Pro Silverlight 2 in C# 2008

Copyright © 2009 by Matthew MacDonald

ISBN-10 (pbk): 1-59059-949-7

ISBN-13 (pbk): 978-1-59059-949-5

ISBN-13 (electronic): 978-1-4302-0564-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Ami Knox
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreaders: Nancy Bell and Nancy Riddiough
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Introducing Silverlight

In the introduction, you learned about the overall goals and design philosophy that underpin Silverlight. Now, you're ready to get your hands dirty and create your first Silverlight application.

The most practical approach for building Silverlight applications is to use Visual Studio, Microsoft's premiere coding tool. In this chapter, you'll see how to create, compile, and deploy a Silverlight application using Visual Studio 2008. Along the way, you'll get a quick look at how Silverlight controls respond to events, you'll see how Silverlight applications are compiled and packaged for the Web, and you'll consider the two options for hosting Silverlight content: either in an ordinary HTML web page or in an ASP.NET web form.

Silverlight and Visual Studio

Although it's technically possible to create the files you need for a Silverlight application by hand, professional developers always use a development tool. If you're a graphic designer, that tool is likely to be Microsoft Expression Blend 2.5, which provides a full complement of features for designing visually rich user interfaces. If you're a developer, you'll probably use Visual Studio 2008, which includes well-rounded tools for coding, testing, and debugging.

Because both tools are equally at home with the Silverlight application model, you can easily create a workflow that incorporates both of them. For example, a developer could create a basic user interface with Visual Studio and then hand it off to a crack design team, who would polish it up with custom graphics in Expression Blend. When the facelift is finished, they would deliver the project back to the developer, who could then continue writing and refining its code in Visual Studio.

Note In this book, you'll focus your attention on Visual Studio. But before you can use Visual Studio 2008 to create Silverlight applications, you need to install a set of extensions for Silverlight development. For complete instructions, see the introduction of this book or the readme.txt file included with the sample code.

Understanding Silverlight Websites

There are two types of Silverlight websites that you can create in Visual Studio:

- **An ordinary website with HTML pages.** In this case, the entry point to your Silverlight application is a basic HTML file that includes a Silverlight content region.
- **ASP.NET website.** In this case, Visual Studio creates two projects—one to contain the Silverlight application files, and one to hold the server-side ASP.NET website that will be deployed alongside your Silverlight files. The entry point to your Silverlight application can be an ordinary HTML file, or it can be an ASP.NET web form that includes server-generated content.

So which approach is best? No matter which option you choose, your Silverlight application will run the same way—the client browser will receive an HTML document, that HTML document will include a Silverlight content region, and the Silverlight code will run on the local computer, *not* the web server. However, the ASP.NET web approach makes it easier to mix ASP.NET and Silverlight content. This is usually a better approach in the following cases:

- You want to create a website that contains both ASP.NET web pages and Silverlight-enhanced pages.
- You want to generate Silverlight content indirectly, using ASP.NET web controls.
- You want to create a Silverlight application that calls a web service, and you want to design the web service at the same time (and deploy it to the same web server).

On the other hand, if you don't need to write any server-side code, there's little point in creating a full-fledged ASP.NET website. Many of the Silverlight applications you'll see in this book use basic HTML-only websites. The examples only include ASP.NET websites when they need specific server-side features. For example, the examples in Chapter 14 use an ASP.NET website that includes a web service. This web service allows the Silverlight application to retrieve data from a database on the web server, a feat that would be impossible without server-side code.

ADDING SILVERLIGHT CONTENT TO AN EXISTING WEBSITE

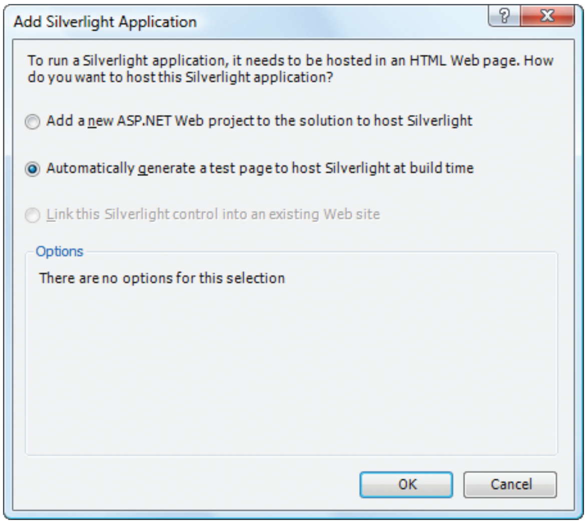
A key point to keep in mind when considering the Silverlight development model is that in many cases you'll use Silverlight to *augment* the existing content of your website, which will still include generous amounts of HTML, CSS, and JavaScript. For example, you might add a Silverlight content region that shows an advertisement or allows an enhanced experience for a portion of a website (such as playing a game, completing a survey, interacting with a product, taking a virtual tour, and so on). You may use Silverlight-enhanced pages to present content that's already available in your website in a more engaging way, or to provide a value-added feature for users who have the Silverlight plug-in.

Of course, it's also possible to create a Silverlight-only website, which is a somewhat more daring approach. The key drawback is that Silverlight is still relatively new, and it doesn't support legacy clients (most notably, it has no support for users of Windows ME and Windows 98, and Internet Explorer–only support for Windows 2000). As a result, Silverlight doesn't have nearly the same reach as ordinary HTML. Many businesses that are adopting Silverlight are using it to distinguish themselves from other online competitors with cutting-edge content, but they aren't abandoning their traditional websites.

Creating a Stand-Alone Silverlight Project

The easiest way to start using Silverlight is to create an ordinary website with HTML pages and no server-side code. Here's how:

1. Select File ➤ New ➤ Project in Visual Studio, choose the Visual C# group of project types, and select the Silverlight Application template. As usual, you need to pick a project name and a location on your hard drive before clicking OK to create the project.
2. At this point, Visual Studio will prompt you to choose whether you want to create a full-fledged ASP.NET website that can run server-side code or an ordinary website with HTML pages (see Figure 1-1). For now, choose the second option ("Automatically generates a test page") to create an ordinary website and click OK.


Add Silverlight Application

To run a Silverlight application, it needs to be hosted in an HTML Web page. How do you want to host this Silverlight application?

○ Add a new ASP.NET Web project to the solution to host Silverlight
[✓ ● Automatically generate a test page to host Silverlight at build time]
○ Link this Silverlight control into an existing Web site

Options
There are no options for this selection

Figure 1-1. *Choosing the type of website*

Every Silverlight project starts with a small set of essential files, as shown in Figure 1-2. All the files that end with the extension .xaml use a flexible markup standard called XAML, which you'll dissect in the next chapter. All the files that end with the extension .cs hold the C# source code that powers your application.

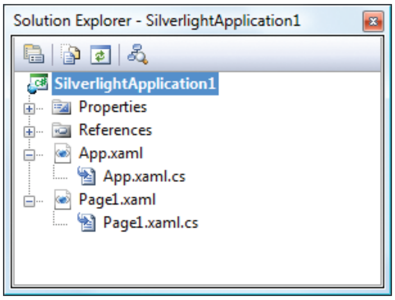


Figure 1-2. *A Silverlight project*

Here's a rundown of the files shown in Figure 1-2:

- **App.xaml and App.xaml.cs.** These files allow you to configure your Silverlight application. They allow you to define resources that will be made available to all the pages in your application (see Chapter 2), and they allow you react to application events such as startup, shutdown, and error conditions (see Chapter 6). In a newly generated project, the startup code in the App.xaml.cs file specifies that your application should begin by showing Page.xaml.

- **Page.xaml.** This file defines the user interface (the collection of controls, images, and text) that will be shown for your first page. Technically, Silverlight pages are *user controls*—custom classes that derive from UserControl. A Silverlight application can contain as many pages as you need—to add more, simply choose Project ➤ Add New Item, pick the Silverlight User Control template, choose a file name, and click Add.

