //if at the last columns, newline, otherwise ";"
        if I < Self.FieldCount - 1 then
             CSVWriter.Write(';')
           else
             CSVWriter.WriteLine;
         end;
         Self.Next; //next record
       end;
     finally
       CSVWriter.Free;
     end;

     finally
       Self.EnableControl;
     end;

     //return to the position where the dataset was before
     if Self.BookmarkValid(CurrPos) then
       Self.Bookmark := CurrPos;
   end;
```

The other helper is a bit more complex, but all the concepts have been already introduced in the *Writing enumerable types* recipe, so this should not be too complex to understand.

The method in the `class` helper simply returns `TDataSetEnumerator` by passing the current dataset to the constructor:

```
function TDataSetHelper.GetEnumerator: TDataSetEnumerator;
begin
  Result := TDataSetEnumerator.Create(Self);
end;
```

Now, some magic happens in the `TDataSetEnumerator`! Methods to access the current record are encapsulated in a `TDSIterator` instance. This class allows you to access the field values using a limited and simpler interface (compared to the `TDataSet` one).

Here's the declaration of the enumerator and the iterator:

```
TDataSetEnumerator = class(TEnumerator<TDSIterator>)
  private
    FDataSet: TDataSet;  //the current dataset
    FDSIterator: TDSIterator; //the current "position"
    FFirstTime: Boolean;
  public
    constructor Create(ADataSet: TDataSet);
    destructor Destroy; override;
```

```
    protected
      //methods to override to support the for..in loop
    function DoGetCurrent: TDSIterator; override;
      function DoMoveNext: Boolean; override;
    end;

  //This is the actual iterator
  TDSIterator = class
    private
      FDataSet: TDataSet;
      function GetValue(const FieldName: String): TField;
      procedure SetDataSet(const Value: TDataSet);
      function GetValueAsString(const FieldName: String): String;
      function GetValueAsInteger(const FieldName: String): Integer;
    public
      constructor Create(ADataSet: TDataSet);
      //properties to access the current record
      //values using the fieldname
      property Value[const FieldName: String]: TField read GetValue;
      property S[const FieldName: String]: String
                read GetValueAsString;
      property I[const FieldName: String]: Integer
                read GetValueAsInteger;
    end;
```

The `TDataSetEnumerator` handles the mechanism needed by the enumerable type. However, instead of implementing all the needed methods directly (as you saw in the *Writing enumerable types* recipe), you've inherited from the `TEnumerator<T>`, so the code to implement is shorter and simpler. Here's the implementation:

```
  { TDataSetEnumerator }

  constructor TDataSetEnumerator.Create(ADataSet: TDataSet);
  begin
    inherited Create;
    FFirstTime := True;
    FDataSet := ADataSet;
    FDSIterator := TDSIterator.Create(ADataSet);
  end;

  destructor TDataSetEnumerator.Destroy;
  begin
    FDSIterator.Free;
    inherited;
  end;
```


www.allitebooks.com

```
function TDataSetEnumerator.DoGetCurrent: TDSIterator;
begin
  Result := FDSIterator;
end;

function TDataSetEnumerator.DoMoveNext: Boolean;
begin
  Result := not FDataSet.Eof;
  if Result then
  begin
    if not FFirstTime then
      FDataSet.Next;
    FFirstTime := False;
  end;
end;
```

It is clear that the current record is encapsulated by a `TDSIterator` instance that uses the current dataset. This class is in charge to handle the real data access to the underlying dataset fields. Here's the implementation:

```
{ TDSIterator }

constructor TDSIterator.Create(ADataSet: TDataSet);
begin
  inherited Create;
  FDataSet := ADataSet;
end;

function TDSIterator.GetValue(const FieldName: String): TField;
begin
  Result := FDataSet.FieldByName(FieldName);
end;

function TDSIterator.GetValueAsInteger(
const FieldName: String): Integer;
begin
  Result := GetValue(FieldName).AsInteger;
end;

function TDSIterator.GetValueAsString(
const FieldName: String): String;
begin
  Result := GetValue(FieldName).AsString;
end;
```

Let's summarize the relation between the three classes involved. The `class` helper adds a method `GetEnumerator` to the `TDataSet` instance, which returns the `TDataSetEnumerator`. The `TDataSetEnumerator` uses the underlying dataset to handle the enumerable mechanism. The *current* element returned by the `DataSetEnumerator` is a `TDSIterator` that encapsulates the dataset current position allowing the user code to iterate the dataset using the `for..in` loop.

There's more...

What we discussed for class helper is valid for record helpers as well. If you find the content of this chapter too difficult, you can refresh your understanding about the helpers by (re)reading (and trying it by yourself) the *Class and record helpers* section in the Embarcadero docwiki website (`http://docwiki.embarcadero.com/RADStudio/XE6/en/Class_and_Record_Helpers_(Delphi)`).

Usually, when I talk about class and record helpers during my live training, just before showing the samples, the attendees ask, "I understand the concepts, but in which cases should I use them?"

See also

- Now, you have seen some nice use cases. However, if you need some others too, read this interesting thread on stack overflow at `http://stackoverflow.com/questions/253399/what-are-good-uses-for-class-helpers`.

Checking strings with regular expressions

A **regular expression** (**RegEx**) is a sequence of characters that forms a search pattern where some characters have a special meaning. It's mainly used to match patterns on strings. A simple case is something like this: check whether the string A matches the criteria defined in string B. Regular expressions follow a specific language to define the criteria. Regular expressions are not present only in Delphi. Many languages have a regular expression library in their standard built-in library. So, if you don't know what a regular expression is, you can read the general documentation at `http://en.wikipedia.org/wiki/Regular_expression` and then check the Delphi-specific built-in implementation at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Regular_Expressions`.

With regular expressions, you'll need an external tool to test the most complex ones (just like you want to test a complex query using a database tool instead of change the SQL in your code over and over again). There are a lot of sites offering such types of tool. One of the most complete websites offering such tools is `http://regex101.com`.

www.allitebooks.com

Getting started

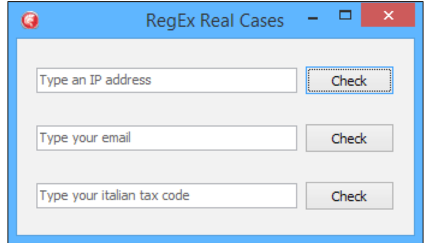
This recipe is a small complete project with specific objectives. It contains a list of checks that could be daunting to code from scratch but are trivial using regular expressions. Just one thing to remember: you always require a `RegEx` string and an input string for the checks, and the RegEx library gives back the result of the match. In this case, the result is true or false.

Here are some samples of very simple regular expressions with some input strings as a test. In the last column, you can see the result of the match. (RegEx can be used to perform smart string replace as well in order to find another string and so on, but the concept is the same as the check. You have only to call the right method, as `IsMatch`, `Split`, `Matches`, and so on, to give the right meaning to the RegEx.)

RegEx	RegEx description	Input string	Result
`rocks`	Contains rocks	delphi rocks	True
		rocks	True
		rocks of the mountain	False
`^rocks`	Starts with rocks	delphi rocks	False
		rocks of the mountain	True
`^[ABC]3`	Starts with A or B or C and then there is a 3. Anything after the 3 matches	A3	True
		B3	True
		C33	True
		F3	False
		A2	False
`^[ABC][01]$`	Starts with A or B or C and then there is 0 or 1. Then the input ends. No more characters allowed.	A0	True
		A1	True
		A2	False
		B1	True
		AA0	False
		C3	False

How to do it...

The test application is shown in the following screenshot:



The RegEx recipe main form with some checks on it

Each button checks the value written to the edit at its left. The checks are not 100 percent foolproof for the sake of simplicity, and they don't test the real validity of the data inserted. They only check the format validity (for example, if the e-mail address is formally valid, the check returns true even if the address doesn't really exist).

Open the recipe project called `RegEx.dproj` in the IDE and show the code of the form.

In Delphi, the needed classes and records to work with regular expressions are contained in the `System.RegularExpressions.pas` unit and follow the standard of the regular expression as handled by the Perl language (one of the first languages that started to use RegEx). The unit is included in the implementation section of the form. I suggest putting all your validation code in a separated unit in some testable validator types. However, in this recipe, the validation code is in the form under the event handler (please, do not do this in your production software!).

Let's start from the IP check. Under the `btnCheckIP` click, you can see the following code:

```
procedure TRegExForm.btnCheckIPClick(Sender: TObject);
begin
  if TRegEx.IsMatch(EditIP.Text,
     '^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$') then
    ShowMessage('IPv4 address is valid')
  else
    ShowMessage('IPv4 address is not valid');
end;
```

The code is really simple, only the RegEx needs some more explanation. The regular expression checks a string that starts with 1, 2, or 3 numbers from 0 to 9 (`[0-9]{1,3}`), then expect a point. Consider that point character in the regex syntax means *any character*, so if you simply want to check a point, you have to remove this meaning escaping the character. This is because there is a \ before the point.

www.allitebooks.com

RegEx continues with the same pattern repeated four times (for the four octets contained in the IPv4 address). The last pattern doesn't expect a point.

Using the static `TRegEx.IsMatch` method, you can easily check whether a string matches with a RegEx.

The second check is about the e-mail address. The code used is shown as follows:

```
procedure TRegExForm.btnCheckEmailClick(Sender: TObject);
begin
  // Email RegEx from
  // http://www.regular-expressions.info/email.html

  if TRegEx.IsMatch(
         EditEMail.Text,
         '^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$',
         [roIgnoreCase]) then
    ShowMessage('EMail address is valid')
  else
    ShowMessage('EMail address is not valid');
end;
```

In this case, the RegEx is a little bit more complicated. The string must start with at least a letter from A to Z, with a number from 0 to 9, or with another of the admitted char (., _, %, +, -). The sign `+` after the square brackets stands for at least one of.... Then there should be a `@` sign. After the `@` sign, the RegEx checks for letters, numbers, dots, and the minus sign (the domain part of the address) and, as last checks, it looks for two, three, or four letters (`.com`, `.it`, `.net`, and so on). The RegEx syntax is case-sensitive, but an e-mail address validity check must be case-insensitive, so I've put the `roIgnoreCase` modifier on the `IsMatch` to make the RegEx case-insensitive ([A-Z] is considered as [A-Z/a-z]).

As you see, if you can read the RegEx syntax, you can easily understand what the RegEx checks. Obviously, there are really complex RegExes, so before you use them, be sure to be confident with what you are using.

The last button checks the Italian tax code. I also put this example because the criteria are not so complex and it is good to understand the RegEx flexibility.

In Italy, there is a tax code called **Codice Fiscale** that is assigned to all citizens when they reach a certain age. The criteria are the following:

- 3 letters
- 3 letters
- 2 numbers
- 1 letter

- 2 numbers
- 1 letter
- 3 numbers
- 1 letter

So, for instance, this is a formally valid Italian tax code: `RSSMRA79S04H501V`.

As you see, it is not complex; however, checking it using plain code Delphi can be boring and error prone. Let's build together the RegEx to check it.

Start with 6 letters:

```
^[A-Z]{6}
```

Then, two numbers:

```
^[A-Z]{6}[0-9]{2}
```

Then, one letter and two numbers:

```
^[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}
```

Then, one letter, three numbers, and another one letter. Then, the code must terminate:

```
^[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-Z]$
```

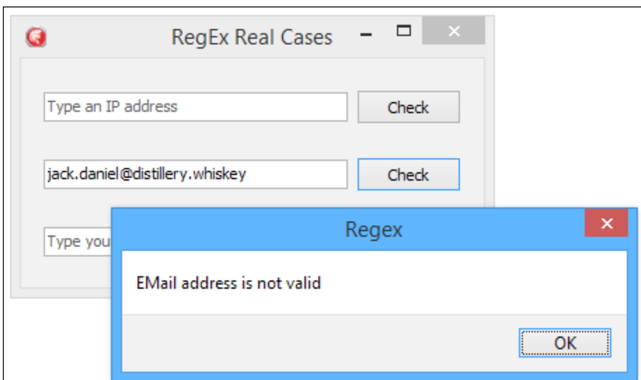
Now, the check is really simple:

```
procedure TRegExForm.btnCheckItalianTaxCodeClick(
                                         Sender: TObject);
begin
  if TRegEx.IsMatch(EditTaxCodeIT.Text,
       '^[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-Z]$',
       [roIgnoreCase]) then
    ShowMessage('This italian tax code is valid')
  else
    ShowMessage('This italian tax code is not valid');
end;
```

After some exercises, you can master the RegEx syntax and you will find it really useful to check and manipulate strings and texts.

www.allitebooks.com

The following screenshot shows the sample application while it is checking a wrong e-mail address:



The RegEx sample application while is checking a not valid e-mail address

There's more...

RegEx can be used to do a lot of string-related tasks. You can match strings, search for strings into another string, split a string using a RegEx as separator, and so on.

Remember to check the Delphi documentation about the built-in RegEx engine syntax at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Regular_Expressions`.

Some nice RegEx samples (not Delphi-related) can be found at `http://www.regular-expressions.info/examples.html`.

As a bonus project, there is a RegEx tester called `RegExTester.dproj` in the attached code that helps you to exploit all the functionalities. Play with it and become a RegEx Ninja!

www.allitebooks.com

3
Going Cross Platform with FireMonkey

In this chapter, we will cover the following recipes:

- Giving a new appearance to the standard FireMonkey controls using styles
- Creating a styled TListBox
- Impressing your clients with animations
- Using master/details with LiveBindings
- Showing complex vector shapes using paths
- Using FireMonkey in a VCL application

Introduction

The FireMonkey FMX framework is the app development and runtime platform behind Delphi, C++Builder, and Appmethod. FireMonkey is the first native GPU-powered application platform. The IT world is becoming more multiplatform with each passing year. FireMonkey is a key technology for Embarcadero because it is designed to build multidevice, true native apps for Windows, Mac, Android, and iOS.

This chapter explains some of the great features of FireMonkey. What is exposed in these recipes will be useful on every platform supported by the framework. Some of the OS-related features may not be available everywhere, but the great part of the concepts are usable on MS Windows, Mac OS X, Android, and iOS. There are ready-to-use recipes that will be useful every day.

Giving a new appearance to the standard FireMonkey controls using styles

Since Version XE2, RAD Studio includes FireMonkey. FireMonkey is an amazing library. It is a really ambitious target for Embarcadero, but it's important for its long-term strategy. VCL is and will remain a Windows-only library, while FireMonkey has been designed to be completely OS and device independent. You can develop one application and compile it anywhere (if *anywhere* is contained in Windows, OS X, Android, and iOS; let's say that is a good part of *anywhere*).

Getting ready

One of the main features of FireMonkey is customization through styles. A styled component doesn't know how it will be rendered on the screen, but the style. Changing the style, you can change the aspect of the component without changing its code. The relation between the component code and style is similar to the relation between HTML and CSS, one is the content and another is the display. In terms of FireMonkey, the component code contains the actual functionalities that the component has, but the aspect is completely handled by the associated style. All the `TStyledControl` child classes support styles.

Let's say you have to create an application to find a holiday house for a travel agency. Your customer wants a nice-looking application to search for the dream house for their customers. Your graphic design department (if present) decided to create a semitransparent look-and-feel, as shown in the following screenshot, and you've to create such an interface. How to do that?

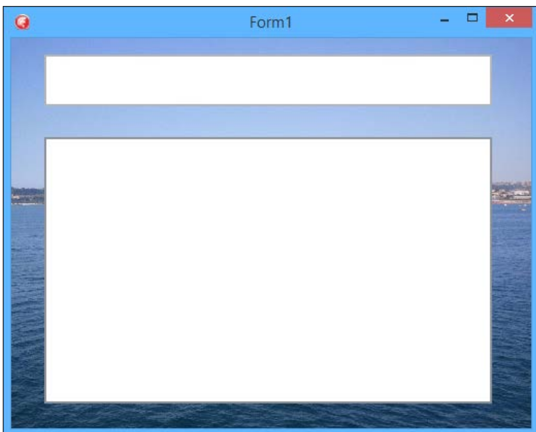
This is the UI we want

www.allitebooks.com

How to do it...

In this case, you require some step-by-step instructions, so here they are:

1. Create a new FireMonkey desktop application (navigate to **File** | **New** | **FireMonkey Desktop Application**).
2. Drop a **TImage** component on the form. Set its **Align** property to **alClient**, and use the **MultiResBitmap** property and its property editor to load a nice-looking picture.
3. Set the **WrapMode** property to **iwFit** and resize the form to let the image cover the entire form.
4. Now, drop a **TEdit** component and a **TListBox** component over the **TImage** component. Name the **TEdit** component as `EditSearch` and the **TListBox** component as `ListBoxHouses`.
5. Set the **Scale** property of the **TEdit** and **TListBox** components to the following values:
    - **Scale.X**: `2`
    - **Scale.Y**: `2`
6. Your form should now look like this:



The form with the standard components

The actions to be performed by the users are very simple. They should write some search criteria in the **Edit** field and click on **Return**. Then, the listbox shows all the houses available for that criteria (with a "contains" search). In a real app, you require a database or a web service to query, but this is a sample so you'll use fake search criteria on fake data.

7. Add the `RandomUtilsU.pas` file from the `Commons` folder of the project and add it to the uses clause of the main form.
8. Create an `OnKeyUp` event handler for the **TEdit** component and write the following code inside it:

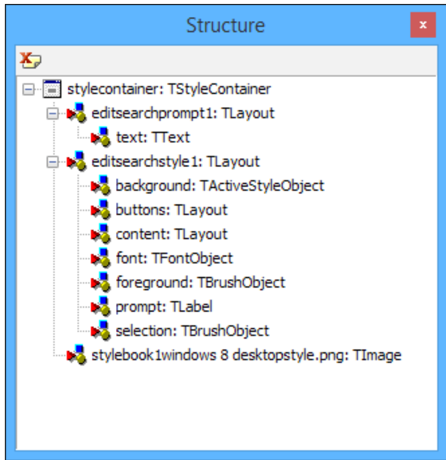
```
procedure TForm1.EditSearchKeyUp(Sender: TObject;
     var Key: Word; var KeyChar: Char; Shift: TShiftState);
var
  I: Integer;
  House: string;
  SearchText: string;
begin
  if Key <> vkReturn then
    Exit;

  // this is a fake search...
  ListBoxHouses.Clear;
  SearchText := EditSearch.Text.ToUpper;

  //now, gets 50 random houses and match the criteria
  for I := 1 to 50 do
  begin
    House := GetRndHouse;
    if House.ToUpper.Contains(SearchText) then
      ListBoxHouses.Items.Add(House);
  end;
  if ListBoxHouses.Count > 0 then
    ListBoxHouses.ItemIndex := 0
  else
    ListBoxHouses.Items.Add('<Sorry, no houses found>');
  ListBoxHouses.SetFocus;
end;
```

9. Run the application and try it to familiarize yourself with the behavior.
10. Now, you have a working application, but you still need to make it transparent. Let's start with the **FireMonkey Style Designer** (**FSD**).

Just to be clear, at the time of writing, the FireMonkey Style Designer is far to be perfect. It works, but it is not a pleasure to work with it. However, it does its job.

www.allitebooks.com

11. Right-click on the **TEdit** component. From the contextual menu, choose **Edit Custom Style** (general information about styles and the style editor can be found at `http://docwiki.embarcadero.com/RADStudio/XE6/en/FireMonkey_Style_Designer` and `http://docwiki.embarcadero.com/RADStudio/XE6/en/Editing_a_FireMonkey_Style`).
12. Delphi opens a new tab that contains the FSD. However, to work with it, you need the **Structure** pane to be visible as well (navigate to **View** | **Structure** or *Shift* + *Alt* + *F11*).
13. In the **Structure** pane, there are all the styles used by the **TEdit** control. You should see a **Structure** pane similar to the following screenshot:

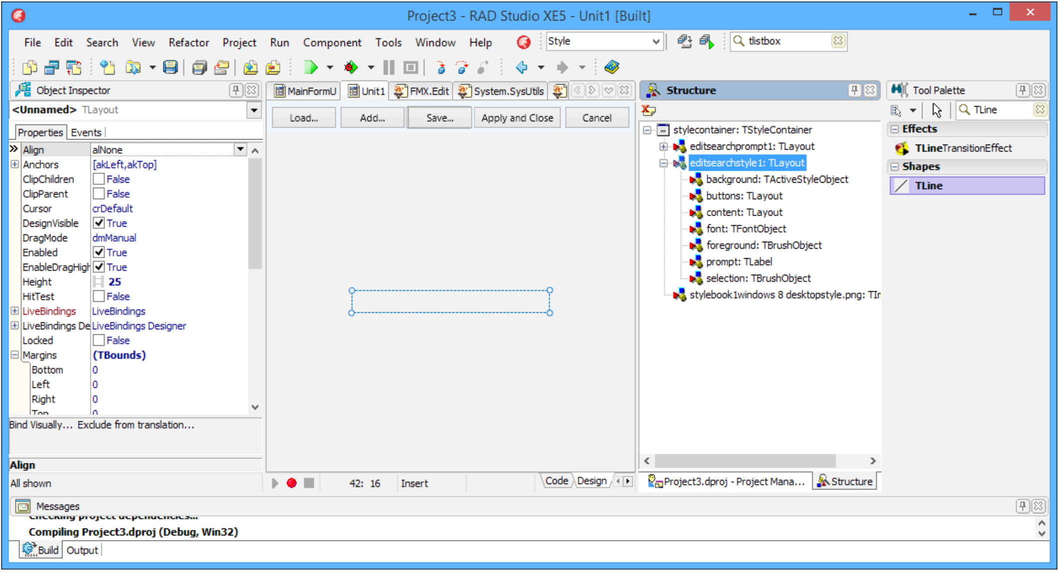


The Structure pane showing the default style for the TEdit control

14. In the **Structure** pane, open the **editsearchstyle1** node, select the **background** subnode, and go to the **Object Inspector**.
15. In the **Object Inspector** window, remove the content of the **SourceLookup** property.

The **background** part of the style is **TActiveStyleObject**. A **TActiveStyleObject** style is a style that is able to show a part of an image as default and another part of the same image when the component that uses it is active, checked, focused, mouse hovered, pressed, or selected. The image to use is in the **SourceLookup** property. Our **TEdit** component must be completely transparent in every state, so we removed the value of the **SourceLookup** property.

16. Now the **TEdit** component is completely invisible. Click on **Apply and Close** and run the application. As you can confirm, the edit works but it is completely transparent. Close the application.
17. When you opened the FSD for the first time, a **TStyleBook** component has been automatically dropped on the form and contains all your custom styles. Double-click on it and the style designer opens again.
18. The edit, as you saw, is transparent, but it is not usable at all. You need to see at least where to click and write. Let's add a small bottom line to the edit style, just like a small underline.
19. To perform the next step, you require the **Tool Palette** window and the **Structure** pane visible. Here is my preferred setup for this situation:



The Structure pane and the Tool Palette window are visible at the same time using the docking mechanism; you can also use the floating windows if you wish

20. Now, search for a **TLine** component in the **Tool Palette** window. Drag-and-drop the **TLine** component onto the **editsearchstyle1** node in the **Structure** pane. Yes, you have to drop a component from the **Tool Palette** window directly onto the **Structure** pane.
21. Now, select the **TLine** component in the **Structure** Pane (do not use the FSD to select the components, you have to use the **Structure** pane nodes). In the **Object Inspector**, set the following properties:
    - **Align**: **alContents**
    - **HitTest**: **False**
    - **LineType**: **ltTop**

www.allitebooks.com

- **RotationAngle**: `180`
- **Opacity**: `0.6`

22. Click on **Apply and Close**.
23. Run the application. Now, the text is underlined by a small black line that makes it easy to identify that the application is transparent. Stop the application.
24. Now, you've to work on the listbox; it is still 100 percent opaque.
25. Right-click on the **ListBoxHouses** option and click on **Edit Custom Style**.
26. In the **Structure** pane, there are some new styles related to the `TListBox` class. Select the **listboxhousesstyle1** option, open it, and select its child style, **background**.
27. In the **Object Inspector**, change the **Opacity** property of the **background** style to `0.6`. Click on **Apply and Close**.
28. That's it! Run the application, write `Calif` in the **Edit** field and press **Return**. You should see a nice-looking application with a semitransparent user interface showing your dream houses in California (just like it was shown in the screenshot in the *Getting ready* section of this recipe). Are you amazed by the power of FireMonkey styles?

How it works...

The trick used in this recipe is simple. If you require a transparent UI, just identify which part of the style of each component is responsible to draw the background of the component. Then, put the **Opacity** setting to a level less than `1` (`0.6` or `0.7` could be enough for most cases). Why not simply change the **Opacity** property of the component? Because if you change the **Opacity** property of the component, the whole component will be drawn with that opacity. However, you need only the background to be transparent; the inner text must be completely opaque. This is the reason why you changed the style and not the component property.

In the case of the **TEdit** component, you completely removed the painting when you removed the **SourceLookup** property from **TActiveStyleObject** that draws the background.

As a thumb rule, if you have to change the appearance of a control, check its properties. If the required customization is not possible using only the properties, then change the style.

There's more...

If you are new to FireMonkey styles, probably most concepts in this recipe must have been difficult to grasp. If so, check the official documentation on the Embarcadero DocWiki at the following URL:

`http://docwiki.embarcadero.com/RADStudio/XE6/en/Customizing_FireMonkey_Applications_with_Styles`

Creating a styled TListBox

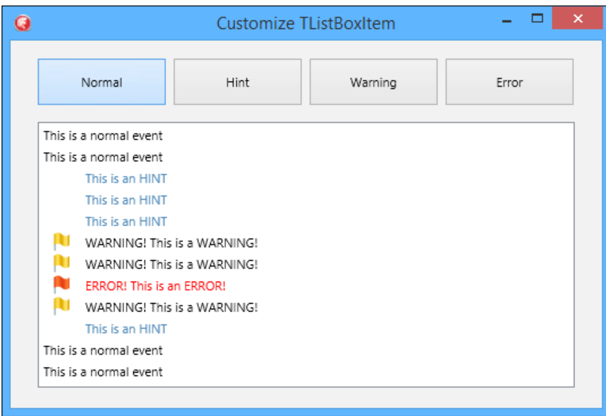
As you saw in the previous recipe, it is possible to style styled controls and completely change their appearance. While in the VCL, the **TListBox** control is a mere wrapper over the correspondent control in the MS Windows API; in FireMonkey, the **TListBox** component is a completely different beast. A **TListBox** component contains a list of `TListBoxItem`, and a **TListBox** item is a `TStyledControl` descendant. This means that every single item in a **TListBox** component can be styled! This feature opens a huge set of new possibilities regarding the use of the control.

Getting started

In this recipe, you'll see a set of styled `TListBoxItem` component that when added to **TListBox**, changes its appearance completely. Let's say you have a listbox containing a log of events that happened in a monitored remote system. Some events are simply informative, while other events can denote a malfunction. Different kinds of events are shown with different graphics in the listbox. Here are the events:

Type	Appearance
Normal	This is the default option for `TListBoxItem`.
Hint	This has blue colored text on a white background. The text is left aligned but indented 40 pixels.
Warning	This has black colored text on a white background. There is a small yellow flag on the left-hand side.
Error	This has red colored text over a white background. There is a small red flag on the left.

What you require is shown in the following screenshot:



The listbox with some types of events logged

www.allitebooks.com

To achieve this result using VCL, you usually rely on owner drawing or some third-party controls. However, with FireMonkey, all these customizations are a matter of style, so are simpler, faster to implement, reusable, and more flexible.

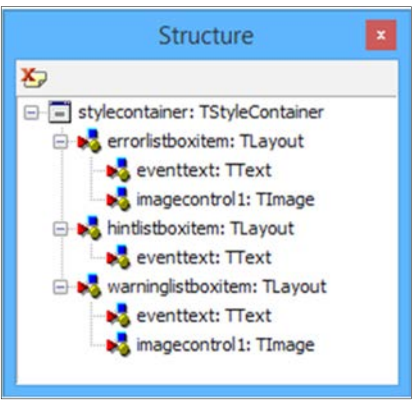
How to do it...

Let's start creating our stunning FireMonkey GUI:

1. Create a new FireMonkey desktop application.
2. Drop four **TButton** components, a **TListBox** component, and a **TStyleBook** component on the form.
3. Double-click on the **TStyleBook** component and open the style editor.
4. Show the **Structure** pane (navigate to **View** | **Structure** or press *Shift* + *Alt* + *F11*).
5. Drop three **TLayout** components to create three different styles.
6. Set the **StyleName** property of the three **TLayout** components as follows:
    - **errorlistboxitem**
    - **hintlistboxitem**
    - **warninglistboxitem**

    The **StyleName** property allows you to reference to the style from your form, so we've created three new styles usable from the main form.
7. Drop in every **TLayout** a **TText** so that every **TLayout** contains a **TText**. Set the **TText.Align** property to **alClient**.
8. Set the **StyleName** property for each **TText** to **eventtext**. Pay attention; every **TText** has the same value in the **StyleName** property. This allows you to use the **StylesData** property independently of the applied style.
9. Now let's work on each style. Select the **hintlistboxitem** style from the **Structure** pane.
10. Now, select the inner **eventtext** component (a **TText** component) and set its **Color** property to **Blue** and its **Margins.Left** property to **40**.
11. Select the **warninglistboxitem** style from the **Structure** pane.
12. Now, drop a **TImage** component into the style at the same level of **TText**.
13. Set **TImage.Align** to **alMostLeft**.
14. Load a small 32 x 32 icon showing a small yellow flag in its **MultiResBitmap** property (some free icons are provided with the code of the book).
15. Set **TImage.Width** to **40**.
16. Select the **errorlistboxitem** style from the **Structure** pane.
17. Set the **TText.Color** property to **claRed**.

18. Now, drop a **TImage** component into the style at the same level of **TText**.
19. Set **TImage.Align** to **alMostLeft**.
20. Load a small 32 x 32 icon showing a small red flag in its **MultiResBitmap** property.
21. Set **TImage.Width** to `40`.
22. Now, your **Structure** pane should look like this:



23. Click on **Apply and Close** on the style designer toolbar.
24. Now, the **TStyleBook** component contains all the custom styles. However, currently those styles are not used. Let's use them.
25. Go to the form class declaration and add the following private method:

```
procedure TForm1.AddEvent(EventType, EventText: String);
var
  LBItem: TListBoxItem;
begin
  LBItem := TListBoxItem.Create(ListBox1);
  LBItem.Parent := ListBox1;
  if EventType.Equals('normal') then
  begin
    LBItem.Text := EventText;
  end
  else
  begin
    LBItem.StyleLookup := EventType + 'listboxitem';
    LBItem.StylesData['eventtext'] := EventText;
  end;
  ListBox1.AddObject(LBItem);
end;
```


www.allitebooks.com

26. Set the button names and captions to the following values:
    - `btnNormal` (**Caption**: **Normal**)
    - `btnHint` (**Caption**: **Hint**)
    - `btnWarning` (**Caption**: **Warning**)
    - `btnError` (**Caption**: **Error**)

27. Create four event handlers, one for each **TButton** component, as shown in the following code:

```
procedure TForm1.btnNormalClick(Sender: TObject);
begin
  AddEvent('normal', 'This is a normal event');
end;

procedure TForm1.btnHintClick(Sender: TObject);
begin
  AddEvent('hint', 'This is an HINT');
end;

procedure TForm1.btnWarningClick(Sender: TObject);
begin
  AddEvent('warning', 'WARNING! This is a WARNING!');
end;

procedure TForm1.btnErrorClick(Sender: TObject);
begin
  AddEvent('error', 'ERROR! This is an ERROR!');
end;
```

28. Hit *F9* (or go to **Run** | **Run**) and try to click on the buttons, and you should see something like the **Structure** pane shown earlier.

How it works...

By clicking on each the button, a new `TListBoxItem` is created (in the **AddEvent** method). Depending on the event type, the correct style is selected from the **TStyleBook** component. There is no need to directly refer to **TStyleBook**, FireMonkey looks for all the **TStyleBook** components currently visible. The **StyleLookup** property sets the style used for `TListBoxItem`, while the **StylesData** indexed property contains the values for every style component with **StyleName**. By setting `StylesData['eventtext']`, you are actually setting the **Text** property of the inner **TText** component.

There's more...

FireMonkey styles are really powerful. The not-so-perfect style designer makes working with styles a little bit harder, but once you grasp the foundation using FireMonkey, styles are addictive! Some links to go deeper with styles are as follows:

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/FireMonkey_Style_Designer`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Customizing_FireMonkey_Applications_with_Styles`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Working_with_Native_and_Custom_FireMonkey_Styles`

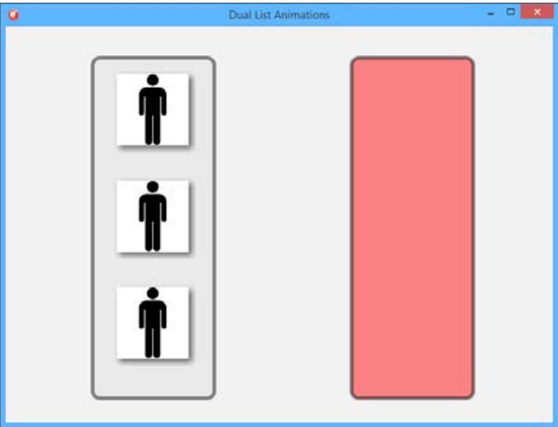
Impressing your clients with animations

Animations are a nice thing. A well-done animation, not too intrusive and with good visual information, can explain what is happening on the UI better than 1,000 words. In this recipe, you'll implement a dual list with the `include<>exclude` paradigm so that what is removed from one list is included in the other list and vice versa. You'll use FireMonkey animations.

FireMonkey animations are really simple to use. Some kinds of property types can be animated. Some of these types are color, bitmap, gradient, and floating point number. The most used animation engine is the **TFloatAnimation**. This is used to animate floating point values such as **Opacity**, **Position.X**, **Position.Y**, **Width**, **Height**, and many more.

How to do it...

What you want to create is shown in the following screenshot:

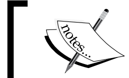


The dual list selection form

www.allitebooks.com

There are three images in the left-hand side gray list and zero images in the red list on the right-hand side. Clicking on an image; the clicked image will slide to the opposite list (gray to red or red to gray) using a nice animation. The steps to reproduce the images are as follows:

1. Create a new FireMonkey desktop application.
2. Drop two **TRectangle** components on the form. Align the first one on the left-hand side and the second one to the right-hand side, as shown in the screenshot in the *Getting ready* section of this recipe.
3. Set the properties of the left-hand side rectangle like this:
   - **Fill.Color**: **#FFE0E0E0**
   - **Fill.Kind**: **bkSolid**
   - **Stroke.Thickness**: `5`
   - **Opacity**: `0.5`
   - **XRadius**: `10`
   - **YRadius**: `10`
4. Set the properties of the right-hand side rectangle as follows:
   - **Fill.Color**: **Red**
   - **Fill.Kind**: **bkSolid**
   - **Stroke.Thickness**: `5`
   - **Opacity**: `0.5`
   - **XRadius**: `10`
   - **YRadius**: `10`
5. Now, drop three **TImage** components on the form and align them into the left-hand side **TRectangle**.
6. Load some kind of picture or icon into **TImages**. You can use the same images for each **TImage** (as I did) or different images for each **TImage**. It depends on what kind of information you want to transfer to your user.



In the included source code, you can find an image called `blackman.png` that is the one that I used.

7. Now, for each image, drop **TShadowEffect**. The effect must be owned by the **TImage** component so that in the **Structure** pane, the **TImage** component contains a subnode named **TShadowEffect**. Perform the same action for each **TImage**.
8. Now, set the **Distance** property to `5` for each **TShadowEffect**.

9. Our UI is created. Now you've to write some code. Use the same event handler for all the three **TImage** components. Let's create the event handler with a double-click on the **TImage1** component. Fill the event handler with this code:

```
procedure TDualListForm.Image1Click(Sender: TObject);
var
  Img: TImage;
const
  LEFT_LIMIT = 150; //must be inside the left rectangle
  RIGHT_LIMIT = 500; //must be inside the right rectangle
begin
  Img := (Sender as TImage);

  if Img.Tag = 0 then
  begin
    Img.Tag := 1;
    Img.AnimateFloat('Position.X',
         RIGHT_LIMIT, 0.8,
         TAnimationType.atOut,
         TInterpolationType.itElastic)
  end
  else
  begin
    Img.Tag := 0;
    Img.AnimateFloat('Position.X',
         LEFT_LIMIT, 0.8,
         TAnimationType.atOut,
         TInterpolationType.itElastic);
  end;

  Img.AnimateFloatDelay('Scale.X', 1.2, 0.2, 0.2);
  Img.AnimateFloatDelay('Scale.Y', 1.2, 0.2, 0.2);

  Img.AnimateFloatDelay('Scale.X', 1, 0.2, 1);
  Img.AnimateFloatDelay('Scale.Y', 1, 0.2, 1);
end;
```

10. In the preceding code, there are some values for `LIMIT_LEFT` and `LIMIT_RIGHT`. Adjust them with values that are good for your form and make them aligned with the rectangles. The objective is that when the image is clicked, it should start from the left-hand side rectangle and should move to the right-hand side rectangle. If the image is clicked a second time, it should return to the left-hand side (with the same animation). In a more complex scenario, it's better to calculate these values instead of putting them as fixed values.

11. As you can see, I've used the `TImage.Tag` property to keep track of the current position. It would be better to have an external data model to hold this kind of visual state instead of putting this information in the graphical components, but for this demo, it's okay.

www.allitebooks.com

12. Now, connect the same `TImage1.OnClick` event handler to `TImage2.OnClick` and `TImage3.OnClick` as well. In this way, you can centralize the behavior in a single event handler.
13. Run (navigate to **Run** | **Run** or press *F9*) and start clicking on the images.

How it works...

This recipe is very simple and is a good example of how animations can be used to gain not only visual *wow* effect (that probably may even disturb your user in some cases), but also some informative content.

The approach is simple: when the user clicks, communicate using animations that something happened and make it mime the real physical world. This is about the eBook reader applications on your smartphones. Is it strictly required to show a *Turning page animation* when you change page? No! However, it makes it clear to the user what is happening. Your animation should be used for the same goal.

There's more...

Some useful basic information about animations can be read on the Embarcadero DocWiki:

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/FireMonkey_Animation_Effects`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Using_FireMonkey_Animation_Effects`

Using master/details with LiveBindings

When you have a customer with his/her orders or an invoice with his/her items, you have a **master/details** (**M/D**) relationship. In this recipe, you'll learn how to use the new LiveBindings technology to show an M/D relationship.

Getting started

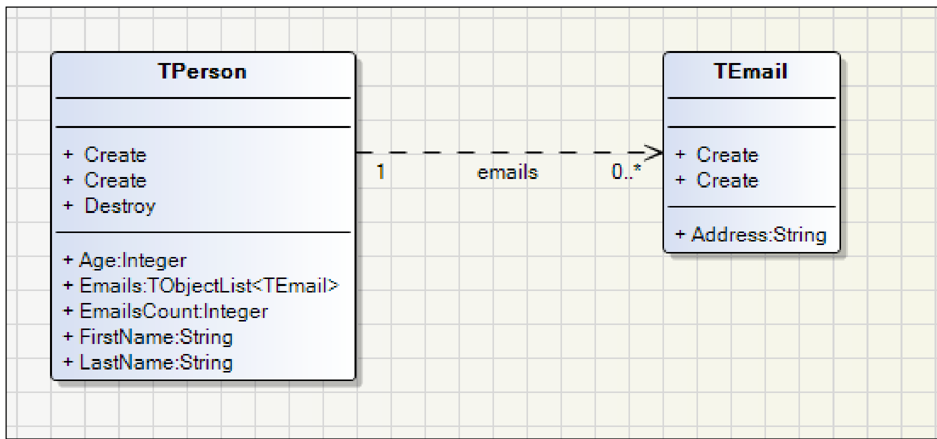
As explained in the Embarcadero wiki:

> *"LiveBindings is a data-binding feature supported by both the VCL and FireMonkey frameworks in RAD Studio. LiveBindings is an expression-based framework, which means it uses bindings expressions to bind objects to other objects or to dataset fields."*

LiveBindings is a very nice technology and can be used in VCL applications also, but its main targets are FireMonkey applications. Indeed, it is the only way to do automatic data binding in the FireMonkey framework. If you don't know what LiveBindings is or what its strengths are, I suggest you stop here and read the article in the Embarcadero wiki at `http://docwiki.embarcadero.com/RADStudio/XE6/en/LiveBindings_in_RAD_Studio`.

What we want to do in this recipe is create a simple but complete FireMonkey application that handles a sort-of M/D relationship. Usually, this kind of thing involves the use of some databases. In this case, however, we are abandoning the SQL-based approach (that uses two or more datasets) in favor of a purely object-oriented approach. In other words, you'll use a list of objects instead of a simple SQL query, and the relationships are child objects contained in the main object, not another query.

The simple UML class diagram generated by Delphi is shown here:

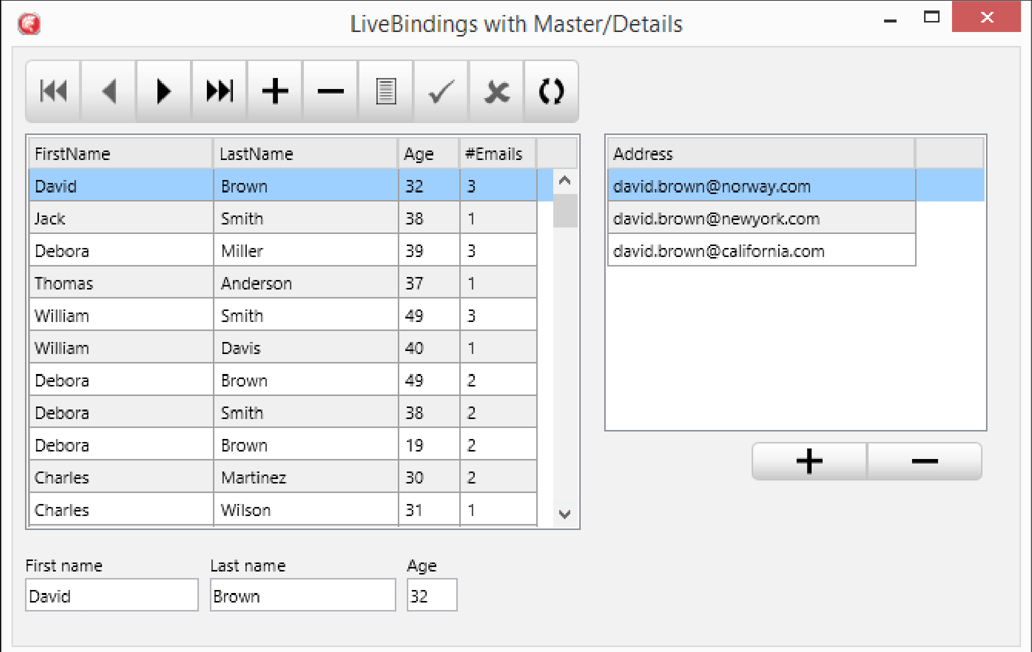


The UML class diagram for the recipe

The main list of objects contains the **TPerson** instances. Each **TPerson** instance, as shown in the preceding screenshot, contains a variable number of e-mails. So, the **TPerson** class has a property called **Emails** that is a list of **TEmail** instances. Instead of filtering all the e-mails and showing only the e-mails related to the selected person (as usually happens in the classic SQL programming), you'll show only the e-mails of that person, no filters are involved. The e-mails are already tied to the person. In this recipe, the difference between the `TDataset` approach and the object-oriented approach will be cleared.

www.allitebooks.com

The final application is shown in the following screenshot:



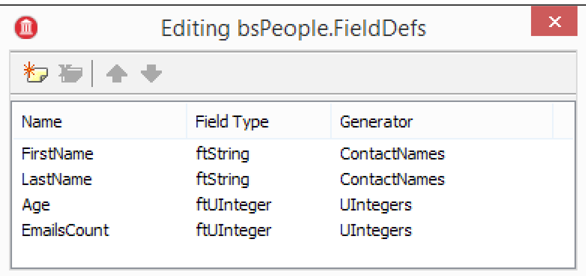
The final aspect of the M/D application that is able to manage people and the related e-mails

How to do it...

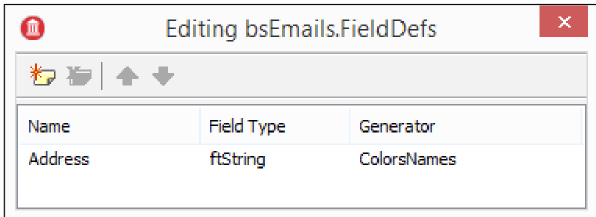
Let's start:

1. Create a new FireMonkey HD desktop application and name the main form as `MainForm`.
2. Drop two **TGrid** components on the form and name them `grdPeople` and `grdEmails`. Set both the `AlternatingRowBackground` properties to `True` for both the components.
3. Drop two **TProtypeBindSource** components in the form and name them `bsPeople` and `bsEmails`.

4. Double-clicking on **bsPeople** shows its field definitions. Using the **Add** (the first button from the left-hand side) button, add four fields as shown in the following screenshot:



5. Close the field definition of **bsPeople**.
6. Double-clicking on **bsEmails** shows its field definitions. Using the **Add** (the first button from the left-hand side) button, add the **Address** field as shown here:



7. Close the field definition of **bsEmails**.
8. Drop a **TBindNavigator** component on the form and connect its **DataSource** property to **bsPeople**.
9. Drop another **TBindNavigator** component on the form and connect its **DataSource** property to **bsEmails**. Then, set all the elements inside its `VisibleButtons` property to `False` and set only `nbInsert` and `nbDelete` to `true` (this will allow you to insert or remove any e-mail from a person).
10. Now, drop three **TEdit** components on the form and name them `EditFirstName`, `EditLastName`, and `EditAge`.
11. Our UI is almost ready. Add some labels and arrange the controls.
12. Now the interesting part begins.
13. Go to **View** | **LiveBindings Designer**.
14. The window shows the famous **LiveBindings Designer**. All the **LiveBindings Enabled** controls with their properties will be shown.

www.allitebooks.com

15. On the left-hand side toolbar, there are a set of buttons useful to change the disposition and zoom of the diagram. Use the buttons; they will save your sanity!
16. Identify the **bsPeople** element and drag-and-drop all its elements on `grdPeople`. You can also drag only the ***** column, but (in this version of Delphi) columns are not created at design time. So, if you want (as usually you will) to change the aspect of the grid columns, drag every field one by one.
17. Perform the same action (as done for **bsPeople**) with the **bsEmails** and `grdEmails`.
18. Now, you've to connect the editable field of the **bsPeople** component to the **TEdits** component.
19. Connect `bsPeople.FirstName` to `EditFirstName.Text`.
20. Connect `bsPeople.LastName` to `EditLastName.Text`.
21. Connect `bsPeople.Age` to `EditAge.Text`.
22. Do not connect the `EmailsCount` field. This field is a read-only field, mapped to a read-only property, and used to show the number of the e-mail addresses related to the current person. This technique can be quite useful when you have to show how many rows are contained in an invoice, how many orders are related to a customer, and so on.
23. If you run the application now, you will see some fake data is generated. You will also notice that there is no M/D relationship between people and e-mails. We are about to fix this in a moment. Close the application and go back to Delphi.
24. Add a new unit, name it `BusinessObjectsU.pas`, and add the following code into it:

```
unit BusinessObjectsU;

interface

uses System.Generics.Collections;

type
  TEmail = class
  private
    FAddress: String;
    procedure SetAddress(const Value: String);
  public
    constructor Create; overload;
    constructor Create(AEmail: String); overload;
    property Address: String
                read FAddress write SetAddress;
  end;
```

```
  TPerson = class
  private
    FLastName: String;
    FAge: Integer;
    FFirstName: String;
    FEmails: TObjectList<TEmail>;
    procedure SetLastName(const Value: String);
    procedure SetAge(const Value: Integer);
    procedure SetFirstName(const Value: String);
    function GetEmailsCount: Integer;
  public
    constructor Create; overload;
    constructor Create(const FirstName, LastName: string;
           Age: Integer); overload; virtual;
    destructor Destroy; override;
    property FirstName: String
                read FFirstName write SetFirstName;
    property LastName: String
                read FLastName write SetLastName;
    property Age: Integer read FAge write SetAge;
    property EmailsCount: Integer read GetEmailsCount;
    property Emails: TObjectList<TEmail> read FEmails;   end;

implementation

uses System.SysUtils;

constructor TPerson.Create(const FirstName, LastName:
                                  string; Age: Integer);
begin
  Create;
  FFirstName := FirstName;
  FLastName := LastName;
  FAge := Age;
end;

// Called by LiveBindings to insert a new Person
constructor TPerson.Create;
begin
  inherited Create;
  FFirstName := '<name>';
  //initialize the emails list
  FEmails := TObjectList<TEmail>.Create(true);
end;
```


www.allitebooks.com

```
destructor TPerson.Destroy;
begin
  FEmails.Free;
  inherited;
end;

function TPerson.GetEmailsCount: Integer;
begin
  Result := FEmails.Count;
end;

procedure TPerson.SetLastName(const Value: String);
begin
  FLastName := Value;
end;

procedure TPerson.SetAge(const Value: Integer);
begin
  FAge := Value;
end;

procedure TPerson.SetFirstName(const Value: String);
begin
  FFirstName := Value;
end;

constructor TEmail.Create(AEmail: String);
begin
  inherited Create;
  FAddress := AEmail;
end;

// Called by LiveBindings to insert a new Email
constructor TEmail.Create;
begin
  Create('<email>');
end;

procedure TEmail.SetAddress(const Value: String);
begin
  FAddress := Value;
end;

end.
```

25. Now, go to the `TMainForm` declaration and add the following code in the `private` section:

```
private
  FPeople: TObjectList<TPerson>;
  bsPeopleAdapter: TListBindSourceAdapter<TPerson>;
  bsEmailsAdapter: TListBindSourceAdapter<TEmail>;
  procedure PeopleAfterScroll(Adapter: TBindSourceAdapter);
  procedure LoadData;
```

26. Create the `PeopleAfterScroll` and `LoadData` methods in the `implementation` section (you can use *Ctrl* + *Shift* + *C* to generate the empty method body; check all the others keyboard shortcuts at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Default_Keyboard_Shortcuts`):

```
procedure TMainForm.LoadData;
var
  I: Integer;
  P: TPerson;
  X: Integer;
begin
  for I := 1 to 100 do
  begin
    // create a random generated person
    P := TPerson.Create(
      GetRndFirstName,
      GetRndLastName,
      10 + Random(50));

    // add some email addresses (1..3) to the person
    for X := 1 to 1 + Random(3) do
    begin
      P.Emails.Add(
        TEmail.Create(P.FirstName.ToLower + '.' +
                      P.LastName.ToLower +
        '@' + GetRndCountry.Replace(' ', '').ToLower +
        '.com'));
    end;
    FPeople.Add(P);
  end;
end;

procedure TMainForm.PeopleAfterScroll(
                  Adapter: TBindSourceAdapter);
begin
```


www.allitebooks.com

```
  bsEmailsAdapter.SetList(
   bsPeopleAdapter.
     List[bsPeopleAdapter.CurrentIndex].Emails, False);
  bsEmails.Active := True;
  bsEmails.First;
end;
```

27. On the main form, create the `FormCreate` and `FormDestroy` event handlers with this code:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Randomize;
  FPeople := TObjectList<TPerson>.Create(True);
  LoadData;
  bsPeopleAdapter.SetList(FPeople, False);
  bsPeople.Active := True;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FPeople.Free;
end;
```

28. Now, show the main form, select **bsPeople**, and create the event handler for the `OnCreateAdapter` event. This event is called when the `TPrototypeBindSource` method has to decide whether to use fake random-generated data or your real data. You have to handle this event and plug the code to provide your data. Write the following code in the event handler:

```
procedure TMainForm.bsPeopleCreateAdapter(Sender: TObject;
          var ABindSourceAdapter: TBindSourceAdapter);
begin
  bsPeopleAdapter := TListBindSourceAdapter<TPerson>.
         Create(self, nil, False);
  ABindSourceAdapter := bsPeopleAdapter;
  bsPeopleAdapter.AfterScroll := PeopleAfterScroll;
end;
```

29. On the main form, select **bsEmails** and create the event handler for the `OnCreateAdapter` event:

```
procedure TMainForm.bsEmailsCreateAdapter(Sender: TObject;
          var ABindSourceAdapter: TBindSourceAdapter);
begin
  bsEmailsAdapter := TListBindSourceAdapter<TEmail>
                             .Create(self, nil, False);
  ABindSourceAdapter := bsEmailsAdapter;
end;
```

30. If you run the application, you should see a working form showing an M/D relationship, or better a "has a" relationship, because a person *has* a list of e-mails. Stop it and add a small trick.
31. If you try to add a new e-mail, the new line is added in the **TGrid** component. I hate data entry directly into grids! In some cases, it is a great feature, but in many cases, it only shows a badly designed UI (this is not the case if you are developing a spreadsheet!). So, let's create `TBindSourceNavigator` to show a dialog to add a new e-mail.
32. Select the `TBindSourceNavigator` component named **bnEmails**, create an event handler for the `BeforeAction` event, and then write the following code:

```
procedure TMainForm.bnEmailsBeforeAction(Sender: TObject;
                Button: TNavigateButton);
var
  email: string;
begin
  if Button = TNavigateButton.nbInsert then
    if InputQuery('Email', 'New email address', email) then
    begin
      bsEmailsAdapter.List.Add(TEmail.Create(email));
      bsEmails.Refresh; // refresh the emails list
      bsPeople.Refresh; // refresh the email count
      Abort; //inhibit the normal behavior
    end;
end;
```

33. Now, run the application and try to add a new e-mail; you'll see a nice dialog comes up.
34. That's all folks!

How it works...

