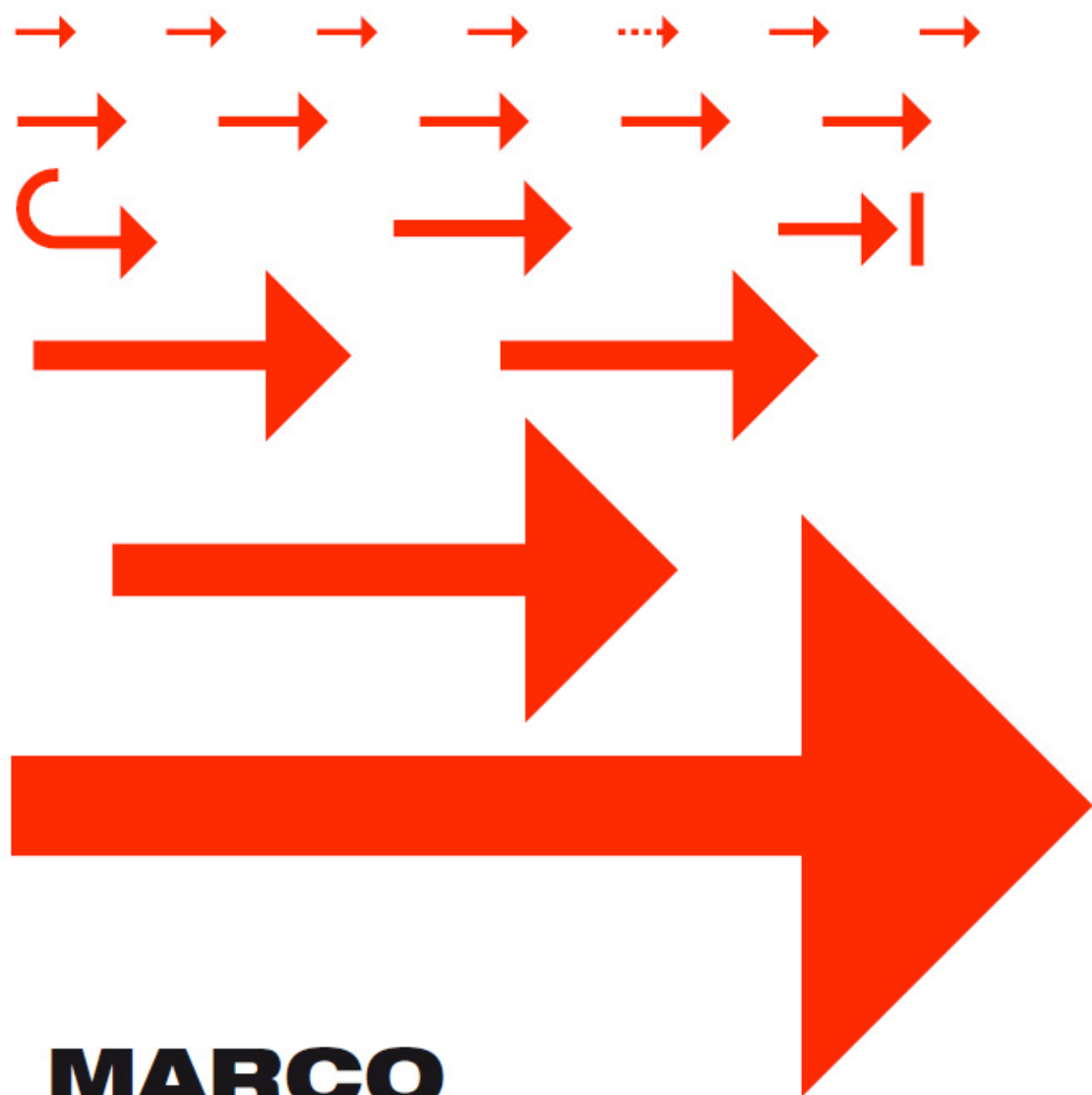




**MARCO
CANTÙ**

DELPHI XE HANDBOOK

Marco Cantù

Delphi XE Handbook

A Guide to New Features in Delphi XE

Piacenza (Italy), June 2011

Author: Marco Cantù

Publisher: Wintech Italia Srl, Italy

Editor: Peter W A Wood

Tech Reviewers: Holger Flick, Andrea Magni, Jeroen Pluimers, Simon J. Stuart

Cover Designer: Fabrizio Schiavi (www.fsd.it)

Copyright 2011 Marco Cantù, Piacenza, Italy. World rights reserved.

The author created example code in this publication expressly for the free use by its readers. Source code for this book is copyrighted freeware, distributed via the web site <http://code.marcocantu.com>. The copyright prevents you from republishing the code in print or electronic media without permission. Readers are granted limited permission to use this code in their applications, as long as the code itself is not distributed, sold, or commercially exploited as a stand-alone product.

Aside from this specific exception concerning source code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, in the original or in a translated language, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Delphi is a trademark of Embarcadero Technologies. Windows Vista and Windows Seven are trademarks of Microsoft. Other trademarks are of the respective owners, as referenced in the text. The author and publisher have made their best efforts to prepare this book, and the content is based upon the final release of the software. The author and publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accepts no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Delphi XE Handbook, First Edition, Revision 01.

ISBN-10: 1463600674

ISBN-13: 978-1463600679

Electronic edition sold by FastSpring, on behalf of Marco Cantù (see web site below for links). Any other download or sale outlet is likely to be illegal. Do not distribute the PDF version of this book.

More information on [**http://www.marcocantu.com/handbooks**](http://www.marcocantu.com/handbooks).

*Dedicated to my two wonderful kids,
Benedetta and Jacopo,
and their lovely mother, Lella.*

4 - Dedication

Introduction

Departing from the recent convention of using years in the product name, the latest version of what many consider the best development tool for Windows is now called “Delphi XE”.

The product delivers extra stability, improvements, and an interesting set of third party tools on top of what was a very good version, Delphi 2010. Delphi XE has a very nice set of new features, often little hidden gems, that this book tries to unveil in full.

Before you ask, XE is not specifically an acronym, but it is a name that Embarcadero added to each of this products to align them to a single naming and versioning scheme. All Embarcadero products released in 2010 have been called XE.

About This Book

The book you have in your hands (or on your screen) is a detailed documentation of all the new extensions in Delphi XE compared to Delphi 2010, with the (significant) exclusion of the DataSnap portion of the product. This area, in fact, has a lot of significant changes, but as I have recently written a white paper for Embarcadero covering the REST side of DataSnap. I decided it wasn't

6 - Introduction

worth including those extra 60 pages in the book, since you can already read them online at no additional cost.

That's why the book has only four chapters:

- 1: Delphi XE IDE
- 2: Integrated Tooling
- 3: Compiler and RTL
- 4: XE Libraries

As mentioned earlier, the book covers only new features in Delphi XE. So if you are looking for the RTTI or an introduction to JSON support you'll have to look for the Delphi 2010 Handbook, or if you are looking for information about Unicode you'll have to delve into my Delphi 2009 Handbook.

There is , however, a PDF collection of the four books, not edited but simply bound together, called **Delphi Handbooks Collection**. You can find information about this PDF on the specific section my web site:

■ <http://www.marcocantu.com/handbooks>

Source Code

Given that Delphi XE directly supports Subversion, I decided to put the repository of the source code of the book in a Subversion server, hosted at code.marcocantu.com. This makes it much easier for me to distribute updates and extra examples over time.

I decided to merge the code of all examples of my various book in a single project. The project home page is:

■ code.marcocantu.com/p/marcode1phibooks

From a Subversion client (don't use these links in your browser) you can download the entire project or the book specific code using these two URLs:

■ <http://code.marcocantu.com/svn/marcocantu/marcode1phibooks>
■ <http://code.marcocantu.com/svn/marcocantu/marcode1phibooks/delphixehandbook>

As an alternative you can browse the source code online, following the link from the project home page, or navigating from:

■ code.marcocantu.com/p/marcode1phibooks/source/tree/HEAD/delphixehandbook

Editor and Tech Reviewers

This book as seen the contribution of an editor and several tech reviewers, involved at various degrees, which provided a huge help and I won't be able to thank enough. The editor of this book (as of all my latest Delphi books) was Peter Wood, an IT professional who lives in Malaysia. I got technical feedback from Holger Flick, Andrea Magni, Jeroen Plumers, and Simon J. Stuart. All four are well known members of the Delphi developers community.

About Myself

I've been in the “Delphi book writing” business ever since the first version of the product, when I released the original “Mastering Delphi”. That was not my first writing experience, as I had previously written works on Borland C++ and the Object Windows Library. The Mastering Delphi series, published by Sybex, was one of the best-selling Delphi book series for several years, with translations into many languages and sold in bookshops all over the world. More recently I started self-publishing the Delphi Handbooks, available in print and as ebooks.

Beside writing, I keep myself busy with consulting (mostly on applications architectures), help selling Delphi in Italy, do code reviews, Delphi mentoring, and general consulting for developers. I'm a frequent speaker at Delphi and general developer conferences (in Europe and in the Unites States), including the online *CodeRage* conferences organized by Embarcadero.

Since 2009, Cary Jensen and I started giving public training in both US and Europe at the jointly organized Delphi Developer Days, about which you can find more information at:

■ <http://www.delphideveloperdays.com>

If you are interested in inviting me to speak at a public event or give a training session (on new Delphi features or any advanced Delphi subject) at your company location, feel free to send me a note by email.

Contact Information

To follow my activities you can use several online resources and communities. In the following list you can see my blog (which is quite active), my Twitter account, my Facebook page, my not-so-up-do-date personal site (a summary of my activities), and my company site (with training offers):

- | <http://blog.marcocantu.com>
- | <http://twitter.com/marcocantu>
- | <http://www.facebook.com/marcocantu>
- | <http://www.marcocantu.com>
- | <http://www.wintech-italia.com>

I have an online mailing list based at Google groups:

- | http://groups.google.com/group/marco_cantu

If you need tech support, use Embarcadero's newsgroups or Stack Overflow (I occasionally dwell in both locations):

- | <http://forums.embarcadero.com>
- | <http://stackoverflow.com>

If you need anything else but tech support, drop me an email at:

- | marco.cantu@gmail.com

Table Of Contents

Introduction.....	5
About This Book.....	5
Source Code.....	6
Editor and Tech Reviewers.....	7
About Myself.....	7
Contact Information.....	8
Table of Contents.....	9
Chapter 1: Delphi XE IDE.....	13
Managing Projects.....	14
Build Groups.....	15
The Configuration Manager.....	16
Build Tools Customization.....	17
Extensions to Searching.....	18
Search For Usages.....	18
Local Search Short Cut Keys.....	19
Editor.....	20
New Live Templates.....	20
XMLDoc Comments and Help Insight.....	22
Navigating Modified Code.....	23
Source Code Formatting.....	24
Formatter Profiles.....	25

10 - Table of Contents

Other Changes in Formatting.....	26
Form Designer.....	26
Updated Wizards.....	28
Install Component Wizard.....	28
Web Server Application.....	29
The Debugger.....	30
Named Threads.....	30
Run Without Debugging.....	30
More Debugger Features.....	31
Chapter 2: Integrated Tooling.....	33
Version Control Integration.....	34
Version Control Systems.....	34
Subversion.....	36
Subversion in Delphi.....	41
Beyond Compare Integration.....	43
The Updated History View.....	44
Version Insight and Its Extensions.....	46
UML Modeling.....	47
Classic Live Diagrams: Packages and Classes.....	47
Ancestor Classes in Class Diagrams.....	48
Sequence Diagrams.....	49
Audits, Metrics, and Documentation from the Command Line.....	50
Third-Party Tools.....	51
CodeSite.....	51
AQTime.....	54
FinalBuilder.....	55
Chapter 3: Compiler and RTL.....	59
The Delphi Compiler.....	60
Different Defaults, Different Result.....	60
No More String Checks.....	60
New Alignments.....	61
RTTI and Virtual Methods Interceptors.....	62
The Run Time Library.....	66
Dates and Files.....	67
Splitting Strings.....	68
Command Line Parsing.....	68
GUID Helper.....	69

Number of CPU Cores.....	70
Binary Reader and Writer.....	70
Regular Expressions.....	73
A Regular Expressions Primer.....	73
Regular Expressions in Delphi XE.....	75
The RegExPrimer Example.....	76
Threading and Synchronization.....	80
Anonymous Threads.....	80
Waiting, While Keeping Busy.....	84
Using a SpinWait.....	86
A Thread-Safe Queue.....	88
Chapter 4: XE Libraries.....	91
Minor VCL Changes.....	91
JSON Processing.....	92
JSON Parsing Cleaned Up.....	92
Google Translate Revisited.....	94
JSON and Tables.....	96
JSON Marshaling.....	97
XML Processing.....	100
WebBroker and Indy Web Servers	102
Building a Sample Server.....	104
SOAP Improvements.....	109
Microsoft's Azure Native Support.....	109
Delphi XE Components for Azure.....	110
The Azure101 Demo.....	111
The Azure Publish and Consume Demo.....	116
Amazon Web Services and More.....	120
Epilogue.....	123
DataSnap White Papers.....	123
Everything XE Demo.....	124
More Books Coming.....	125
Index.....	127

Chapter 1:

Delphi XE IDE

Our starting point for exploring Delphi XE is its Integrated Development Environment, or IDE. As one of the focal points of this release is programmer productivity, the IDE has received a number of new features, probably more than any other part of the product.

This follows a few other releases with a strong push in terms of IDE features. It was the recent Delphi 2010 that added IDE Insight¹, a tool that lets you press the F6 key to start a textual search throughout the entire development environment, looking for menu commands, new components, project options, recent files, components in the current designer, and much more.

Other recent features of the IDE include improved management of recent files (Delphi 2010), the code formatter (Delphi 2010), build configurations (from Delphi 2007 to 2010), the class browser (Delphi 2009) and several extensions in the editor and designers.

¹ By the way, IDE Insight has been extended in Delphi XE with the ability to search in the Object Inspector, so you can easily locate a property of a component active in the designer by typing its name. A nice addition to an already effective tool.

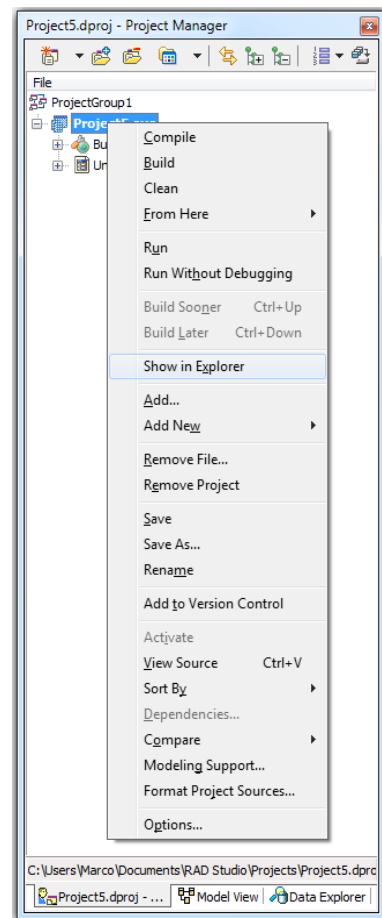
In this chapter I'll discuss a number of new features of the IDE, but not all of them. Some of the main features introduced by Delphi XE, like Subversion integration, modeling extensions, and the third party tools that have been added to the IDE, will be covered in the next chapter. I won't delve into these topics in detail, particularly the third party tools, as they are well documented by their respective vendors.

Managing Projects

The Project Manager is one of the areas that had been traditionally very neglected in Delphi compared to other IDEs, but has been extended significantly over the last few versions, for example with support for multiple build configurations. This frequently used window sees many enhancements in Delphi XE, because it is the central location for working with version control systems and other add-ins. You can see the initial list of commands for a newly created project here on the side of the page.

Other simpler extensions are equally interesting. The new Show in Explorer local menu item lets you open Windows Explorer to display the folder containing the given project or file. This is much better than the File Browser pane (activated by the View | File Browser main menu item), which is not kept in synch with the project folder in any way.

Other new local commands are the pair Run and Run without Debugging, which were previously only part of the main Run menu (where the former is simply called Run). There is also a Compare command (related to



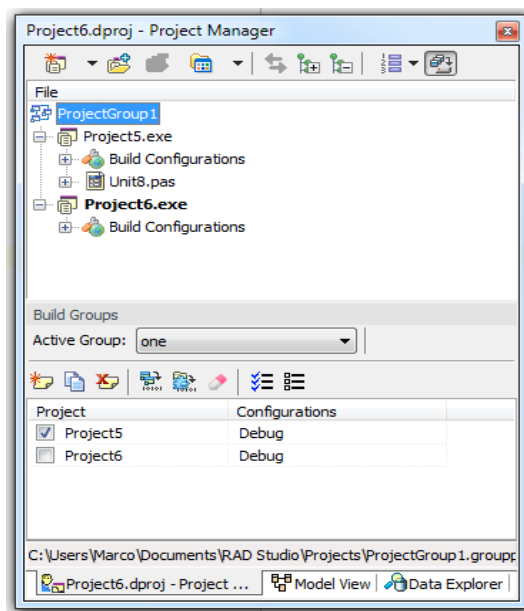
the Beyond Compare integration) and a Format Project Sources command (covered later in this chapter)².

Build Groups



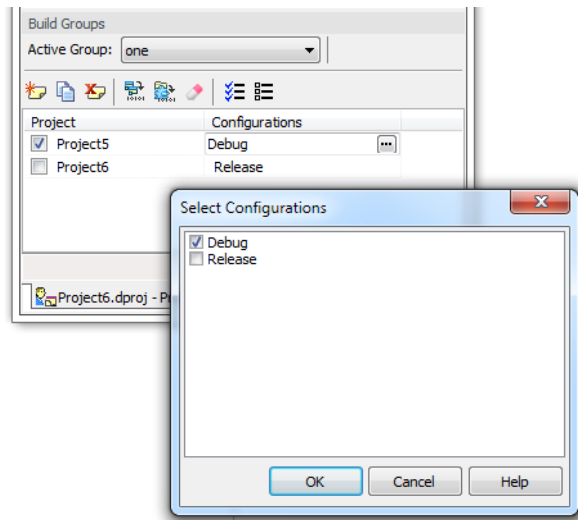
Beside the fact that the Project Manager status bar is on by default, the most significant addition to this window is the Build Groups pane. This pane is hidden by default and you can display it by using the last button of the Project Manager toolbar (shown here on the left).

The Build Groups pane is displayed at the bottom of the Project Manager (as in the figure below) and lets you pick some of the projects or packages of a Project Group and compile or build only those, and not the entire group. You can also define multiple Build Groups within the same Project Group, which gives you a lot of flexibility in terms of managing builds.



-
- 2 I sincerely hope they stop adding items to the Project Manager local menu, as 25 menu items are a lot and finding the command you need is becoming slower. Maybe they can find a different way to make these valuable features available. Still, sub-menus are not always a good idea and the Subversion sub-menu is already three levels deep.

Most of the operations within the Build Groups pane are fairly easy, as you can create, duplicate and delete groups. You can also issue a compile, build, or clean command on the active one. For each project in the group, the pane shows the build configuration (like *Debug* in the previous screen shot): this is not the current build configuration of the project, but rather a specific build configuration within the build group. To change it, you have to double click on the entry and open the Select Configurations dialog box that shows by clicking on the ellipsis button:



In other words, if you want to be able to build some or all of the project in a group in both debug and release configuration (or any other), the simplest way now is to create two different build groups and set the configurations for the various projects. Too bad that there seems to be no way to change the configuration for each project in the build group at once, like you can do in the Configuration Manager for the projects in a Project Group.

The Configuration Manager

Also the Configuration Manager (now View | Configuration Manager, while in the past it was under the Project menu) has been improved somewhat, particularly by cleaning up a rather messy user interface. Beside listing the current build configuration for each project in a group, and letting you change all or many of them at once, the Configuration Manager is particularly helpful if you

need to manage “options sets” (which are a set of project options saved to an external file and possibly referenced by multiple projects, covered in my Delphi 2009 Handbook).

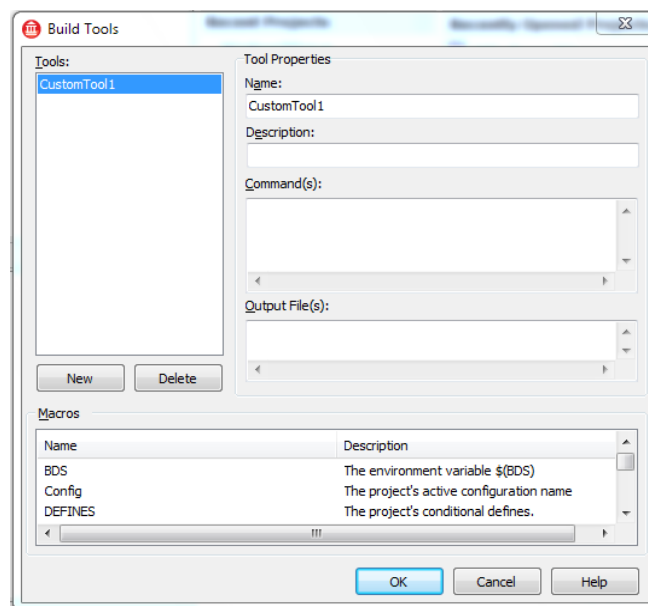
The Configuration Manager now shows the referenced options sets in the list of projects, rather than in a separate pane; it has a new combo box for activating a different configuration, and has a nicer toolbar.

Build Tools Customization

The last of the elements related with the management of projects and build configurations is the new dialog box to define Build Tools. These are command line applications that you can use to process a file stored in a project, and are generally matched by extension.

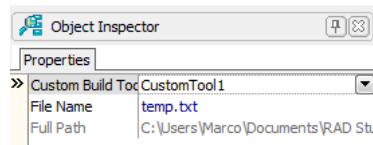
The Build Tools you define in the corresponding dialog box have very vague relationships with the Build Events. You can add to the Project Options to have extra operations done before or after the project build process.

The Build Tools dialog box (displayed with the Tools | Build Tools menu item) let's you configure external tools and give them a name:



18 - Chapter 1: Delphi XE IDE

Once you've defined a Build Tool, based on a command-line program or utility, you can tie it to one of the files of your project (with the exclusion of .pas files), by using the Object Inspector while a file is selected in the Project Manager:



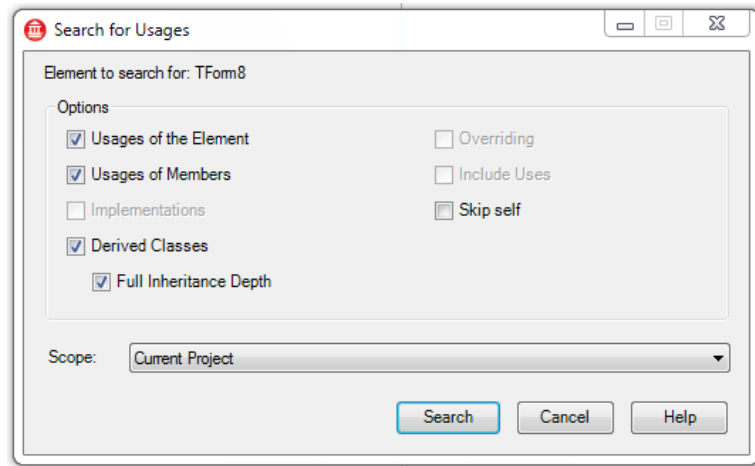
Extensions to Searching

Delphi 2010 saw a significant redesign of the searching capabilities of the IDE, with the use of a bar at the bottom of the editor as a replacement for a dialog for simple searches (activated with Ctrl+F). I know some developers didn't like the change, but I don't want to enter a debate here. Rather, I want to cover the further changes made in Delphi XE to improve searches.

Search For Usages

For several versions, the Delphi IDE has had a "Find references" command, based on the in-memory parsed version of the source code (not the plain source code text). In Delphi XE this feature is still available, but you can improve these searches using the new Search for Usages command of the local menu of the editor. This will open up a dialog box, in which you can fine-tune and narrow your search. The enabled options of this dialog box depend on the kind of symbol you are looking for.

For example, if you search for a type, the dialog lets you include in the search members, derived types, plus variables of the given type. If you search for a method, the dialog lets you include overridden ones. In the dialog box you can filter what to search for and pick full inheritance support by selecting some of the options displayed in the next page:



In the past, this feature was available only to projects for which you had activated Modeling. Now the Search for Usages command is directly available in the Class Explorer pane and in the local menu of the editor.

Local Search Short Cut Keys

There are new short cuts you can use after invoking a local search (Ctrl+F) in the current unit. These new keyboard commands are consistent with the search keystrokes found in moderns browsers like Firefox and Chrome, among other applications. (As I mentioned above, the local search was significantly modified in Delphi 2010 turning the Find dialog box into a pane at the bottom of the editor).

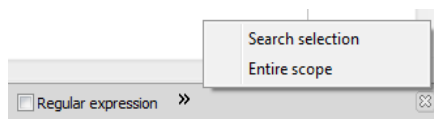
After entering the text you want to search, you can now:

- Press Enter to move to the next entry, like before
- Press Shift+Enter to search backwards
- Press Ctrl+Enter to go back at the beginning of the unit

You cannot repeat the search operation with these keys, though, because these operations place you back in the editor so by pressing Enter again you'll actually add a new line and not search again, which you can achieve by pressing F3 (or Shift-F3 to go backwards)

Speaking of the Find pane (Ctrl+F), there is now a check box for searching the “Entire Scope” and not start from the cursor position. This options doesn't

affect Incremental Searches (Ctrl+E). Notice that unless you have a large screen resolution, this option might show up as an extra entry you can show by clicking on the >> element:



Editor

The editor in itself has seen no significant changes in Delphi XE; but some of the tools you use while editing core (like Live Templates), code formatting, and other related tools have seen updates worth covering.

New Live Templates

Live Templates, available in the RAD Studio IDE since Delphi 2006, offer a fast way to write common code statements and declarations. Since the initial group of templates, they have been expanded over time with new entries, some of which have been suggested by the community. Like many other developers I do have my own extra set of Live Templates and encourage you to build your own³.

In Delphi XE the number of new templates is quite large, with many of them focused on helping you write XMLDoc-based comments, as discussed in the next section. Among these Live Templates there are *summary*, *para*, *remarks*, *param*, and *returns*. As an example, this is the code produced by typing *summary*:

```
/// <summary>
///
/// </summary>
```

Two more Live Templates (*ifnil* and *ifnass*) let you write the standard code used to assign a new object to an uninitialized variable:

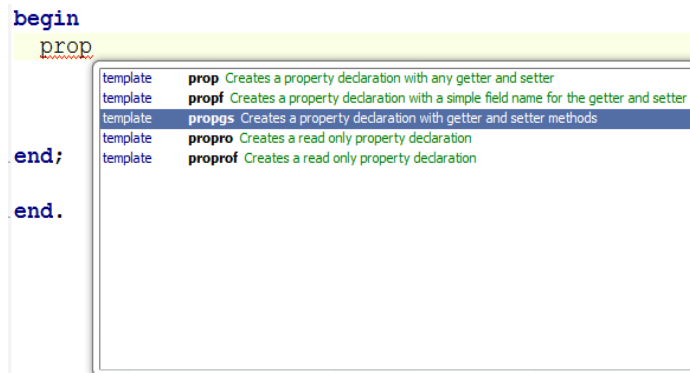
```
// ifnil
if variable = nil then
    variable := classtype.create();
```

3 Information about writing Live Templates can be found in my Delphi 2007 Handbook and at the Delphi wiki: http://delphi.wikia.com/wiki/Delphi_Live_Templates.

```
// ifnass
if not Assigned(variable) then
    variable := classtype.create();
```

Five other new Live Templates can help you out with the declaration of a property, providing field mapping, methods mapping, and let you fill in the setter and the getter (which I've left as such in the following examples):

```
// prop = define property
property Prop1: string read getter write setter;
// propgs = property with getter and setter (gs) methods
property Prop2: string read GetProp2 write SetProp2;
// propf = property mapped to field (f)
property Prop3: string read FProp3 write FProp3;
// propo = read-only (ro) property
property Prop4: string read getter2;
// propof = read-only (ro) property mapped to field (f)
property Prop5: string read fProp5;
```



Another new Live Template, *Surround Windows*, generates an `IFDEF` and can help while marking out code not suitable for future cross-platform versions of Delphi. This is meant only as a Surround template, wrapped around the selected text (in fact, you cannot type it but only select it from a list, because it has a space in its name):

```
{IFDEF MSWINDOWS}
    some code
{ENDIF}
```

Another set of templates helps instrumenting the code for CodeSite logging, as I'll shortly mention in Chapter 2.