- **Page.xaml.cs.** This file includes the code that underpins your first page, including the event handlers that react to user actions.

Along with these four essential files, there are a few more ingredients that you'll only find if you dig around. Under the Properties node in the Solution Explorer, you'll find a file named AppManifest.xml, which lists the assemblies that your application uses. You'll also find a file named AssemblyInfo.cs, which contains information about your project (such as its name, version, and publisher) that's embedded into your Silverlight assembly when it's compiled. Neither of these files should be edited by hand—instead, they're modified by Visual Studio when you add references or set projects properties.

Lastly, the gateway to your Silverlight application is an automatically generated but hidden HTML file named TestPage.html (see Figure 1-3). To see this file, make sure you've compiled your application at least once. Then, click the Show All Files button at the top of the Solution Explorer, and expand the Bin\Debug folder (which is where your application is compiled). You'll take a closer look at the content of the TestPage.html file a bit later in this chapter.

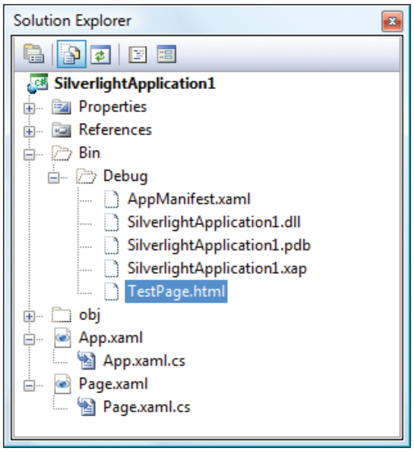
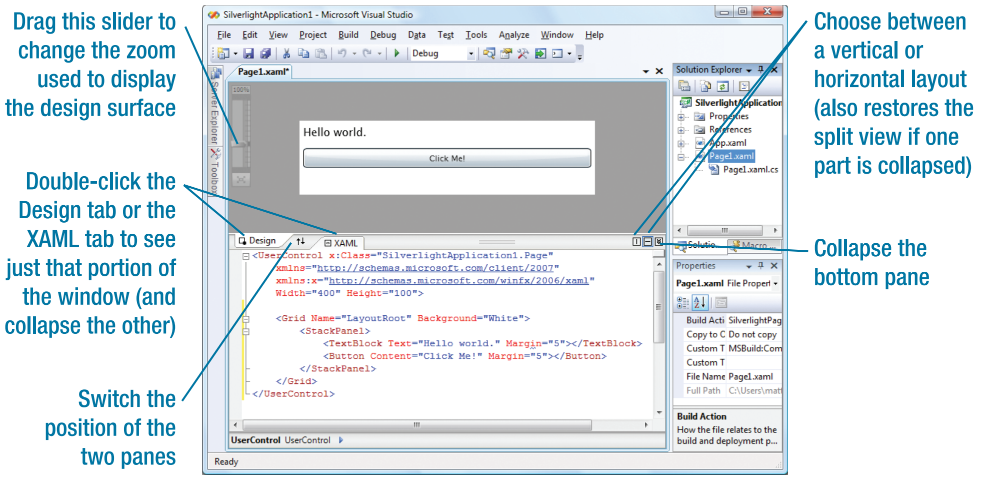


Figure 1-3. *The HTML test page*

Creating a Simple Silverlight Page

As you've already learned, every Silverlight page includes a markup portion that defines the visual appearance (the XAML file) and a source code file that contains event handlers. To customize your first Silverlight application, you simply need to open the Page.xaml file and begin adding markup.

Visual Studio gives you two ways to look at every XAML file—as a visual preview (known as the *design surface*) or the underlying markup (known as the *source view*). By default, Visual Studio shows both parts, stacked one on the other. Figure 1-4 shows this view and points out the buttons you can use to change your vantage point.


Drag this slider to change the zoom used to display the design surface
Double-click the Design tab or the XAML tab to see just that portion of the window (and collapse the other)
Switch the position of the two panes
Choose between a vertical or horizontal layout (also restores the split view if one part is collapsed)
Collapse the bottom pane
SilverlightApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Data Test Tools Analyze Window Help
Debug
Page1.xaml*
Hello world.
Click Me!
Design
XAML
Solution Explorer
SilverlightApplication
Properties
References
App.xaml
Page1.xaml
Page1.xaml.cs
Properties
Page1.xaml File Properties
Build Action
How the file relates to the build and deployment p...
UserControl UserControl
Ready

```
<UserControl x:Class="SilverlightApplication1.Page"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="100">

    <Grid Name="LayoutRoot" Background="White">
        <StackPanel>
            <TextBlock Text="Hello world." Margin="5"></TextBlock>
            <Button Content="Click Me!" Margin="5"></Button>
        </StackPanel>
    </Grid>
</UserControl>
```

Figure 1-4. *Viewing XAML pages*

As you've no doubt guessed, you can start designing your XAML page by dragging controls from the Toolbox and dropping them onto the design surface. However, this convenience won't save you from learning the full intricacies of XAML. In order to organize your elements into the right layout containers, change their properties, wire up event handlers, and use Silverlight features like animation, styles, templates, and data binding, you'll need to edit the XAML markup by hand.

To get started, you can try creating the page shown here, which defines a block of text and a button. The portions in bold have been added to the basic page template that Visual Studio generated when you created the project.

```
<UserControl x:Class="SilverlightApplication1.Page"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="100">

    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel>
            <TextBlock x:Name="lblMessage" Text="Hello world."
             Margin="5"></TextBlock>
            <Button x:Name="cmdClickMe" Content="Click Me!" Margin="5"></Button>
        </StackPanel>
    </Grid>
</UserControl>
```

This creates a page that has a stacked arrangement of two elements. On the top is a block of text with a simple message. Underneath it is a button.

■**Note** In Silverlight terminology, each graphical widget that meets these criteria (appears in a window and is represented by a .NET class) is called an *element*. The term *control* is generally reserved for elements that receive focus and allow user interaction. For example, a TextBox is a control, but the TextBlock is not.

Adding Event Handling Code

You attach event handlers to the elements in your page using attributes, which is the same approach that developers take in WPF, ASP.NET, and JavaScript. For example, the Button element exposes an event named Click that fires when the button is triggered with the mouse or keyboard. To react to this event, you add the Click attribute to the Button element, and set it to the name of a method in your code:

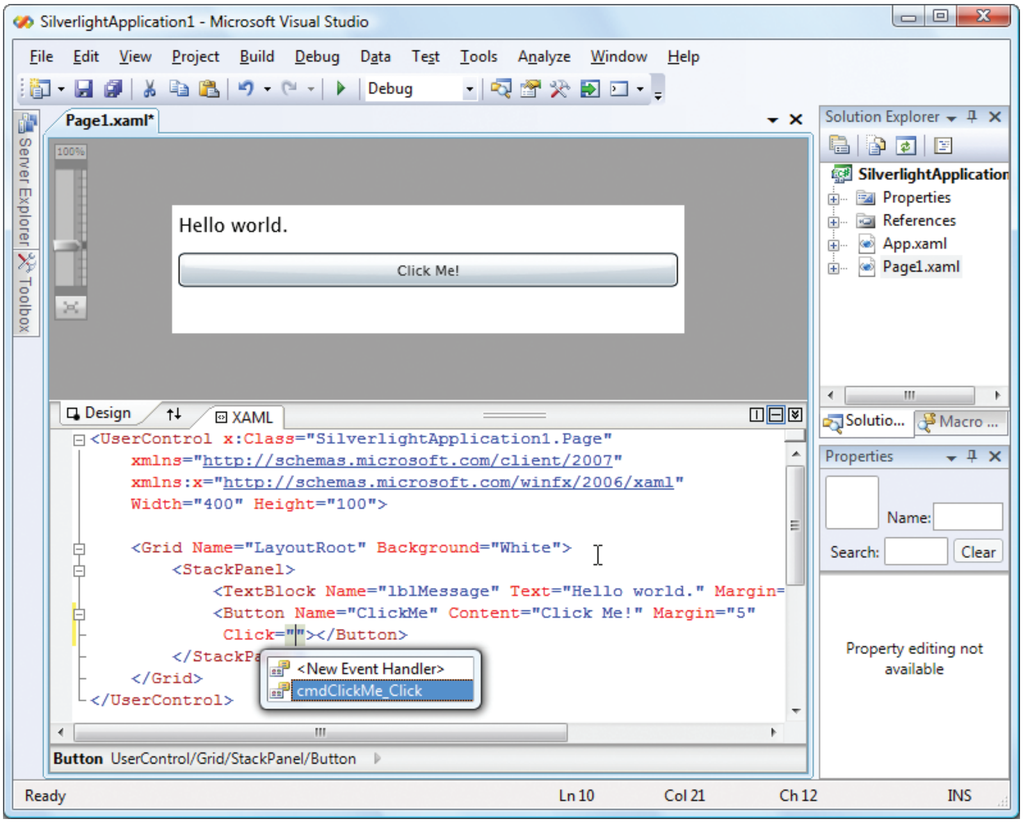
```
<Button x:Name="cmdClickMe" Click="cmdClickMe_Click" Content="Click Me!"
 Margin="5"></Button>
```