There are a few concepts involved in LiveBindings, but these concepts must be well understood to create a working application. Let's analyze this application.

At the beginning, the `TPrototypeBindSource` components are initialized with the `TListBindSource<T>` instances so that they show actual data instead of fake data. Then in the `FormCreate` event handler, you created the actual list of objects that will contain your people and load some data in it using the `LoadData` method. This method loads some random data but in a real application, it should read data from some query or from some web service. This is one of LiveBindings strengths; you can visualize your data wherever its origin is. You are no longer tied to `TDataSet`!

www.allitebooks.com

After loading the data, you set the **bsPeople** list of objects to your people and then activated it. This is okay for one single list of data, but how to handle the M/D relationship?

In the `bsPeople.OnCreateAdapter` event, you set an `AfterScroll` event handler for `bsPeopleAdapter` (the internal adapter used by `TPrototypeBindSource`). This event is called when the selected person changes. So, you can handle the data visualization on the e-mails grid from this event. The code in this event handler is self-explanatory:

```
procedure TMainForm.PeopleAfterScroll(
                     Adapter: TBindSourceAdapter);
begin
  //sets the email object list to the emails of the
  //selected person in the bsPeopleAdapter
  bsEmailsAdapter.SetList(
    bsPeopleAdapter.
      List[bsPeopleAdapter.CurrentIndex].Emails, False);
  //now the bsEmails is no more active, let's activate it
  bsEmails.Active := True;
  bsEmails.First;
end;
```

Usually, working with the internal adapter of `TPrototypeBindSource` is a bit messy because you have to write something like this:

```
//sets a new list of objects as data source
(bsPeople.InternalAdapter as TListBindSourceAdapter<TPerson>).
SetList(MyList);
```

Saving a reference when you are creating the actual adapter in the `OnCreateAdapter` method saves a lot of casting and makes code more readable. There are other solutions, but I really like this one.

There's more...

LiveBindings is a new technology. It has changed a lot since its introduction in Delphi XE2, at least in the high-level components. The good old Delphi programmer seems to not completely understand its power (probably because `TDataSet` along with VCL really does a good job for classic client/server applications), but there is still time to explore the capabilities. However, when you use FireMonkey, LiveBindings is mandatory, so I strongly suggest you try it because, sooner or later, you will have to use it for some mobile stuff or some general FireMonkey applications.

There are many things to say about LiveBindings—we've only scratched the surface. For example, if you are building a big project and you have to handle or show some kind of recurrent entities, such as customers, orders, invoices, or users, you can create a `TListBindSourceAdapter<T>` descendant, compile it in a package, and install it in the tool palette so that every time you require it, you can simply drag-and-drop it on your data module or form.

Here are some links where you can find more information about LiveBindings:

- *XE3 Visual LiveBindings: User defined objects* at `http://blogs.embarcadero.com/jimtierney/2012/12/11/31961`
- *LiveBindings GridColumns* at `http://www.youtube.com/watch?v=K6Xu90Rtbys`
- *TBindSourceDB* at `http://www.malcolmgroves.com/blog/?p=1072`
- *TAdapterBindSource and binding to Objects* at `http://www.malcolmgroves.com/blog/?p=1084`
- *Updating Objects via an Adapter* at `http://www.malcolmgroves.com/blog/?p=1186`
- *Formatting your Fields* at `http://www.malcolmgroves.com/blog/?p=1226`
- *XE3 Visual LiveBindings: Samples* at `http://blogs.embarcadero.com/jimtierney/2012/10/21/31944`
- If you are interested in the core of LiveBindings, you can read an old article of mine that is still valid: `http://www.danieleteti.it/2011/08/30/in-the-core-of-livebindings-expressions-of-rad-studio-xe2/`

Showing complex vector shapes using paths

One of the biggest advantages of FireMonkey compared to VCL is its vector-based nature. Various visual parts can be created in FireMonkey using vector-based graphics (even if in some cases, using a bitmapped approach can be faster). In terms of vectorial graphics, there is a nice language called **Scalable Vector Graphics** (**SVG**) that allows you to define primitive shapes using a set of coordinates and not a raster image. So, you can stretch the image without losing its resolution, because the image is not actually stretched, but completely redrawn using the new coordinates. That's it; the SVG file is made up of coordinates and mathematical formulae to join them.

Inside the SVG language, there is an element called SVG path. The path element is used to define a path. So, what's a path?

A path is a sequence of instructions to draw something using primitives. Think of an SVG path as a language into another language (let's say a sort of internal DSL).

www.allitebooks.com

The following commands are available for path data:

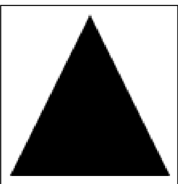
- `M`: This represents the `moveto` command (without drawing)
- `L`: This represents the `lineto` command (like `M` but drawing)
- `H`: This represents the `horizontal lineto` command
- `V`: This represents the `vertical lineto` command
- `C`: This represents the `curveto` command
- `S`: This represents the `smooth curveto` command
- `Q`: This represents the `quadratic Bézier curve` command
- `T`: This represents the `smooth quadratic Bézier curveto` command
- `A`: This represents the `elliptical Arc` command
- `Z`: This represents the `closepath` command

All of the commands can also be expressed with lowercase letters. Uppercase letters mean absolutely positioned and lowercase means relatively positioned.

So, the path `M50 0 L100 100 L0 100 Z` means:

- MoveTo X50 Y0
- LineTo X100 Y100
- LineTo X0 Y100
- ClosePath (draw a line to X150 Y0)

It draws a triangle like the following:

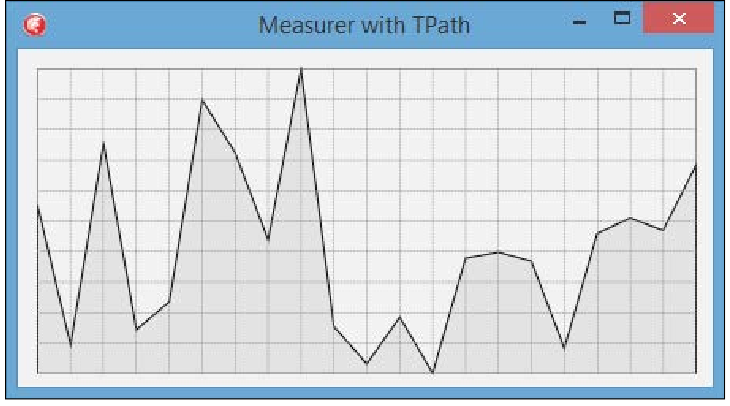


The triangle drawn by the sample path data

Getting started

In the FireMonkey framework, there is a component called **TPath** (it is defined in the `FMX.Objects.pas` unit; do not confuse it with the **TPath** component defined in the `System.IOUtils.pas` unit). The **TPath** component is able to interpret and show an SVG path. In this recipe, you'll see how to use it to draw complex vector shapes and fonts.

Let's say you want to monitor a continuous stream of data, maybe a value read from some kind of hardware or some value related to finance stock quotes. You want fresh data pushed from the right-hand side and oldest data removed from the left-hand side. At any time, you will see the last 20 values scrolling from the right-hand side to left-hand side. This is shown in the following screenshot:



Scrolling data in a line graph; new data are pushed from the right-hand side and old data are removed from the left-hand side

Usually, in order to write something like this, you require some third-party components or you have to write a lot of code to write all the values and axes and deal with proportional issues. Using the **TPath** component, you don't have to do all this! The **TPath** component with a proper `SVG PATH` is completely in charge to stretch and redraw your graphic in order to fit the drawing area.

How to do it...

1. Create a new FireMonkey desktop application.
2. Drop a **TPanel** component on to the form. In the **TPanel** component, put a **TPath** component and set its **Align** property to **alClient**. Now, the **TPath** component should fit into the **TPanel** component.
3. Drop another **TPath** component onto the first one and again, set its **Align** property to **alClient**.
4. Now you should have **TPanel** with two nested **TPath** components inside it.
5. Show the structure of the form (*Shift + Alt + F11*).
6. Name the first **TPath** component as `PathValues` and the second **TPath** component as `PathAxis`.

www.allitebooks.com

7. Drop a **TTimer** component on the form and double-click on it. Into the `OnTimer` event handler, write the following code:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  FValuesQueue.Add(Trunc(Random * 100));
  RefreshGraph;
end;
```

8. Set the **Interval** property of the timer property to `50`.
9. Now, go to the code editor and declare a private form instance variable:

```
FValuesQueue: TList<Integer>;
```

10. Create the `FormCreate` and `FormDestroy` event handlers and fill them with the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
  svggrid: string;
begin
  FValuesQueue := TList<Integer>.Create;
  for I := 0 to 19 do
    FValuesQueue.Add(0);

  svggrid := '';
  for I := 0 to FValuesQueue.Count - 1 do
    svggrid := svggrid + ' M' + I.ToString + ',0 V100';
  for I := 1 to 10 do
    svggrid := svggrid + ' M0,' +
                IntToStr(I * 10) + ' H20';
  PathAxis.Data.Data := svggrid;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FValuesQueue.Free;
end;
```

11. So far, you've declared and initialized your data container (the `TList<Integer>` item named `FValuesQueue`); now let's do something with its data. Create a private procedure named `RefreshGraph` and fill it with the following code:

```
procedure TMainForm.RefreshGraph;
var
  I: Integer;
  svg: string;
```

```
begin
  svg := 'M0,100 ';
  if FValuesQueue.Count > 19 then
  begin
    svg := svg + 'L0,' +
          (100 - FValuesQueue.First).ToString;
    FValuesQueue.Delete(0); //remove the first
  end;

  for I := 0 to FValuesQueue.Count - 1 do
  begin
    svg := svg + ' L' + I.ToString + ',' +
          (100 - FValuesQueue[I]).ToString;
  end;

  svg := svg + ' L' +
   IntToStr(FValuesQueue.Count - 1) + ' 100 ';
  PathValues.Data.Data := svg;
end;
```

12. Run the application.
13. Are you disappointed with the performance? In this case, the debugger load on the execution speed is heavy. So, to check the real drawing speed, run it without the debugger (*Shift* + *Ctrl* + *F9*).
14. You should now see the graph scrolling at a good speed.

How it works...

The architecture is simple—the timer is the (fake) data producer that fills the list. Then the list is used to draw the graph. After drawing the graph, the first list element is removed while waiting for the next one.

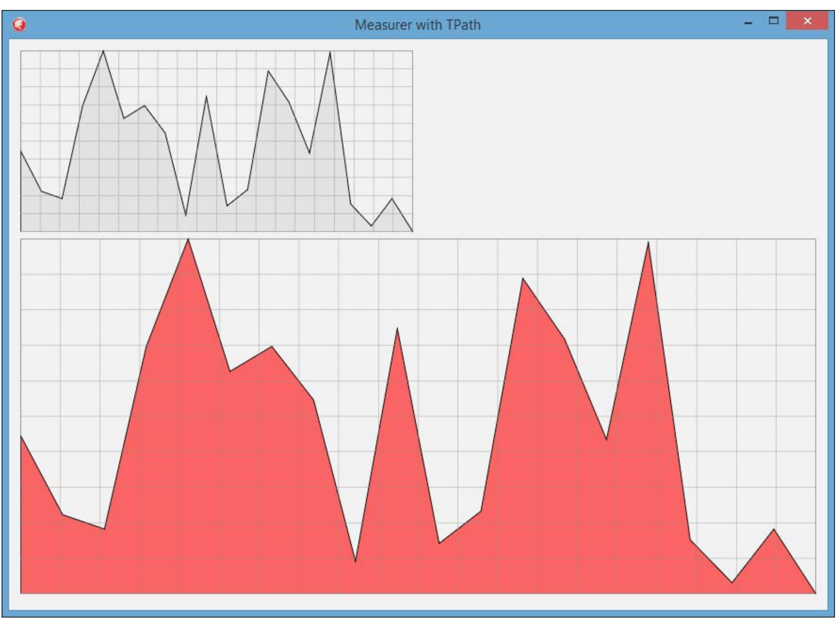
In a real-world application, some tuning may be necessary and in this case, a classic producer/consumer pattern is more suited to do this compared to a simple **TTimer**. However, in this sample, a normal **TTimer** component is enough.

A good thing to note is that you have a fixed coordinate system when drawing the values in the graph. You don't have to worry about form size, relative or absolute coordinates, and so on. All the details are handled by the **TPath** component.

www.allitebooks.com

So, if you need to add another scrolling graph of different size, you could use the same `SVG PATH` data to show the same graph on another area.

Let's add another `TPanel->TPath->TPath` triad on the form and make the **TPanel** component bigger than the previous one. With a little change in the code (the full code is available), you can have something like this:



Another scrolling graph showing the same values added without changing the drawing code

There's more...

The `SVG PATH` data can be very useful. If you require complex `SVG PATH` data, I suggest that you use a proper editor to generate the path. There is a nice online editor that can generate this kind of information called Method Draw and it's available at `http://editor.method.ac/`. The `SVG PATH` data can be also used to drive animations using the `TPathAnimation` component.

The producer/consumer cited in this recipe is a classic concurrency pattern. You can find more information on this at `http://javarevisited.blogspot.it/2012/02/producer-consumer-design-pattern-with.html`.

Using FireMonkey in a VCL application

As you probably know, VCL is incompatible with FireMonkey. What does it mean? As Embarcadero explains in the DocWiki:

> *FireMonkey (FMX) and the Visual Component Library (VCL) are not compatible and should not be used together in the same module. That is, a module should be exclusively one or the other, either FireMonkey or VCL. The incompatibility is caused by framework differences between FireMonkey (FMX) and VCL.*

However, there is still something that can be done to use FireMonkey functionalities in a VCL application.

Getting started

It's very probable that a VCL application could gain benefits by using some components or functionalities present only in the FireMonkey framework. So what could be the solution? One solution is to create a Windows DLL that contains all the FireMonkey code and exposes a set of raw functions to access them. Then, the VCL application can load the DLL and call the exposed functions. Let's see this in action.

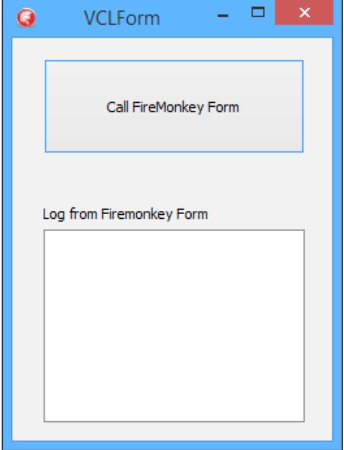
This recipe requires familiarity with some advanced Delphi concepts, so there will not be a step-by-step section; I'll only talk about the project code.

How to do it...

Let's begin!

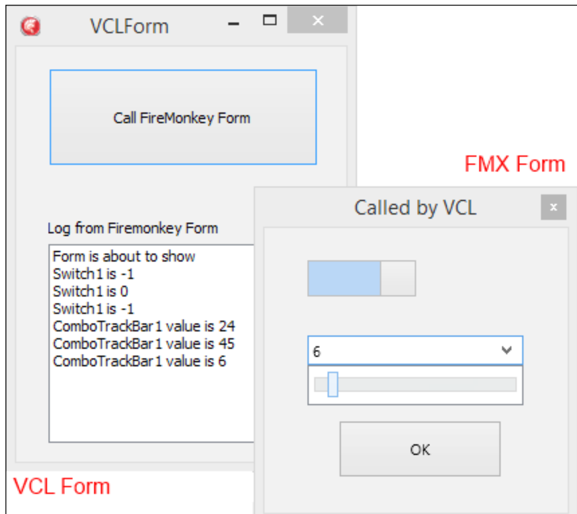
1. Open the recipe project group called `UsingFMXfromVCL.groupproj`. The group contains two projects:
   - A VCL application (`vclmainproject.exe`) that is your legacy application
   - A DLL project (`fmxproject.dll`) that contains all the FireMonkey stuff
2. To have an idea about the projects, go to **Project** | **Build all Projects**, select the `vclmainproject.exe` file, and hit *F9* to run it. The `fmxproject.dll` file has been compiled in the same folder of `vclmainproject.exe`. You should see the form shown here:

www.allitebooks.com



The VCL form that will use the DLL containing the FireMonkey code

3. By clicking on the **Call FireMonkey Form** button, you can call the FireMonkey DLL that will show a FireMonkey form that is able to send to the main form some information using a callback (we'll talk about it in a moment). The callback makes your project a little bit difficult, but being able to send something to the caller is a fundamental part of any integration.
4. If you click on the button and play with the FMX controls, you should get something like this:



The FireMonkey form used by the VCL application

How it works...

The `CommonsU.pas` unit is shared between the VCL and FMX projects and contains the declaration for the callback function as shown here:

```
type
  TDLLCallback = procedure(const Value: String);
```

The `DLLImportU.pas` unit is used only by the VCL project (because it needs to import the DLL functions). It is really simple and refers to the `TDLLCallBack` declaration:

```
unit DLLImportU;

interface

uses
  CommonsU;

procedure Execute(const Caption: String;
         Callback: TDLLCallback); stdcall; external 'fmxproject';

implementation

end.
```

These two files are the *bridge* between the VCL project and the FMX project. Now, let's see how the VCL project calls the FireMonkey DLL.

Using the **Project Manager**, select the VCL project main form. The `Button Click` event handler calls the `Execute` external function with the following code:

```
procedure MyCallBack(const Value: String);
begin
  VCLForm.ListBox1.Items.Add(Value);
  VCLForm.ListBox1.Update;
end;

procedure TVCLForm.btnCallFMXClick(Sender: TObject);
begin
  Execute('Called by VCL', MyCallBack);
end;
```

Notice that the `MyCallBack` procedure is not a form method but a simple procedure. This is the reason why I had to use an instance name of the form, `VCLForm`, and cannot use the implicit `Self` reference. Also, a normal string and a function pointer are passed to the `Execute` function. Notice that the function pointer is `MyCallBack` and not `MyCallBack()` (with parenthesis it means *call the procedure* and without parenthesis it means *the address of*).

www.allitebooks.com

The VCL project doesn't require further explanation. Let's switch to the FMX DLL. Using the **Project Manager**, select the `fmxproject.dll` file and go to **Project** | **View Source**.

The `library` project file contains the exported functions and the startup code to show the FMX form. Its code is shown here:

```
library fmxproject;

uses
  System.ShareMem, Winapi.Windows,
  System.SysUtils,System.Classes,
  FMXMainForm in 'FMXMainForm.pas' {Form1},
  CommonsU in 'CommonsU.pas';

{$R *.res}

procedure Execute(const Caption: String;
                            Callback: TDLLCallback); stdcall;
var
  frm: TForm1;
begin
  frm := TForm1.Create(nil);
  try
    frm.Caption := Caption; //use the passed string as Caption
    frm.FCallback := Callback; //link callback as form property
    frm.ShowModal;
  finally
    frm.Free;
  end;
end;

{ This is exported function that will be used by the VCL form }
exports Execute;

begin

end.
```

As you can see, the `callback` pointer has been assigned to a `form` property to be accessible from it. How will the FMX form use the `callback` pointer? In this recipe, it uses the `callback` pointer to send to the main VCL form some information about the components on it.

This is the relevant code of the main VCL form:

```
type
  TForm1 = class(TForm)
    btnClose: TButton;
    Switch1: TSwitch;
```

```
    ComboTrackBar1: TComboTrackBar;
    procedure btnCloseClick(Sender: TObject);
    procedure Switch1Switch(Sender: TObject);
    procedure ComboTrackBar1Change(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject;
                         var Action: TCloseAction);
  private

  public
    {This is the function pointer to the main VCL form callback}
    FCallback: TDLLCallback;
  end;

implementation

{$R *.fmx}

procedure TForm1.ComboTrackBar1Change(Sender: TObject);
begin
  //send the value of TComboTrackBar
  FCallback('ComboTrackBar1 value is ' +
            ComboTrackBar1.Value.ToString);
end;

procedure TForm1.FormClose(Sender: TObject;
                            var Action: TCloseAction);
begin
  //inform the main form about FMX form closing
  FCallback('Form is about to close');
end;

procedure TForm1.FormShow(Sender: TObject);
begin
  //inform the main form about FMX form showing
  FCallback('Form is about to show');
end;

procedure TForm1.Switch1Switch(Sender: TObject);
begin
  //inform the main form about the state of the Switch
  FCallback('Switch1 is ' + Switch1.IsChecked.ToString);
end;
```


www.allitebooks.com

The FMX-side code is not complex, and you can use whatever complex data structure you want to send information from the FMX form to the VCL form. A good solution for this is to define a simple textual protocol to allow a single callback to bring multiple types of information. For this kind of thing, I used to use JSON-serialized string. If the values sent by the callback are many, you can queue the values and process them as soon as possible; this is something like the producer/consumer design pattern.

There's more...

This recipe follows the *official* approach and uses two different projects (one VCL and one FireMonkey) to use the FireMonkey framework from a VCL application.

What if you have a legacy project where you'd like to use a FireMonkey DLL, but the legacy project is not in Delphi VCL (let say it is in C#, Visual C++, Python, or any other language that can load a DLL)? You can still use the same approach, but you cannot use Delphi-specific data types. So your strings should be `PChar` and so on. You can find more information on this at `http://delphi.about.com/od/objectpascalide/a/dlldelphi.htm`.

Just to be clear, keep in mind that mixing FireMonkey and VCL forms in the same application isn't officially supported. However, there are a number of libraries that aim to integrate VCL and FireMonkey forms in the same project.

Here's a short list in no particular order:

- Delphisorcery: `https://bitbucket.org/sglienke/dsharp` (using `DSharp.Windows.FMXAdapter.pas`)
- *firemonkey-container at* `https://code.google.com/p/firemonkey-container/`
- *MonkeyMixer updated for Delphi XE5* at `http://delphi.org/2013/09/monkeymixer-updated-for-delphi-xe5/`
- RemObjects *Hydra4* at `http://www.remobjects.com/hydra/`

In the recipe, you used a function pointer as a callback. If you want to know more about this type and others types of callback, check the following link:

`http://www.delphi-central.com/callback.aspx`

www.allitebooks.com

4
The Thousand Faces of Multithreading

In this chapter, we will cover the following topics:

- Synchronizing shared resources with `TMonitor`
- Talking with the main thread using a thread-safe queue
- Synchronizing multiple threads using `TEvent`
- Displaying a measure on a 2D graph like an oscilloscope

Introduction

Multithreading can be your biggest problem if you cannot handle it with care. In this chapter, we will discuss some of the main techniques to handle single or multiple background threads. We'll talk about shared resources synchronization and thread-safe queues and events. Multithreaded programming is a huge topic. So, after reading this chapter, although you will not become a master of it, you will surely be able to approach the concept of multithreaded programming with confidence and will have the basics to jump on to more specific stuff when (and if) you require them.

Synchronizing shared resources with TMonitor

`TMonitor` is a record used to synchronize threads. Just to be clear, we are talking about `System.TMonitor`, not `Vcl.Forms.TMonitor`.

Since Delphi 2009, the `TObject` instance size has been doubled to make room for an additional 4 bytes. What are these 4 bytes for? They provide `TMonitor` support!

Now, every `TObject` descendant can be used as a lock. The type that allows this is the `System.TMonitor` record, which implements a generic Monitor synchronization structure.

Getting ready

In this recipe, you'll face one of the classic multithreading problems—concurrent access to a shared file. Specifically, you'll have a lot of threads writing some information on a file—the same file—and all the threads have to be synchronized for this. Otherwise, the file will not be accessible due to locking, which will cause exceptions in your program code. This problem can be solved in a lot of ways, but `TMonitor` offers the simplest solution. Let's start.

How to do it...

Follow these step-by-step instructions to synchronize shared resources with `TMonitor`:

1. Create a new **VCL Forms Application** (navigate to **File** | **New** | **VCL Forms Application**).
2. Drop a **TButton**, a **TListBox**, and a **TTimer** component on the form.
3. Name the **TButton** component as `btnStart` and change the value of **Caption** to `Multiple writes on a shared file`.
4. Add a new unit to the project, call it `FileWriterThreadU.pas`, and add the following code to it:

```
unit FileWriterThreadU;

interface

uses
  System.Classes, System.SyncObjs,
  System.SysUtils, System.IOUtils;

type
  TThreadHelper = class helper for TThread
  public
    function WaitFor(
ATimeout: Cardinal): LongWord; platform;
  end;

  TFileWriterThread = class(TThread)
  private
    FStreamWriter: TStreamWriter;
  protected
    procedure Execute; override;
  public
```


www.allitebooks.com

```
    constructor Create(
AStreamWriter: TStreamWriter);
  end;

implementation

{$IF Defined(MSWINDOWS)}
uses
  Winapi.Windows;
{$IFEND}

constructor TFileWriterThread.Create(
   AStreamWriter: TStreamWriter);
begin
  FStreamWriter := AStreamWriter;
  inherited Create(False);
end;

procedure TFileWriterThread.Execute;
var
  I: Integer;
  NumLines: Integer;
begin
  inherited;
  NumLines := 11 + Random(50);
  for I := 1 to NumLines do
  begin
    TThread.Sleep(200);
    //here we are locking the shared resource
    TMonitor.Enter(FStreamWriter);
    try
      FStreamWriter.WriteLine(
         Format('THREAD %5d - ROW %2d',
         [TThread.CurrentThread.ThreadID, I]));
    finally
      //unlock the shared resource
      TMonitor.Exit(FStreamWriter);
    end;
    if Terminated then
      Break;
  end;
end;

function TThreadHelper.WaitFor(
   ATimeout: Cardinal): LongWord;
begin
```

```
{$IF Defined(MSWINDOWS)}
  Result := WaitForSingleObject(Handle, ATimeout);
{$ELSE}
  raise Exception.Create('Available only on MS Windows');
{$IFEND}
end;

initialization

Randomize; // we'll use Random function in the thread

end.
```

5. Go back to the form and add the following units in the `interface uses` section:
   - `System.Generics.Collections`
   - `FileWriterThreadU`
6. In the `private` section of the form, declare the following variables:

```
private
    FOutputFile: TStreamWriter;
    FRunningThreads: TObjectList<TFileWriterThread>;
```

7. In the `FormCreate` and `FormClose` event handlers, add the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FRunningThreads := TObjectList<TFileWriterThread>.Create;
  FOutputFile := TStreamWriter.Create(
    TFileStream.Create('OutputFile.txt',
                      fmCreate or fmShareDenyWrite));
end;

procedure TMainForm.FormClose(Sender: TObject;
   var Action: TCloseAction);
var
  Th: TFileWriterThread;
begin
  for Th in FRunningThreads do
    Th.Terminate;
  FRunningThreads.Free; // Implicit WaitFor...
  FOutputFile.Free;
end;
```

With the preceding code, you created a data structure to hold the thread list and file access. The `FOutputFile` variable is your shared resource for all the threads.

www.allitebooks.com

8. Create the `OnClick` event handler for `btnStart` and add the following code to it:

```
procedure TMainForm.btnStartClick(Sender: TObject);
var
  I: Integer;
  Th: TFileWriterThread;
begin
  for I := 1 to 10 do
  begin
    Th := TFileWriterThread.Create(FOutputFile);
    FRunningThreads.Add(Th);
  end;
end;
```

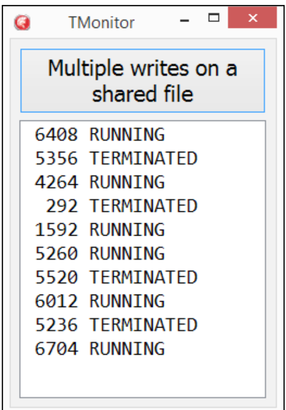
   The preceding code creates 10 threads that will contend for the shared resource `FOutputFile`.

9. Now, threads can run without problems but the UI doesn't have any information about their jobs. We want to check whether a thread is still running or is already terminated. So, let's create the event handler for the `Timer1.OnTimer` event using the following code:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
var
  Th: TFileWriterThread;
begin
  ListBox1.Items.BeginUpdate;
  try
    ListBox1.Items.Clear;
    for Th in FRunningThreads do
    begin
      if Th.WaitFor(0) = WAIT_TIMEOUT then
        ListBox1.Items.Add(
                 Format('%5d RUNNING', [Th.ThreadID]))
      else
        ListBox1.Items.Add(
                Format('%5d TERMINATED', [Th.ThreadID]))
    end;
  finally
    ListBox1.Items.EndUpdate;
  end;
end;
```

   The preceding code will iterate over the thread list and check the state of each of them. The resultant check will fill the `ListBox1` component.

10. Run the application and click on the button that is available on the form. You should see something like the following:



The main form showing thread statuses

11. `ListBox1` contains thread statues. When all threads terminate, you can open the file and see that each of them wrote information without interference from the others; no crashes, no data loss. Your multithreading application is working alright.
12. If you want to see the file while the threads are writing it, you can use one of the Unix tail clone options for Windows suggested in the *Creating a Windows service* recipe of *Chapter 1*, *Delphi Basics*.

How it works...

The `btnStart` event creates 10 threads and puts each of them in a simple generic list declared as `TObjectList<TFileWriterThread>`. This list will be used to iterate over the threads when terminating or checking the status of threads. Threads are not configured with `FreeOnTerminate` because we require a live reference to check their status.

The real work is done in the `Execute` method of `TFileWriterThread`. Let's check it out:

```
procedure TFileWriterThread.Execute;
var
  I: Integer;
  NumLines: Integer;
begin
  inherited;
  //decide how many numbers to write
  NumLines := 11 + Random(50);
  for I := 1 to NumLines do
  begin
```


www.allitebooks.com

```
      //wait a bit of time to simulate a higher workload
      TThread.Sleep(200);
      //acquire the lock on FStreamWriter.
      TMonitor.Enter(FStreamWriter);
      try
        //only one thread at a time can execute this code
        FStreamWriter.WriteLine(
              Format('THREAD %5d - ROW %2d',
              [TThread.CurrentThread.ThreadID, I]));
      finally
           //Be sure to release the lock. Otherwise all threads
           //will hang waiting to acquire the lock
        TMonitor.Exit(FStreamWriter);
      end;
      //if thread is terminated exit from the loop
      if Terminated then
        Break;
    end;
  end;
```

Another important piece of code is under the `TTimer` event handler:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
var
  Th: TFileWriterThread;
begin
  ListBox1.Items.BeginUpdate;
  try
    ListBox1.Items.Clear;
    for Th in FRunningThreads do
    begin
      //check if the thread if still running. Method WaitFor has
      //been introduced by a class helper in the
      //FileWriterThreadU.pas file, it is not part of TThread
      if Th.WaitFor(0) = WAIT_TIMEOUT then
        ListBox1.Items.Add(Format('%5d RUNNING',
                 [Th.ThreadID]))
      else
        ListBox1.Items.Add(Format('%5d TERMINATED',
         [Th.ThreadID]))
    end;
  finally
    ListBox1.Items.EndUpdate;
  end;
end;
```

The `WaitFor` method used in the `TTimer` event handler is not part of the standard `TThread` class but has been introduced using a class helper. Why? Because the standard `WaitFor` method present on the `TThread` class doesn't provide a timeout for the waiting, so it waits forever. If you want to check whether a thread is terminated or simply if you want to have the GUI responsive while waiting for the thread termination, you cannot do it using the `WaitFor` method. So, we added a new `WaitFor` method that provides a timeout. When you are calling `WaitFor(0)`, you are only asking whether a thread is still running. This is another good utilization of class helpers.

There's more...

Monitors are not a Delphi-specific concept; Wikipedia mentions it as follows:

> *"Monitors were invented by C. A. R. Hoare and Per Brinch Hansen, and were first implemented in Brinch Hansen's Concurrent Pascal language."*

To have a clear understanding of what a Monitor is and what's its main utilization, please read the Wikipedia article at `http://en.wikipedia.org/wiki/Monitor_%28synchronization%29`.

As a plus, a `TMonitor` class used in a smart way allows you to create a sort of "new language construct". Consider the following code:

```
procedure ExecWithLock(const ALockObj: TObject;
   const AProc: TProc);
begin
   System.TMonitor.Enter(ALockObj);
   try
      AProc();
   finally
      System.TMonitor.Exit(ALockObj);
   end;
end;
```

Using the preceding code, it is possible to write something like the following:

```
ExecWithLock(Obj,
  procedure
  begin
    //Here you have thread safe access to Obj
  end);
```

Cool, isn't it?

www.allitebooks.com

Talking with the main thread using a thread-safe queue

Using a background thread and working with its private data is not difficult, but safely bringing information retrieved or elaborated by the thread back to the main thread to show them to the user (as you know, only the main thread can handle the GUI in VCL as well as in FireMonkey) can be a daunting task. An even more complex task would be establishing a generic communication between two or more background threads. In this recipe, you'll see how a background thread can talk to the main thread in a safe manner using the `TThreadedQueue<T>` class. The same concepts are valid for a communication between two or more background threads.

Getting ready

Let's talk about a scenario. You have to show data generated from some sort of device or subsystem, let's say a serial, a USB device, a query polling on the database data, or a TCP socket. You cannot simply wait for data using `TTimer` because this would freeze your GUI during the wait, and the wait can be long. You have tried it, but your interface became sluggish… you require another solution!

In the Delphi RTL, there is a very useful class called `TThreadedQueue<T>` that is, as the name suggests, a particular parametric queue (a FIFO data structure) that can be safely used from different threads. How to use it? In the programming field, there is mostly no single solution valid for all situations, but the following one is very popular. Feel free to change your approach if necessary. However, this is the approach used in the recipe code:

1. Create the queue within the main form.
2. Create a thread and inject the form queue to it.
3. In the thread `Execute` method, append all generated data to the queue.
4. In the main form, use a timer or some other mechanism to periodically read from the queue and display data on the form.

How to do it…

Open the recipe project called `ThreadingQueueSample.dproj`. This project contains the main form with all the GUI-related code and another unit with the thread code.

The `FormCreate` event creates the shared queue with the following parameters that will influence the behavior of the queue:

- `QueueDepth = 100`: This is the maximum queue size. If the queue reaches this limit, all the push operations will be blocked for a maximum of `PushTimeout`, then the `Push` call will fail with a timeout.

- `PushTimeout = 1000`: This is the timeout in milliseconds that will affect the thread; in this recipe, it is the producer of a producer/consumer pattern.
- `PopTimeout = 1`: This is the timeout in milliseconds that will affect the timer when the queue is empty. This timeout must be very short because the pop call is blocking in nature, and you are in the main thread that should never be blocked for a long time.

The button labeled **Start Thread** creates a `TReaderThread` instance passing the already created queue to its constructor (this is a particular type of dependency injection called **constructor injection**).

The thread declaration is really simple and is as follows:

```
type
  TReaderThread = class(TThread)
  private
    FQueue: TThreadedQueue<Byte>;
  protected
    procedure Execute; override;
  public
    constructor Create(AQueue: TThreadedQueue<Byte>);
  end;
```

While the `Execute` method simply appends randomly generated data to the queue, note that the `Terminated` property must be checked often so the application can terminate the thread and wait a reasonable time for its actual termination. In the following example, if the queue is not empty, check the termination at least every 700 msec ca:

```
procedure TReaderThread.Execute;
begin
  while not Terminated do
  begin
    TThread.Sleep(200 + Trunc(Random(500)));
    // e.g. reading from an actual device
    FQueue.PushItem(Random(256));
  end;
end;
```

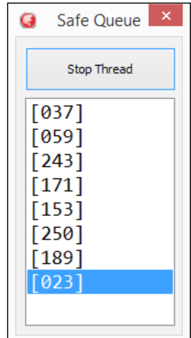
So far, you've filled the queue. Now, you have to read from the queue and do something useful with the read data. This is the job of a timer. The following is the code of the timer event on the main form:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
var
  Value: Byte;
begin
```


www.allitebooks.com

```
    while FQueue.PopItem(Value) = TWaitResult.wrSignaled do
    begin
      ListBox1.Items.Add(Format('[%3.3d]', [Value]));
    end;
    ListBox1.ItemIndex := ListBox1.Count - 1;
  end;
```

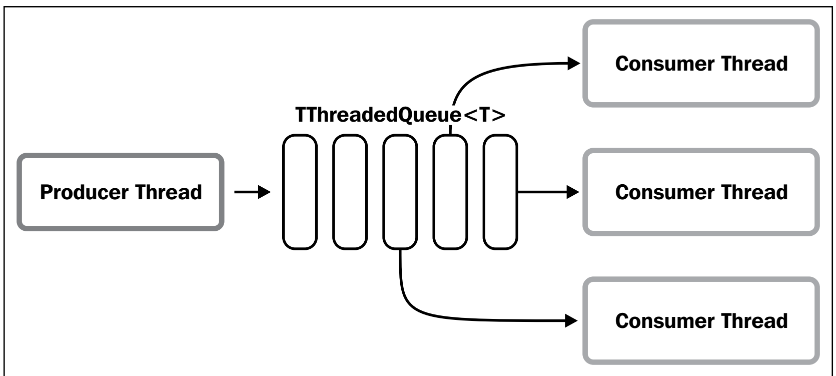
That's it! Run the application and see how we are reading the data coming from the threads and showing the main form. The following is a screenshot:



The main form showing data generated by the background thread

There's more

The `TThreadedQueue<T>` is very powerful and can be used to communicate between two or more background threads in a consumer/producer schema as well. You can use multiple producers, multiple consumers, or both. The following screenshot shows a popular schema used when the speed at which the data generated is faster than the speed at which the same data is handled. In this case, usually you can gain speed on the processing side using multiple consumers.



Single producer, multiple consumers

Synchronizing multiple threads using TEvent

The synchronization details we discussed so far were related to a data flow that is generated in the background thread context and has to be used in another thread. The other thread can be the main thread or another background thread. In this recipe, you'll use a simple synchronization mechanism called **event** that can be useful when you have to notify a new state, not necessarily new data. Obviously, the new state could also mean *there is new data to handle*. In such cases, the state change alerts you about new data being available.

Getting ready

The recipe scenario is simple: you have a lot of running threads that are doing something for you. You want to know when all of them are terminated. In this case, you can use a `TEvent` object (this is a tiny wrapper around OS `Event` object).

How to do it...

This recipe is a bit articulated, so we'll not discuss steps to recreate it. Please open the recipe project code named `ThreadsTermination.dproj`; let's look at it together.

The GUI is minimal; there is a button to run the threads and a listbox to show the current state of threads. The `FormCreate` event initializes a list to hold the threads that will be used later. When you click on the button, the program launches five threads. Each thread waits for a random amount of time then generates a random number that should represent your output data. The main thread has to be notified about the thread termination. The thread code is as follows:

```
unit MyThreadU;

interface

uses
  System.Classes, System.SyncObjs;

type
  TMyThread = class(TThread)
  private
    FEvent: TEvent;
    FData: Integer;
  protected
    procedure Execute; override;
  public
    constructor Create(AEvent: TEvent);
    destructor Destroy; override;
```


www.allitebooks.com

```
    property Event: TEvent read FEvent;
    function GetData: Integer;
  end;

implementation

uses System.SysUtils;

constructor TMyThread.Create(AEvent: TEvent);
begin
  FEvent := AEvent;
  inherited Create(False);
end;

destructor TMyThread.Destroy;
begin
  FreeAndNil(FEvent);
  inherited;
end;

procedure TMyThread.Execute;
begin
  TThread.Sleep(2000 + Random(4000));
  FData := Random(1000);
  // This call sets the internal event state to signaled
  FEvent.SetEvent;
end;

function TMyThread.GetData: Integer;
begin
  Result := FData;
end;

end.
```

In the thread, the constructor is injected a `TEvent` instance. When the thread does its job, it calls the `SetEvent` method on the event instance. This call sets the internal event state to `signaled`. What's that for? It is required because the main thread is waiting for this change. To be more precise, it is waiting to know when all the threads have called their `SetEvent` methods. The following function is used to check whether there are any running threads:

```
function TMainForm.AreThereThreadsStillRunning: Boolean;
var
  H: THandleObject;
begin
  Result := TEvent.WaitForMultiple(
                  Handles, 1, True, H) = wrTimeout;
end;
```

In the preceding code, the variable `Handles` is an array containing all the `Events` that have to be checked for termination.

The button event handler requires a bit of explanation. The code is as follows:

```
procedure TMainForm.btnStartClick(Sender: TObject);
var
  i: Integer;
  Evt: TEvent;
begin
  if (FThreads.Count > 0) and AreThereThreadsStillRunning then
  begin
    ShowMessage('Please wait, there are threads still running');
    Exit;
  end;
  FThreads.Clear;
  for i := 0 to High(Handles) do
  begin
    Evt := TEvent.Create;
    Handles[i] := Evt;
    FThreads.Add(TMyThread.Create(Evt));
  end;
  ListBox1.Items.Add('Threads running');
  Timer1.Enabled := True;
end;
```

When the user clicks on the button, the application checks whether there are any running threads from previous clicks. If so, inform the user with a `ShowMessage` and exit. If there are no running threads, the code fills the thread list with five threads. Each thread has its own `TEvent` instance to talk with. The reference to the `TEvent` variable is passed to the threads, but the threads have a property of accessing it during its runtime.