Among the existing ones, the *for* template has been slightly modified to automatically refer to *Low* and *High* rather than *o* and *List.Count*. I understand the reason to change it to make it more general, but I think I've got so used to it now that I'll have to change my habit or provide a custom one for going through

a list of elements. As an aside, Code Completion now shows the template description as a further hint.

Finally, the help mentions the *todo* Live Template as a new feature, but that was already in Delphi 2010. Still, it is worth mentioning this template that expands into the following *special* comment:

```
| {TODO -oOwner -cGeneral : ActionItem}
```

XMLDoc Comments and Help Insight

I've already covered Help Insight in my Delphi 2007 Handbook and in a few blog posts after that, but it is worth providing a new summary, in case you've never used this feature.

For many versions, the Delphi compiler has included support for *XMLDoc*, the production of a single XML file with all symbols from each Delphi unit that gets compiled. This feature is enabled by turning on the compiler option Generate XML Documentation. While in the past the compiler placed the resulting XML files in the source code folder, in Delphi XE (due to what I consider to be a bug) the XML files are all placed in the C++ header files folder. By default the folder is *hpp* under the shared (public) documents folder section for RAD Studio 8.

On my computer, this is:

```
| C:\Users\Public\Documents\RAD Studio\8.0\hpp
```

You can customize this folder in the global Options dialog, in the section Delphi Options | Library, by changing the HPP output directory.

Now to get back to our topic, you can decorate these XML files with extra developer information that the compiler can extract from special comments, marked with a triple slash (///). These are the comments generated by some of the new Live Templates mentioned in the previous section, like “*summary*”.

If you create a new project, move before the form class declaration, add a new line and activate the *summary* Live Template, you'll get code like the following:

```
| type
|   /// <summary>
|   /// This is my nice new main form
|   /// </summary>
|   TForm10 = class(TForm)
|   private
|       { Private declarations }
```

Now in the XMLDoc file you'll see something like:

```

<class name="TForm10" file="Unit10.pas" line="13">
  <devnotes>
    <summary>
      This is my nice new main form
    </summary>
  </devnotes>
  <ancestor name="TForm" namespace="Forms">
    ...

```

However, what is more important regarding XMLDoc is the ability to get the same information immediately in the editor, without even compiling or enabling the compiler option, as it automatically shows up in the Help Insight pane for the given symbol:



It is possible to customize the output of this popup by modifying the associated CSS and also the XSL file used to produce the HTML output of the Help Insight from the XML data. More information on these customizations is on my blog:

http://blog.marcocantu.com/blog/help_insight_customization.html
http://blog.marcocantu.com/blog/help_insight_xsl.html

Navigating Modified Code

In the last few versions of the editor, there are change bars on the side of the main window marking out lines that have been modified since you last saved (yellow) and since you started editing the given file (green). In a long file, however, locating these modified lines is far from simple, unless you move to the History tab to view Differences.

That's why Delphi XE introduces special keystrokes for navigating around modified lines. You can move to the next or previous modified line (including both green and yellow ones) of the current unit with `Ctrl+Shift+F8` and `Ctrl+Shift+F7`, respectively; you can move to the next or previous modified and unsaved line (yellow ones only) with `Alt+Shift+F8` and `Alt+Shift+F7`.

Source Code Formatting

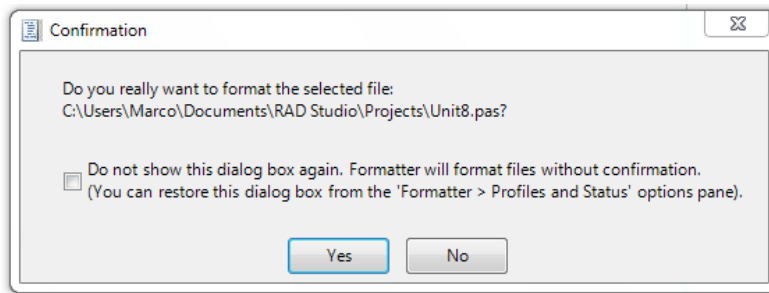
Source code formatting was introduced in Delphi 2010, but it had a couple of weaknesses that limited its usefulness. First, you could format the selected lines of a unit or a full unit, but not all of the units of a project at once. Second, if you customized the format settings there was no obvious way to copy the settings to another computer or share them with the other members of your team, to keep the source code format consistent.

Delphi XE addresses both issues, adding some further formatting options. The first and more obvious change is that you can now use the Project | Format Project Sources menu item or the corresponding entry of the Project Manager local menu (on a project node). As the name implies, this applies the current source format settings to all of the units of the current project, after showing a pretty dull confirmation dialog.

As you reformat a project you get some status information on a dialog which you can barely glance at, but this is all added to a specific tab of the Messages pane, like in the following simple case:

```
Starting format sources in project C:\Users\Marco\Documents\RAD
Studio\Projects\Project5.dproj
  Formatting C:\Users\Marco\Documents\RAD
Studio\Projects\Project5.dpr ...
  Formatting C:\Users\Marco\Documents\RAD Studio\Projects\Unit8.pas
...
Format project sources finished
```

If you format an individual unit, instead, the dialog below will let you disable further confirmations (see below) and won't output its status to the Messages pane.



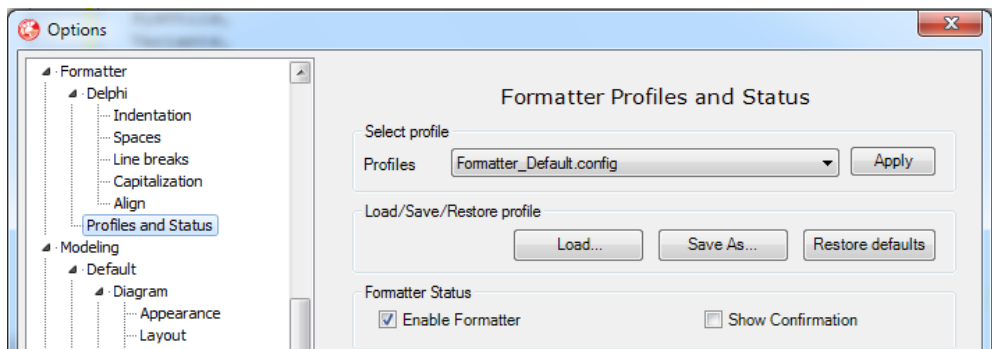
Formatter Profiles

The **Formatter | Profiles and Status** page of the Options dialog box mentioned in the message box above contains the specified settings and also the option to disable the code formatter, removing the corresponding menu items.

Its main goal, though, is to let you manage the active settings of the formatter, picking one of the predefined options, updating it, and saving the current settings to a custom file. If you save the files in the Delphi user's "roaming" folder, like in my case:

■ C:\Users\Marco\AppData\Roaming\Embarcadero\BDS\8.0

and it's named with the pattern "Formatter_*.config" it will be picked by the IDE and listed in combo box at the top of the page, if not you can still locate it with a File Open Dialog. The top of the page is shown next:



Interestingly enough, a formatter profile is a rather plain XML file you might even edit or process directly. At the beginning there are the global settings, followed by the specific settings for each language, grouped by category using some comments:

```
<?xml version="1.0"?>
<TgConfig Version="3" SubLevelDisabled="False">
  <Assemblies>
    <Assembly Name="Embarcadero.Modeling.OpenAPI3"
      Culture="neutral" />
  </Assemblies>
  <Option Name="Formatter.Enabled" Type="System.Boolean"
    Category="Editor" Value="True" Levels="Formatter" />
  <Option Name="Formatter.Confirmation" Type="System.Boolean"
    Category="Editor" Value="True" Levels="Formatter" />
  <!-- DELPHI -->
  <!-- Indentation -->
  <Option Name="DelphiFormatter.TabSize" Type="System.Int32"
```

```
Category="DelphiEditor" Value="2" Levels="Formatter" />
<option Name="DelphiFormatter.IndentSize" Type="System.Int32"
Category="DelphiEditor" Value="2" Levels="Formatter" />
<option Name="DelphiFormatter.UseTabs" Type="System.Boolean"
Category="DelphiEditor" Value="False" Levels="Formatter" />
<!-- SubCategory=General -->
...
```

Other Changes in Formatting

There are a couple of other interesting changes to formatting. The first is the ability to control alignments with the new set of Align options, which include the alignment of initializations, comments, and the like.

Another very nice feature, if you happen to have many Delphi projects, is the ability to run the code formatter from the command line, as a stand-alone tool, without having to open a project in the IDE. Just run `Formatter.exe` in Delphi's `bin` folder passing the project to format, format options, directories to format recursively, and so on.

This is particularly suited if you use an external building tool (like FinalBuilder⁴), which can reformat the source code of a project before committing changes to a version control system⁵. I guess you can easily find other situations in which this can prove useful.

Form Designer

There is no significant overhaul in the form designer and other Delphi visual designers, but a change certainly worth noticing, is the ability to copy the graphical representation of a form (more about this in a second).

4 FinalBuilder will be introduced in Chapter 2.

5 Beware of automatically formatting code for projects under version control. A change in the format is considered a change, and may cause useless conflicts with real changes written by other developers. On the other hand, applying a consistent and shared format setting before committing changes can help avoid the very same problem of future changes done for the sole purpose of using a consistent formatting style.

Other minor additions include the new Insert Separator command in the Menu Designer and the tool tip with coordinates in the Form Positioner, the small bottom-right image of the position of the form within the computer screen.

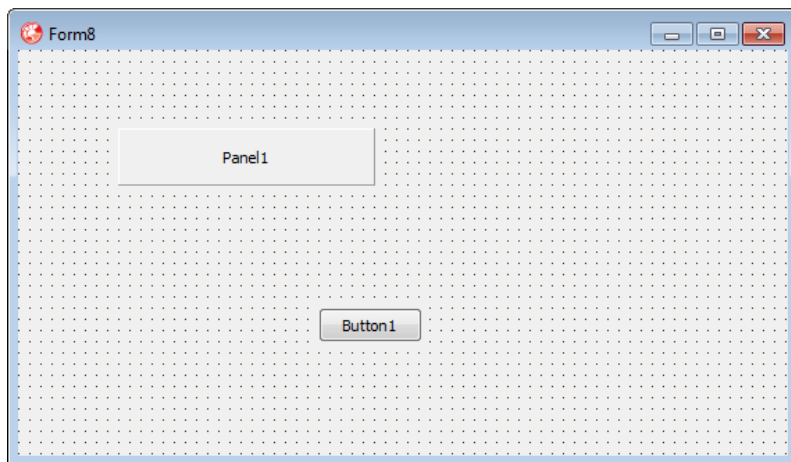
As I just mentioned, the nicest new feature is the ability to select a form at design time (not form's component, but the form itself), press Ctrl+C or issue an Edit | Copy command, and get in the clipboard both the textual description of the form, like in the past, and the image of the GUI design.

So you can paste the clipboard content both to a textual editor and a graphical program, or use the Paste Special feature of a word processor to get either one of the other. Needless to say this is a real bonus for a book author, but anyone providing code documentation can certainly benefit from the feature.

As an example, I've create a new form, added a few components, copied the form, pasted the text below as usual... and then pasted the graphical representation in the image shown after the code:

```
object Form8: TForm8
  Caption = 'Form8'
  ClientHeight = 290
  ClientWidth = 554
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  PixelsPerInch = 96
  TextHeight = 13
  object Panel1: TPanel
    Left = 72
    Top = 56
    Width = 185
    Height = 41
    Caption = 'Panel1'
    TabOrder = 0
  end
  object Button1: TButton
    Left = 216
    Top = 184
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 1
  end
end
```

Pasting the bitmap from the clipboard (using the Paste Special feature of my word processor), I also got the image in the next page:



Notice this also works for data modules at design-time, in which case you'll get the place-holder for the non visual components in the designer.

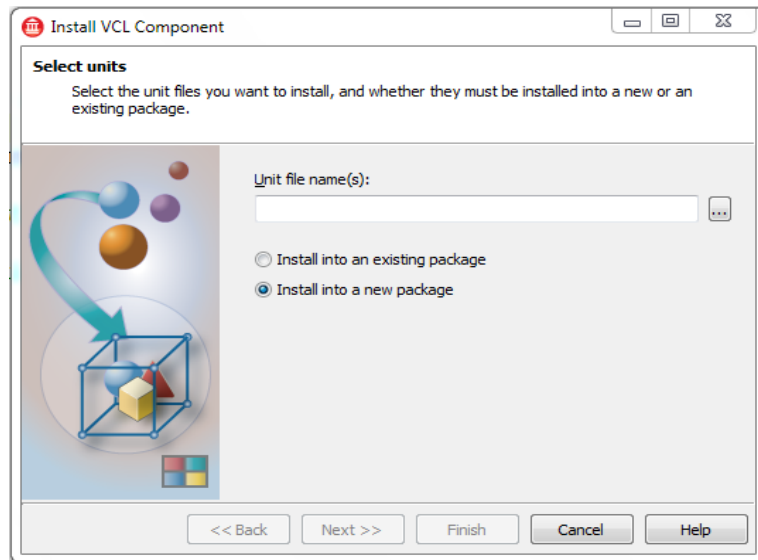
Updated Wizards

An area of the IDE that has received a significant cleanup is that of the Wizards that you can use to get started with development of new applications or to add specific features to existing projects. The most significant changes are in the new Wizards for DataSnap multi-tier applications. A wizard that's not part of the New Items dialog box is the Install Component Wizard, described below.

Install Component Wizard

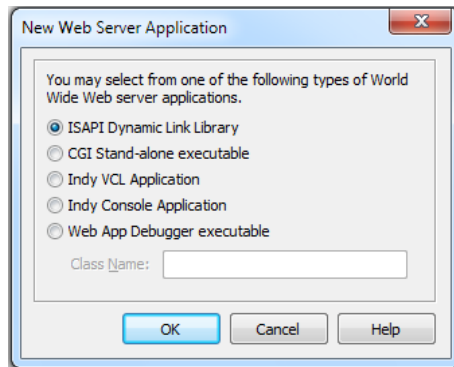
Delphi XE sees the return of an Install Component wizard that was previously available until Delphi 7, and was later removed. Now the wizard (which originates from a contribution by Rudy Velthius⁶) is part of the product again, as you can see in the next page:

⁶ You can see the original description of the Wizard written by Rudy for an earlier version of Delphi at <http://rvelthuis.de/programs/compinstall.html>.



Web Server Application

The Web Server Application option of the Repository has been extended with new options. This is partially a change in the dialog box (see below) but also a change in the underlying library:



Delphi now discourages the use of the Web App Debugger option, and encourages using the standalone Indy HTTP Server integration for Web Broker⁷.

⁷ This integration and some other new related components are covered in Chapter 4.

The Debugger

In Delphi 2010 the debugger was significantly extended to better support multi-threaded applications. Delphi XE sees the final touches with cleaner support for named threads, plus a set of other minor new features.

Named Threads

Named threads have officially been supported since Delphi 2010, but in Delphi XE many of the debugger panes now fully support them, including the Watch List window, the Local Variables window, the Call Stack window, the Debug Inspector, the Evaluate/Modify dialog, and the CPU and FPU views.

In the Thread Status pane there is now a local menu to temporarily name a thread at run time. In general, though, it is more powerful to name a thread in code⁸, making the thread name persistent between debugging sessions.

Run Without Debugging

The Run Without Debugging feature has now been promoted to a top feature, with its own graphic and button on the toolbar. Notice it is quite easy to mix up the Run and Run Without Debugging icons (below on the right), as the latter is very similar to how the former looked in past versions of Delphi (on the left):



On a related topic, you can now specify more precisely what happens when you use the Load Process dialog to start a process. Rather than simply asking the debugger to execute the startup code or not, you can pick multiple options like do not run, run to the first source, or run⁹.

8 Named threads were covered in my Delphi 2010 Handbook. In short, you can associate a debug-time name to a thread by calling the `NameThreadForDebugging` class method of the `TThread` class.

9 Another option, Run to the program entry point, makes sense only for C++ applications. There was more information in an old Embarcadero blog post by Chris Hesik, dated May 24th 2007, but this is no longer available.

More Debugger Features

Here are more new assorted debugger features:

- Delphi is distributed with debug version of the DCP (the symbol information files for packages), which helps debug applications compiled with run time packages.
- There is an option to generate TDS debug information files, debug symbols used by C++Builder.
- Beside closing editor files opened while debugging, you can use the Debugger Options page of the Options dialog box of the IDE to automatically close other debug windows.
- In the same page of the Options dialog box, you can disable the prompt to rebuild modified projects while debugging (“*Source has been modified. Rebuild?*”). It can be quite annoying at times.

Chapter 2:

Integrated Tooling

Beside the core features of the development environment (IDE), Delphi XE sees the integration of several other native and third-party tools. From Subversion support right inside the Project Manager, to extended Modeling features, from Final Builder to AQTime, from CodeSite to the Beyond Compare engine, there are many new features that help you in the development process, although they are not all meant for “writing code”.

A few years back we would have called them ALM tools, but given this word has peaked it's marketing hype and that Delphi developers got quite upset of “old Borland” trying to push ALM their way, willing or not, I'll try to refrain from using it!

This chapter hasn't got a specific focus and it really doesn't get in depth on third party tools, that are very well documented on their respective web sites. It will introduce Subversion, help you figure out which tools are now available in Delphi, what is their role, and why you might want to use them. Since there are some free tools as part of Delphi XE, it wouldn't be wise to ignore them.

Version Control Integration

One of the most significant new features of the Delphi XE IDE is the integration of version control systems. This takes place through a new module called “Version Insight”, which adds an entry to the File menu and plugs right into the Project Manager and its local menu. Before we delve into the version control integration that ships with Delphi XE and we look into one of its extensions, let me start with a short and fast-paced introduction to version control systems in general and Subversion in particular.

Version Control Systems

It is quite common knowledge that when multiple developers¹⁰ want to work together on any given project, it is good practice to have version control system (VCS) or Source Code Control System (SCCS). The key idea is to let multiple developers share some source code files and any changes that any of the developers make on the source code files. Although this could be done by accessing a physically shared repository, the general approach is for each developer to be able to create a personal copy of the various files and work on them. How this is accomplished without interfering with other developers, depends on the overall approach of the VCS.

Some VCS use a ***pessimistic approach*** based on locking. A developer needing to edit a file asks for exclusive ownership of the file, makes some changes to his local copy, send the new file back to the repository, and relinquishes control over the file. The advantage of this approach is that conflicting changes are impossible and developers must synchronize their work. Some big disadvantages include the need of interacting with the VCS frequently (to avoid keeping a file locked for more time that's needed), the need to specifically ask for write permission as you start editing a file (and in Delphi maybe also the need to close a read-only file, only to reopen it in the editor with write permissions after

¹⁰ This doesn't mean that a single developer won't benefit from a version control system. Quite the contrary, having a VCS helps a single developer build a fully versioned backup of your code, and also work from multiple computers or virtual machines on the same source.

this has been granted), and the impossibility of working on the files when not connected to the network where the VCS resides.

There are many pessimistic version control systems, as this was originally the most popular approach, and people with a lot of experience in them don't see how any other approach could work. These VCS might actually work fine for small in-house teams, but tend to fail when the team size and the distance among team members grow.

That's why other VCS use an ***optimistic approach*** based on the idea that every developer has a full copy of the code and full rights to edit any file. Only, team members might frequently post their changes and retrieve changes made by other developers. If two different developers edit a file, as long as the changes affect different lines in the source code, merging can take place automatically. In the rare event of changes to the same line of code, you'll need to manually merge the changes.

Optimistic version control systems can work well from single developers (acting more like a versioned backup) to large teams and have been used by many large open source projects. Unless the team is completely disorganized, different developers tend to focus on different areas of the projects and its source code files, minimizing the chances of a real conflict. Some of the advantages of these VCSs is that developer can indeed work remotely, and there might be many people with read-only rights, something common for large projects in which only a select few have the right to make changes to the system.

Of the most popular VCS systems, the open source CVS and Subversion follow the optimistic metaphor, while Microsoft's Visual Source Safe is more oriented towards the pessimistic model. Truly, most VCS can be configured for both concurrency models, although they'll fall more naturally in one or the other camp.

In recent years, there has been a very strong adoption for yet another model, called *distributed*. In most of the classic VCS, whatever the concurrency model, have a client/server architecture, with a central repository to which all clients connect. In a distributed VCS, instead, there is no central server. Some of them have peer repositories keeping in synch to each other, while in other configurations there is a hierarchical structure with a super master and a tree of distributed children. In general, all nodes have equal power, so you can synchronize with any of the peer nodes or (in case of a hierarchy) with one of the children, creating a grand children node. In both scenarios, you can have a few

nodes working on a temporary separate version from the other peers or the super master node.

Distributed version control systems, in fact, perform synchronization by exchanging patches (change-sets) among nodes. This means that at any time there are multiple working copies of the software, but no official code base. Proponents cite as advantages the speed of peer-to-peer updates, compared to centralized ones, and the fact that everyone that is part of the project can propose changes and based on the relevance of its node, these changes will be accepted by other or further revised.

Among distributed VCSs, the rage at the moment seems to be on git and Mercurial. The latter seems most popular for Windows projects, whilst git is used more in the Linux world. I personally have limited experience with distributed VCSs, but I see a lot of reasons to adopt them. As I'll explain at the end of the session, there is an extension to the current Delphi version control IDE integration that opens up the support for distributed VCSs.

Now that I have covered the core features of a VCS, let's focus primarily on one of them, as it is the only one officially supported in Delphi XE, Subversion.

Subversion

Subversion is a VCS originally meant to extend the (then) ubiquitous CVS, remaining very familiar to CVS users. If that was the original goal, now the goal is to be a best-of-the-breed, at least within its class of optimistic client/server version control systems.

Subversion was originally developed by CollabNet (www.collab.net), but it is now managed by the Apache Foundation. The formal project name is “Apache Subversion” and it is available at:

■ <http://subversion.apache.org/>

Subversion is a centralized VCS and works through a server (how to have one in minutes is explained later). Subversion uses optimistic locking, and although it is possible to place a lock on a file, you are generally discouraged to do so.

A specific feature of Subversion is its use of absolute version numbers. CVS and some of the pessimistic VCSs use a version number for each file, so that a file at version 12 has had 11 updates after the initial version. This is quite intuitive, but causes the problem that if you release the software on a given date, you

need to save the version numbers of each file in what is called a “revision”. Otherwise you won't be able to recreate the given situation.

Subversion, instead, uses a project-wide revision number. Each file is at a given revision, but revisions get incremented for the entire system. So after 11 revisions, a file could be at version 823, in cases when other files have been updated many times. The given file will have version numbers in sequence but not in a continuous sequence. As an example, if you have three files (A, B, and C) and update them in a round robin fashion you'll have:

```
A at 1, B at 1, C at 1 // initial status
A at 2, B at 1, C at 1 // commit A
A at 2, B at 3, C at 1 // commit B
A at 2, B at 3, C at 4 // commit C
A at 5, B at 3, C at 4 // commit A
```

Now if you built the software using the latest version of each file you'll build revision 5. If you go back to revision 4, you take each file at version 4 or the last version before 4, that is the second last line above.

I know this might sound confusing at first, but it has the distinctive advantage that you don't need to create a “revision” or anything like that, as project-wide versions are automatically maintained due to the inherent way versions are created.

Subversion Operations

Which are the most common operations you perform on the client for a Subversion repository? Here is a short list:

- **Checkout** is the operation of getting a version of the project code (either the latest or HEAD revision, or a given intermediate revision) creating a local copy.
- **Update** is the operation of getting the current version of a given file from the central repository (or all the files in the project).
- **Commit** is the operation of sending local changes to the server, generally adding a comment that specifies the reason for the change (or group of changes). Committing a file already changed by another developer is not allowed, as you need to Update the file (fetch the other developer changes) and do a local merge first if needed. At that point, you'll be able to commit your changes incorporating (or removing) changes by the other developer.

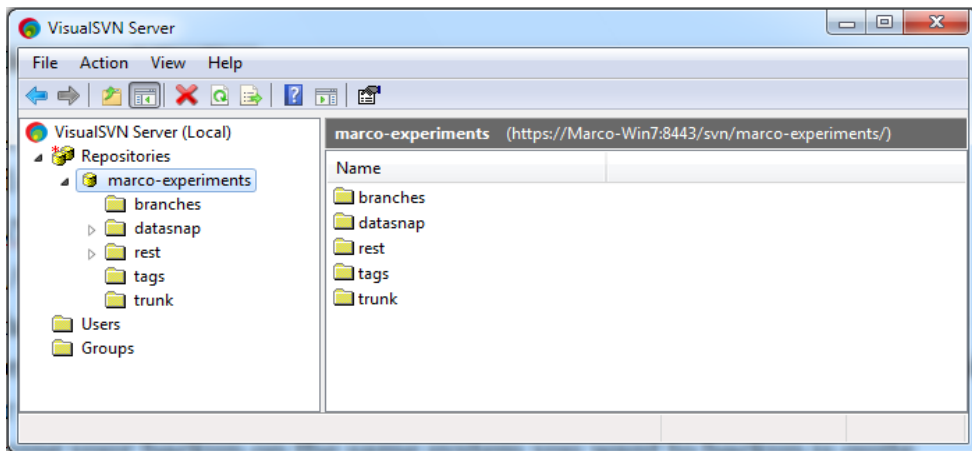
It is recommended to make frequent commit and update operations, and to update files before making changes, but there are specific situations in which

you'd rather follow a different path, like developing new features while ignoring what the rest of the team is doing.

Subversion Servers

Since Subversion is a centralized client/server VCS, before you start you need to have a server. While most Linux distributions include Subversion in their standard installation, configuring a Subversion server and integrating it with Apache (to allow HTTP access) is often far from trivial.

A very good starting point if you want to install your custom subversion server, at least on Windows, is to use VisualSVNServer¹¹, a very easy to use application packaging both Subversion and a customized version of Apache and offering one-click installation and a very simple visual user interface for creating projects and configuring users:



Notice that while installing VisualSVNServer on your development computer will be very fast, even if you are working alone this is generally far from a good idea. For single users (but also for small teams) a VCS acts also as a backup, and running your backup on the same system that you want to backup is quite dumb. Also, if you need multiple developers to reach your server remotely, you should consider hosting your Subversion repository on a server they can access. For open source projects, most repositories (like SourceForge and Google Code) offer Subversion integration and you'll be up and running in minutes.