■**Tip** Although it's not required, it's a common convention to name event handler methods in the form ElementName_EventName. If the element doesn't have a defined name (presumably because you don't need to interact with it in any other place in your code), consider using the name it *would* have.

This example assumes that you've created an event handling method named cmdClickMe_Click. Here's what it looks like in the Page.xaml.cs file:

```
private void cmdClickMe_Click(object sender, RoutedEventArgs e)
{
    lblMessage.Text = "Goodbye, cruel world.";
}
```

You can't coax Visual Studio into creating an event handler by double-clicking an element or using the Properties window (as you can in other types of projects). However, once you've added the event handler, you can use IntelliSense to quickly assign it to the right event. Begin by typing in the attribute name, followed by the equal sign. At this point, Visual Studio will pop up a menu that lists all the methods that have the right syntax to handle this event, and currently exist in your code behind class, as shown in Figure 1-5. Simply choose the right event handling method.



```
<UserControl x:Class="SilverlightApplication1.Page"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="100">

    <Grid Name="LayoutRoot" Background="White">
        <StackPanel>
            <TextBlock Name="lblMessage" Text="Hello world." Margin=
            <Button Name="ClickMe" Content="Click Me!" Margin="5"
             Click=""></Button>
        </StackPa
    </Grid>
</UserControl>
```

Figure 1-5. *Attaching an event handler*

It's possible to use Visual Studio to create and assign an event handler in one step by adding an event attribute and choosing the <New Event Handler> option in the menu.

Tip To jump quickly from the XAML to your event handling code, right-click the appropriate event attribute in your markup and choose Navigate to Event Handler.

You can also connect an event with code. The place to do it is the constructor for your page, after the call to InitializeComponent(), which initializes all your controls. Here's the code equivalent of the XAML markup shown previously:

```
public Page()
{
    InitializeComponent();
    cmdClickMe.Click += cmdClickMe_Click;
}
```

The code approach is useful if you need to dynamically create a control and attach an event handler at some point during the lifetime of your window. By comparison, the events you hook up in XAML are always attached when the window object is first instantiated. The code approach also allows you to keep your XAML simpler and more streamlined, which is perfect if you plan to share it with non-programmers, such as a design artist. The drawback is a significant amount of boilerplate code that will clutter up your code files.

If you want to detach an event handler, code is your only option. You can use the -= operator, as shown here:

```
cmdClickMe.Click -= cmdClickMe_Click;
```

It is technically possible to connect the same event handler to the same event more than once. This is almost always the result of a coding mistake. (In this case, the event handler will be triggered multiple times.) If you attempt to remove an event handler that's been connected twice, the event will still trigger the event handler, but just once.

Browsing the Silverlight Class Libraries

In order to write practical code, you need to know quite a bit about the classes you have to work with. That means acquiring a thorough knowledge of the core class libraries that ship with Silverlight.

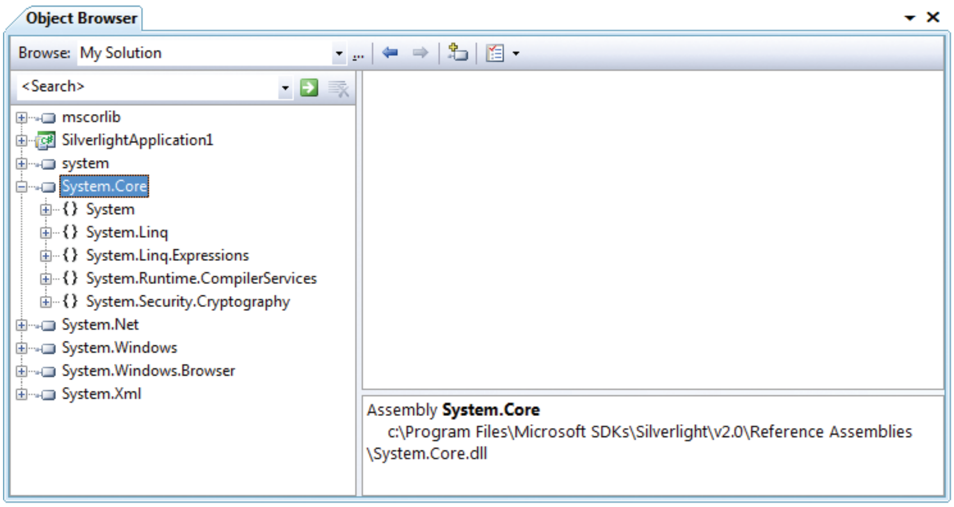
Silverlight includes a subset of the classes from the full .NET Framework. Although it would be impossible to cram the entire .NET Framework into Silverlight—after all, it's a 4MB download that needs to support a variety of browsers and operating systems—Silverlight includes a remarkable amount of functionality.

The Silverlight version of the .NET Framework is simplified in two ways. First, it doesn't provide the sheer number of types you'll find in the full .NET Framework. Second, the classes that it does include often don't provide the full complement of constructors, methods, properties, and events. Instead, Silverlight keeps only the most practical members of the most important classes, which leaves it with enough functionality to create surprisingly compelling code.

■**Note** The Silverlight classes are designed to have public interfaces that resemble their full-fledged counterparts in the .NET Framework. However, the actual plumbing of these classes is quite different. All the Silverlight classes have been rewritten from the ground up to be as streamlined and efficient as possible.

Before you start doing any serious Silverlight programming, you might like to browse the Silverlight version of the .NET Framework. One way to do so is to open a Silverlight project, and then show the Object Browser in Visual Studio (choose View ➤ Object Browser). Along with the assembly for the code in your project, you'll see the following Silverlight assemblies (shown in Figure 1-6):

- **mscorlib.dll.** This assembly is the Silverlight equivalent of the mscorlib.dll assembly that includes the most fundamental parts of the .NET Framework. The Silverlight version includes core data types, exceptions, and interfaces in the System namespace; ordinary and generic collections; file management classes; and support for globalization, reflection, resources, debugging, and multithreading.
- **System.dll.** This assembly contains additional generic collections, classes for dealing with URIs, and classes for dealing with regular expressions.
- **System.Core.dll.** This assembly contains support for LINQ. The name of the assembly matches the full .NET Framework, which implements new .NET 3.5 features in an assembly named System.Core.dll.
- **System.Net.dll.** This assembly contains classes that support networking, allowing you to download web pages and create socket-based connections.
- **System.Windows.dll.** This assembly includes many of the classes for building Silverlight user interfaces, including basic elements, shapes and brushes, classes that support animation and data binding, and a version of the OpenFileDialog that works with isolated storage.
- **System.Windows.Browser.dll.** This assembly contains classes for interacting with HTML elements.
- **System.Xml.dll.** This assembly includes the bare minimum classes you need for XML processing: XmlReader and XmlWriter.



```
mscorlib
SilverlightApplication1
system
System.Core
  { } System
  { } System.Linq
  { } System.Linq.Expressions
  { } System.Runtime.CompilerServices
  { } System.Security.Cryptography
System.Net
System.Windows
System.Windows.Browser
System.Xml
```

Assembly **System.Core**
c:\Program Files\Microsoft SDKs\Silverlight\v2.0\Reference Assemblies
\System.Core.dll

Figure 1-6. *Silverlight assemblies in the Object Browser*

Note Some of the members in the Silverlight assemblies are only available to .NET Framework code, and aren't callable from your code. These members are marked with the SecurityCritical attribute. However, this attribute does not appear in the Object Browser, so you won't be able to determine whether a specific feature is usable in a Silverlight application until you try to use it. (If you attempt to use a member that has the SecurityCritical attribute, you'll get a SecurityException.) For example, Silverlight applications are only allowed to access the file system through the isolated storage API or the OpenFileDialog class. For that reason, the constructor for the FileStream class is decorated with the SecurityCritical attribute.