What is the best way to read the thread status? In a `TTimer` class. The code under the `OnTimer` event is the following; consider that this timer is normally disabled:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
var
  th: TMyThread;
begin
  if not AreThereThreadsStillRunning then
  begin
    Timer1.Enabled := False;
    ListBox1.Items.Add('All threads terminated');
    for th in FThreads do
    begin
      ListBox1.Items.Add(
```


www.allitebooks.com

```
            Format('Th %4.4d = %4d', [th.ThreadID, th.GetData]));
      end;
    end;
  end;
```

With this last procedure, you retrieved the thread status; when all threads finished running, you also retrieved the *calculated* value.

There's more...

The event object is used to send a signal to a thread indicating that a particular event has occurred inside another thread. The event does not carry information; it simply informs that "something has happened". It is simple, but can be useful in creating very complex synchronization mechanisms between two or more threads.

Events can be in a signaled state or not. If you want to have a deeper knowledge about the event objects and its utilization, visit the following links:

- `http://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Waiting_for_a_Task_to_Be_Completed`

Displaying a measure on a 2D graph like an oscilloscope

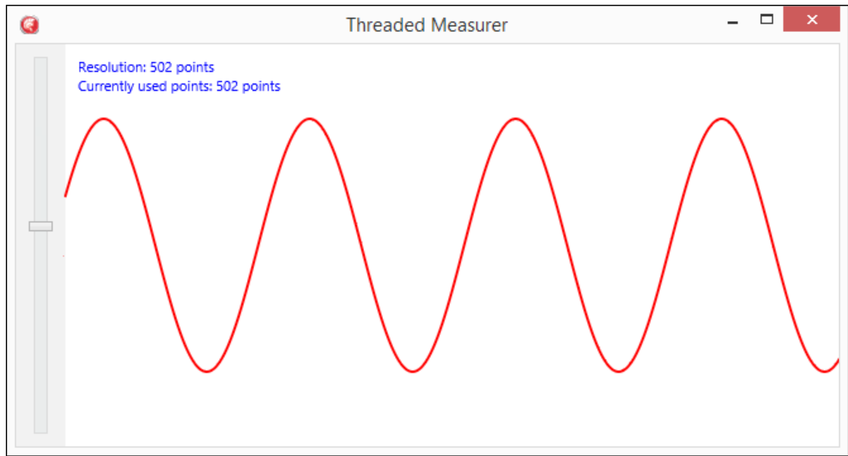
An **oscilloscope** is a type of electronic test instrument that allows the observation of constantly varying signal voltages. Usually, information is shown as a two-dimensional plot graph of one or more signals as a function of time. In this recipe, you'll implement a type of oscilloscope to display data generated by a background thread. Obviously, in this recipe, you'll not create an accurate oscilloscope, rather a nice real-world utilization of retrieving data and using it continuously in the GUI.

Getting ready

You'll use the `TThreadedQueue<Extended>` class to bring out data from the background thread to the main thread. The approach is similar to that shown in the recipe *Talking with the main thread using a thread-safe queue*, but in this case, we've to show data in a complex way—on a 2D graph showing only the last *n* data retrieved.

How to do it...

This recipe has a background thread acting like an *analogic signal generator* that is able to generate a sine style stream of data and a graph that plots these data. The resulting application is as follows:



The main form showing a sine function generated by a background thread

You can adjust the resolution of the plot (number of points used to draw the sine) using the trackbar on the left-hand side. Let's see the most important parts.

The thread used as signal generator is very simple. As shown in the following code, it uses the `System.Math.Sin` function to generate a sine wave form. Every 10 milliseconds c.a., a new value is appended to the queue; this value is the "sample" you get from the measured system. The code is as follows:

```
procedure TSignalGeneratorThread.Execute;
var
  Value: Extended;
begin
  inherited;
  Value := 0;
  while not Terminated do
  begin
    TThread.Sleep(10);
    FQueue.PushItem(Sin(Value) * 100);
    Value := Value + 0.05;
    if Value >= 360 then
      Value := 0;
  end;
end;
```


www.allitebooks.com

Being a classic producer/consumer, this architecture has to deal with the classic problem of a queue being full and not accepting any data until someone starts to dequeue from it. At regular intervals, a `TTimer` dequeues all the values from the queue and appends them to a different queue living in the main thread.

This queue must have a fixed size, so if there are more values than what is defined by the resolution, the oldest values are dequeued until the queue size is equal to the maximum length permitted. This adjustment is done in the timer event handler with the following code:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
var
  Value: Extended;
  QueueSize: Integer;
begin
  // put read values in the display list...
  // max FMaxValuesCount values
  while FValuesQueue.PopItem(QueueSize, Value) = wrSignaled do
  begin
    FDisplayList.Add(Value);
  end;
  // remove values from the head of the list...
  while FDisplayList.Count > FMaxValuesCount do
  begin
    FDisplayList.Delete(0);
  end;
  // RefreshGraph;
  pb.Repaint;
end;
```

The actual values are plotted on a simple 2D graph using `TPaintBox` as canvas. Remember that only the main thread should repaint and call paint procedures. The following is the code in the `OnPaint` event used to draw the plot:

```
procedure TMainForm.pbPaint(Sender: TObject; Canvas: TCanvas);
var
  Values: TPolygon;
  I: Integer;
  XStep: Extended;
  YCenter: Integer;
begin
  Canvas.BeginScene;
  // prepare scene
  Canvas.Stroke.Thickness := 1;
  Canvas.Fill.Color := TAlphaColorRec.White;
```

```
    Canvas.FillRect(RectF(0, 0, Canvas.Width, Canvas.Height),
                          0, 0, [], 1);
    Canvas.Fill.Color := TAlphaColorRec.Blue;
    Canvas.FillText(RectF(10, 10, Canvas.Width, 40),
         'Resolution: ' + MaxValuesCount.ToString + ' points',
          false, 1, [], TTextAlign.taLeading, TTextAlign.taLeading);
    Canvas.FillText(RectF(10, 25, Canvas.Width, 40),
          'Currently used points: ' + FDisplayList.Count.ToString +
          ' points', false, 1, [],
          TTextAlign.taLeading, TTextAlign.taLeading);

    // drawing points
    Canvas.Stroke.Thickness := 2;
    Canvas.Stroke.Color := TAlphaColorRec.Red;
    SetLength(Values, FDisplayList.Count);
    XStep := Canvas.Width / FDisplayList.Count;
    YCenter := Canvas.Height div 2;
    for I := 0 to FDisplayList.Count - 1 do
    begin
      Values[I].X := XStep * I;
      Values[I].Y := YCenter - FDisplayList[I];
    end;
    Canvas.DrawPolygon(Values, 1);
    Canvas.EndScene;
  end;
```

There's more...

Showing dynamically changing data is always a challenge and is a typical synchronization problem if you have to read from a blocking and very fast data source. However, using queues in an efficient way can help to reach the correct architecture. If you have very high concurrency (many consumers or many producers) or a very high producer speed compared to the consumer's speed, you may have some performance improvements using lock-free data structures.

Unluckily, in Delphi there are no ready-to-use lock-free data structures; however, there are very good libraries, even open source, that implement it in the context of multithreaded programming. One of the most popular libraries is the open source **OmniThreadLibrary** from Primož Gabrijelčič (`https://code.google.com/p/omnithreadlibrary/`).

www.allitebooks.com

5
Putting Delphi on the Server

In this chapter, we will cover the following recipes:

- Web client JavaScript application with WebBroker on the server
- Converting a console service application to a Windows service
- Serializing a dataset to JSON and back
- Serializing objects to JSON and back using RTTI
- Sending a POST HTTP request encoding parameters
- Implementing a RESTful interface using WebBroker
- Controlling remote applications using UDP
- Using App Tethering to create a companion app
- Creating DataSnap Apache modules

Introduction

In this chapter, we'll see how nicely Delphi can behave when it runs on the server. Most server-side technology today is scripted or managed, and this is usually a good thing. However, Delphi can be used to create very powerful enterprise servers with no external dependencies and great performances, and to do all these things you require much less hardware power and memory as compared to, let's say, a J2EE server. Moreover, we'll see how to handle some of the most common problems faced with web servers such as serialization, MIME types, HTML encoding, and many more.

Web client JavaScript application with WebBroker on the server

In this recipe, we'll learn how to use web client JavaScript application with WebBroker on the server. We'll also look at retrieving the people's list, creating or updating a person's record, deleting a person's record, and running the application.

Getting ready

This recipe uses two external open source projects:

- `DelphiMVCFramework`: This is a powerful Delphi framework to develop RESTful web services. This is written by Daniele Teti (me). The project website is `https://code.google.com/p/delphimvcframework/`.
- `jTable`: This is a jQuery plugin to create Ajax-based CRUD tables. This is written by Halil İbrahim Kalkan. The project website is `http://jtable.org/`.

We'll start by downloading these libraries and put each ZIP file in a folder, let's say `C:\DelphiBook\Libs`.

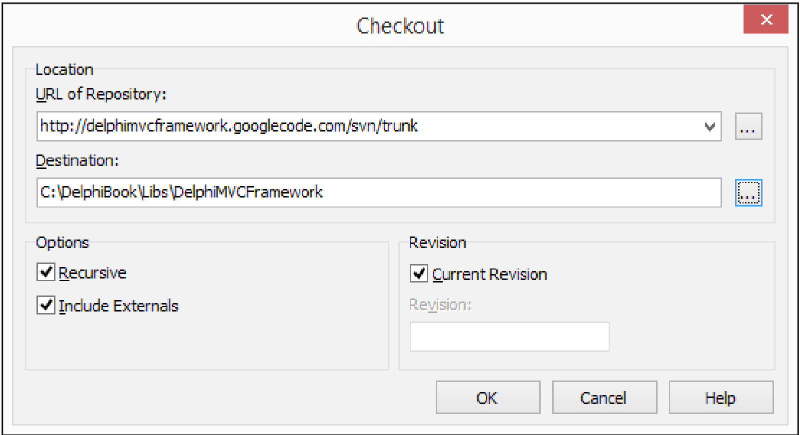
To download the DelphiMVCFramework (DMVCFramework for short), go to the project website and download the development trunk using a subversion client. There are a lot of subversion clients. A good general-purpose solution is TortoiseSVN, a well-integrated Windows shell extension able to access remote and local SVN repositories directly from Windows Explorer (TortoiseSVN can be downloaded from `http://tortoisesvn.net/`). You can also use the command-line version and then use the following command line:

```
svn checkout http://delphimvcframework.googlecode.com/svn/trunk/
   delphimvcframework
```

Alternatively, you can use Delphi directly to download the repository. Go to **File** | **Open from version control...**.

www.allitebooks.com

Then, in the window that appears, write the following information and click on **OK**:



The dialog used to download the DMVCFramework project from its public SVN repository

Now, the wizard-integrated SVN client will download all the necessary files. At the end of the process, the wizard will ask you about which project to open. Click on **Cancel** and close the dialog. The DMVCFramework files have been downloaded in the `C:\DEV\DMVCFramework` directory; configure the Delphi library path to point there.

Now, download the `jTable.zip` file from the URL `http://jtable.org/Home/Downloads`.

The latest version at the time of writing is Version 2.4.0, which is available directly from the URL `http://jtable.org/downloads/jtable.2.4.0.zip`.

Download the most updated version. Unzip the ZIP file and put the folder in `C:\DelphiBook\Libs\jtable.2.4.0`.

In the recipe project, there is a downloaded copy of the sources. However, you can use this procedure if you want to download a fresh version of the sources.

Now that you know how and where to retrieve the external projects used in this recipe, let's start with the explanation.

How to do it...

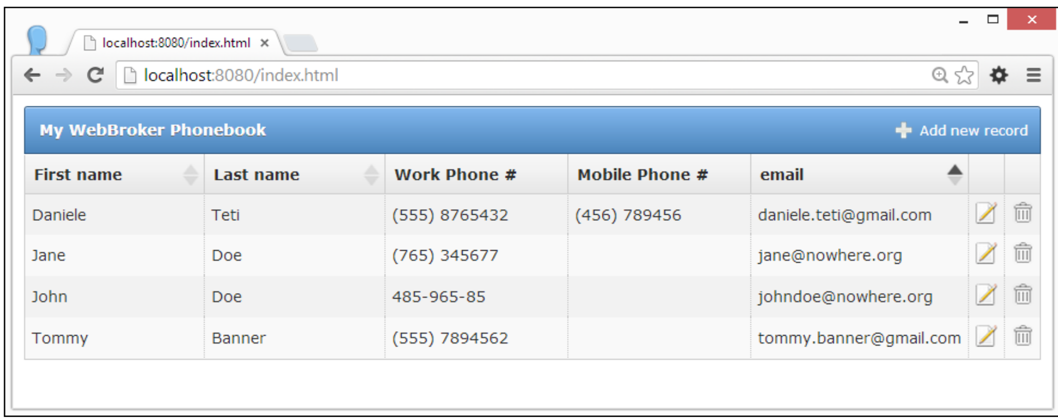
1. Open the recipe project `PhoneBookServer.dproj` from this chapter's recipe folder.

   This is a WebBroker project. WebBroker is a technology available since Delphi 4, which helps in creating web server application in an HTTP/HTTPS interface.

More information about WebBroker can be found at the following URLs:

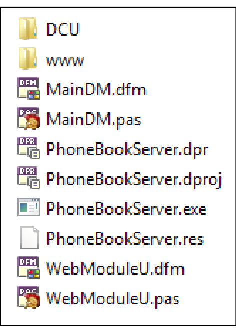
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Creating_Internet_server_applications_Index`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Using_Web_Broker_Index`

2. In this recipe, we'll see a simple **Create Retrieve Update and Delete** (**CRUD**) for an InterBase database table. Here's the final application running in a browser:



The final web application running in a browser

Take a look at the project folder. When you write WebBroker applications, the relative position of the static files used by the web application is important, and we've to deliver some static files to our clients.



The project folder layout

www.allitebooks.com

The DCU folder contains all the generated DCUs, while the www folder will be our document's root folder for the static files. In the www folder, you have an index.html file and a lib folder. In the lib folder, there is a folder containing the jTable library. Our application is a web client app, which means that what a user sees in the browser is not completely generated by the server and sent to the client, but the client has only the initial HTML, it will use JavaScript code to request data to the server using Ajax. When the server data are on the client (usually the data are transferred as JSON), the JavaScript code assembles the data and HTML to generate the final DOM. In this recipe, we'll use the jTable to avoid all boring HTML writing to create a simple CRUD interface.

Let's start from the initial HTML file retrieved by the client. This is the file that *starts* our application, and the JavaScript inside it will download the actual data to show. If you open it using a normal text editor (better if with syntax highlighting), you will see that the following external files are loaded:

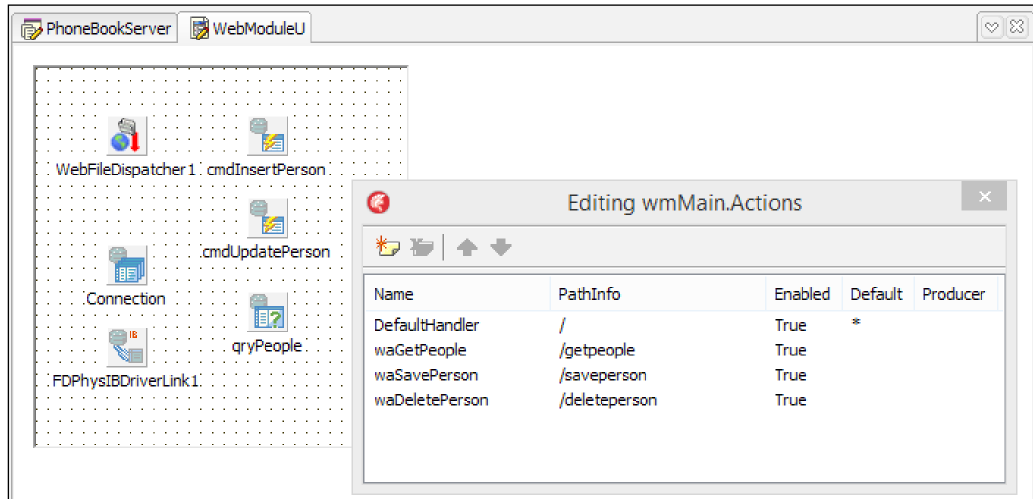
- The jQuery library from the Google CDN
- The jQuery-UI library from http://code.jquery.com/
- The jTable from a local copy
- The jQuery-UI CSS for a specific theme from the http://code.jquery.com
- The jTable CSS theme from our local copy

These files are required by our web client app. The jTable library allows you to generate a complete grid with embedded editing functionalities, only providing specific URLs to be invoked. We'll provide the following URLs in the WebBroker server:

- /index.html: This delivers the main file.
- /getpeople: This returns a JSON array of JSON objects with the database data.
- /saveperson: This can be invoked to create or update a person on the database. If there is an ID field, the person will be updated, otherwise created and a new ID will be provided by the database.
- /deleteperson: This deletes a person with a specified ID.

Note that this server is not a RESTful server. All the HTTP resources are invoked using a POST method. We are using plain WebBroker here, and the DMVCFramework is used only to easily serialize data retrieved from the database. A real RESTful server will be developed in the *Sending a POST HTTP request encoding parameters* recipe of this chapter.

Let's get back to Delphi and the recipe project. Open the `WebModule`, and the show its Actions property. You should see something similar to the following screenshot:



The WebModule and its actions

The `WebFileDispatcher` is configured to point to the `www` folder as its main root folder. In this way, all the files in that folder (that have permitted extensions) will be visible to the client.

FireDAC components are used to access the database. There's an FDConnection pointing to a local InterBase database placed in the DATA folder (to run this project, you have to start the InterBase service from the Service Control Panel). For each SQL statement, there is a component dedicated apart from the `DELETE` that is executed directly on the connection.

At the startup, we've to activate the database connection. Here's the FDConnection `BeforeConnect` event handler:

```
procedure TwmMain.ConnectionBeforeConnect(Sender: TObject);
begin
  Connection.Params.Values['Database'] :=
    TPath.GetDirectoryName(WebApplicationFileName) +
      '\..\..\DATA\SAMPLES.IB';
end;
```

Retrieving the people list

The client will issue a request to `/getpeople` and the server has to respond with a JSON array of JSON objects. This request is handled by the action `waGetPeopleAction`. The event handler contains the following code:

```
procedure TwmMain.wmMainwaGetPeopleAction(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse;
```


www.allitebooks.com

```
    var Handled: Boolean);
var
  JPeople: TJSONArray;
  SQL: string;
  OrderBy: string;
begin
  SQL := 'SELECT * FROM PEOPLE ';
  OrderBy := Request.QueryFields.Values['jtSorting'].Trim.ToUpper;
  if OrderBy.IsEmpty then
  begin
    SQL := SQL + 'ORDER BY FIRST_NAME ASC';
  end
  else
  begin
    if TRegEx.IsMatch(OrderBy, '^[A-Z,_]+[ ]+(ASC|DESC)$') then
    begin
      SQL := SQL + 'ORDER BY ' + OrderBy;
    end
    else
      raise Exception.Create('Invalid order clause syntax');
  end;

  // execute query and prepare response
  qryPeople.Open(SQL);
  try
    JPeople := qryPeople.AsJSONArray; //ObjectsMappers
  finally
    qryPeople.Close;
  end;
  PrepareResponse(JPeople, Response);
end;
```

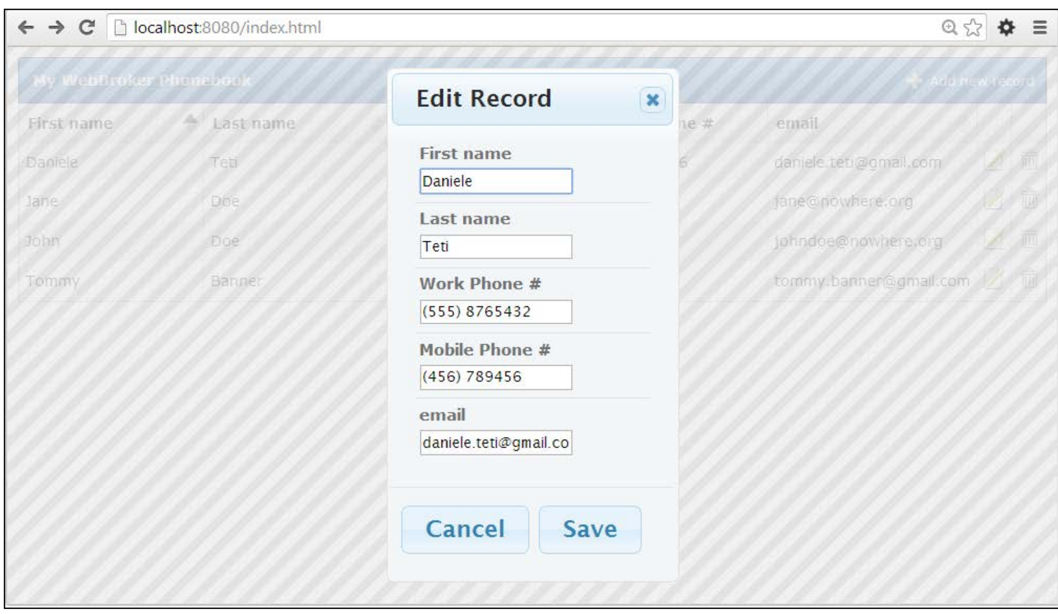
This method executes a query on the `PEOPLE` table, and then serializes the dataset returned using a class helper introduced by the `ObjectsMappers.pas` unit (which is part of the DMVCFramework).

jTable can also handle the sorting on the grid columns. To do this, send another request to the server with a parameter named `jSorting` containing the field and the direction of the order in the form: `first_name asc` or `last_name desc`. This is a nice feature; however, we cannot simply concatenate this string to the SQL. We've to sanitize it to avoid SQL injection attack. So, there is a regular expression to check whether the jSorting parameter contains only allowed characters and is composed by two words. We do not control the field on which ordinate is a valid field because the select will issue an error in that case.

The `PrepareResponse` method is required to correctly prepare the response to communicate with the jTable. If you want to understand the details, check the jTable Getting Started page at `http://jtable.org/GettingStarted`.

Creating or updating a person

jTable allows the user to create a new record or modify a record already created. Here, the GUI used to modify a person's request is shown as follows:



The Edit dialog generated by the web client app

When the data has been filled, the user can click on **Save** and then all the data are sent to the server in a `POST` request. This request is handled by the action `waSavePersonAction` invoked with the `/saveperson` path. Here's the code used to create or update a record:

```
procedure TwmMain.wmMainwaSavePersonAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  InsertMode: Boolean;
  JObj: TJSONObject;
  LastID: Integer;
  HTTPFields: TStrings;
  procedure MapStringsToParams(AStrings: TStrings;
    AFDParams: TFDParams);
  var
    i: Integer;
```


www.allitebooks.com

```
    begin
      for i := 0 to HTTPFields.Count - 1 do
      begin
        if AStrings.ValueFromIndex[i].IsEmpty then
          AFDParams.ParamByName(AStrings.Names[i].ToUpper).Clear()
        else
          AFDParams.ParamByName(AStrings.Names[i].ToUpper).Value :=
            AStrings.ValueFromIndex[i];
      end;
    end;

  begin
    HTTPFields := Request.ContentFields;
    InsertMode := HTTPFields.IndexOfName('id') = -1;
    if InsertMode then
    begin
      MapStringsToParams(HTTPFields, cmdInsertPerson.Params);
      cmdInsertPerson.Execute();
      LastID := Connection.GetLastAutoGenValue('GEN_PEOPLE_ID');
    end
    else
    begin
      MapStringsToParams(HTTPFields, cmdUpdatePerson.Params);
      cmdUpdatePerson.Execute();
      LastID := HTTPFields.Values['id'].ToInteger;
    end;

    // execute query and prepare response
    qryPeople.Open('SELECT * FROM PEOPLE WHERE ID = ?', [LastID]);
    try
      PrepareResponse(qryPeople.AsJSONObject, Response);
    finally
      qryPeople.Close;
    end;
  end;
```

The simple trick used in this code to determine whether it requested an `INSERT` or an `UPDATE` query is to check whether a field named `ID` is present in the `POST` fields. If an `ID` field is present, then we have to generate an `UPDATE`, otherwise, an `INSERT`.

Deleting a person

Deleting a person's record is the simplest method. The code of the `waDeletePerson` action is invoked with the `/deleteperson` path. Here's the code:

```
procedure TwmMain.wmMainwaDeletePersonAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  Connection.ExecSQL('DELETE FROM PEOPLE WHERE ID = ?',
    [Request.ContentFields.Values['id']]);
  PrepareResponse(nil, Response);
end;
```

Only one thing to note, we didn't use a specific command to issue the SQL statement but directly made the connection.

Running the application

Hitting *F9*, you should see a console window informing you that a server is started. Open the browser and point it to `http://localhost:8080`.

You should see what is shown in the first screenshot. If not, try to check the following:

- Is the port 8080 free?
- Is the InterBase database running correctly?
- Is the URL written correctly?

There's more...

This is only a small introduction to what you can do with WebBroker and a number of good JavaScript libraries. There are a lot of articles about WebBroker; some of them are a bit old, but most are still applicable to XE6. After reading the current documentation on the Embarcadero DocWiki, have a look to the article at `http://delphi.about.com/library/bluc/text/uc060901a.htm`.

WebBroker can also create ISAPI DLLs for Microsoft Internet Information Server and Apache module DLLs for the Apache `httpd` webserver. If you plan to deploy your web application on a production public server, you should consider putting your application behind a full-flagged web server such as Apache or IIS.

Another solution is to use the simple webserver created by Delphi and put a reverse proxy (`http://en.wikipedia.org/wiki/Reverse_proxy`) in front of it.

However, if you use the application in your intranet, it is safe enough to publish it as a console application, or (recommended) you can use this application as a Windows Service, directly on a server in your LAN.

www.allitebooks.com

More good news is that WebBrokers WebModules are independent from the final program type where they will be linked. So, you can develop a console application, debug it, and then convert it into a Windows service, an Apache module, or an ISAPI DLL with few clicks.

Converting a console service application to a Windows service

Writing and debugging a Windows service can be difficult and slow. In the *Creating a Windows service* recipe in *Chapter 1*, *Delphi Basics*, we learned how to write and debug a Windows service from scratch, but in some cases you already have a console or VCL application that already does its job. However, it would be better if the console or VCL application could be recreated as a Windows service.

Getting ready

In this recipe, we'll take the WebBroker application created in the previous recipe as a console application and convert it to a full flagged Windows service. The same approach can be used for any type of service-like application that currently is not built as a service.

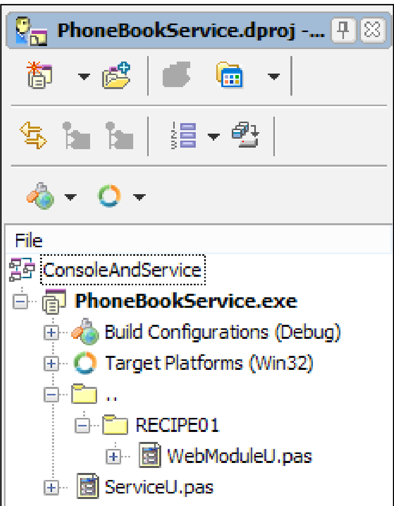
As a bonus, we'll learn that if correctly architected, a project can be compiled as a console or VCL application and, without much change, as a Windows service as well. WebBroker is particularly well architected to do so, so our application will benefit from it.

How to do it...

Let's execute the following steps to convert a console service application to a Windows service:

1. Create a new Service Application by going to **File** | **New** | **Other**... and then going to **Delphi Projects** | **Service Application**.
2. As soon as Delphi creates the project template, save all the files with the following names:
   - Save the project as `PhoneBookService.dproj`
   - Save the service module as `ServiceU.pas`
3. Show the object inspector for the service module and set the following properties:
   - `AllowPause = False`
   - `DisplayName = PhoneBookService`

4. Now, add to the project the web module from the *Web client JavaScript application with WebBroker on the server* recipe (the file is named `WebModuleU.pas` and should be under `Chapter05\CODE\RECIPE01`). This step allows us to reuse the same code written for the console application, and for the service application as well.
5. Now, your Project Manager should look like the following screenshot:



The Project Manager after adding the WebModuleU.pas from the previous recipe

6. Now we've to wire some things. Open the `ServiceU.pas` file and add the `IdHTTPWebBrokerBridge` unit in the `uses` clause. This allows us to create an internal HTTP service in our Windows service.
7. Now in the private part of the `TPhoneBook` declaration, add the following lines of code:

```
private
  LServer: TIdHTTPWebBrokerBridge;
```

8. In the implementation section of the `ServiceU.pas`, add the following `uses` clause:

```
uses
    Web.WebReq, WebModuleU;
```

9. Now, we've to handle the TCP server and the class registration for the WebBroker. Let's create some `TPhoneService` event handlers.
10. Create the `OnCreate`, `OnStart`, and `OnStop` event handlers and fill them with the following code:

```
procedure TPhoneBook.ServiceCreate(Sender: TObject);
begin
  if WebRequestHandler <> nil then
    WebRequestHandler.WebModuleClass := WebModuleClass;
```


www.allitebooks.com

```
end;

procedure TPhoneBook.ServiceStart(Sender: TService;
  var Started: Boolean);
begin
  LServer := TIdHTTPWebBrokerBridge.Create(nil);
  LServer.DefaultPort := 8080;
  LServer.Active := True;
end;

procedure TPhoneBook.ServiceStop(Sender: TService;
  var Stopped: Boolean);
begin
  LServer.Free;
end;
```

11. Build the project.
12. Copy the `www` folder from the previous recipe and put it at the same level of the compiled service.
13. Our service should be ok. Start a command prompt as the administrator, go to the folder where the service executable is, and write the following command line:

    ```
    PhoneBookService.exe /install
    ```

14. A message dialog should inform you that the service has been installed correctly.
15. Now, go to the Services management console and you should see the new service named `PhoneBookService` listed among the others. Start it and navigate with your browser to the following URL:

    ```
    http://localhost:8080
    ```

16. Now, you should see the WebBroker Phone Book page with some people listed.
17. If the peoples' list is not loading, probably the service didn't reach the database. Check whether the database is running and if the code under the `OnBeforeConnect` of the database connection sets the correct connection string.

How it works...

This recipe is really simple. The entire dirty job is done by the WebBroker framework and by the `TIdHTTPWebBrokerBridge` class. As a general rule, when you have a TCP service that should listen while the service is running, simply start the TCP service in the `OnStart` event handler and stop it in the `OnStop` event handler. If your logic is more complex, you should be able to separate all the things that make the service available (start) and put them in the `OnStart` event handler, while all the things that make the service unavailable and free the resources (stop) should be put in the `OnStop` event handler.

If you have to support the paused state as well, you have to find out what a paused state means for your service. For this recipe, a paused state is equivalent to a stopped state, so I simply removed the possibility to pause the service.

There's more...

Every application could have a different way to be converted as a Windows service; however, you should be aware that your service runs in a different environment with respect to your *normal* application. Two notable differences are as follows:

- Services can run out of any user context, and they usually do. They usually run as Local System Account (as the service in this recipe) but can be configured to run as a particular user.
- The current folder for a service is not the folder where the executable is, but the `System32` folder under `C:\Windows\` for the 64-bit service, the same as for 32-bit services when run on 32-bit machines, and `C:\Windows\SysWOW64` for 32-bit services that run on 64-bit machines.

Serializing a dataset to JSON and back

At the time when almost all the Delphi program was client/server, or in general, when the Delphi program was always connected to the database server in a full-connected scenario, the datasets' serialization was a niche topic. There were really few situations where you really need this kind of functionality in the core of your application. Were the '90s!' Now, however, making your data available to other programs or getting data from other software running somewhere in the world is the normal. In some cases, the other "programs" are not written in Delphi, so the `DataSet.SaveToFile` method (or other serialization that use a proprietary or *exotic* format) is no longer enough. Let's say we've a JavaScript frontend for our Delphi application server. Your data should be "dedelphized" (I've just coined this word) and should be independent from the backend programming language or framework used. Delphi have a lot of serialization facilities, but there isn't a well-known way to serialize a dataset in the JSON standard format and deserialize a standard JSON in a DataSet (there are some units containing JSON serialization stuff but the resultant JSON is very Delphi-oriented and not well suited to be used to communicate with other non-Delphi programs). In the DataSnap framework, there are classes devoted to do such things and are all contained in the `Data.DBXJSONCommon` unit; however, at the time of writing, they are not designed to be flexible enough to be used in heterogeneous scenarios. Don't be afraid. In this recipe, we'll solve all these problems!

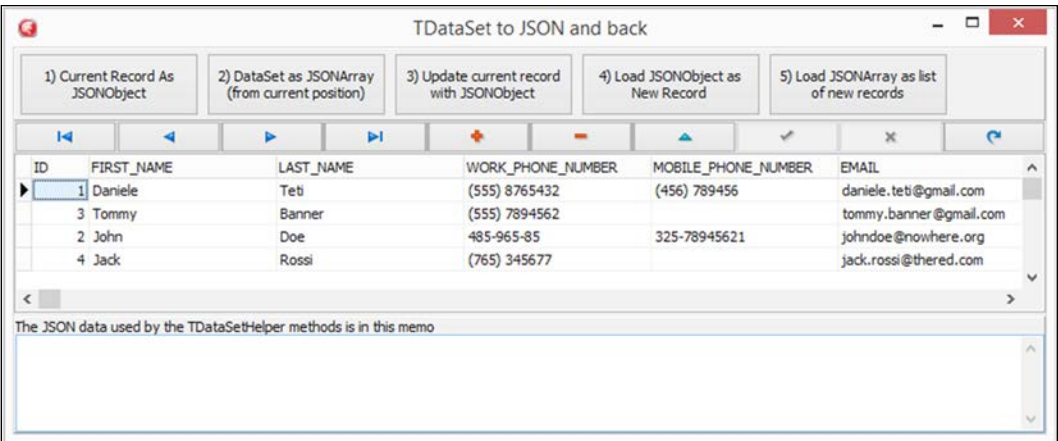
www.allitebooks.com

Getting ready

We'll use a subproject of the already mentioned DelphiMVCFramework (more information can be found at `https://code.google.com/p/delphimvcframework/`) called Mapper.

The Mapper is a micro framework aimed at developers in mappings and conversions and will be used in this recipe and in the next one. Firstly, get the DelphiMVCFramework using the snapshot ZIP file (as mentioned in the project website) or the subversion repository. Then, create a new VCL project to do some experiments. This recipe is not a complete project, but a set of demos showing what you can do with your datasets using this open source micro framework.

The `Demo` project is a simple list of buttons: a **TDBGrid** and a **TMemo** component to show the last JSON serialization. This is shown in the following screenshot:



Demo for DataSets JSON serialization

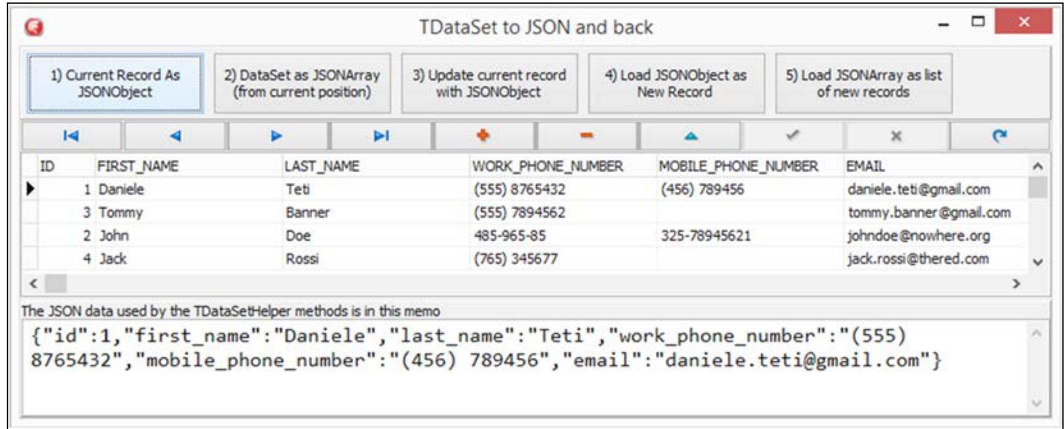
How to do it...

Under each button, a particular mapper feature is used. The mapper serializes data in the JSON format using a simple object or an array of objects. To be clear, a single record will be serialized as a JSON object while a full dataset (or a set of records) are serialized as a JSON array containing JSON objects, one for each serialized record. In the `uses` clause of the form, there is a reference to the `ObjectsMappers.pas` unit. This unit adds some method to all the `TDataSet` descendants and we'll use some of them in this project.

The first button converts the current dataset record (the dataset is called `qryPeople` and is owned by a data module called `dm`) in a JSONObject; the code used for this is as follows:

```
Log := dm.qryPeople.AsJSONObjectString;
```

`Log` is a property used as a variable, but it writes its new value to the memo in its setter. So, when you click on the button **1**, you will have this situation in the form:



The memo shows the serialized version of the dataset current record

This is really simple! You don't even need to know how to access the serialization engine; just include the `ObjectsMappers` unit and all your dataset are able to serialize and deserialize themselves.

Button **2** serializes the dataset as a JSON array of JSON Objects, starting from the current position:

```
Log := dm.qryPeople.AsJSONArrayString;
```

Go to first record and click on the button **2**. The memo will show a JSON array like the following one:

```
[{"id":1,"first_name":"Daniele","last_name":"Teti",
   "work_phone_number":"(555) 8765432",
   "mobile_phone_number":"(456) 789456",
      "email":"daniele.teti@gmail.com"},
    {"id":3,"first_name":"Tommy","last_name":"Banner",
      "work_phone_number":"(555) 7894562",
      "mobile_phone_number":null,"email":
      "tommy.banner@gmail.com"},
    {"id":2,"first_name":"John","last_name":"Doe",
      "work_phone_number":"485-965-85",
      "mobile_phone_number":"325-78945621",
      "email":"johndoe@nowhere.org"},
    {"id":4,"first_name":"Jack","last_name":"Rossi",
      "work_phone_number":"(765)
      345677","mobile_phone_number":null,
      "email":"jack.rossi@thered.com"}]
```


www.allitebooks.com

As you can see, the mapper takes care of null fields and serializes them as JSON null.

The third button performs an update on the record using a JSON object:

```
dm.qryPeople.Edit;
dm.qryPeople.LoadFromJSONObjectString(
  Log,
  TArray<String>.Create('id'));
dm.qryPeople.Post;
```

It uses the previously serialized data (contained in the `Log` property) to update the current record. To use it, perform the following steps:

1. Go to the first record.
2. Click on the button **1** (the memo fills with the serialized data as JSON object).
3. Go to the record that you want to update.
4. Click on the button **3**.
5. The record is updated!

You want to update all of the fields but the primary key. So, as the second parameter of `LoadFromJSONObjectString`, we've to pass an array of strings representing the name of the fields that we don't want to update in the dataset. In this case, we don't want to update the `ID` field. So, when just after, call `qryPeople.Post`, the dataset sends an update to the database.

The fourth button is similar, but it is used to create a new record starting from a JSON object. The code is as follows:

```
dm.qryPeople.Append;
dm.qryPeople.LoadFromJSONObjectString(
  Log,
  TArray<String>.Create('id'));
dm.qryPeople.Post;
```

To use the fourth button, perform the following steps:

1. Go to the first record (or another record that you want to clone).
2. Click on the button **1** (the memo fills with the serialized data as a JSON object).
3. Click on the button **4**.
4. A new record is created!

Obviously, you can use any JSON object to create the new record. To prove this, follow these steps:

1. Go to the first record.
2. Click on the button **1** (the memo fills with the serialized data as a JSON object).
3. In the memo, change a JSON property, let's say the `last_name` property. Look for `last_name:"some string"` and change the value to something else.
4. Click on the button **4**.
5. A new record is created with the new value!

The JSON object can arrive from everywhere and can directly be put in your database using this simple `json->dataset` mapping. In the last year, I've used a lot of these techniques in real-world web and mobile applications (the next recipe will focus on a more object-oriented approach compared to this one, which is based on TDataSet).

The fifth button allows us to append a JSON array of JSON objects directly to the dataset:

```
dm.qryPeople.AppendFromJSONArrayString(
  Log,
  TArray<String>.Create('id'));
```

There's more...

Serialization and deserialization are huge topics. All the Internet services, ultimately, depend on some kind of serialization. The average Delphi user is very skilled on dataset and normally tends to rely on some particular functionality present in the data access component suite chosen. However, when the deserializer is not a Delphi program, some problems can arise. The mapper framework resolves this kind of problem in a simple and elegant way (IMHO). As a real example, the JSON format doesn't provide a specific type for dates and times. If you try to blindly serialize `TDate`, `TDateTime`, and `TTime` Delphi data types in JSON (using the underline double data type), you will get numbers that are perfectly valid for another Delphi program but completely useless for JavaScript, Java, .NET, Python, and so on. So the mapper takes care of this and other problems using the standard representation where JSON doesn't provide specific data types. In this case, all `DateTime` data are serialized and deserialized using the ISO format that can be understood by all the libraries and programming languages. Moreover, the mapper is not dependant on the regional settings of the machine, so you can generate a JSON on an English-speaking PC and deserialize it on an Italian speaking machine without problems of decimal separator, date format, currency formatting, and so on.

www.allitebooks.com

Serializing objects to JSON and back using RTTI

When you are using a domain model pattern (and you should do most of the time for non-trivial applications), the entities managed by your program are contained in objects. An object has a state and methods to change its state, just like any actual object in the real world.

Getting ready

As the datasets in the previous recipe is very popular with the needs to serialize an object in a JSON object, send the object somewhere, and then recreate that object as it was before. In this recipe, we'll use the new `TJSON` class present in Delphi XE6 and will extend it with new functionalities.

How to do it...

Let's execute the following steps to serialize objects to JSON:

1. Create a new VCL forms application.
2. Drop four **TButtons** components and a **TMemo** component on the form. Organize the **TButton** component in a single row as a sort of toolbar and align the **TMemo** component to cover the remaining part of the form.
3. Name the **TButton** components as follows:
   - `btnObjToJSON`
   - `btnJSONtoObject`
   - `btnListToJSONArray`
   - `btnJSONArrayToList`
4. Add a new unit to the project, name it `JSON.Serializer.pas`, and fill it with the following code:

```
unit JSON.Serialization;

interface

uses
  REST.JSON, System.Generics.Collections, System.JSON;

type
  TJSONUtils = class(TJSON)
  public
```

```
    class function
      ObjectsToJSONArray<T: class, constructor>(
        AList: TObjectList<T>): TJSONArray;
    class function
      JSONArrayToObjects<T: class, constructor>(
        AJSONArray: TJSONArray): TObjectList<T>;
  end;

implementation

uses
  System.SysUtils;

{ TJSONHelper }

class function TJSONUtils.JSONArrayToObjects<T>(
    AJSONArray: TJSONArray): TObjectList<T>;
var
  I: Integer;
begin
  Result := TObjectList<T>.Create(True);
  try
    for I := 0 to AJSONArray.Size - 1 do
      Result.Add(TJSON.JsonToObject<T>(AJSONArray.Get(I)
        as TJSONObject));
  except
    FreeAndNil(Result);
    raise;
  end;
end;

class function TJSONUtils.ObjectsToJSONArray<T>(
    AList: TObjectList<T>): TJSONArray;
var
  Item: T;
begin
  Result := TJSONArray.Create;
  try
    for Item in AList do
      Result.AddElement(TJSON.ObjectToJsonObject(Item));
  except
    FreeAndNil(Result);
    raise;
  end;
end;

end.
```


www.allitebooks.com

5. Add another unit to the project and name it `PersonU.pas`.
6. In `PersonU.pas`, declare a class as follows and let Delphi autocreate the property setters using *Ctrl* + *Shift* + *C*:

```
type
  TPerson = class
  public
       property ID: Integer;
       property FirstName: String;
       property LastName: String;
       property WorkPhone: String;
       property MobilePhone: String;
       property EMail: String;
     end;
```

7. After *Ctrl* + *Shift* + *C*, save the file and go back to the main form.
8. While on the main form code, hit *Alt* + *F11* and add to the interface uses clause the `JSON.Serializer.pas` unit. Repeat the procedure and add the `PersonU.pas` unit.
9. Now, create a `read/write` property named `Log` in the main form. This property does not have an internal field, but reads and writes its value from the `Memo1.Lines.Text` property, acting like a proxy for it.
10. To have some objects to work with, we require some fake data. So, create a method in the private section of the form called `GetPeople` with the following code:

```
  private
    function GetPeople: TObjectList<TPerson>;
```

11. Hit *Ctrl* + *Shift* + *C* and create the method body with the following code:

```
function TMainForm.GetPeople: TObjectList<TPerson>;
var
  P: TPerson;
begin
  Result := TObjectList<TPerson>.Create(True);
  P := TPerson.Create;
  P.ID := 1;
  P.FirstName := 'Daniele';
  P.LastName := 'Teti';
  P.WorkPhone := '555-4353432';
  P.MobilePhone := '(328) 7894562';
  P.EMail := 'me@danieleteti.it';
  Result.Add(P);

  P := TPerson.Create;
  P.ID := 2;
```

```
    P.FirstName := 'John';
    P.LastName := 'Doe';
    P.WorkPhone := '457-6549875';
    P.EMail := 'john@nowhere.com';
    Result.Add(P);

    P := TPerson.Create;
    P.ID := 3;
    P.FirstName := 'Jane';
    P.LastName := 'Doe';
    P.MobilePhone := '(339) 5487542';
    P.EMail := 'jane@nowhere.com';
    Result.Add(P);
  end;
```

12. Now, create the event handlers for the four buttons using the following code:

```
procedure TMainForm.btnJSONtoObjectClick(Sender: TObject);
var
  JObj: TJSONObject;
  Person: TPerson;
begin
  JObj := TJSONObject.ParseJSONValue(Log) as TJSONObject;
  try
    Person := TJSONUtils.JsonToObject<TPerson>(JObj);
    try
      ShowMessage(Person.FirstName + ' ' +
        Person.LastName);
    finally
      Person.Free;
    end;
  finally
    JObj.Free;
  end;
end;

procedure TMainForm.btnListToJSONArrayClick(Sender:
  TObject);
var
  People: TObjectList<TPerson>;
  JArr: TJSONArray;
begin
  People := GetPeople;
  try
    JArr := TJSONUtils.ObjectsToJSONArray<TPerson>(People);
```


www.allitebooks.com

```
    try
      Log := JArr.ToString;
    finally
      JArr.Free;
    end;
  finally
    People.Free;
  end;
end;

procedure TMainForm.btnObjToJSONClick(Sender: TObject);
var
  People: TObjectList<TPerson>;
  JObj: TJSONObject;
begin
  People := GetPeople;
  try
    JObj := TJSONUtils.ObjectToJsonObject(People[0]);
    try
      Log := JObj.ToString;
    finally
      JObj.Free;
    end;
  finally
    People.Free
  end;
end;

procedure TMainForm.btnJSONArrayToListClick(Sender:
  TObject);
var
  JArr: TJSONArray;
  People: TObjectList<TPerson>;
  Person: TPerson;
  S: String;
begin
  JArr := TJSONObject.ParseJSONValue(Log) as TJSONArray;
  try
    People := TJSONUtils.JSONArrayToObjects<TPerson>(JArr);
    try
      S := '';
      for Person in People do
        S := S + sLineBreak +
          Person.FirstName + ' ' + Person.LastName;
```

```
        finally
          People.Free;
        end;
      finally
        JArr.Free;
      end;
      ShowMessage(S);
    end;
```

13. Hit *F9* and see the application running.

How it works...

The `TJSON` class of Delphi RTL contains two interesting methods.

```
class function ObjectToJsonObject(AObject: TObject): TJSOnObject;
```

The preceding method converts an object in its JSON representation.

```
class function JsonToObject<T: class, constructor>(AJsonObject:
  TJSOnObject): T;
```

The preceding method takes a `JSONObject` and recreates the related object.

Good, but it is not enough. Usually, we deal with a list of objects and an array of JSON objects. This is the reason why the `JSON.Serialization.pas` unit extends the `TJSON` class, because we have to serialize and deserialize a list of objects too.

Here's the public interface of `TJSONUtils`:

```
type
  TJSONUtils = class(TJSON)
  public
    class function ObjectsToJSONArray<T: class, constructor>(
      AList: TObjectList<T>): TJSONArray;
    class function JSONArrayToObjects<T: class, constructor>(
      AJSONArray: TJSONArray): TObjectList<T>;
  end;
```

With these four methods, we will be able to do the following serializations:

- `TObject` **->** `TJSONObject`
- `TJSONObject` **->** `TObject`
- `TObjectList<T>` **->** `TJSONArray` of `TJSONObject`
- `TJSONArray` of `TJSONObject` **->** `TObjectList<T>`

www.allitebooks.com

There's more...

The `TJSON` class allows us to define specific serialization and deserialization strategies based on data types and field names. If you want to serialize a field in a specific way, you can define a `JSONReflect` attribute on that field using the name of the class descended from `TJSONInterceptor`. In the recipe folder, there is a bonus project called `JSONInterceptorSample` that shows how even a stream can be serialized using an interceptor and the `JSONReflect` attribute.

Sending a POST HTTP request encoding parameters

The HTTP protocol supports some types of verbs. A *verb* is a way to ask something to a remote server. Some of these verbs are `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `PATCH`, `TRACE`, and `OPTIONS`. For a detailed description of the HTTP protocol, you can read the related RFCs at the following URLs:

- `http://www.w3.org/Protocols/rfc2616/rfc2616.html`: RFC2616 about HTTP/1.1 protocol
- `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`: RFC2616-specific section about the available verbs in HTTP/1.1 protocol

When you write a URI in the browser address bar and hit return, you are issuing a `GET` request to the remote HTTP server. However, when you have to send form data to the server, usually the HTML form uses the `POST` method. `POST` is designed to allow a uniform method to send a block of data, such as the result of submitting a form, to a data-handling process or to post a message to a bulletin board, newsgroup, mailing list, or similar group of articles. In other words, while `GET` is intended to retrieve a resource from the server, `POST` is intended to transfer data from the client to the server. When sending data to the server, the client should inform it about the type of the content (in case of body data). This information is transferred in a specific request header called content type. If you are sending a JSON, the content type should be `application/json`; if a browser is sending data that a user wrote in an HTML form, the default content type is `application/x-www-form-urlencoded`. The content type is used by the client to inform the server about the type of the content it is sending, and it is used by the server to inform the client about the type of the content it is returning to.

