

APPENDIXES B - F

Silverlight 5

IN ACTION

Revised Edition of
Silverlight 4 in Action

Pete Brown



MANNING



Silverlight 5 in Action

by Pete Brown

Appendixes B - F

Copyright 2012 Manning Publications

online appendixes

APPENDIX B	MEDIA BASICS	1
APPENDIX C	RAW MEDIA, WEBCAM, AND MICROPHONE.....	38
APPENDIX D	INTRODUCTION TO WCF RIA SERVICES.....	68
APPENDIX E	WCF RIA ENDPOINTS, SECURITY, BUSINESS LOGIC, AND DECOUPLING	105
APPENDIX F	WCF RIA SERVICES AND MVVM.....	130

brief contents of Silverlight 5 in Action

PART 1 CORE SILVERLIGHT 1

- 1 ■ Introducing Silverlight 3
- 2 ■ XAML and the property system 26
- 3 ■ The application model and the plug-in 47
- 4 ■ Working with HTML and browsers 73
- 5 ■ Out-of-browser applications 95
- 6 ■ The security model and elevated trust 114

PART 2 CREATING THE USER INTERFACE 125

- 7 ■ Rendering, layout, and transforming 127
- 8 ■ Panels 160
- 9 ■ Human input 180
- 10 ■ Text fundamentals 199
- 11 ■ Editing plain and rich text 225
- 12 ■ Control basics and UserControls 247
- 13 ■ Animation and behaviors 272

- 14 ▪ Resources, styles, and control templates 307
- 15 ▪ Extensions, converters, custom controls, and panels 337

PART 3 WORKING WITH DATA AND SERVICES 365

- 16 ▪ Binding 367
- 17 ▪ Data controls: DataGrid and DataForm 410
- 18 ▪ Input validation 433
- 19 ▪ Networking basics 460
- 20 ▪ Working with SOAP services 491
- 21 ▪ RESTful services with the ASP.NET Web API 520
- 22 ▪ Working with XML, JSON, RSS, and Atom 549
- 23 ▪ Duplex, sockets, and local connections 575

PART 4 2D AND 3D GRAPHICS 601

- 24 ▪ Graphics and effects 603
- 25 ▪ Working with images 630
- 26 ▪ Introduction to 3D 649
- 27 ▪ 3D lighting, texturing, and animation 679

PART 5 MAKING THE MOST OF THE PLATFORM 709

- 28 ▪ Pop-ups, windows, and full-screen applications 711
- 29 ▪ Navigation 734
- 30 ▪ Working with files and directories 761
- 31 ▪ Printing 798
- 32 ▪ COM, Native Extensions, and p-invoke 832

PART 6 BEST PRACTICES 867

- 33 ▪ Structuring and testing with the MVVM pattern 869
- 34 ▪ Debugging your application 913
- 35 ▪ The install experience and preloaders 929

appendix A ▪ Database, connection, and data model setup 939

index 945

appendix B: *Media basics*

If you ask most non-Silverlight developers what Silverlight is, 8 out of 10 will probably say it's Microsoft's web media player. Part of that reputation comes from Silverlight 1.0, which was only good as a media player. The other part comes from the incredible advances the Silverlight team has made in making Silverlight a first-class media platform for the web.

Silverlight excels at delivering high-quality HD media. In fact, it was one of the first web technologies to support true 720p and 1080p HD media over decent but not abnormal network pipes. Silverlight has been the driving media force behind Netflix, as well as many online events such as the Olympics and March Madness. Media is what has helped Silverlight expand onto the majority of internet-connected desktops.

Top-notch media support is core to Silverlight.

Throughout this appendix, you'll learn how to use items from within the `System.Windows.Controls` namespace to enable playback and interaction with media. You'll first see the most basic of media elements, the aptly named `MediaElement` control. Then, you'll learn how to manage the media experience through the use of playlists and interactive playback.

You'll then take a look at every couch potato's best friend: the remote control. Silverlight includes built-in support for Windows and Mac remote controls, as well as the dedicated and multifunction media keys found on most modern keyboards. You can play, pause, forward, rewind, and do all the normal media operations you'd expect. This truly enables Silverlight for the 10-foot living room experience and lets you keep your hands off the mouse even when watching video on your regular desktop or laptop.

One of the most exciting media enhancements for Silverlight 5 is trick play, the ability to shorten the amount of time you spend just watching videos without making the speaker sound like they inhaled a balloon full of helium. This enables you to watch something like a tutorial video at 1.5x or 2x normal speed while still hearing

fully intelligible audio. It even supports playback in slow motion in case the speaker normally speaks at 2 times normal human speed, like the Micro Machines guy from the ads in the 1980s. Want to slow down a video to see what chords the guitarist was using while still keeping the pitch accurate to the original playing? It's excellent for that too.

From there, you'll learn about accessing protected content using Digital Rights Management (DRM), a dirty word in some circles, but an essential feature for large content publishers such as Netflix. The appendix will wrap up with a look at the Microsoft Media Platform Player Framework, a great way to have a fully functional and feature-rich player in no time at all.

B.1 *Audio and video*

Integrating media into a Silverlight application is incredibly simple. To include a rich media experience, you employ a `MediaElement` object. This general-purpose object empowers you to deliver rich audio and video content. For a user to enjoy this high-fidelity content, the media item must first be loaded and configured.

In this section, you'll learn how to load and configure audio and video content. This section begins with an in-depth discussion about the `MediaElement`'s `Source` property. From there, you'll see the properties that you can use to configure both audio and video items. Next, you'll explore the items directly related to audio content. I then shift to a focus on video content. This section concludes with an explanation of the lifecycle of a media file within a `MediaElement`.