SILVERLIGHT'S ADD-ON ASSEMBLIES

The architects of Silverlight have set out to keep the core framework as small as possible. This design makes the initial Silverlight plug-in small to download and quick to install—an obvious selling point to web surfers everywhere.

To achieve this lean-and-mean goal, the Silverlight designers have removed some functionality from the core Silverlight runtime and placed it in separate add-on assemblies. These assemblies are still considered to be part of the Silverlight platform, but if you want to use them, you'll need to package them with your application. This represents an obvious trade-off, because it will increase the download size of your application. (The effect is mitigated by Silverlight's built-in compression, which you'll learn about later in this chapter.)

You'll learn about Silverlight's add-on assemblies throughout this book. Two commonly used ones are:

- **System.Windows.Controls.dll.** This assembly contains a few new controls, including the Calendar, DatePicker, TabControl, and GridSplitter.
- **System.Windows.Controls.Data.dll.** This assembly has Silverlight's new built-from-scratch DataGrid.

Both of these assemblies add new controls to your Silverlight toolkit. In the near future, Microsoft plans to make many more add-on controls available. Eventually, the number of add-on controls will dwarf the number of core controls.

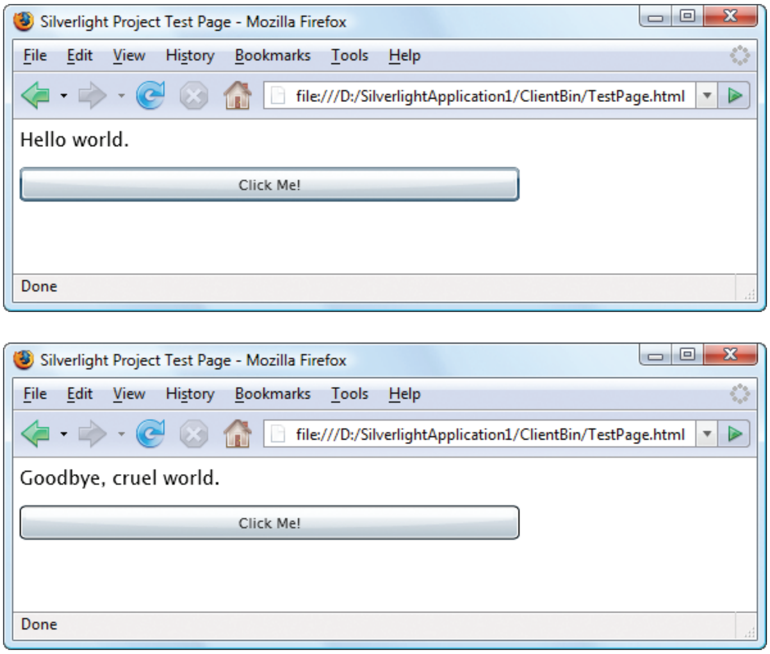
When you drag a control from an add-on assembly onto a Silverlight page, Visual Studio automatically adds the assembly reference you need. If you select that reference and look in the Properties window, you'll see that the Copy Local property is set to true, which is different from the other assemblies that make up the core Silverlight runtime. As a result, when you compile your application, the assembly will be embedded in the final package. Visual Studio is intelligent enough to recognize assemblies that aren't a part of the core Silverlight runtime—even if you add them by hand, it automatically sets Copy Local to true.

Testing a Silverlight Application

You now have enough to test your Silverlight project. When you run a Silverlight application, Visual Studio launches your default web browser and navigates to the hidden browser test page, named TestPage.html. The test page creates a new Silverlight control and initializes it using the markup in Page.xaml.

■ **Note** Visual Studio sets TestPage.html to be the start page for your project. As a result, when you launch your project, this page will be loaded in the browser. You can choose a different start page by right-clicking an HTML file in the Solution Explorer and choosing Set As Start Page.

Figure 1-7 shows the previous example at work. When you click the button, the event handling code runs and the text changes. This process happens entirely on the client—there is no need to contact the server or post back the page, as there is in a server-side programming framework like ASP.NET. All the Silverlight code is executed on the client side by the scaled-down version of .NET that's embedded in the Silverlight plug-in.


Silverlight Project Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///D:/SilverlightApplication1/ClientBin/TestPage.html
Click Me!
Done
Silverlight Project Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///D:/SilverlightApplication1/ClientBin/TestPage.html
Click Me!
Done

```
Hello world.
```

```
Goodbye, cruel world.
```

Figure 1-7. *Running a Silverlight application (in Firefox)*

If you're hosting your host Silverlight content in an ordinary website (with no server-side ASP.NET), Visual Studio won't use its integrated web server during the testing process. Instead, it simply opens the HTML test page directly from the file system. (You can see this in the address bar in Figure 1-7.)

In some situations, this behavior could cause discrepancies between your test environment and your deployed environment, which will use a full-fledged web server that serves pages over HTTP. The most obvious difference is the security context—in other words, you could configure your web browser to allow local web pages to perform actions that remote web content can't. In practice, this isn't often a problem, because Silverlight always executes in a stripped-down security context, and doesn't include any extra functionality for trusted locations. This simplifies the Silverlight development model, and ensures that features won't work in certain environments and break in others. However, when production testing a Silverlight application, it's best to create an ASP.NET test website (as described at the end of this chapter) or—even better—deploy your Silverlight application to a test web server.

Silverlight Compilation and Deployment

Now that you've seen how to create a basic Silverlight project, add a page with elements and code, and run your application, it's time to dig a bit deeper. In this section, you'll see how your Silverlight is transformed from a collection of XAML files and source code into a rich browser-based application.