To learn more about the different content types, check `http://en.wikipedia.org/wiki/Internet_media_type`.

In this recipe, we'll learn how to send data to a remote web server using a `POST` method.

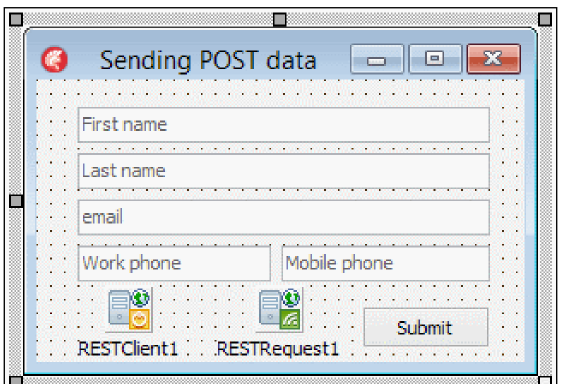
Getting ready

In this recipe, we'll use the web server created in the *Web client JavaScript application with WebBroker on the server* recipe but this time, we're going to create a Delphi client to post data to that server. The data sent will be stored in the database and will be available through the already present web interface.

How to do it...

This recipe is really simple. So, start the WebBroker project created in the *Web client JavaScript application with WebBroker on the server* recipe (run the executable without debug) and follow these instructions:

1. Create a new VCL forms application.
2. On the main form, drop five **TEdit components,** one **TButton** component, one **TRESTClient** component, and one **TRESTRequest** component. Organize the controls as shown in the following screenshot:



The client form used to send POST data to the web server

3. Give meaningful names to the **TEdit** components to avoid confusion in the next phase.
4. Set `RESTClient1.BaseURL` to `http://localhost:8080`.
5. Set the following properties on `RESTRequest1`:
    - `RESTRequest1.Client` = `RESTClient1`
    - `RESTRequest1.Method` = `rmPOST`
    - `RESTRequest1.Resource` = `saveperson`

www.allitebooks.com

6. Double-click on the **TButton** component and add the following code in its `OnClick` event:

```
procedure TMainForm.btnSubmitClick(Sender: TObject);
begin
  RESTRequest1.AddParameter('FIRST_NAME',
    edtFirstName.Text);
  RESTRequest1.AddParameter('LAST_NAME',
    edtLastName.Text);
  RESTRequest1.AddParameter('WORK_PHONE_NUMBER',
    edtWorkPhone.Text);
  RESTRequest1.AddParameter('MOBILE_PHONE_NUMBER',
    edtMobilePhone.Text);
  RESTRequest1.AddParameter('EMAIL', edtEmail.Text);
    RESTRequest1.Execute;
end;
```

7. Run the program, write some data into the edits, and click on the button. That's it! Your data has been saved on the database by the already created WebBroker application. Simple, isn't it?

The TREST* components have been introduced in XE5 and are a fundamental part of a bigger strategic technology from Embarcadero. So, while this recipe could be realized easily also with a simple `TidHTTP` component, it's better to start to use these new components. In the recipe folder, there is also the project that uses the `TidHTTP` component, you choose what to use when.

How it works...

The URL where we've to send the data is `http://localhost:8080/saveperson`. The HTTP request is automatically created and sent to the server by the `TRESTRequest` and `TRESTClient` components. The `TRESTClient` component defines the endpoint for all the requests, while the `TRESTRequest` define details for each different request. In this case, the `BaseURL` property contains the server name with the port (`http://localhost:8080`), while the request has only the `Resource` property set to the second part of the URL to `saveperson`.

We are adding a set of `POST` parameters with their values. Do you remember? `RESTRequest1.Method` is `rmPOST`, so will be created and sent a `POST` request. The parameters' names depend on what the server expects and we have to know the parameters' names to correctly build a request.

As the name says, the TREST* components are mainly to be used with REST services, but can also be used with a normal HTTP service as this recipe showed.

There's more...

The REST client library is very powerful. To have more information about it and to know how to use it when dealing with the RESTful web service, read the following entry in the DocWiki:

`http://docwiki.embarcadero.com/RADStudio/XE6/en/REST_Client_Library`

If you want to see the REST client library in action with different kinds of services, check the RESTDemo sample at the following URL:

`http://docwiki.embarcadero.com/CodeExamples/XE6/en/RESTDemo_Sample`

Implementing a RESTful interface using WebBroker

What's REST? Wikipedia defines it as follows (`http://en.wikipedia.org/wiki/Representational_state_transfer`):

> *Representational state transfer (REST) is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.*
>
> *The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine.*
>
> *The REST architectural style is also applied to the development of web services as an alternative to other distributed-computing specifications such as SOAP.*

So, how to build a RESTful system in Delphi? There are a lot of solutions. However, according to the mentioned definitions, RESTful is not a set of libraries or algorithms; it is an architectural style, and as each style, can be respected at 100 percent, 60 percent, or 30 percent, and so on. There is a sort of scale used to measure how much a system is RESTful or not. This scale was first introduced by Leonard Richardson at the QCon conference, so it is called the **Richardson Maturity Model** (**RMM**). To get all the benefits that a RESTful approach brings, you should tend to a RMM level 3. Be happy, the system we'll develop in this recipe is compliant with RMM level 3.

www.allitebooks.com

Getting ready

Our REST service handles a database table called `PEOPLE`. It provides CRUD methods plus some specific features to paginate the data. Remember that RESTful doesn't mean expose a method to do CRUD on a table but expose a method to handle a resource. A resource can or cannot have a representation on a database table. Moreover, a resource can be also very complex with multiple nested objects. So while a table can be represented as a simple resource, generally a resource is not a mere table but an object graph stored on one, two, or more tables, or not stored at all. This is the HTTP REST interface that we'll implement:

HTTP Verb	URL	Description
`GET`	`/people`	This returns a JSON array containing one JSON object for each record present in the `PEOPLE` table. In each object, the property names are the names of the fields, while the property values are the values of the fields.
`GET`	`/people/($id)` `$id` is an URL parameter	This returns a JSON object representing the specific person which have the `ID = $id` value.
`POST`	`/people`	This creates a new person in the table people. This requires a request body containing the new person to create as JSON object. The request content-type must be `application/json`.
`PUT`	`/people/($id)`	This updates the person with `ID = $id` with the data passed in the request body. This requires a request body containing the new person to update as a JSON object. The request content-type must be `application/json`.
`DELETE`	`/people/($id)`	This deletes the person with `ID = $id`.
`POST`	`/people/searches` `/people/searches?page=[x]`	This returns a JSON array containing JSON objects. It executes a search over the `PEOPLE` table, returning only the record that matches the filter passed as a JSON object in the request body. This requires a JSON object as request body. The parameter is passed as property `"TEXT"` in the request body, for example, `{"TEXT":"ele"}`

This recipe uses the DelphiMVCFramework, a Delphi open source framework based on WebBroker that allows you to create powerful RESTful web services. You can find the project code at `https://code.google.com/p/delphimvcframework/`.

Check out the project using the instruction on the website and put it into a folder on your filesystem. There are no components or controls, only units. Now, you have to configure your IDE to find the DMVCFramework units.

Go to **Tools** | **Options** | **Environment Options** | **Delphi Options** | **Library**. Then, click on the **... on the Library Path** edit and add the following paths one by one (change `C:\DEV\DMVCFramework` with the appropriate path on your machine):

- `C:\DEV\DMVCFramework\sources`
- `C:\DEV\DMVCFramework\lib\delphistompclient`
- `C:\DEV\DMVCFramework\lib\luadelphibinding`
- `C:\DEV\DMVCFramework\lib\iocpdelphiframework\Base`

This recipe uses many DMVCFramework features and could be a little confusing if you don't know the basics of REST and DMVCFramework. If so, please read the following documentations before going ahead:

- *Building Web Services the REST Way* at `http://www.xfront.com/REST-Web-Services.html`
- *RESTful Web services: The basics* at `https://www.ibm.com/developerworks/webservices/library/ws-restful/`
- *DelphiMVCFramework* `Documentation at https://delphimvcframework.googlecode.com/svn/trunk/docs/ITDevCON%202013%20-%20Introduction%20to%20DelphiMVCFramework.pdf`
- DelphiMVCFramework Samples available in the `\Samples` folder in the project root folder

From this point onwards, I'll not repeat concepts and information already explained in the mentioned articles. So, read them with care.

How to do it...

1. Create a new Delphi project by going to **Delphi Project** | **Web Broker** | **Web Server Application**.
2. Now, the wizard asks you what type of web server application you want to create. This demo will be built as a console application. However, you can take advantage of the flexibility of WebBroker and add another type of application, for instance, an ISAPI DLL or a Windows Service. At this point, select **Stand-alone console application** and click on **Next**.
3. The wizard proposes a TCP port where the service will listen. Click on **Test port**; if the test port succeeded, use it. Otherwise, change the port until the test passes. In this recipe, the port 8080 is used.
4. Click on **Finish**.

www.allitebooks.com

5. Save all. Name the project `PeopleManager.dproj` and the web module `WebModuleU.pas`.
6. We start from the business objects classes. This web service will manage people, so let's create a new unit and declare the following class:

```
TPerson = class
public
  property ID: Integer;
  property FIRST_NAME: String;
  property LAST_NAME: String;
  property WORK_PHONE_NUMBER: String;
  property MOBILE_PHONE_NUMBER: String;
  property EMAIL: String;
end;
```

7. Hit *Ctrl* + *Shift* + *C* to autocomplete the declaration, and then save the file as `PersonBO.pas`. Note that in projects where you have a lot of different types of classes (business objects, controllers, and data modules), it will be good to organize the units in different folders. So, I saved the `PersonBo.pas` file in a folder named `BusinessObjects`. Feel free to do it as you wish.
8. Now, it is time to create a `DMVCFramework` controller. This is the class where there will be all the code to handle the HTTP requests and responses. Here, there should not be any business logic code.
9. Create a new unit, name it `PeopleControllerU.pas`, and save it into the `Controllers` folder.
10. Fill the `PeopleControllerU.pas` unit with the following code:

```
unit PeopleControllerU;

interface

uses MVCFramework, PeopleModuleU;

type
  [MVCPath('/people')]
  TPeopleController = class(TMVCController)
  private
    FPeopleModule: TPeopleModule;
  protected
    procedure OnAfterAction(Context: TWebContext;
      const AActionNAme: string); override;
    procedure OnBeforeAction(Context: TWebContext;
      const AActionNAme: string;
      var Handled: Boolean); override;
  public
```

```
    [MVCPath]
    [MVCHTTPMethod([httpGET])]
    procedure GetPeople(CTX: TWebContext);

    [MVCPath('/($id)')]
    [MVCHTTPMethod([httpGET])]
    procedure GetPersonByID(CTX: TWebContext);

    [MVCPath]
    [MVCHTTPMethod([httpPOST])]
    [MVCConsumes('application/json')]
    procedure CreatePerson(CTX: TWebContext);

    [MVCPath('/($id)')]
    [MVCHTTPMethod([httpPUT])]
    [MVCConsumes('application/json')]
    procedure UpdatePerson(CTX: TWebContext);

    [MVCPath('/($id)')]
    [MVCHTTPMethod([httpDELETE])]
    procedure DeletePerson(CTX: TWebContext);

    [MVCPath('/searches')]
    [MVCHTTPMethod([httpPOST])]
    [MVCConsumes('application/json')]
    procedure SearchPeople(CTX: TWebContext);
  end;

implementation

uses
  PersonBO, SysUtils, System.JSON, ObjectsMappers, System.Math;

procedure TPeopleController.CreatePerson(CTX: TWebContext);
var
  Person: TPerson;
begin
  Person := CTX.Request.BodyAs<TPerson>;
  try
    FPeopleModule.CreatePerson(Person);
    CTX.Response.Location := '/people/' +
      Person.ID.ToString;
    Render(201, 'Person created');
  finally
```

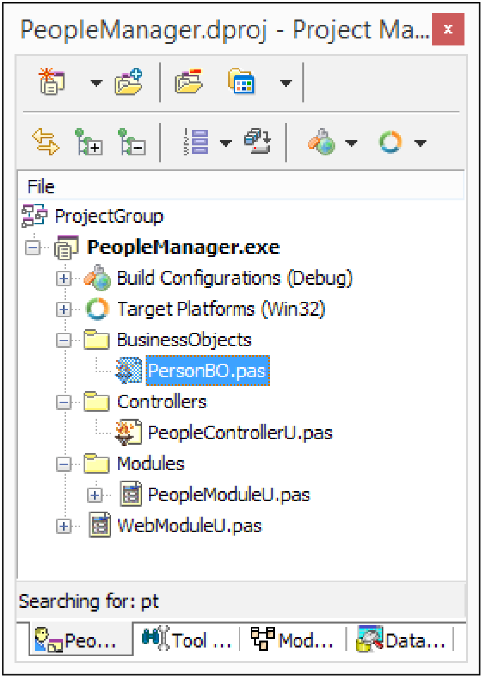

www.allitebooks.com

```
    Person.Free;
  end;
end;

procedure TPeopleController.UpdatePerson(CTX: TWebContext);
var
  Person: TPerson;
begin
  Person := CTX.Request.BodyAs<TPerson>;
  try
    Person.ID := CTX.Request.ParamsAsInteger['id'];
    FPeopleModule.UpdatePerson(Person);
    Render(200, 'Person updated');
  finally
    Person.Free;
  end;
end;

procedure TPeopleController.DeletePerson(CTX: TWebContext);
begin
  FPeopleModule.DeletePerson(
    CTX.Request.ParamsAsInteger['id']);
  Render(204, 'Person deleted');
end;

procedure TPeopleController.GetPersonByID(CTX: TWebContext);
var
  Person: TPerson;
begin
  Person := FPeopleModule.GetPersonByID(
    CTX.Request.ParamsAsInteger['id']);
  if Assigned(Person) then
    Render(Person)
  else
    Render(404, 'Person not found');
end;

procedure TPeopleController.GetPeople(CTX: TWebContext);
begin
  Render<TPerson>(FPeopleModule.GetPeople);
end;

procedure TPeopleController.OnAfterAction(Context:
  TWebContext;
```

```
  const AActionNAme: string);
begin
  inherited;
  FPeopleModule.Free;
end;

procedure TPeopleController.OnBeforeAction(Context:
  TWebContext;
  const AActionNAme: string;
  var Handled: Boolean);
begin
  inherited;
  FPeopleModule := TPeopleModule.Create(nil);
end;

procedure TPeopleController.SearchPeople(CTX: TWebContext);
var
  Filters: TJSONObject;
  SearchText, PageParam: string;
  CurrPage: Integer;
begin
  Filters := CTX.Request.BodyAsJSONObject;
  if not Assigned(Filters) then
    raise Exception.Create('Invalid search parameters');
  SearchText := Mapper.GetStringDef(Filters, 'TEXT');
  if (not TryStrToInt(CTX.Request.Params['page'], CurrPage))
    or (CurrPage < 1) then
    CurrPage := 1;
  Render<TPerson>(FPeopleModule.FindPeople(SearchText,
    CurrPage));
  CTX.Response.CustomHeaders.Values[
    'dmvc-next-people-page'] :=
    Format('/people/searches?page=%d', [CurrPage + 1]);
  if CurrPage > 1 then
    CTX.Response.CustomHeaders.Values[
      'dmvc-prev-people-page'] :=
      Format('/people/searches?page=%d', [CurrPage - 1]);
end;

end.
```


www.allitebooks.com

11. Quite long, but all our RESTful interface is implemented in this unit. Now, we've to write the part that actually accesses the database. In this recipe, we'll use a simple design pattern called **Table Data Gateway** (**TDG**). TDG was defined for the first time by Martin Fowler in his fundamental and highly recommended book *Patterns of Enterprise Application Architecture*, *Addison-Wesley Professional* (`http://www.amazon.com/gp/product/0321127420`). TDG is defined as follows: an object that acts as a Gateway to a database table. One instance handles all the rows in the table (`http://martinfowler.com/eaaCatalog/tableDataGateway.html`).
12. Let's create our TDG using a data module. Add a new data module, name it `PeopleModule`, and save it into the `Modules` folder as `PeopleModuleU.pas`.
13. Now your Project Manager should looks like the following screenshot:



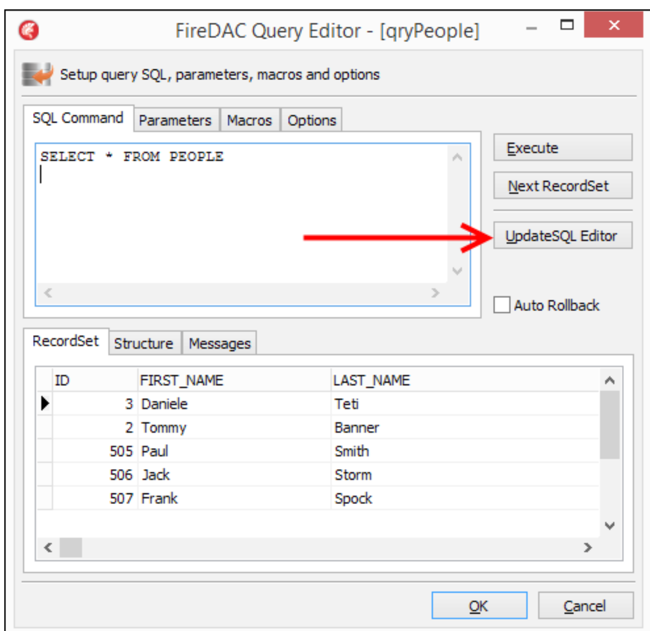
The Project Manager

14. Now, drop the components on the data module and link each other (this is an extract of the `dfm` file):

```
object Conn: TFDConnection
  Params.Strings = (
    'Database=C:\Delphi
      Cookbook\BOOK\Chapter05\DATA\SAMPLES.IB'
    'User_Name=sysdba'
    'Password=masterkey'
    'DriverID=IB')
```

```
    ConnectedStoredUsage = [auDesignTime]
    Connected = True
    LoginPrompt = False
  end
  object qryPeople: TFDQuery
    Connection = Conn
    UpdateObject = updPeople
  end
  object updPeople: TFDUpdateSQL
    Connection = Conn
  end
  object FDPhysIBDriverLink1: TFDPhysIBDriverLink
  end
end
```

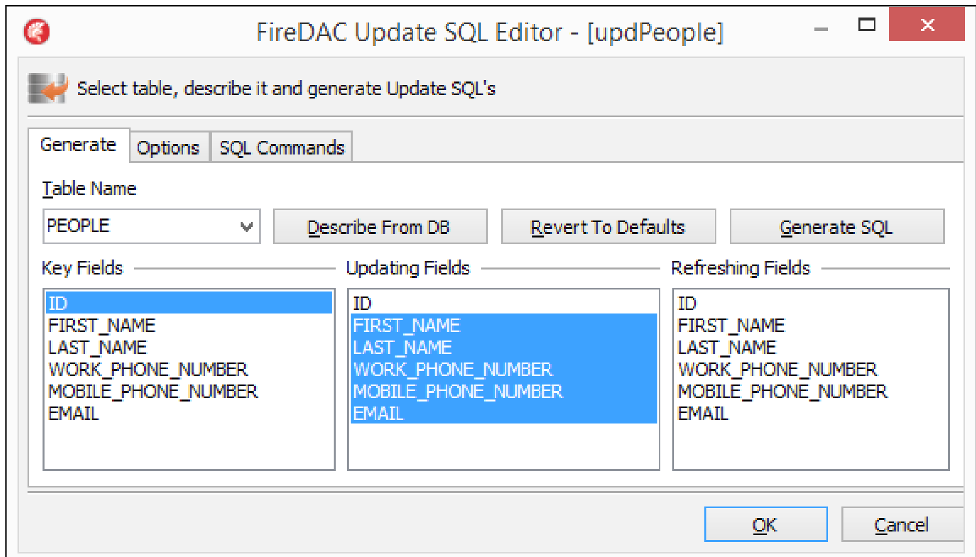
15. Change the `FDConnection` parameter according to your machine.
16. Now, we've to configure some data access stuff.
17. Double-click on `qryPeople` and the component editor shows up. Write the query `SELECT * FROM PEOPLE`, and click on **Execute**. Hold the window open. This will be the query used to generate all the CRUD statements.
18. If you have correctly connected `qryPeople.UpdateObject` to `updPeople`, you should see the **UpdateSQL Editor** button on the right side of the component editor form.



The qryPeople component editor showing the SQL and the button to configure the TFDUpdateSQL linked to the qryPeople

www.allitebooks.com

19. Click on the **UpdateSQL Editor** button and you will get another component editor. This time it is related to the **updPeople** component.
20. Select fields as shown in the following screenshot, click on **Generate SQL,** and then click on **OK**. Now your `updPeople` component has been configured with all the SQL statements required to correctly update the `PEOPLE` table.



The updPeople component editor used to configure the INSERT, UPDATE and DELETE SQL statements

21. Now, we've to create the methods used to create, retrieve, update, and delete records. Go to the `PeopleModuleU.pas` code view. Declare the following method in the public section of the class:

```
public
  procedure CreatePerson(APerson: TPerson);
  procedure DeletePerson(AID: Integer);
  procedure UpdatePerson(APerson: TPerson);
  function GetPersonByID(AID: Integer): TPerson;
  function FindPeople(ASearchText: String;
    APage: Integer): TObjectList<TPerson>;
  function GetPeople: TObjectList<TPerson>;
end;
```

22. Hit *Ctrl* + *Shift* + *C* to autogenerate methods bodies and fill them with the following code:

```
procedure TPeopleModule.CreatePerson(APerson: TPerson);
var
  InsCommand: TFDCustomCommand;
```

```
begin
  InsCommand := updPeople.Commands[arInsert];
  Mapper.ObjectToFDParameters(InsCommand.Params, APerson,
    'NEW_');
  InsCommand.Execute;
  APerson.ID := Conn.GetLastAutoGenValue('gen_people_id');
end;

procedure TPeopleModule.DeletePerson(AID: Integer);
var
  DelCommand: TFDCustomCommand;
begin
  DelCommand := updPeople.Commands[arDelete];
  DelCommand.ParamByName('OLD_ID').AsInteger := AID;
  DelCommand.Execute;
end;

function TPeopleModule.FindPeople(ASearchText: String;
  APage: Integer): TObjectList<TPerson>;
var
  StartRec, EndRec: Integer;
begin
  Dec(APage); // page 0 => 0,9, page 1 => 10,19, page 3 =>
    20,29
  StartRec := (10 * APage);
  EndRec := StartRec + 10 - 1;
  qryPeople.Open('SELECT * FROM PEOPLE WHERE ' +
    'FIRST_NAME CONTAINING :SEARCH_TEXT_1 OR ' +
    'LAST_NAME CONTAINING :SEARCH_TEXT_2 OR ' +
    'EMAIL CONTAINING :SEARCH_TEXT_3 ' +
    'ORDER BY LAST_NAME, FIRST_NAME ' +
    Format('ROWS %d TO %d', [StartRec, EndRec]),
    [ASearchText, ASearchText, ASearchText]);
  Result := qryPeople.AsObjectList<TPerson>;
end;

function TPeopleModule.GetPersonByID(AID: Integer): TPerson;
begin
  qryPeople.Open('SELECT * FROM PEOPLE WHERE ID = :ID',
    [AID]);
  Result := qryPeople.AsObject<TPerson>;
end;

function TPeopleModule.GetPeople: TObjectList<TPerson>;
```


www.allitebooks.com

```
begin
  qryPeople.Open;
  Result := qryPeople.AsObjectList<TPerson>;
end;

procedure TPeopleModule.UpdatePerson(APerson: TPerson);
var
  UpdCommand: TFDCustomCommand;
begin
  UpdCommand := updPeople.Commands[arUpdate];
  Mapper.ObjectToFDParameters(
    UpdCommand.Params,
    APerson, 'NEW_');
  UpdCommand.ParamByName('OLD_ID').AsInteger := APerson.ID;
  UpdCommand.Execute;
end;
```

23. These methods will be called by the controller capping data retrieved by the HTTP request. As you can see, the CRUD methods do not have reference to the HTTP environment, JSON object, or whatever is related to the particular environment. These methods, and the whole class itself, can be used everywhere—even in a classic client/server application. Remember that the dependencies between the classes should be reduced as much as you can. More on this will be covered in the *How it works...* section of this recipe.
24. Add the `ObjectsMappers` unit in the implementation `uses` clause of the `TPersonModule`.
25. Just one more thing to do in the `TPersonModule`. Create the `OnBeforeConnect` event handler on the `TFDConnection` and write the following code. Then, adapt it to point to the correct database path on your system. The code is as follows:

```
procedure TPeopleModule.ConnBeforeConnect(Sender: TObject);
begin
  inherited;
  Conn.Params.Values['Database'] := '..\..\..\..\DATA\SAMPLES.IB';
end;
```

26. We're about to finish. Go back to the `WebModuleU.pas` file and create the `OnCreate` event handler. Here, we've to configure the DelphiMVCFramework starting point. It is really simple, just two lines of code:

```
procedure TwmMain.WebModuleCreate(Sender: TObject);
begin
  MVC := TMVCEngine.Create(Self);
  MVC.AddController(TPeopleController);
end;
```

27. The MVC variable must be declared in the `private` section of the class, and you have to add the `PeopleControllerU` unit in the implementation uses clause.
28. Now your project should compile. If not, check the dependencies between all the units.
29. On running the project, you get a sad console window that informs you that an HTTP server is running on port 8080. Launch a browser (Google Chrome or Mozilla Firefox if possible) and request the following URL: `http://localhost:8080/people`.
30. Your browser should show all the data available in the `PEOPLE` table as JSON array of JSON objects:

localhost:8080/people

localhost:8080/people

[{"ID":3,"FIRST_NAME":"Daniele","LAST_NAME":"Teti","WORK_PHONE_NUMBER":"(555) 7894562 ","EMAIL":"tommy.banner@gmail.com"},
{"ID":2,"FIRST_NAME":"Tommy","LAST_NAME":"Banner","WORK_PHONE_NUMBER":"UPDATED","MOBILE_PHONE_NUMBER":"UPDATED","EMAIL":"daniele.teti@gmail.com"},
{"ID":505,"FIRST_NAME":"Paul","LAST_NAME":"Smith","WORK_PHONE_NUMBER":"(555) 8765432","MOBILE_PHONE_NUMBER":"(456) 789456","EMAIL":"paul.smith@gmail.com"},
{"ID":506,"FIRST_NAME":"Jack","LAST_NAME":"Storm","WORK_PHONE_NUMBER":"(555) 1234578","MOBILE_PHONE_NUMBER":"(456) 98765432","EMAIL":"daniele.teti@gmail.com"},
{"ID":507,"FIRST_NAME":"Frank","LAST_NAME":"Spock","WORK_PHONE_NUMBER":"(555) 1234578","MOBILE_PHONE_NUMBER":"(456) 98765432","EMAIL":"daniele.teti@gmail.com"}]

The JSON array of JSON object returned by the http call from the browser

31. If you want to try something different, get a valid person ID from the list of the `PEOPLE` (search for the ID: `<some number>`) and append it to the URL after a slash. This should be the effect:

The JSON object representing a single person returned by the HTTP call from the browser

www.allitebooks.com

How it works...

Wow! This recipe is very long! However, it summarizes all the concepts already seen in the previous recipes, so it's worth it.

The application is organized in three layers:

- Controller (`TPeopleController`)
  - This takes care of all the machinery required to deserialize the JSON data into Delphi objects
  - This coordinates the job with the Table Module
- TDG (`TPeopleModule`)
  - This handles all the persistence requirements
  - This gets objects and persists them
  - This retrieves datasets and convert them to objects
- Business objects (`TPerson`)
  - This implements all the business logic required by the domain problem. In this sample, we don't have business logic; however, if present, it should be inside the `TPerson` class.

When an HTTP request arrives to the server, the DMVCFramework router starts to find a suitable controller using the `MVCPath` attributes defined on all its controllers.

When a matching controller and action is found, the request and response object are packed in a `TWebContext` object and passed to the selected action. Here, we can read information from the request and build the response accordingly.

All the action methods perform the following tasks:

- Read information from the HTTP request
- Invoke some methods on the `TPersonModule` instance
- Build the response for the client

Let's have a look at the following action used to create a new `PERSON` object:

```
[MVCPath]
[MVCHTTPMethod([httpPOST])]
[MVCConsumes('application/json')]
procedure CreatePerson(CTX: TWebContext);

. . .
```

```
procedure TPeopleController.CreatePerson(CTX: TWebContext);
var
  Person: TPerson;
begin
  //read information from the request
  Person := CTX.Request.BodyAs<TPerson>;
  try

    //invoke some methods on the TPeopleModule instance
    FPeopleModule.CreatePerson(Person);

    //build the response for the client
    CTX.Response.Location := '/people/' +
      Person.ID.ToString;
    Render(201, 'Person created');

  finally
    Person.Free;
  end;
end;
```

What's that `CTX.Response.Location` line for? One of the RESTful features is the use of hypermedia controls. The point of hypermedia controls is that they tell us what we can do next and the URI of the resource we have to manipulate to do it. Instead of having to know where to get our newly created person, the hypermedia controls in the response tell us where to get the new person.

Another interesting action is mapped to `POST /people/searches`. The following is the code:

```
 [MVCPath('/searches')]
 [MVCHTTPMethod([httpPOST])]
 [MVCConsumes('application/json')]
procedure SearchPeople(CTX: TWebContext);

. . .

procedure TPeopleController.SearchPeople(CTX: TWebContext);
var
  Filters: TJSONObject;
  SearchText, PageParam: string;
  CurrPage: Integer;
begin
  //read informations from the requests
  Filters := CTX.Request.BodyAsJSONObject;
```


www.allitebooks.com

```
    if not Assigned(Filters) then
      raise Exception.Create('Invalid search parameters');
    SearchText := Mapper.GetStringDef(Filters, 'TEXT');
    if (not TryStrToInt(CTX.Request.Params['page'], CurrPage))
      or (CurrPage < 1) then
      CurrPage := 1;

    //call some method on the TPeopleModule
    Render<TPerson>(FPeopleModule.FindPeople(SearchText, CurrPage));

    //prepare the response (also if render has been already called)
    CTX.Response.CustomHeaders.Values['dmvc-next-people-page'] :=
      Format('/people/searches?page=%d', [CurrPage + 1]);
    if CurrPage > 1 then
      CTX.Response.CustomHeaders.Values['dmvc-prev-people-page'] :=
        Format('/people/searches?page=%d', [CurrPage - 1]);
  end;
```

This action is a bit longer, but the three steps are still clearly defined. This action executes a search on the people table using a pagination mechanism. The URL to get the next and the previous pages is returned along with the response in the `dmvc-next-people-page` and `dmvc-prev-people-page` headers. So, clients don't have to know which kind of call they have to do to get the second page; they can simply navigate through the returned information.

Now, this is a last note about the `TPersonModule` that heavily uses the `DataSet` helpers introduced in the *Serializing a dataset to JSON and back* recipe. Look at the following code used to get a `person` object by ID:

```
function TPeopleModule.GetPersonByID(AID: Integer): TPerson;
begin
  qryPeople.Open('SELECT * FROM PEOPLE WHERE ID = :ID', [AID]);
  //uses the dataset helper to convert a record to an object
  Result := qryPeople.AsObject<TPerson>;
end;
```

This could not be simpler! Also, the method to create a new person is really simple when using some of the Mapper methods:

```
procedure TPeopleModule.CreatePerson(APerson: TPerson);
var
  InsCommand: TFDCustomCommand;
begin
  //gets the Insert statement contained in the TFDUpdateSQL
  InsCommand := updPeople.Commands[arInsert];

  //Maps the object properties to the command parameters
```

```
    Mapper.ObjectToFDParameters(InsCommand.Params, APerson, 'NEW_');

    //execute the statement
    InsCommand.Execute;

    //retrieve the last assigned ID
    APerson.ID := Conn.GetLastAutoGenValue('gen_people_id');
  end;
```

There's more...

What a huge topic in this recipe! To test the RESTful service that you will develop from now on, you can use the `RESTDebugger.exe` program provided since Delphi XE5 (in the `bin` folder), or the free POSTMan Chrome extension (`http://alturl.com/6ycza`). These tools allow you to send all the HTTP VERB requests while the browser, using only the address bar, can only issue the `GET` request.

Remember that if you don't know well the fundamental principle of REST, you could break all the benefits. Don't be tempted to put verbs on the URL, such as `http://server.com/people/create` or `http://server.com/people/get`. This is not REST. This is a sort of remote procedure call. It is not necessarily bad, but it is another thing—it's not REST. Also, be coherent with the HTTP VERB used. All the HTTP methods must be idempotent but `POST` and `PATCH`. So if your request is executed once, twice, or 1,000 times, the system will not change further.

Read this article for a good overview on idempotence in HTTP:
`http://restcookbook.com/HTTP%20Methods/idempotency/`

Controlling remote applications using UDP

What's UDP? UDP is a connectionless protocol used by everyone every day, but it seems that not too many people know it. However, it can really be useful to solve particular network problems. Like TCP, UDP works at transport layer in the TCP/IP model, but they have very different uses.

UDP

Compared to TCP, UDP is a simpler message-based, connectionless protocol. Connectionless protocols do not set up a dedicated end-to-end connection; instead, communication is achieved by transmitting information in one direction from source to destination without verifying the readiness or state of the receiver. However, one primary benefit of UDP over TCP is the application to the **voice-over-Internet protocol** (**VoIP**) where latency and jitter are the primary concerns. It is assumed in VoIP UDP that the end users provide any necessary real-time confirmation that the message has been received.

www.allitebooks.com

Here are some features of UDP as exposed by Wikipedia (`http://en.wikipedia.org/wiki/User_Datagram_Protocol`):

- **Unreliable**: When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission, or timeout.
- **Not ordered**: If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.
- **Lightweight**: There is no ordering of messages, no tracking connections, and so on. It is a small transport layer designed on top of IP.
- **Datagrams**: Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, which means a read operation at the receiver socket will yield an entire message as it was originally sent.
- **No congestion control**: UDP itself does not avoid congestion, and it's possible for high bandwidth applications to trigger congestion collapse, unless they implement congestion control measures at the application level.

Getting ready

In this recipe, we'll use UDP to autoconfigure an application in a LAN. Let's say you have some classic client/server applications (however, the same approach is valid for any type of applications) in a LAN or a big LAN. Every application uses a database on a specific machine and uses internal web services. Usually, in this scenario, you have some kind of configuration stored somewhere on the client PC that is read at the startup. However, what if the database IP change because something is changed on the network? Or, what if some part of the configuration could be subject to change for some external reasons? If the change is only about the IP, a simple internal DNS does the job. However, what about a port change? Furthermore, what if this changes something else? Ok, I think that you got the point; you have to change the configuration on all the machines (if you don't have some type of software distribution, this could be a daunting and boring task). Let's think about a well-known network service, the DHCP (`http://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol`).

When a machine with dynamic IP configuration starts, the operating system sends a broadcast on the network to ask for an IP. It doesn't know who will send the IP, and it doesn't know if someone can reply with an IP. It doesn't know anything! In this situation, the DHCP server replies to the broadcast with the assigned IP for that machine. The machine gets its IP and can join the network. This is the same approach that we'll use in this recipe. We have a database application that doesn't know where the database it should connect to is. So, this sends a broadcast on the network saying: "Hey, I'm application X, which database should I connect to?".

On the network, there is another program called `ConfigDispatcher` that replies to the broadcast with the correct connection information for that specific application. So, the client reads the `ConfigDispatcher` reply and can happily connect to the correct database. No config files, no default server, no hardcoded names, but a simple autoconfiguration. Wow, this is the power of UDP.

How to do it...

This recipe is composed of two projects: the `ConfigDispatcher` and the real application. Let's start with the `ConfigDispatcher`.

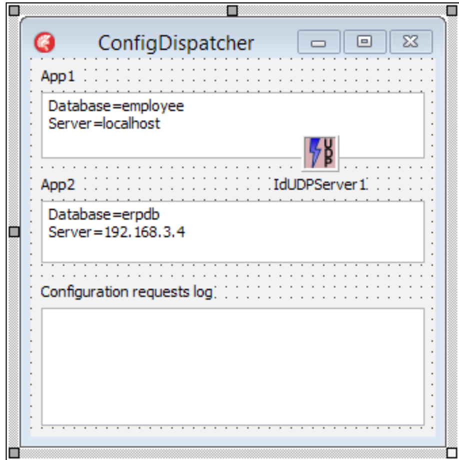
1. Create a new VCL forms application and save it as `ConfigDispatcher`.
2. Drop three **TMemo** components on the form and name them `MemoLog`, `MemoConfigApp1`, `MemoConfigApp2`.
3. In the `MemoConfigApp1.Lines` property, add the following lines:

   ```
   Database=employee
   Server=localhost
   ```

4. In the `MemoConfigApp2.Lines` property, add the following lines:

   ```
   Database=erpdb
   Server=192.168.3.4
   ```

5. In this recipe, we'll use only the first configuration. However, for the sake of completeness, there is also a second (fake) configuration available that will remain unused.
6. Drop a `TidUDPServer` component then drop three `TLabel` and arrange them in the form as show in the following screenshot:



The ConfigDispatcher main form

www.allitebooks.com

7. Now, set the `idUDPServer1` properties as follows (this is the relevant part of the form `dfm`, so it should not be difficult to read):

```
object IdUDPServer1: TIdUDPServer
  BroadcastEnabled = True
  DefaultPort = 8888
  Active = True
end
```

8. Now, create the `OnUDPRead` event handler for the `idUDPServer1` component and fill it with the following code:

```
procedure TMainForm.IdUDPServer1UDPRead(
  AThread: TIdUDPListenerThread;
  const AData: TIdBytes;
  ABinding: TIdSocketHandle);
var
  ClientCommand, ClientConfig: string;
  CommandPieces: TArray<string>;
begin
  ClientCommand := BytesToString(AData);
  MemoLog.Lines.Add(ClientCommand);
  CommandPieces := ClientCommand.Split(['#']);
  if (Length(CommandPieces) = 2) and
    (CommandPieces[0] = 'GETCONFIG') then
  begin
    if CommandPieces[1] = 'APP001' then
    begin
      ClientConfig := MemoConfigApp1.Lines.Text;
    end;
    if CommandPieces[1] = 'APP002' then
    begin
      ClientConfig := MemoConfigApp2.Lines.Text;
    end;
    ABinding.Broadcast(ToBytes(ClientConfig),
      9999, ABinding.PeerIP);
  end;
end;
```

9. At this time, the project doesn't compile. Add the idGlobal unit in the uses clause interface section and it should.

The `ConfigDispatcher` is finished. Let's start the `ClientDBApplication`.

1. Add to the project group a new VCL forms application (by navigating to **ProjectGroup** | **Add New Project** | **VCL Forms Application**).
2. Save the new project as `ClientDBApplication` and give a meaningful name to the form.
3. Drop the following components on the main form and set their properties as follows:

```
object FDConnection1: TFDConnection
  Params.Strings = (
    'User_Name=sysdba'
    'Password=masterkey'
    'Protocol=TCPIP'
    'DriverID=IB')
  ConnectedStoredUsage = [auDesignTime]
  LoginPrompt = False
end

object FDQuery1: TFDQuery
  Connection = FDConnection1
  SQL.Strings = ('select * from customer')
end

object DataSource1: TDataSource
  DataSet = FDQuery1
end

object FDPhysIBDriverLink1: TFDPhysIBDriverLink
end

object FDGUIxWaitCursor1: TFDGUIxWaitCursor
end

object Timer1: TTimer
  Interval = 3000
end

object IdUDPServer1: TIdUDPServer
  DefaultPort = 9999
  Active = True
end
```

4. Drop **TDBGrid** and **TDBNavigator** components and hook them to `DataSource1`.

www.allitebooks.com

5. Now, if you try to activate the `FDQuery1`, you should see the query data in the grid.
6. Double-click on the `Timer1` and fill in the `OnTimer` event with the following code:

```
procedure TMainFormClient.Timer1Timer(Sender: TObject);
begin
  Caption := 'Waiting for configuration...';
  IdUDPServer1.Broadcast(
    ToBytes('GETCONFIG#APP001'), 8888);
end;
```

7. Include the `idGlobal` unit in the `uses interface` clause.
8. Now, create the `OnUDPRead` event handler for the `idUDPServer1` component and fill it with the following code:

```
procedure TMainFormClient.IdUDPServer1UDPRead(
  AThread: TIdUDPListenerThread; const AData: TIdBytes;
  ABinding: TIdSocketHandle);
var
  ServerConfig: TStringList;
  i: Integer;
begin
  Timer1.Enabled := False;
  try
    Caption := 'Configuration OK...';
    ServerConfig := TStringList.Create;
    try
      ServerConfig.Text := BytesToString(AData);
      for i := 0 to ServerConfig.Count - 1 do
      begin
        FDConnection1.Params.Values[ServerConfig.Names[i]]
          :=
          ServerConfig.ValueFromIndex[i];
      end;
    finally
      ServerConfig.Free;
    end;
    FDConnection1.Open;
    FDQuery1.Open;
    Caption := 'Connected';
  except
    Caption := 'Wrong configuration or cannot connect';
    Timer1.Enabled := true;
  end;
end;
```

9. Now, check that the InterBase service is started on your machine. If it's not started, start it.
10. Run the `ConfigDispatcher` without debugging and then run the `ClientDBApplication`. After 3 seconds, you should see the data in the grid. The configuration has been requested with a broadcast to the `ConfigDispatcher`, then has been parsed, understood, and used to connect to the database.
11. You can try to start the `ClientDBApplication` first, wait for 6 seconds, and then start the `ConfigDispatcher`. It just works.

How it works...

This is a long recipe but the behavior is really simple. The `ConfigDispatcher` uses two memos to maintain the strings to send to the client that request a specific configuration.

When a client requests a configuration, the server receives a command string similar to the following:

```
GETCONFIG#APP001
```

It parses the command and replies to the client with the contents of one of the memos. For `APP001`, it sends the `MemoConfigApp1` content, while for `APP002` it sends the contents of `MemoConfigApp2`. That's it, the `ConfigDispatcher` job is finished.

The client is simple too. When it starts, it waits for 3 seconds, gets configured in the timer, and asks for a configuration. If some data arrive on the UDP server, the `UDPRead` event handler is called. The code disables the timer, reads the data sent by the `ConfigDiaspatcher`, and tries to use it to configure it to database connection. If the configuration is correct, the `ClientDBApplication` connects to its database. Otherwise, the timer is re-enabled and after 3 seconds, another configuration request is broadcasted and the cycle goes on until the client is able to connect.

You can see that the application talks to each other without any kind of predefined knowledge or configuration. This is the power of UDP!

There's more...

Network programming and network protocols are a really large topic. As a software developer, you have to be aware—if not yet—of the possibilities that the standard networking infrastructure offers to you.

The UDP protocol allows you to create *strange* applications that find and talk to each other using broadcasts. You could even create a complex application protocol based on UDP to remotely control some running applications. In the chapter devoted to mobile programming, there is another sample of the UDP power.

www.allitebooks.com

Here are some more Delphi samples of UDP programming:

- *Chat application with Delphi source* at `http://delphi.about.com/library/weekly/aa101105a.htm`
- A fun utility to invoke fake and harmless BSODs on colleagues' machines at `http://www.atozed.com/indy/demos/10/index.en.aspx`

Using App Tethering to create a companion app

App Tethering is one of the features introduced in RAD Studio XE6. App Tethering allows you to connect applications to exchange in a so-called *serverless* mode. In other words, this gives to your applications the ability to interact with other applications running either on the same machine or on a remote machine without using a server because the applications communicate directly with each other.

Currently, App Tethering works between applications running on the same LAN, but their features do not depend on a specific transport or protocol. New protocols can be implemented using the app tethering API.

To enable an application to use app tethering, only two components are required:

- `TTetheringManager`: This is used to discover other applications that are using app tethering on the same LAN, or even on the same machine or devices
- `TTetheringAppProfile`: This is used to define the actions and data that your application shares with other applications previously paired using the `TTetheringManager`

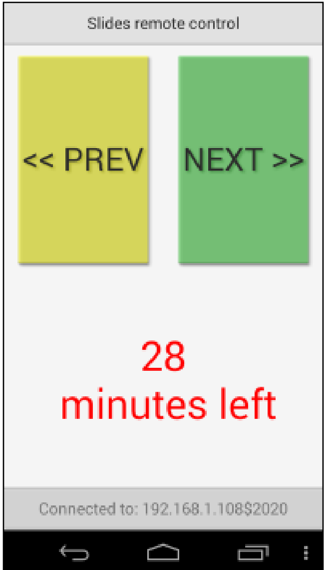
The App Tethering technology roughly follows the Bluetooth model, where there are a set of Bluetooth devices that are able to interact with each other and each application exposes a set of profiles usable by the other applications.

One of the strengths of this technology is that it is completely independent of the platform on which the resultant application runs. You can use App Tethering to connect a VCL application to a mobile app running on Android or iOS, or between a FireMonkey MacOSX application and an iOS app, or even a VCL Windows service to a FireMonkey desktop application. I think you got the point; you can use App Tethering to create an application network that is able to make your applications more usable.

App Tethering is designed to develop so-called companion apps. What's a companion app? Well, a companion app is an app designed to make another application more usable. Let's say you developed a media center running on an Android TV or on a PC. You can play videos and music, but how to control the player while you are on the sofa? You require a remote controller! Using App Tethering, you can create a companion app running on your phone able to control the media center to play and stop a video, to go forward, or to go to the next video. The remote controller is a typical companion app of your media center.

Getting ready

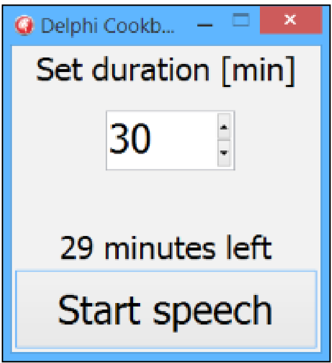
There are some nice examples of app tethering on the Internet and some others have been provided by Embarcadero. In this recipe, we'll talk about a completely new app. We'll develop a "presenter assistant" (I've just coined this term!). What's a presenter assistant? Well, during my trainings, while I'm talking at conferences or while I present the new version of Delphi to the Italian community, I use a lot of slides. So, I run my MS PowerPoint presentation (or OpenOffice.org Impress) and talk over the slides about the new Delphi features. For the past few years, I have been using a presenter pointer that allows me to go to the next slides easily without going back to the PC and pressing the Space bar key (because I walk a lot during the presentation, usually I'm too far from the PC to go back at each slide). A "presenter assistant" is a small device with two buttons: **NEXT** and **PREVIOUS**. However, I love so much to talk about programming (and Delphi) that often I run out of time. Here's the idea for this recipe: a Presenter Assistant app running on my Android smartphone that allows me to go to the next slide, to the previous slide, and also to display how many minutes I have before the end of the speech. Here's the Presenter Assistant app while doing its job:



The Presenter Assistant app running on my Android phone

www.allitebooks.com

The Presenter VCL application is in charge of mimicking a keyboard key press when the mobile app sends the proper commands and to send the remaining minutes to the mobile app every 5 seconds (we don't require a clock; an update every 5 seconds is enough). The following is the screenshot of the VCL application:



The VCL application that controls the desktop application showing the slides

In the App Tethering model, there isn't a server or a client. There is an application (or an app) that connects to other apps, but then the two or more apps are peers.

Each application can do the following:

- **Share resources**: When other apps subscribe to a shared resource, every time the shared resource changes all the subscribed apps are notified following the publish/subscribe model.
- **Share actions**: An app can discover and invoke actions published by other apps.
- **Send strings**: One of the apps can send a string to one of the other apps. The string can contain anything, even a complex JSON object.
- **Send streams**: One of the apps can send a stream to one of the other apps. The stream can also contain binary data such as an image or an MP3 file.