B.1.1 *Media source*

The `Source` property of the `MediaElement` specifies the location of the audio or video file to play. This file can be referenced by using either a relative or absolute URL. If you have a video file called `video.wmv` in a subdirectory called `Media` within your web application, you can use it by setting the `Source` property to `Media/video.wmv`. This example shows a `MediaElement` that uses a relative media file:

```
<Grid x:Name="LayoutRoot" Background="White">
    <MediaElement x:Name="VideoPlayer" Source="Media/video.wmv" />
</Grid>
```

This code shows a video that belongs to the same web application as the Silverlight application. Note the use of the forward slash (/) in the `Source` property. This property allows you to use forward slashes but not backslashes (\). In addition, the `Source` property has support for cross-domain URIs.

Cross-domain URIs allow you to specify an absolute path to a media file. This feature gives you the flexibility to use a media asset stored on another server. If you choose to use this approach, it's important to gain permission to use the file before doing so. You have my permission to reference the video shown here (it's around 12MB):

```
<Grid x:Name="LayoutRoot">
    <MediaElement x:Name="VideoPlayer"
        Source="http://10rem.net/pub/sl5ia/NetduinoRobot_SmallM.wmv" />
</Grid>
```

This example shows a video being accessed from a remote server. When accessing content from a remote server, you must use one of the three acceptable protocols. Silverlight supports the HTTP, HTTPS, and MMS protocols. In addition, the `Source` property expects certain formats.

SUPPORTED FORMATS

Have you ever wanted a snack or soda and accidentally put foreign currency in your local vending machine? Or have you ever put a DVD into a CD player? What happened? Most likely, either nothing happened or some type of error was displayed. These scenarios show that devices are created with specific formats in mind. Likewise, the `MediaElement` expects certain formats.

The `MediaElement` supports a powerful array of audio and video formats that empower you to deliver high-quality media experiences over the internet. The accepted audio formats ensure a truly high-fidelity aural experience. At the same time, the supported video formats ensure a viewing experience that can scale from mobile devices all the way up to high-definition displays. Table B.1 shows the formats supported by the `MediaElement`.

Table B.1 Media containers and codecs supported by Silverlight

Container	Codec
Windows Media	Windows Media Audio 7, 8, 9 (WMA Standard) Windows Media Audio 9, 10 (WMA Professional) WMV1 (Windows Media Video 7) WMV2 (Windows Media Video 8) WMV3 (Windows Media Video 9)
MP4	H.264 (ITU-T H.264 / ISO MPEG-4 AVC), AAC-LC
MP3	ISO MPEG-1 Layer III (MP3)

By targeting these media formats, the Silverlight runtime can be a self-contained environment for media experiences. Once your users install the Silverlight runtime, they can run all the supported media formats without having to download and install additional codecs.

The format for media is important, but the delivery method is equally so. Table B.2 lists the delivery methods Silverlight recognizes for audio and video.

Table B.2 Supported media delivery methods

Delivery method	Supported containers
Progressive download	Windows Media, MP4, MP3, ASX
Windows Media Streaming over HTTP	Windows Media Server-Side Play List (SSPL)
Smooth Streaming	fMP4 (Fragmented MP4)
ASX	Windows Media, MP4, ASX

Table B.2 Supported media delivery methods (*continued*)

Delivery method	Supported containers
PlayReady DRM	MP4
Server-side playlist	Windows Media
MediaStreamSource	Any container, as long as you write a parser for it

In addition to the progressive download formats, table B.2 shows two different streaming methods: Smooth Streaming and Windows Media Streaming over HTTP.

SMOOTH STREAMING WITH IIS

Smooth Streaming is an HTTP-based multiple bit rate (MBR) adaptive media streaming service implemented on Internet Information Server (IIS) on Windows servers. Smooth Streaming dynamically detects client bandwidth and CPU usage and adapts to conditions in close to real time. Smooth Streaming provides:

- Automatic adaptation to CPU constraints
- Automatic adaptation to bandwidth constraints
- Simplified caching and support for content delivery networks (CDN)

For example, if you're watching an HD video on your client and suddenly you start a CPU-intensive process such as a large compile, rather than drop frames Smooth Streaming detects the condition and lowers the quality of the video (it lowers the bit rate, which typically means a lower resolution) so your viewing session continues uninterrupted.

Similarly, if you're watching an HD video and someone in your house starts a large download, effectively taking up a large portion of your internet bandwidth, Smooth Streaming will adapt to that as well, lowering the bit rate to fit into the available bandwidth.

Finally, Smooth Streaming supports simplified caching of content, as the individual chunks are individual files, easily cached using standard HTTP file caching mechanisms. The caches need not know anything about media formats; the bits are just files. For the same reasons, proxies work just as well, requiring no special open ports or knowledge of the formats.

Smooth Streaming delivers small content fragments (about two to four seconds worth of video) to the client and verifies (with the help of Silverlight) that the content all arrived on time and played at the expected quality level. If a fragment doesn't meet these requirements due to bandwidth or processor restrictions, the next fragment will be delivered at a lower quality level. If the conditions were favorable, the next fragment will be delivered at the same or higher quality level.

Similarly, if the video is available in 1080p HD but the user is watching it on a display at 720p resolution, Smooth Streaming will send down only the 720p size chunks, saving bandwidth and processing time.