Compiling a Silverlight Application

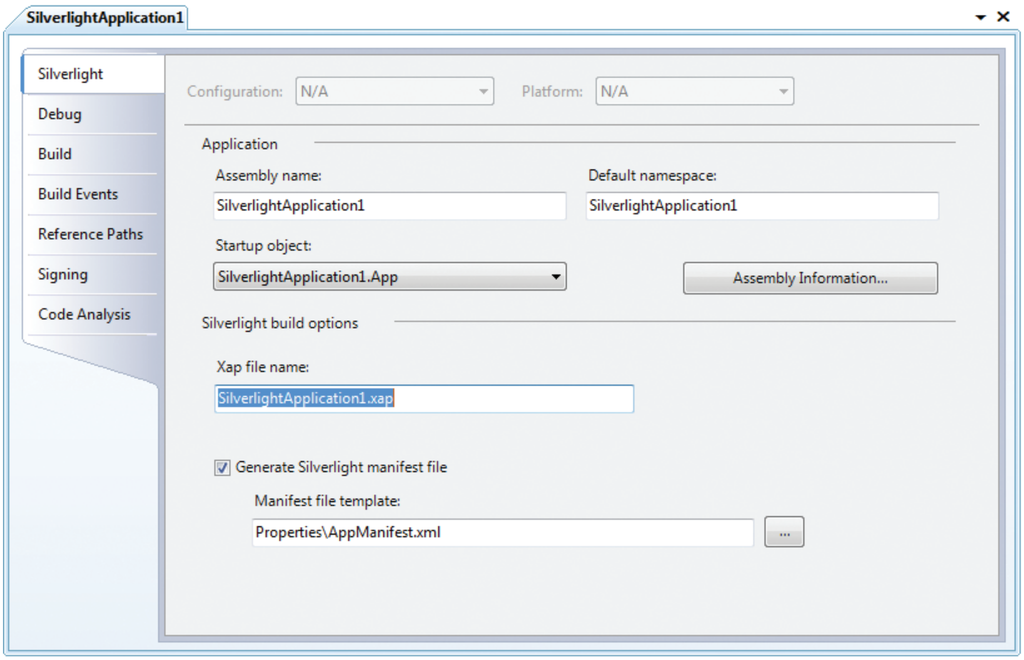
When you compile a Silverlight project, Visual Studio uses the same csc.exe compiler that you use for full-fledged .NET applications. However, it references a different set of assemblies and it passes in the command-line argument *nostdlib*, which prevents the C# compiler from using the standard library (the core parts of the .NET Framework that are defined in mscorlib.dll). In other words, Silverlight applications can be compiled like normal .NET applications written in standard C#, just with a more limited set of class libraries to draw on. The Silverlight compilation model has a number of advantages, including easy deployment and vastly improved performance when compared to ordinary JavaScript.

Your compiled Silverlight assembly includes the compiled code *and* the XAML documents for every page in your application, which are embedded in the assembly as resources. This ensures that there's no way for your event handling code to become separated from the user interface markup it needs. Incidentally, the XAML is not compiled in any way (unlike WPF, which converts it into a more optimized format called BAML).

Your Silverlight project is compiled into a DLL file named after your project. For example, if you have a project named SilverlightApplication1, the csc.exe compiler will create the file SilverlightApplication1.dll. The project assembly is dumped into a Bin\Debug folder in your project directory, along with a few other important files:

- **A PDB file.** This file contains information required for Visual Studio debugging. It's named after your project assembly (for example, SilverlightApplication1.pdb).
- **AppManifest.xaml.** This file lists assembly dependencies.
- **Dependent assemblies.** The Bin\Debug folder contains the assemblies that your Silverlight project uses, provided these assemblies have the Copy Local property set to true. Assemblies that are a core part of Silverlight have Copy Local set to false, because they don't need to be deployed with your application. (You can change the Copy Local setting by expanding the References node in the Solution Explorer, selecting the assembly, and using the Properties window.)
- **TestPage.html.** This is the entry page that the user requests to start your Silverlight application.
- **A XAP file.** This is a Silverlight package that contains everything you need to deploy your Silverlight application, including the application manifest, the project assembly, and any other assemblies that your application uses.

Of course, you can change the assembly name, the default namespace (which is used when you add new code files), and the XAP file name using the Visual Studio project properties (Figure 1-8). Just double-click the Properties node in the Solution Explorer.



Configuration: N/A
Platform: N/A

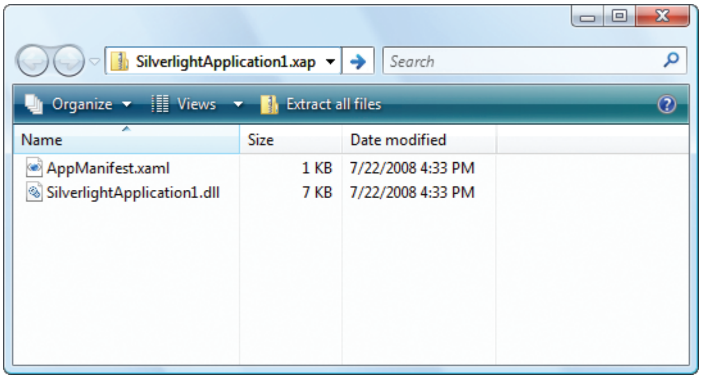
Assembly name: SilverlightApplication1
Default namespace: SilverlightApplication1
Startup object: SilverlightApplication1.App
Xap file name: SilverlightApplication1.xap
[✓] Generate Silverlight manifest file
Manifest file template: Properties\AppManifest.xml

Figure 1-8. *Project properties in Visual Studio*

Deploying a Silverlight Application

Once you understand the Silverlight compilation model, it's a short step to understanding the deployment model. The XAP file is the key piece. It wraps the units of your application (the application manifest and the assemblies) into one neat container.

Technically, the XAP file is a ZIP archive. To verify this, rename a XAP file like SilverlightApplication1.xap to SilverlightApplication1.xap.zip. You can then open the archive and view the files inside. Figure 1-9 shows the contents of the XAP file for the simple example shown earlier in this chapter. Currently, it includes the application manifest and the application assembly. If your application uses add-on assemblies like System.Windows.Controls.dll, you'll find them in the XAP file as well.



Name	Size	Date modified
AppManifest.xaml	1 KB	7/22/2008 4:33 PM
SilverlightApplication1.dll	7 KB	7/22/2008 4:33 PM

Figure 1-9. *The contents of a XAP file*

The XAP file system has two obvious benefits.

- **It compresses your content.** Because this content isn't decompressed until it reaches the client, it reduces the time required to download your application. This is particularly important if your application contains large static resources (see Chapter 6), like images or blocks of text.
- **It simplifies deployment.** When you're ready to take your Silverlight application live, you simply need to copy the XAP file to the web server, along with TestPage.html or a similar HTML file that includes a Silverlight content region. You don't need to worry about keeping track of the assemblies and resources.

Thanks to the XAP model, there's not much to think about when deploying a simple Silverlight application. Hosting a Silverlight application simply involves making the appropriate XAP file available, so the clients can download it through the browser and run it on their local machines.

■ **Tip** Microsoft provides a free hosting solution that offers an impressive 10GB of space for Silverlight applications. To sign up, see `http://silverlight.live.com`.

However, there's one potential stumbling block. When hosting a Silverlight application, your web server must be configured to allow requests for the XAP file type. This file type is included by default in IIS 7, provided you're using Windows Server 2008 or Windows Vista with Service Pack 1. If you have Windows Vista without Service Pack 1, you have an earlier version of IIS, or you have another type of web server, you'll need to add a file type that maps the .xap extension to the MIME type application/x-silverlight-app. For IIS instructions, see `http://learn.iis.net/page.aspx/262/silverlight`.

Tip In some situations, you may want to optimize startup times by splitting your Silverlight application into pieces that can be downloaded separately. In Chapter 6, you'll learn how to use this advanced technique to go beyond the basic single-XAP deployment model.

The HTML Test Page

The last ingredient in the deployment picture is the HTML test page. This page is the entry point into your Silverlight content—in other words, the page the user requests in the web browser. Visual Studio names this file TestPage.html (in a Silverlight-only solution), although you'll probably want to rename it to something more appropriate.