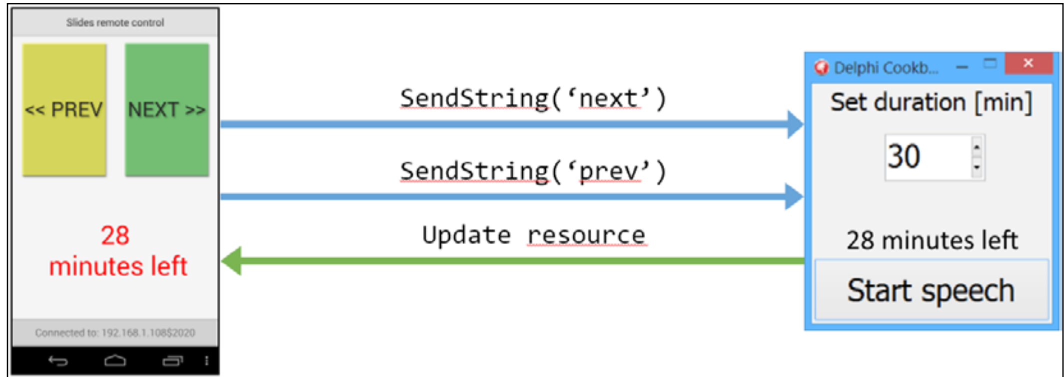
The presenter assistant we're talking about is very simple. The mobile app has to send two strings to the desktop application. The first when we want the next slide and the second when we want the previous slide. The VCL application running on the PC has to publish a resource showing the remaining minutes.

How to do it...

Open the project group in `Chapter5\RECIPE08`. There are two projects: `Presenter.dproj` (the VCL application) and `PresenterRemote.dproj` (the Android app).

Let's start showing how the applications work. Run the presenter application, and then run the PresenterMobile app on your phone and press **Connect**. If your phone is connected to the same network as your PC, you should be able to connect and see something like **Connected to: 192.168.1.101$2020** on your phone. This means that the mobile app is connected to the VCL application listening on port 2020. Now go to your desktop, write an integer number in the **SpinEdit**, and press **Start Speech**. The application goes to the taskbar. Now, open MS PowerPoint with a presentation (or another program which is sensible to the left and right arrow, also the Delphi source code editor is good) and press repeatedly the left or right button in the mobile app. You should see the slides (or the cursor) moving.

The following schema shows the communication between the mobile app and the VCL application after the discovering and pairing phases.



The communication between the mobile app and the VCL application

When the PresenterMobile app sends the `NEXT` command (using `SendString`) the Presenter application receives it and sends a `VK_RIGHT` Windows keyboard event. By sending a Windows keyboard key event, the application is mimicking a user who is using the keyboard, so the key sent is intercepted by the window which has got the focus in that moment (just like a normal keyboard works). If in the foreground there is an MS PowerPoint (or OpenOffice.org Impress) presentation, you get the next slide (because if you hit the right arrow during a presentation, you go to the next slide). The same is applicable for the `PREV` command, which in turn sends a `VK_LEFT` key to MS PowerPoint.

The relevant part about this message exchange is shown as follows:

```
const
  NEXT_SLIDE = Ord(VK_RIGHT);
  PREV_SLIDE = Ord(VK_LEFT);
  DEFAULT_MINUTES = 30;

procedure SendKey(const C: Word);
var
  kb: TInput;
```


www.allitebooks.com

```
begin
  kb.Itype := INPUT_KEYBOARD;
  kb.ki.wVk := C;
  kb.ki.wScan := MapVirtualKey(C, 0);
  kb.ki.dwFlags := 0;
  SendInput(1, kb, SizeOf(kb));
  kb.ki.dwFlags := KEYEVENTF_KEYUP;
  SendInput(1, kb, SizeOf(kb));
end;

procedure TMainForm.TetheringAppProfile1ResourceReceived(
  const Sender: TObject;
  const AResource: TRemoteResource);
var
  Cmd: string;
begin
  Caption := AResource.Value.AsString;
  if AResource.Hint.Equals('cmd') then
  begin
    Cmd := AResource.Value.AsString;
    if Cmd.Equals('prev') then
      SendKey(PREV_SLIDE)
    else if Cmd.Equals('next') then
      SendKey(NEXT_SLIDE);
  end;
end;
```

How about the connection between the applications? For this app, I've used the Group feature of App Tethering. As you know, there are two ways to connect your applications:

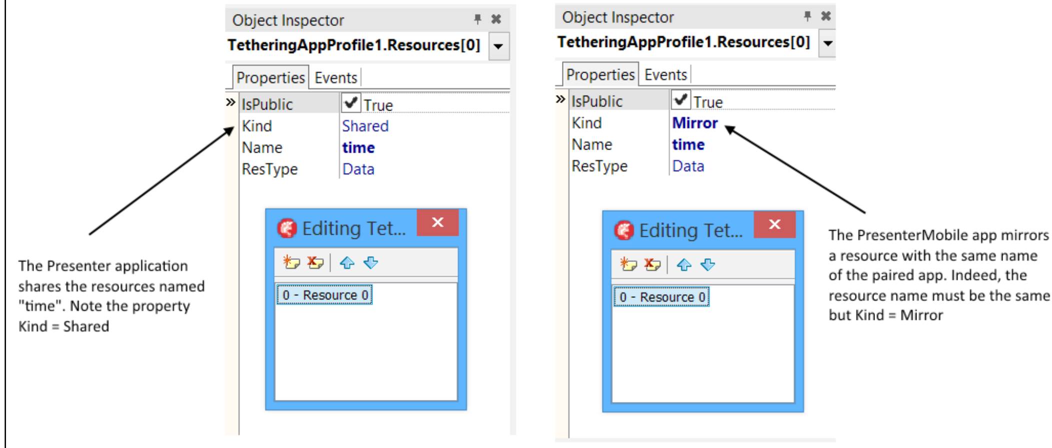
- Define two applications as belonging to the same group and use automatic discovering and pairing. This approach is very simple, but not so flexible.
- Obtain a list of discovered applications and then request to pair with specific applications. This approach is more flexible but requires a bit of work.

Considering the scenario, I've used the `Group` property and the Autoconnect feature. Here's the code under the **Connect** button in the mobile app:

```
procedure TMainForm.btnConnectClick(Sender: TObject);
begin
  TetheringManager1.AutoConnect(2000);
end;
```

In order to make the AutoConnect feature work properly, both the `TetheringAppProfile` components must have the same value in the `Group` property. In our case, the value is `com.danieleteti.presenters`.

Also, the presenter application shares a resource with the PresenterMobile. In order to automatically subscribe to the resource update notification, the resource's name must be the same on all the paired apps, as shown in the following screenshot:



The resources configuration

With this configuration, you can simply update the value of the resource in the desktop application using the following code:

```
TetheringAppProfile1.Resources.
    FindByName('time').Value := MinutesLeft;
```

Updating the local resource `time` causes an update to the remote resource with the same name and the following event handler is executed on the mobile app:

```
procedure TMainForm.TetheringAppProfile1Resources0ResourceReceived
  (const Sender: TObject; const AResource: TRemoteResource);
begin
  lblMinutes.Text := AResource.Value.AsString +
    sLineBreak + ' minutes left';
end;
```

There's more...

App Tethering is a nice technology. However, it is not a replacement for a full-fledged server, but a good tool to easily create companion applications. Here's some documentation about it:

- A fast introduction to *App Tethering with Delphi XE6* at `https://www.youtube.com/watch?v=oeMQdvxi560`
- *Using App Tethering* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Using_App_Tethering`

www.allitebooks.com

- *Adding App Tethering to Your Application* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Adding_App_Tethering_to_Your_Application`
- *Connecting to Remote Applications Using App Tethering* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Connecting_to_Remote_Applications_Using_App_Tethering`
- *Sharing and Running Actions on Remote Applications Using App Tethering* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Sharing_and_Running_Actions_on_Remote_Applications_Using_App_Tethering`
- *Sharing Data with Remote Applications Using App Tethering* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Sharing_Data_with_Remote_Applications_Using_App_Tethering`
- *Fun with Delphi XE6 App tethering and barcodes* at `http://fixedbycode.blogspot.it/2014/04/fun-with-delphi-xe6-app-tethering-and.html`

Creating DataSnap Apache modules

One of the Delphi features most awaited by server side Delphi developers is the support for the building of Apache webserver module. Delphi XE6 brings this feature! The most recent Apache versions supported are Versions 2.0, 2.2 and 2.4. An apache module is compatible only with the specific version for which it has been compiled. So, be sure about the apache version you have to deploy your module before creating the project. However, it's possible to change the target apache version just by changing its unit name.

Getting ready

In this recipe, we'll create a very simple REST service with only one method, which returns a list of people. The service will be built using the Embarcadero DataSnap framework and the service itself will be packaged as an Apache Webserver module. The real goal of this recipe is to show how to use the Delphi strength in creating Apache module, and a very light introduction to DataSnap.

How to do it...

This recipe has the following steps:

1. The Apache HTTP Server (`httpd`) is a project of The Apache Software Foundation and has been the most popular web server on the Internet since April 1996. On Windows, one of the recommended binary distributions is maintained by the Apache Lounge community. Go to `http://www.apachelounge.com/download/` and download the recently updated 2.4.x Version as a ZIP file. In this recipe, we'll use the Win32 Version, so download that one.

2. Unzip the Apache distribution in a folder named `Apache24` (for example, `C:\DEV\Apache24`).
3. The Apache main configuration is contained in the `httpd.conf` file in `C:\DEV\Apache24\conf\`. Open it with a good text editor. This file contains all the main configurations and includes a bounce of other configuration files. Configuring Apache is easy; however, in this recipe, we'll configure it to let it run our module. Let's start with a very basic configuration; however, the `http.conf` syntax can be complex, so pay attention to the following steps.
4. Look for `ServerRoot`. Currently, it should look like this:

   ```
   ServerRoot "c:/Apache24"
   ```

5. Change the folder name to `C:/DEV/Apache24`. Please note that we're using / as folder separator and not \. Also, don't terminate the folder name with a trailing slash. Now the line should look like this:

   ```
   ServerRoot "c:/DEV/Apache24"
   ```

6. Look for `DocumentRoot`. This path is where static files are placed. Currently, this should look like this:

   ```
   DocumentRoot "c:/Apache24/htdocs"
   <Directory "c:/Apache24/htdocs">
   ```

7. Change the folder name to `C:/DEV/Apache24/htdocs` on the two lines:

   ```
   DocumentRoot "c:/DEV/Apache24/htdocs"
   <Directory "c:/DEV/Apache24/htdocs">
   ```

8. Look for `ServerName`. The `ServerName` directive gives the name and port that the server uses to identify itself. Currently, the line of code is commented as follows:

   ```
   #ServerName www.example.com:80
   ```

9. Just after the commented line, add the following line:

   ```
   ServerName localhost:80
   ```

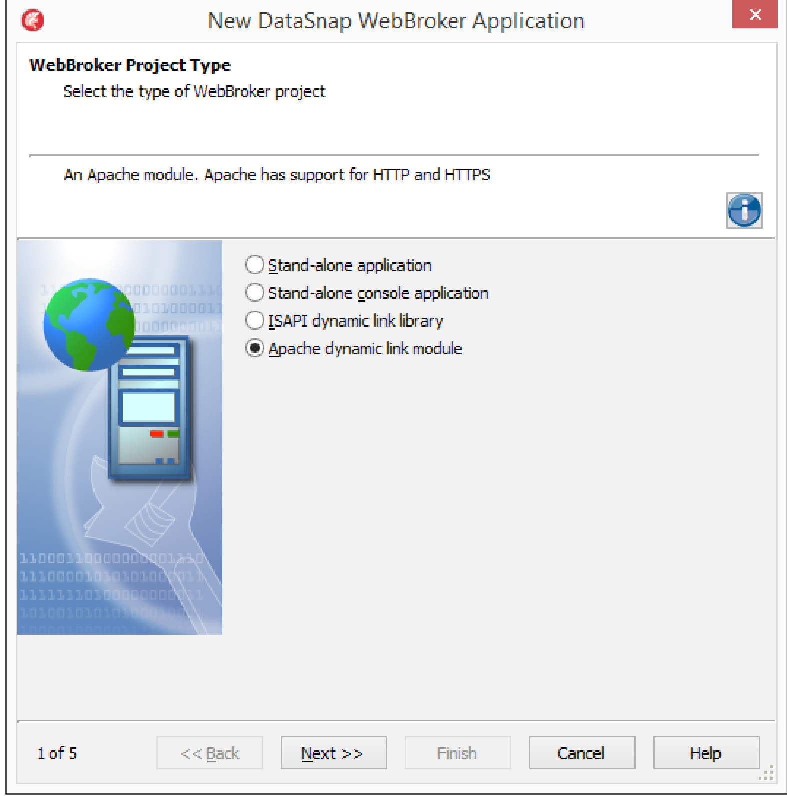
10. Let's test if our Apache is correctly configured. Open a command prompt, go to the `C:\DEV\Apache24` folder, and run the following command:

    ```
    bin\httpd.exe
    ```

11. Errors will be printed on the standard output. If no errors have been printed, launch a browser and navigate to `http://localhost`. You should get a white page with **It works!** text on it. If so, your Apache installation is running correctly. Now, Apache is running in application mode. It is possible to install it as a service with a simple command that we'll see later.

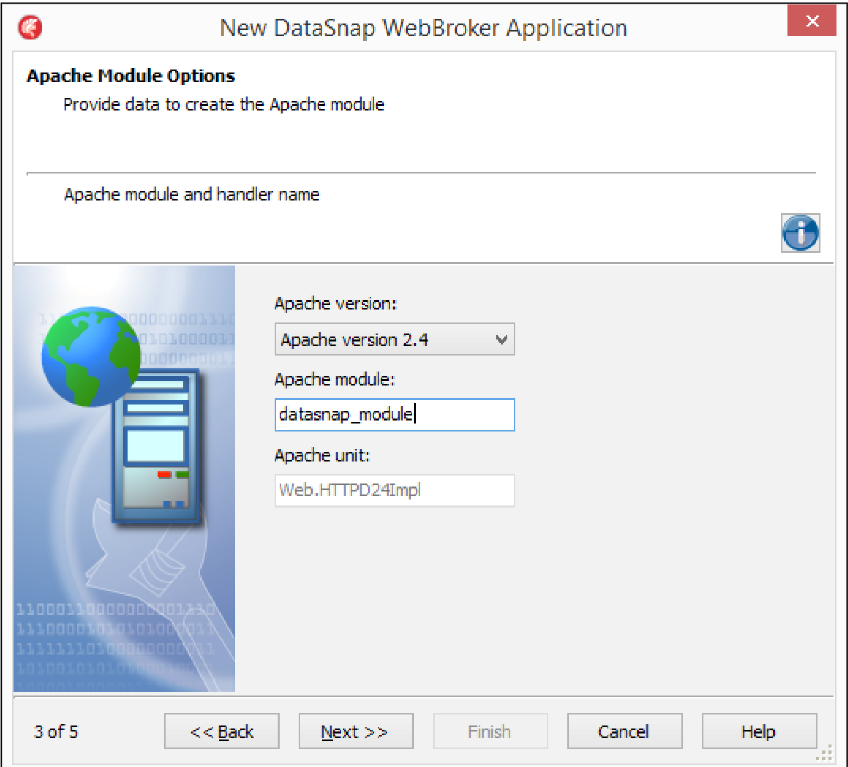
www.allitebooks.com

12. Note that we are configuring Apache just to run our modules. It is not configured to be exposed on the Internet. So, please read carefully the documentation about the configuration or ask some Apache expert before letting your server go into the wild!
13. Terminate Apache by pressing *Ctrl* + *C*, leave the command prompt for a moment, and go back to Delphi.
14. Let's create our DataSnap WebBroker project as Apache 2.4 module.
15. Go to **File** | **New** | **Other**, and then go to **Delphi Projects** | **DataSnap Server** | **DataSnap WebBroker Application**.
16. The wizard asks which kind of project we're about to create. Select **Apache dynamic link module** and click on **Next** (as shown in the following screenshot).



The DataSnap Wizard. We choose the Apache module option

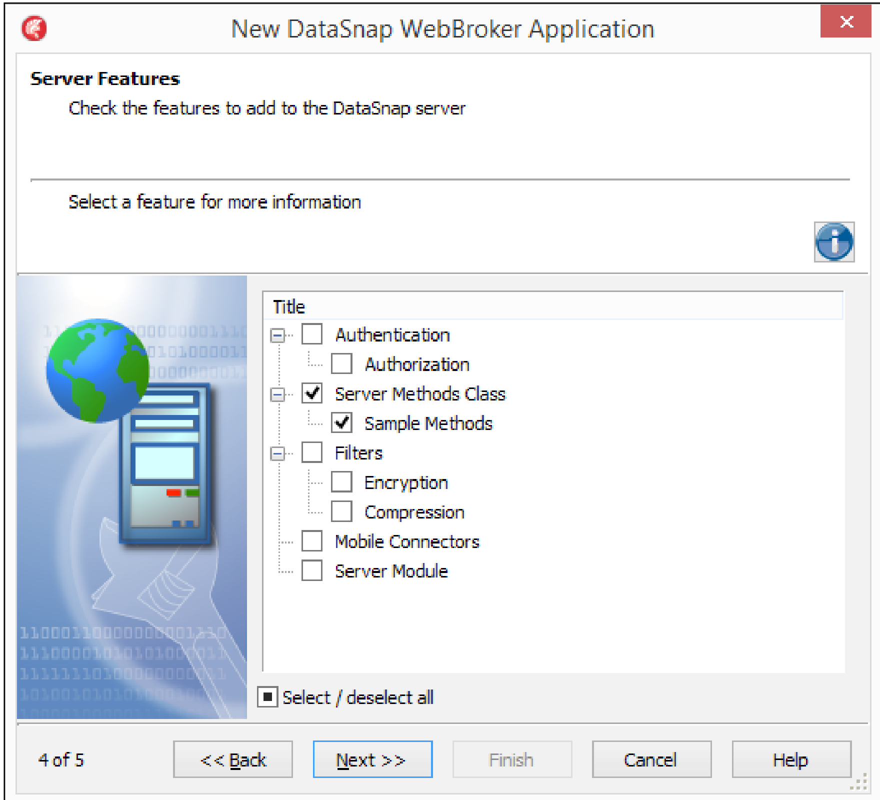
17. Then, the wizard asks which Apache version our module will be built for. Select **Apache version 2.4**, name it `datasnap_module`, and click on **Next**.



The wizard allows us to define the Apache module name and the target Apache version for the module

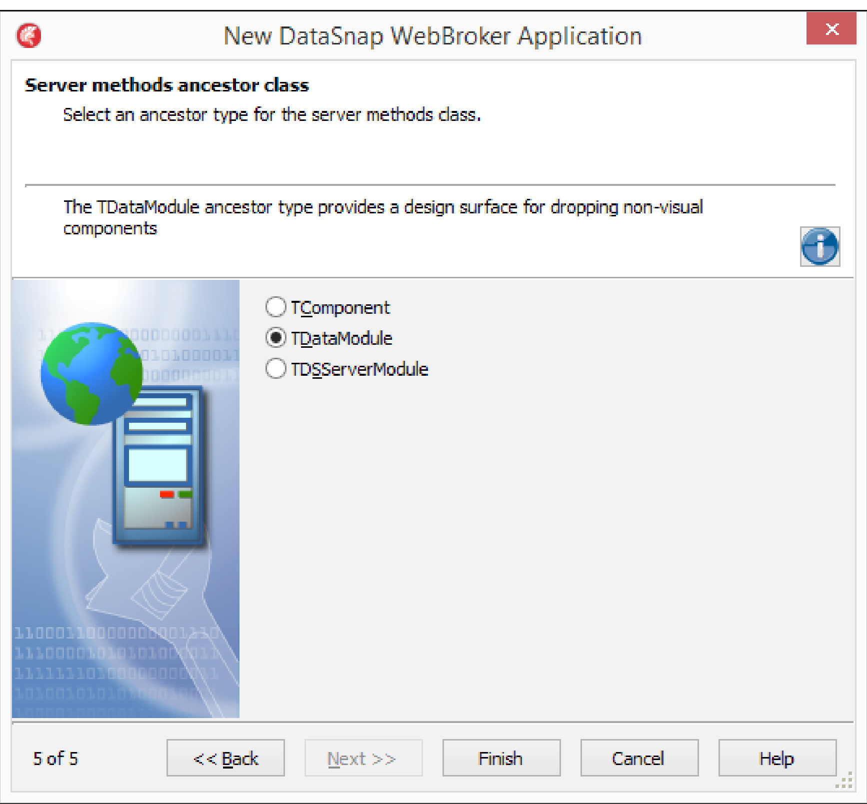
www.allitebooks.com

18. On the next screen, the wizard asks about the functionalities that we want to include in our `DataSnap` module. Leave the defaults and click on **Next**.



Let the wizard include some sample methods in the `DataSnap` module

19. In the next screen, select **TDataModule** and click on **Finish**.



Using the TDataModule as ancestor class we got a design surface without the overhead of the IAppServer interface that we'll not use

20. Delphi has created a complete Apache 2.4 module containing a DataSnap REST server. Wow! Now let's add some features to it.
21. Save the project using the default names.
22. Open `ServerMethodsUnit1.pas`, make the designer visible, and drop on it a `TFDConnection` and a `TFDQuery`. Connect the `TFDQuery` to the `TFDConnection`, and then configure the `TFDConnection` to point at the sample database in the `DATA` folder contained in this recipe. The connection configuration parameters should be similar to the following:

```
Database=C:\DEV\Chapter05\CODE\RECIPE08\DATA\SAMPLES.IB
User_Name=sysdba
Password=masterkey
DriverID=IB
```


www.allitebooks.com

23. Go to the code editor and declare the following method in the `public` section of the `TDataModule`:

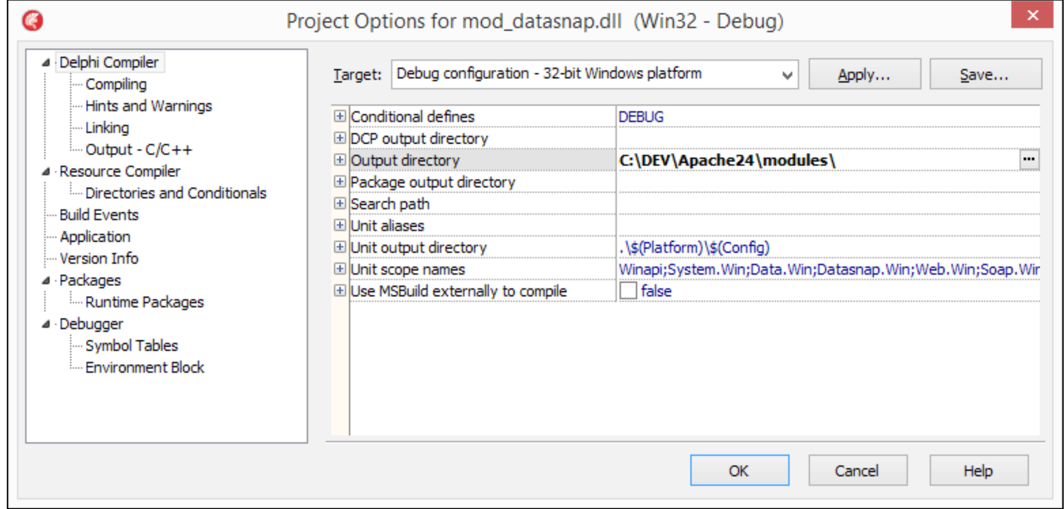
```
public
  . . . //other methods
  function GetEmployees: TJSONArray;
end;
```

24. Press *Ctrl* + *Shift* + *C* to implement the method body and fill it with the following code:

```
function TServerMethods1.GetEmployees: TJSONArray;
begin
  FDQuery1.Open('SELECT * FROM PEOPLE');
  Result := FDQuery1.AsJSONArray;
end;
```

25. Go to the `implementation uses` clause and add the `ObjectsMappers` unit (it is a unit contained in the DelphiMVCFramework project that we'll use to do standard `DataSet` serialization).

26. Build the project. Now, our Apache module is ready, but how to test and debug it? As the first thing, we've to put the compiled DLL in the right place. To allow Apache loads, our module is useful to have it at the same level of the built-in modules. Go to **Project** | **Options** | **Delphi** and write in the **Output directory** section, the `C:\DEV\Apache24\modules\` path as shown, then press **OK**:



Configure the project output directory to compile directly where Apache looks for modules

27. Compile the project and go back to the `httpd.conf` file.

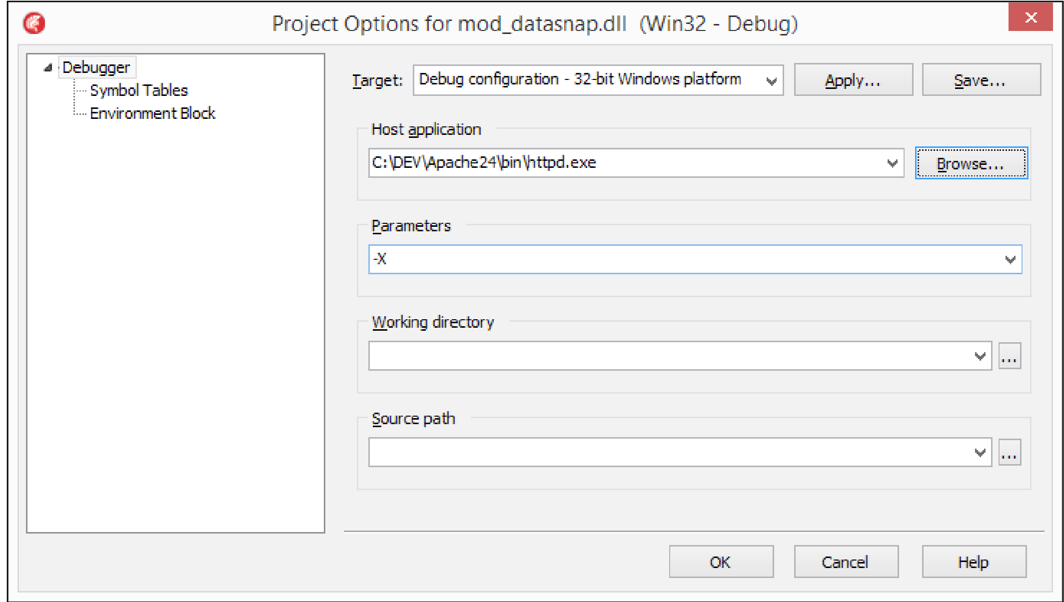
28. Look for the `LoadModule` string in the file. You will find a lot of lines with this directive and many of them are commented. Just after the last `LoadModule` line (doesn't matter if it is commented or not), add the following lines and save the file:

```
LoadModule datasnap_module modules/mod_datasnap.dll

<Location /api>
   SetHandler mod_datasnap-handler
</Location>
```

29. Now, go back to the command prompt. Go to the `Apache24` folder under `C:\DEV\` and launch the following command:

```
bin\httpd.exe
```

30. Go to a browser, and navigate to the `http://localhost/api/datasnap/rest/TServerMethods1/getemployees` URL. You should get the DataSnap JSON response from the Apache module just created.
31. How to debug our module? Terminate Apache by pressing *Ctrl* + *C* from the command line and go back to Delphi.
32. Go to **Run** | **Parameters**, configure the values as shown in the following screenshot, and click on **OK**:



Let's set up the debugger to debug the module. Note the `-X` parameter passed to the `httpd.exe` executable.

www.allitebooks.com

33. Now, Delphi will start Apache for us and we'll be able to debug the module as for any Delphi program. The `-X` parameter passed to `httpd.exe` launches Apache in the debug mode with only one worker, so Delphi doesn't need to debug the Web server spawned processes.
34. Run the project. Apache will silently start, launched by Delphi, and our module is loaded by the `httpd.exe` process. Now, we are able to debug the module using breakpoints and all the ordinary things.

How it works...

Apache is configured to load a particular module, our module. That source code of that module is opened in the Delphi IDE. When Delphi compiles the DLL module, it writes the module where Apache is expecting. Just after the compilation, Delphi launches Apache in the application mode with the `-X` parameters (avoiding spawned processes). Apache loads the DLL as configured in the `httpd.conf` file, and Delphi attaches its debugger to the `httpd.exe` process and to its `datasnap_module.dll` file. This approach is valid for any DLL that is loaded at runtime by some other software, and it is still valid for every WebBroker program compiled as Apache module or ISAPI DLL.

There's more...

There were a lot of concepts in this recipe. DataSnap is a complex and powerful framework from Embarcadero able to create TCP/IP and HTTP/S servers. I have held many training sessions on it, and suggest you to give it a try. It is also present in Delphi XE6 Enterprise and more. Here are some links if you want to learn more:

- *DataSnap Overview and Architecture* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/DataSnap_Overview_and_Architecture`
- *Using a DataSnap Server with an Application* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Tutorial:_Using_a_DataSnap_Server_with_an_Application`
- *Using a REST DataSnap Server with an Application* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Tutorial:_Using_a_REST_DataSnap_Server_with_an_Application`

Now, let's look at something about the Apache webserver.

Apache HTTP Server security tips can be found at the following URL:

`http://httpd.apache.org/docs/current/misc/security_tips.html`

After you have configured and secured your Apache webserver, you can install it as a Windows service using the following command line:

```
.\bin\httpd.exe -k install -n "My DataSnap Server"
```

To uninstall, use the following command line:

```
.\bin\httpd.exe -k uninstall -n "My DataSnap Server"
```

In this way, you can package a customized Apache distribution to deploy and run your custom modules. I do it very often with my services that have to be published on the Internet, because Apache is stronger and more secure compared to the Delphi built-in web server based on INDY (to each his own).

However, even if in this recipe we've used a dedicated Apache installation to host our module, you can also use an already deployed instance (and often you will do so). The deployment process is the same: copy your module in some path accessible from the webserver, change the `httpd.conf` file to load your module, and restart the server. That's it!

www.allitebooks.com

6

Riding the Mobile Revolution with FireMonkey

In this chapter, we will cover the following recipes:

- Taking a photo, applying effects, and sharing it
- Using listview to show and search local data
- Do not block the main thread!
- Using SQLite databases to handle a to-do list
- Using a styled TListView to handle a long list of data
- Taking a photo and location and sending it to a server continuously
- Talking to the backend
- Making a phone call from your app!
- Tracking the application's life cycle

Introduction

In this chapter, we'll see how to develop mobile apps using Delphi. The recipes in this chapter require a working development configuration of your PC and in the case of iOS, your Mac to talk with the Android or iOS device. A detailed tutorial on how to properly configure your system for this purpose can be found on the Embarcadero DocWiki. To develop and deploy an app for iOS, you require an Apple computer and an actual iOS device, while to develop and deploy for Android, you need to have only the device. There is also an emulator in the SDK where you can deploy an app but, currently, it is very slow; if you want to really develop for Android, having an actual device where deploying is faster than using an emulator is recommended.

Visit the following links for more information and relevant documentation that will help you to configure different environments:

- **For Android configuration**: The *Set Up Your Development Environment on Windows PC (Android)* documentation can be found at `http://docwiki.embarcadero.com/RADStudio/XE5/en/Mobile_Tutorial:_Set_Up_Your_Development_Environment_on_Windows_PC_(Android)`
- **For iOS configuration**: The *Set Up Your Development Environment on the Mac (iOS)* documentation can be found at `http://docwiki.embarcadero.com/RADStudio/XE5/en/Mobile_Tutorial:_Set_Up_Your_Development_Environment_on_the_Mac_(iOS)`
- **For Windows configuration**: The *Set Up Your Development Environment on Windows PC (iOS)* documentation can be found at `http://docwiki.embarcadero.com/RADStudio/XE5/en/Mobile_Tutorial:_Set_Up_Your_Development_Environment_on_Windows_PC_(iOS)`

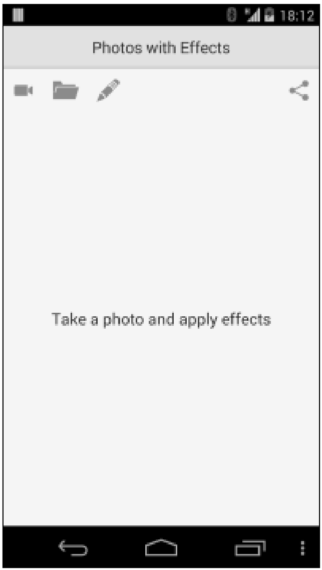
Taking a photo, applying effects, and sharing it

This recipe will introduce the mobile development world using a simple app that shows how to take a photo directly from the camera or from the photo library, apply some effects to it, and then share it using one of the installed apps on the device.

Getting ready

This recipe makes an extensive use of Delphi **actions**. Actions are an implementation of the GoF Command design pattern and are an important tool for the Delphi developer since the initial versions of Delphi. You can use them as much as you can. In the mobile era, actions are even more important and useful. Indeed, actions can be used to execute common tasks such as taking a photo from camera, getting a photo from the library, or sharing some content with the other apps. Here's how our app will look:

www.allitebooks.com



The Photo with Effects app with buttons on the top; three out of four buttons are bound to standard actions

How to do it...

Now we are about to create our first FireMonkey mobile app. Let's start!

1. Create a new mobile app by navigating to **File** | **New** | **FireMonkey mobile application – Delphi**.
2. Select the **Header/Footer** template and click on **OK**.
3. The IDE has just created a base for us. Name the form as `MainForm` and let's add our logic and adapt the UI.
4. Select the **HeaderLabel** label and change its **Text** property to **Photos with Effects**.
5. Select the **TToolbar** component named **Footer** and delete it.
6. Now, drop a **TPanel** component and align it to the **alTop** value so that it'll be just below the header.
7. Add four buttons to the just dropped **TPanel** component. Align three of them to the left-hand side and the other one to the right-hand side. Now, starting from the left-hand side, set the following values for their **StyleLookup** property:
    - **cameratoolbutton**
    - **organizetoolbutton**
    - **composetoolbutton**
    - **actiontoolbutton**

8. Now the buttons should look like the one in the previous screenshot.
9. Drop a **TImage** component in the center of the form and align it to **Client**. This component will be our main working area.
10. Set **TImage.MarginWrapMode** to **iwFit**.
11. Drop a **TListView** component at the center of the form, make it a bit wider, and name it `lvEffects`. This listview will be used to show the available effects to the user.
12. Drop a **TActionList** component, double-click on it, and then, from the little menu button on the left-hand side, click on **New Standard Action** (or you can use *Ctrl* + *Ins*).
13. From the resultant window, select **TTakePhotoFromCameraAction** and click on **OK**. Repeat the process and add the **TTakePhotoFromLibraryAction** and **TShowShareSheetAction** actions. Note that these actions are actually invisible components with properties and events just like a persistent field in a dataset. In a few moments, we'll be back to these components to customize their default behaviors.
14. Starting from the left-hand side, connect the following actions to the buttons placed in the **TPanel** component at the top.
    1. Set the first **Action** button to **TakePhotoFromCameraAction1**
    2. Set the second **Action** button to **TakePhotoFromLibraryAction1**
    3. Do not assign an action, but name it `btnEffects`
    4. Set the fourth **Action** button to **ShowShareSheetAction1**
15. In the app, there will be a mechanism to dynamically load the available effects inspecting the **TFilterEffect** descendants placed on the form. So, we can simply drop some effects on the form and the app will automatically load them in a list allowing the user to use them. Drop the following effects on the form: **TEmbossEffect**, **TRadialBlurEffect**, **TContrastEffect**, **TColorKeyAlphaEffect**, **TInvertEffect**, **TSepiaEffect**, **TTilerEffect**, **TPixelateEffect**, **TToonEffect**, **TPencilStrokeEffect**, **TRippleEffect**, **TWaveEffect**, **TWrapEffect**, and **TInnerGlowEffect**.
16. Now we've to write some code. In the `private` section of the `TMainForm` class, declare the following instance members:

```
private
  FItemsEffectsMap: TDictionary<Integer, TFilterEffect>;
  FUndoEffectsList: TObjectStack<TFilterEffect>;
  FUndoEffectItem: TListViewItem;
  FTopWhenShown: Extended;
  procedure LoadPhoto(AImage: TBitmap);
  procedure RecalcMenuPosition;
  procedure RemoveCurrentEffect(ARemoveFromList: boolean);
  function EffectNameByClassName(
                    const AClassName: String): String;
```


www.allitebooks.com

17. Hit *Ctrl* + *Shift* + *C* to create empty methods and fill them with the following code:

```
procedure TMainForm.LoadPhoto(AImage: TBitmap);
begin
  Label1.Text := '';
  RemoveCurrentEffect(False);
  FUndoEffectsList.Clear;
  Image1.Bitmap.Assign(AImage);
end;

procedure TMainForm.RecalcMenuPosition;
begin
  FTopWhenShown := ClientHeight / 2 - lvEffects.Height / 2;
  lvEffects.Height := ClientHeight / 2;
  lvEffects.Position.X := ClientWidth / 2 -
        lvEffects.Width / 2;
end;

procedure TMainForm.RemoveCurrentEffect(ARemoveFromList: boolean);
begin
  if FUndoEffectsList.Count = 0 then
    Exit;
  Image1.RemoveObject(FUndoEffectsList.Peek);
  if ARemoveFromList then
    FUndoEffectsList.Pop;
  Image1.Repaint;
end;

function TMainForm.EffectNameByClassName(
   const AClassName: String): String;
begin
  Result := AClassName.Substring(1);
  Result := TRegEx.Replace(Result, '[A-Z]', ' $0').TrimLeft;
end;
```

18. To compile this code, add `System.Generics.Collections` in the `uses interface` section and `System. RegularExpressions` in the uses `implementation` section. Build the project just to ensure that everything is alright.

19. Now create the `OnCreate` event handler for the form and add the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  eff: TFmxObject;
  lbi: TListViewItem;
begin
```

```
    FItemsEffectsMap := TDictionary<Integer, TFilterEffect>.Create;
    FUndoEffectsList := TObjectStack<TFilterEffect>.Create(False);
    lvEffects.Position.Y := -lvEffects.Height;
    lvEffects.BeginUpdate;
    try
      FUndoEffectItem := lvEffects.Items.Add;
      FUndoEffectItem.Text := 'Undo';

      for eff in Children do
      begin
        //if it is an effect, add it to the listview and to the
        //dictionary. Use the class name to create a friendly name
        if eff is TFilterEffect then
        begin
          lbi := lvEffects.Items.Add;
          lbi.Text := EffectNameByClassName(eff.ClassName);
          FItemsEffectsMap.Add(lbi.Index, TFilterEffect(eff));
        end;
      end;
    finally
      lvEffects.EndUpdate;
    end;
    lvEffects.ApplyStyleLookup;
  end;
```

20. Now create the `FormResize` and `FormShow` event handlers. In the `body` section of these event handlers, call the `RecalcMenuPosition` procedure.
21. Select the listview and create the `OnItemClick` event handler. This event will be called when the user selects an effect from the list. Now, we've to remove, with an animation, the list from the form and apply the effect. Fill the event handler with this code:

```
procedure TMainForm.lvEffectsItemClick(
   const Sender: TObject;const AItem: TListViewItem);
begin
  lvEffects.AnimateFloat(
    'Position.Y', -lvEffects.Height, 0.3,
    TAnimationType.&In,
    TInterpolationType.Quadratic);

  if AItem = FUndoEffectItem then // undo effect
  begin
    RemoveCurrentEffect(true);
    if FUndoEffectsList.Count > 0 then
      Image1.AddObject(FUndoEffectsList.Peek);
```


www.allitebooks.com

```
  end
  else
  begin // apply new effect
    RemoveCurrentEffect(False);
    FUndoEffectsList.Push(FItemsEffectsMap[AItem.Index]);
    Image1.AddObject(FUndoEffectsList.Peek);
  end;
  Image1.Repaint;
end;
```

Now we've to create something that is able to show the list of available effects when the user needs to apply one of them. The effect list will drop down from the top of the form with a bouncing effect and will go away in the same way (but in a reversed manner).

22. Create `btnEffects` on the `click` event handler and fill it with the following code:

```
procedure TMainForm.btnEffectsClick(Sender: TObject);
begin
  if FUndoEffectsList.Count = 0 then
    FUndoEffectItem.Text := '<No effect to undo>'
  else
    FUndoEffectItem.Text := '[Undo ' +
       EffectNameByClassName(FUndoEffectsList.Peek.ClassName) +
      ']';
   lvEffects.AnimateFloat('Position.Y', FTopWhenShown, 0.6,
   TAnimationType.Out, TInterpolationType.Bounce);
  lvEffects.ApplyStyleLookup;
end;
```

23. We've to customize the actions' behaviors. Double-click on **TActionList1**, select the **ShowShareSheet1** action, create the `OnBeforeExecute` event handler, and then fill it with the following code:

```
procedure TMainForm.ShowShareSheetAction1BeforeExecute(
                                                  Sender: TObject);
begin
  if FUndoEffectsList.Count > 0 then
    FUndoEffectsList.Peek.ProcessEffect(nil, Image1.Bitmap, 0);
  ShowShareSheetAction1.Bitmap.Assign(Image1.Bitmap);
end;
```

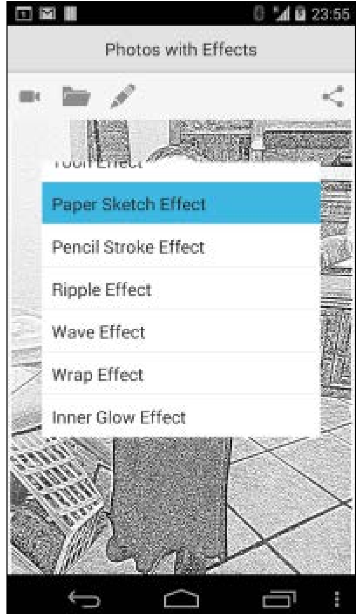
24. Create the `OnDidFinishTaking` event handler for the **TakePhotoFromCameraAction1** and **TakePhotoFromLibraryAction1** actions and fill both with the following code:

```
procedure TMainForm.TakePhotoFromCameraAction1DidFinishTaking(
   Image: TBitmap);
```

```
begin
  LoadPhoto(Image);
end;

procedure TMainForm.TakePhotoFromLibraryAction1DidFinishTaking(
    Image: TBitmap);
begin
  LoadPhoto(Image);
end;
```

25. Select an available target in the **Project Manager** window (in your phone or an available emulator in the case of Android) and run the app.

    Tap the first button from the left-hand side and take a photo. The image should be placed in the main area. Tap on the **btnEffects** button, and you should see the listview falling from the top to allow you to choose effects. The first item should be **<No effect to undo>**. Select an effect and see how the effect is applied to the photo. Tap **btnEffect** again, and you should see the first item saying **[Undo ...]**. Play with the app by adding effects and using the **undo** features to sequentially go back to the beginning. Note that the effects will not be added (so you cannot have **Emboss** along with **Blur** applied at the same time). When you are satisfied with the result, tap on the button on the right-hand side to share the photo with effects applied using an installed app:



A photo taken from the camera with the Paper Sketch effect applied; the menu is visible and ready to apply another effect

www.allitebooks.com

How it works...

When launched, the app loads the available effects inspecting all the **TEffectFilter** descendants placed on the form and stores the component reference in a dictionary indexed with the **ListItem** index in the list. To create a friendly effect name for the UI, the effect's class name is used. Indeed, all the effect classes have the typical Pascal case naming convention (just like all the other things in Delphi) and the `EffectNameByClassName` method uses a regular expression to make a string such as `TRadialBlurEffect` in something like the **Radial Blur** effect. To do it, the initial `'T'` is removed and then it has been used a regular expression to split the words as shown in the following code:

```
function TMainForm.EffectNameByClassName(
                    const AClassName: String): String;
begin
  Result := AClassName.Substring(1);
  Result := TRegEx.Replace(Result, '[A-Z]', ' $0').TrimLeft;
end;
```

Another nice feature implemented is the **UnDo** stack. Each time a new effect is applied to the image, the current one is pushed onto the stack. So, when you tap on **Undo <current effect>**, the current effect is removed and the top of the stack is used to retrieve the last effect. With this approach, which is used in multiple scenarios, we can go back to the beginning without losing any steps.

The last note goes to the share functionality. The effects are applied by adding the related components to the children controls' list of the image. Following the parenting relation, FireMonkey performs all the drawing jobs; however, the image itself is not transformed, only its visual representation is "effected." Now, if you try to read programmatically, the bitmap contained by the **TImage** control, the image is not "effected", and you get the original image. So how do you actually apply the effect to the image? Check the `ShowShareSheetAction1BeforeExecute` event handler:

```
procedure TMainForm.ShowShareSheetAction1BeforeExecute(
                                                     Sender: TObject);
begin
  if FUndoEffectsList.Count > 0 then
    FUndoEffectsList.Peek.ProcessEffect(nil, Image1.Bitmap, 0);
  ShowShareSheetAction1.Bitmap.Assign(Image1.Bitmap);
end;
```

As you can see, the `effect` component has a `ProcessEffect` method that actually takes an image and applies the transformation to it. In this case, the effect is not only visually applied but is actually applied. So, when you share the affected image, the image is really affected.

There's more...

A lot of concepts are covered in this first mobile recipe. As you will see, the main approach about the mobile development is not different from a normal FireMonkey application. This is an extraordinary feature of FireMonkey: one framework for all platforms. If you are good at FireMonkey, you are at least 80 percent good for all the supported platforms. However, in the mobile scope, all the things get a bit more difficult and slow due to the platform limits and the inherent slower `edit/run/test` loop.

To get more info about effects, you can check the following articles:

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/FireMonkey_Image_Effects`
- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Applying_FireMonkey_Image_Effects`

To get more information about regular expressions as implemented in Delphi, check the following articles:

- `http://docwiki.embarcadero.com/RADStudio/XE6/en/Regular_Expressions`
- `http://docwiki.embarcadero.com/CodeExamples/XE6/en/RTL.RegExpression_Sample`

To get some information about the **Command** design pattern and the other 22 fundamental patterns, you can read the classic book, *Design Patterns: Elements of Reusable Object-Oriented Software*, *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*, *Addison Wesley Professional* (`http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612`).

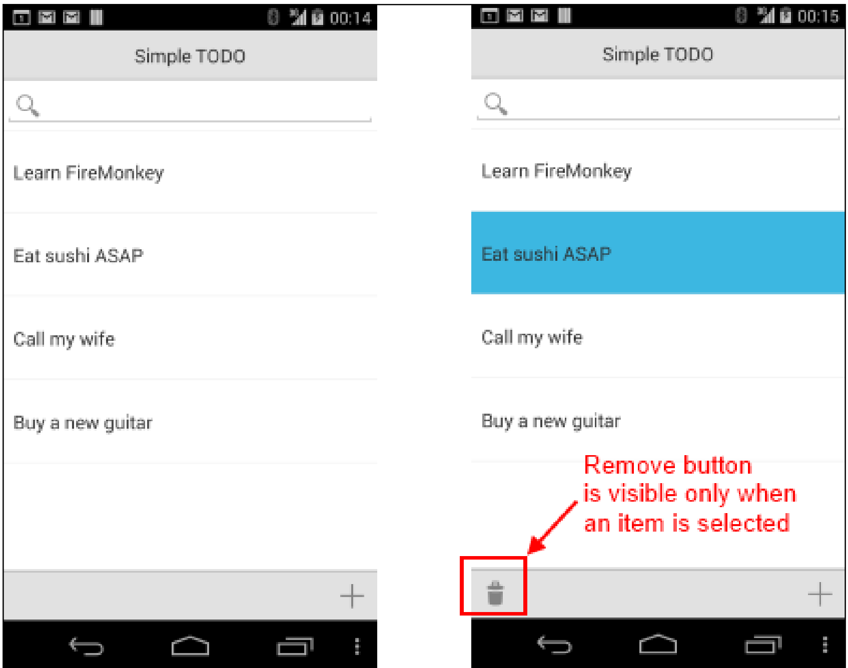
Using listview to show and search local data

In many cases, on a mobile app, data is read from remote servers and then locally stored to make it available without Internet connection. In this recipe, you'll see how to read and write to a file as well as how to show and search that data in a listview.

Getting ready

This recipe is simple and short, but it is really useful because the concepts exposed are reusable and allow you to gain confidence with some very important best practices. The final aspect of the app is shown in the following screenshot. Note that the Delete button is visible only when an item is selected.

www.allitebooks.com



The Simple TODO app; when an item is selected, the Delete button is visible

How to do it...

1. Create a new mobile app by navigating to **File** | **New** | **Other...** | **Delphi Projects** | **FireMonkey Mobile Application**.
2. Choose the **Header/Footer** template and click on **OK**.
3. As soon as Delphi creates the project template, save all the files with the following names:
    - Save the project as `SimpleTODO.dproj`
    - Save the form as `MainFormU.pas`
4. Drop a **TListView** component on the form and set the following properties (the relevant properties are extracted from the `MainFormU.fmx` file):

```
object ListView1: TListView
  Align = Client
  ItemAppearance.ItemHeight = 80
  ItemAppearanceObjects.ItemObjects.Text.WordWrap = True
  ItemAppearanceObjects.ItemObjects.Accessory.Visible = False
  SearchVisible = True
end
```

5. Drop a **TActionList** component on the form and add two actions. Name them `acNew` and `acDelete`.
6. Create the `OnExecute` event handler for the two actions using the following code:

```
procedure TMainForm.acDeleteExecute(Sender: TObject);
begin
  if Assigned(ListView1.Selected) then
    ListView1.Items.Delete(ListView1.Selected.Index);
end;

procedure TMainForm.acNewExecute(Sender: TObject);
var
  Value: string;
begin
  if InputQuery('TODO', 'Write your new TODO', Value) then
    AddItem(Value);
end;
```

7. Directly on the **ActionList1** component, create the `OnUpdate` event handler and fill it with the following code. This code makes the Delete button invisible when no item is selected on the list.