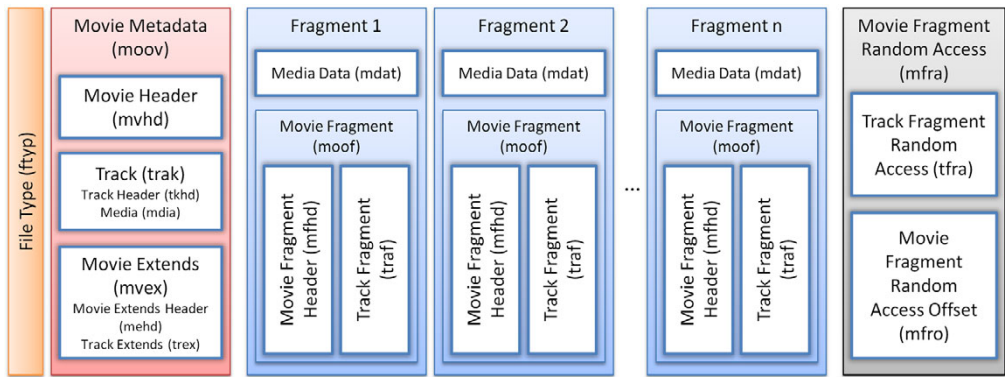


Figure B.1 The Smooth Streaming server-side file format

On the server, this requires that the videos be encoded to several formats. IIS Smooth Streaming keeps all the chunks for a given format in a single MP4 file but delivers the chunks as individual logical files. This makes server file management (and file access) easier, while still providing for caching of chunks by local proxies and downstream servers. Smooth Streaming files have the extension .ismv for video plus audio and .isma for audio only. Figure B.1 shows the structure of the Smooth Streaming file on the server.

The file includes a file type header to let you know this is the Smooth Streaming file. Next, it includes Movie Metadata (moov) that describes what the file contains. Following that are the individual two-second fragments for the entire movie. Each fragment contains header information for the fragment, as well as the fragment bits themselves. The file closes with an “mfra” index that allows for easy and accurate seeking within the file.¹

In addition to the media file described here, Smooth Streaming uses a ISM manifest file for the server, which describes the relationships between the server files, and a ISMC client manifest file, which describes the available streams, codecs, bit rates, markets, and so on. This ISMC file is what’s first delivered to the client when the video is requested.

You can see an online example of Smooth Streaming with IIS and Silverlight on the IIS Smooth Streaming site: www.iis.net/media/experiencesmoothstreaming. Other examples of Smooth Streaming through a CDN may be seen at www.smoothhd.com.

To encode video for use with Smooth Streaming, you use Microsoft Expression Encoder. Once the videos are encoded, you can use the Expression Encoder Smooth Streaming template to serve as the start of your video player, or you can use the Microsoft Media Platform: Player Framework.

¹ IIS Smooth Streaming Technical Overview, Alex Zambelli, Microsoft, March 2009. <http://bit.ly/SmoothStreamingTech>. Note that this technology is a moving target, highly affected by advances on the server OS, so refer to the most up-to-date information when implementing your solution.

The Microsoft Media Platform: Player Framework (formerly known as the Silverlight Media Framework or SMF) is the easiest way to incorporate Smooth Streaming into your application. Before I cover that, let's look at other forms of streaming and downloading available to you.

WINDOWS MEDIA STREAMING

Though now generally out of favor due to the introduction of Smooth Streaming, Silverlight still supports streaming media content over HTTP through server-side playlists and the MMS protocol. The MMS protocol was built for sending many short messages to a client and uses a URI that begins with `mms://` instead of `http://` or `https://`. When a media file is streamed through this protocol, your Silverlight application maintains an open connection with the hosting server. This has two advantages. It enables you to jump to any point in time within a media file, and streaming usually provides a more cost-effective approach for delivering audio and video content because only the requested content is downloaded, plus a little extra. This content is configurable through the `BufferingTime` property.

TIP When evaluating media streaming options for HD content, lean toward IIS Smooth Streaming over Windows Media Streaming. IIS Smooth Streaming is better optimized to provide a great user experience with high bit rate content, such as HD video.

The `BufferingTime` property enables you to view or specify how much of a buffer should be downloaded. By default, this `TimeSpan` value is set to buffer 5 seconds' worth of content. If you're streaming a 1-minute video, the video won't begin playing until at least 5 seconds of it has been retrieved. While this retrieval is occurring, the `CurrentState` property of the `MediaElement` (which I'll discuss shortly) will be set to `Buffering`. While the `MediaElement` is in a `Buffering` state, it'll halt playback. You can check to see what percentage of the buffering is completed by checking the `BufferingProgress` property.

The `BufferingProgress` property gives you access to the percentage of the completed buffering. Because this property value is always between 0.0 and 1.0, you need to multiply it by 100 to get the percentage. When this property changes by a value greater than 5 percent, the `BufferingProgressChanged` event will be fired. This event gives you the flexibility to keep your users informed through a progress bar or some other UI construct. As you can imagine, this type of component can be valuable when you're streaming content.

Often, streamed content can be quite lengthy. Because of this, it can be advantageous to use MBR files. MBR files enable you to provide the highest quality experience based on the available bandwidth. The really cool part is that the `MediaElement` will automatically choose which bit rate to use based on the available bandwidth. In addition, the `MediaElement` will automatically attempt to progressively download the content if it can't be streamed. That's thinking progressively.

PROGRESSIVE DOWNLOAD

Progressive downloading involves requesting a media file over the HTTP or HTTPS protocol. When this occurs, the requested content is temporarily downloaded to a user's computer, enabling the user to quickly access any part of the media that has been downloaded. In addition to fast access, using a progressive download generally provides a higher-quality media experience. Progressive downloading usually requires a longer initial wait time than streaming, so you may want to keep your users informed of how much wait time is left.

Keeping your users informed is made possible through two key items within the `MediaElement`. The first item is a property called `DownloadProgress`. It gives you access to the percentage of the content that has been downloaded. The other item is an event called `DownloadProgressChanged`. This event gives you the ability to do something such as update a progress bar whenever the `DownloadProgress` property changes. In this listing, both items are used to show the percentage of requested content that's available.