¹¹ VisualSVNServer is hosted at <http://www.visualsvn.com> and has a basic free version.

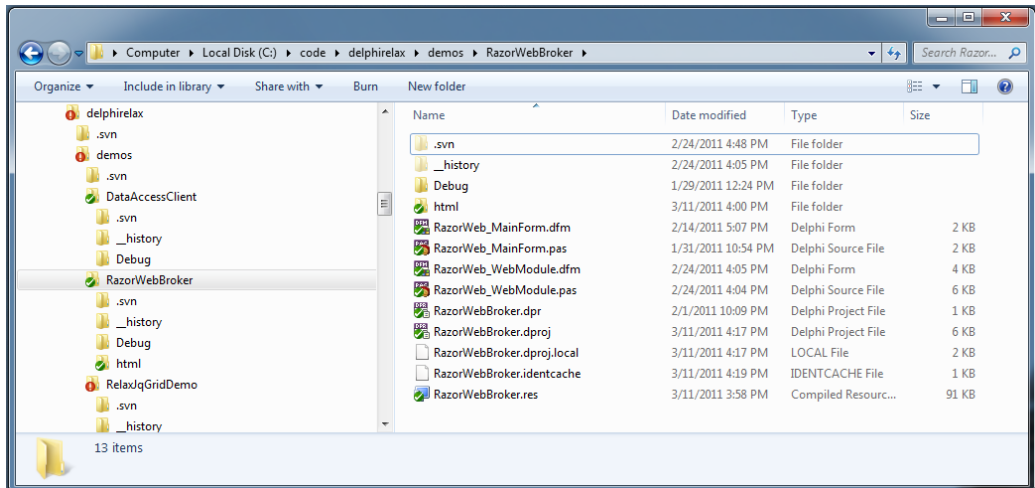
There are also many similar hosting services, allowing both public and private projects for very limited fees. For example, I'm using the Indefero code hosting service (www.indefero.net) to manage multiple projects (some of which are open to anyone, while others are only for specified users) on:

■ <http://code.marcocantu.com>

Subversion Clients

Once you have a server up and running, you need a client to interact with it. You can install the default CollabNet client¹² (which is installed as part of the Delphi XE setup) and use the command line interface to interact with subversion. Most users, however, prefer using a more visual and intuitive approach.

One of the most popular Subversion clients for Windows is TortoiseSVN¹³, which integrates with Windows Explorer letting you see the status of a projects and its individual files by looking at some over-imposed icons (like the green check and the red exclamation mark in the image below) and letting you work on files and folders with specific local menus.



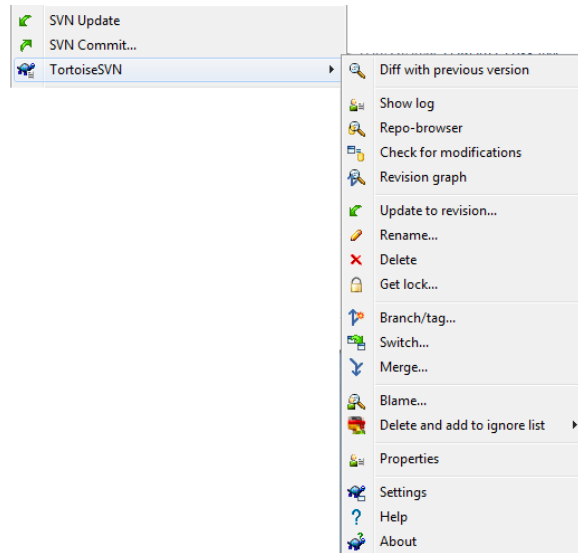
The TortoiseSVN client adds to any folder under Subversion (that is, with a .svn sub-folder) a very large set of menu items to perform most of the available operations, starting with the basic ones (Update, Commit) and including

¹² More information at <http://www.collab.net/products/subversion>

¹³ Available at <http://tortoisesvn.tigris.org>

40 - Chapter 2: Integrated Tooling

branching support, reverting to previous versions, manual merge management, blaming, locking, and more. Here you can see an example of the items and sub-items added to the Explorer local menu by TortoiseSVN when a file under version control is selected:

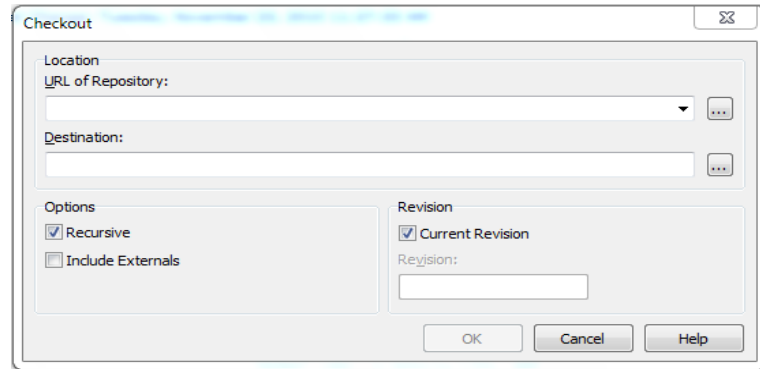


In other folders, instead, the TortoiseSVN client adds a Checkout command, the link to the repository browser, the ability to export a project, and a few other commands.

Notice that all of these clients (including the Delphi IDE) create `.svn` folders under the actual folders to store version and status information of the various files. Since all Subversion clients use the same local data structure, they can be used interchangeably. You can alternate using TortoiseSVN and the Delphi IDE on the same project, and at times you'll need to do so since the Delphi IDE offers only a limited subset of a complete Subversion client.

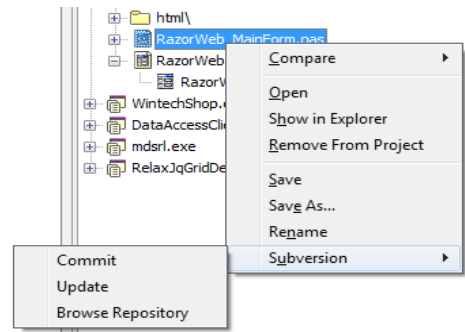
Subversion in Delphi

Without giving up using specific Subversion clients, you can certainly take advantage of the Delphi XE IDE integration¹⁴. First, you can now open a project directly from a Subversion repository, with the menu command File | Open From Version Control. This shows the following dialog:



As you can see, you can pick the URL of a repository (picking a previously used one in the combo box or activating a repository browser with the ellipsis button), pick a local destination folder, and select a few options. In particular you can ask for the current revision (*HEAD*) or enter a revision number. At the end of the operation, the versions control integration will ask you to pick one of the projects or project groups just downloaded from the repository and open it in the IDE.

Regardless of whether you open a project directly from a Subversion repository or if you open a project that has Subversion support, the Project Manager has a new Subversion item with a set of sub-items (here on the right) that lets you perform the core Subversion

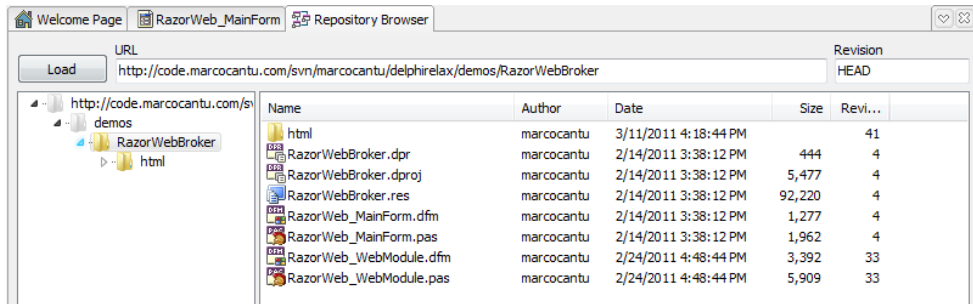


¹⁴ Delphi XE Subversion integration originated from the DelphiSVN project originally written by Ondrej Kelle (also known as TONdrej) and later maintained by Uwe Schuster, who did a lot of the work on the Delphi XE version. Information about Uwe and this project is available on his blog at www.bitcommander.de/blog.

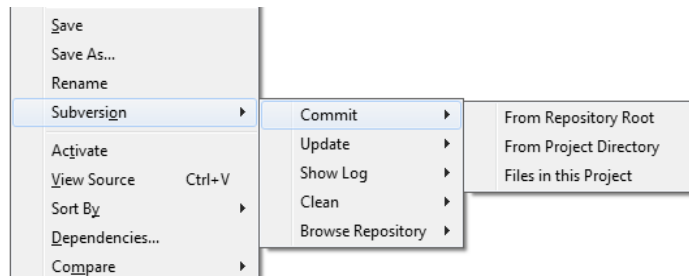
42 - Chapter 2: Integrated Tooling

operations. The various operations can be done on a single file or on a project-wide level. On a single file you can do the operations visible in the previous image.

Notice that as you Commit a file it will be automatically saved, to avoid committing an earlier version and missing changes in the editor buffer. The Browse Repository command opens a new pane in the editor, which lets you fully examine the remote repository and its files (but doesn't provide any further commands):



The Project Manager commands available at the project-wide level for projects under version control can be invoked on the project files, the project folder, or the entire repository:



The project-level Subversion commands are:

- **Commit** the entire repository root, the project directory, or the files in the project
- **Update** the entire repository root, the project directory, or the files in the project
- **Show Log** for the repository root or for the project directory
- **Clean** the repository root or the project directory

- **Browse Repository** starting at the repository root or project directory level

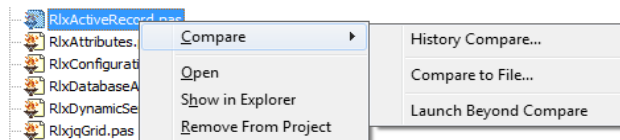
If a project is not under version control, the Project Manager will have a different menu item, Add to Version Control. This is similar to the Import command in TortoiseSVN, but it is smarter since it can work on project source files rather than all files in the folder and will automatically skip local configuration files and by-products of the compilation¹⁵. Also, a specific Import page in the editor pane will let you deselect some of the project files before adding them to the version control.

Beyond Compare Integration

One of the most common operations when you have multiple versions of the same file is the ability to compare two revisions line by line, and eventually merge them. That's why Delphi ships with a limited version of Beyond Compare¹⁶, called Beyond Compare Lite, limited to comparing individual files and not the complete set of files in folders and sub-folders.

You can access some of the features of Beyond Compare from the Edit | Compare menu items and its sub-items, but I generally find it easier to access the file comparison tools from the Project Manager. This can be Beyond Compare Lite or a different file comparison tool you can configure in the Delphi IDE, by using the Difference Viewer and Merge Viewer pages of the Environment Options dialog box.

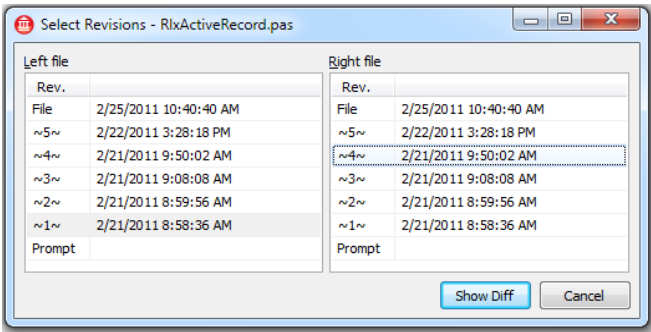
Beside having its individual item in the main menu, this file comparison tool hooks into the Project Manager, providing a few sub-items in the Compare menu:



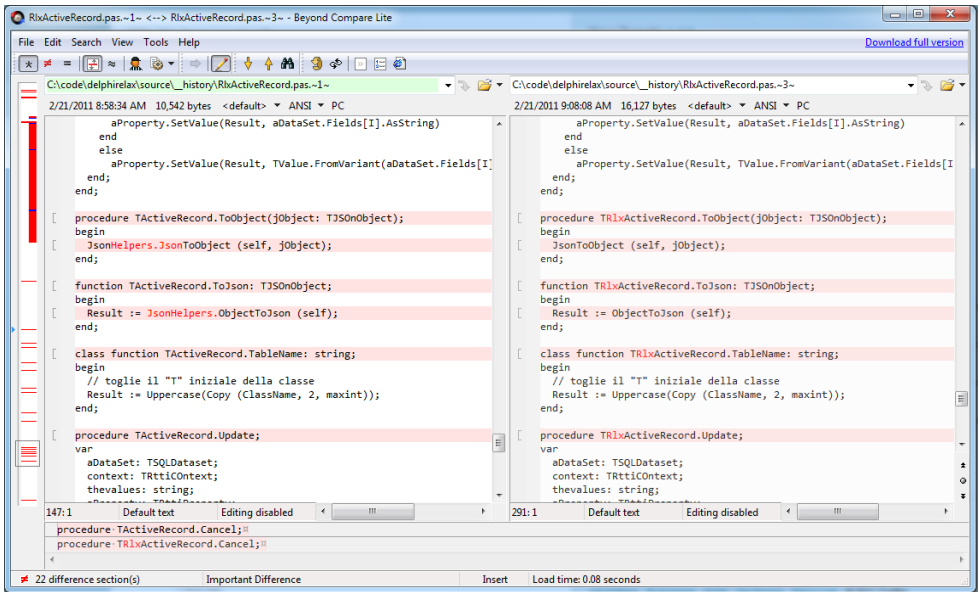
-
- 15 You can also define rules for ignoring files by extension at the command line or with TortoiseSVN, although you'd have to do it manually the first time.
- 16 Beyond Compare is a great files and folders comparison utility written in Delphi and well worth its cost (I've owned a team license for many years now). Complete information at www.scootersoftware.com.

44 - Chapter 2: Integrated Tooling

From this menu you can (in bottom to top order) start Beyond Compare, compare the current file to any file in the file system, or compare the file to a previous revision. The following dialog will let you pick any two versions of the given file (not necessarily the current one):



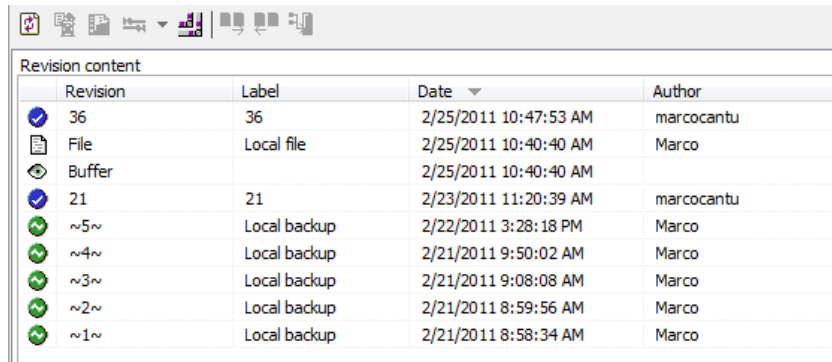
As you make this selection, you'll see the two files open in the differencing engine (by default Beyond Compare), as shown below:



The Updated History View

You can open the external differences viewer from the History pane of the editor, which has been enhanced in several ways. As it did originally with

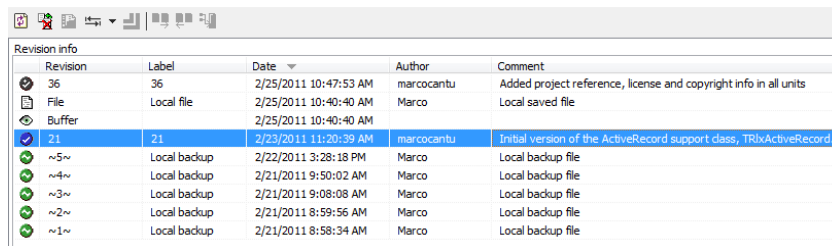
StarTeam integration, you can see local files history merged with the version control information, as in the next page image:



Revision	Label	Date	Author
36	36	2/25/2011 10:47:53 AM	marcocantu
File	Local file	2/25/2011 10:40:40 AM	Marco
Buffer		2/25/2011 10:40:40 AM	
21	21	2/23/2011 11:20:39 AM	marcocantu
~5~	Local backup	2/22/2011 3:28:18 PM	Marco
~4~	Local backup	2/21/2011 9:50:02 AM	Marco
~3~	Local backup	2/21/2011 9:08:08 AM	Marco
~2~	Local backup	2/21/2011 8:59:56 AM	Marco
~1~	Local backup	2/21/2011 8:58:34 AM	Marco

You can see that local backups are intermixed with versions in the VCS, indicated by the revision number. Also the author name is either the local computer user (in this case, Marco) or the Subversion user (in this case, marcocantu).

If you move from the Content page to the Information page (using the tabs at the bottom) you'll also be able to see the comments from the repository:



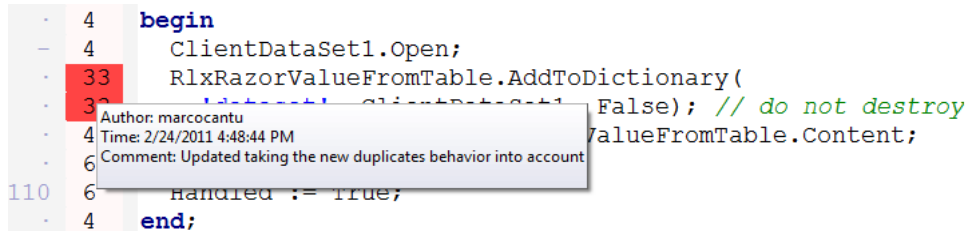
Revision	Label	Date	Author	Comment
36	36	2/25/2011 10:47:53 AM	marcocantu	Added project reference, license and copyright info in all units
File	Local file	2/25/2011 10:40:40 AM	Marco	Local saved file
Buffer		2/25/2011 10:40:40 AM		
21	21	2/23/2011 11:20:39 AM	marcocantu	Initial version of the ActiveRecord support class, TRlxActiveRecord.
~5~	Local backup	2/22/2011 3:28:18 PM	Marco	Local backup file
~4~	Local backup	2/21/2011 9:50:02 AM	Marco	Local backup file
~3~	Local backup	2/21/2011 9:08:08 AM	Marco	Local backup file
~2~	Local backup	2/21/2011 8:59:56 AM	Marco	Local backup file
~1~	Local backup	2/21/2011 8:58:34 AM	Marco	Local backup file

The fourth button in the toolbar of this page (the one with the facing arrows) lets you open the Difference viewer, picking one of those available. Needless to say that the same operation is available as you move to the Differences tab, although in that page you'll see the built-in differences viewer, which I don't find very usable.

Another extension to the history view is the ability to show revision information and annotations (or *blames*) right in the editor. If you go back to the Contents page and pick a revision from the version control system, you can select the Annotate button (the third of the toolbar). After some processing, you'll see for each line the version it was introduced or last modified, with highlighting for

46 - Chapter 2: Integrated Tooling

the most recent versions and the author and command in a hint as you move the mouse over it:



Version Insight and Its Extensions

As mentioned earlier, Delphi XE Subversion integration is part of a larger project called Version Insight. The first release of this project shipped with Delphi XE, but Uwe Schuster kept working on the project, officially called “RAD Studio Version Insight Community Version”. It's available on SourceForge at:

■ <http://radstudioverins.sourceforge.net>

You can disable the current IDE packages, download the project (from its Subversion repository), and compile and install it in the IDE. You'll enjoy a number of fixes and optimizations, and also brand new features including:

- Support for git and Mercurial integration, along side Subversion. This triples the number of VCSs supported, and opens the system to distributed version control systems.
- Live blame in the editor (that is the ability to see who wrote each line and in which revision it was modified right into the Delphi editor, not the history view).

For more and up-to-date information about the project download its source code from the URL above and follow Uwe's blog at:

■ <http://www.bitcommander.de/blog>

I am hopeful that these extensions will be “officially” integrated in a future version of the Delphi IDE.

UML Modeling

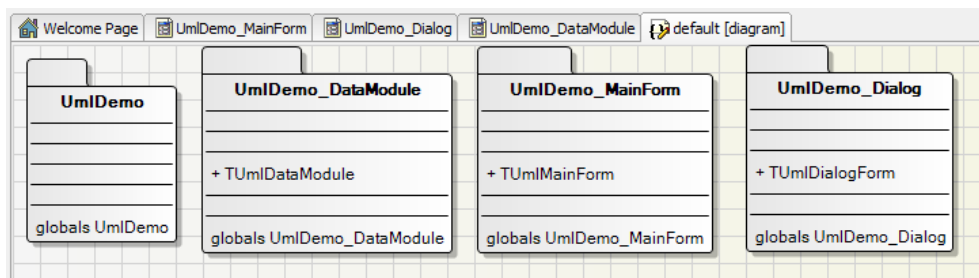
The Delphi IDE has had integrated support for UML modeling for quite a few years now. Originally based on the Together technology, which remained with Borland and was later acquired by MicroFocus, the Delphi integration was built by the Delphi R&D team and remained part of CodeGear and later Embarcadero, but without the “Together” name.

Of the large number of UML diagrams available in the product, the most relevant ones are those with a direct mapping to the source code: package diagrams, class diagrams, and (starting in XE) sequence diagrams. The UML modeling support has a few other relevant features I plan mentioning very briefly.

Classic Live Diagrams: Packages and Classes

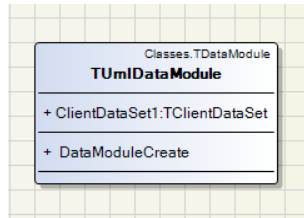
Among the UML diagrams you can obtain directly from the source code of a program, there are package diagrams and class diagrams. These have been around for several versions of Delphi. Package diagrams have nothing to do with Delphi packages, but should rather be called “units” diagrams, since the terms package here means the physical source code containers making up the application.

If you open an application (I've created a simple VCL program with a secondary unit and a data module, that is three units in total), you'll see a package diagram like the following:



This diagram shows the various units of the project, listing the classes and global variables defined by each of them. By selecting a symbol, you'll jump to

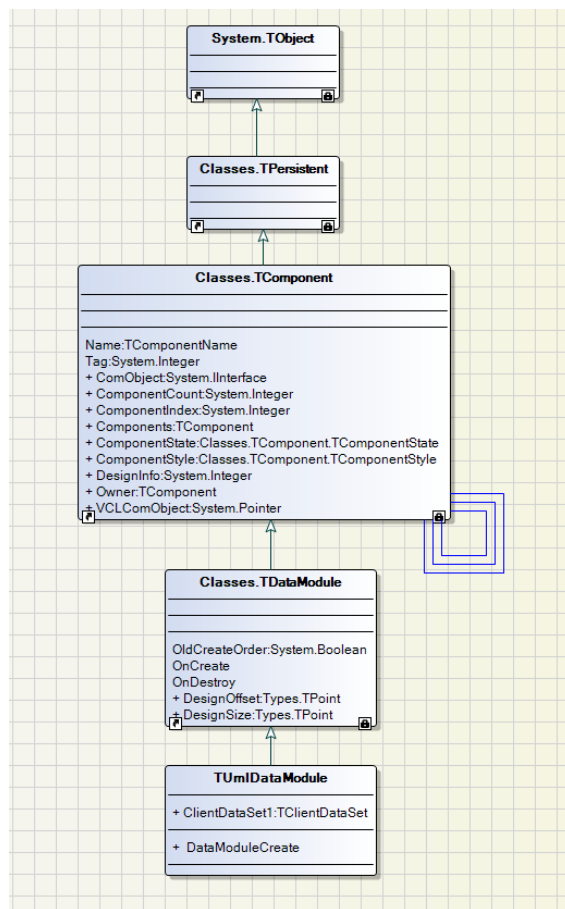
the source code, but you can also select the unit/ package and open its class diagram, like in the following very simple case:



Ancestor Classes in Class Diagrams

The features briefly introduced in the previous section were already in past versions of Delphi, what's new in XE is the ability to easily add ancestor classes defined in different units to a class diagram. This also includes the case of base classes that are part of the VCL.

So, for example, you can select the representation of the **TUmIDataModule** class above and double click on the base class (**Classes.TDataModule**), you'll see new menu items for adding to the diagram either the base class or all ancestor classes, which I've done. You'll get a large diagram, but after collapsing some of the details it will more or less look like in the image on the side.



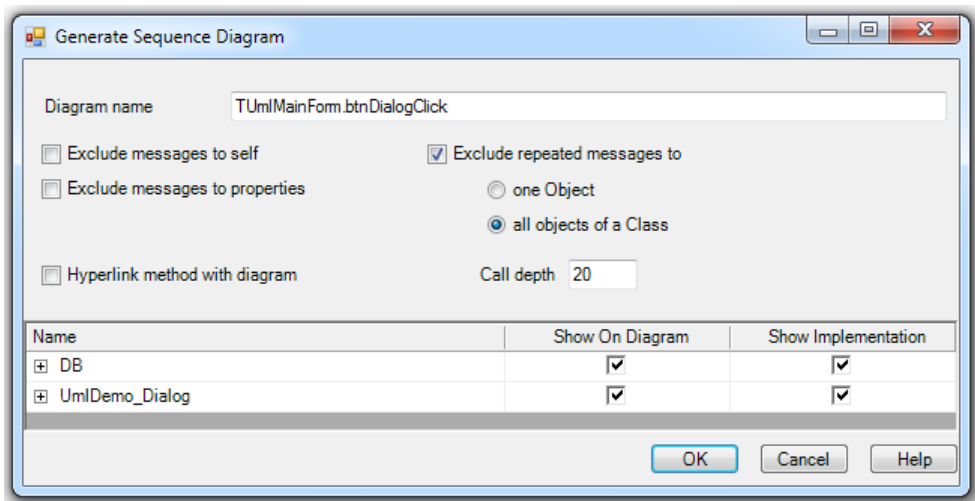
(Notice that all these diagrams are part of the UmlDemo example so you can easily see them in action in the IDE without having to follow these steps.)

Sequence Diagrams

Another brand new feature of the UML modeling support in Delphi XE is the ability to create sequence diagrams. These diagrams, which show the step-by-step execution of a method in terms of calling methods of other objects, are very tedious to create manually, making the ability to generate them in a few seconds extremely relevant.

Class diagrams can be designed before starting to code, but are also quite useful to document existing code and keeping the documentation up to date, which is why the “two-way” approach of the Delphi modeling Class Diagram support is quite nice. Similarly, the purpose of sequence diagrams is certainly to help you design a system and its iterations before coding it, but it is also quite useful to extract the code structure from the actual code you've built.

As an example, I've added to the demo some initialization code for the dialog, create by a button in the main form. By selecting a method in the model view pane, you can use the Generate Sequence Diagram entry of the local menu for this purpose. In this case, I did this operation for the `btnDialogClick` method of the `TUmlMainForm`. As you invoke this menu, the modeling support will show a configuration dialog box, which allows you to pick some relevant properties like the method call depth, exclusion rules, and more:

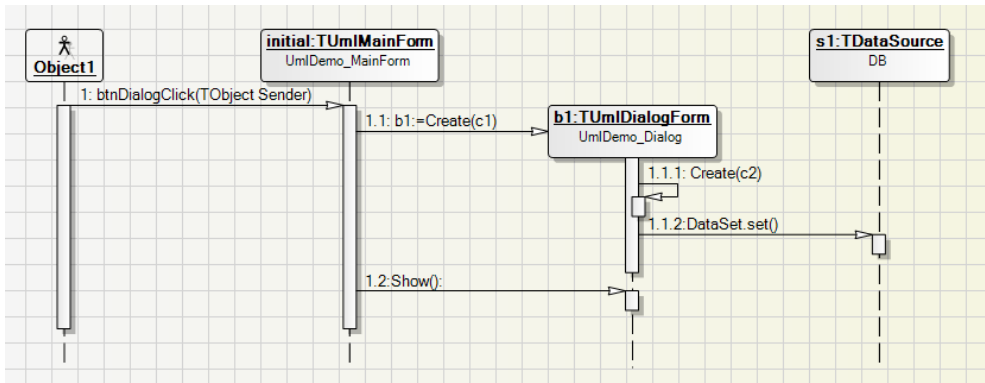


Here is a traditional listing of the code being executed as a user clicks the only button on the main form:

```
procedure TUm1MainForm.btnDialogClick(Sender: TObject);
begin
    Um1DialogForm := TUm1DialogForm.Create(self);
    Um1DialogForm.Show;
end;

constructor TUm1DialogForm.Create(AOwner: TComponent);
begin
    inherited;
    DataSource1.DataSet := Um1DataModule.ClientDataSet1;
end;
```

Given this code, here is its visualization in terms of a sequence diagram:



You can edit and customize this diagram at will, but it won't remain connected to the original code. If you change the code, you'll have to generate the diagram again. Even with this limitation, having the ability to generate such high level visual documentation of your code in a matter of minutes, is certainly a significant improvement for Delphi's native modeling support.