```
procedure TMainForm.ActionList1Update(Action: TBasicAction;
                             var Handled: Boolean);
begin
  acDelete.Visible := Assigned(ListView1.Selected);
end;
```

8. Go to the main form declaration and in the `private` section, declare the following variables:

```
private
  FDataFileName: String;
  procedure LoadFromFile;
  procedure SaveToFile;
  procedure AddItem(const TODO: String);
```

9. Hit *Ctrl* + *Shift* + *C* and fill the method bodies with the following code:

```
procedure TMainForm.LoadFromFile;
var
  FileReader: TStreamReader;
begin
  ListView1.ClearItems;
  if TFile.Exists(FDataFileName) then
  begin
    FileReader := TFile.OpenText(FDataFileName);
    try
```


www.allitebooks.com

```
      while not FileReader.EndOfStream do
      begin
        AddItem(FileReader.ReadLine);
      end;
    finally
      FileReader.Close;
    end;
  end;
end;

procedure TMainForm.SaveToFile;
var
  item: TListViewItem;
  FileWriter: TStreamWriter;
begin
  FileWriter := TFile.CreateText(FDataFileName);
  try
    for item in ListView1.Items do
    begin
      FileWriter.WriteLine(item.Text);
    end;
  finally
    FileWriter.Close;
  end;
end;

procedure TMainForm.AddItem(const TODO: String);
var
  item: TListViewItem;
begin
  item := ListView1.Items.Add;
  item.Text := TODO;
  ListView1.ItemIndex := item.Index;
end;
```

10. As you can see, the name of the file used to store the data is in the `FDataFileName` variable.
11. Create the `OnCreate` and `OnClose` event handlers for the form:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FDataFileName := TPath.Combine(
         TPath.GetDocumentsPath, 'datafile.txt');
  LoadFromFile;
end;
procedure TMainForm.FormClose(Sender: TObject;
                            var Action: TCloseAction);
```

```
begin
  SaveToFile;
end;
```

12. The last thing to do is to connect the `acNew` and `acDelete` actions to two buttons. Drop two **TButton** components on the lower **TToolbar** named **Footer**, name them `btnDelete` and `btnNew`, and set the following properties:

    ```
    object btnDelete: TButton
      Action = acDelete
      Align = alLeft
      StyleLookup = 'trashtoolbutton'
    end
    object btnNew: TButton
      Action = acNew
      Align = alRight
      StyleLookup = 'additembutton'
    end
    ```

13. Run the app. For testing purposes, you can run the app using the **Mobile Preview** option.

How it works...

When the app starts, it looks into its documents path for a file named `datafile.txt`. If it exists, it is loaded and all the lines become items in the listview. Remember that Delphi allows you to write cross-platform applications, so you must be aware about the way Delphi allows you to normalize the differences between operating systems; otherwise, the risk is that you could frustrate the Delphi and FireMonkey power. The `TPath` class is useful to be ignorant about system default paths, path separator, and other stuff related to the filesystem. We want to put our data into the `documents` folder. However, in Android, the `document` folder is different from the iOS one (and if your code has to run in the desktop environment as well, the paths are also different). So, using the `TPath` class, we can be completely ignorant about where the file is actually stored. We can know the path, but we don't want to explicitly define it; let `TPath` do its job. These are some well-known paths that `TPath` already knows. Whenever you need the specific path, ask `TPath`:

```
class function GetHomePath: string; static;
class function GetDocumentsPath: string; static;
class function GetSharedDocumentsPath: string; static;
class function GetLibraryPath: string; static;
class function GetCachePath: string; static;
class function GetPublicPath: string; static;
class function GetPicturesPath: string; static;
class function GetCameraPath: string; static;
class function GetMusicPath: string; static;
```


www.allitebooks.com

```
class function GetMoviesPath: string; static;
class function GetAlarmsPath: string; static;
```

Let's go back to our app. When the items are loaded into the listview, the `acNew` and `acDelete` actions allow the user to add and remove items from the list. When the form is about to close, the `FormClose` event saves all the items—one item for a line—into the `datafile.txt` file.

In a more complex situation, it is much better to have an in-memory representation of your data model not bound to any visual control. Suppose you need to access the data in another form. How to do that? If your data are bound to the GUI, you are bound to it too! The state of your app should not be stored only on the visual controls. However, for a simple situation like this recipe, it is not a big problem.

There's more...

Starting from XE5 Update 2, a very useful tool is available to debug mobile apps: **Mobile Preview**. More information about **Mobile Preview** can be found at `http://docwiki.embarcadero.com/RADStudio/XE5/en/Mobile_Preview`.

Do not block the main thread!

Long requests to external systems such as storage, databases, hardware, and network have always been difficult to handle from a user experience point of view. For the programmers, it is simple to run the long request and when finished (after seconds, minutes, or hours), inform the user that their data are there. However, we should care about user experience, even more in the mobile world.

Getting ready

If your app runs a long-running request and the UI is frozen, the user might think that something is going wrong and start to tap here and there to try to unblock the app. After some seconds, if not the operating system itself, the user will push the **Home** button to close your app and then, usually, uninstall it. Yes, the user experience is one of the most important things on the mobile. Consider that, also on a desktop, the user experience should be of primary importance, but what I want to emphasize is this: while on desktop you may have patient users because they are sitting in front of a PC (or a Mac), on the mobile you certainly have impatient users who want immediate feedback from your app. Mobile apps can be used in mobility, so the user may be busy doing something else while they are using your app, so the app must be fast and should give feedback as soon as possible. If some long operation is running, the app should inform the user and the GUI should never get frozen. In this recipe, we'll see not how to have 0 seconds latency, but how to inform the user that something completely regular is going on and that the app is actually working as expected, and so, the only thing that the user should do is to wait!

How to do it...

The scenarios exposed in this recipe are very frequent, so this demo will have to face with real timings and real problems. We'll do, as long-running request, a REST call to an open web service that provides weather forecasts. I've used this app for a while and the forecasts even seem accurate! Cool!

The service is provided by `OpenWeatherMap.org` and we'll issue the REST request at the following endpoint:

```
http://api.openweathermap.org/data/2.5/forecast
```

All the parameters required for the request will be defined at runtime by the app. Let's start by creating the app.

1. Create a new mobile app by navigating to **File** | **New** | **Other...** | **Delphi Projects** | **FireMonkey Mobile Application**.
2. Choose the **Header/Footer** template and click on **OK**.
3. As soon as Delphi creates the project template, save all the files with the following names:
    - Save the project as `WeatherForecasts.dproj`
    - Save the form as `MainFormU.pas`
4. Drop a **TPanel** component just below the header toolbar and align it to **alTop**.
5. Into the panel just dropped, drop two **TEdits** components and a **TButton** component and name them `EditCity`, `EditCoutry`, and `btnGetForecasts` respectively. Then, set the other properties as shown in the following code:

```
object EditCity: TEdit
  Align = Left
  Text = 'Rome'
  TextPrompt = 'City'
end
object EditCountry: TEdit
  Align = Right
  Text = 'IT'
  TextPrompt = 'Country'
end
object btnGetForecasts: TButton
  Align = Right
  Text = 'btnGetForecasts'
  StyleLookup = 'refreshtoolbutton'
end
```


www.allitebooks.com

6. Drop a **TAniIndicator** component into the header toolbar and align it to **alRight**.
7. Drop a **TListView** component on the form's center and set the following properties (the relevant properties extracted from the `MainFormU.fmx` file):

```
object ListView1: TListView
  AllowSelection = False
  Align = Client
  ItemAppearanceObjects.ItemObjects.Text.WordWrap = True
  ItemAppearanceObjects.ItemObjects.Text.Height = 50
  ItemAppearanceObjects.ItemObjects.Accessory.Visible = False
  CanSwipeDelete = False
end
```

8. Drop a **TLabel** component into the footer toolbar, align it to **alClient**, and name it `lblInfo`.
9. Drop the **TRESTClient** and **TRESTResponse** components and leave the default properties and names.
10. Your form at design time should look like the following:

The Weather forecasts form at design time

11. Now, let's write some code. In the `private` section of the form, declare a string instance field called `Lang`.

12. Create the `FormCreate` event handler and fill it with the following code:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  LocaleService: IFMXLocaleService;
begin
  if TPlatformServices.Current.SupportsPlatformService(
                                  IFMXLocaleService) then
  begin
    LocaleService :=
      TPlatformServices.Current.GetPlatformService(
      IFMXLocaleService) as IFMXLocaleService;
    Lang := LocaleService.GetCurrentLangID;
  end
  else
    Lang := 'US';

  EditCountry.Text := Lang;
  RESTClient1.BaseURL := 'http://api.openweathermap.org/data/2.5';
  RESTRequest1.Resource :=
       'forecast?q={country}&mode=json&lang={lang}&units=metric';
  AniIndicator1.Visible := False;
end;
```

13. In the `implementation` section of `uses`, add the following units:

```
uses
  Data.DBXJSON, System.DateUtils, FMX.Platform;
```

14. Create an `OnClick` event handler for the **btnGetForecasts** button and fill it with the following code:

```
procedure TMainForm.btnGetForecastsClick(Sender: TObject);
begin
  ListView1.ClearItems;
  RESTRequest1.Params.ParameterByName('country').Value :=
    String.Join(',', [EditCity.Text, EditCountry.Text]);
  RESTRequest1.Params.ParameterByName('lang').Value := Lang;
  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;

  //Run the REST request in asynch mode. When ExecuteAsynch
  //returns, the anonymous method passed as parameter is called in
  //the main thread context
  RESTRequest1.ExecuteAsync(
    procedure
    var
```


www.allitebooks.com

```
  dt: TDateTime;
  jv: TJSONValue;
  JObj, MainForecast, ForecastItem, JObjCity: TJSONObject;
  Weather, Forecasts: TJSONArray;
  TempMin, TempMax: Double;
  WeatherDescription, Day, LastDay, AppRespCode: string;
  Item: TListViewItem;
begin
  JObj := RESTRequest1.Response.JSONValue as TJSONObject;

  // check for errors
  AppRespCode := JObj.GetValue('cod').Value;
  if AppRespCode.Equals('404') then
  begin
    lblInfo.Text := 'City not found';
    Exit;
  end;
  if not AppRespCode.Equals('200') then
  begin
    lblInfo.Text := 'Error ' + AppRespCode;
    Exit;
  end;

  // parsing response...
  Forecasts := JObj.GetValue('list') as TJSONArray;
  for jv in Forecasts do
  begin
    ForecastItem := jv as TJSONObject;
    dt := UnixToDateTime((ForecastItem.GetValue('dt')
                         as TJSONNumber).AsInt64);
    MainForecast := ForecastItem.GetValue('main')
                         as TJSONObject;
    TempMin := (MainForecast.GetValue('temp_min')
                         as TJSONNumber).AsDouble;
    TempMax := (MainForecast.GetValue('temp_max')
                         as TJSONNumber).AsDouble;
    Weather := ForecastItem.GetValue('weather')
                         as TJSONArray;
    WeatherDescription := TJSONObject(Weather.Items[0])
                    .GetValue('description').Value;
    Day := DateToStr(DateOf(dt));
    if Day <> LastDay then
    begin
```

```
          Item := ListView1.Items.Add;
          Item.Purpose := TListItemPurpose.Header;
          Item.Text := Day;
        end;
        LastDay := Day;
        Item := ListView1.Items.Add;
        Item.Text := FormatDateTime('HH', dt) + ' '
          + WeatherDescription
          + Format(' (min %2.2f max %2.2f)', [TempMin, TempMax]);
      end;

      // display the city name at the bottom
      JObjCity := JObj.GetValue('city') as TJSONObject;
      lblInfo.Text := JObjCity.GetValue('name').Value + ', ' +
        JObjCity.GetValue('country').Value;

      // stop the waiting animation
      AniIndicator1.Visible := False;
      AniIndicator1.Enabled := False;
    end);
end;
```

15. The parsing code is not simple, but now you should have all the information needed to correctly understand what's going on with this code.
16. Hit *F9* and see the application running.
17. Insert a city name and a state code (such as `Roma` and `IT`), and you will get the weather forecasts for the upcoming days organized day by day.

How it works...

This recipe is simple from an architectural point of view. There are two parameters the user can enter. These parameters affect the request to the server that will respond with a JSON structure. Apart from the parsing code, the interesting things happen when the request is sent to the server. If we had sent a normal synchronous request to the server, the UI would be blocked until the response arrives to the client. Using the `ExecuteAsynch` method executes the actual request on a background thread so that the main thread remains free to update the UI. When the request finishes the execution, an anonymous method is called in the main thread context. The **TAniIndicator** component is started just before the request starts and is stopped after the parsing is finished. In this way, the user is aware that something is happening. Consider that any request to an external system could potentially last for hours. Be aware of this!

The code used to fill the list uses the grouping feature of the **TListView** component to show the forecasts organized day by day.

www.allitebooks.com

Another thing to note is that the web service can use localized response for descriptive texts. So, in the `FormCreate` event, we use the `IFMXLocaleService` service to read the current system language. Later, we use that language code to inform the remote service about the preferred localization language.

Here's the app running in the mobile preview on an Italian PC:

The Weather forecasts app running in the Mobile Preview on an Italian Windows PC

There's more...

Multithreading can be difficult, but the built-in features in the REST client library allow you to send HTTP requests in a background thread in a very simple manner. You can use it as much as you can. If you are not so confident with the new REST client library, here's some documentation:

- *Delphi XE5 Mobile REST Client Demo* at `https://www.youtube.com/watch?v=OkRVbgF4VMI`
- *REST Client Library* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/REST_Client_Library`

Another topic that should be deeply understood to correctly design and implement FireMonkey applications (and mobile apps are only a particular type of FireMonkey application) is the FireMonkey platform services. More info on platform services can be found at: `http://docwiki.embarcadero.com/RADStudio/XE6/en/FireMonkey_Platform_Services`.

Using SQLite databases to handle a to-do list

Usually, the mobile apps read or write data using the network. In many cases, however, you need a local storage to save your data. A local database can be useful for a number of things:

- To buffer information while the Internet connection is not available
- To save information that will be realigned on the central server when back to the office
- To allow you a fast search on a relatively small set of data retrieved from the central databases and stored on the device
- To store some structured data.

In all these cases, you have to handle a database. This recipe will show you how to do it.

Getting ready

This recipe is about a to-do list. It is similar to the *Using listview to show and search local data* recipe, but in this case, we'll use a SQL database and show data to the user using LiveBindings. Moreover, we'll see how to create output converters for LiveBindings.

How to do it...

When you need a database on the mobile, you have two choices in Delphi: SQLite (an open source embedded database) and InterBase ToGo.

RAD Studio XE6 includes InterBase XE3 ToGo and IBLite editions for embedded application development. You can deploy your mobile applications to iOS or Android devices with a InterBase ToGo license (at a cost) or IBLite license (free).

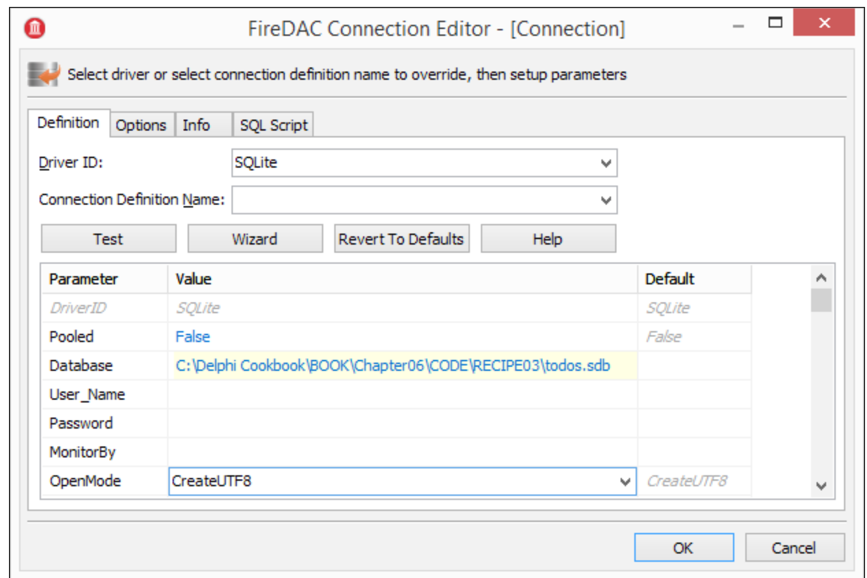
If your app is a bit complex or you need encryption, stored procedures, a number of data types, you have to definitely go for InterBase ToGo. Otherwise, you can use SQLite. Consider that IBLite is the same engine as InterBase ToGo, but limited on some extent. The biggest limit is the lack of encryption. However, an app that uses IBLite doesn't require updates if you need to scale to InterBase ToGo (change the license and you are okay).

This recipe is very simple in terms of database requirements, so we'll use SQLite. However, the same concepts are applicable to InterBase ToGo and IBLite.

Open the `TODOList.dproj` project. The main form has all the components that are required to access the database (in a real-world app, consider to use a data module for this, just like the desktop applications). The app has been created using the **Header/Footer** mobile template. The first `TabItem` contains the to-do lists, while the second `TabItem` allows you to update an existing to-do list or create a new to-do list.

www.allitebooks.com

When the application starts, the **TFDConnection** components connect to the database. If the database file doesn't exist, the SQLite engine is configured to create a brand new database file. This feature is very useful and can be configured by setting the `OpenMode` parameter to `CreateUTF8`. (The UTF8 encoding is almost always the best choice for international applications; in this case, it is the default setting for the **TFDConnection** components). Here's the relevant part of the **TFDConnection** parameters:



The connection parameters

Another problem to solve is related to the database path. In Windows, you can develop your mobile app using the **Mobile Preview** and a local path on your system; however, when the app runs on the device, you have to use another path. How to solve this? In the connection `BeforeConnect` event handler, the following code is the solution:

```
procedure TMainForm.ConnectionBeforeConnect(Sender: TObject);
begin
{$IF DEFINED(IOS) or DEFINED(ANDROID)}
  Connection.Params.Values['Database'] :=
         TPath.GetDocumentsPath + PathDelim + 'todos.sdb';
{$ENDIF}
end;
```

With this code, on the mobile, the database will be created in the proper iOS or Android `document` folder.

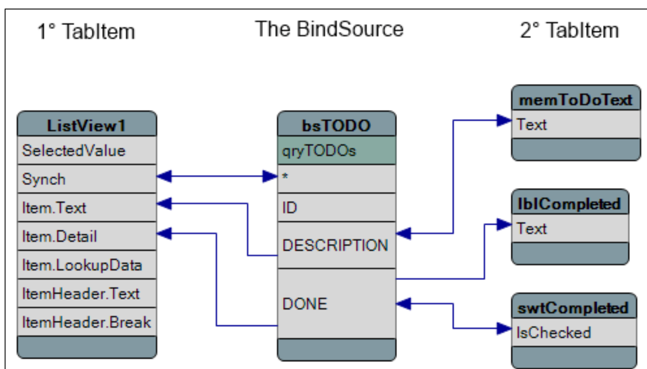
The next problem is related to the database structure. When and how to create the table that we need? Let's check the `AfterConnect` event handler on the connection:

```
procedure TMainForm.ConnectionAfterConnect(Sender: TObject);
begin
  Connection.ExecSQL('CREATE TABLE IF NOT EXISTS TODOS( ' +
    ' ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, ' +
    ' DESCRIPTION    CHAR(50) NOT NULL, ' +
    ' DONE           INTEGER  NOT NULL  ' +
    ')');
  qryTODOs.Active := True;
end;
```

Just after the database is created, and at any subsequent run, the app tries to create the database table if it doesn't exist yet. Then, open the dataset connected to the bind source to show the data present. The listview is configured with the following code:

```
ItemAppearance.ItemAppearance = 'ListItemRightDetail'
ItemAppearance.ItemHeight = 100
SearchVisible = True
```

The second tab contains a **TMemo** component, a **TSwitch** component, and two **TLabel** components. The `TBindSourceDB` data source connected to the `qryTODO` dataset is connected to the list and to the `detail` component placed on the second `TabItem` as well. This is shown in the following screenshot (integrated with some clarifying text):

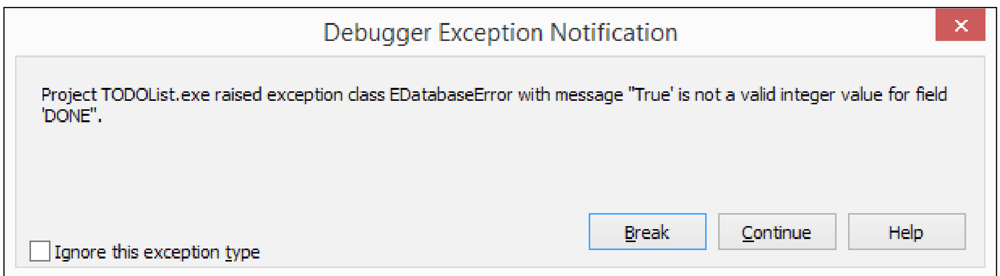


The LiveBindings designer showing the binding connections between the Bindsource, the listview, and the detail components

All the code used to handle the dataset is normal dataset-oriented code, just like the code used to manage datasets on a desktop application.

www.allitebooks.com

This recipe shows a nasty problem. SQLite doesn't have the Boolean field type, so the **DONE** field in the **TODO** table is of type integer, where `1` means `true` and `0` means `false`. However, we want to connect the **DONE** field to a `TSwitch.IsChecked` property of type Boolean. In this situation, when you try to change the switch value, you will get an error like the following:



The exception raised by Delphi when you try to connect an Integer field to a Boolean component property

How to solve this? The LiveBindings engine has a powerful mechanism to convert data from one type to another. When a result value is of type X and the property where that value needs to be written is of type Y, the engine looks for a valid output converter that is able to convert type X to type Y. The available output converters are shown on the `BindingList1.OutputConverters` property.

To solve our problem, we've to register another `OutputConverter` object able to convert a Boolean value (`swtCompleted.IsChecked`) to a string value (because LiveBindings use the `TField.SetText` method to set a value of a field). This output converter is registered in the `BoolToStringConverterU.pas` unit. The procedure is used to register the new converter and makes it visible to the LiveBindings engine, as shown in the following code:

```
const
  sBoolToString = 'BoolToString';

procedure RegisterOutputConversions;
begin
  //unregister the default converter bool->string
  TValueRefConverterFactory.UnRegisterConversion(
TypeInfo(Boolean), TypeInfo(String));

  //register the new converter bool->string
  //This converter is able to handle 1=true and 0=false
  TValueRefConverterFactory.RegisterConversion(
         TypeInfo(Boolean), TypeInfo(String),
           TConverterDescription.Create(
        procedure(const InValue: TValue; var OutValue: TValue)
        begin
          if InValue.AsBoolean then
```

```
              OutValue := '1'
            else
              OutValue := '0';
          end, sBoolToString, sBoolToString, '',
                              True, sBoolToString, nil));
end;
```

Now the app works correctly. However, be careful, now all the conversion from Boolean to string will consider true when `1` and false when `0`. This internal mechanism of LiveBindings needs to be clearly understood, because it can cause a lot of headache if not in trivial cases.

On the second tab, there is a label that describes the meaning of the switch. When the switch is checked, the label says **The task is completed**; otherwise, it says **The task is not completed**. This feature has been implemented using LiveBindings expressions. Go to the LiveBindings designer and select the arrow that connects the **DONE** field to the `lblCompleted.Text` property. Now hit *F11* to show the **Object Inspector** window and check the `CustomFormat` property. Here, a logic used by the label. The expression is reported as follows:

```
"The task is " + IfThen(value = 1, "completed","not completed")
```

This code is a relational expression that transforms a value read from a dataset field to a text value shown in a label. Normally, the value is read from the source component and written on the target property component. However, using the `CustomFormat` property, you can change this default behavior to get more complex and useful information. This expression is a good example of that.

There's more...

As you can see, mobile development is a mix of well-known things and new things. The LiveBindings framework is a big new thing, and you can be frightened by it. However, don't be afraid, all your needs are there. Here are some useful links to go deeper within the concepts exposed in this recipe:

- Another approach to the Integer-as-Boolean problem can be found at `http://www.malcolmgroves.com/blog/?p=1490`
- Information on formatting fields using LiveBindings can be found at `http://www.malcolmgroves.com/blog/?p=1226`
- Documentation about output converters can be found at `http://docwiki.embarcadero.com/RADStudio/XE6/en/LiveBindings_Output_Converters`
- Some tutorials on LiveBindings in RAD Studio can be found at `http://docwiki.embarcadero.com/RADStudio/XE6/en/LiveBindings_in_RAD_Studio`

www.allitebooks.com

Using a styled TListView to handle a long list of data

The **TListBox** control is very flexible. You can customize every aspect of each item in the list. However, it is not suitable if you want to handle a long list of data, because flexibility comes at the cost of the system being slow when data rows grow. Embarcadero specifies that you should use **TListView** to display a collection of items in a list that is optimized for LiveBindings and for fast and smooth scrolling.

Getting ready

In this recipe, we'll use the *Do not block the main thread!* recipe as a base to customize a listview using custom styles. In that recipe, we get a list of weather forecasts from a REST web service and then fill the listview with that data using a standard style. In this recipe, that data will be nicely inserted in a custom listview with colors, alignment, and summary footer. There is no design-time support with this approach, because all the controls created into each item are created at runtime; however, this approach can be very useful if you want complete control over the look and feel of your list. To be clear, the recommended approach in this case is to write a custom style for the **TListView** component; put the component in a package, install it into the IDE, and then use it from the **Object Inspector** window. To have two samples of this approach, check the following projects provided as samples (the `Sample` folder on my machine is `C:\Users\Public\Documents\Embarcadero\Studio\14.0\Samples` where `14.0` is the version of the IDE).

Within the `Sample` folder, open `Object Pascal\Mobile Samples\User Interface\ListView\`.

In this folder, you have a number of projects and packages that show you how to use some advanced stuff related the **TListView** components. To see the new style, you have to install the package and open the related demo project.

The package to install the `RatingListItem` list item style is `SampleListViewRatingsAppearancePackage.dproj`.

The project that shows how to use the `RatingListItem` style is `SampleListViewRatingsAppearanceProject.dproj`.

The package to install the `MultiDetailItem` list item style is `SampleListViewMultiDetailAppearancePackage.dproj`.

The project that show how to use the `MultiDetailItem` style is `SampleListViewMultiDetailAppearanceProject.dproj`.

It is not too complex create a custom list item style; however, there are these two samples provided by Embarcadero that should be enough to start with. In this recipe, we'll create the list item style element directly in the code. When you are satisfied by the result, you can create the proper package as shown in the mentioned samples.

How to do it...

1. Copy the code of the *Using SQLite databases to handle a to-do list* recipe into a new folder (or start reading from the beginning of the *Using SQLite databases to handle a to-do list* recipe).
2. Open the project and save it as `WeatherForecastsEx.dproj`.
3. In the `private` section of the form declaration, add the following methods:

```
procedure SetupFooter(Item: TListViewItem;
         MinInTheDay, MaxInTheDay: Double);
procedure SetupHeader(Item: TListViewItem; TextLabel: String);
procedure SetupItem(Item: TListViewItem; dt: TDate;
    Description: String; TempMin, TempMax: Double);
```

4. Press *Ctrl* + *Shift* + *C* to create the method bodies, and then add the following code:

```
procedure TMainForm.SetupFooter(Item: TListViewItem;
               MinInTheDay, MaxInTheDay: Double);
var
  ItemSeparator: TListViewItem;
begin
  Item.Purpose := TListItemPurpose.Footer;
  Item.Text := Format('min %2.2f max %2.2f',
               [MinInTheDay, MaxInTheDay]);
  //separator
  ItemSeparator := Item.Parent.Items.Add;
  ItemSeparator.Height := 10;
  ItemSeparator.Purpose := TListItemPurpose.Footer;
end;

procedure TMainForm.SetupHeader(Item: TListViewItem;
                                       TextLabel: String);
begin
  Item.Purpose := TListItemPurpose.Header;
  Item.Text := TextLabel;
end;

procedure TMainForm.SetupItem(Item: TListViewItem;
               dt: TDate; Description: String;
               TempMin, TempMax: Double);
```


www.allitebooks.com

```
var
  lb: TListItemText;
begin
  Item.Objects.Clear;
  Item.Height := 24;

  lb := TListItemText.Create(Item);
  lb.PlaceOffset.X := 0;
  lb.TextAlign := TTextAlign.Leading;
  lb.Name := 'WeatherDescription';

  lb := TListItemText.Create(Item);
  lb.TextAlign := TTextAlign.Trailing;
  lb.TextColor := TAlphaColorRec.Blue;
  lb.Name := 'MinTemp';

  lb := TListItemText.Create(Item);
  lb.TextAlign := TTextAlign.Trailing;
  lb.TextColor := TAlphaColorRec.Red;
  lb.Name := 'MaxTemp';

  Item.Data['WeatherDescription'] :=
            FormatDateTime('HH', dt) + ' ' + Description;
  Item.Data['MinTemp'] := Format('m %2.2f', [TempMin]);
  Item.Data['MaxTemp'] := Format('M %2.2f', [TempMax]);
end;
```

5. Now we've to use these methods. In the `btnGetForecastsClick` method, substitute the code with the following:

```
procedure TMainForm.btnGetForecastsClick(Sender: TObject);
begin
  ListView1.ClearItems;
  RESTRequest1.Params.ParameterByName('country').Value :=
    String.Join(',', [EditCity.Text, EditCountry.Text]);
  RESTRequest1.Params.ParameterByName('lang').Value := Lang;
  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;
  RESTRequest1.ExecuteAsync(
    procedure
    var
      dt: TDateTime;
      jv: TJSONValue;
      JObj, MainForecast, ForecastItem, JObjCity: TJSONObject;
      Weather, Forecasts: TJSONArray;
```

```
    TempMin, TempMax, MinInTheDay, MaxInTheDay: Double;
    Day, LastDay, WeatherDescription: string;
    Item: TListViewItem;
  begin
    JObj := RESTRequest1.Response.JSONValue as TJSONObject;

    // check for errors
    if JObj.GetValue('cod').Value = '404' then
    begin
      Label1.Text := 'City not found';
      Exit;
    end;

    if JObj.GetValue('cod').Value <> '200' then
    begin
      Label1.Text := 'Error ' + JObj.GetValue('cod').Value;
      Exit;
    end;

    // parsing forecasts
    MinInTheDay := 1000;
    MaxInTheDay := -MinInTheDay;
    Forecasts := JObj.GetValue('list') as TJSONArray;
    for jv in Forecasts do
    begin
      ForecastItem := jv as TJSONObject;
      dt := UnixToDateTime((ForecastItem.GetValue('dt')
                as TJSONNumber).AsInt64);
      MainForecast := ForecastItem.GetValue('main')
                as TJSONObject;
      TempMin := (MainForecast.GetValue('temp_min')
                as TJSONNumber).AsDouble;
      TempMax := (MainForecast.GetValue('temp_max')
                as TJSONNumber).AsDouble;
      Weather := ForecastItem.GetValue('weather') as TJSONArray;
      WeatherDescription :=
        TJSONObject(Weather.Items[0]).
              GetValue('description').Value;
      Day := DateToStr(DateOf(dt));
      if Day <> LastDay then
      begin
        if not LastDay.IsEmpty then
          SetupFooter(ListView1.Items.Add,
```


www.allitebooks.com

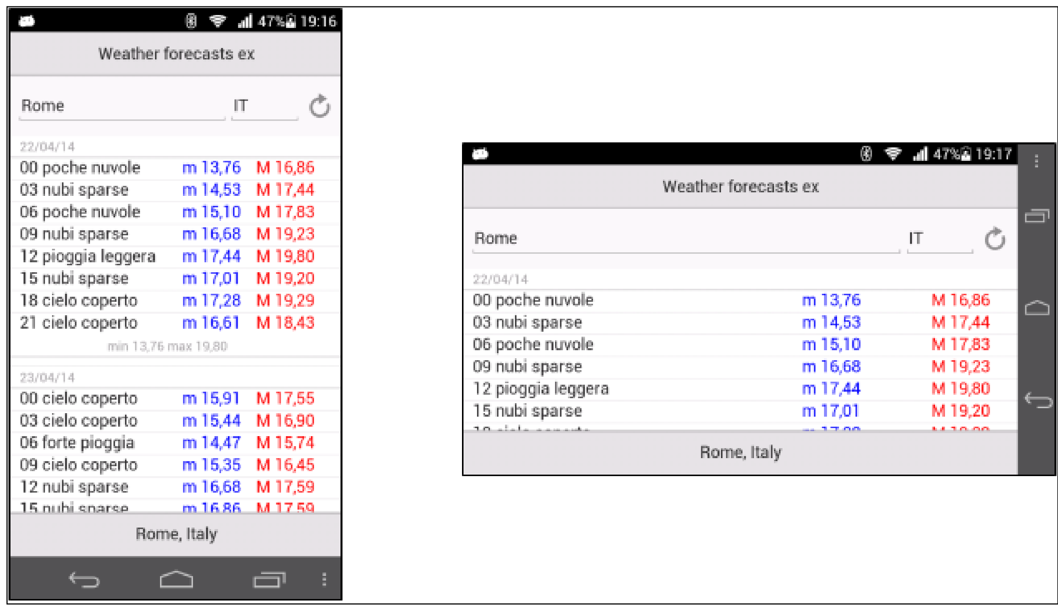
```
                        MinInTheDay, MaxInTheDay);
          SetupHeader(ListView1.Items.Add, Day);
          MinInTheDay := 1000;
          MaxInTheDay := -MinInTheDay;
        end;
        LastDay := Day;
        Item := ListView1.Items.Add;
        MinInTheDay := Min(MinInTheDay, TempMin);
        MaxInTheDay := Max(MaxInTheDay, TempMax);
        SetupItem(Item, dt, WeatherDescription, TempMin, TempMax);
      end; // for in

      if not LastDay.IsEmpty then
        SetupFooter(ListView1.Items.Add,
                    MinInTheDay, MaxInTheDay);

      JObjCity := JObj.GetValue('city') as TJSONObject;
      Label1.Text := JObjCity.GetValue('name').Value + ', '
                     + JObjCity.GetValue('country').Value;
      AniIndicator1.Visible := False;
      AniIndicator1.Enabled := False;
    end);
end;
```

6. The main difference between the *Using SQLite databases to handle a to-do list* recipe and this recipe is the complete flexibility of data visualization. To get this flexibility, we added individual controls to each list item. We defined all the needed properties, width, alignment, colors, and so on. When the device goes into landscape orientation, some alignment needs to be changed according to the larger horizontal space available. For this situation, a very handy listbox `UpdateObjects` event handler is available. Create an `UpdateObjects` event handler on the listbox and add this code:

```
procedure TMainForm.ListView1UpdateObjects(const Sender: TObject;
                                   const AItem: TListViewItem);
var
  MinTemp, MaxTemp: TListItemText;
  AQuarter: Double;
begin
  AQuarter := (AItem.Parent.Width -
               AItem.Parent.ItemSpaces.Left -
               AItem.Parent.ItemSpaces.Right) / 4;

  MinTemp := AItem.Objects.FindObject('MinTemp') as TListItemText;
  if Assigned(MinTemp) then
  begin
```

```
    MinTemp.PlaceOffset.X := AQuarter * 2;
    MinTemp.Width := AQuarter;
  end;

  MaxTemp := AItem.Objects.FindObject('MaxTemp') as TListItemText;
  if Assigned(MaxTemp) then
  begin
    MaxTemp.PlaceOffset.X := AQuarter * 3;
    MaxTemp.Width := AQuarter;
  end;

  if AItem.Purpose = TListItemPurpose.Footer then
    AItem.Objects.TextObject.TextAlign := TTextAlign.Center;
end;
```

7. With this adjustment, texts inside the list item are always aligned correctly.
8. Run the app. For testing purposes, you can run the app using the **Mobile Preview** option. Here's the app running on an Android phone:



The Weather forecasts ex app running in portrait and in landscape modes on an Italian Android phone; note how the temperature columns are realigned between the two orientations

www.allitebooks.com

How it works...

After reading the JSON using the `TRESTClient` component, in the parsing code, we added controls to each item to represent the three columns we require. You cannot add every kind of control into the **TListViewItem** component, so add only those that inherit from **TListItemObject**. However, you can inherit your own class from **TListItemObject** to implement all the advanced visualizations you require.

The relevant part about the customization happens in the `SetupItem` method:

```
procedure TMainForm.SetupItem(Item: TListViewItem; dt: TDate;
               Description: String; TempMin, TempMax: Double);
var
  lb: TListItemText;
begin
  Item.Objects.Clear; //remove objects already created
  Item.Height := 24;

  lb := TListItemText.Create(Item);  //add the first column
  lb.PlaceOffset.X := 0;
  lb.TextAlign := TTextAlign.Leading;

  //name the control for future addressing
  lb.Name := 'WeatherDescription';


  lb := TListItemText.Create(Item);  //add the second column
  lb.TextAlign := TTextAlign.Trailing;
  lb.TextColor := TAlphaColorRec.Blue;
  //name the control for future addressing
  lb.Name := 'MinTemp';

  lb := TListItemText.Create(Item);  //add the third column
  lb.TextAlign := TTextAlign.Trailing;
  lb.TextColor := TAlphaColorRec.Red;
  //name the control for future addressing
  lb.Name := 'MaxTemp';

  //Write to the Data property of component 'WeatherDescription'
  Item.Data['WeatherDescription'] :=
                        FormatDateTime('HH', dt) + ' ' +
Description;

  //Write to the Data property of component 'MinTemp'
  Item.Data['MinTemp'] := Format('m %2.2f', [TempMin]);

  //Write to the Data property of component 'MaxTemp'
  Item.Data['MaxTemp'] := Format('M %2.2f', [TempMax]);
end;
```

Here, as a first thing, all the already present controls in the items are removed using `Item.Objects.Clear`. Then, all the new items are created and added to the `ListViewItem` instance. Note that each control has a name. This name is used by the last three lines to write text into the `Data` property of the control. So, just to be clear, if in the item you add a **TListItemText** control named `MinTemp`, you can use `Item.Data['MinTemp']` to read and write (it depends on the actual object, but technically it is possible) generic data on the `MinTemp.Data` property. As you know, the `Data` property is handled by each control with a different meaning. In this specific case, all the `Data` properties represent the text written in the controls.

Then, the problem related to the orientation change is handled by the very useful `UpdateObjects` event on the listview. Here, we organize the horizontal space to split it into four columns and give the first two columns to the weather description, the third to the minimum temperature, and the fourth to the maximum temperature. You can organize all the cool things you need in this event, because it's called every time there is an update in the objects visualization. The code to perform these actions is as follows:

```
procedure TMainForm.ListView1UpdateObjects(const Sender: TObject;
          const AItem: TListViewItem);
var
  MinTemp, MaxTemp: TListItemText;
  AQuarter: Double;
begin
  //calculate the real available horizontal space
  //for the list item and divide it by 4
  AQuarter := (AItem.Parent.Width -
               AItem.Parent.ItemSpaces.Left -
               AItem.Parent.ItemSpaces.Right) / 4;

  //if MinTemp is created define its new offset and width
  MinTemp := AItem.Objects.FindObject('MinTemp') as TListItemText;
  if Assigned(MinTemp) then
  begin
    MinTemp.PlaceOffset.X := AQuarter * 2;
    MinTemp.Width := AQuarter;
  end;

  //if MaxTemp is created define its new offset and width
  MaxTemp := AItem.Objects.FindObject('MaxTemp') as TListItemText;
  if Assigned(MaxTemp) then
  begin
    MaxTemp.PlaceOffset.X := AQuarter * 3;
    MaxTemp.Width := AQuarter;
  end;
```


www.allitebooks.com

```
    //Set all the footers with centered alignment
    if AItem.Purpose = TListItemPurpose.Footer then
      AItem.Objects.TextObject.TextAlign := TTextAlign.Center;
  end;
```

There's more...

Listviews are tremendously helpful in mobile development and you must be familiar with them to implement good looking and efficient apps. Using the custom list item style in a package, you also get the LiveBindings support, while the solution exposed in this recipe doesn't provide this support. Consider developing a custom list item style and package it in a package if you want the design-time support. This recipe gives you the starting point to develop custom style for the listview items. When you are satisfied by the result, create the proper package as shown in the samples provided by Embarcadero.

Taking a photo and location and sending it to a server continuously

In this recipe, we'll talk about a lot of things. We'll see how to continuously get an image from the camera, how to get location information, and how to send binary data to a web server. Then, moving on to the server side, we'll see how to read binary data from the client and how to generate content on the fly. All these things will be used to implement a simple monitoring system.

Getting ready

This recipe is divided into client and server sides. The client side is a mobile app acting as a *special* camera able to get image and location and then send it to a remote server. There is also a live preview on the main form, so you can see what you are sending to the server. The server simply gets the information and stores them in the filesystem. This recipe is quite complex, so I avoided an actual SQL (or NoSQL) database to store all the information and used the filesystem.

How to do it...

Launch two instances of Delphi and open one project in each of them (this will help in the debug phase). The server project is named `MonitorServer.dproj`, while the client app is named `MonitorMobile.dproj`.

Let's start with the client side.

The client side

On the main form, there are the **TCameraComponent** and **TLocationSensor** components; the **TSwitch** control on the top of the form is used to activate them. As soon as the camera has enough data to create a frame, the **TCameraComponent** calls its `SampleBufferReady` event handler, and then the process begins. Here's the code in the `SampleBufferReady` event handler:

```
procedure TMainForm.CameraComponent1SampleBufferReady(
                      Sender: TObject; const ATime: Int64);
var
  Frame: TFrameInfo;
begin
  CameraComponent1.SampleBufferToBitmap(Image1.Bitmap, True);
  if SecondsBetween(now, FLastSent) >= 4 then
  begin
    Frame := TFrameInfo.Create;
    Frame.Bitmap := Image1.Bitmap;
    Frame.TimeStamp := now;
    Frame.Lat := CurrLocation.Latitude;
    Frame.Lon := CurrLocation.Longitude;
    FImagesQueue.PushItem(Frame);
    FLastSent := now;
  end;
end;
```

Listing 7.1

Information retrieved by the camera is converted into an actual bitmap using the handy `SampleBufferToBitmap` method provided by **TCameraComponent** itself. Now we've an image. But where does the location information come from? The `TLocationSensor` component has the `OnLocationChanged` event that is called whenever the actual location, considering the different ways to get the location (such as GPS, Wi-Fi, and GPS combined with Wi-Fi), actually changes. In the `LocationSensor1LocationChanged` procedure, we save the new location in a form field as shown in the following code:

```
procedure TMainForm.LocationSensor1LocationChanged(
   Sender: TObject; const OldLocation,
   NewLocation: TLocationCoord2D);
begin
  CurrLocation := NewLocation;
end;
```

Listing 7.2

www.allitebooks.com

Now, go back to *Listing 7.1*. The information is used to fill an instance of `TFrameInfo` and then this instance is pushed on a `TThreadedQueue<TFrameInfo>` property shared with a background thread. The main thread pushes the `TFrameInfo` instances into the queue, while the background thread reads the `TFrameInfo` instances, creates a proper HTTP request, and then sends it to the server. The `TFrameInfo` type contains all the information required by the server:

```
type
  TFrameInfo = class
  private
    { . . . some private declarations . . . }
  public
    constructor Create;
    property Bitmap: TBitmap read FBitmap write SetBitmap;
    property Lat: Double read FLat write SetLat;
    property Lon: Double read FLon write SetLon;
    property TimeStamp: TDateTime read FTimeStamp
                             write SetTimeStamp;
  end;
```

Listing 7.3

The complex stuff actually runs on the background thread. Let's see its `Execute` method:

```
procedure TImageSenderThread.Execute;
var
  Parameters: TIdMultiPartFormDataStream;
  HTTP: TidHTTP;
  FrameInfo: TFrameInfo;
  FileName, EncodedParams: string;

  procedure RemoveOldFiles;
  begin
    while FilesToDelete.Count > 1 do
    begin
      try
        TFile.Delete(FilesToDelete.First);
        FilesToDelete.Delete(0);
      except
        on E: EInOutError do
        begin
          FilesToDelete.Delete(0);
        end
        else
```

```
          begin
           // do nothing now, maybe the file is locked.
           //the file will be deleted the next time        end;
        end
      end;
    end;

  begin
    FilesToDelete := TList<String>.Create;
    HTTP := TidHTTP.Create(nil);
    HTTP.ConnectTimeout := 2000;
    HTTP.ReadTimeout := 1000;
    while not Terminated do
    begin
      if FImagesQueue.PopItem(FrameInfo) <> wrTimeout then
      begin
        FileName := TPath.ChangeExtension(
                            TPath.GetTempFileName, '.png');
        try
          //save image to file
          FrameInfo.Bitmap.SaveToFile(FileName);
          //prepare an HTTP request to send the image to the server
          Parameters := TIdMultiPartFormDataStream.Create;
          Parameters.AddFile('file', FileName, 'image/png');
          EncodedParams := Format('ts=%s&lat=%s&lon=%s', [
            FormatDateTime('YYYY-MM-DD HH-NN-SS',
                                  FrameInfo.TimeStamp),
            FormatFloat('##0.00000000', FrameInfo.Lat),
            FormatFloat('##0.00000000', FrameInfo.Lon)]);
          EncodedParams := TIdURI.ParamsEncode(EncodedParams);
          //send actual HTTP POST request
          HTTP.Post(MONITORSERVERURL + '/photo?' +
                          EncodedParams, Parameters);
          //add the file in the queue of
          //the files that needs to be deleted
          FilesToDelete.Add(FileName);
          RemoveOldFiles;
        except
          //the best way to handle this exception, and keep this
          //code simple, is to send the next frame.
          //The same approach of the video
          //streaming protocols: "in case of error, send the next
```


www.allitebooks.com

```
        //frame"
        //so, do nothing
      end;
    end;
  end;
end;
```

Listing 7.3

In the usual thread loop, we try to read the next `TFrameInfo` instance from the queue. If such instance is present, we create an HTTP request using a simple HTTP `POST` method with the image file in the `request` body and the other information in the `querystring` parameter. Then, in order to avoid filling the storage after some minutes of work, we use a mechanism to delete old files. See the `RemoveOldFiles` function in the recipe code for details. In this code, you can note some suppressed exceptions (`try..except` with an empty `except` block). Usually, this is not good. However, in this case, if we loose a frame for some reason, the best way to fix the problem is to send the next one. So in some places, the exceptions are suppressed because the next frame will solve the problem. Moreover, the threaded queue had a size of only two elements. If the main thread tried to append a third `FrameInfo` object in the queue, it is stopped for 500 milliseconds; if it still cannot append the data, that data is lost. This is one of the approaches available when you are dealing with queues: if the queue is full, new data is discarded until the queue consumes its current content. To save space and battery energy, the image is resized before sending. The actual resizing is done by the `TFrameInfo` object when the image taken by the camera is assigned to its `Bitmap` property, as shown in the following code:

```
procedure TFrameInfo.SetBitmap(const Value: TBitmap);
var
  Prop: Extended;
  LongerSide: Double;
begin
  FBitmap.Assign(Value);
  Prop := 1;
  LongerSide := Max(FBitmap.Width, FBitmap.Height);
  if LongerSide > 640 then
    Prop := 640 / LongerSide;
  FBitmap.Resize(Trunc(FBitmap.Width * Prop),
                 Trunc(FBitmap.Height * Prop));
end;
```

Listing 7.4

The server side

The serve side is a WebBroker project with only two actions configured, as shown in the following table:

Action name	PathInfo	HTTP method
`DefaultHandler`	`/`	`mtGet`
`waPhoto`	`/photo`	`mtPost`

Note that this recipe cannot be compiled with Delphi XE6 without Update 1 because the `ReqMulti.pas` unit is missing from the product (visit `http://qc.embarcadero.com/wc/qcmain.aspx?d=124366`), but this recipe can be compiled in XE7, XE3, XE4, and XE5 Update 2.

The `waPhoto` action receives the client request, reads the data, and saves them on the filesystem. This action saves two files:

- The actual image file as a `.png` image file
- Another file containing all the location information in JSON format

Here's the code for the `waPhoto` action