Listing B.1 The percentage of content ready for use within a `MediaElement`
XAML:

```
<UserControl x:Class="MediaElementExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <TextBlock x:Name="tb" HorizontalAlignment="Left"
            VerticalAlignment="Top" />

        <MediaElement x:Name="me" Margin="0 25 0 0"
            Source="http://localhost:5150/Media/NetduinoRobot_Large.wmv"
            DownloadProgressChanged="me_DownloadProgressChanged" />
    </Grid>
</UserControl>
```

 **MediaElement**
C#:

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace MediaElementExample
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }
        void me_DownloadProgressChanged(object sender, RoutedEventArgs e)
```

```

    {
        double percentage = me.DownloadProgress * 100.0;
        string text = String.Format("{0:f}", percentage) + "%";
        tb.Text = text;
    }
}

```

This code shows a large video file (~83 MB) being progressively downloaded. As this download progresses, the completion percentage is calculated. This percentage is then formatted and presented to the user as the video is downloaded.

I encourage you to download the videos once, store them, and refer to them locally as I've done here. You'll use both videos throughout media examples. You can download them using these URLs:

- http://10rem.net/pub/sl5ia/NetduinoRobot_SmallM.wmv—with markers
- http://10rem.net/pub/sl5ia/NetduinoRobot_Large.wmv—no markers

Whether you stream content or progressively download it, the `MediaElement` expects certain formats. These file formats are then retrieved over one of the accepted protocols (HTTP, HTTPS, or MMS). The `Source` property simplifies this retrieval process, and it works with both audio and video files. Once the media source is loaded, the `MediaElement` can be used to configure the playback of a media item or obtain status information. These items are available through a set of commonly used properties.

B.1.2 Common properties

The `MediaElement` provides a number of properties common to both audio and video files. Interestingly, you've already seen several—the `Source`, `BufferingTime`, `BufferingProgress`, and `DownloadProgress` properties. There are five other properties so fundamental to the `MediaElement` that I should discuss them now. These properties are `AutoPlay`, `CanPause`, `CurrentState`, `NaturalDuration`, and `Position`.

AUTOPLAY

The `AutoPlay` property specifies whether the `MediaElement` will automatically begin playing. By default, a `MediaElement` will begin playing as soon as the content referenced in the `Source` property is loaded. You can disable this default behavior by changing the `AutoPlay` `bool` property to `false`. As you can imagine, once a media file has begun playing, there may be times when you want to be able to pause it.

CANPAUSE

Sometimes you may want to allow a user to halt the playback of a `MediaElement`. By default, the `MediaElement` will let you do this. But by setting the `CanPause` property of the `MediaElement` to `false`, you can prevent your users from pausing the playback. If you allow the pausing function and a user decides to halt the playback, it'll change the value of the `CurrentState` property.

CURRENTSTATE

The `CurrentState` property represents the mode the `MediaElement` is in. This mode is exposed as a value of the `System.Windows.Media.MediaElementState` enumeration. This enumeration provides all the possible states a `MediaElement` can be in, as described in table B.3

Table B.3 The options available within the `MediaElementState` enumeration

Option	Description
<code>AcquiringLicense</code>	Occurs while a protected file is obtaining a license key (see section B.4.3).
<code>Buffering</code>	This signals that the <code>MediaElement</code> is in the process of loading a media file.
<code>Closed</code>	The media has been unloaded from the <code>MediaElement</code> .
<code>Individualizing</code>	Occurs while Silverlight is obtaining PlayReady components (see section B.4.2).
<code>Opening</code>	The <code>MediaElement</code> is trying to open the media item referenced through the <code>Source</code> property.
<code>Paused</code>	This signals that the <code>MediaElement</code> has halted playback.
<code>Playing</code>	This signals that the <code>MediaElement</code> is moving forward and the media is being enjoyed.
<code>Stopped</code>	The <code>MediaElement</code> has media loaded. It isn't currently playing, and <code>Position</code> is located at the start of the file.

The `MediaElementState` enumeration is used by the read-only `CurrentState` property. Considering that this property is read-only, how does it get set? This property is altered through a variety of methods you'll learn about later in this appendix. Any time the `CurrentState` property value is changed, an event called `CurrentStateChanged` is fired. The state of the media item is a natural part of working with the `MediaElement`, as is the duration.

NATURALDURATION

The `NaturalDuration` property gives you access to the natural duration of a media item: its actual, complete duration when played at normal speed. This duration is available once the `MediaElement` has successfully opened a media stream, so you shouldn't use the `NaturalDuration` property until the `MediaOpened` event has fired. Once the `MediaOpened` event has fired, you can access the total length of a media item, as shown here:

```
void me_MediaOpened(object sender, RoutedEventArgs e)
{
    tb.Text = "Your video is " + me.NaturalDuration + " long.";
}
```

This example displays the total length of a media item in an assumed `TextBlock`. This task takes place when the `MediaOpened` event of a `MediaElement` has triggered, so you

can assume that the media stream has been successfully accessed. Then, you use the `NaturalDuration` property to show the length of the media stream. This length is stored as a `TimeSpan` within the `NaturalDuration` property.

The `NaturalDuration` property is a `System.Windows.Duration` entity. This type of entity is a core element of the .NET Framework, and it exposes a property called `HasTimeSpan` that signals whether a `TimeSpan` is available. In the case of a `MediaElement`, this property value will always be `true`, enabling you to access highly detailed information about the length of a media stream through the `TimeSpan` property. This property is demonstrated in this example:

```
void me_MediaOpened(object sender, RoutedEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("Your video is ");
    sb.Append(me.NaturalDuration.TimeSpan.Minutes);
    sb.Append(" minutes, ");
    sb.Append(me.NaturalDuration.TimeSpan.Seconds);
    sb.Append(" seconds, and ");
    sb.Append(me.NaturalDuration.TimeSpan.Milliseconds);
    sb.Append("milliseconds.");
    tb.Text = sb.ToString();
}
```

This code shows how to access detailed information about the length of a media item. As you probably know, this information, as well as the position of the playback, is part of almost any online media player.

POSITION

The `Position` property represents a point, or location, within a `MediaElement`. This value can be read regardless of the `CurrentState` of the `MediaElement`, and it can be set if the `MediaElement` object's `CanSeek` property is `true`.

The `CanSeek` property determines whether the `Position` can be programmatically changed. This read-only property is set when a media item is loaded into a `MediaElement`. If the referenced media item is being streamed, this property will be set to `false`. If the referenced media item is being downloaded progressively, the `CanSeek` property will be set to `true`.

When the `CanSeek` property is set to `true`, you can set the `Position` property to any `TimeSpan` value. It's recommended that you use a `TimeSpan` within the `NaturalDuration` of a `MediaElement`. If you use a `TimeSpan` beyond the `NaturalDuration`, the `MediaElement` will jump to the end of the media item.

The `Position` is an important part of any media item—and so are the other common properties shared across audio and video files. These properties include `NaturalDuration`, `CurrentState`, `CanPause`, and `AutoPlay`. Additional properties are specific to the audio part of a media stream.

B.1.3 Audio-specific properties

The `MediaElement` exposes several properties directly linked to audio features. These features can be used to give users greater control over their listening experiences and to engulf your users in your Silverlight application. These features can be delivered through the `AudioStreamCount`, `AudioStreamIndex`, `Balance`, `IsMuted`, and `Volume` properties.

AUDIOSTREAMCOUNT/AUDIOSTREAMINDEX

Occasionally, audio or video files will contain more than one audio track. As an example, a song may have one track for the guitar, one for the drums, and one for the vocals. Usually, you won't work with these kinds of audio files. Instead, you may come across multilingual videos where each language has its own track. In both these situations, you can access the track-related information through the `AudioStreamCount` and `AudioStreamIndex` properties.

The `AudioStreamCount` and `AudioStreamIndex` properties give you access to the individual audio tracks of a media file. The read-only `AudioStreamCount` property stores the number of tracks available. The `AudioStreamIndex` property specifies which of the available tracks to play (or is playing). Neither property means anything until the `MediaOpened` event has fired.

When the `MediaOpened` event is fired, the `AudioStreamCount` and `AudioStreamIndex` properties are set on the client's machine. When this occurs, the audio tracks in the media file are read. While these tracks are being read, a collection is being created in the background. When this collection is fully created, the `AudioStreamCount` property is set to match the number of tracks in the collection. Then, the `AudioStreamIndex` property is set to begin using the first track in the collection. Alternatively, if the `AudioStreamIndex` property is set at design time, that track will be used. Either way, once an audio track is playing, it's important to make sure that the sound is balanced.

BALANCE

The `Balance` property enables you to effortlessly simulate sounds such as waves gently lapping a sandy shoreline or a swirling wind. These types of sounds often involve sound shifting from one ear to the other; it would be startling if the sounds spastically jumped from one ear to the other. The balance of the volume across your ears makes these sounds much more natural.

With the `Balance` property, you can gracefully spread out your sounds by specifying a double-precision value between -1.0 and 1.0 . If you set the property value to -1 , you can project sound entirely from the left-side speakers. If you set the value to 1 , the sound will leap from the right speakers. If you're seeking a balance between the left and right speakers, you set the value to 0 .

This property is more than an enumerator between the left, right, and center positions. It gives you the flexibility to do things like project 70 percent of a sound from the right speaker by using a value of 0.7 . The remaining 30 percent projects from the

left speaker. As you can imagine, you can easily depict a lifelike audible environment. Sometimes it's nice to shut out the sounds of life—enter `IsMuted`.

ISMUTED

Anything with an audio source should expose the ability to temporarily mute the audio. Thankfully, the `MediaElement` exposes an `IsMuted` property.

This property allows you to programmatically determine whether the sound associated with a `MediaElement` is audible. If a `MediaElement` is playing and this Boolean property is set to `true`, the `MediaElement` will continue to play, but it won't be audible.

As a `bool`, the `IsMuted` property is all or nothing. Usually, you'll need to find a happy medium between audible and inaudible. Silverlight also gives you this type of control through the `Volume` property.

VOLUME

The `Volume` property is a double-precision, floating-point value that specifies the audible level of a `MediaElement`. This property value can range from an inaudible (0.0) all the way up to a room-shaking 1.0. The room-shaking capabilities are ultimately restrained by the user's computer volume. By default, the `Volume` value is in the middle of this range at 0.5.

The `Volume` property is one of the five properties that address audio-related features. The other properties are the `IsMuted`, `Balance`, `AudioStreamCount`, and `AudioStreamIndex` properties. The `MediaElement` also exposes a pair of properties that are specific to the visual part of a media file.

B.1.4 Video-specific properties

The `MediaElement` exposes four properties directly related to videos. The first two are the `DroppedFramesPerSecond` and `RenderedFramesPerSecond` properties, both of which deal with video frame rates. The other two properties, `NaturalVideoHeight` and `NaturalVideoWidth`, deal with the dimensions of a video.

The `MediaElement` exposes two read-only double-precision values related to the frame rate of a video. `RenderedFramesPerSecond` gives you the number of frames that are rendered per second. The other property, `DroppedFramesPerSecond`, lets you know how many frames are being dropped per second. You can use these two properties to monitor the smoothness of a video. If a video begins to become jerky, the `DroppedFramesPerSecond` value will increase. In this scenario, you may want to consider using a video with smaller natural dimensions.

The natural dimensions of a video are provided through two read-only properties. The `NaturalVideoHeight` property represents the height of a video, and the `NaturalVideoWidth` property represents the video's width. These `int` properties are both read-only because they represent the original dimensions, in pixels, of a requested video. These values are useful when a video is the primary focus of your UI. If you're using an audio file instead of a video file, these two properties will stay at their default values of 0. For this reason, these properties are specific to video scenarios. Both video and audio files are involved in a standard lifecycle.

B.1.5 The lifecycle of a media file

Throughout this section, you've seen a wide variety of properties. Some of these property values are likely to change during the life of a media file, so it's beneficial to listen for those changes. As you might expect, the `MediaElement` provides a rich set of events that enables you to watch for those changes (see table B.4).

Table B.4 The events of the `MediaElement`

Event	Description
<code>BufferingProgressChanged</code>	Triggered anytime the <code>BufferingProgress</code> property changes.
<code>CurrentStateChanged</code>	Fired any time the <code>CurrentState</code> property is altered.
<code>DownloadProgressChanged</code>	Occurs whenever the <code>DownloadProgress</code> property changes.
<code>MarkerReached</code>	Discussed in section B.3.2.
<code>MediaEnded</code>	Fired when the <code>MediaElement</code> is no longer playing audio and video.
<code>MediaFailed</code>	Triggered if the media item referenced in the <code>Source</code> property can't be found. Alternatively, this event will trigger if there's a problem with the media file itself.
<code>MediaOpened</code>	Occurs after the information associated with the media has been read and the media stream has been validated and opened.

The table shows the events exposed by the `MediaElement`. Note that some state changes trigger multiple events. For instance, if a video file runs its route within a `MediaElement`, the `CurrentStateChanged` and `MediaEnded` events will both fire. As a result, you may need to create checks and balances within your code. To better understand the typical life of a media file, review figure B.2.

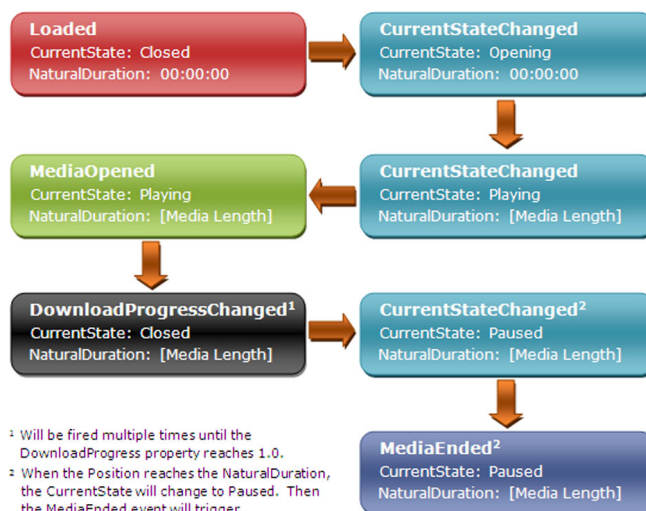


Figure B.2 The cycle of events as a media item plays progressively within a `MediaElement`

This figure shows the lifecycle of a media item that has played progressively through a `MediaElement`. The `Loaded` event used in the figure is of the `FrameworkElement` variety. This event shows when the `NaturalDuration` is set. As you can see, this property is set when the `CurrentState` is switched to `Playing`.

If you reference a media item that can't be found, the `MediaFailed` event will fire, but the `CurrentStateChanged` event won't be triggered. In other words, if you reference a media file that doesn't exist, only the `Loaded` and `MediaFailed` events will be triggered.

The events of the `MediaElement` reflect the lifecycle of a media item. This item can be impacted by a variety of audio- or video-related properties. Several properties are common to both audio and video files. One of these properties represents the `Source` of the media and can be referenced through a relative or remote `Uri`. Even more interesting is the fact that you can use the `Source` property to reference playlists.

B.2 *Playlists*

A playlist is a list of audio or video tracks arranged in a specific order. These lists give you a way to manage media elements that are part of a larger scheme such as a CD. Playlists are more than ordered media items, though. Playlists give you a way to generate revenue through advertising. Regardless of how you intend to use them, Silverlight has support for two playlist types.

In this section, you'll learn about the two types of playlists supported in Silverlight. The first kind of playlist, a client-side playlist, enables your Silverlight application to fully control interaction with the playlist. The other kind of playlist, a server-side playlist (SSPL), gives the hosting server complete control over the media experience.

B.2.1 *Understanding client-side playlists*

A client-side playlist is an XML file that can be interpreted by a `MediaElement`. This XML file follows a special format known as ASX, which you'll explore in a moment. Once this file has been parsed by a `MediaElement`, the `MediaElement` will decide whether to begin playing. This decision will be based on the `AutoPlay` property. If this property is set to `true`, each of the items in the client-side playlist will begin playing one after the other. Amazingly, all this happens naturally by pointing the `Source` property to an ASX file as shown here:

```
<MediaElement x:Name="VideoPlayer"
  Source="http://10rem.net/pub/sl5ia/Playlist.asx" />
```

This code shows how to request a client-side playlist. Note that this playlist uses the `.asx` file extension. This file extension is the one typically used for client-side playlists, but you can reference an ASX file with an extension of `.asx`, `.isx`, `.wax`, `.wvx`, `.wmx`, or `.wpl`. This restriction may seem odd considering that an ASX file is an XML file. Without this distinction, the `MediaElement` would be unable to quickly tell the difference between a client-side playlist and any of the other supported formats.

A client-side playlist can be an effective way to deliver multiple media tracks. To take advantage of client-side playlists, you must understand how to masterfully use

ASX files. These files can have rich descriptive information, known as *metadata*, surrounding each of the tracks.

USING ASX FILES

Client-side playlists are defined as *Advanced Stream Redirector* (ASX) files—this is just a fancy name for a specific XML format. Because this format is XML, you can create a client-side playlist with your favorite text editor, Windows Media Player, or server-side application. Regardless of your choice, this file will always follow a common structure, which is shown in this example:

```
<ASX Version="3.0">
  <Title>Pete's Tutorial Videos</Title>
  <Entry>
    <Title>Silverlight Introduction</Title>
    <Author>Pete Brown</Author>
    <Ref Href="http://10rem.net/pub/sl5ia/NetduinoRobot_SmallM.wmv" />
  </Entry>
  <Entry>
    <Title>Using Media with Silverlight</Title>

    <Author>Pete Brown</Author>
    <Ref Href="http://10rem.net/pub/sl5ia/NetduinoRobot_Large.wmv" />