Audits, Metrics, and Documentation from the Command Line

Among other features, modeling support includes the ability to generate three different types of reports: source code audits, metric analysis, and documentation including source code symbols and all of the diagrams you have generated. This hasn't changed significantly since previous versions of Delphi.

What is new is the ability to run these operations from the command line. The advantage here is not in the ability to manually invoke these commands, but the fact you can run them from external build configuration tools, like Final-Builder (which ships with the Enterprise version of Delphi XE).

The new modeling command line tools, plus an extra one for formatting source code, are available in Delphi's `bin` folder:

- `AuditsCLI.exe` can be used to generate audit and metric reports in XML or HTML formats
- `Formatter.exe` lets you format the source code of an entire project using a given set of formatting rules, from the command line
- `DocGenCli.exe` lets you run the documentation generator (which produces extensive documentation in HTML format) from the command line.

Third-Party Tools

Delphi XE includes a significant set of third party tools. I've already mentioned Beyond Compare (which is included in a feature-limited version), while talking about Subversion. The three tools introduced here are more tied to the development process and they are also included as feature-limited versions.

Since the integrated tools are quite complex and feature rich, and considering that they are quite well documented on their respective sites, in this section I'm only going to list them, mention their key features, and offer a minimal introduction in case you have no idea about them.

CodeSite

CodeSite is an “*advanced logging solution*” by Raize Software, Ray Konopka's company. Ray is a Delphi guru, well-known book author and conference speaker, and also author of a very interesting set of visual controls for the VCL, known as Raize Components. For more information see

■ www.raize.com

The version of CodeSite included in Delphi XE is called CodeSite Express and it has most of the features of the full blown version. CodeSite lets you add

high-level statements to your code, that will either send the information to a live viewer or save them to a file for later inspection.

The approach used by CodeSite is conceptually comparable to the use of the `OutputDebugString` API function, which sends log information to a debug viewer external to the application (or Delphi's own Event page in case you are debugging the application). All `OutputDebugString` can do, though, is send strings to a viewer. CodeSite offers significant advantages over this simple approach in two separate areas.

First, it can send strings, categorize them, add annotations, and also log the status of entire objects, with all of their properties; its advanced viewers display log entries in categories, can nest log entries (like those of nested method calls, although the automatic method tracer is not available in the Express edition of the product) and display the status of objects and components with a view similar to Delphi's Object Inspector.

Second, you can have a local live viewer, remote viewers, log to file and open the file in a viewer later on, providing a very large degree of flexibility. To open Live Viewer you use the corresponding menu of the Tools | CodeSite menu.

To start CodeSite from your programs, you first need to add the corresponding unit to the uses statement (`CodeSiteLogging`), which you can expand by using the `cs1` Live Template. Simply type those three letters and you get the complete unit name. There other Live Templates which you can use to add logging statements. For example, you can use `css`, `csw`, and `cns` to generate the following calls:

```
CodeSite.Send( 'data', data );
CodeSite.SendWarning( 'msg' );
CodeSite.SendNote( 'msg' );
```

The `Send` method can be used to pass a simple string message or a message and an object. The other calls are treated differently by the viewer.

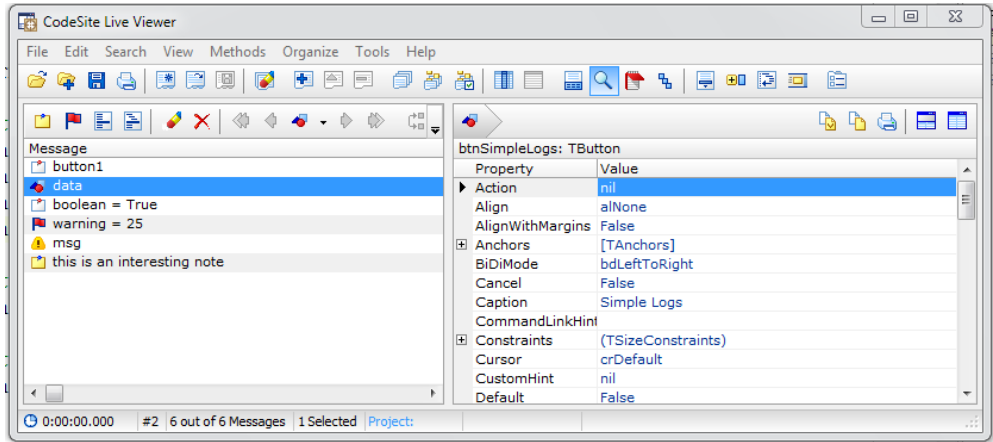
As an example in the `LoggingDemo` project, I've written the following code:

```
{css template}
CodeSite.Send( 'button1' );
CodeSite.Send( 'data', Sender );
CodeSite.Send( 'boolean', Sender is TObject);
CodeSite.Send( csmCheckPoint, 'warning', Left);

{csw template}
CodeSite.SendWarning( 'msg' );

{cns template}
CodeSite.SendNote( 'this is an interesting note' );
```

The output of these statements in the viewer can be seen below:

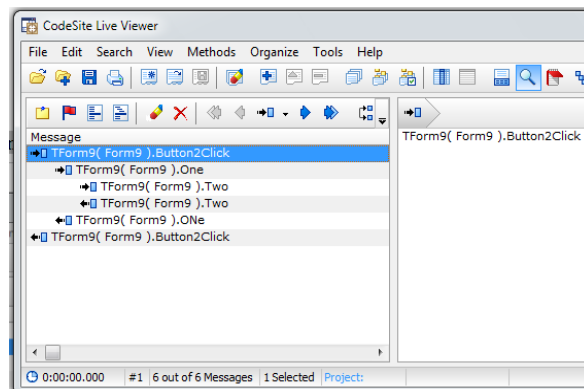


Notice that to obtain the image above I selected the message with an object attached to it. The object's data is visible on the right side.

Besides logging, you can also use CodeSite to trace the execution of methods and nested methods, by instrumenting your code with calls to `EnterMethod` and `ExitMethod` (which as I already mentioned can be added automatically in the full version of the product). This is an example, in which the `One` procedure (also instrumented) calls a method `Two`:

```
procedure TForm9.BtnMethodsTracingClick(Sender: TObject);
begin
    CodeSite.EnterMethod( Self, 'Button2Click' );
    One;
    CodeSite.ExitMethod( Self, 'Button2Click' );
end;
```

As you execute this code, you'll get the following log:



AQTime

AQTime is a well-known performance profiler by SmartBear Software, or (in their own words), “a performance profiler and memory and resource allocation debugger for 32-bit and 64-bit Windows”. The product web site is currently at:

■ smartbear.com/products/development-tools/performance-profiling/

To profile a program in Delphi, rather than running it as a standalone application or in the debugger, you have to run it through the profiler, using the Run with Profiling command of the AQTime menu of the IDE. In this way the tool will track the methods being called and at the end it will produce various types reports: performance profiling (where is the application spending most CPU time), coverage analysis (which methods are never executed), and others. You can also focus on memory and resource usage.

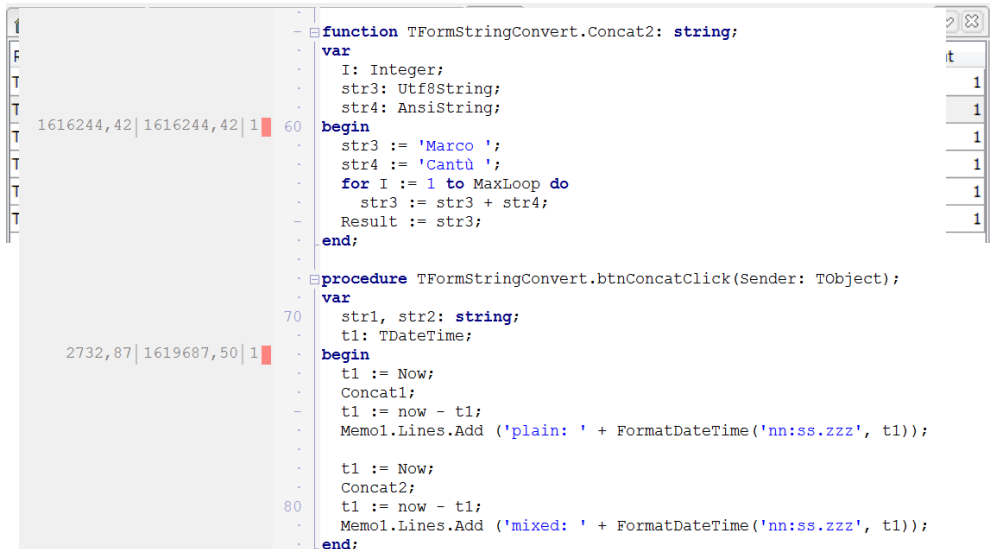
As a simple example I've adapted an existing program so that it can be profiled more easily. The program has a couple of buttons calling some rather slow methods, so that we can understand the resulting report more easily. These are some of the key methods:

```
procedure TFormStringConvert.btnConcatClick(Sender: TObject);
var
    str1, str2: string;
    t1: TDateTime;
begin
    t1 := Now;
    Concat1;
    t1 := now - t1;
    Memo1.Lines.Add ( 'plain: ' + FormatDateTime('nn:ss.zzz', t1));

    t1 := Now;
    Concat2;
    t1 := now - t1;
    Memo1.Lines.Add ( 'mixed: ' + FormatDateTime('nn:ss.zzz', t1));
end;
```

We can run this application through the profiler, using the AQTime | Run with Profiler menu. Leave the default options (Win32 Threads as Thread model and Elapsed Time as Active counter) and run the application. Now click on its buttons (I've done this on each of them), stop the program, and you'll be presented with a report like the the one in the following page.

The same information listed in this report (from which I excluded system function calls) is visible also along-side the source code, with a red colored element indicating the bottlenecks:



Of course, this display can be completely configured. AQTime has a large number of settings described in the details in the online documentation you can find at their web site.

FinalBuilder

FinalBuilder is a tool that lets you automate complex compilation and deployment processes, with the clear goal of supporting the continuous integration development model. Unlike the other third party tools, which are available in all editions of Delphi, FinalBuilder is a free tool only for the users of the Enterprise and Architect editions.

FinalBuilder is a product of VSoft Technologies Pty Ltd, led by long time Delphi expert Vincent Parrett. You can find information about the product at:

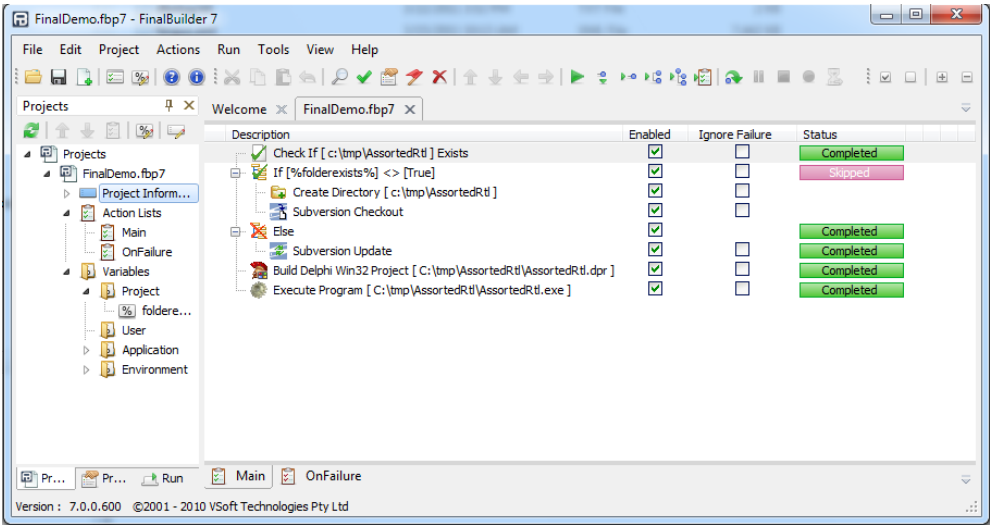
■ www.finalbuilder.com

Rather than writing batch files or using XML scripts to automate builds, FinalBuilder offers an easy-to-use graphical user interface, for both creating and running your builds. It has a large collection of over 500 core scripts, which let you integrate with most version control systems, compile code with different tools, create installers, send notifications, run test suites, deploy files with FTP, and many others.

56 - Chapter 2: Integrated Tooling

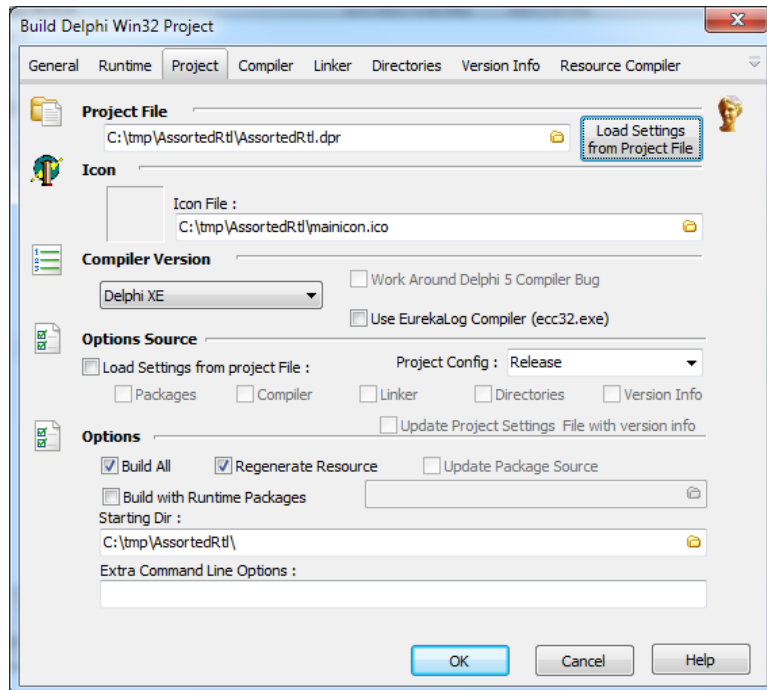
The Professional version of the software includes a continuous integration server, which lets you use a server-based build management system through a web interface, giving you remote control over your build projects.

Since the tool is mostly visual in nature, it isn't easy to show you the steps for creating a project. You can see the visual representation of my simple project (called FinalDemo) in the image below:



Here is a short description. The first action checks the existence of a given folder and sets a custom project variable. If the variable is set to False, it will create the folder and do a checkout operation on subversion, otherwise it will do an update subversion request. The two final steps are the project compilation and the execution of the resulting program.

Below I've included a second screen shot with the settings for the compilation action, but I really suggest you open to the project the play with it, to understand how it works:



The demo project uses my public repository, and should have no dependencies on file locations on my computer, but it is hard to tell it will run everywhere. Also, remember that if you have the Professional or Starter versions of Delphi, FinalBuilder is not included.

Chapter 3:

Compiler And RTL

Having looked at a number of new features of the Delphi XE IDE, it is now time to start getting to the updates to the Delphi compiler and core libraries. In this chapter I'll focus on the Run Time Library changes, while in the next I'll focus on the other Delphi XE libraries.

To be honest, Delphi XE is not a big update in these areas, not having the same impact as the last few versions. While the compiler has only minor changes, the RTL sees a significant number of enhancements, from the inclusion of a regular expressions library to significant changes to threading and synchronization support.

As you'll see by reading this chapter, it might not be an astonishing new version, but the number of additions is enough to keep most developers happy.

The Delphi Compiler

With the compiler R&D team focused on multi-platform support and the new native Win64 Delphi compiler, it doesn't come as a surprise that there are no significant changes to the language in Delphi XE, only some minor corrections and tweaks.

These are still enough to make the compiler in Delphi XE a new version (which implies DCU incompatibility with the previous one, as usual). The version is now 22.0, and the corresponding predefined symbol is

`VER220`

Different Defaults, Different Result

The first notable change to the compiler, is not a change to the compiler itself. For new Delphi projects, the Debug configuration has two changes compared to Delphi 2010:

- Compiler optimizations are turned off (\$O-)
- Stack frames are turned on (\$W+)

These new settings make sense for a debug version, and in fact the standard Release configuration has the optimizations turned on as in the past. Having no optimizations makes the debugging experience much better, as you don't get to see the common “value not available due to optimization” all over the place.

However, if you compare the efficiency of the Debug version of an application compiled in Delphi 2010 and Delphi XE you might find some surprises: with optimizations turned off you can expect the Delphi XE version to be somewhat slower, at times even much slower.

No More String Checks

In Delphi 2009 the Delphi compiler introduced the \$STRINGCHECKS directive to help migration of the C++Builder code to Unicode. The directive used to introduce extra string compatibility checks at run time, slowing down execu-

tion, even if marginally¹⁷. I used to suggest turning the directive off, unless you had Delphi code called by C++Builder code.

Now in Delphi XE not only the directive is off, but it has been disabled. Leaving it in your code won't produce an error, but it won't have any effect at all. The only possible behavior is now

```
■ {$STRINGCHECKS OFF}
```

This means the *EnsureUnicode* and *EnsureANSI* calls previously added by the compiler are now gone forever. Of course, the related setting has also been removed from the Compiler options in the IDE.

New Alignments

While Delphi has had a directive used to align data structures to different byte boundaries for a long time, this has been now extended to include 16-byte alignment. Data alignment is used by data structures like records and classes to improve the access speed to individual fields for some CPU architectures. With `{$ALIGN 1}` the compiler will save on memory usage by using all possible bytes, just like when you use the packed specifier for a record. At the other extreme, the new `{$ALIGN 16}` will use the largest alignment.

As an example (which code is in the *AlignTest* project) let's consider the following structure:

```
type
  TMyRecord = record
    c: AnsiChar;
    w: Word;
    b: Boolean;
    i: Integer;
    d: Double;
  end;
```

With `{$ALIGN 1}` the structure takes 16 bytes (the value returned by `SizeOf`) and the fields will be at the following relative addresses¹⁸:

-
- 17 For a purely historical perspective of the effect of this directive you can refer to the *StringConvert* example part of my *Delphi 2009 Handbook*. Notice, however, that it really makes no sense to try it out in Delphi XE, since the directive and the underlying behavior have been removed.
 - 18 Relative addresses are computed by allocating the record and computing the difference between the numeric value of a pointer to the structure and that of a pointer to the given field, with an expression like: `Integer(@MyRec.w) - Integer(@MyRec1)`.

```
| c: 0 w: 1 b: 3 i: 4 d: 8
```

In contrast, if you change the alignment to 4 (which can lead to optimized data access) the size will be 20 bytes and the relative addresses:

```
| c: 0 w: 2 b: 4 i: 8 d: 12
```

Again, the only change from the past for this important directive is the introduction of `{ $ALIGN 16 }`, which for the structure requires 24 bytes and maps the fields as follow:

```
| c: 0 w: 2 b: 4 i: 8 d: 16
```

There is also, however, a related directive that is brand new in Delphi XE. This is the `$CODEALIGN` directive, which has a similar effect for code, that is the position of the beginning of the code of a function or method in the compiled code. Possible values are the same of the `$ALIGN` directive.

RTTI and Virtual Methods Interceptors

One of the most interesting changes to the compiler in Delphi 2010 was the Extended RTTI, which was fully covered in my Delphi 2010 Handbook. Delphi XE provides one extra feature on top of it, which is the ability to intercept the execution of virtual methods of an existing class, by creating a proxy class for an existing object. In other words, you can take an existing object and change its virtual methods (a specific one, or all of them at once).

Why would you want to do this? In a standard Delphi application, you probably would not use this feature. If you need an object with a different behavior, just change it or create a subclass. Things are different for libraries, because libraries should be written in a very generic way, knowing little about the objects they'll be able to manipulate, and imposing as little burden as possible on the objects themselves. This is the kind of scenario for which Virtual Methods Interceptors were added to Delphi XE. However, this feature is basically not used anywhere in the RTL or VCL¹⁹, at least for now!

19 Barry Kelly claims the original idea was to use this feature in DataSnap, but it was later introduced in the language even if DataSnap didn't use it. You can find this idea, along with a very nice description of Virtual Method Interceptors (to which I owe a lot) in his blog at <http://blog.barrkel.com/2010/09/virtual-method-interception.html>.

Before we focus on possible scenarios, let me discuss the technology itself. Suppose you have an existing class with at least one virtual method, like the following:

```
type
  TPerson = class
  ...
  public
    property Name: string read FName write SetName;
    property BirthDate: TDate read FBirthDate write SetBirthDate;
    function Age: Integer; virtual;
    function ToString: string; override;
  end;

function TPerson.Age: Integer;
begin
  Result := YearsBetween (Date, FBirthDate);
end;

function TPerson.ToString: string;
begin
  Result := FName + ' is ' + IntToStr (Age) + ' years old';
end;
```

Now what you can do is to create a `TVirtualMethodInterceptor` object (a new class defined in the `RTTI` unit) tied to the class of the object, changing the object's static class to the dynamic one:

```
var
  vmi: TVirtualMethodInterceptor;
begin
  vmi := TVirtualMethodInterceptor.Create(TPerson);
  vmi.Proxyify(Person1);
```

Once you have the `vmi` object you can *install* special handlers for its events (`OnBefore`, `OnAfter`, and `OnException`) using anonymous methods. These will be triggered before any virtual method call, after any virtual method call, and in case of an exception in the virtual method. These are the signatures for the three anonymous method types:

```
TInterceptBeforeNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue);
TInterceptAfterNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; var Result: TValue);
TInterceptExceptionNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out RaiseException: Boolean;
  TheException: Exception; out Result: TValue);
```

64 - Chapter 3: Compiler and RTL

In each event you get the object, the method reference, the parameters, and the result (which might be already set or not). In the `OnBefore` event you can set the `DoInvoke` parameter to disable standard execution. In the `OnExcept` event you get information about the exception.

In the `InterceptBaseClass` demo, which uses the `TPerson` class above, I've intercepted the class virtual methods with this logging code:

```
procedure TFormIntercept.btnInterceptClick(Sender: TObject);  
begin  
    vmi := TVirtualMethodInterceptor.Create(TPerson);  
    vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;  
        const Args: TArray<TValue>; out DoInvoke: Boolean;  
        out Result: TValue)  
    begin  
        Log('Before calling ' + Method.Name);  
    end;  
    vmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;  
        const Args: TArray<TValue>; var Result: TValue)  
    begin  
        Log('After calling ' + Method.Name);  
    end;  
    vmi.Proxify(Person1);  
end;
```

Notice that the `vmi` object needs to be kept around at least until the `Person1` object is in use, or you'll be use a dynamic class that's not available any more and you'll be calling anonymous methods that have already been released. In the demo, I've saved it as a form field, just like the object to which it refers.

The program uses the object by calling its methods and checking the base class name:

```
Log ('Age: ' + IntToStr (Person1.Age));  
Log ('Person: ' + Person1.ToString);  
Log ('Class: ' + Person1.ClassName);  
Log ('Base Class: ' + Person1.ClassParent.ClassName);
```

Before you install the interceptor, the output is:

```
Age: 26  
Person: Mark is 26 years old  
Class: TPerson  
Base Class: TObject
```

After you install the interceptor, the output becomes:

```
Before calling Age  
After calling Age  
Age: 26  
Before calling ToString  
Before calling Age  
After calling Age  
After calling ToString
```

```

Person: Mark is 26 years old
Class: TPerson
Base Class: TPerson

```

Notice that the class has the same name as the base class, but it is in fact a different one, the dynamic class created by the Virtual Method Interceptor.

Although there has no official way to restore the class of the target object to the original one, the class itself is available in the Virtual Method Interceptor object and also as base class of the object. Still, you can use *brute force* to assign to the class data of the object (its initial four bytes) the correct class reference:

```

PPointer(Person1)^ := vmi.OriginalClass;

```

As a further example, I've modified the `OnBefore` code so that in case you are calling `Age`, it returns a given value and skips executing the actual method:

```

vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue)
begin
  Log( 'Before calling ' + Method.Name);
  if Method.Name = 'Age' then
  begin
    Result := 33;
    DoInvoke := False;
  end;
end;

```

The output changes from the version above, as follows (notice that the `Age` calls and the relative `OnAfter` events are skipped):

```

Before calling Age
Age: 33
Before calling ToString
Before calling Age
After calling ToString
Person: Mark is 33 years old
Class: TPerson
Base Class: TPerson

```

Now that we have seen the technical details behind Virtual Methods Interceptor, we can get back to figure out in which scenarios you'd want to use this feature. Again, there is basically no reason to use this in a standard application. The focus, instead, is mostly for those who develop advanced libraries and need to implement custom behavior for testing or processing objects.

For example, this could help building a Unit Testing library, although it would be limited to virtual methods only. You'd also possibly use this along with custom attributes to implement a coding style similar to Aspect Oriented

Programming. I guess we'll need to wait a few years before we can conclude whether this had been a useful addition to the Delphi compiler.

The Run Time Library

While the compiler in Delphi XE has seen only limited changes (with the compiler team already focused on building new versions of the compiler for Mac OS X and 64-bit Windows), the RTL sees a few relevant and somewhat strategic extensions, plus a ton of minor fixes and improvements.

Among the key extensions to the RTL there is native support for regular expressions, binary readers and writers, and significant improvements in the threading and synchronization support. I'll focus on these three topics in later sections of this chapter, but will start with a list of minor changes, some of which I find particularly handy.

String List Changes

Considering that `TStringList` is one of the most widely used classes of the Delphi RTL, it is worth starting with this one.

With the introduction of Unicode, you can specify a second parameter indicating the encoding to the `SaveToFile` method of the string list. The optional parameter is a `TEncoding` derived class. But the rule was that if you didn't provide a parameter the file would use the classic ANSI encoding. In Delphi 2009 handbook I showed *trick* based on a class helper to change the default format.

Now in Delphi XE there is a new specific property to indicate the default encoding (the property is called `DefaultEncoding`) to be used when saving the strings to a file when no encoding parameter is passed. A second new property `WriteBOM` lets you skip the Begin of file Marker, which is very common for Unicode text files but it is not considered mandatory (in fact, some systems require you *not* to use it).