```
procedure TwmMain.wmMainwaPhotoAction(Sender: TObject; Request:
      TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  FS: TFileStream;
  fname: string;
  Lat, Lon: Double;
  Info: TJSONObject;

  procedure SaveInfoFile;
  begin
    Info := TJSONObject.Create;
    if TryStrToFloat(Request.QueryFields.Values['lat'], Lat) then
      Info.AddPair('lat', TJSONNumber.Create(Lat));
    if TryStrToFloat(Request.QueryFields.Values['lon'], Lon) then
      Info.AddPair('lon', TJSONNumber.Create(Lon));
    TFile.WriteAllText('images' + PathDelim + fname +
                                  '.info', Info.ToString);
  end;

begin
  if Request.Files.Count > 0 then
  begin
```


www.allitebooks.com

```
      TDirectory.CreateDirectory('images');
      fname := Request.QueryFields.Values['ts'] + '.png';
      FS := TFileStream.Create('images' + PathDelim + fname,
                    fmCreate);
      try
        FS.CopyFrom(Request.Files[0].Stream, 0);
      finally
        FS.Free;
      end;
      SaveInfoFile;
      Response.StatusCode := 200;
      DeleteFilesOlderThan(fname);
    end
    else
    begin
      Response.StatusCode := 400;
    end;
    Handled := true;
  end;
```

Listing 7.5

Now, data is saved in a couple of files with names similar to the following:

- `2014-04-27 23-14-53.png`: This is a plain `.png` image file
- `2014-04-27 23-14-53.png.info`: This is a JSON text file containing location information related to the previous file

Now, the `DefaultHandler` action is used to generate some HTML to let the remote user see the image and location information. Here's the code for this action:

```
procedure TwmMain.WebModule1DefaultHandlerAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  HTMLOut: TStringBuilder;
  FileName, JSONInfoString: string;
  Start, FileTimeStamp: TDateTime;
  Times: Integer;
  JSONInfo: TJSONObject;
  Lat, Lon: Double;
begin
  HTMLOut := TStringBuilder.Create;
  try
    HTMLOut.AppendLine('<html><head>');
```

```
HTMLOut.AppendLine('<style>');
HTMLOut.AppendLine('  body {font-family: Verdana; padding:
                                   50px 10px 50px 50px; }');
HTMLOut.AppendLine('  pre  {font-size: 200%;}');
HTMLOut.AppendLine('</style>');
HTMLOut.AppendLine('<meta http-equiv = "refresh"
                                       Content = "4">');
HTMLOut.AppendLine('</head><body>');
HTMLOut.AppendLine('<h1>Delphi Mobile Monitor</h1>');
Start := Now;
Times := 0;
while true do
begin
  Times := Times + 1;
  FileTimeStamp := Start - OneSecond * Times;
  FileName := 'images' + PathDelim +
     FormatDateTime(DATEFORMAT, FileTimeStamp) + '.png';
  if TFile.Exists(FileName) then
  begin
    HTMLOut.AppendFormat('<h3>Last update %s</h3>',
                         [DateTimeToStr(FileTimeStamp)]);
    HTMLOut.AppendFormat('<img src="%s"><br>', [FileName]);
    if TFile.Exists(FileName + '.info') then
    begin
      JSONInfoString := TFile.ReadAllText(FileName + '.info');
      JSONInfo := TJSONObject.ParseJSONValue(JSONInfoString)
                                        as TJSONObject;
      Lat := (JSONInfo.GetValue('lat') as TJSONNumber).
                                           AsDouble;
      Lon := (JSONInfo.GetValue('lon') as TJSONNumber).
                                           AsDouble;
      HTMLOut.AppendFormat('<pre>Lat: %3.8f Lon: %3.8f</pre>',
                                           [Lat, Lon]);
    end;
    Break;
  end
  else if Times >= 60 * 5 then
  begin
    HTMLOut.Append('<h2>No image available in the last 5
                                       minutes</h2>');
    Break;
  end;
end;
```


www.allitebooks.com

```
    HTMLOut.AppendLine('</body></html>');
    Response.Content := HTMLOut.ToString;
  finally
    HTMLOut.Free;
  end;
end;
```

Listing 7.6

This method creates some HTML on the fly and looks for the most recent snapshot saved on the server. When it finds an image, it inserts the image file name into the HTML to let the browser request for it. Then, it opens the `.info` JSON file, reads the location information, and inserts it in the HTML as well. Note that this monitoring app doesn't have a proper synchronization mechanism between file writing and file reading, so in many parts of the code, you see an empty `try except` block. For this recipe, it is enough. However, in more critical systems, a proper mechanism (such as critical sections, monitors, or mutex) is required to synchronize file access and avoid empty frames, especially with multiple clients.

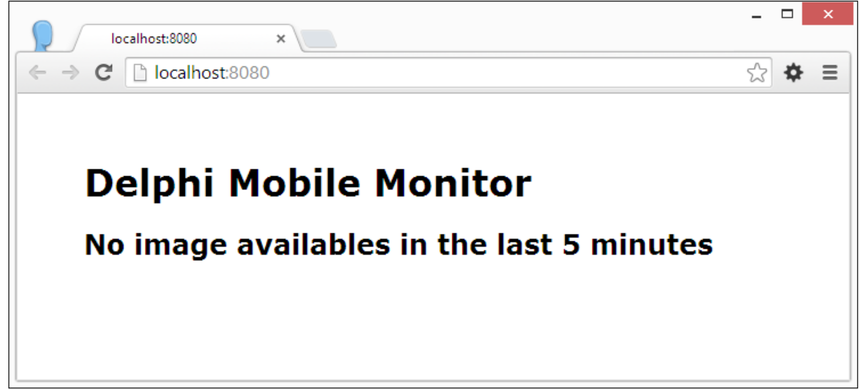
To update the image displayed on the HTML page, there is a special meta tag in the HTML document header, as follows:

```
<meta http-equiv = "refresh" Content = "4">
```

With this line, the page gets updated every 4 seconds (more information about the `http-equiv` meta tags can be found at `http://www.w3schools.com/Tags/att_meta_http_equiv.asp`).

To try the application, launch the server and navigate to the URL `http://localhost:8080` on your browser.

You should see a page like the following:



The monitoring system page when it cannot find images for the last 5 minutes

Now, in the mobile project, open the `ImageSenderThreadU.pas` unit and locate `const MONITORSERVERURL`. Change the `const` value to point to your machine IP. Note that the phone (or the tablet) and your PC must be on the same Wi-Fi network. In my case, the constant is configured as follows:

```
const
  MONITORSERVERURL = 'http://192.168.1.100:8080';
```

Replace the IP with yours, and leave the protocol (`http`) and the port (`8080`) as is.

In a real-world app, put a small configuration section in the mobile app to let the user enter the actual URL where the server listens.

Run the mobile app, activate the camera using the switch on the top-right corner, and after a couple of seconds, you should see an image and the location information coming up in the web page. The final web page should look like the following:

The monitoring system running while showing a sort of recursive image of itself

www.allitebooks.com

There's more...

This recipe acts like a training for a lot of concepts. If you want to go deeper into them, you can read the following articles and information:

- *Using Location Sensors* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Mobile_Tutorial:_Using_Location_Sensors_(iOS_and_Android)`
- *Uses Permissions* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Uses_Permissions`
- *FMX.Media.TCameraComponent* at `http://docwiki.embarcadero.com/Libraries/XE6/en/FMX.Media.TCameraComponent`

Talking to the backend

This recipe will introduce you to the real-world business mobile apps and their related application servers. It is not a simple world. It is full of well-known traps and full of specific traps related to your infrastructure, your business logic, your application transactions, and so on. Just to be clear, you have to take care of your design and the way you implement this design in a much more extent compared to a classic client/server application. On going deeper into the mobile programming (and in general, in all asynchronous scenarios), you will see that things become harder than usual. In the mobile world, things can get messy really fast and your customers will complain even faster. Be warned!

This recipe is a mobile client for the **People Manager** application server developed in the *Implementing a RESTful interface using WebBroker* recipe in *Chapter 5*, *Putting Delphi on the Server*.

Getting ready

As already mentioned, this recipe is composed by the application server and the mobile client. The UI is not blocking so that all the REST requests are executed in a background thread using the built-in features of `RESTClient`.

How to do it...

The app is based on the **Header/Footer with Navigation** mobile template. In the first `TTabItem` object, there is a list of people. In the second `TTabItem` object, there are the selected person's details. Data is read from the `REST` services exposed by the `PeopleManager.dproj` server.

The client implements a simple CRUD operation and uses a subset of the server services. The service used and the relative URL are mentioned in the following table (you can implement search functionality as an exercise):

HTTP verb	URL	Description
`GET`	`/people`	This returns a JSON array containing one JSON object for each record present in the `PEOPLE` table. In each object, the property name is the name of the field, while the property values are the value of the fields.
`POST`	`/people`	This creates a new person in the `PEOPLE` table. This requires a `request` body containing the new person's data to create a JSON object. The `content-type` request must be `application/json`.
`PUT`	`/people/{id}`	This updates the person with `id` with the data passed in the `request` body. This requires a `request` body containing the person to update as JSON object. The `content-type` request must be `application/json`.
`DELETE`	`/people/{id}`	This deletes the person with `id`.

The `GET people/:id` method is available from the server too, but the client doesn't use it because the `GET /people` method already returns an array with all the complete entities. In a real-world app, you perhaps have lots of entities, or a lot of entity attributes or nested objects, so it makes sense to use the `GET` verb to get a single entity representation.

Locally, the data is stored in a `TDataSet` component, a `TFDMemTable` component (yes, I love it) to be precise, and are loaded using the class helper declared in `ObjectsMappers.pas` (contained in the `DelphiMVCFramework` project and already used in *Chapter 5*, *Putting Delphi on the Server*).

All the logic is implemented in a data module created before the main form is created (go to **Project** | **Options** | **Forms** to check the form creations order). Methods provided by the data module to the main form are as follows:

```
public
  procedure SavePerson(AOnSuccess: TProc;
            AOnError: TProc<Integer, String> = nil);
  procedure DeletePerson(AOnSuccess: TProc;
            AOnError: TProc<Integer, String> = nil);
  procedure LoadAll(AOnSuccess: TProc;
            AOnError: TProc<Integer, String> = nil);
  function CanSave: Boolean;
```


www.allitebooks.com

CanSave is used to enable or disable UI actions depending on the dsPeople dataset state. The LoadAll method is called from the FormShow event handler, and it requests data to the server and populates the in-memory dataset. Seeing that all remote requests are asynchronous, we need some callback to update the UI after the request is finished in order to show data in the case of success, or to show error messages in the case of errors. Here's the code for the data module LoadAll method:

```
procedure TdmMain.LoadAll(AOnSuccess: TProc; AOnError:
                            TProc<Integer, String>);
begin
  if dsPeople.State in [dsInsert, dsEdit] then
    dsPeople.Cancel;   //cancel all unposted data
  dsPeople.Close;

  RESTRequest.ClearBody;
  RESTRequest.Resource := 'people';
  RESTRequest.Method := TRESTRequestMethod.rmGET;

  //execute remote request asynchronously
  //WARNING! The anonymous method passed as parameter to the
  //ExecuteAsynch is execute within the main thread, so there is
  //no need to synchronize UI access
  RESTRequest.ExecuteAsync(
    procedure
    begin
      if RESTRequest.Response.StatusCode = 200 then
      begin
        //load response jsonarray in the dataset
        dsPeople.Active := True;
        dsPeople.AppendFromJSONArrayString(
           RESTRequest.Response.JSONValue.ToString);
        if Assigned(AOnSuccess) then
        //call the 'success' user callback
        AOnSuccess();
      end
      else
      begin
        if Assigned(AOnError) then
          //call the 'error' user callback
           AOnError(RESTRequest.Response.StatusCode,
                        RESTRequest.Response.StatusText);
      end;
    end);
end;
```

This method is declared in the data module. How to call this method in the `acRefreshData` action within the main form? Here's the code:

```
procedure TMainForm.acRefreshDataExecute(Sender: TObject);
begin
  DoStartWait('Please wait while retrieving the people list');
  dmMain.LoadAll(
    procedure
    begin
      DoEndWait;
    end,
    procedure(StatusCode: Integer; StatusText: String)
    begin
      DoEndWait;
      ShowError(Format('Error [%d]: %s',
                [StatusCode, StatusText]));
    end);
end;
```

Remember, a call to the `LoadAll` method is not blocking for the main thread. So, any code after a call to `LoadAll` is executed as soon as possible (as the OS decides) and not after the data is retrieved. This is the reason why we need the callbacks. The first anonymous method is our `success` callback, and it is executed when data is already in the dataset and the user can see it in the listview. The second anonymous method is our `error` callback, and it is executed if some errors occur in the call. The other remote calls work in the same manner.

If you, for some reason, would like to use a different HTTP component to do the REST HTTP calls and this library doesn't support asynchronous client requests, you can always rely on the good old anonymous thread. The following code is included in the `LoadAll` method, but it is commented to show an alternative way to do remote call without using the `ExecuteAsynch` method:

```
  TThread.CreateAnonymousThread(
    procedure
    begin
      try
        //synch call, but executed in an anonthread
        RESTRequest.Execute;
        TThread.Synchronize(nil,
          procedure
          begin
            if RESTRequest.Response.StatusCode = 200 then
            begin
              dsPeople.Active := True;
              dsPeople.AppendFromJSONArrayString(
                RESTRequest.Response.JSONValue.ToString);
```


www.allitebooks.com

```
          if Assigned(AOnSuccess) then
            AOnSuccess();
        end
        else
          AOnError(RESTRequest.Response.StatusCode,
                   RESTRequest.Response.StatusText);
      end);
    except
      on E: Exception do
      begin
        if Assigned(AOnError) then
        begin
          ErrMsg := E.Message;
          TThread.Synchronize(nil,
            procedure
            begin
              //Passing 'Zero' to the callback means that some
              //non-protocol related exception has been raised
              AOnError(0, ErrMsg);
            end);
        end
      end;
    end;
  end).Start;
```

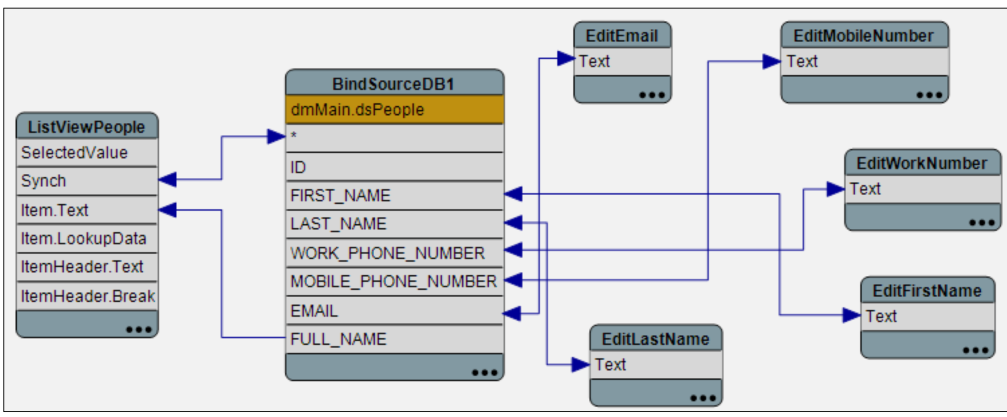
An important feature of well-designed mobile apps is the feedback to the user. Your user *must* know what your application is doing after their input; otherwise, he/she would probably stop it. So, we need to show a **Please wait** screen. To do so, this app uses a `TPopup` component. This component has a property called `IsOpen` that is used to show it or hide it. Just before each request, we set an instance form variable to `true` and after the request, when the response is visible somewhere in the UI, we set that variable to `false`. Here's the code to handle the **Please wait** screen:

```
procedure TMainForm.DoEndWait;
begin
  FBackgroundOperationRunning := False;
end;

procedure TMainForm.DoStartWait(AWaitMessage: String);
begin
  //this label is placed inside the "Please wait" screen
  lblMessage.Text := AWaitMessage;
  FBackgroundOperationRunning := True;
end;
```

How to actually show the `TPopup` component? In the mobile **Header/Footer with Navigation** template, there is a `TControlAction` component to update the header caption according to the selected tab. Its `OnUpdate` event handler is a good source to update the UI when the app is in idle state. Here's the code:

```
procedure TMainForm.TitleActionUpdate(Sender: TObject);
begin
  //this is the code generated by the template wizard
  if Sender is TCustomAction then
  begin
    if TabControl1.ActiveTab <> nil then
      TCustomAction(Sender).Text := TabControl1.ActiveTab.Text
    else
      TCustomAction(Sender).Text := '';
  end;

  //this is the code added
  acRefreshData.Enabled := (not FBackgroundOperationRunning) and
         (TabControl1.ActiveTab = TabItem1);
  AniIndicator1.Visible := FBackgroundOperationRunning;
  TabItem1.Enabled := not FBackgroundOperationRunning;
  ppMessage.IsOpen := FBackgroundOperationRunning;
end;
```

Data is linked to the UI using the LiveBindings engine. Here's the LiveBindings designer showing the links:



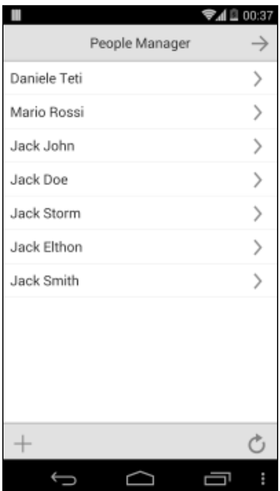
The LiveBindings designer showing the links between the dsPeople and the UI

www.allitebooks.com

After launching the app, you will get this wait screen:

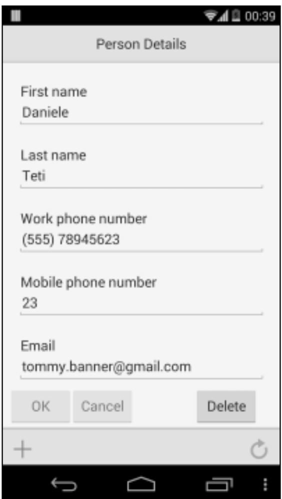
The wait screen

Then, when the data is retrieved, parsed, and loaded, this is the screen you will get:



The list of people loaded in the listview

If you tap an item, you will get the editing screen:



The editing screen showing the person's information

There's more...

A lot of topics in this recipe! Mobile apps can be really complex as this simple example demonstrates. However, using the LiveBindings engine, the local storage offered by SQLite and IBLite, and the nice Delphi components to load data in memory, you can create mobile apps easily enough. Here are some other demos about the technologies involved in developing these type of apps:

- *FireDAC IBLite with Delphi XE6* at `https://www.youtube.com/watch?v=jbRJCqNgNDc`
- *Delphi XE5 Mobile REST Client Demo* at `https://www.youtube.com/watch?v=OkRVbgF4VMI`
- *Delphi XE5 Mobile REST Client Demo Source* at `http://delphi.org/2013/09/delphi-xe5-mobile-rest-client-demo-source/`
- *The New REST Client Library, A Tool of Many Trades* at `https://www.youtube.com/watch?v=nPXYLK4JZvM`

Making a phone call from your app!

Many mobile devices, especially in the consumer market, are phones or are devices that can make phone calls. In some cases, your mobile app may have the ability to make a call or just to monitor the incoming or outgoing calls.

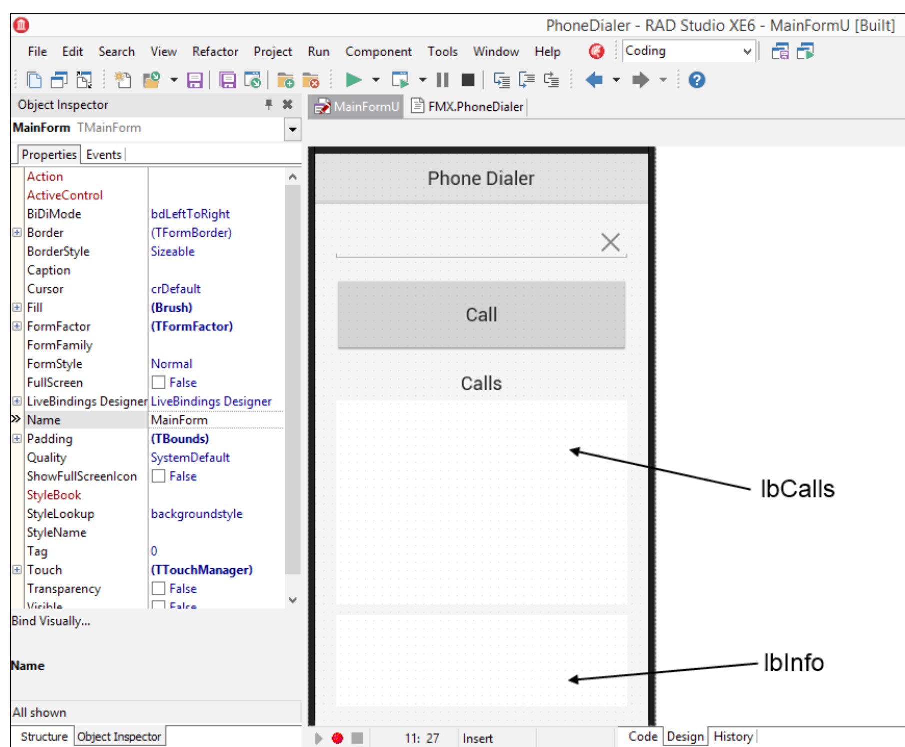
www.allitebooks.com

Getting ready

In this recipe, we'll see how to make a call and how to monitor the current calls as well. Also, in this case, the useful FireMonkey platform services framework come handy.

How to do it...

1. Create a new mobile app by navigating to **File** | **New** | **FireMonkey mobile application – Delphi**.
2. Select the **Header/Footer** template and click on **OK**.
3. Drop the following components on the main form:
   - **TEdit** (`edtPhoneNumber`)
   - **TButton** (`btnCall`)
   - **TListBox** (`lbCalls`)
   - **TListBox** (`lbInfo`)
4. Arrange the components as shown in the following screenshot:



The form with all the controls arranged

5. Put some labels to explain what the listboxes will contain, as shown in the preceding screenshot.
6. Now, create the `FormCreate` event handler and fill it with this code:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  lbInfo.Clear;
  if TPlatformServices.Current.
      SupportsPlatformService(IFMXPhoneDialerService,
                          IInterface(FPhoneDialerService))
  then
  begin
    FPhoneDialerService.OnCallStateChanged := CallStateChanged;
    lbInfo.ItemHeight := lbInfo.ClientHeight / 4;
    lbInfo.Items.Add('Carrier Name: ' +
      FPhoneDialerService.GetCarrier.GetCarrierName);
    lbInfo.Items.Add('ISO Country Code: ' +
      FPhoneDialerService.GetCarrier.GetIsoCountryCode);
    lbInfo.Items.Add('Network Code: ' +
      FPhoneDialerService.GetCarrier.GetMobileCountryCode);
    lbInfo.Items.Add('Mobile Network: ' +
      FPhoneDialerService.GetCarrier.GetMobileNetwork);
    btnCall.Enabled := True;
  end
  else
    lbInfo.Items.Add('No Phone Dialer Service');
end;
```

7. In the form's `private` section, declare the following methods:

```
  private
    FPhoneDialerService: IFMXPhoneDialerService;
    procedure CallStateChanged(const ACallID: string;
                          const AState: TCallState);
    function CallStateAsString(AState: TCallState): String;
```


www.allitebooks.com

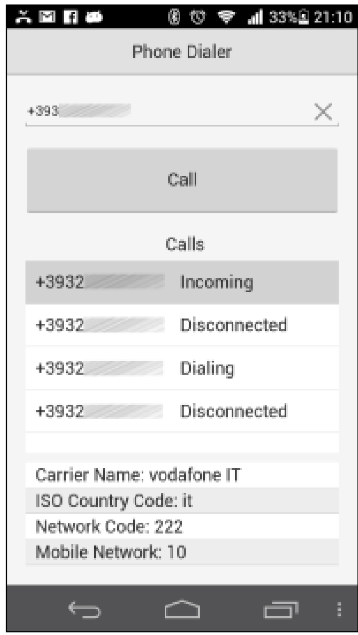
8. Press *Ctrl* + *Shift* + *C* and fill the methods just created with the following code:

```
function TMainForm.CallStateAsString(AState: TCallState): String;
begin
  case AState of
    TCallState.None:
      Result := 'None';
    TCallState.Connected:
      Result := 'Connected';
    TCallState.Incoming:
      Result := 'Incoming';
    TCallState.Dialing:
      Result := 'Dialing';
    TCallState.Disconnected:
      Result := 'Disconnected';
    else
     Result := '<unknown>';
  end;
end;
procedure TMainForm.CallStateChanged(const ACallID: string;
  const AState: TCallState);
begin
  lbCalls.Items.Add(Format('%-16s %s',
                [ACallID, CallStateAsString(AState)]));
end;
```

9. Now, create the `OnClick` event for the `btnCall` method and fill it with this code:

```
procedure TMainForm.btnCallClick(Sender: TObject);
begin
  if not edtPhoneNumber.Text.IsEmpty then
    FPhoneDialerService.Call(edtPhoneNumber.Text)
  else
  begin
    ShowMessage('No number to call, please type a phone number.');
    edtPhoneNumber.SetFocus;
  end;
end;
```

10. Run the app on your phone. Note the `lbInfo` method showing all the information about your mobile network. Write a phone number in the editing area and click on the **Call** button. Note what happens to the `lbCalls` method during the outgoing calls and during the incoming calls. This activity is shown in the following screenshot:



The Phone Dialer app running on a phone, after some in/out calls; note the events in the first list

How it works...

This recipe is very simple. All the work is done at the beginning when the `FormCreate` event handler asks the system whether it supports the `IFMXPhoneDialerService` interface. This interface has the following methods:

```
{ Interface of Phone Dialer }
IFMXPhoneDialerService = interface (IInterface)
  ['{61EE0E7A-7643-4966-873E-384CF798E694}']
  // Make a call by specified number
  function Call(const APhoneNumber: string): Boolean;
  // Get current carrier
  function GetCarrier: TCarrier;
  // Get all currrent calls. If the current calls aren't
  // present, the array will be empty
  // The developer shall delete array cells after use
  function GetCurrentCalls: TCalls;
```


www.allitebooks.com

```
    // Getter, Setter and property for work with event of tracing
    // of state change of a call
    function GetOnCallStateChanged: TOnCallStateChanged;
    procedure SetOnCallStateChanged(
                     const AEvent: TOnCallStateChanged);
    property OnCallStateChanged: TOnCallStateChanged
               read GetOnCallStateChanged
               write SetOnCallStateChanged;
  end;
```

There's more...

Using the monitor functionality, you can implement a system to track the phone calls' duration and type (incoming or outgoing). Using this service, you can implement a list of contacts centralized on a server and allow your user to call those contacts without having the contact in the phone's address book. Another utilization is to monitor the allowed numbers to call, and if some special blocked number is called, you can send a notification to a remote server. There are endless possibilities—explore them yourself.

Tracking the application's life cycle

In the "safe" MS Windows desktop application development land, our application has a life cycle but it is not so crucial take care of it. Usually, you have a set of events to handle such as `FormCreate`, `FormClose` (at the form level), or `Application OnRestore`, or `application OnTerminate`. In some cases, you have to handle the state where the main application window is minimized, and this is still simple. In the mobile world, as usual, things are a bit more complex. The concept of life cycle is an evidence. Just to make things messier, the Android activity's life cycle is different from the iOS view life cycle. Remember, when an app is in background, it can be completely destroyed.

Getting ready

But, hey! Why I should care about the life cycle? That's a very good point! There are a lot of things that you should or must do while your application is switching from one state to another.

Here are some examples:

- Handle current input control's state. You can save or discard data, but you cannot send the **Do you want to save?** message to the user. If a user touches the **Home** button, you cannot stop them.
- Stop or restart some CPU intensive work related to some calculation.
- Look for some previously saved data on the filesystem.
- Search some Bluetooth devices or App-Tethering-enabled applications.
- Update a remote resource more frequently than when the app was running in background. In background, you may check a particular HTTP resource once an hour, while if the app is in foreground, you can decide to check that resource once at minute.
- Append a system notification to remind something to the user just before terminating an app.
- Stop the audio output (if applicable for your app).
- Stop the GPS monitoring (if applicable for your app).
- Going into power saving mode, whatever it means for your app, and many more.

As you can see, the application life cycle is very important. Let's see how we can hook to it.

How to do it...

This recipe is not a standard recipe. We'll not build a complete app, but a reference app. You can launch this app every time you want to know which event (app event or form event) is fired and when.

As the first thing, in the `FormCreate` event, we've to hook to the system FireMonkey messaging system and subscribe to the `TApplicationEventMessage` message type:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  TPlatformServices.Current.SupportsPlatformService
  (IFMXLoggingService, IInterface(FLoggingService));

  FSubscrID := TMessageManager.DefaultManager.
   SubscribeToMessage(TApplicationEventMessage,
             procedure(const Sender: TObject;
                             const Msg: TMessage)
            var
             AppEvent: TApplicationEventMessage;
            begin
                 AppEvent := TApplicationEventMessage(Msg);
```


www.allitebooks.com

```
                    case AppEvent.Value.Event of
                          TApplicationEvent.FinishedLaunching:
                            LogEvent('App Finished Launching');
                          TApplicationEvent.BecameActive:
                            LogEvent('App Became Active');
                          TApplicationEvent.WillBecomeInactive:
                            LogEvent('App Will Become Inactive');
                          TApplicationEvent.EnteredBackground:
                            LogEvent('App Entered Background');
                          TApplicationEvent.WillBecomeForeground:
                           LogEvent('App Will Become Foreground');
                          TApplicationEvent.WillTerminate:
                             LogEvent('App Will Terminate');
                          TApplicationEvent.LowMemory:
                             LogEvent('App Low Memory');
                          TApplicationEvent.TimeChange:
                             LogEvent('App Time Change');
                          TApplicationEvent.OpenURL:
                             LogEvent('App Open URL');
                    end;
            end);

  LogEvent('Event FormCreate');
end;
```

With this code, every time the system raises a message regarding our app, we'll be informed. The `System.Messaging.pas` unit, added in the `implementation` uses clause, contains the classes needed to access to the system's messaging system.

How does this messaging system work? Once you have an instance of `TMessageManager`, you can subscribe message-handling methods to specific types of messages. Message-handling methods may be methods of an object or anonymous methods. In our case, we've used an anonymous method. This messaging mechanism can also be used in your app or component as an independent messaging system. However, FireMonkey also uses it to send system messages using the default messaging manager instance. In this recipe, we're using it to subscribe to the system messages.

An instance of `TApplicationEvent`, the type on which we're doing the big case statement, represents the application-related messages and may have any of the following values:

Event	**Description**
`BecameActive`	This indicates that an application has gained the focus.
`EnteredBackground`	This indicates that the application is running in the background because the user is no longer using it.
`FinishedLaunching`	This indicates that the application has been launched.

Event	Description
LowMemory	This event is a warning for the application that the device is running out of memory. In this case, the application should reduce memory usage, freeing structures and data that are not fundamental or that can be reloaded as per requirements at a later point.
OpenURL	This indicates that the application has received a request to open a URL (only for iOS).
TimeChange	This indicates that there has been a significant change in time (only for iOS). This event might happen, for example, when the day changes or when the device changes to or from daylight savings time.
WillBecomeForeground	This indicates that the user is now using the application, which was previously running in the background.
WillBecomeInactive	This indicates that the application is going to loose the focus and become inactive.
WillTerminate	This indicates that the user or the operating system is quitting the application.

Remember, to be a good FireMonkey citizen, when you subscribe to a system notification, you have to unsubscribe it too. We do it in the `FormDestroy` event just after logging—the last thing:

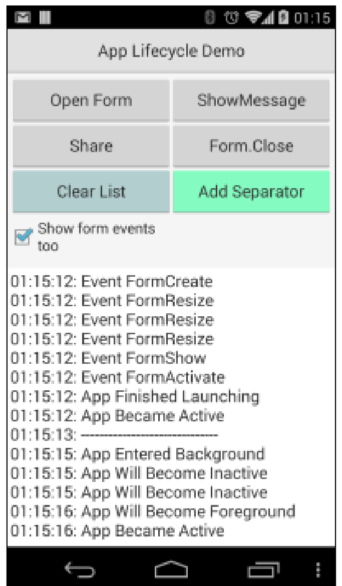
```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  LogEvent('Event FormDestroy');
  TMessageManager.DefaultManager.Unsubscribe(
    TApplicationEventMessage, FSubscrID, True);
end;
```

The `LogEvent` method appends the message text to the listbox and writes the same message to the system log as well using the reference to `IFMXLoggingService` retrieved in the `FormCreate` event handler. Moreover, whereas the form events could be many, there is a checkbox to exclude them from the logging. Here's the code for the `LogEvent` method:

```
procedure TMainForm.LogEvent(Msg: string);
begin
  if (not CheckBox1.IsChecked) and
    Msg.StartsWith('event', True) then Exit;
  Memo1.Lines.Add(Format('%s: %s', [TimeToStr(Now), Msg]));
  Memo1.GoToTextEnd; //memo goes to the last line
  if Assigned(FLoggingService) then
    FLoggingService.Log('LifeCycle: %s', [Msg]); //syslog
end;
```


www.allitebooks.com

This is the infrastructure code, but what events are we waiting for? In the main form, there are some test buttons that raise specific system and form events. Here's the app while it is logging form events and system messages:



The app while it is logging form events and system messages

In the main form, every interesting event that could be raised, whether the app state changes or not, is filled with code similar to the following one:

```
procedure TMainForm.FormActivate(Sender: TObject);
begin
  LogEvent('Event FormActivate');
end;
```

Now that everything is traced, app state changes and forms events. Now you can connect your device, if it's not already connected, and launch the proper tool to see the device logger (launch `Monitor.bat` for Android devices or see the device console for iOS devices). Start the app and play with the buttons.

Try to tap the **Open Form** button and then close the newly opened form by tapping on the **Close** button. As you can see in the list, only form events are called (`FormDeactivate` and `FormActivate`) and this is reasonable. Now tap on the **ShowMessage** button and see what happens. Form events are not raised but an app message arrives. Look! The app goes in the inactive state for a `ShowMessage` call! This is the case where this sort of testing tool is very handy. If you don't know exactly when an app switches its state from one to another, you cannot rely to this state change to do anything useful and reliable. But now you have the right tool!

There's more...

Experimenting with the life cycle, you can find interesting utilization that makes your user happy with your app.

Another interesting thing that I suggest you to study is the messaging system based on a variation of the well-known and more general **Observer** design pattern of the GoF fame. Simply speaking, this messaging system is just something that triggers an event to which anyone can listen. Different libraries offer different implementations and for different purposes, but the basic idea is to provide a framework for issuing events and subscribing to them.

More information about the `System.Messaging.pas` unit can be found in the following articles:

- *Sending and Receiving Messages Using the RTL* at `http://docwiki.embarcadero.com/RADStudio/XE6/en/Sending_and_Receiving_Messages_Using_the_RTL`
- *List of FireMonkey Message Types at* `http://docwiki.embarcadero.com/RADStudio/XE6/en/List_of_FireMonkey_Message_Types`
- System.Messaging (Delphi) at `http://docwiki.embarcadero.com/CodeExamples/XE6/en/System.Messaging_(Delphi)`
- Where all the messaging system has begun: the Observer design pattern at `http://en.wikipedia.org/wiki/Observer_pattern`

www.allitebooks.com

7

Using Specific Platform Features

In this chapter, we will cover the following topics:

- Using Android SDK Java classes
- Using iOS Objective-C SDK classes
- Displaying PDF files in your app
- Sending Android intents
- Letting your phone talk – using the Android TextToSpeech engine

Introduction

There are situations where if you need a particular Android or iOS feature, FireMonkey doesn't help you. FireMonkey does a very good job in supporting all the common mobile features, but it is still relatively young. This means that not all the APIs have been already imported, polished, and wrapped in nice Object Pascal reusable classes or components. So, what can you do in such cases? The good news is that you can import classes from the underlying SDK (and NDK, in case of Android) and wrap them just like Embarcadero does on the FireMonkey platform.

In this chapter, we'll see some classes import examples. Keep in mind that the code using imported classes is not cross platform. That is, if you import an Android SDK class and your code uses it, you lose the possibility to compile that specific code for iOS. However, you can, as usual, use some IFDEFs to statically select the Android-specific code from the iOS-specific code.

Using Android SDK Java classes

In this recipe, we'll talk about the mechanisms that the compiler offers to import classes from the Android SDK and NDK. This is not a standard recipe, but is more of a showcase of the possibilities offered by the Delphi compiler, and the processes needed to fully use them when dealing with built-in libraries of the OS.

Getting ready

What we'll do is import a well-known Android class used everywhere in the Android ecosystem—`Toast`. The Android documentation states the following:

> *"A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. For example, navigating away from an email before you send it triggers a "Draft saved" toast to let you know that you can continue editing later. Toasts automatically disappear after a timeout."*

So, how do we use a Toast in a Delphi app?

The first thing to do is have a clear vision of the class methods and all the other types involved in their definition. You can get this information from the official documentation at `http://developer.android.com/reference/android/widget/Toast.html`. The following tables show the most relevant class members as explained by the Android Java SDK documentation.

The following table shows the `Toast` class constants:

Type	Constant
int	LENGTH_LONG
int	LENGTH_SHORT

The following table shows the public instance methods of `Toast`:

Type	Method
void	cancel()
int	getDuration()
int	getGravity()
float	getHorizontalMargin()
float	getVerticalMargin()
View	getView()
int	getXOffset()

www.allitebooks.com

Type	Method
int	getYOffset()
void	setDuration(int duration)
void	setGravity(int gravity, int xOffset, int yOffset)
void	setMargin(float horizontalMargin, float verticalMargin)
void	setText(int resId)
void	setText(CharSequence s)
void	setView(View view)
void	show()

The following table shows public static methods of `Toast` (like the class methods in Delphi):

Type	Method
Toast	makeText(Context context, int resId, int duration)
Toast	makeText(Context context, CharSequence text, int duration)

This is the typical usage of the `Toast` class inside an Android activity:

```
Toast.makeText(getContext(),
"Hello Toast World",
Toast.LENGTH_LONG).show();
```

Now with this information, we can define our import Delphi class.

How to do it...

The Android Delphi compiler allows you to declare a specific class as a generic Java import of an SDK Java class. The class that does this magic is declared within the `Androidapi.JNIBridge.pas` unit as follows:

```
TJavaGenericImport<C: IJavaClass; T: IJavaInstance>
```

`TJavaGenericImport` is a generic class that we can use to make the declaration of the imported Java object factories easier. Using this class, we split the class and instance methods into two interfaces. This class blends the two interfaces into a single factory that can produce instances of Java objects, or provide a reference to an instance representing the Java class. Moreover, Android Java SDK uses Java `String` objects, while Delphi uses strings. If you need to pass a string to a method imported from the SDK that expects a JString (the type used by the Delphi compiler to match the Java `String` object), you have to use the `StringToJString` function defined in `Androidapi.Helpers.pas` to convert it.

So, the next step to use the `Toast` class is to define two interfaces. The first one declares all the class methods (static in Java) with the same signature as that of the Java ones. The second one declares all the instance methods with the same Java signature as well.

How to map Java types to the Delphi types? In the Delphi RTL, there are many samples of the imported Java classes. The following table gives you a small summary of what you can understand from the already imported classes and from the `api-version.xml` file present in the Android SDK, which contains the declaration of all the SDK classes (`<Public Documents>\Embarcadero\Studio\14.0\PlatformSDKs\adt-bundle-windows-x86-20131030\sdk\platform-tools\api\api-versions.xml`):

Java type	Delphi type
boolean	Boolean
byte	ShortInt
char	WideChar
double	Double
float	Single
int	Integer
long	Int64
short	SmallInt
void	If used as return type, use procedure instead of function
java/lang/CharSequence	JCharSequence
android/view/View	JView
java/lang/String	JString

All the methods must be declared with the `cdecl` calling convention to be compatible with the Java calling convention. Moreover, the interface declaring the interface methods must be decorated with the `JavaSignature` RTTI attribute that defines the full Java package, where the mapped class is declared in the SDK. It may seem complex, but the resultant code is not. The following code is the final import declaration for the `Toast` class:

```
type
  [JavaSignature('android/widget/Toast')]
  JToast = interface(JObject)
    ['{AC116FB8-FE4D-47E8-BEC9-96E919A01CC7}']
    procedure cancel; cdecl;
    function getDuration: Integer; cdecl;
    function getGravity: Integer; cdecl;
    function getHorizontalMargin: Single; cdecl;
    function getVerticalMargin: Single; cdecl;
    function getView: JView; cdecl;
```


www.allitebooks.com

```
    function getXOffset: Integer; cdecl;
    function getYOffset: Integer; cdecl;
    procedure setDuration(duration: Integer); cdecl;
    procedure setGravity(gravity: Integer; xOffset:
              Integer; yOffset: Integer); cdecl;
    procedure setMargin(horizontalMargin: Single;
              verticalMargin: Single); cdecl;
    procedure setText(resId: Integer); cdecl; overload;
    procedure setText(s: JCharSequence); cdecl; overload;
    procedure setView(view: JView); cdecl;
    procedure show; cdecl;
  end;

  JToastClass = interface(JObjectClass)
                ['{127EA3ED-B569-4DBF-9BCA-FE1491FC615E}']
    function init(context: JContext): JToast; cdecl;
    function makeText(context: JContext; resId: Integer;
             duration: Integer): JToast; cdecl; overload;
    function makeText(context: JContext;
             text: JCharSequence;
             duration: Integer): JToast; cdecl; overload;
  end;
```

Now, with these two interfaces, we can declare our `TJToast` class inheriting it from `TJavaGenericImport`, as shown in the following code:

```
  TJToast = class(TJavaGenericImport<JToastClass, JToast>)
  const
    LENGTH_LONG = 1;
    LENGTH_SHORT = 0;
  end;
```

As you can see, the body of the class is almost empty because all the methods will be used with the help of an internally created object returning an interface reference. `LENGTH_LONG` and `LENGTH_SHORT` are simple constants in Java, so I added them as `const` in the `TJToast` declaration. The `TJToast` class can be used as follows using the same methods documented for the Android Java SDK:

```
procedure TMainForm.Button3Click(Sender: TObject);
var
  Toast: JToast;
begin
  Toast := TJToast.JavaClass.makeText(SharedActivityContext,
    StrToJCharSequence('Hello World'), TJToast.LENGTH_SHORT);
  Toast.show();
end;
```

However, if you run the preceding code, you will get the following exception:

Java.lang.RuntimeException: Can't create handler inside thread that has not called Looper.prepare()

This is because the Toast must be synchronized with the UI thread. So, using the `CallInUiThread` function declared into `FMX.Helpers.Android.pas`, we can synchronize the call with the main thread. The following is the complete code:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  CallInUiThread(
    procedure
    var
      Toast: JToast;
    begin
      Toast := TJToast.JavaClass.makeText(SharedActivityContext,
        StrToJCharSequence('Hello World'), TJToast.LENGTH_SHORT);
      Toast.show();
    end);
end;
```

Now the code works, but the utilization pattern is not too Delphi-like. Indeed, we're using Java classes and methods using the Delphi syntax. However, we can write some helper code to make the Toast utilization more similar to the Delphi RTL and the Delphi programmer mind-set, as follows:

```
    interface

{$SCOPEDENUMS ON}

type
  TToastDuration = (Short = 0, Long = 1);
  TToastPosition = (Default = 0, TOP = 48,
                    BOTTOM = 80, CENTER = 17,
                    VerticalCenter = 16, HorizontalCenter = 1);
procedure ShowToast(const AText: string;
     const ADuration: TToastDuration = TToastDuration.Short;
     const APosition: TToastPosition = TToastPosition.Default);

implementation

uses
  FMX.Helpers.Android, AndroidAPI.Helpers;

procedure ShowToast(const AText: string;
                    const ADuration: TToastDuration;
                    const APosition: TToastPosition);
```


www.allitebooks.com

```
  begin
    CallInUiThread(
      procedure
      var
        Toast: JToast;
      begin
        Toast := TJToast.JavaClass.makeText(SharedActivityContext,
          StrToJCharSequence(AText), Integer(ADuration));
        if APosition <> TToastPosition.Default then
          Toast.setGravity(Integer(APosition), 0, 0);
        Toast.show();
      end);
  end;
```

In this version, we've used the `setGravity` method to define the Toast position on the screen. We've used an enumerated type mapped to the same integer values defined in the `android.view.Gravity` class. Also, check the call to `SharedActivityContext` to get the activity context needed by the method.

Now, we can use the `Toast` class using a very Delphi-styled function. Here are some sample calls:

```
ShowToast('Hello Toast World');
ShowToast('Hello Toast World', TToastDuration.Long,
                               TToastPosition.Center);
ShowToast('Hello Toast World', TToastDuration.Short);
```

As a suggestion, try to make your imports as intuitive as possible for your Delphi users (even if you are the only user) because the rest of your code is written using the Delphi libraries. Stay as homogeneous as possible; it's a good principle for whatever language you use. Encapsulate the imported classes in proper Delphi code structures (classes, records, functions, and whatever is appropriate) and the style of your code will benefit from it being much more coherent with itself.

In the recipe code, there is a complete app showing different kinds of Toast utilization.

There's more...

Complex classes require more work to be imported, but there are tools that can help in this hard work, for example, **Java2Pas** (`http://www.softwareunion.lu/en/downloads/`).

These tools do a good job and help in the boring methods declaration phase. However, you cannot simply import a class and use it in your Delphi code. In many cases, you have to do additional work to arrange a good class structure in your units to avoid circular unit references.

However, if you are interested and want to know more, I suggest you check the wonderful presentation held by Brian Long at the CodeRage 8 conference, where he talks about accessing the Android and iOS APIs. The presentation can be found at the following URL:

`http://blog.blong.com/2013/10/my-coderage-session-files.html`

Delphi XE7 allows you to use your own or third-party Java libraries in RAD Studio applications in a simpler way. Check the following link for more information:

`http://docwiki.embarcadero.com/RADStudio/XE7/en/Using_a_Custom_Set_of_Java_Libraries_In_Your_RAD_Studio_Android_Apps`

As FireMonkey and the mobile "soul" of Delphi mature, third-party mobile components will become available in the market. Even if you are not interested in native widgets, you can study the code from the project *D.P.F Delphi Android Native Components*. It will really help to see how to import complex Java classes into Delphi. You can find this at the following URL:

`http://sourceforge.net/projects/dpfdelphiandroid/`

Moreover, you can also use native NDK `.so` files. To get an idea on how to do this, check the `Androidapi.Log.pas` unit, where the function used by the `IFMXLoggingService` service on Android is declared. As you will see, there is a declaration very similar to the declaration usually used for the Windows DLL:

```
const
  AndroidLogLib         = '/usr/lib/liblog.so';

function __android_log_write(Priority: android_LogPriority;
         const Tag, Text: MarshaledAString): Integer; cdecl;
  external AndroidLogLib name '__android_log_write';
```

As time passes, Embarcadero will add more and more imports for the Android SDK, but until then, if you need to use specific SDK classes or third-party Java classes (to be packaged in the generated APK will require a bit of work), you can rely on the compiler support and the RTL class `TJavaGenericImport` to declare and use it.

Using iOS Objective-C SDK classes

Just like we saw about Android in the previous recipe, Delphi can access the iOS SDK as well. In this section, we'll talk about the mechanisms that the compiler offers to import classes from the iOS SDK. This is not a standard recipe, but is more of a showcase of the possibilities offered by the Delphi compiler, and the process needed to fully use them when dealing with the OS built-in libraries. The mechanism is similar to the Android ones, but there are some notable differences.

www.allitebooks.com

Getting ready

In Objective-C, all the classes have `NSObject` as a common ancestor. The iOS SDK is composed of some frameworks. The iOS framework comprises of a number of classes specialized for a single purpose. For example, UIKit is the framework containing all the basic classes related to the UI; the iAd framework contains all the stuff related to advertising, and MapKit wraps up all the mapping-related classes.