  </Entry>
</ASX>
```

This example shows a pretty basic client-side playlist that uses a small portion of the full ASX schema. This segment isn't that far off from the full schema supported within Silverlight. Silverlight only supports a subset of the full ASX schema, but this subset still provides plenty of elements that can be used to deliver a rich client-side playlist.

Table B.5 The ASX elements supported within Silverlight

Element	Description
Abstract	Provides a description for a client-side playlist or an entry within the playlist. This element exposes an attribute called <code>Version</code> . This attribute should use the value 3.0 for Silverlight applications.
Asx	The root element of a client-side playlist.
Author	Specifies the name(s) of the individual(s) who created a client-side playlist or an entry within the playlist. Only one <code>Author</code> element can be used per <code>ASX</code> or <code>Entry</code> element.
Base	Represents a URL that will get prepended before playing within the client.
Copyright	States the copyright information for an <code>ASX</code> or <code>Entry</code> element.
Entry	Defines an item in a client-side playlist. This element provides a Boolean attribute called <code>ClientSkip</code> . This attribute can be used to prevent a user from skipping tracks.
MoreInfo	Enables you to specify a URL that provides more detailed information about the playlist or media item.
Param	Represents a custom parameter associated with a media item.

Table B.5 The ASX elements supported within Silverlight (continued)

Element	Description
Ref	This element is the item that specifies which file to refer to for a media clip. The <code>Ref</code> element exposes a single attribute called <code>Href</code> that points to the URL of a media clip.
Title	Signifies the moniker of a playlist or media item. For instance, if a playlist represents a CD, the <code>Title</code> element in that case would represent the name of the CD. The <code>Title</code> can also be used to specify the name of an individual track.

Table B.5 shows the ASX elements supported within Silverlight. As you can see, an ASX file is more than a list of URLs that point to media files. The ASX file format gives you the opportunity to provide a lot of valuable metadata with a playlist. In fact, the ASX format lets you specify metadata for the media items within the playlist, so it's important to understand how to access that metadata.

ACCESSING THE METADATA

The metadata for a media item can be found within a read-only property called `Attributes`. This member of the `MediaElement` class exposes the metadata as a `Dictionary<string, string>`. There are two interesting characteristics about this property that deserve mentioning.

The first is in regard to what metadata is exposed. Surprisingly, the metadata embedded within a media item isn't included. Unfortunately, there isn't an elegant way to get this information. The descriptive information stored within the ASX file *is* included, so if you're using client-side playlists you should provide as much metadata as you can.

The other interesting item is related to the lifecycle of the `Attributes` property. This property stores the metadata associated with an individual media item, so the `Attributes` property is cleared and repopulated each time a different track in an ASX file is started. If you're changing your UI based on the values within the `Attributes` property, you may consider doing this in the `MediaOpened` event. Alternatively, you may decide to bypass client-side playlists altogether and use a server-side playlist.

B.2.2 Using server-side playlists

Server-side playlists empower content administrators to dynamically determine what content is played and when. The server streaming the content has complete control over how the content is distributed. This approach provides several advantages over client-side playlists, including:

- *Lower bandwidth costs*—Generally client-side playlists serve content as separate streams for each entry. This causes your Silverlight application to reconnect to the server multiple times, wasting precious bandwidth. Because server-side playlists use a continuous stream, the Silverlight application only has to connect once.
- *Dynamic playlist creation*—Server-side playlists allow you to change a playlist even after a Silverlight application has connected.

To take advantage of these features, you must write a script using the Synchronized Multimedia Integration Language (SMIL). This script must be placed inside a file with the .wsx extension. As you’ve probably guessed, this file extension is used for server-side playlists. Once these server-side playlists are created, you can use a `MediaElement` to reference them.

CREATING WSX FILES

Server-side playlists are defined as WSX files. These files are XML files that follow a specific XML format, which is demonstrated in the following sample WSX file:

```
<?wsx version="1.0"?>
<smil>
  <seq id="sq1">
    <media id="advertisement1" src="advertisement1.wmv" />
    <media id="movie" src="NyanCatMovie.wmv" />
    <media id="advertisement2" src="advertisement2.wmv" />
  </seq>
</smil>
```

This XML example shows a basic WSX file. This playlist uses three of the elements supported by the SMIL format in Silverlight—`Media`, `Seq`, and `Smil`. Silverlight supports five elements, which are described table B.6.

Table B.6 The SMIL elements supported within Silverlight

Element	Description
Excl	“Exclusive.” A container for media items. These items can be played in any order, but only one will be played at a time.
Media	References an audio or video file through an <code>src</code> attribute.
Seq	“Sequential.” A container for media items. These items will be played in sequential order.
Smil	The root element for a server-side playlist.
Switch	A container for a series of items that can be interchanged if one of the items fails.

The elements listed in the table give a content administrator the flexibility to control how content is distributed. To distribute this content, you use a `MediaElement` to reference the WSX file.

REFERENCING SERVER-SIDE PLAYLISTS

After your WSX file has been created, you can publish it on your server. You must publish a server-side playlist before a Silverlight application can use it. Although publishing a server-side playlist is beyond the scope of this book, connecting to one isn’t. You can do this from a `MediaElement` as shown in this example (with a fictional URL):