Another very nice extension is the availability of conversion functions from a string list to an array of strings (`TArray<string>`) and the ability to add an array to a string list:

```

type
  TStrings = class(TPersistent)
  public
    procedure AddStrings(const Strings: TArray<string>); overload;
    function ToStringArray: TArray<string>;

```

Dates and Files

There are several extensions related with dates. In the DateUtils unit there is a new `TTimeZone` class supporting conversion of times among time zones, printing out the names of the time zones, and much more. For example, a handy feature is the ability to get the name of the current time zone and its abbreviation by calling:

```

TTimeZone.Local.DisplayName
TTimeZone.Local.Abbreviation

```

I have these calls in the AssortedRtl demo project and they return, the following values in the case of standard or daylight savings time, respectively:

```

W. Europe Standard Time
GMT+01
W. Europe Daylight Time
GMT+02

```

There are other properties and methods to return a `TTimeSpan` from UTC, check for daylight savings and more. By the way, in the TimeSpan unit there are new overloaded operators for `TTimeSpan`.

Times stamps are also used by files: The `TSearchRec` structure in the SysUtils unit now has a `TimeStamp` property which returns the `TDateTime` object containing the date and time the file was last modified. This is for easy cross-platform consumption when using the traditional `FindFirst` and `FindNext` functions.

Also on the file system, there is a `GetHomePath` function in SysUtils, which is also wrapped by the `TPath.GetHomePath` method of the recent IOUtils unit (introduced in Delphi 2010 and now considerably cleaned up in anticipation of its cross-platform role). The call on Windows returns the applications data folder, *AppData\Roaming*). As you can see in the AssortedRtl demo project, calling this method on my computer returns the value:

```

C:\Users\Marco\AppData\Roaming

```

Splitting Strings

Among the global functions, there is a new one in StrUtils that I find very interesting, `SplitString`:

function `SplitString`(const S, Delimiters: string): TStringDynArray;

Compared to other techniques, like the very popular use of a `TStringList` to split a long string based on a line separator, using `SplitString` is much easier and the resulting code is much faster.

As a test, in the AssortedRtl demo project I've written the following code that keeps calling the `SplitString` function in a loop (and also assigns the same string to the `Text` of a string list):

```
var
  res: TStringDynArray;
begin
  for i := 2 to 1000 do
    begin
      str := str + '|' + IntToStr (i);
      res := SplitString (str, '|');
    end;

  var
    sl: TStringList;
  begin
    sl:= TStringList.Create;
    sl.LineBreak := '|';
    for i := 2 to 1000 do
      begin
        str := str + '|' + IntToStr (i);
        sl.Text := str;
      end;
```

The timing for the first call on my computer is (on average) 116 milliseconds, while the second *classic* version takes about 2,174 milliseconds, which is more than 10 times as much. Quite impressive.

Command Line Parsing

When your application receives parameters on the command line, like the names of the files to work with or specific options for your application, you can use the `ParamCount` and `ParamStr` functions to retrieve the number of parameters and the value of each.

In case you let users pass command line parameters to your application, it is handy to find them using the `FindCmdLineSwitch` function, that receives as input the switch name (either case sensitive or not). Now there are new overloads to this function for writing portable code, since the characters used to mark out parameters (like `/` and `-`) depend on the operating system platform. In Linux, for example, the `/` cannot be used for command line parameters, as it is the path delimiter.

Another extension lets you specify how a value follows the switch name, as in “`-p Value`”, “`-pValue`” or “`-p:Value`”. There are now different switch types you can use to enforce a given format or let it be flexible, accepting multiple styles. This is the signature of this version of the function:

```
function FindCmdLineSwitch(const Switch: string;
    var Value: string; IgnoreCase: Boolean = True;
    const SwitchTypes: TCmdLineSwitchTypes =
        [clstValueNextParam, clstValueAppended]): Boolean; overload;
```

The default is to ignore the switch case and accept all switch formats. As an example, I've added the following code to the `OnCreate` event handler of the main form of the `AssortedRtl` application:

```
var
    pValue: string;
begin
    if FindCmdLineSwitch ('p', pValue) then
        Log ('-p ' + pValue);
```

This means that if you pass “`-p myparameter`” on the command line you'll get the same value in the Memo component used by `Log`. (From Delphi you can use the `Run | Parameters` menu to display the `Debug Configuration` dialog box and set the `Parameters` field.)

GUID Helper

If you've ever user GUIDs directly in your code, you probably know they are slightly complicated to handle. The `TGUID` record is rather plain, and the support functions are spread in different units, and often hard to remember.

This is why those who use GUIDs will certainly consider a nice addition the `TGuidHelper` record helper introduced in Delphi XE. It lets you apply GUID support functions directly to the record structure. As the record helper definition clearly shows, it adds several `Create` initialization function, a handy `NewGuid`, and a couple of conversion functions to strings and byte arrays:

```

type
  TGUIDHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID;
      overload; static;
    class function Create(const S: string): TGUID;
      overload; static;
    class function Create(A: Integer; B: SmallInt; C: SmallInt;
      const D: TBytes): TGUID; overload; static;
    class function Create(A: Integer; B: SmallInt; C: SmallInt;
      D, E, F, G, H, I, J, K: Byte): TGUID; overload; static;
    class function Create(A: Cardinal; B: Word; C: Word;
      D, E, F, G, H, I, J, K: Byte): TGUID; overload; static;
    class function NewGuid: TGUID; static;
    function ToByteArray: TBytes;
    function ToString: string;
  end;

```

Number of CPU Cores

Finally, a feature that's part of multi-threading support, but can also be useful also as a general purpose routine, is the ability to ask the system for the number of processor cores, using `ProcessorCount` class property of the `TThread` class. This is the last piece of code of the `AssortedRtl` demo I'm going to list:

```

Log ( 'TThread.ProcessorCount: ' +
  IntToStr (TThread.ProcessorCount));

```

Binary Reader and Writer

Along with Unicode support, Delphi 2009 saw the introduction of a new powerful mechanism for working with text files, rather than relying on the direct use of streams and having to cope manually with Unicode string formats and other low level issues. The higher level abstract classes introduced were `TTextReader` and `TTextWriter`, and the implementation classes were `TStreamReader` and `TStreamWriter`, plus the lesser used `TStringReader` and `TStringWriter`. I covered these classes, which mimic their .NET counterparts, in my *Delphi 2009 Handbook*.

The reason for bringing this up is that Delphi XE extends this file processing architecture with two new classes, `TBinaryReader` and `TBinaryWriter`, which are meant for managing binary data rather than text files. These classes

generally encapsulate a stream (a file stream or any type of in-memory stream, including sockets and BLOB fields) and have overloaded Read and Write methods.

As a (rather simple) example I've written the BinaryFiles application. In its first part, this program writes a couple of binary elements to a file (the value of a property and the current time) and reads them back, re-assigning the property value:

```
procedure TFormBinary.btnWriteClick(Sender: TObject);
var
    bw: TBinaryWriter;
begin
    bw := TBinaryWriter.Create('test.data', False);
    try
        bw.Write(Left);
        bw.Write(Now);
        Log ('File size: ' + IntToStr (bw.BaseStream.Size));
    finally
        bw.Free;
    end;
end;

procedure TFormBinary.btnReadClick(Sender: TObject);
var
    br: TBinaryReader;
    time: TDateTime;
begin
    br := TBinaryReader.Create('test.data');
    try
        Left := br.ReadInt32;
        Log ('Left read: ' + IntToStr (Left));
        time := br.ReadDouble;
        Log ('Time read: ' + TimeToStr (time));
    finally
        br.Free;
    end;
end;
```

The key rule in using these reader and writer classes is that you have to read the data in the same order you wrote it, or else you'll totally mess up the data. In fact, only the binary data of individual fields is saved, with no information about the field itself. Nothing prevents you from interposing data and metadata in the file, like saving the size of the next data structure before the actual value or a token referring the field (or even a string, although this partially defeats the idea of a binary stream).

Using these reader and writer classes is nice, but I don't find them as useful as the text readers and writers. Suppose for example you have a simple record structure like the following:

```
type
  TMyRecord = record
    one: Integer;
    two: Integer;
    three: Double;
  end;
```

How do you save it to a binary stream using this interface? One option is to write (and read) the individual fields directly, as in:

```
bw := TBinaryWriter.Create( 'record.data', False);
try
  bw.Write(rec.one);
  bw.Write(rec.two);
  bw.Write(rec.three);
```

This doesn't scale really well, I have to say. It might end up being easier to use the traditional low-level stream and save all of the data at once with a call like:

```
aStream.Write (rec, Sizeof (rec));
```

Another alternative would be to use the Extended RTTI to get the list of fields and their values. The problem, however, is that we cannot pass a `TValue` to the binary writer, but need to call a method specific for the type and also extract the data from the `TValue` depending on the type. This is certainly possible, but it is far from trivial and might not be worth it.

In the demo, I've used this approach with support limited to integer and double fields (look in the project code for the function calls and some more details):

```
procedure WriteRecordBinary (recordType: TRttiRecordType;
  instance: Pointer; bw: TBinaryWriter);
var
  field: TRttiField;
begin
  for field in recordType.GetFields do
    begin
      case field.FieldType.TypeKind of
        tkInteger: bw.Write (field.GetValue(instance).AsInteger);
        tkFloat: bw.Write (field.GetValue(instance).AsType <Double>);
      end;
    end;
  end;

procedure ReadRecordBinary (recordType: TRttiRecordType;
  instance: Pointer; br: TBinaryReader);
var
  field: TRttiField;
begin
```

```

    for field in recordType.GetFields do
    begin
        case field.FieldType.TypeKind of
            tkInteger: field.SetValue(instance, br.ReadInteger);
            tkFloat: field.SetValue(instance, br.ReadDouble);
        end;
    end;
end;

```

Given the complexity of this code and the limited advantage compared to some more traditional coding styles (like dumping entire data structures with a direct stream write) this might not be always worth it. However, dumping data structures doesn't work for strings and other reference types, so it is true that if you have complex data structures you want to save to a binary stream, the RTTI based code above could be a useful starting point.

Regular Expressions

One of the significant new features of Delphi XE is the availability of a regular expression engine plugged right into the core RTL. It was possible, in fact, to plug-in regular expression support into Delphi applications in the past using a third-party component. One of such component has been licensed by Embarcadero and added to Delphi, after creating a nice set of wrapper classes for it.

A Regular Expressions Primer

What is a regular expression or *regex*? It can be described as wild-cards on steroids, or (more formally) as a pattern describing a character string or a portion of text. In its simplest form, the regular expression “a” will let you locate the letter *a* within a target text. The expression “text” will match the sequence of those four characters. Sounds simple? Well, it isn't! Besides literal values there are 11 meta-characters you can use to express complex rules. This makes the regex notation extremely powerful but far from trivial and almost unreadable.

The 11 meta-characters have special meanings and cannot be used directly in a regex (if you need to use them you have to escape them with a backslash): [, \, ^, \$, ., |, ?, *, +, (, and). Some of the ways these symbols are used is

explained in the following table, which might help you understand the following demos, but it far from a complete guide:

[]	Character class and shorthands	[aeiou] matches any vowel, \w is any letter or digit, [0-9] matches any digit (its shorthand is \d)
[^]	Negated class	[^aeiou] matches anything but vowels
*	Repeat 0 or more	p[0-9]* matches a <i>p</i> optionally followed by numbers
+	Repeat 1 or more	x[0-9]+ matches an <i>x</i> followed by at least a number
{ }	Repeat given times	[aeiou]{2,4} matches a sequence of 2 to 4 vowels
	alternation (or)	oo ee matches one of these two double letters
?	optional	fi re? matches fire and fir, as the <i>e</i> is optional
()	grouping	f(i re)? matches fire and f, the <i>ire</i> group is optional
^	start	^a matches a string starting with a
\$	end	\$a matches a string ending with a
.	anything	. matches any character, but the new line

We'll see some examples of regular expressions in the Delphi demos of the following sections. As there is no way a short section of a book can make justice to this complex topic, I can recommend the great web site by Jan Goyvaerts²⁰ for more information on the topic:

■ <http://www.regular-expressions.info>

The site has both a quick start and a complete online guide, and references to the book Jan has written for O'Reilly, "Regular Expressions Cookbook".

²⁰ Jan Goyvaerts is also a Delphi developer. He's an expert in Unicode (and helped me with the Unicode section of my Delphi 2009 Handbook) and, of course, of regular expressions. He has authored regex software including editors to help you write regular expressions and the engine that Embarcadero has added to Delphi XE.

Regular Expressions in Delphi XE

Jan Goyvaerts has also donated his TPerlRegEx Delphi regex engine to Embarcadero, for inclusion in Delphi XE. Along side this, Vincent Parrett has offered his .NET-style regex interface classes, which the company modified and extended for inclusion in the product. As a result of these steps, we now have two units in Delphi XE:

- The `RegularExpressionCore` unit has the interface to the PCRE (Perl-Compatible Regular Expressions) engine written by Philip Hazel and available at <http://www.pcre.org/>. The Delphi unit refers to the library API (see the `RegularExpressionsAPI` unit), but it links in the compiled C language *.obj* files, so that the external DLL is not required. The `RegularExpressionCore` unit can be called directly, but it is generally used through the interface provided by the `RegularExpressions` unit.
- The `RegularExpression` unit contains several records that simplify using the Regular Expressions engine. The three main record types are `TGroup`, `TMatch`, and `TRegEx`: the latter is the entry point for matching and replacing strings using result expressions. Although they are records, these data structures resemble the corresponding .NET classes. Records are used to simplify memory management.

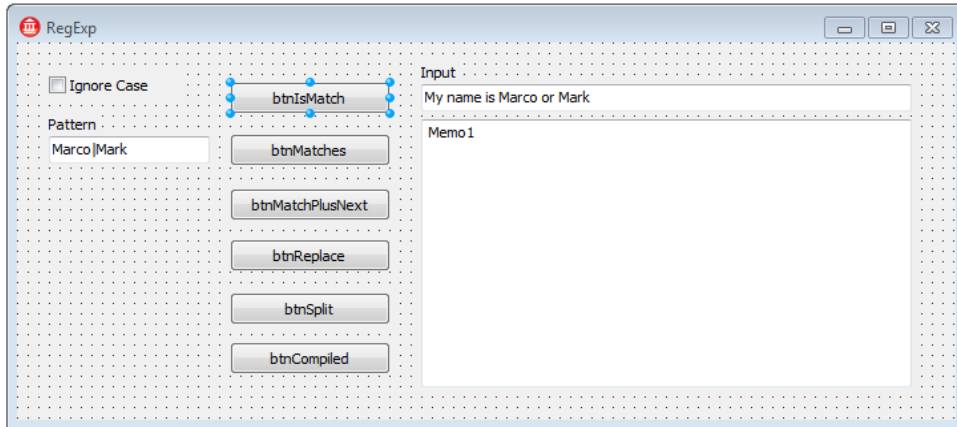
The main structure you use for regular expressions operations is `TRegEx`, which has a limited set of methods (each with several overloaded versions):

- `IsMatch` returns a Boolean value, validating the input string
- `Match` returns the first match, and you can later ask for following matches
- `Matches` returns a collection of matches
- `Replace` lets you modify the matched values
- `Split` returns an array of strings, split on the matching elements

All of these methods are available both as class methods and regular ones. The reason lies in the fact that the class methods are certainly easier to use, as you simply call them directly. However, the PCRE works by compiling and executing regular expressions as two separate steps. If you need to use the same regular expression multiple times, compiling it once and reusing the compiled version is more efficient. This is what happens if you create a record, initialize it with the regular expression, and then use the same record over and over.

The RegExPrimer Example

Now that you know the basics of the library and the Delphi interface, we can look at an actual Delphi example. The form of this example has two input boxes for the string to check and the RegEx pattern (the value of these input boxes is read by two internal functions, `InputText` and `PatternText`) and buttons for the various operations:



The first operation is the core `IsMatch` test, you use to verify if a string satisfies a given expression (the two parameters are in that order):

```
procedure TFormRegEx.btnIsMatchClick(Sender: TObject);
var
    bMatch: Boolean;
    regOpt: TRegExOptions;
begin
    if cbIgnoreCase.Checked then
        regOpt := [roIgnoreCase];
    bMatch := TRegEx.IsMatch(InputText, PatternText, regOpt);
    Log(PatternText + ' matches ' + InputText + ':' +
        BoolToStr (bMatch, True));
end;
```

The code uses the status of a check box to make the match case insensitive or not. With this flag disabled, you'll get these results (of course, the program is there for you to test more regular expressions):

```
Marco|Mark matches My name is Marco or Mark:True
John matches My name is Marco or Mark:False
Mark matches My name is Marco or Mark:True
M matches My name is Marco or Mark:True
```

The second and third buttons are used to find all of the matches not just verifying that the input satisfies the expression. The first version of the code simply gets all of the matches in an array:

```
procedure TFormRegEx.btnMatchesClick(Sender: TObject);
var
    match : TMatch;
    matches: TMatchCollection;
begin
    matches := TRegEx.Matches (InputText, PatternText);
    for match in matches do
        begin
            Log( 'Found ' + match.Value + ' at ' + IntToStr (match.Index));
        end;
    end;

    // output (with default values)
    Found Marco at 12
    Found Mark at 21
```

This is easy to use, but in case of a long string will find many matches and take quite some time. So it might be better to find the matches one by one and decide each time whether to continue looking or to stop (in the specific demo the output will be the same):

```
procedure TFormRegEx.btnMatchPlusNextClick(Sender: TObject);
var
    regex: TRegEx;
    match : TMatch;
begin
    regex := TRegEx.Create (PatternText);
    match := regex.Match(InputText);
    while match.Success do
        begin
            Log( 'Found ' + match.Value + ' at ' + IntToStr (match.Index));
            match := match.NextMatch;
        end;
    end;
```

Notice that in this case I had to explicitly declare an instance of the TRegEx record, rather than using a class function. In fact, due to a memory management problem²¹, if you write the following rather than the first two statements, you'll easily get an Access Violation error:

```
| match := TRegEx.Match (InputText, PatternText); // watch out!
```

21 The error is due to the fact that the temporary TRegEx record goes out of scope too early and frees an interface it refers to. More details on another blog post by Jan Goyvaerts, at <http://www.regexguru.com/2010/09/bug-in-delphi-xe-regularexpressions-unit/>.

A different operation is to use regular expressions to replace some of the text with a different characters. This can be extremely powerful, particularly if the replacement depends on the actual match. In fact, you can use a simple replacement (same text for each match) like the following:

```
Log (TRegEx.Replace(InputText, strPattern, 'Joe'));
// outputs: My name is Joe or Joe
```

However, you can also provide the `Replace` function with an event handler²² you'll use to replace the text depending on the value of the match (in fact the method has the `Match` as its only parameter):

```
function TFormRegEx.ReplaceEvaluator(const Match: TMatch): string;
begin
  if Match.Value = 'Marco' then
    Result := 'Giovanni'
  else
    Result := 'Joe';
end;

// the actual call
Log (TRegEx.Replace(InputText, strPattern, ReplaceEvaluator));
// outputs: My name is Giovanni or Joe
```

The last button does a further operation, splitting the original string into sub-strings using the pattern as separator:

```
procedure TFormRegEx.btnSplitClick(Sender: TObject);
var
  strArray: Tarray<string>;
  str1: string;
begin
  strArray := TRegEx.Split(InputText, PatternText);
  for str1 in strArray do
  begin
    Log (str1);
  end;
end;
```

With the default input this call produces the following two lines of text, with two spaces around the “or” unless you `Trim` the strings before logging them:

```
My name is
or
```

22 Now, before you ask, I don't have the slightest idea why they used an event handler rather than an anonymous method, which seems to be more appropriate in a similar circumstance. Since the `TRegEx` is not a component, you'll have to assign the event handler manually anyway, and using an anonymous methods would have given more flexibility (assigning an existing method or passing the function's code inline). It is also quite unusual for Delphi to have a procedural type (the type of an event handler) with a function signature!

Finally, I want to give you a feeling of the effect of caching the compiled regular expression (although the actual advantage likely depends on its complexity). In this code I first use a single `TRegEx` record ten thousand times and then repeat the process with ten thousand class function calls:

```
procedure TFormRegExp.btnCompiledClick(Sender: TObject);
var
  w: TStopwatch;
  regex: TRegEx;
  I: Integer;
begin
  w := TStopwatch.Create;

  w.Start;
  regex := TRegEx.Create(PatternText, [roCompiled]);
  for I := 1 to 10000 do
    begin
      regex.IsMatch(InputText);
    end;
  w.Stop;
  Log ('Compiled: ' + IntToStr (w.ElapsedMilliseconds));

  w.Start;
  for I := 1 to 10000 do
    begin
      TRegEx.IsMatch(InputText, PatternText);
    end;
  w.Stop;
  Log ('Not Compiled: ' + IntToStr (w.ElapsedMilliseconds));
end;
```

Notice that using the `roCompiled` flag is simply a request to compile the regular expression in the `TRegEx` constructor itself, rather than before the first usage, but the regular expression will be compiled anyway. So even without that flag you'll get the same exact timing. Speaking of this, what is the difference? Here is the output:

```
Compiled: 34
Not Compiled: 1514
```

That is 34 milliseconds versus one and a half seconds. What if we bump this up to hundred thousand? Not surprisingly we get the same ratio, at a different scale: 342 versus 15,229.

In other words, every time you want to apply the same regular expression to different input strings, use a single compiled `TRegEx` record: The speed difference is too large to ignore.

Threading and Synchronization

An area of the Delphi RTL that has seen a lot of changes in recent versions of the product is threading and synchronization support. Over the last few years anonymous methods in the `Synchronize` and `Queue` calls of the `TThread` class, and `TMonitor` were introduced. Also extensions tied to thread naming and new debugger features for multi-threaded applications were made. Delphi XE adds another significant set of additions, worth exploring in some detail.

Anonymous Threads

Many times you need to create a derived thread class only for the sake of embedding an algorithm into a background thread. As I covered in my Delphi 2009 Handbook, you can simplify repetitive operations in a thread by passing to a generic thread class a function to be executed, generally an anonymous method. This is what is directly available in Delphi XE using anonymous threads, that is calling the new `CreateAnonymousThread` class function of the `TThread` class.

Consider the need to execute a slow algorithm, like the following (which is basically useless, but serves my purpose of using a lot of CPU cycles):

```
const
    mil20 = 20000000; // 20 million

procedure TThreadingForm.btnPlainClick(Sender: TObject);
var
    X: Extended;
    I: Integer;
begin
    (Sender as TButton).Caption := 'Computing';
    X := 100;
    for I := 0 to mil20 do
        X := Sqrt(X) * Sqrt(X);
    (Sender as TButton).Caption := FloatToStr(X);
end;
```

This code is blocking, meaning that for the few seconds it takes to execute the application we'll be totally stuck. That's only one case for a background thread, the other (common one) is to perform socket-based requests using a blocking architecture like Indy. We'll see an example of this second case later.

What would be the traditional solution? Inherit a class from `TThread`, override the virtual abstract `Execute` method, and then create an object of this custom class. In complex situations, this is certainly worth it, but for a simple algorithm it is a lot of work.

Now you can use a ready-to-use `TThread` descendant classes, which is called `TAnonymousThread`. The class is not created directly but through a call to the `CreateAnonymousThread` class method. This creates an instance of the internal class, passing an anonymous method to it, as in this example implementing the previous algorithm in a thread:

```
TThread.CreateAnonymousThread(
  procedure()
  var
    X: Extended;
    I: Integer;
  begin
    X := 100;
    for I := 0 to mil20 do
      x := Sqrt (X) * Sqrt (X);
      (Sender as TButton).Caption := FloatToStr (X);
    end);
```

The thread is created suspended (so you'll have to call the `Start` method of the object returned by the function) and automatically disposed when the execution terminates as its `FreeOnTerminate` flag is turned on. Also, we cannot simply update the caption of the button (as in the code above) since this is not a thread-safe operation.

But how can we call synchronize if we don't have an instance of the thread object within the thread's anonymous method? We can use the `GetCurrentThread` function (or use an explicit thread object reference, as shown in the following case). This is the final version of the code based on the anonymous thread:

```
procedure TThreadingForm.btnAutoThreadClick(Sender: TObject);
begin
  (Sender as TButton).Caption := 'Computing';
  TThread.CreateAnonymousThread(
    procedure()
    var
      X: Extended;
      I: Integer;
    begin
      X := 100;
      for I := 0 to mil20 do
        x := Sqrt (X) * Sqrt (X);
      TThread.Synchronize (TThread.CurrentThread,
        procedure
```

```

        begin
            (Sender as TButton).Caption := FloatToStr (X);
        end);
    end).Start;
end;

```

Anonymous Threads for Indy

As I mentioned earlier, a very specific case for using simple, anonymous threads is when you need to perform an operation on a blocking sockets, as Indy components do. Consider this code that returns the first 200 characters from my web site:

```

var
    strHtml: string;
begin
    strHtml := IdHttp1.Get('http://www.marcocantu.com');
    memo1.Lines.Add(Copy (strHtml, 1, 200));

```

The code is blocking and we have no idea of how much time it will take, as the site might be temporarily unavailable or simple slow to respond. So we should create a thread for this operation, either using a full-blown thread or passing an anonymous method to a simple one, since we really have a limited use for the thread itself. Here is the modified code, which uses a reference to the thread object, to avoid the `CurrentThread` call (notice you have to call `Start` in a separate statement after the assignment):

```

var
    aThread: TThread;
begin
    aThread := TThread.CreateAnonymousThread(
        procedure ()
            var
                strHtml: string;
            begin
                strHtml := IdHttp1.Get('http://www.marcocantu.com');
                aThread.Synchronize(aThread,
                    procedure ()
                        begin
                            memo1.Lines.Add(Copy (strHtml, 1, 200));
                        end);
                    end);
    end);
    aThread.Start;

```

I suggest you use similar code every time you are using a blocking socket, either explicitly or implicitly and indirectly.

Other New Thread Methods

Beside the new `CreateAnonymousThread` class method, the `TThread` class has several other new features. There are two new class static procedures, `Sleep` and `Yield`, which map to the corresponding Windows API function. The reason for using these methods, though, is that on different operating systems, they are going to be mapped differently. In fact, if we look to the source code of `Sleep` in the RTL we can see the following:

```
class procedure TThread.Sleep(Timeout: Integer);
begin
  {$IFDEF POSIX}
    usleep(Timeout * 1000);
  {$ENDIF}
  {$IFDEF MSWINDOWS}
    windows.Sleep(Timeout);
  {$ENDIF}
end;
```

The sooner you migrate you code to these calls (and this migration at times might even be automatic, when you call `Sleep` from a thread the method will take precedence over the global function), the easier you'll be able to migrate your code to non-Windows platforms.

Two other extensions of the `TThread` class, instead, offer new functionality, namely the ability to get the number of processor cores available on the current computer. The two class properties are `ProcessorCount` (an `Integer`) and `IsSingleProcessor` (a `Boolean`). You can see a trivial example of using `ProcessorCount` in the `AssortedRtl` example.

The information about the number of processors is relevant whenever you are creating background threads to optimize performance on multi core systems. For example, in *Delphi 2009 Handbook* I wrote a sample *parallel for* loop, which should be optimized by using as many parallel threads as there are processor cores in the target computer.