The HTML test page doesn't actually contain Silverlight markup or code. Instead, it simply sets up the content region for the Silverlight plug-in, using a small amount of JavaScript. (For this reason, browsers that have JavaScript disabled won't be able to see Silverlight content.) Here's a slightly shortened version of the HTML test page that preserves the key details:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<!-- saved from url=(0014)about:internet -->
<head>
    <title>Silverlight Project Test Page</title>

    <style type="text/css">
        ...
    </style>

    <script type="text/javascript">
        ...
    </script>
</head>

<body>
    <!-- Runtime errors from Silverlight will be displayed here. -->
    <div id='errorLocation' style="font-size: small;color: Gray;"></div>

    <!-- Silverlight content will be displayed here. -->
    <div id="silverlightControlHost">
        <object data="data:application/x-silverlight,"
         type="application/x-silverlight-2" width="100%" height="100%">
            <param name="source" value="SilverlightApplication1.xap"/>
            <param name="onerror" value="onSilverlightError" />
            <param name="background" value="white" />

            <a href="http://go.microsoft.com/fwlink/?LinkID=108182">
              <img src="http://go.microsoft.com/fwlink/?LinkId=108181"
               alt="Get Microsoft Silverlight" style="border-style: none"/>
            </a>
        </object>
        <iframe style='visibility:hidden;height:0;width:0;border:0px'></iframe>
    </div>
</body>
</html>
```

The key details in this markup are the two highlighted <div> elements. Both of these <div> elements are placeholders that are initially left empty. The first <div> element is reserved for error messages. If the Silverlight plug-in is launched but the Silverlight assembly fails to load successfully, an error message will be shown here, thanks to this JavaScript code, which is featured earlier in the page:

```
<script type="text/javascript">
    function onSilverlightError(sender, args) {
        if (args.errorType == "InitializeError")  {
            var errorDiv = document.getElementById("errorLocation");
            if (errorDiv != null)
                errorDiv.innerHTML = args.errorType + "- " + args.errorMessage;
        }
    }
</script>
```

Tip The error display is a debugging convenience. When you're ready to deploy the application, you should remove the <div> element and error handling code so sensitive information won't be shown to the user (who isn't in a position to correct the problem anyway).

This second <div> element is more interesting. It represents the Silverlight content region. It contains an <object> element that loads the Silverlight plug-in and an <iframe> element that's used to display it in certain browsers. The <object> element includes four key attributes: data (which identifies it as a Silverlight content region), type (which indicates the required Silverlight version), and height and width (which determine the dimensions of the Silverlight content region).

```
<object data="data:application/x-silverlight,"
  type="application/x-silverlight-2" width="100%" height="100%">
    ...
</object>
```

Sizing the Silverlight Content Region

By default, the Silverlight content region is given a width and height of 100%, so the Silverlight content can consume all the available space in the browser window. You can constrain the size of Silverlight content region by hard-coding pixel sizes for the height and width (which is limiting and usually avoided). Or, you can place the <div> element that holds the Silverlight content region in a more restrictive place on the page—for example, in a cell in a table, in another fixed-sized element, or between other <div> elements in a multicolumn layout.

Even though the default test page sizes the Silverlight content region to fit the available space in the browser window, your XAML pages may include hard-coded dimensions. By default, Visual Studio assigns every new Silverlight page a width of 400 pixels and a height of 300 pixels, and the example you saw earlier in this chapter limited the page to 400×100 pixels. If the browser window is larger than the hard-coded page size, the extra space won't be used. If the browser window is smaller than the hard-coded page size, part of the page may fall outside the visible area of the window.

Hard-coded sizes make sense when you have a graphically rich layout with absolute positioning and little flexibility. If you don't, you might prefer to remove the Width and Height attributes from the <UserControl> start tag. That way, the page will be sized to match the Silverlight content region, which in turn is sized to fit the browser window, and your Silverlight content will always fit itself into the currently available space.

To get a better understanding of the actual dimensions of the Silverlight content region, you can add a border around it by adding a simple style rule to the <div>, like this:

```
<div id="silverlightControlHost" style="border: 1px red solid">
```

You'll create resizable and scalable Silverlight pages in Chapter 3, when you explore layout in more detail.

Configuring the Silverlight Content Region

The <object> element contains a series of <param> elements that specify additional options to the Silverlight plug-in. Here are three of the options in the standard test page that Visual Studio generates:

```
<param name="source" value="SilverlightApplication1.xap"/>
<param name="onerror" value="onSilverlightError" />
<param name="background" value="white" />
```

Table 1-1 lists all the parameters that you can use. You'll use several of these parameters in examples throughout this book, as you delve into features like HTML access, splash screens, transparency, and animation.

Table 1-1. *Parameters for the Silverlight Plug-In*

Name	Value
source	A URI that points to the XAP file for your Silverlight application. This parameter is required.
background	The color that's used to paint the background of the Silverlight content region, behind any content that you display (but in front of any HTML content that occupies the same space). If you set the Background property of a page, it's painted over this background.

Name	Value
enableHtmlAccess	A Boolean that specifies whether the Silverlight plug-in has access to the HTML object model. Use true if you want to be able to interact with the HTML elements on the test page through your Silverlight code (as demonstrated in Chapter 12).
initParams	A string that you can use to pass custom initialization information. This technique (which is described in Chapter 6) is useful if you plan to use the same Silverlight application in different ways on different pages.
maxFramerate	The desired frame rate for animations. Higher frame rates result in smoother animations, but the system load and processing power of the current computer may mean that a high frame rate can't be honored. The value is 60 (for 60 frames per second). Animation is discussed in Chapter 9.
splashScreenSource	The location of a XAML splash screen to show while the XAP file is downloading. You'll learn how to use this technique in Chapter 6.
windowless	A Boolean that specifies whether the plug-in renders in windowed mode (the default) or windowless mode. If you set this true, the HTML content underneath your Silverlight content region can show through. This is ideal if you're planning to create a shaped Silverlight control that integrates with HTML content, and you'll see how to use it in Chapter 12.
onSourceDownloadProgressChanged	A JavaScript event handler that's triggered when a piece of the XAP file has been downloaded. You can use this event handler to build a startup progress bar, as in Chapter 6.
onSourceDownloadComplete	A JavaScript event handler that's triggered when the entire XAP file has been downloaded.
onLoad	A JavaScript event handler that's triggered when the markup in the XAP file has been processed and your first page has been loaded.
onResize	A JavaScript event handler that's triggered when the size of a Silverlight content region has changed.
onError	A JavaScript event handler that's triggered when a unhandled error occurs in the Silverlight plug-in or in your code).

■ **Note** By convention, all of these parameter names should be written completely in lowercase (for example, splashscreensource rather than splashScreenSource). However, they're shown with mixed case here for better readability.

Alternative Content

The <div> element also has some HTML markup that will be shown if the <object> tag isn't understood or the plug-in isn't available. In the standard test page, this markup consists of a "Get Silverlight" picture, which is wrapped in a hyperlink that, when clicked, takes the user to the Silverlight download page.

```
<a href="http://go.microsoft.com/fwlink/?LinkID=108182">
  <img src="http://go.microsoft.com/fwlink/?LinkId=108181"
  alt="Get Microsoft Silverlight" style="border-style: none"/>
</a>
```

The Mark of the Web

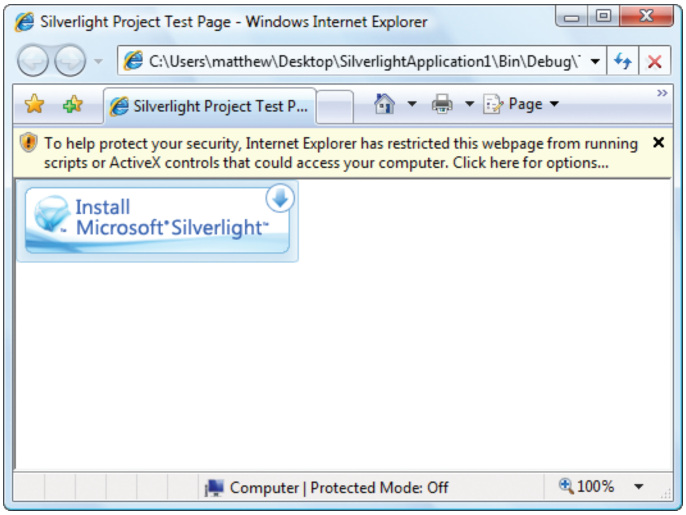
One of the stranger details in the HTML test page is the following comment, which appears in the second line:

```
<!-- saved from url=(0014)about:internet -->
```

Although this comment appears to be little more than an automatically generated stamp that the browser ignores, it actually has an effect on the way you debug your application. This comment is known as the *mark of the web*, and it's a specialized flag that forces Internet Explorer to run pages in a more restrictive security zone than it would normally use.

Ordinarily, the mark of the web indicates the website from which a locally stored page was originally downloaded. But in this case, Visual Studio has no way of knowing where your Silverlight application will eventually be deployed. It falls back on the URL `about:internet`, which simply signals that the page is from some arbitrary location on the public Internet. The number (14) simply indicates the number of characters in this URL. For a more detailed description of the mark of the web and its standard uses, see `http://msdn.microsoft.com/en-us/library/ms537628(VS.85).aspx`.

All of this raises an obvious question—namely, why is Visual Studio adding a marker that's typically reserved for downloaded pages? The reason is that without the mark of the web, Internet Explorer will load your page with the relaxed security settings of the local machine zone. This wouldn't cause a problem, except for the fact that Internet Explorer also includes a safeguard that disables scripts and ActiveX controls in this situation. As a result, if you run a test page that's stored on your local hard drive, and this test page doesn't have the mark of the web, you'll see the irritating warning message shown in Figure 1-10, and you'll need to explicitly allow the blocked content. Worst of all, you'll need to repeat this process every time you open the page.


Silverlight Project Test Page - Windows Internet Explorer

C:\Users\matthew\Desktop\SilverlightApplication1\Bin\Debug\

To help protect your security, Internet Explorer has restricted this webpage from running scripts or ActiveX controls that could access your computer. Click here for options...

Computer | Protected Mode: Off

Silverlight Project Test P...
Page
Install Microsoft® Silverlight™
100%

Figure 1-10. *A page with disabled Silverlight content*

This problem will disappear when you deploy the web page to a real website, but it's a significant inconvenience while testing. To avoid headaches like these, make sure you add a similar mark of the web comment if you design your own custom test pages.

CHANGING THE TEST PAGE

Visual Studio generates the test page each time you run the project. As a result, any changes you make to it will be discarded. If you want to customize the test page, the easiest solution is to create a new test page for your project. Here's how:

1. Run your project at least once to create the test page.
2. Click the Show All Files icon at the top of the Solution Explorer.
3. Expand the Bin\Debug folder in the Solution Explorer.
4. Find the TestPage.html file, right-click it, and choose Copy. Then right-click the Bin\Debug folder and choose Paste. This duplicate will be your custom test page. Right-click the new file and choose Rename to give it a better name.
5. To make the custom test page a part of your project, right-click it and choose Include in Project.
6. To tell Visual Studio to navigate to your test page when you run the project, right-click your test page and choose Set As Start Page.

The Application Manifest

As you've seen, the Silverlight execution model is quite straightforward. First, the client requests the HTML test page (such as TestPage.html). At this point, the browser downloads the HTML file and processes its markup. When it reaches the <object> element, it loads the Silverlight plug-in and creates the Silverlight content region. After this step, the client-side plug-in takes over. First, it downloads the linked XAP file (which is identified by the source parameter inside the <object> element). Then, it looks at the AppManifest.xaml file to decide what to do next.

Here's the content of the AppManifest.xaml for a newly generated Visual Studio project, which also matches the AppManifest.xaml in the simple example you saw earlier in this chapter:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 EntryPointAssembly="SilverlightApplication1"
 EntryPointType="SilverlightApplication1.App" RuntimeVersion="2.0.30904.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="SilverlightApplication1"
     Source="SilverlightApplication1.dll" />
  </Deployment.Parts>
</Deployment>
```

The EntryPointAssembly and EntryPointType attributes are the key details that determine what code the Silverlight plug-in will execute next. EntryPointAssembly indicates the name of the DLL that has your compiled Silverlight code (without the .dll extension). EntryPointType indicates the name of the application class in that assembly. When the Silverlight plug-in sees the AppManifest.xaml shown here, it loads the SilverlightApplication1.dll assembly, and then creates the App object inside. The App object triggers a Startup event, which runs this code, creating the first page:

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    // Load the main control.
    this.RootVisual = new Page();
}
```

If you've added a different user control to your application, and you want to show it as the first page, simply edit the App.xaml.cs file, and replace the Page class with the name of your custom class:

```
this.RootVisual = new CustomPage();
```

There's one other setting that you can add to the AppManifest.xaml file—the External-CallersFromCrossDomain setting. To understand the purpose it plays, you need to realize that Silverlight supports *cross-domain* deployment. This means Silverlight allows you to place your XAP file on one web server and your HTML or ASP.NET entry page on another. In this situation, you'll obviously need to edit the test page and modify the source parameter in the <object> element so that it points to the remote XAP file. However, there's one catch. To defeat certain types of attacks, Silverlight doesn't allow the hosting web page and your Silverlight code to interact if they're on different servers. If you do need this ability (which is described in Chapter 12), you need to set the ExternalCallersFromCrossDomain setting like this:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 ExternalCallersFromsCrossDomain="ScriptableOnly" ...>
```

The only value you can use other than ScriptableOnly is NoAccess.

SILVERLIGHT DECOMPILATION

Now that you understand the infrastructure that underpins a Silverlight project, it's easy to see how you can decompile any existing application to learn more about how it works. Here's how:

1. Surf to the entry page.
2. View the source for the web page, and look for the <param> element that points to the XAP file.
3. Type a request for the XAP file into your browser's address bar. (Keep the same domain, but replace the page name with the partial path that points to the XAP file.)
4. Choose Save As to save the XAP file locally.
5. Rename the XAP file to add the .zip extension. Then, open it and extract the project assembly. This assembly is essentially the same as the assemblies you build for ordinary .NET applications. Like ordinary .NET assemblies, it contains IL (Intermediate Language) code.
6. Open the project assembly in a tool like Reflector (`http://www.red-gate.com/products/reflector`) to view the IL and embedded resources. Using the right plug-in, you can even decompile the IL to C# syntax.

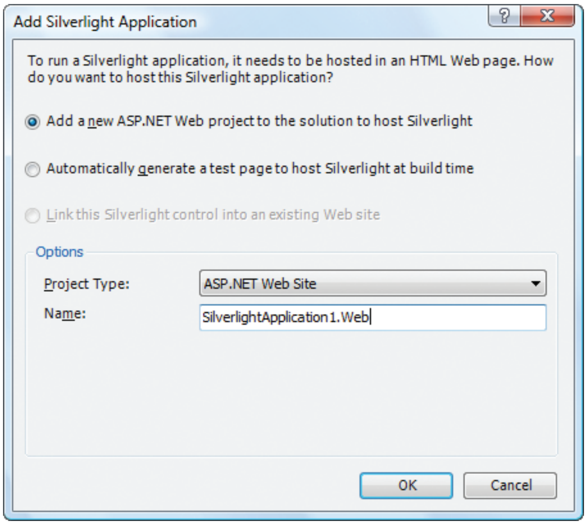
Of course, many Silverlight developers don't condone this sort of behavior (much as many .NET developers don't encourage end users to decompile their rich client applications). However, it's an unavoidable side effect of the Silverlight compilation model.

Because IL code can be easily decompiled or reverse engineered, it's not an appropriate place to store secrets (like encryption keys, proprietary algorithms, and so on). If you need to perform a task that uses sensitive code, consider calling a web service from your Silverlight application. If you just want to prevent other hotshots from reading your code and copying your style, you may be interested in raising the bar with an *obfuscation* tool that uses a number of tricks to scramble the structure and names in your compiled code without changing its behavior. Visual Studio ships with a scaled-down obfuscation tool named Dotfuscator, and many more are available commercially.

Creating an ASP.NET-Hosted Silverlight Project

Although Silverlight does perfectly well on its own, you can also develop, test, and deploy it as part of an ASP.NET website. Here's how to create a Silverlight project and an ASP.NET website that uses it in the same solution:

1. Select File ➤ New ➤ Project in Visual Studio, choose the Visual C# group of project types, and select the Silverlight Application template. It's a good idea to use the "Create directory for solution" option, so you can group together the two projects that Visual Studio will create—one for the Silverlight assembly and one for ASP.NET website.
2. Once you've picked the solution name and project name, click OK to create it.
3. When asked whether you want to create a test web, choose the first option, "Add a new Web." You'll also need to supply a project name for the ASP.NET website. By default, it's your project name with the added word Web at the end, as shown in Figure 1-11.