Note that Objective-C uses the `NSString` objects while Delphi uses strings. If you need to pass a string to an iOS API, which expects an `NSString` object, you can use the `StrToNSStr` function defined in `Macapi.Helpers.pas` to convert it.

Let's say we need to use the `UIDevice` class from the iOS SDK (the process is applicable for every class in the SDK). The Apple documentation states the following:

> *"The UIDevice class provides a singleton instance representing the current device. From this instance you can obtain information about the device such as assigned name, device model, and operating-system name and version."*

How to do it...

The iOS Delphi compiler allows you to declare a specific class as a Generic Objective-C import of an SDK class. The class that does this magic is declared within the `Macapi.ObjectiveC.pas` unit as follows:

```
TOCGenericImport<C: IObjectiveCClass; T: IObjectiveCInstance>
```

`TOCGenericImport` is a generic class that we can use to make the declaration of the imported Objective-C object factories easier. Using this class, we split the class and instance methods into two interfaces. This class blends the two interfaces into a single factory that can produce instances of Objective-C objects or provide a reference to an instance representing the Objective-C class.

How do we define the methods in the two interfaces?

Reading the iOS documentation for the `UIDevice` class, you can read the signatures of the methods and properties. Let's translate some of the most significant signatures.

The first property we want to translate is `model`. The `model` property returns the model of the device (this can be `iPhone` or `iPod touch` or other values identifying the device model). This property is read-only.

The following is the complete signature:

```
@property(nonatomic, readonly, retain) NSString *model
```

In Object Pascal, it is translated as follows:

```
function model: NSString; cdecl;
```

As you can see, a read-only property is mapped to a function with the name of the property as the function name, and with the Objective-C property type as the Object Pascal return value. However, what about the R/W (read/write) properties?

The next property we want to translate is `proximityMonitoringEnabled`—an R/W property of the type boolean indicating whether proximity monitoring is enabled or not.

The following is the complete signature:

```
@property(nonatomic,getter=isProximityMonitoringEnabled)
                            BOOL proximityMonitoringEnabled
```

In Object Pascal the preceding code is translated as follows:

```
procedure setProximityMonitoringEnabled(
          proximityMonitoringEnabled: Boolean); cdecl;
function isProximityMonitoringEnabled: Boolean; cdecl;
```

An R/W property is mapped to a procedure (the setter) and function (the getter). The procedure name starts with `set` followed by the Objective-C property name (`proximityMonitoringEnabled` becomes `setProximityMonitoringEnabled`) and accepts a parameter of the same type as the property. The function name is defined by the property signature; in this case, it is `isProximityMonitoringEnabled`, returning a value of the same type as the property. If the property signature does not impose the getter name, the translation is similar to the following:

- Objective-C:

  ```
  @property(nonatomic, retain) NSString *accessibilityLabel
  ```

- Delphi:

  ```
  function accessibilityLabel: NSString; cdecl;
  procedure setAccessibilityLabel(accessibilityLabel: NSString);
  cdecl;
  ```

The `UIDevice` import looks like the following (only some methods were imported):

```
UIDeviceClass = interface(NSObjectClass)
    ['{A2DCE998-BF3A-4AB0-9B8D-4182B341C9DF}']
    function currentDevice: Pointer; cdecl;
end;

UIDevice = interface(NSObject)
    ['{70BB371D-314A-4BA9-912E-2EF72EB0F558}']
    function batteryLevel: Single; cdecl;
```


www.allitebooks.com

```
    function batteryState: UIDeviceBatteryState; cdecl;
    function isBatteryMonitoringEnabled: Boolean; cdecl;
    function isMultitaskingSupported: Boolean; cdecl;
    function isProximityMonitoringEnabled: Boolean; cdecl;
    function localizedModel: NSString; cdecl;
    function model: NSString; cdecl;
    function name: NSString; cdecl;
    function orientation: UIDeviceOrientation; cdecl;
    procedure playInputClick; cdecl;
    function proximityState: Boolean; cdecl;
    function systemName: NSString; cdecl;
    function systemVersion: NSString; cdecl;
    function uniqueIdentifier: NSString; cdecl;
  end;
  TUIDevice = class(TOCGenericImport<UIDeviceClass, UIDevice>)
  end;
```

Now, the `UIDevice` class can be used as follows (however, a single use is suggested, using the `currentDevice` property as singleton; here, it is used as a normal instance just to illustrate):

```
var
  device: UIDevice;
begin
  device := TUIDevice.Create;
  ShowMessage(NSStrToStr(device.model));
end;
```

Note that the class methods defined in the `UIDevice` class can also be used by Delphi. You don't need to create an instance (just like normal class methods), but the returning pointer must be wrapped in the appropriate class type:

```
var
  device: UIDevice;
  model: string;
begin
  //wraps the pointer to the proper type using the Wrap method
  device := TUIDevice.Wrap(TUIDevice.OCClass.currentDevice);
  model := NSStrToStr(device.model);
  ShowMessage(model);
end;
```

There's more...

The topic about the Objective-C class imports is huge and a deep explanation of it is out of the scope of this book. However, if you are interested and want to know more, I suggest you check the wonderful presentation held by Brian Long at the CodeRage 8 conference where he talks about accessing the iOS and Android APIs. The presentation can be found at the following URL:

`http://blog.blong.com/2013/10/my-coderage-session-files.html`

As FireMonkey and the mobile "soul" of Delphi mature, third-party mobile components start to be available to the market. Even if you are not interested in the native widget, you can study the code from the project *D.P.F Delphi iOS Native Components*. It really helps to see how to import complex Objective-C classes into Delphi (`http://sourceforge.net/projects/dpfdelphiios/`).

Displaying PDF files in your app

In the mobile world, you often need to show PDF files to your user. Maybe these PDF files are used as reports (usually generated by some reporting tool on the remote server), a statement about something that the user should do, a small book, or simply as a products catalog. So, how do we show a PDF that is deployed within the app or downloaded from some remote server and stored locally? How do we do it on Android and iOS? This is the topic of this recipe.

Getting ready

Let's say we have to create an app that contains some PDF files. In this case, we don't download the files but simply deliver them within the app. Later, we'll see how to download them from the network.

To deploy additional files within our app, we have to use the Deployment Manager, which is accessible by navigating to **Project** | **Deployment**. If you need to know how to use it, check the Embarcadero documentation at (`http://docwiki.embarcadero.com/RADStudio/XE6/en/Deployment_Manager`).

The additional file will be deployed in the private documents folder. Under Android, the private documents folder is identified as `./asset/internal`, while on iOS, it is identified as `.\Startup\Documents`. Using the Deployment Manager, place a PDF file in these folders for each platform so that it will be included in the generated app package.

www.allitebooks.com

How to do it...

All the required code to show the PDF is encapsulated in a single unit called `xPlat.OpenPDF.pas`. The main form contains a button, which once clicked calls the function `OpenPDF` passing the name of the file to be seen:

```
procedure TMainForm.btnOpenPDFClick(Sender: TObject);
begin
  OpenPDF('samplefile.pdf');
end;
```

Let's analyze the `OpenPDF` function in the `xPlat.OpenPDF.pas` unit. The following is the complete code:

```
unit xPlat.OpenPDF;

interface

procedure OpenPDF(const APDFFileName: string);

implementation

uses
  System.SysUtils, IdURI, FMX.Forms, System.Classes,
  System.IOUtils, FMX.WebBrowser, FMX.Types, FMX.StdCtrls
{$IF defined(ANDROID)}
    , Androidapi.JNI.GraphicsContentViewText
    , FMX.Helpers.Android
    , Androidapi.Helpers
    , AndroidSDK.Toast
    , Androidapi.JNI.Net
    , Androidapi.JNI.JavaTypes
{$ENDIF}
{$IF defined(IOS)}
    , iOSapi.Foundation
    , Macapi.Helpers
    , FMX.Helpers.iOS
    , FMX.Dialogs
{$ENDIF}
    ;

{$IF defined(ANDROID)}

procedure OpenPDF(const APDFFileName: string);
var
  Intent: JIntent;
```

```
  FilePath, SharedFilePath: string;
begin
  FilePath := TPath.Combine(TPath.GetDocumentsPath, APDFFileName);
  SharedFilePath := TPath.Combine(
                    TPath.GetSharedDocumentsPath, APDFFileName);
  if TFile.Exists(SharedFilePath) then
    TFile.Delete(SharedFilePath);
  TFile.Copy(FilePath, SharedFilePath);

  Intent := TJIntent.Create;
  Intent.setAction(TJIntent.JavaClass.ACTION_VIEW);
  Intent.setDataAndType(
         StrToJURI('file://' + SharedFilePath),
         StringToJString('application/pdf'));
  try
    SharedActivity.startActivity(Intent);
  except
    on E: Exception do
      ShowToast('Cannot open PDF' + sLineBreak +
        Format('[%s] %s', [E.ClassName, E.Message]),
        TToastDuration.Long);
  end;
end;
{$ENDIF}

{$IF defined(IOS)}
type
  TCloseParentFormHelper = class
  public
    procedure OnClickClose(Sender: TObject);
  end;

procedure TCloseParentFormHelper.OnClickClose(Sender: TObject);
begin
  TForm(TComponent(Sender).Owner).Close();
end;

procedure OpenPDF(const APDFFileName: string);
var
  NSU: NSUrl;
  OK: Boolean;
  frm: TForm;
  WebBrowser: TWebBrowser;
  btn: TButton;
  evnt: TCloseParentFormHelper;
begin
  frm := TForm.CreateNew(nil);
```


www.allitebooks.com

```
  btn := TButton.Create(frm);
  btn.Align := TAlignLayout.Top;
  btn.Text := 'Close';
  btn.Parent := frm;
  evnt := TCloseParentFormHelper.Create;
  btn.OnClick := evnt.OnClickClose;
  WebBrowser := TWebBrowser.Create(frm);
  WebBrowser.Parent := frm;
  WebBrowser.Align := TAlignLayout.Client;
  WebBrowser.Navigate('file://' + APDFFileName);
  frm.ShowModal();
end;
{$ENDIF}

end.
```

Showing the PDF file on Android

To show the PDF file on Android, we've used an Android-specific mechanism called **intents** (check the *Sending Android intents* recipe to learn more about Android intents). The file is actually shown by an external app already installed on the device; if such an app is not present, the PDF will not be shown, rather a message will be shown to the user. You can install Adobe PDF Reader or another app capable of showing PDF files that is "intent-compatible" with the one from Adobe. To comply with the Android I/O security and let another app read the PDF file in our `assets/internal` folder, we have to copy the file from the private documents folder, which is private to the app and not accessible from other apps, to the shared documents folder (readable from all the other apps installed on the device).

Just after the copy, we create an intent and configure it to launch an app capable of showing that PDF. The configuration is simple enough as shown:

```
//create the Intent directly from the Android SDK
Intent := TJIntent.Create;
//We need to show the PDF, so ACTION_VIEW is ok
Intent.setAction(TJIntent.JavaClass.ACTION_VIEW);
//Where is the file? Which mime type?
Intent.setDataAndType(
       StrToJURI('file://' + SharedFilePath),
       StringToJString('application/pdf'));
try
  //ask to the OS to find a proper app to handle the intent
  SharedActivity.startActivity(Intent);
except
  //TODO: there aren't apps able to show the PDF. Inform the user!
end;
```

Showing the PDF file on iOS

On iOS, there aren't intents, but we can use another mechanism to show our PDF file. The iOS **WebView** component can show PDFs, so we create a form on the fly containing WebView and a button to close the form. The `OpenPDF` iOS implementation does not use iOS-specific mechanisms apart from the WebView capabilities.

After having created the form at runtime (remember that if you don't have an `fmx` file associated with the `TForm` instance, you cannot use `TForm.Create()` to create the form; you'll have to use `TForm.CreateNew()`). The code is reported with some comments, as shown:

```
//create the form without an fmx
frm := TForm.CreateNew(nil);
//create the button used to close the form.
//On iOS there is not a "back" button as in Android
btn := TButton.Create(frm);
btn.Align := TAlignLayout.Top;
btn.Text := 'Close';
btn.Parent := frm;
evnt := TCloseParentFormHelper.Create;
//set the Button OnClick event handler
btn.OnClick := evnt.OnClickClose;
//create the TWebBrowser component which wraps the iOS WebView
WebBrowser := TWebBrowser.Create(frm);
WebBrowser.Parent := frm;
WebBrowser.Align := TAlignLayout.Client;
//point the web browser to the local file under the private folder
WebBrowser.Navigate('file://' + APDFFileName);
frm.ShowModal();
```

There's more...

This code does its job. However, Android and iOS users don't have the same user experience. On Android, you can use whatever app you have installed on the device to show the PDF, so you can also change the file with annotations, highlights, and by drawing directly on the file. Note that the file is also readable from other apps. This can be a problem in some situations. On iOS, conversely, you cannot modify the PDF with annotations and so on. You also don't have full control on the file and the file remains "private" for your app. These facts must be carefully analyzed and you have to be aware of the pros and cons about every choice you make. If you want to provide a uniform set of functionalities, additional work and third-party components and libraries will be needed.

One particular mention is required for the **TMS iCL** component suite (`http://www.tmssoftware.com/site/tmsicl.asp`). It is specific for FireMonkey on iOS (so that it doesn't compile on Android) but contains a component called `TTMSFMXNativePDFLib`, which can create new PDFs, open the existing PDF documents, and so on.

www.allitebooks.com

Using Google Docs Viewer

If your PDF is located on a public URL, you can also use the PDF visualizer included in Google Docs. Point WebView to the following URL and your PDF will show up:

```
"https://docs.google.com/gview?embedded=true&url="+PDFURL
```

Downloading the PDF file from the server

Let's say we have an application server that generates reports from some database data and saves them as PDF files.

We can download these files simply by using a `TidHTTP` component and storing them locally using something similar to the following code:

```
var
  FileStream: TStream;
  FilePath: String;
begin
   FilePath := TPath.Combine(
   TPath.GetSharedDocumentsPath, 'myreport.pdf');
   FileStream := TFileStream.Create(FilePath, fmCreate);
   try
     IdHttp1.Get(
     'http://www.myserver.com/reports/myreport.pdf', FileStream);
   finally
     FileStream := nil;
   end;
end;
```

Sending Android intents

One of the most useful things about the Android development is the dispatching mechanism of intents. The Android developer site (`http://developer.android.com/reference/android/content/Intent.html`) says:

> *"An intent is an abstract description of an operation to be performed.*
>
> *...*
>
> *An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed."*

Intents are widely used in Android, and if you want to fully integrate your Delphi app with the Android OS, you will probably have to deal with intents. Delphi uses intents internally to deal with some fundamental Android services (`TShareSheetAction`, `TTakePhotoFromCameraAction`, and so on). In this recipe, we'll see how to directly use intents in our app with the help of some examples.

Getting ready

The primary, and mandatory, pieces of information in an intent are as follows:

- `action`: This is the general action to be performed, such as `ACTION_VIEW`, `ACTION_EDIT`, and `ACTION_MAIN`
- `data`: This is the data to operate on such as a person's record in the contacts database expressed as a URI

There are two kinds of intents: explicit and implicit. They are explained as follows:

- **Explicit intent**: The app defines the target component directly in the intent
- **Implicit intent**: The app asks the Android system to evaluate registered components based on the intent data and other optional information

Using Java and the Android SDK you can send an implicit intent with the following code:

```
Intent myIntent = new Intent(
Intent.ACTION_VIEW, Uri.parse("http://www.danieleteti.it"));
startActivity(myIntent);
```

The preceding code asks the Android system to view a web page. If the OS finds that an activity can handle this kind of information (based on the action and data), then that activity will be started and the intent data will be passed to it.

Intents are also available to Delphi users. The previous Java code can be translated in Delphi as follows:

```
var
  Intent: JIntent;
begin
  Intent := TJIntent.Create;
  Intent.setAction(TJIntent.JavaClass.ACTION_VIEW);
  Intent.setData(StringToJString('http://www.danieleteti.it'));
  SharedActivity.startActivity(Intent);
end;
```

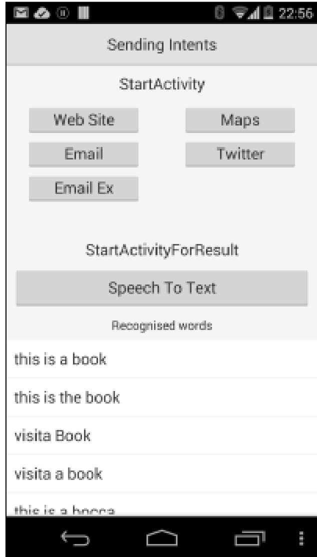

www.allitebooks.com

As you can see, the code is very similar to the Java version. Note that this code is no longer compliable on any platform but Android, so if you want to add this code in a cross-platform app (for Android, iOS, Windows, and Mac OS X), you will have to surround it with some IFDEFs.

There are a lot of components that can respond to some kind of intent. The Android documentation is very good on this topic. In this recipe, we'll open a web page, start Google Maps pointing to a specific address, open an e-mail client, open the Twitter app, and ask for speech-to-text recognition.

How to do it...

In the main form, there are six buttons, a listbox, and some labels. The following is how the form is rendered at runtime (after using it to recognize the phrase "this is a book"):



The app with the five buttons that will send intents

Let's open the project `SendingAndroidIntents.dproj` and start studying it.

The first four buttons, as you can see while reading the events handler, call a form method called `LaunchViewIntent` passing a URI:

```
procedure TMainForm.btnMapsClick(Sender: TObject);
begin
  //launch Google Maps (or similar app)
  LaunchViewIntent(
        'geo://0,0?q=Piazza del Colosseo 1,00184 Roma');
end;
```

```
procedure TMainForm.btnEmailClick(Sender: TObject);
begin
  //launch an email client with an empty email
  LaunchViewIntent('mailto:daniele.teti@gmail.com', false);
end;

procedure TMainForm.btnTwitterClick(Sender: TObject);
begin
  //launch twitter client (if installed)
  LaunchViewIntent('http://twitter.com/danieleteti');
end;
```

The `LaunchViewIntent` procedure is defined as follows:

```
procedure TMainForm.LaunchViewIntent(AURI: string;
AEncodeURL: boolean);
var
  Intent: JIntent;
  URI: JString;
begin
  if AEncodeURL then
    AURI := TIdURI.URLEncode(AURI);
  Intent := TJIntent.Create;
  Intent.setAction(TJIntent.JavaClass.ACTION_VIEW);
  URI := StringToJString(AURI);
  Intent.setData(TJnet_Uri.JavaClass.parse(URI));
  SharedActivity.startActivity(Intent);
end;
```

The preceding method executes all the steps needed to create an intent with the purpose of showing something; indeed, the action `ACTION_VIEW` means, "I want to view something" and asks the OS to show the information described in the data property (and other intent properties, if present).

Firstly, we check if the URI needs to be encoded; if yes, we use the `TidURI.URLEncode` method from the INDY library to perform the encoding. Then, an intent is created and configured with `ACTION_VIEW` as the action and the passed URI as the data. Having the intent configured, the last thing to do is ask the OS what the intent is for. In this case, we want to start an activity capable of performing the work defined in the intent. The activity used by the FireMonkey framework is accessible using the `SharedActivity` function from the RTL. So the last line uses `SharedActivity.startActivity` to actually send the intent. This kind of intent is the most simple.

www.allitebooks.com

More complex intents – sending a full flagged e-mail

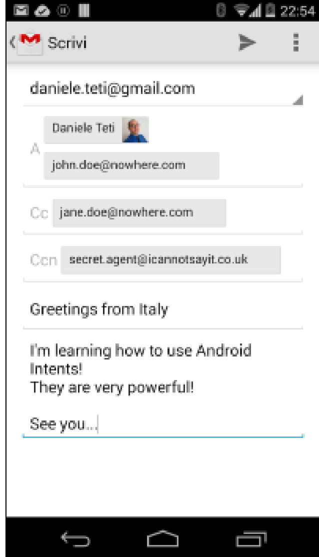
The fifth button with the caption **Email Ex** sends an e-mail just like the **Email** button but is more powerful because the prepared e-mail will also have the subject, body, and CC and BCC fields correctly filled. Let's see how it is possible.

In this case, the simple `ACTION_VIEW` with some data is not enough. The following is the code used to send a more complex e-mail:

```
procedure TMainForm.btnEmailExClick(Sender: TObject);
var
  Intent: JIntent;
  URI: JString;
  AddressesTo: TJavaObjectArray<JString>;
  AddressesCC, AddressesBCC: TJavaObjectArray<JString>;
begin
  Intent := TJIntent.Create;
  Intent.setAction(TJIntent.JavaClass.ACTION_SENDTO);
  Intent.setData(
         TJnet_Uri.JavaClass.parse(StringToJString('mailto:')));
  AddressesTo := TJavaObjectArray<JString>.Create(2);
  AddressesTo.Items[0] := StringToJString('daniele.teti@gmail.com');
  AddressesTo.Items[1] := StringToJString('john.doe@nowhere.com');

  AddressesCC := TJavaObjectArray<JString>.Create(1);
  AddressesCC.Items[0] := StringToJString('jane.doe@nowhere.com');

  AddressesBCC := TJavaObjectArray<JString>.Create(1);
  AddressesBCC.Items[0] :=
               StringToJString('backup@mywebsite.com');

  Intent.putExtra(TJIntent.JavaClass.EXTRA_EMAIL, AddressesTo);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_CC, AddressesCC);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_BCC, AddressesBCC);
  Intent.putExtra(TJIntent.JavaClass.EXTRA_SUBJECT,
         StringToJString('Greetings from Italy'));
  Intent.putExtra(TJIntent.JavaClass.EXTRA_TEXT,
    StringToJString('I''m learning how to use Android Intents!'+
                    sLineBreak + 'They are very powerful!' +
                    sLineBreak + sLineBreak + 'See you...'));
  SharedActivity.startActivity(Intent);
end;
```

As you can see, we set more properties in the intent than the previous example. Also, `TJavaObjectArray<JString>` is used to pass a Delphi wrapper of a Java array to the intent. Also, note how generics can be used to talk to the Android SDK.

By tapping this button, you will get a fully prepared e-mail, as shown in the following screenshot; note how the subject, CC, and BCC fields have been filled using information sent by the intent:



Gmail ready to send the e-mail prepared by our app

Starting an activity for results – the SpeechToText engine

Sometimes you may want to get a result back from an activity when it completes its job. For example, you may start an activity that lets the user pick a photo from a photo gallery, and after using it, returns the selected image, or select a person from a list of contacts, and after using it, returns the contact that was selected.

To do this, we call the `SharedActivity.startActivityForResult` method. The result will come back through a FireMonkey message readable using the global `TMessageManager` instance.

The `startActivityForResult` method receives two parameters: the first one is the intent itself, while the second is an integer value that identifies the request code. This request code will be passed to the message handler when the activity ends. This is because `startActivityForResult` is not blocking. When the launched activity ends, you have to know from which request it had been launched.

www.allitebooks.com

When an activity exits, some data should be returned to its parent. It must always supply result code, which can be the standard results `RESULT_CANCELED` and `RESULT_OK`, or any custom values starting at `RESULT_FIRST_USER` (all these values are defined in the Android documentation at `http://developer.android.com/reference/android/app/Activity.html`). In addition, it can optionally return an intent containing any additional data it wants. All of this information appears again on the parent message handler along with the integer identifier it originally supplied.

The last button launches the `SpeechToText` engine activity, asks the user to say something, and then ends, and sends the possible recognized phrases to the parent activity:

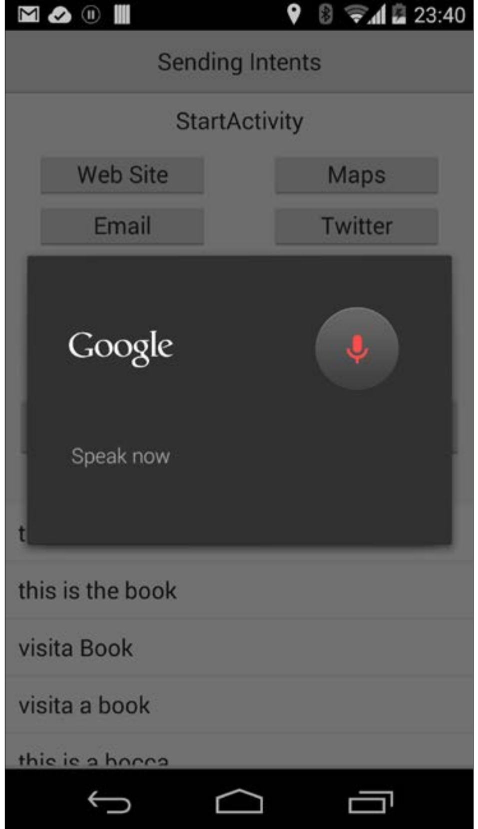
```
procedure TMainForm.btnSTTClick(Sender: TObject);
var
  Intent: JIntent;
  ReqCode: Integer;
const
  STT_REQUEST = 1001;
  ACTION_RECOGNIZE_SPEECH = 'android.speech.action.RECOGNIZE_SPEECH';
  EXTRA_LANGUAGE_MODEL = 'android.speech.extra.LANGUAGE_MODEL';
  EXTRA_RESULTS = 'android.speech.extra.RESULTS';
begin

  //assign a code to this request
  ReqCode := STT_REQUEST;
  //create and configure the intent (check android SDK docs)
  Intent := TJIntent.Create;
  Intent.setAction(StringToJString(ACTION_RECOGNIZE_SPEECH));
  Intent.putExtra(StringToJString(EXTRA_LANGUAGE_MODEL),
                                  StringToJString('en-US'));
  //when the launched activity ends, this handler will be called.
  //Here we've to read the data sent back from the launched activity
  TMessageManager.DefaultManager.SubscribeToMessage(
    TMessageResultNotification,
    procedure(const Sender: TObject; const Message: TMessage)
    var
      M: TMessageResultNotification;
      i: Integer;
      Words: JArrayList;
      TheWord: string;
    begin
      M := TMessageResultNotification(message);
      //is this request the right one?
       if M.RequestCode = ReqCode then
```

```
      begin
      //The request returned OK?
        if (M.ResultCode = TJActivity.JavaClass.RESULT_OK) then
        begin
          Words := M.Value.getStringArrayListExtra(
                   StringToJString(EXTRA_RESULTS));
          ListBox1.Clear;
          //if there are recognized words, fill the listbox
          if Words.size > 0 then
          begin
            ListBox1.BeginUpdate;
            try
              for i := 0 to Words.size - 1 do
              begin
                TheWord := JStringToString(JString(Words.get(i)));
                ListBox1.Items.Add(TheWord);
              end;
            finally
              ListBox1.EndUpdate;
            end;
          end
          else
            ShowToast('Some problems occurred');
        end
        else
          ShowToast('Nothing to recognize');
      end;
    end);

  //start the activity for result passing the specific ReqCode
  SharedActivity.startActivityForResult(Intent, ReqCode);
end;
```

The code is not simple, but the main parts are clearly identifiable. Firstly, we configure the intent to launch the speech recognizer. Then, before launching the intent, we subscribe to system messages of type `TMessageResultNotification`. This kind of message is sent by FireMonkey when an Android activity has been launched with `startActivityForResult`. Inside the message handler, we have to check if the message is from our launched activity (so, we check `ReqCode`), and whether the activity is returned with errors (so, we check `RESULT_OK`). If everything is okay, we can read the information contained in the returned intent (this time, the intent is used to send back information from the launched activity to the parent app).

www.allitebooks.com

The following screenshot shows the speech recognition activity:



The SpeechToText engine activity is listening

Play with the app and discover how different kinds of intents work.

There's more...

Intents are fundamental parts of the Android ecosystem. FireMonkey uses them in the components and RTL; a developer who wants to deeply integrate his app with the Android OS must know how intents work and the possibilities that they open up. Think, for example, that every app installed on your device can be elegantly integrated into your app without too much effort. A good point can be studying all the common intents available and usable in your Android device. You will learn useful things and get in touch with many practical uses of intents.

All the available "common" intents are explained in the article available at `http://developer.android.com/guide/components/intents-common.html`.

Letting your phone talk – using the Android TextToSpeech engine

In this recipe, we'll do some fun stuff. On your Android phone, run an app with a listening UDP server on it. When another application, in our case a VCL application, sends a UDP broadcast with some text, the Android app will pronounce the text using the Android TTS engine.

Getting ready

The first thing to do is import the TTS classes from the Android SDK in our Delphi project. This is not a simple task; however, luckily, someone already did the job. Indeed, Jeff Overcash, the maintainer of the **InterBase Express** (**IBX**) components wrote **Android Text To Speech JNI Translation**. His translation with a simple demo app is available at CodeCentral (`http://cc.embarcadero.com/item/29594`).

In this recipe, we'll use the imported classes to let our Android device read the text sent via a UDP broadcast. Note that the message will be read by each device that receives it. Thus, if you have two, three, or four phones, you will be able to listen to the message read by all the phones simultaneously.

How to do it...

Open the project group containing the mobile app and VCL application.

In the mobile app, we have an empty form with a label on it aligned to `Client`. In the form startup (1 second after the form creation), we configure the TTS engine with the following code:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled := False;
  FTTS := TJTextToSpeech.JavaClass.init(
          SharedActivityContext, FTTSListener);
end;
```

The `FTTSListener` instance is a `TJavaLocal` descendant implementing the required `JTextToSpeech_OnInitListener` interface. The TTS system gets initialized and, when done, the listener `OnInit` method is called (check the `TTSListenerU.pas` unit). However, if the TTS engine is correctly initialized, we have to configure it, setting the language to be used when it will talk. So, in the listener constructor, I've added an anonymous method that will be called by the listener to configure the engine after initialization. The code is written inside the `FormCreate` event handler, as shown in the following code snippet:

```
constructor TMainForm.Create(AOwner: TComponent);
begin
  inherited;
```


www.allitebooks.com

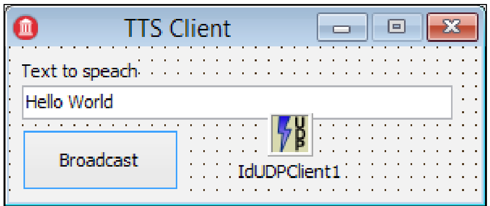
```
    FTTSListener := TttsOnInitListener.Create(
      procedure(AInitOK: boolean)
      var
        Res: Integer;
      begin
        if AInitOK then
        begin
          Res := FTTS.setLanguage(TJLocale.JavaClass.ENGLISH);
          if (Res = TJTextToSpeech.JavaClass.LANG_MISSING_DATA) or
            (Res = TJTextToSpeech.JavaClass.LANG_NOT_SUPPORTED) then
            Label1.Text := 'Selected language is not supported'
          else
          begin
            Label1.Text := 'READY To SPEAK!';
            IdUDPServer1.Active := True;
          end;
        end
        else
          Label1.Text := 'Initialization Failed!';
      end);
  end;
```

If the configuration goes well, `TidUDPServer`, configured to listen on all interfaces on port `9999`, is activated. In the `idUDPServer1.OnUDPRead` event handler, there is a hook between the data sent over the network and the TTS engine:

```
procedure TMainForm.IdUDPServer1UDPRead(
   AThread: TIdUDPListenerThread;
   const AData: TIdBytes; ABinding: TIdSocketHandle);
var
  bytes: TBytes;
begin
  bytes := TBytes(AData);
  Speak(TEncoding.ASCII.GetString(bytes));
end;

procedure TMainForm.Speak(const AText: string);
begin
  FTTS.Speak(
    StringToJString(AText),
    TJTextToSpeech.JavaClass.QUEUE_FLUSH, nil);
end;
```

The method `Speak` is the entry point to the TTS engine. The mobile app is now completed. Now, let's talk about the VCL application that has to send the UDP packets.

Open the `VCLTTSClient` project and you will see a form similar to the following screenshot:



The simple VCL form that will send the UDP messages to the mobile app

This application is even simpler than the mobile one. Shortly after, when the user clicks on the button, the event handler sends the text entered in the textbox to all the available broadcast addresses (considering its subnet as a Class C network). In other words, if the PC where the application is running has a single IP address, let's say `192.168.1.50`, the UDP packet is sent to the broadcast address `192.168.1.255`, and so on for each Ethernet interface configured on the Windows machine (to get all the IP addresses, I've used a handy class named `TIdStackLocalAddressList` that comes with INDY. Moreover, this is just a demo; if you want to be sure about the broadcast addresses, you will have to do some additional work, but this is a network-specific topic and is out of the scope of this book). To replace the last address part (for example, `.50` must become `.255`), I've used a simple regular expression to replace the last octet. Note that the following code actually works only for IPv4 addresses:

```
procedure TMainForm.btnSendClick(Sender: TObject);
var
  CurrIP, BrdcstIP: string;
  i: Integer;
begin
  for i := 0 to FAddressesList.Count - 1 do
  begin
    if FAddressesList.Addresses[i].IPVersion = Id_IPv4 then
    begin
      CurrIP := FAddressesList.Addresses[i].IPAddress;
      BrdcstIP := FToIPv4Broadcast.Replace(CurrIP, '.255');
      IdUDPClient1.Broadcast(Edit1.Text, 9999, BrdcstIP);
    end;
  end;
end;
```


www.allitebooks.com

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FAddressesList := TIdStackLocalAddressList.Create;
  GStack.GetLocalAddressList(FAddressesList);
  FToIPv4Broadcast := TRegEx.Create('\.\d{1,3}$');
end;
```

That's it! Run the mobile app on your Android phone and check that it is currently connected to the same Wi-Fi where the PC is connected. Then, run the VCL application on your PC, write something in the textbox, and hit the button. Your Android device should now start talking.

There's more...

The option of setting up TCP or UDP servers on our mobile devices opens a great range of possibilities. However, you should open ports on your phone conscientiously.

Thanks to Jeff Overcash, the TTS wrapper has greatly simplified the work required to let an Android phone talk. If you want to go deeper when using the TTS engine, you should read the following Android documentations:

- The Java documentation about the main class used in this recipe (`http://developer.android.com/reference/android/speech/tts/TextToSpeech.html`)
- The Java package where the classes have been imported from (`http://developer.android.com/reference/android/speech/tts/package-summary.html`)
- An introduction to the TTS engine in Android (`http://android-developers.blogspot.it/2009/09/introduction-to-text-to-speech-in.html`)

www.allitebooks.com

Index

A

actions 214
activity
  starting, for results 296-299
ADOM XML 38
Android
  PDF file, displaying on 289
Android2DelphiImport
  URL 281
Android configuration
  URL 214
Android intents
  sending 291-294
Android SDK Java classes
  using 276-281
Android Text To Speech JNI Translation 300
animations
  about 102
  clients, impressing with 102-105
  references 105
anonymous methods 60
Apache HTTP Server
  URL, for security tips 212
Apache Lounge community
  URL, for downloading 203
Apache modules, DataSnap
  creating 203-211
app
  PDF files, displaying in 286
  phone call, making from 265-268
application life cycle
  tracking 269-273
applications
  customizing, VCL styles used 8, 9
App Tethering
  about 197
  references 202, 203
  used, for creating companion app 197-202
App Tethering model application
  functionalities 199

B

backend
  working with 257-264

C

callback, Delphi
  URL 127
C++Builder XE6
  VCL styles 10
class
  configuring, at runtime 68-71
  helpers, creating for 76-84
class helper
  references 84
clients
  impressing, with animations 102-105
Codice Fiscale 87
Command design pattern
  reference link 222
companion app
  creating, App Tethering used 197-202
complex intents 295
complex vector shapes
  displaying, paths used 116-120
configuration, class
  at runtime 68-71

console service application
converting, to Windows service 157-159
content types
reference link 171
Create Retrieve Update and Delete (CRUD) 150
custom VCL style
using 13

D

datagrams 191
dataset
serializing, to JSON 160-163
DataSnap
about 211
Apache modules, creating 203-211
references 211
datatypes, JSON 27
default keyboard shortcuts
references 112
Delphi
references 64, 127
VCL styles 10
Delphi documentation, built-in RegEx engine syntax
references 89
Delphi language 59
DelphiMVCFramework (DMVCFramework)
about 148
downloading 148
references 148, 149, 161, 176
Delphi samples, UDP programming
references 197
Delphisorcery
URL 127
Delphi Web Script library
URL 31
Delphi XE5 Mobile REST Client Demo
URL 233, 264
Delphi XE5 Mobile REST Client Demo Source
URL 264
Delphi XE7 282
Deployment Manager 286
deserialization 164
DHCP
reference link 191
DoGetCurrent method 67
DoMoveNext method 67
D.P.F Delphi Android Native Components
URL 282
draw combos
using 20-22
Duck typing
about 75
reference 72
RTTI used 72-75

E

enumerable types
about 67
references 68
writing 64-67
event 140
event objects
references 143
ExecuteAsynch method 232
explicit intent 292
extended RTTI
reference 72
eXtensible Markup Language. *See* **XML**
external open source projects
DelphiMVCFramework 148
jTable 148

F

file extension
associating, with application on Windows 53-58
Filter function 60, 63
FireDAC components 152
FireDAC IBLite with Delphi XE6
URL 264
FireMonkey
about 92
references, for image effects 222
using, in VCL application 122-127
firemonkey-container
URL 127

www.allitebooks.com

FireMonkey controls
styles, used for customizing 92-97
FireMonkey Style Designer (FSD)
about 94
references 95
FireMonkey styles
references 97, 102
FMX.Media.TCameraComponent
reference link 257
FormCreate event
about 140
parameter 137, 138
FOutputFile variable 132
full flagged e-mail
sending 295
functionalities, App Tethering model application
actions, sharing 199
resources, sharing 199
streams, sending 199
strings, sending 199

G

GetEnumerator method 64
Google Docs Viewer
using 291

H

helpers
creating, for classes 76-84
higher-order functions
Filter 60, 63
Map 60, 61
Reduce 60, 62
reference 60
using 60-64
http-equiv meta tags
URL 255
HTTP protocol
references 171
HTTP Verb
DELETE 175
GET 175
POST 175
PUT 175

I

Icojam
references 23
idempotence
reference link 190
implicit intent 292
intents
about 289, 299
explicit 292
implicit 292
reference link 299
intents, Android
reference link 291
sending 291-294
InterBase Express (IBX) 300
iOS
PDF file, displaying on 290
iOS configuration
URL 214
iOS Objective-C SDK classes
using 283-285
ISAPI DLLs 156

J

Java2Pas
URL 281
JavaScript Object Notation. *See* **JSON**
jQuery-UI CSS
reference link 151
JSON
about 26
dataset, serializing to 160-163
datatypes 27
manipulating 27-31
objects, serializing to 165-170
URL 26
JSON Delphi library
URL 31
JSON parsers
JSON Delphi library 31
Superobject 31
jTable
about 148, 153
references 148, 154

L

listboxes
using 20-22
listview
about 247
used, for displaying local data 222-227
used, for searching local data 222-227
LiveBindings
about 105, 115, 238
master/details (M/D) relationship, using with 105-115
references 106, 116, 238
Location Sensors
reference link 257

M

Map function 60, 61
master/details (M/D) relationship
using, with LiveBindings 105-115
Method Draw
about 121
URL 121
methods, TDataSet descendants
GetEnumerator 77
SaveToCSV 77
Microsoft Message Compiler
about 53
reference 53
Mobile Preview
about 227
URL 227
Monitor
reference link 136
monitoring system implementation
about 247
client side 248-251
server side 252-256
MonkeyMixer, Delphi XE5
URL 127
MSXML 38
multiple threads
synchronizing, TEvent used 140-143
multithreading 129, 233

O

objects
serializing, to JSON 165-170
Observer design pattern
about 274
URL 26
OmniThreadLibrary
URL 146
OnDrawITem function 23
oscilloscope
implementing 143-145
owner drawing 20

P

packets 191
paths
used, for displaying complex vector shapes 116-120
PDF files
displaying, in app 286
displaying, on Android 289
displaying, on iOS 290
downloading, from server 291
phone call
making, from app 265-268
photo
effects, applying 214-221
sharing 214-221
taking 214-221
Plain Old Delphi Object (PODO) 67
POST HTTP request encoding parameters
sending 171-173
POSTMan Chrome extension 190
PrepareResponse method 154
presentation, Android and iOS APIs access
URL 282, 286
producer/consumer design pattern
reference 121

R

record helper
references 84
Reduce function 60, 62

www.allitebooks.com

regular expression (RegEx)
about 84
references 84, 89, 222
strings, checking with 84-89
RemObjects Hydra4
URL 127
remote applications
controlling, UDP used 191-196
Representational state transfer (REST) 174
REST Client library
about 174
reference link 174, 233
RESTDemo sample
reference link 174
RESTful interface
implementing, WebBroker used 174-189
results
activity, starting for 296-299
reverse proxy
reference link 156
Richardson Maturity Model (RMM) 174
RTTI
about 71
objects, serializing to JSON 165-170
runtime
class, configuring at 68-71

S

Scalable Vector Graphic. *See* **SVG**
serialization 164
server
PDF file, downloading from 291
shared resources
synchronizing, with TMonitor 129-136
SQLite databases
used, for handling to-do list 234-238
stack of embedded forms
creating 23-25
startActivityForResult method 296
streams
about 38
utilization examples 39-41
streams types
System.Classes.TBinaryReader 39
System.Classes.TBinaryWriter 39
System.Classes.TReader 39
System.Classes.TStreamReader 39
System.Classes.TStreamWriter 39
System.Classes.TStringReader 39
System.Classes.TStringWriter 39
System.Classes.TTextReader 39
System.Classes.TTextWriter 39
System.Classes.TWriter 39
strings
checking, with regular expression (RegEx) 84-89
stunning FireMonkey GUI
creating 99-101
styled TListBox
creating 98-101
styled TListView
used, for handling long data list 239-246
styles
used, for customizing FireMonkey controls 92-97
Superobject
URL 31
SVG 116
SVG PATH data 121
System.Messaging.pas unit
reference link 274
System.Messaging.TMessageManager class
reference link 26

T

Table Data Gateway (TDG)
about 181
references 181
TActiveStyleObject style 95
TAnonymousThread<T> constructor, methods
function: T 47
procedure (E: Exception) 47
procedure (Value: T) 47
TApplicationEvent instance
BecameActive 271
EnteredBackground 271
FinishedLaunching 271
LowMemory 272
OpenURL 272
TimeChange 272
WillBecomeForeground 272

WillBecomeInactive 272
WillTerminate 272
TDBGrid
customizing 14-19
TEvent
used, for synchronizing multiple threads 140-143
TextToSpeech engine
reference link 303
using 300-303
TFileEnumerable type 65
third-party Java libraries, RAD Studio applications
URL 282
thread
block, avoiding 227-232
thread, on stack overflow
reference link 84
thread-safe queue
using 137-139
TJavaGenericImport class 277
TJSON class 171
TMemo component 28
TMonitor
about 129
shared resources, synchronizing 130-136
TMS iCL component suite
about 290
URL 290
toast
about 276
reference link 276
Toast class
constants 276
public instance methods 276, 277
public static methods 277
to-do list
handling, SQLite databases used 234-238
TortoiseSVN
URL, for downloading 148
TPathAnimation component 121
tray
VCL application, inserting in 42-48
tray icon
usage 48
TService.LogMessage method
using 52
TStreamWriter class 40
TStyleManager class
about 11
methods 13
TTetheringAppProfile component 197
TTetheringManager component 197
TThreadedQueue<T> class 137, 139
TXMLDocument, DOMVendor implementation
ADOM XML 38
MSXML 38
XSLT 38

U

UDP
about 190
features 191
used, for controlling remote applications 191-196
UIDevice class 283
URLs, WebBroker server
/deleteperson 151
/getpeople 151
/index.html 151
/saveperson 151
Uses Permissions
reference link 257

V

VCL 122
VCL application
FireMonkey, using in 122-127
inserting, in tray 42-47
style, modifying at runtime 10-13
VCL styles
about 8
references 10
used, for customizing applications 8, 9
Visual Component Library. *See* **VCL**
voice-over-Internet protocol (VoIP) 190

www.allitebooks.com

W

waGetPeopleAction action 152
WebBroker
  about 149
  references 150, 151, 156
  used, for implementing RESTful
    interface 174-189
  web client JavaScript application,
    creating 149-152
web client JavaScript application
  creating, with WebBroker 149-152
  people list, retrieving 152
  person's record, creating 154, 155
  person's record, deleting 156
  person's record, updating 154, 155
  running 156
WebFileDispatcher 152
WebView component 290
Windows configuration
  URL 214
Windows service
  about 52
  console service application, converting
    to 157-159
  creating 48-51

X

XML
  about 32
  documents, manipulating 32-37
  documents, transforming 32-37
  reference link 37
XML ecospace 37
XML Schema, JSON
  URL 32
XSLT
  about 38
  references 38

www.allitebooks.com

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

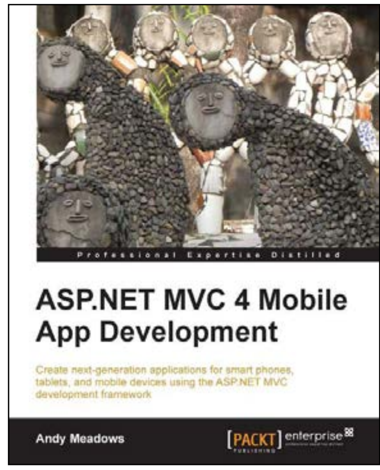
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

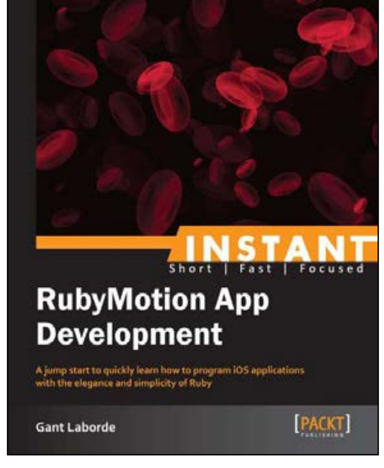


ASP.NET MVC 4 Mobile App Development

ISBN: 978-1-84968-736-2 Paperback: 356 pages

Create next-generation applications for smart phones, tablets, and mobile devices using the ASP.NET MVC development framework

1. Learn and utilize the latest Microsoft tools and technologies to develop mobile web apps with a native feel.
2. Create web applications for the traditional and mobile web.
3. Discover techniques used to overcome the pitfalls of developing Internet-ready apps.



Instant RubyMotion App Development

ISBN: 978-1-84969-652-4 Paperback: 54 pages

A jump start to quickly learn how to program iOS applications with the elegance and simplicity of Ruby

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Learn the structure of iPhone and iPad applications.
3. Discover how to simplify iOS apps with Ruby.
4. Get to grips with how to leverage Ruby libraries to quickly and efficiently write apps!

Please check **www.PacktPub.com** for information on our titles

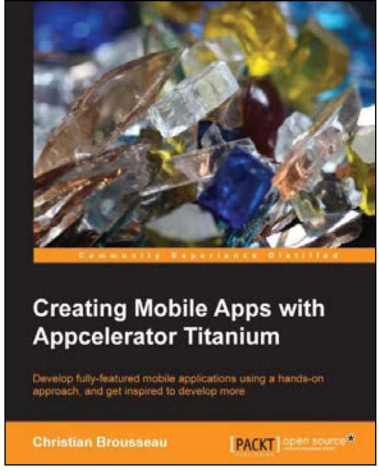
www.allitebooks.com

Building Web and Mobile ArcGIS Server Applications with JavaScript

ISBN: 978-1-84969-796-5 Paperback: 274 pages

Master the ArcGIS API for JavaScript, and build exciting, custom web and mobile GIS applications with the ArcGIS Server

1. Develop ArcGIS Server applications with JavaScript, both for traditional web browsers as well as the mobile platform.
2. Acquire in-demand GIS skills sought by many employers.
3. Step-by-step instructions, examples, and hands-on practice designed to help you learn the key features and design considerations for building custom ArcGIS Server applications.



Creating Mobile Apps with Appcelerator Titanium

ISBN: 978-1-84951-926-7 Paperback: 298 pages

Develop fully-featured mobile applications using a hands-on approach, and get inspired to develop more

1. Walk through the development of 10 different mobile applications by leveraging your existing knowledge of JavaScript.
2. Allows anyone familiar with some object-oriented programming (OOP), reusable components, and AJAX closures take their ideas and heighten their knowledge of mobile development.
3. Full of examples, illustrations, and tips with an easy-to-follow and fun style to make app development fun and easy.

Please check **www.PacktPub.com** for information on our titles