Finally, Delphi XE introduces another class to help in making your code cross-platform, the new `TInterlocked` class defined in the `SyncObjs` unit. This is a sealed class, with a set of class methods (here listed only in the versions that manipulate `Integers`, but there are many overloaded versions of these methods):

```
type
  TInterlocked = class sealed
    class function Increment(var Target: Integer): Integer;
      overload; static; inline;
```

```

class function Decrement(var Target: Integer): Integer;
  overload; static; inline;
class function Add(var Target: Integer;
  Increment: Integer): Integer; overload; static;
class function BitTestAndSet(var Target: Integer;
  BitOffset: TBitOffset): Boolean; static;
class function Exchange(var Target: Integer;
  Value: Integer): Integer; overload; static;
class function CompareExchange(var Target: Integer;
  Value: Integer; Comparand: Integer): Integer;
  overload; static;
end;

```

The goal of all of these functions is to make their operations atomic in term of multi-threading, so that another thread manipulating the same global value at the very same time won't cause a conflict. Using this new coding style, rather than the classic Windows specific:

```

| InterlockedIncrement(aValue);

```

you can now write code that will work on any platform (as in a trivial demo part of the AssortedRtl example):

```

| TInterlocked.Increment(aValue);

```

Waiting, While Keeping Busy

The same SyncObjs unit that hosts the new `TInterlocked` class has a number of new features, mostly tied to the concept of “busy waiting”. But what is “busy waiting”? When a thread has to wait for a condition to occur (possibly tied to an operation by a different thread) it basically has two chances.

- The first option is to give up the control of the CPU so that other threads can finish their tasks faster. When the thread gets back control of the CPU, it will determine if it is its turn to operate, or if it has to wait some more time. This common approach generally optimizes CPU usage, as CPU cycles are used for actual work, but if a thread needs to wait a few cycles only (as the wait is really minimal) giving up control will considerably slow down that thread.
- The second option is to keep control of the CPU, continuously checking the condition and implementing a simple delay in terms of useless operations, so that as soon as the wait is over the thread can act immediately. This approach will generally use extra CPU cycles, and can potentially slow down other threads, but on the other hands lets the given thread resume its work faster. This is the “busy waiting” approach.

Now the relative importance of these two techniques varies significantly with the number of CPU cores available:

- In a single CPU, single core computer, implementing busy-waiting is generally short-sighted, as we need other threads to do something but are using 100% of the CPU, so that the other threads cannot proceed.
- On the other hand, in a quad-core system, while one thread is busy-waiting there are three other threads executing at the same time, and we are “consuming” only 25% of the CPU power.

That's why in recent years the busy-waiting model, and its variations, are gaining importance compared to the past.

In Delphi XE, support for busy-waiting is provided by new classes like `TSpinWait` and `TSpinLock`. These classes have some rather advanced implementations of the concept, something you won't easily clone. As an example, `TSpinWait` (designed after .NET's `System.Threading.SpinnableWait` class) implements an *exponential back-off algorithm* described in the source code documentation as follows:

“If the `CPUCount` > 1, then the first 10 (`YieldThreshold`) spin cycles (calls to `SpinCycle`) will use a base 2 exponentially increasing spin count starting at 4. After 10 cycles, then the behavior reverts to the same behavior as when `CPUCount` = 1.

If the `CPUCount` = 1, then it will sleep (`TThread.Sleep`) 1ms every modulus 20 cycles and sleep 0ms every modulus 5 cycles. All other cycles simply yield (`TThread.Yield`).”

Notice how the code tends to adapt to the number of CPUs, and has behavior that adapts over time, so that it is applicable in a rather large number of situations and on different computers.

Similarly, `TSpinLock` (designed after .NET's `System.Threading.SpinLock` class) implements a simple spin lock, which doesn't block the calling thread (as most other locks) but consumes a few extra CPU cycles with an algorithm similar to the `TSpinWait` class.

As the source code comments,

This is typically faster than fully blocking if the length of time the lock is held is relatively few cycles. In these cases the thread switching overhead will usually far outpace the few cycles burned by simply spin waiting.

Other new synchronization classes, all cross-platform, are:

- `TLightweightEvent`, used in the implementation of the `TInterlocked` class and of the internal `TMonitor` structure
- `TLightweightSemaphore`, which implements a semaphore for atomic operations
- `TCountdownEvent`, a synchronization object signaled when its count is zero (like .NET's `System.Threading.CountdownEvent`).

Using a SpinWait

After this introduction to the available classes, let me show you at least one practical example. I'm going to create an application that uses multiple synchronization mechanisms, so we can compare their behavior. Since the real-world usage conditions can vary enormously, I'm not really trying to prove that one solution is better than another. I'm only focused on showing the difference in a specific scenario.

The `SpinWaiting` project creates four threads (and shows their status with a Progress Bar each). The threads keep executing and doing some slow mathematical computation:

```
procedure TBaseThread.Execute;
var
  I: Integer;
begin
  FreeOnTerminate := True;
  total := 0;
  for I := 1 to 10000 do
    begin
      // do something slow
      total := SomethingsSlow;
      // update progress
      if I mod 200 = 0 then
        Synchronize(procedure ()
          begin
            Progress.Position := I div 100;
          end);
      // wait for other threads
      PostValue (I);
    end;
  end;
```

In the base version (of the `TBaseThread` class) there is no other code and the `PostValue` method is empty and each thread progresses at its own pace. Now suppose that rather than having each thread independent, they have to wait

until all other threads have finished computing a value of the loop before proceeding to the next one.

This is a situation in which we can apply both scenarios, call `sleep` to give up control and implement a spin wait model. This is done by two subclasses of the `TBaseThread` class, with the following `PostValue` methods:

```
procedure TSleepThread.PostValue(step: Integer);
begin
  TInterlocked.Increment(GlobalSteps);
  // wait all other threads by sleeping
  while step > (GlobalSteps div ThreadCount) do
    Sleep (1);
end;

procedure TSpinThread.PostValue(step: Integer);
begin
  TInterlocked.Increment(GlobalSteps);
  // wait all other threads by sleeping
  TSpinwait.SpinUntil(
    function (): Boolean
    begin
      Result := step <= (GlobalSteps div ThreadCount)
    end);
end;
```

The approach is the same, each thread increments a global counter and checks if it is at least 4 times the number of steps (the for loop counter in the `Execute` method). However, calling `Sleep` or `SpinUntil` makes the code behave very differently: in case the single function call takes quite some time (`EachCycle` = 2000), the difference is only noticeable:

```
TSleepThread: 18406 (4 threads)
TSpinThread: 18899 (4 threads)
TSleepThread: 9994 (2 threads)
TSpinThread: 9920 (2 threads)
```

For an average length (`EachCycle` = 400)

```
TSleepThread: 10035 (4 threads)
TSpinThread: 6840 (4 threads)
TSleepThread: 9607 (2 threads)
TSpinThread: 3943 (2 threads)
```

When the single function call is very fast (`EachCycle` = 100), the effect of letting threads give up control becomes critical:

```
TSleepThread: 10031 (4 threads)
TSpinThread: 2305 (4 threads)
TSleepThread: 9564 (2 threads)
TSpinThread: 1146 (2 threads)
```

Again, as I mentioned, I don't want to draw any generic rules from these numbers, but only highlight that there are situations in which using the spin waiting approach delivers a very significant benefit in terms of total time of execution.

A Thread-Safe Queue

Still speaking of threading support, there is also an additional class in the Generics.Collection unit, a thread-safe generic queue, called `TThreadedQueue<T>`. This class is used in the `DSServer` unit to manage callbacks notification queues. Even if this class is relatively simple to use, as there aren't that many thread-safe data structures in Delphi, a small demo will probably be helpful.

The nice feature of this structure is that it is quite simple to use, since it encapsulates a proper synchronization mechanism so you can have multiple threads that push elements into the queue or pop them from it, in a safe way.

For the demo (called `ThreadedQueue`) I've declared a global queue of integers, defined and initialized as follows:

```
const
    MaxQueueElements = 10000;
var
    ThreadedQueue: TThreadedQueue<Integer>;

initialization
    ThreadedQueue := TThreadedQueue<Integer>.
        Create (MaxQueueElements);
```

This data structure is filled by some threads and depleted by others, until it is either full or empty (in which case the given thread terminates):

```
procedure TPushThread.Execute;
begin
    TInterlocked.Increment(nPushThreads);
    try
        while not Terminated do
            begin
                if ThreadedQueue.QueueSize < MaxQueueElements then
                    begin
                        ThreadedQueue.PushItem (Random (100000));
                        Sleep(random (10));
                    end
                else
                    Terminate;
                end;
            finally
                TInterlocked.Decrement(nPushThreads);
            end;
```

```

end;

procedure TPopThread.Execute;
begin
  TInterlocked.Increment(nPopThreads);
  try
    while not Terminated do
      begin
        if ThreadedQueue.QueueSize > 0 then
          begin
            ThreadedQueue.PopItem;
            Sleep(random (10));
          end
        else
          Terminate;
        end;
      finally
        TInterlocked.Decrement(nPopThreads);
      end;
    end;
  end;
end;

```

The program lets you create multiple threads of each kind at the same time, by pressing one of the two buttons (whose captions show the number of threads of each type). The main thread uses a timer to update the overall status:

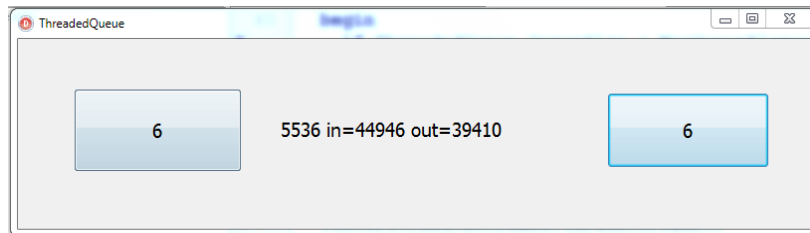
```

procedure TFormThQueue.Timer1Timer(Sender: TObject);
begin
  lblInfo.Caption := Format ('%d in=%d out=%d', [
    ThreadedQueue.QueueSize, ThreadedQueue.TotalItemsPushed,
    ThreadedQueue.TotalItemsPopped]);

  btnPushThread.Caption := IntToStr (nPushThreads);
  btnPopThread.Caption := IntToStr (nPopThreads);
end;

```

The output shows the current number of elements, and how many have been added and removed over time, like in the image below:



Notice that if you create more threads of one kind or the other, the queue will soon become full or empty, and the threads respectively adding or removing elements will be killed, thus restoring a more stable situation... or unbalancing in the opposite direction.

Again, the interesting element of the demo is the simplicity in using a thread-safe queue and the amount information about the data structure you can access. Even better, since this is a generic data structure, you can adapt it to specific data types keeping it compile-time type safe.

Chapter 4: XE Libraries

We have seen that the core run time library (RTL) has a number of significant new features related to regular expressions support, threading support, and more. In Delphi XE there are no large enhancements in the controls of the Visual Component Library (VCL), but a number of significant enhancement in several portions of other libraries.

From the database access layer, to JSON support, to WebBroker, all the way to SOAP support, there are nice features worth considering that I've collected in this chapter. In other words this chapter has no specific focus and will list everything new outside of the RTL, covered in Chapter 3.

Minor VCL Changes

There is really very little to say about the controls and the “visual” portion of the VCL for Delphi XE. There are changes in the source code, but most of them relate to bug fixing and to extensions that are extremely focused in scope.

If the `ColorGrid` component has been promoted from a “sample control” to an actual VCL control, this doesn't really mean a lot to end users. Or if there is a new `ControlStyle` value, called `csPaintBlackOpaqueOnGlass...` it is worth listing them?

For example, I could point out changes like the addition of a new methods in both the standard and 2D canvas classes, called `Arcto` and `AngleArc`, but I really doubt it will raise an eyebrow for most Delphi developers. So I'll stop with this low level approach right away... and mention only the things that I find relevant.

However, since I mentioned this and for pure information purposes, here are the signatures of the two new methods of the `TCustomCanvas` class:

```
procedure Arcto(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);  
procedure AngleArc(X, Y: Integer; Radius: Cardinal;  
    StartAngle, SweepAngle: Single);
```

On the other hand, there is a change worth mentioning in the database portion of the VCL. This is not a new feature, though, but one that was removed, namely the `dbExpress` support for the `BlackFish` SQL database.

The reason is that `BlackFish` SQL itself (a Java or .NET managed database written by Embarcadero) has been also deprecated.

JSON Processing

One area with significant additions in Delphi XE is JSON support²³. While this is mostly tied to the multi-tier `DataSnap` and REST support, mapping objects and tables to JSON is certainly quite useful in itself.

As an example, I'll show you the positive effect of the fixes on a client side application interacting with a Google service, which has nothing to do with `DataSnap` but uses JSON.

JSON Parsing Cleaned Up

A couple of significant changes affect shortcomings of the JSON support in Delphi 2010. Specifically, as I mentioned in my *Delphi 2010 Handbook*, there

²³ The key reference for the JavaScript Object Notation (JSON) is <http://json.org>.

were two problems in parsing JSON in Delphi 2010: the lack of support for Unicode and the problems caused by extra white space.

As I covered in my Delphi 2010 Handbook, once you have a string representing a JSON structure, you can pass it to the `ParseJSONValue` class method of the `TJSONObject`, which returns a `TJSONValue` object. As I wrote:

The `ParseJSONValue` class method doesn't accept a string as parameter, but requires an array of bytes with an ANSI encoding. So we need to take the string and encode it using the `TEncoding` class, that is by calling `TEncoding.ASCII.GetBytes`.

In Delphi XE, the interface is still based on an array of bytes (`TBytes`), but there is an extra parameter indicating whether the string is a UTF8 string. If not, it is considered an ANSI string:

```
class function ParseJSONValue(
  const Data: TBytes; const Offset: Integer;
  IsUTF8: Boolean = True): TJSONValue; overload; static;
```

There are also two easier-to-use functions which let you pass a string directly:

```
class function ParseJSONValue(
  const Data: String): TJSONValue; overload; static;
class function ParseJSONValue(
  const Data: UTF8String): TJSONValue; overload; static;
```

Notice that there is also a `ParseJSONValueUTF8` method, introduced in Delphi XE but marked as *immediately* deprecated in a comment. Now the `JsonTests` example I used in Delphi 2010 Handbook can be improved thanks to the fixes in the parsing code and the new overloaded versions. Originally I had this code:

```
procedure TFormJson.btnParseObjClick(Sender: TObject);
var
  strParam: string;
  jsonObj: TJSONObject;
begin
  strParam := '{"value": 3}'; // space was an issue in 2010
  jsonObj := TJSONObject.ParseJSONValue(
    TEncoding.ASCII.GetBytes(strParam), 0)
    as TJSONObject;

  if Assigned (jsonObj) then
  begin
    // complete object
    Log (jsonObj.ToString);
    // last name/value pair
    Log (jsonObj.Get (jsonObj.Size - 1).ToString);
  end
  else
    Log ('Error in parsing ' + strParam);
  jsonObj.Free;
```

```
| end;
```

This was broken in 2010 because of the extra space in the string, while in XE it works fine. The second advantage is that we can simplify the call to the `ParseJSONValue` method by using one of the new overloaded versions (and skipping the `TEncoding` call):

```
| jsonObj := TJSONObject.ParseJSONValue(strParam)
| as TJSONObject;
```

Google Translate Revisited

As I mentioned, a second enhancement (or bug fix...) relates with the ability to parse Unicode strings, something basically not possible in Delphi 2010. I had this problem in an application I used to parse the JSON returned by the Google Translate REST API²⁴. Again this is a project I covered in my Delphi 2010 Handbook and also in the white paper on REST which I wrote for Embarcadero in late 2009 (but not in the similar one I wrote in early 2011). In short, quoting from the 2010 book:

Another simple and interesting example of a REST API provided by Google is their translation service, called Google Translate REST API:

```
| http://code.google.com/apis/ajaxlanguage/documentation/
```

You can ask for a translation in your Browser with an URL like:

```
| http://ajax.googleapis.com/ajax/services/language/
| translate?v=1.0&q=what%20a%20nice%20day&langpair=en|de
```

The output of this call will be a JSON string:

```
| {
|   "responseData": {
|     "translatedText": "was für ein schöner Tag" },
|   "responseDetails": null,
|   "responseStatus": 200 }
```

To simplify the code for making the REST call and processing the result, I originally wrote a simple support component (`TBabelGoogleRest`). The core code was the following:

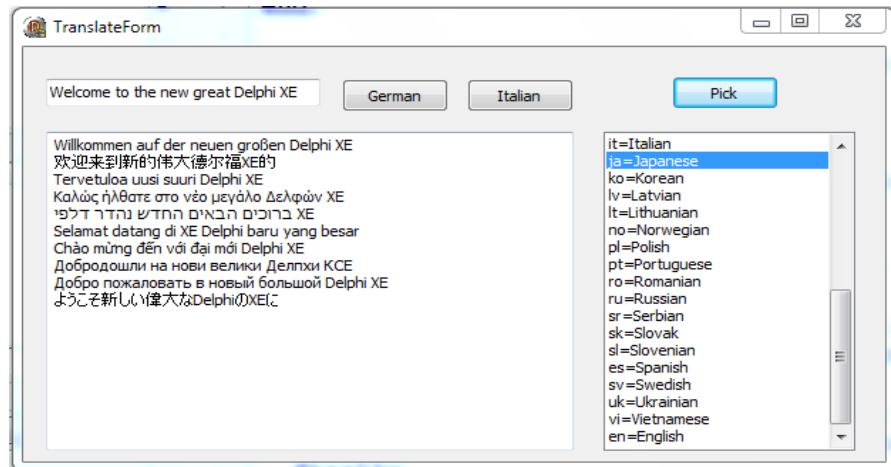
24 Google recently announced it is going to discontinue the Translate API on December 1, 2011. Too bad, as I found it quite handy to use. But probably too many people shared the same feeling and used it putting Google servers under too much stress (with no benefit to the company). Actually, they've later stated that they might leave the service running asking for a fee. At the time of writing, this is still undecided.

```

function TBabelGoogleRest.DoTranslate(strIn: string): string;
var
    strUrl, strResult: string;
    nPosA, nPosB: Integer;
begin
    strUrl := Format (
        'http://ajax.googleapis.com/ajax/services/language/' +
        'translate?v=1.0&q=%s&langpair=%s',
        [TIdUri.ParamsEncode (strIn), FFromLang + '%7C' + FToLang]);
    strResult := Http1.Get(strUrl);
    Result := ResultFromJSON (strResult);
end;

```

Now the problem is that to to implement the `ResultFromJSON` function in the past I had to resort to manual parsing the JSON string returned by the web service, since for many languages this string includes Unicode code points, as you can see in the image below:



This is the output of the program after asking it to translate a string in a few languages based on different alphabets. Now we can extract these Unicode strings with the native parsing, by calling the `ParseJSONValue` method²⁵:

```

function TBabelGoogleRest.ResultFromJSON(
    const strJson: string): string;
var
    jobject, jResponseData: TJSONObject;
begin
    jobject := TJSONObject.ParseJSONValue(strJson) as TJSONObject;
    try
        if not Assigned (jobject) then

```

²⁵ In this code notice that `jResponseData` is a reference to the JSON value inside `jobject` and hence doesn't need freeing.

```

        Exit ('Error parsing ' + strJson);

    // read the value of the first pair of the object
    jResponseData := jobject.Get(0).JsonValue as TJSONObject;

    // get the value of the only element of the responseData
    Result := jResponseData.Get(0).JsonValue.Value;
finally
    jobject.Free;
end;
end;

```

JSON and Tables

Another set of extensions in the Delphi JSON support is available in the new unit `DBXJSONCommon`. Particularly, this unit has methods for mapping `dbExpress` data sets to JSON, part of the `TDBXJSONTools` class. The simplest and highest level method is `TableToJSON`, defined as:

```

class function TableToJSON(const Value: TDBXReader;
    const RowCount: Integer; const IsLocalConnection: Boolean):
    TJSONObject; static;

```

To test its behavior I've added the following event handler to the `JsonTests` example mentioned earlier, asking for the first 10 records returned by a simple select statement:

```

uses
    DBXCommon, DBXJSONCommon;

procedure TFormJson.btnDbxDatasetClick(Sender: TObject);
var
    aCommand: TDBXCommand;
    aReader: TDBXReader;
    jobject: TJSONObject;
begin
    FBConnection.Open;
    aCommand := FBConnection.DBXConnection.CreateCommand;
    try
        aCommand.Text := 'select * from EMPLOYEE';
        aReader := aCommand.ExecuteQuery; // owned by the command
        // copy from the reader to a JSON object
        jobject := TDBXJSONTools.TableToJSON(aReader, 10, True);
        Log (jobject.ToString);
        jobject.Free;
    finally
        aCommand.Free;
    end;
end;

```

The code uses a Firebird connection but you can easily convert to any other dbExpress connection. From this connection I've created a command, and from the command a reader. These classes are part of the low-level DBX interfaces implemented in the DBXCommon unit.

The resulting JSON structure, partially listed below, is quite unusual. It lists some meta data (the *table* pair, with an array of fields) followed by the actual data *arranged by column*, rather than by row. In fact, for each field there is a pair with the field name and an array of values:

```
{
  "table": [
    ["EMP_NO",5,0,0,0,2,2,0,false,false,0,false,false],
    ["FIRST_NAME",1,1,0,0,16,15,0,false,false,0,false,false],
    ["LAST_NAME",1,2,0,0,21,20,0,false,false,0,false,false],
    ... all metadata],
  "EMP_NO": [2,4,5,8,9,11,12,14,15,20],
  "FIRST_NAME": ["Robert", "John", "Robert", "Robert", "Robert",
    "K. J.", "Terri", "Stewart", "Katherine", "Chris"],
  "LAST_NAME": ["Read", "Nelson", "Lamberto", "Johnson", "Forest",
    "Weston", "Changed", "Hall", "Young", "Papadopoulos"],
  ... all data
}
```

This JSON structure is used also when you ask for a dataset to a DataSnap REST server, as highlighted in my white paper on “REST Servers in Delphi XE” published by Embarcadero and referenced in the book Epilogue.

JSON Marshaling

One further extension to JSON support relates with the marshaling mechanism, the ability to convert a Delphi object to a JSON representation (using the new extended RTTI) and re-creating the same object from the JSON string.

The core marshaling mechanism based on private fields and introduced in Delphi 2010 is still in place. What has changed is the way you can customize the mapping of sub-objects. Rather than calling `TJSONMarshal.RegisterConverter` for individual objects or data types, passing anonymous methods for converting the data structures, there is now a mechanism based on attributes.

In fact you can use the `JSONReflect` attribute to customize the marshaling or reflection mechanism for the field to which the attribute is applied. The attribute is defined in the `DBXJSONReflect` unit, which you have to remember to

add to the uses statement or your code won't work, with the compiler emitting only an easy-to-miss and misleading “*unsupported language feature: custom attribute*” warning.

To show this new capability I've heavily modified the JsonMarshal example (yet another example extended from my Delphi 2010 Handbook). First, I've removed the previous code, which used custom converters and reverters. Next I've edited the code of the class with an internal data structure (a TStringList) that I want to be able to save in JSON format and recreate later on. In the definition of this class, listed below, you can see the use of the JSONReflect attribute and its reference to the TStringListInterceptor support class (defined in the DBXJSONReflect unit), a ready-to-use class with the code for the actual data mapping:

```

type
  TDataWithList = class
  private
    theName: String;
    [JSONReflect(ctTypeObject, rtTypeObject,
      TStringListInterceptor, nil, true)]
    theList: TStringList;
  public
    constructor Create (const aName: string); overload;
    constructor Create; overload;
    function ToString: string; override;
    destructor Destroy; override;
  end;

```

With this declaration, we can now convert a string list to a JSON string and later re-create the data structure from it. The generated JSON code will look like the following:

```

{
  "type": "JsonMarshal_MainForm.TDataWithList",
  "id": 1,
  "fields": {
    "theName": "john",
    "theList": {
      "type": "DBXJSONReflect.TSerStringList",
      "id": 2,
      "fields": {
        "FSerStringItemList": [
          {
            "type": "DBXJSONReflect.TSerStringItem",
            "id": 3,
            "fields": {
              "FString": "345",
              "FObject": null
            }
          }
        ]
      }
    }
  },
}

```

```

        {
            "type": "DBXJSONReflect.TSerStringItem",
            "id": 4,
            "fields": {
                "FString": "321",
                "FObject": null
            }
        },
        ...
    ]
}

```

Not the simplest data structure to process in JSON terms, for example by a remote JavaScript application, but quite powerful as it can also stream the objects stored in each item of the string list, beside the strings themselves. The real power of this comes in the ability to read this JSON data (in the same application or even in a different one) and re-create the same exact objects structure, as showcased by the demo program as you press the `btnUnmarshalList` button.

If you are interested in customizing the conversions to and from JSON you can use as an example the `TISODateTimeInterceptor` class available in the *MarshallingUtils* unit of the *ServerProjectWithUserTypes* demo, part of the official set of Delphi XE DataSnap demos. The class is declared²⁶ as:

```

type
  TISODateTimeInterceptor = class(TJSONInterceptor)
  public
    function StringConverter(Data: TObject; Field: string):
      string; override;
    procedure StringReverter(Data: TObject; Field: string;
      Arg: string); override;
  end;

```

This is the last of the several extensions (and corrections) to Delphi's JSON support, which is a significant cornerstone of the multi-tier DataSnap support, not only in its REST variation.

26 The implementation of this class uses code based on the older marshaling customization mechanism and refers to Daniele Teti's blog entry <http://www.danieleteti.it/?p=146>

XML Processing

Another area of Delphi XE libraries that saw some interesting changes is XML support. Rather than in the core DOM mapping or other core XML features, this extension is meant to simplify the use of local customization files in XML format as a replacement of INI files. The new class, in fact, is called `TXmlIniFile` and inherits from `TCustomIniFile`. The class is part of the new `XMLIniFile` unit.

The way you use this class is very similar to the classic INI file counterpart. In other words, you are supposed to use a limited subset of XML if you want to read it with this class. More specifically, while sections are located using the name of the corresponding XML nodes, section item names are indicated by attaching a name attribute to each node. This is likely done for compatibility, so that item names can have spaces, unlike XML node names.

Here is a sample XML file suitable for the `TXmlIniFile` class:

```
<root>
  <section1>
    <entry1 Name="FirstEntry">one</entry1>
    <entry2 Name="Second Entry">two</entry2>
    <item Name="3rdEntry">three</item>
    <item Name="Duplicate">four</item>
    <item Name="Duplicate">five</item>
  </section1>
  <section2>
    <entry1 Name="One">1</entry1>
  </section2>
</root>
```

To initialize an `TXmlIniFile` object you have to pass the root node of an XML DOM to the constructor (this code is part of the `XmlIniTest` application):

```
var
  xmlIni: TXmlIniFile;
  sList: TStringList;
begin
  XMLDocument1.LoadFromFile('..\..\test.xml');
  XMLDocument1.Active := True;
  xmlIni := TXmlIniFile.Create(XMLDocument1.DocumentElement);
```

Now you can use standard INI file processing commands like:

```
sList := TStringList.Create;
xmlIni.ReadSections (sList);
...
xmlIni.ReadSectionValues ('section1', sList);
xmlIni.ReadString('section1', 'FirstEntry', '');
```

Given the previous sample file, these commands will read in string lists or individual strings with this content (notice how the values of the name attributes are used, and the fact they can even be duplicated):

```
// ReadSections
section1
section2

// ReadSectionValues
FirstEntry=one
Second Entry=two
3rdEntry=three
Duplicate=four
Duplicate=five

// ReadString
Specific value:
one
```

As you can read from this structure, you can also create one by writing to the `TXmlIniFile` structure, adding or replacing values:

```
var
  xmlIni: TXmlIniFile;
begin
  XMLDocument1.XML.Text := '<root></root>';
  XMLDocument1.Active := True;
  xmlIni := TXmlIniFile.Create(
    XMLDocument1.DocumentElement);
  try
    xmlIni.WriteString('aSession', 'item1', 'one');
    xmlIni.WriteInteger('aSession', 'item2', 2);
```

This is the XML resulting from the code above, which creates a brand new XML document:

```
<root>
  <aSession>
    <aSession Name="item1">one</aSession>
    <aSession Name="item2">2</aSession>
  </aSession>
</root>
```

Now the format used is very specific, but this class offers probably the easiest way to interact with XML files, and one that is fully compatible with the code used for INI files.