Add Silverlight Application

To run a Silverlight application, it needs to be hosted in an HTML Web page. How do you want to host this Silverlight application?

[✓ Add a new ASP.NET Web project to the solution to host Silverlight]

Automatically generate a test page to host Silverlight at build time

Link this Silverlight control into an existing Web site

Options

Project Type: ASP.NET Web Site

Name: SilverlightApplication1.Web

Figure 1-11. *Creating an ASP.NET website to host Silverlight content*

4. In the Project Type box, choose the way you want Visual Studio to manage your project—either as a web project or a website. The choice has no effect on how Silverlight works. If you choose a web project, Visual Studio uses a project file to track the contents of your web application and compiles your web page code into a single assembly before you run it. If you choose a website, Visual Studio simply assumes everything in the application folder is a part of your web application. Your web page code will be compiled the first time a user requests a page (or when you use the precompilation tool aspnet_compiler.exe).

■ **Tip** For more information about the difference between web projects and projectless websites, and other ASP.NET basics, refer to *Pro ASP.NET 3.5 in C# 2008*.

5. Finally, click OK to create the solution.

■ **Note** If you create an ordinary HTML-only website, you can host it on any web server. In this scenario, the web server has an easy job—it simply needs to send along your HTML files when a browser requests them. If you decide to create an ASP.NET website, your application's requirements change. Although the Silverlight portion of your application will still run on the client, any ASP.NET content you include will run on the web server, which must have the ASP.NET engine installed.

There are two ways to integrate Silverlight content into an ASP.NET application:

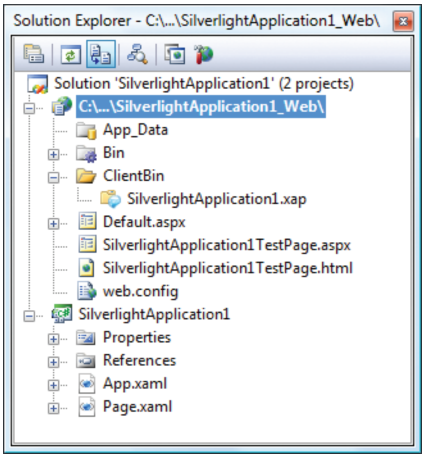
- **Create HTML files with Silverlight content.** You place these files in your ASP.NET website folder, just as you would with any other ordinary HTML file. The only limitation of this approach is that your HTML file obviously can't include ASP.NET controls, because it won't be processed on the server.
- **Place Silverlight content inside an ASP.NET web form.** To pull this trick off, you need the help of the Silverlight web control. You can also add other ASP.NET controls to different regions of the page. The only disadvantage to this approach is that the page is always processed on the server. If you aren't actually using any server-side ASP.NET content, this creates an extra bit of overhead that you don't need when the page is first requested.

Of course, you're also free to mingle both of these approaches, and use Silverlight content in dedicated HTML pages and inside ASP.NET web pages in the same site. When you create a Silverlight project with an ASP.NET website, you'll start with both. For example, if your Silverlight project is named SilverlightApplication1, you can use SilverlightApplication1TestPage.html or SilverlightApplication1TestPage.aspx.

The HTML file is identical to the test page in the ordinary Silverlight-only solution you saw earlier. The only difference is that the page is generated once, when the ASP.NET website is first created, not every time you build the project. As a result, you can modify the HTML page without worrying that your changes will be overridden.

The .aspx file is an ASP.NET web form that uses ASP.NET's Silverlight web control to show your Silverlight application. The end result is the same as the HTML test page, but there's a key difference—the Silverlight control creates the test page markup dynamically, when it's processed on the server. This extra step gives you a chance to use your own server-side code to perform other tasks when the page is initially requested, before the Silverlight application is downloaded and launched. You'll explore the Silverlight web control in Chapter 13.

Figure 1-12 shows how a Silverlight and ASP.NET solution starts out. Along with the two test pages, the ASP.NET website also includes a Default.aspx page (which can be used as the entry point to your ASP.NET website) and web.config (which allows you to configure various website settings).

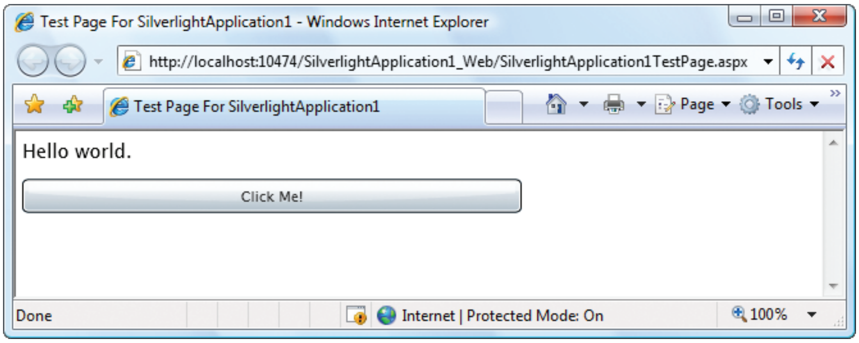

Solution Explorer - C:\...\SilverlightApplication1_Web\

```
Solution 'SilverlightApplication1' (2 projects)
C:\...\SilverlightApplication1_Web\
    App_Data
    Bin
    ClientBin
        SilverlightApplication1.xap
    Default.aspx
    SilverlightApplication1TestPage.aspx
    SilverlightApplication1TestPage.html
    web.config
SilverlightApplication1
    Properties
    References
    App.xaml
    Page.xaml
```

Figure 1-12. *Creating an ASP.NET website to host Silverlight content*

The Silverlight and ASP.NET option provides essentially the same debugging experience as a Silverlight-only solution. When you run the solution, Visual Studio compiles both projects, and copies the Silverlight assembly to the ClientBin folder in the ASP.NET website. (This is similar to assembly references—if an ASP.NET website references a private DLL, Visual Studio automatically copies this DLL to the Bin folder.)

Once both projects are compiled, Visual Studio looks to the startup project (which is the ASP.NET website) and looks for the currently selected page. It then launches the default browser and navigates to that page. The difference is that it doesn't request the start page directly from the file system. Instead, it communicates with its built-in test web server. This web server automatically loads up on a randomly chosen port. It acts like a scaled-down version of IIS, but accepts requests only from the local computer. This gives you the ease of debugging without needing to configure IIS virtual directories. Figure 1-13 shows the same Silverlight application you considered earlier, but hosted by ASP.NET.


Test Page For SilverlightApplication1 - Windows Internet Explorer

http://localhost:10474/SilverlightApplication1_Web/SilverlightApplication1TestPage.aspx

Hello world.

Test Page For SilverlightApplication1
Page
Tools
Click Me!
Done
Internet | Protected Mode: On
100%

Figure 1-13. *An ASP.NET page*

To navigate to a different page from the ASP.NET project, you can type in the address bar of the browser.

■ **Note** Remember, when building a Silverlight and ASP.NET solution, you add all your Silverlight files and code to the Silverlight project. The ASP.NET website consumes the final, compiled Silverlight assembly, and makes it available through one or more of its web pages.

The Last Word

In this chapter, you took your first look at the Silverlight application model. You saw how to create a Silverlight project in Visual Studio, add a simple event handler, and test it. You also peered behind the scenes to explore how a Silverlight application is compiled and deployed.

In the following chapters, you'll learn much more about the full capabilities of the Silverlight platform. Sometimes, you might need to remind yourself that you're coding inside a lightweight browser-hosted framework, because much of Silverlight coding feels like the full .NET platform, despite the fact that it's built on only a few megabytes of compressed code. Out of all of Silverlight's many features, its ability to pack a miniature modern programming framework into a slim 4MB download is surely its most impressive.