WebBroker and Indy Web Servers

Since the early days of Delphi, at the core of Web development support there has been the WebBroker architecture. This is a set of classes for building Web server extensions and writing Web sites or Web services. In fact, WebBroker is also used by SOAP servers and (in Delphi XE) by DataSnap REST Servers.

The goal of WebBroker is to offer a unified interface for interacting with web requests (`TWebRequest`) and responses (`TWebResponse`), regardless of the actual model used to integrate with the web server. These classes and structural web application class use different derived classes to support one of the available models, and changing from one to the other is only a matter of changing the project file and picking a different set of classes.

These are the available options offered by WebBroker in the past:

- CGI Application, a stand alone program you can invoke to render a dynamic page from almost any Web server.
- ISAPI Library for Microsoft's IIS (originally also usable as an NSAPI DLL for a Netscape web server).
- Web App Debugger application, a standalone executable that can be invoked by the Web App Debugger, a separate tool you run from the Delphi IDE to monitor the traffic between a browser and the web application. This model is well suited for debugging, as you can easily run the program and set breakpoints, unlike an ISAPI library that you have to debug by hooking the debugger to the hosting Web server.
- Apache Module, currently not available in the various Wizards but still supported at the VCL level. The Apache support is comparable to the IIS support, although you might have to change the Apache version number (or magic number) in the source code unit for compatibility with newer versions.

Now, if this has been the situation for several versions of Delphi, the current release shows a significant addition:

- Indy HTTP Server integration, that is your application will be a standalone HTTP server, possibly serving files along with dynamic content. This is also a good model in terms of debugging, comparable to the Web App Debugger.

At first site, deploying a custom Web server might seem a very unusual option, also considering it potentially exposes your application to more attacks and

exploits than one focused exclusively on providing specific pages. Also in terms of performance, it might not be easy to match the speed of a Web server.

There are however several distinct advantages of using this model for a Delphi server side application:

- It is easier to deploy a standalone application than a library or module you have to integrate with an existing web server, on the servers of your customers. You don't have to worry about server versions, configuration, and similar settings. For special purpose and focused servers, the simplicity of this model is certainly quite interesting.
- It lets you update given files when circumstances so dictate. In fact, there is a specific component called `WebFileDispatcher` that you can use to serve files from an Web application of this kind, eventually updating files on request by handling the `BeforeDispatch` and `AfterDispatch` events of this component.
- You can use this approach to cache dynamic data. For example even an HTML file or a JavaScript file or an image might change over time but not for every request. From time to time you'll update the file, but then serve the static file for a few second, minutes, or hours depending on the model.

Finally, consider that even if you deploy your application as a standalone server, this doesn't need to be directly accessible from the outside world, but it might be accessible through a proxy server. As an example, you can have an existing Apache web server and configure it so that all requests for a specific sub-domain or virtual folder or extension type are forwarded to your custom server, running on the same server at a different port (or a different physical or virtual server). This approach would allow you to work with an existing web server that isn't running on Windows, redirecting the HTTP request to your Delphi application running on Windows. In this scenario, your custom server (the Delphi WebBroker application) remains in memory, even in when you need to reboot the Web server facing the end users.

As an example, this is a simple proxy configuration I'm using on Apache²⁷, which serves directly files in the *images*, *css*, *include*, and *files* folders, plus the

27 You can find a very long and detailed description of the Apache Proxy configuration in the server documentation at http://httpd.apache.org/docs/2.2/mod/mod_proxy.html. The features covered in this documentation include all of the basics, but also advanced topics like load balancing among multiple servers.

favicon.ico file, but redirects any other URL (the dynamic pages) to the port 8888 on the same computer:

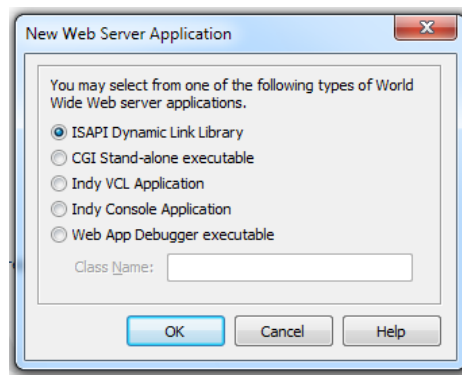
```
<VirtualHost *:80>
  ServerName mysite.marcocantu.com
  DocumentRoot "/web/mysite"
  <Directory /web/mysite/>
    .
  </Directory>

  ProxyPass /images/ !
  ProxyPass /css/ !
  ProxyPass /include/ !
  ProxyPass /files/ !
  ProxyPass /favicon.ico !

  ProxyPass / http://localhost:8888/
  ProxyPassReverse / http://localhost:8888/
</VirtualHost>
```

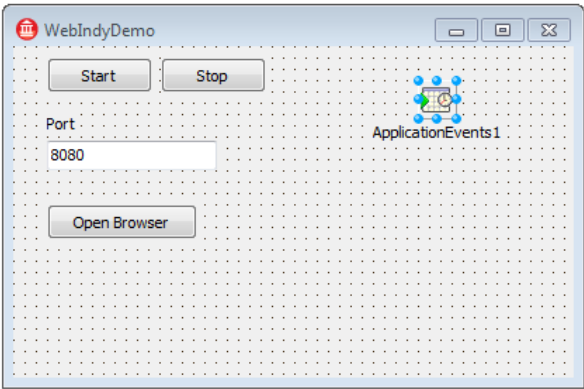
Building a Sample Server

As mentioned, you can build a Web Broker server with the corresponding wizard. The new option is represented by the 3rd and 4th radio buttons, with the difference you'll obtain a VCL or console application. I tend to use VCL applications for debugging purposes, but both make sense... and you might also want to change the structure later to turn it into a service. This is the wizard UI:



If you go with the “Indy VCL Application” the project will have a main form with the role of hosting the application structure and an object of the class `TIdHTTPWebBrokerBridge`, plus a `WebModule` like any other Web Broker application. The `WebModule` is a special purpose data module where you can also define Web Actions, through an editor enabled by right clicking on the

WebModule surface. The main form is not particularly important in terms of its user interface or visual components:



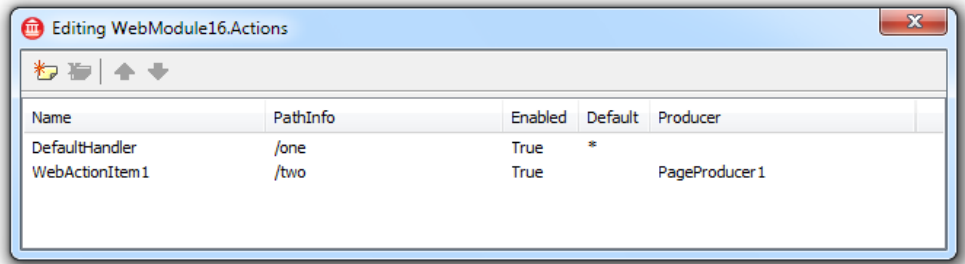
The key role, in fact, is played by a private field declared as:

```
FServer: TIDHTTPWebBrokerBridge;
```

This is the key element of the application, since its class inherits from Indy's TIdCustomHTTPServer class, constructed in the OnCreate event handler. The server is started as you press the corresponding button:

```
procedure TForm15.StartServer;
begin
  if not FServer.Active then
  begin
    FServer.Bindings.Clear;
    FServer.DefaultPort := StrToInt(EditPort.Text);
    FServer.Active := True;
  end;
```

The second unit is the most relevant for the WebBroker application and it is not tied to the specific architecture of the program within this framework. As mentioned earlier, this WebModule can host non-visual components and define a set of actions, each tied to the specific path. For this example I've kept the generated default action, and added a second action tied to a PageProducer:



This is the definition of these actions in the DFM file:

```

Actions = <
  item
    Default = True
    Name = 'DefaultHandler'
    PathInfo = '/one'
    OnAction = WebModule16DefaultHandlerAction
  end
  item
    Name = 'WebActionItem1'
    PathInfo = '/two'
    Producer = PageProducer1
  end>

```

Now the default action has this rather trivial code to generate a very basic page:

```

procedure TWebModule16.WebModule16DefaultHandlerAction (
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<html><heading></body>' +
    'Web Server Application</body></html>';
end;

```

For the second page, I've added a PageProducer tied to a file and responding to the OnHTMLTag event with the code that I'll show you in a while:

```

object PageProducer1: TPageProducer
  HTMLFile = 'templates\Sample.html'
  OnHTMLTag = PageProducer1HTMLTag
end

```

The second component of a page is a WebFileDispatcher, which will be used to return a CSS file and an image, both referenced in the HTML of the page. The WebFileDispatcher uses the standard configuration (basically ready for the DataSnap REST Server model). This component has two collections:

- `webFileExtensions` is a list of mime types returned depending on the file extension (like *'text/css'* for *'css'* or *'text/javascript'* for *'js'*)
- `webDirectories` is a collection of folders (generally sub-folders of the application folder) to be included or excluded when searching for files. I've kept the default settings of including any sub-folder except the *templates* one:

```

  item
    DirectoryAction = dirInclude
    DirectoryMask = '*'
  end
  item
    DirectoryAction = dirExclude
    DirectoryMask = '\templates\*'
  end

```

According to the folder structure behind the component, in fact, the templates folders hosts the HTML file that are not returned directly, but rather through a PageProducer with some server side processing. For the example the `Sample.html` file connected with the second action has one special processing tag for the PageProducer, `<#randomlist>`:

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

<html>

<head>
<title>web Broker Indy Demo</title>
<link rel="stylesheet" type="text/css" href="css/main.css" />
</head>

<body>
<div id="wrapper">
  <div class="banner">
    <a href="/one">First page</a> -
    <a href="/two">Second page</a>
  </div>

  <div class="content-top">
    <h1>web Broker Indy Demo</h1>
  </div>

  <div class="content">

    
    <hr>

    <div>Dynamic list of items:
      <br>
      <div id="list"
        style="padding-left: 40px"><#randomlist></div>
      <br>
    </div>
    <hr>

  </div> <!-- content -->
  <div class="banner">
    <div>
      Demo by <a href="http://www.marcocantu.com">Marco Cantù</a>
    </div>
  </div>
</div> <!-- wrapper -->
</body>
</html>
```

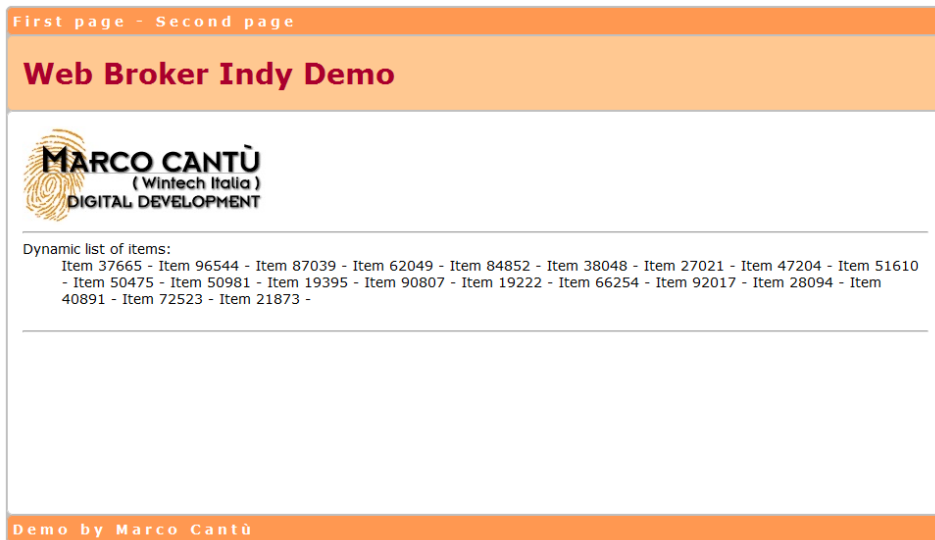
The special tag, marked with the hash (#) is processed by the server and replaced with some dynamic HTML, in this case a random list of items:

```

procedure TwebModule16.PageProducer1HTMLTag(
  Sender: TObject; Tag: TTag;const TagString:
  string; TagParams: TStrings; var ReplaceText: string);
var
  I: Integer;
begin
  if (TagString = 'randomlist') then
  begin
    for I := 1 to 20 do
      ReplaceText := ReplaceText +
        'Item ' + IntToStr (Random (100000)) + ' - ';
    end;
  end;

```

When rendered with the provided CSS file (not described here but available in the source code of the example) and the image (my company logo), both served through the WebFileDispatcher component, the resulting page will look like the following:



This new WebBroker architecture is available also for SOAP applications and for the new DataSnap REST Server Application structure, both of which rely on WebBroker as their foundation.

SOAP Improvements

Beside the integration with the new Indy VCL WebBroker structure, SOAP support has been extended by adding support for HTTPS, the ability to integrate Client Certificates in the HTTPRIO component (the one used for the actual remote calls to Web Services) and the ability to import and use the standard *WS-Security* classes.

The certificate support is provided by the new `ClientCertificate` property of the `HTTPWebNode` property of `THTTPIO`. This lets a developer select an existing certificate at design time, or (to be more precise) the serial number of a certificate. For security reasons, in fact, the certificate itself is not embedded in the program, but has to be distributed along with the application.

Microsoft's Azure Native Support

One of the emerging trends of the current landscape is certainly a large set of technologies collectively indicated by the term “cloud computing”. In general, the term implies using some computing services (storage, CPU time, or even complete virtual machines) offered by a web farm along with Internet connectivity. In the early days, having set up a large Web farm, Amazon started selling some extra storage and bandwidth as a side activity, but this is now turning into a big business for Amazon, Microsoft, Google, and most other major Internet players.

As an example, rather than saving a file on your own web server machine, you can host it on a cloud server, which can make the file available to thousands of concurrent users thanks to higher bandwidth and more balanced servers than you can probably achieve with a single machine or small web farm. Notice that even if the cost is very limited, having your own servers will generally be cheaper... unless you have a usage peak, in which case you'll have to invest a lot of money in infrastructure up front. The nice element of cloud computing is you can use it on-demand and it will easily scale.

Although the cloud computing offerings are growing on a daily basis, the core tenets are a few: host files (images, PDFs, download files, audio and video to be

streamed), host databases (both NOSQL²⁸ database and relational ones), host individual applications or offer complete virtual machines.

How do you interact with these systems? Mostly using the REST protocol over HTTP, so that building Delphi client applications for these services can be rather simple (the complexity is generally in managing authentication and encryption for security reasons). The data is moved as XML or JSON, two formats for which Delphi has internal components and extended third-party support.

In Delphi XE there is native and specific support for three of the Microsoft's Azure cloud services: Queue, Tables, and Blobs. There is also integrated support in the IDE for deployment over Amazon EC2 (Elastic Cloud Computing), a service to which Microsoft is going to offer competition quite soon.

As a practical demo, we are going to show you how to use the Azure service components available in the Delphi XE box for a very simple demo: publish some database data on a hosted database, and create an application that can consume that data. We are going to create two Delphi applications, but an interesting feature is that you can let multiple languages and platforms access the same data.

Delphi XE Components for Azure

Delphi XE includes some components for Azure, but also and moreover a number of lower level classes, which is where the real power is. These classes provide ready-to-use methods for performing the various operations and deal with the complicated security requirements of the Azure platform.

The core non-visual component is the one used to configure the connection to the service, called `TAzureConnectionString`. Although this component has a few quirks in its user interface (it messes up the Object Inspector in the IDE²⁹) it is a handy starting point in which to enter the user name and password of your account.

²⁸ NOSQL (or NoSQL) databases are a new breed of databases that don't follow the relational model. For more information you can refer to <http://nosql-database.org/> or to <http://en.wikipedia.org/wiki/NoSQL>.

²⁹ See for example the bug report <http://qc.embarcadero.com/wc/qcmain.aspx?d=86173>

Other visual components offer direct navigation and management of the respective data structures:

- `TAzureTableManagement` is for NoSQL tables³⁰
- `TAzureBlobManagement` is for binary data (that is, files, images, documents...)
- `TAzureQueueManagement` is for managing queuing services

These management components are very nice to learn about these services and examine their status, but aren't generally a good end user tool, safe for the real power users. In practical terms, you'll generally use the low-level classes behind these components rather than the visual “management” TreeView controls. The three core classes, defined in the `DSAzure` unit, are:

- The `TAzureTableService` class
- The `TAzureBlobService` class
- The `TAzureQueueService` class

The Azure101 Demo

In the `Azure101` project I introduce both the visual components and the low level classes, focusing on the Table and Blob services. The programs loads the account settings from an INI file:

```
procedure TAzure101Form.FormCreate(Sender: TObject);
var
  fIni: TMemIniFile;
begin
  fIni := TMemIniFile.Create(GetHomePath + PathDelim +
    'azure.ini');
  try
    AzureConnectionString1.AccountName :=
      fIni.ReadString('azure', 'AccountName', '');
    AzureConnectionString1.AccountKey :=
      fIni.ReadString('azure', 'AccountKey', '');
  finally
    FreeAndNil (fIni);
  end;
end;
```

³⁰ Microsoft's Azure supports both relational databases (a cloud version of SQL Server) and NoSQL tables. You can read about the differences among the two on <http://www.intertech.com/Blog/post/Windows-Azure-Table-Storage-vs-Windows-SQL-Azure.aspx>. In terms of supporting the Azure SQL Server from Delphi, you can even use `dbExpress` with the proper connection string.

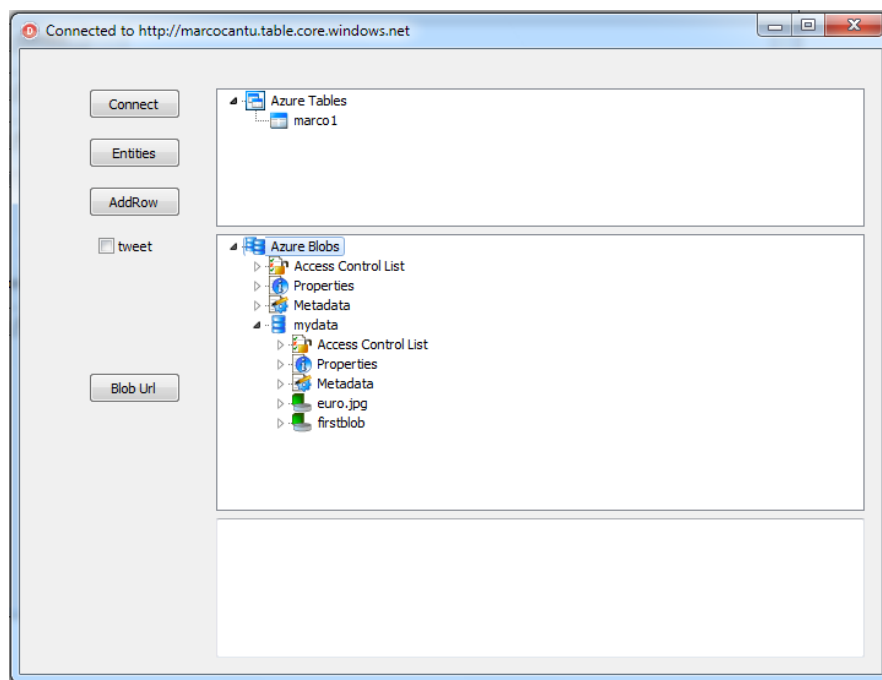
There are two Azure visual controls hooked to this configuration component:

```
object AzureTableManagement1: TAzureTableManagement
    ConnectionInfo = AzureConnectionString1
    Active = False
end
object AzureBlobManagement1: TAzureBlobManagement
    ConnectionInfo = AzureConnectionString1
    Active = False
end
```

As with a database component, you won't generally activate them at design time. Given they might take some time for the setup, you won't probably even turn them on in the OnCreate event handler. In this demo I do it when a button is pressed:

```
procedure TAzure101Form.btnConnectClick(Sender: TObject);
begin
    Caption := 'Connected to ' +
        AzureConnectionString1.TableURL;
    AzureTableManagement1.Active := True;
    AzureBlobManagement1.Active := True;
end;
```

At this point you can use the two controls for browsing the current Tables and Blob containers of your account. You can double click on each of them to open a dialog box with more details. Here is the same information in TreeViews:



The information about tables is limited, and you'll have to double click on a table to open a dialog and see more. For the Blobs, instead, you can drill into the list of elements in each container and also see the list of the properties, the meta data, and the access control list right into the main TreeView. For the Blobs you can also use this view to load new files and set their permissions.

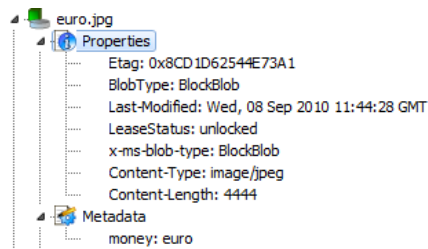
The button on the side will copy to the clipboard the main URL for your azure data, which in my case is:

■ `http://marcocantu.blob.core.windows.net`

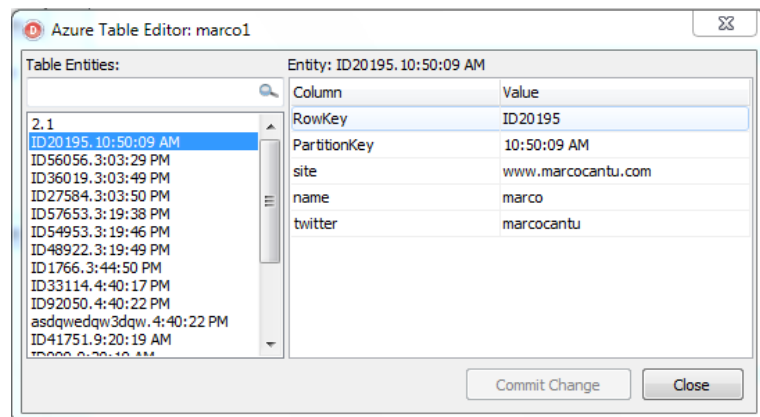
You can combine this URL, the container name, and the object name to figure out its public URL, like (if you look again at the image above):

■ `http://marcocantu.blob.core.windows.net/mydata/euro.jpg`

This can be properly seen in a browser only because I set its *Content-Type* property to a format browsers should recognize (see the details in the image below). Using a URL like the one above anyone can see the image I uploaded with a Delphi program. This is the set of properties for the object:



In terms of the tables, you need to open the details of one to see its structure:



Differently from what you might expect, the table data is not displayed in a grid. In fact, this is not a relational database table, but rather a NoSQL table structure, in which each row can have different fields or, to be more precise, named attributes.

Now what is more interesting, of course, is to interact with this table in your code. For example, you can add a new row to a table using the `InsertEntity` method of the `TAzureTableService` class. The row information is passed in JSON format, so you can use Delphi's `TJSONObject` class (from the `DBXJSON` unit) to define the proper data structure. This must have the *RowKey* and *PartitionKey* attributes, defined in Delphi with two constants called `XML_ROWKEY` and `XML_PARTITION`.

The other element is that some rows can have extra fields, which doesn't mean that other rows have those fields empty, but that they won't have them. In fact there is no schema or definition of a table: the table structure is defined by the contents of the records you add to it. Here is the code used to add a new row to the table:

```
procedure TAzure101Form.btnAddRowClick(Sender: TObject);
var
  TableService: TAzureTableService;
  RowObj: TJSONObject;
begin
  TableService := TAzureTableService.Create(
    AzureConnectionString1);
  try
    // prepare the "row" data
    RowObj := TJSONObject.Create;
    RowObj.AddPair(XML_ROWKEY, TJSONString.Create(
      'ID' + IntToStr (Random (100000))));
    RowObj.AddPair(XML_PARTITION, TJSONString.Create(
      TimeToStr (now)));
    RowObj.AddPair('name', TJSONString.Create('marco'));
    RowObj.AddPair('site', TJSONString.Create(
      'www.marcocantu.com'));
    if CheckBox1.Checked then
      RowObj.AddPair('twitter', TJSONString.Create('marcocantu'));

    // now add the row to the table
    Log (TableService.InsertEntity('marco1', RowObj));
  finally
    TableService.Free;
  end;
end;
```

The other operation done by the program is to query the table for the list of records. This is accomplished with some rather simple code:

```

var
  TableService: TAzureTableService;
begin
  TableService := TAzureTableService.Create(
    AzureConnectionString1);
  try
    Log(TableService.QueryEntities('marco1'));
  
```

The result of this call, however, uses an XML format (basically an ATOM format) that the program logs without parsing it. You'll get something like:

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed>
  <title type="text">marco1</title>
  <id>http://marcocantu.table.core.windows.net/marco1</id>
  <updated>2011-05-27T22:11:48Z</updated>
  <entry>
    <id>http://marcocantu.table.core.windows.net/
      marco1(PartitionKey='1',RowKey='2')</id>
    <title type="text"></title>
    <updated>2011-05-27T22:11:48Z</updated>
    <content type="application/xml">
      <m:properties>
        <d:PartitionKey>1</d:PartitionKey>
        <d:RowKey>2</d:RowKey>
        <d:Timestamp>2010-09-08T11:36:24.4320437Z</d:Timestamp>
        <d:name>myself</d:name>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://marcocantu.table.core.windows.net/
      marco1(PartitionKey='10%3A50%3A09%20AM',RowKey='ID20195')</id>
    <title type="text"></title>
    <updated>2011-05-27T22:11:48Z</updated>
    <content type="application/xml">
      <m:properties>
        <d:PartitionKey>10:50:09 AM</d:PartitionKey>
        <d:RowKey>ID20195</d:RowKey>
        <d:Timestamp>2010-11-18T09:50:16.7935308Z</d:Timestamp>
        <d:name>marco</d:name>
        <d:site>www.marcocantu.com</d:site>
        <d:twitter>marcocantu</d:twitter>
      </m:properties>
    </content>
  </entry>
  ...
</feed>

```

The interesting element of this approach is that once we have created this data structure (or table), it can be used by anyone with the proper permissions and an Internet connection. There is a cost associated, of course, but quite minimal compared to alternative hosting solutions.

The Azure Publish and Consume Demo

The Azure101 demo can get you started using Delphi's Azure support, but it certainly falls short in terms of demonstrating its usefulness. The “management” views are quite nice, but that's hardly the interface you'll want to present to an average end user. That's why I decided to add a second demo, made of two programs:

- The first is a *publishing* application that pulls data from a database and makes it available in an online Azure Table, while the content of graphical fields is published in an Azure Blob.
- The second is a *browsing* application that pulls the data from the Azure cloud storage and shows it to the end user. Notice that this way the end user doesn't require a connection to the database and can benefit from the high bandwidth offered by Azure.

To stick to a classic Delphi theme, the database in question is the Biolife table from the FishFact database. In this case I'm using the ClientDataSet version for simplicity.

Let's look at the CloudPublish application first. The main form of this program has two fields referring to the services we are going to use:

```
private
    TableService: TAzureTableService;
    BlobService: TAzureBlobService;
```

These are initialized in the `OnCreate` event handler after loading the account settings from the INI file, like in the previous example. Before publishing the data, you have to create the corresponding containers.

```
const
    tablename = 'biolife';

procedure TFormCloudPublish.btnCreateTableClick(Sender: TObject);
begin
    TableService.CreateTable (tablename);
    BlobService.CreateContainer(tablename)
end;
```

There is also a second button for emptying and deleting these containers, in case you want to start over.

To publish the data of a single record, we have to create a JSON data structure, include a row key (in this case the record id) and a partition key (in this case a time stamp), and also post a stream with the image using as name the same record id:

```

procedure TFormCloudPublish.btnPostOneClick(Sender: TObject);
var
  strTable: string;
  RowObj: TJSONObject;
  content: TBytes;
begin
  RowObj := TJSONObject.Create;
  RowObj.AddPair (XML_ROWKEY,
    TJSONString.Create (cdsBioSpeciesNo.AsString));
  RowObj.AddPair (XML_PARTITION,
    TJSONString.Create (TimeToStr (now)));

  RowObj.AddPair('category',
    TJSONString.Create(cdsBioCategory.AsString));
  RowObj.AddPair('commonname',
    TJSONString.Create(cdsBioCommon_Name.AsString));
  RowObj.AddPair('speciesname',
    TJSONString.Create(cdsBioSpeciesName.AsString));
  RowObj.AddPair('length',
    TJSONString.Create(cdsBioLengthcm.AsString));
  RowObj.AddPair('notes',
    TJSONString.Create(cdsBioNotes.AsString));
  TableService.InsertEntity(tablename, RowObj);
  BlobService.PutBlockBlob(tablename,
    cdsBioSpeciesNo.AsString + '.jpg',
    JpegContentOf,
    '', nil, 'image/jpeg');
end;

```

The image is published as a JPEG, which means we have to transform it from the original bitmap format to the new format, using a temporary memory stream:

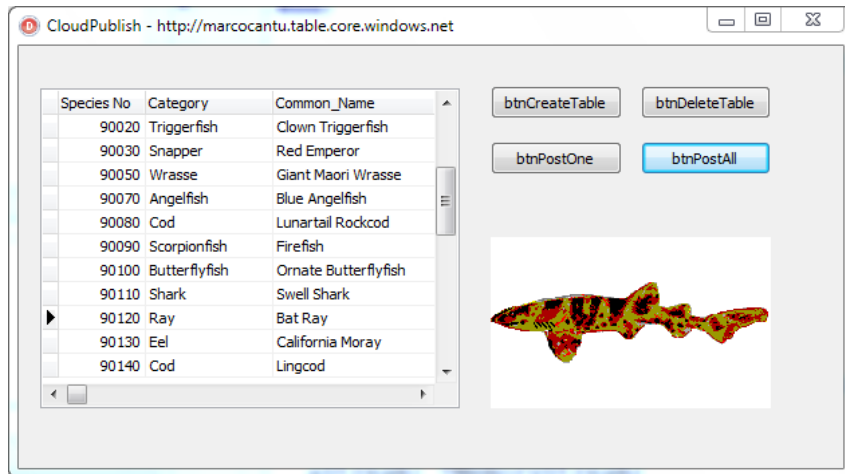
```

function TFormCloudPublish.JpegContentOf: TBytes;
var
  jpgImg: TJPEGImage;
  aStream: TMemoryStream;
begin
  Image1.Picture.Assign(cdsBioGraphic);

  jpgImg := TJPEGImage.Create;
  aStream := TMemoryStream.Create;
  try
    jpgImg.Assign(Image1.Picture.Graphic);
    jpgImg.SaveToStream (aStream);
    aStream.Position := 0;
    SetLength(Result, aStream.Size);
    aStream.ReadBuffer(Result[0], aStream.Size);
  finally
    jpgImg.Free;
    aStream.Free;
  end;
end;

```

Another button let's you scan the database table and post each record. I haven't told much about the user interface of this program because it is very bare bones.



Now let's look at the browsing application, called CloudBrowse. When this form is created, it does an initialization similar to the other Azure programs. For this demo, I'm using a single account, but you can set up Azure to have a separate user with read-only rights.

The first specific operation of the browsing demo is a call to `QueryEntities` to get a list of the records as XML. This return string is processed using an XML-mapper generated interface. However, to parse the actual content of each node we cannot use the interface, as the presence of a nested XML name space causes the interface to fail whilst reading the information (as requests return empty strings in case of a name space mismatch). That's why in the code I end up parsing the innermost XML data structure as a string:

```
procedure TFormCloudBrowse.btnListClick(Sender: TObject);
var
  strXml: string;
  iFeed: IXMLFeedType;
  iEntry: IXMLEntryType;
  I: Integer;
  strContent: string;
  initPos: Integer;
  countPos: Integer;
  strRowKey: string;
  strPartKey: string;
begin
  // ask for the entities
```

```

strXml := TableService.QueryEntities(tablename);
XMLDocument1.LoadFromXML(strXml);
XMLDocument1.Active := True;

iFeed := AzureTableXmlInterfaces.Getfeed(XMLDocument1);
for I := 0 to iFeed.Entry.Count - 1 do
begin
    // fixup (namespace issues workaround)
    strContent := iFeed.Entry[I].Content.XML;

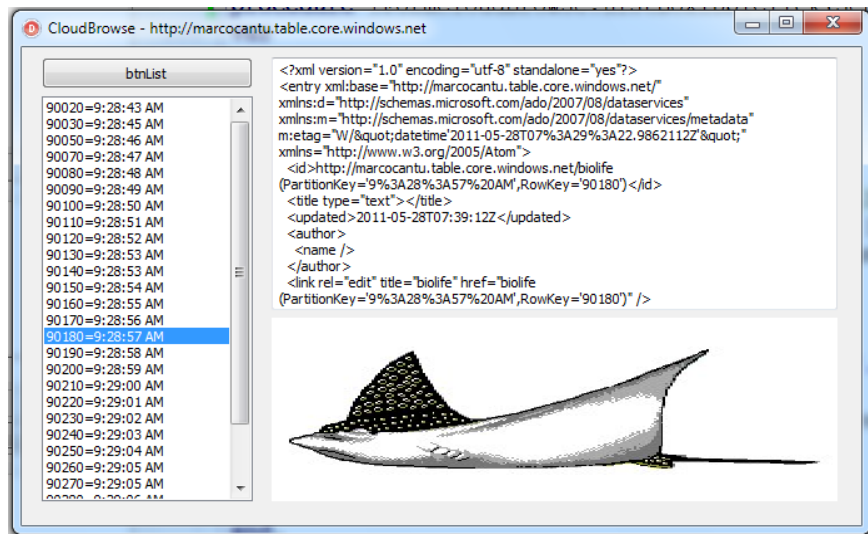
    initPos := Pos (rowkeynode, strContent) + Length (rowkeynode);
    countPos := Pos ('</d:RowKey>', strContent) - initPos;
    strRowKey := copy (strContent, initPos, countPos);

    initPos := Pos (partitionkeynode, strContent) +
        Length (partitionkeynode);
    countPos := Pos ('</d:PartitionKey>', strContent) - initPos;
    strPartKey := copy (strContent, initPos, countPos);

    ListBox1.Items.Add(strRowKey + '=' + strPartKey);
end;
end;

```

The list will show the row and partition keys, used later to retrieve the details as a user double clicks on a list item:



The code used to fetch the record details (which logs the XML, without parsing it) and the image is the following:

```

procedure TFormCloudBrowse.ListBox1DbClick(Sender: TObject);
var
    strRowKey: string;
    strXml: string;

```

```

    memStream: TMemoryStream;
    strPartKey: string;
begin
    strRowKey := ListBox1.Items.Names [
        ListBox1.ItemIndex];
    strPartKey := ListBox1.Items.ValueFromIndex [
        ListBox1.ItemIndex];
    strXml := TableService.QueryEntities(tablename,
        TIDUri.ParamsEncode(strPartKey), strRowKey);
    memo1.Lines.Text := strXml;

    memStream := TMemoryStream.Create;
    try
        BlobService.GetBlob(tablename,
            strRowKey + '.jpg', memStream);
        memStream.Position := 0;
        Image1.Picture.Graphic.LoadFromStream(memStream);
    finally
        memStream.Free;
    end;
end;

```

Again, notice that this *browsing* application doesn't require a connection to the original database or to a custom web server, but relies only on the tables and images provided by Azure cloud storage, which has almost infinite bandwidth.

It is hard to tell how much cloud computing will be a temporary fad and how much of our computing power will reside in web farms owned by a few companies in a few years time, but what is relevant to notice here is that Delphi can be used in this scenario and that there is no “single preferred vendor” (like in other architectures) but rather the ability to use services of multiple companies.

The net result is that in many cases you can extend your native Windows application with a new reach to the Internet, without having to rewrite them as Web applications.

Amazon Web Services and More

AWS (Amazon Web Services, information at <http://aws.amazon.com>) is a much older and currently more complete service than Microsoft's Azure. AWS has services similar to those we have used, like S3 (Simple Storage Service) that corresponds to Azure Blobs and Simple DB that corresponds to Azure Tables.

Although Delphi XE has no support in the box for these services, I have written some component to help use them, available (among other Delphi REST client components) at:

http://code.marcocantu.com/p/delphirestclients/

Still, Delphi XE has native support for a different AWS service, which is a hosting service more than a storage one. Called EC2 (Elastic Cloud Computing), this service let's you create and run an instance of Windows (or Linux) in the Amazon cloud, deploy applications to it, and run your Delphi application in this Windows instance.

Before you can try this step, you need to create an AWS account and use the AWS console to create Windows instance. As you start this instance, it will receive a public DNS name and an IP. You'll generally use the DNS name for configuration (although it will change when you stop and re-start the instance). The configuration for my instance in the AWS console (with the public DNS highlighted) looks like this:

The screenshot shows the AWS Management Console for the 'empty' EC2 instance. The instance is running on the 't1.micro' type with the 'ami-c3e40daa' AMI. The public DNS is highlighted as 'ec2-50-17-154-40.compute-1.amazonaws.com'.

Name	Instance	AMI ID	Root Device	Type	Status	Security Groups	Key Pair Name
☒	empty	i-d33f0db9	ami-c3e40daa	ebs	t1.micro	running	quick-start-1 marcoxe

Source/Dest. Check:

Placement Group:

RAM Disk ID:

Key Pair Name: marcoxe

Monitoring: basic

Elastic IP:

Root Device Type: ebs

Lifecycle: normal

Block Devices: sda1

Public DNS: ec2-50-17-154-40.compute-1.amazonaws.com

Private DNS: ip-10-245-114-107.ec2.internal

Private IP Address: 10.245.114.107

Launch Time: 2011-05-30 12:18 GMT+0200

State Transition Reason:

Virtualization: hvm

Reservation: r-b86a64d3

Platform: windows

Kernel ID:

AMI Launch Index: 0

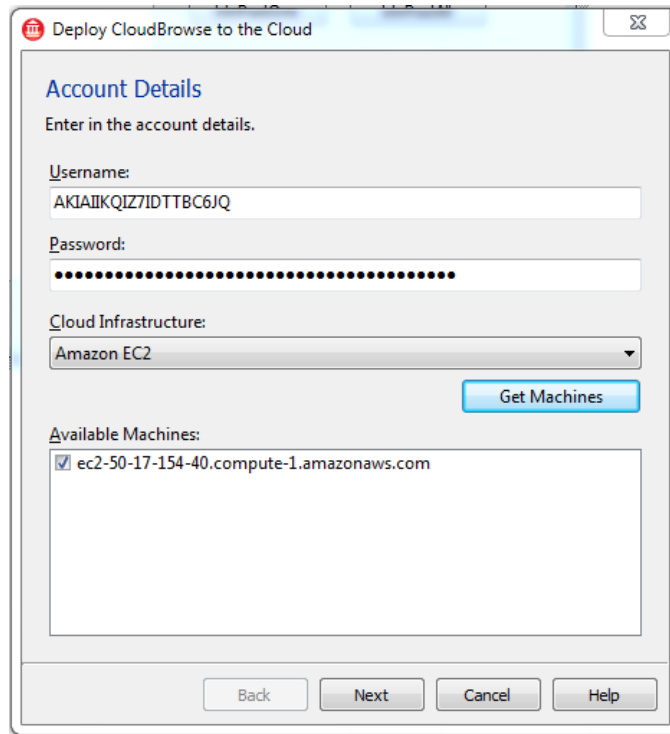
Root Device: sda1

Tenancy: default

Deploy to the Cloud

Deployment to EC2 is integrated in the Delphi IDE. After selecting the Project | Deploy to the Cloud menu item, you have to fill in the dialog box with your public access key in the field called “user name” and your secret access key in the field called “password” (the name of these fields are quite confusing).

Next click the Get Machines button, pick the hosting web farms (in my case *us-east-1*), and you'll see the list of your running instances in that region:



Next, you have to pick one or more files to deploy (notice that the deployment wizard won't automatically pick the executable file of the current project or any other file), and select a target folder for each. The target folder must be an existing folder on the target computer. In the next page, you'll get a chance to review the list of files to deploy, and you can press the Deploy button to perform the actual process.

As you press the Deploy button you'll get another password request. In this case you need to use the Windows user name and password of the EC2 Windows instance we are deploying onto, not the EC2 account password. Consider also that you cannot deploy to the same folder of an existing version if the program is running... and that you can also deploy by simply hooking your local drivers to the instance running in the cloud using the Microsoft's Remote Desktop Connection.

Epilogue

The chapter on libraries ends this book on Delphi XE. As mentioned, I'm not covering the updates in the DataSnap portion of the product, which are significant (but would have further delayed the release of this book). I did cover some of the JSON extensions and Azure support (which is formally part of DataSnap, but in practice completely unrelated).

DataSnap White Papers

One of the reasons I decided not to cover this area is the availability of a couple of in-depth free Embarcadero white papers on the topic, one of which I wrote:

- Bob Swart's white paper titled “Multi-Tier Application Development with DataSnap in RAD Studio XE” is currently hosted at:
| <http://www.embarcadero.com/rad-in-action/datasnap-xe>
- My white paper titled “REST Servers in Delphi XE using DataSnap” (and covering also the development of client jQuery applications for the Delphi REST server) is currently hosted at:
| <http://www.embarcadero.com/rad-in-action/datasnap-rest>

Everything XE Demo

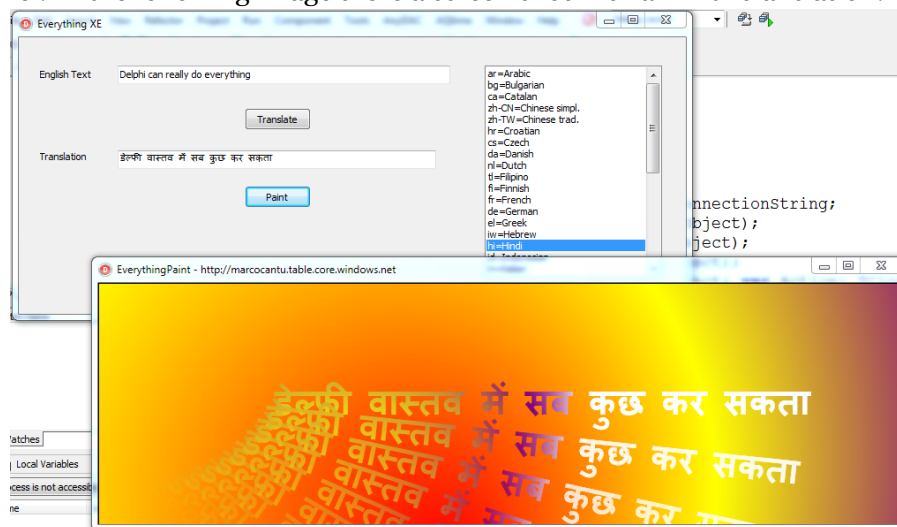
Finally, I want to point out a further demo, included among the book source code in the “epilogue” section, which I wrote to highlight a number of recent features of Delphi in a single, all-encompassing demo. The idea is to create a demo which would be extremely difficult to write in Delphi 2007, thus suggesting that upgrading to a recent version of Delphi really adds a lot to your code.

The goal of the program is to take an input string, translate it to a different language using Google Translate, create a colorful bitmap with the translated string (using Direct2D), transform it to a JPEG, publish it on Azure, and copy the public image URL to the clipboard.

The program uses several recent features of Delphi:

- Unicode support (available since Delphi 2009)
- Anonymous threads (Delphi XE)
- JSON processing (Introduced in Delphi 2010 and updated in Delphi XE)
- Azure components (Delphi XE)
- Direct2D support (Delphi 2010)
- JPEG conversion (Delphi 2009)

Here I don't want to list the source code of the program, as most of the techniques are actually covered elsewhere in the book, but only to underline once more that recent versions of Delphi really brought the product to a new, higher level. In the following image there a screen shot with a Hindi translation:



By right clicking on the secondary form (the one with the image), you can post it on Azure. In my case, this image is uploaded at the URL:

■ <http://marcocantu.blob.core.windows.net/translations/hi.jpg>

You can see a video about this program on YouTube at:

■ <http://www.youtube.com/watch?v=4nZ0jp9NAjs>

More Books Coming

As a conclusion, I want to point out I'm planning to keep covering the features of new versions of Delphi, like in this and past “Delphi Handbook” volumes, but also offer solid foundations for Delphi developers (like in the old “Mastering Delphi” volumes) as well as more feature-oriented summaries. This is a lot of work and it will take some time. Any feedback about this book and the overall plan is appreciated. Stay tuned to my blog and twitter account for up-to-date information.

Index

About Myself.....	7	Borland.....	47
AlignTest example.....	61	Browse Repository.....	42
Amazon Web Services.....	120	Build Events.....	17
Andrea Magni.....	7	Build Groups.....	15
AngleArc.....	92	Build Tools.....	17
Anonymous threads.....	80, 124	busy waiting.....	84
Apache.....	103	Cary Jensen.....	7
Apache Foundation.....	36	CGI.....	102
Apache Module.....	102	Checkout.....	37
AQTime.....	54	Chris Hesik.....	30
ArcTo.....	92	Class Diagrams.....	48
Aspect Oriented Programming.....	65	cloud computing.....	109
AssortedRtl example.....	67pp., 84	CloudBrowse example.....	118
attribute.....		CloudPublish example.....	116
JSONReflect.....	97	CodeRage.....	7
Audits.....	50	CodeSite.....	51, 53
AuditsCLI.exe.....	51	CollabNet.....	36, 39
AWS console.....	121	Command Line.....	50
Azure.....	109p., 124	Command Line Parsing.....	68
Azure SQL Server.....	111	Commit.....	37
Azure101 example.....	111	Compare.....	43
Barry Kelly.....	62	compiled regular expression.....	79
Beyond Compare.....	15, 43p.	Compiler.....	60
BinaryFiles example.....	71	Configuration Manager.....	16
BlackFish SQL.....	92	Content-Type.....	113
blog.....	8	ControlStyle.....	92
Bob Swart.....	123	CreateAnonymousThread.....	80p.

128 - Index

CurrentThread.....	82	Format Project Sources.....	15, 24
CVS.....	35	Formatter Profiles.....	25
Daniele Teti.....	99	Formatter.exe.....	51
DataSnap.....	123	Formatting.....	24
DateUtils unit.....	67	FreeOnTerminate.....	81
DefaultEncoding.....	66	Generate Sequence Diagram.....	49
Delphi Developer Days.....	7	GetCurrentThread.....	81
Delphi wiki.....	20	GetHomePath.....	67
Delphi XE.....	5	git.....	36, 46
Difference Viewer.....	43	Google.....	94
Direct2D.....	124	Google Code.....	38
Distributed version control systems.....	36	Google groups.....	8
DocGenCli.exe.....	51	Google Translate.....	94, 124
Elastic Cloud Computing.....	110, 121	GUID.....	69
email.....	8	hash.....	107
Embarcadero.....	123	Help Insight.....	22p.
EnterMethod.....	53	History View.....	44
Everything XE example.....	124	Holger Flick.....	7
Example.....		HTML.....	107
AlignTest.....	61	HTTPS.....	109
AssortedRtl.....	67pp., 84	IDE Insight.....	13
Azure101.....	111	Incremental Searches.....	20
BinaryFiles.....	71	Indefero.....	39
CloudBrowse.....	118	Indy.....	82, 102
CloudPublish.....	116	Indy HTTP Server.....	29, 102
Everything XE.....	124	INI files.....	100
FinalDemo.....	56	InsertEntity.....	114
InterceptBaseClass.....	64	Install Component wizard.....	28
JsonMarshal.....	98	Integrated Development Environment.....	13
JsonTests.....	93, 96	InterceptBaseClass example.....	64
LoggingDemo.....	52	ISAPI.....	102
ParseJSONValue.....	93	IsMatch.....	75
RegExPrimer.....	76	IsSingleProcessor.....	83
SpinWaiting.....	86	Jan Goyvaerts.....	74, 77
ThreadedQueue.....	88	JavaScript Object Notation.....	92
UmlDemo.....	49	Jeroen Pluimers.....	7
XmlIniTest.....	100	JPEG.....	117, 124
ExitMethod.....	53	JSON.....	92, 94, 96, 114
exponential back-off algorithm.....	85	JSON Marshaling.....	97
Fabrizio Schiavi.....	2	JSONMarshal example.....	98
Facebook page.....	8	JSONReflect.....	97
File Browser.....	14	JsonTests example.....	93, 96
FinalBuilder.....	26, 55	Live Templates.....	20, 22, 52
FinalDemo example.....	56	LoggingDemo example.....	52
Find references.....	18	Marshaling.....	97
FindCmdLineSwitch.....	69	Mastering Delphi.....	7, 125

Match.....	75	RTTI.....	62, 72
Matches.....	75	Rudy Velthius.....	28
Mercurial.....	36, 46	Run Time Library.....	66
Metrics.....	50	Run with Profiler.....	54
MicroFocus.....	47	Run Without Debugging.....	14, 30
Microsoft's Azure.....	109	Search for Usages.....	18p.
Microsoft's Visual Source Safe.....	35	Search Short Cut Keys.....	19
Named Threads.....	30	Select Configurations dialog box.....	16
NameThreadForDebugging.....	30	Sequence Diagrams.....	49
NewGuid.....	69	Show in Explorer.....	14
NOSQL.....	110	Simon J. Stuart.....	7
O'Reilly.....	74	Simple DB.....	120
OnAfter.....	63	Simple Storage Service.....	120
OnBefore.....	63pp.	Sleep.....	83, 87
Ondrej Kelle.....	41	SmartBear Software.....	54
OnExcept.....	64	SOAP.....	109
OnException.....	63	Source Code.....	6
OnHTMLTag.....	106	Source Code Control System.....	34
Open From Version Control.....	41	Source code formatting.....	24
OutputDebugString.....	52	SourceForge.....	38, 46
Package Diagrams.....	47	SpinUntil.....	87
PageProducer.....	105pp.	SpinWaiting example.....	86
ParamCount.....	68	Split.....	75
ParamStr.....	68	SplitString.....	68
ParseJSONValue.....	93	Subversion.....	6, 36
ParseJSONValueUTF8.....	93	Clients.....	39
PartitionKey.....	114	CollabNet.....	39
Paste Special.....	27	In Delphi IDE.....	41
PCRE.....	75	Indefero.....	39
Peter Wood.....	2, 7	Live blame.....	46
Philip Haze.....	75	Servers.....	38
ProcessorCount.....	70, 83	TortoiseSVN.....	39
Profiling.....	54	VisualSVNServer.....	38
Project Manager.....	14p., 41, 43	Surround Windows.....	21
proxy configuration.....	103	Synchronization.....	80
QueryEntities.....	118	TableToJSON.....	96
RAD Studio Version Insight.....	46	TAzureBlobManagement.....	111
Raize Software.....	51	TAzureConnectionString.....	110
Ray Konopka.....	51	TAzureQueueManagement.....	111
RegExPrimer example.....	76	TAzureTableManagement.....	111
Regular Expressions.....	73	TAzureTableService.....	114
Remote Desktop Connection.....	122	TBabelGoogleRest.....	94
Replace.....	75, 78	TBinaryReader.....	70
REST.....	94, 110	TBinaryWriter.....	70
roCompiled.....	79	TCountdownEvent.....	86
RowKey.....	114	TDBXJSONTools.....	96

130 - Index

TEncoding.....	66, 93	unit.....	
TGroup.....	75	DBXCommon.....	97
TGuidHelper.....	69	DBXJSONCommon.....	96
Thread Status.....	30	DBXJSONReflect.....	97
Thread-Safe Queue.....	88	DSAzure.....	111
ThreadedQueue example.....	88	Generics.Collection.....	88
Threading.....	80	IOUtils.....	67
THTTTPRIO.....	109	RegularExpression.....	75
TIdCustomHTTPServer.....	105	RegularExpressionCore.....	75
TIdHTTPWebBrokerBridge.....	104	StrUtils.....	68
time zones.....	67	SyncObjs.....	83p.
TInterlocked.....	83p.	SysUtils.....	67
TISODateTimeInterceptor.....	99	TimeSpan.....	67
TJPEGImage.....	117	XMLIniFile.....	100
TJSONMarshal.....	97	Update.....	37
TJSONObject.....	93	UTF8.....	93
TJSONValue.....	93	Uwe Schuster.....	41, 46
TLightweightEvent.....	86	VER220.....	60
TLightweightSemaphore.....	86	Version Control Systems.....	34
TMatch.....	75	Version Insight.....	46
Together.....	47	Vincent Parrett.....	55
TortoiseSVN.....	39, 43	Virtual Method Interceptor.....	65
TPerlRegEx.....	75	Virtual Methods Interceptors.....	62
TRegEx.....	75, 77pp.	VisualSVNServer.....	38
TSpinLock.....	85	VSoft Technologies Pty Ltd.....	55
TSpinWait.....	85	Web App Debugger.....	29, 102
TStopWatch.....	79	Web Server Application.....	29
TStringList.....	66	Web Sites.....	132
TStringListInterceptor.....	98	WebBroker.....	102
TTextReader.....	70	WebFileDispatcher.....	103, 106, 108
TTextWriter.....	70	White Papers.....	123
TThread.....	70, 80, 83	Wintech Italia Srl.....	2
TThreadedQueue<T>.....	88	Wizards.....	28
TTimeZone.....	67	WriteBOM.....	66
TValue.....	72	WS-Security.....	109
TVirtualMethodInterceptor.....	63	XML.....	100, 119
TWebRequest.....	102	XMLDoc.....	20, 22
TWebResponse.....	102	XmlIniTest example.....	100
Twitter account.....	8	Yield.....	83
TXmlIniFile.....	100	YouTube.....	125
UML Modeling.....	47	\$ALIGN.....	61
UmlDemo example.....	49	\$CODEALIGN.....	62
Unicode.....	124	\$STRINGCHECKS.....	60

Web Sites by Marco Cantù

Here is a partial list of the diverse and somewhat unrelated web sites I manage (or don't manage enough, as some of them are quite old and static)

Main sites are in English language:

- <http://www.marcocantu.com>
- <http://blog.marcocantu.com>
- <http://www.thedelphisearch.com>
- <http://www.wintech-italia.com>
- <http://dev.newswat.com>
- <http://delphi.newswat.com>
- <http://ajax.marcocantu.com>
- <http://www.delphimmentor.com>

Here are other sites in Italian language:

- <http://www.marcocantu.it>
- <http://www.wintech-italia.it>
- <http://shop.wintech-italia.com>
- <http://www.delphiedintorni.it>
- <http://www.piazzacavalli.net>

Personal pages on community sites and micro-blogging sites:

- <http://twitter.com/marcocantu>
- <http://www.facebook.com/marcocantu>
- <http://www.linkedin.com/in/marcocantu>

My online shops (where you can buy books, tools, and services) include:

- <http://sites.fastspring.com/wintechitalia>
- <http://blog.marcocantu.com/bookstore.html>
- <http://shop.wintech-italia.com> (Italian)