```
<MediaElement Source="mms://www.silverlightinaction.com:1234/myPlaylist" />
```

This line of markup shows how to reference a server-side playlist from a `MediaElement`. You may have noticed that the playlist doesn't include the `.wsx` file extension. This extension usually gets removed during the publishing process. A `MediaElement` must use the MMS protocol to request a server-side playlist. This playlist can be used to stream content but can't be used to serve downloadable content in Silverlight.

Server-side playlists provide a way for content administrators to control the distribution of their content. Client-side playlists turn that control over to the requesting application. Either way, both options give you a way to distribute that web-based mix tape you've always wanted to send. Of course, playlists (and media players in general) aren't very useful without providing control over the playback.

B.3 *Interactive playback*

As you've seen up to this point, Silverlight makes it easy to deploy media content with the `MediaElement`. This content could come in the form of an individual media item or playlist. Regardless of where that media comes from, users generally want to control their own media experiences, and Silverlight makes it easy to make each experience an interactive one.

The interactive playback features of Silverlight enable you to interact with media in a variety of ways. Over the course of this section, you'll see three key items that can enhance a media experience. For starters, you'll see how to control the play state on the fly. Then, you'll learn about interacting with your users throughout the course of an audio or video file. Finally, you'll see how to take advantage of Silverlight's full-screen mode to deliver a memorable media experience.

B.3.1 *Controlling the play state*

The `MediaElement` gives you the ability to programmatically change the play state of a media item. This can be useful for providing things such as play, pause, and stop buttons. Note that you can't change the play state directly through the read-only `CurrentState` property; you must rely on three basic methods to control the momentum of a media item. These methods are part of the `MediaElement` class and are described in table B.7.

Table B.7 The methods that control the progress of a `MediaElement`

Method	Description
Play	Begins moving the <code>Position</code> of the <code>MediaElement</code> forward from wherever it's currently located. If you're 5 seconds into a video and you pause it, this method will start playing the video 5 seconds in. Calling this method will change the <code>CurrentState</code> property to <code>Playing</code> .
Pause	Halts the playback of a media item at the current <code>Position</code> . This method will change the <code>CurrentState</code> property to <code>Paused</code> .
Stop	Stops the downloading, buffering, and playback of a media item. In addition, this method resets the <code>Position</code> to the beginning of the media item. Calling this method changes the <code>CurrentState</code> property to <code>Stopped</code> .

The table shows the three methods that can be used to control the play state. These methods are fairly straightforward and hardly worth mentioning, but this section would be incomplete without them. You probably expected the ability to play and stop a media item before seeing this list. In addition, you probably expected the ability to pause an item, but you may not have anticipated the fact that pausing a media item isn't always an option.

The `Pause` method will only work if the `CanPause` property is set to `true`. This read-only property will be set to `true` if the user's machine has the ability to halt playback of a media file. Regardless of the user's machine, a streaming media file will always set the `CanPause` property to `false`. In these situations where the `CanPause` property is `false`, you can still call the `Pause` method—it just won't do anything.

Providing an interactive experience often involves controlling the play state. This ability enables users to send a message to the `MediaElement` about what they want. Significantly, the `MediaElement` lets you send something back to the user when you want. That's only partially true. You'll see what I mean as you learn about interacting with your users in a timely fashion.

B.3.2 Working with the timeline

The `MediaElement` enables you to interact with your users at specific points in time. This can be a great way to provide captions or subtitles in your videos. In addition, this feature enables you to deliver advertisements, or other types of information, that are relevant to a portion of a video. Regardless of your need, time-sensitive information can be bundled with your media in the form of a *timeline marker*.

A timeline marker is metadata that's relevant to a specific point in time. This information is generally part of a media file itself and is bundled during encoding. Significantly, there are two different kinds of timeline markers. The first is known as a *basic marker*. It's intended to be used when you need to provide fixed information. The other kind of timeline marker is a *script command*. It can be used to run a piece of code. Both kinds of markers will be represented as a `TimelineMarker` whose properties are shown in table B.8.

Table B.8 The properties associated with a `TimelineMarker`

Property	Description
Text	A value associated with marker. This <code>string</code> can be any value you want. You may want to think of this as the value of a parameter.
Time	The position of the marker within the media. This position is represented as a <code>TimeSpan</code> .
Type	This <code>string</code> exposes the kind of marker for a script command. If a basic marker is being used, this value will be <code>NAME</code> .

In general, the properties in table B.8 get populated when a `TimelineMarker` is created. `TimelineMarker` objects are usually created when a `MediaElement` initially reads a media file. During this process, the metadata within the header of the file is used to create `TimelineMarker` objects. These objects then are added to a publicly visible collection called `Markers`.

The `Markers` collection is a collection of timeline markers associated with a media file. The items associated with this collection can't be added through XAML, unlike the majority of other collections in Silverlight, because the markers come from the media item set as the `Source` of the owning `MediaElement`. Whenever one of these timeline-marker element's `Time` has come, the `MediaElement` will fire the `MarkerReached` event. This event provides an opportunity to recapture the data associated with a marker, which can be useful for any number of things, including showing a caption.

Listing B.2 Using the `MarkerReached` event to show a caption on a `MediaElement`

XAML:

```
<UserControl x:Class="MarkerExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot"
        Background="White">
        <TextBlock x:Name="tb"
            TextWrapping="Wrap" FontSize="30" Margin="10"
            HorizontalAlignment="Center" VerticalAlignment="Top" />
        <MediaElement x:Name="me" AutoPlay="True"
            Margin="0 100 0 0"
            Source="http://localhost:5150/Media/NetduinoRobot_SmallM.wmv"
            MarkerReached="me_MarkerReached" />
    </Grid>
</UserControl>
```

C#:

```
void me_MarkerReached(object sender, TimelineMarkerRoutedEventArgs e)
{
    tb.Text = e.Marker.Text;
}
```

← | **Displaying marker text**

The listing shows one way you can use the `MarkerReached` event. This event provides a `TimelineMarkerRoutedEventArgs` parameter that gives you access to the `TimelineMarker` that tripped the event. Common uses for this event are captioning, displaying ads (the text contains an ID or URL), text overlays, or displaying links to videos related to that marker. Many sites such as YouTube use similar functionality to display notes you add at specific points in the video.

Markers add a whole new level of interactivity to your media player. To support basic interaction, the `MediaElement` provides three simple methods that let you control the play state. Typically the user will control those via a mouse, but other input devices are becoming increasingly popular.

B.4 Responding to remote controls and media keyboards

Increasingly, consumers have more computing intelligence sitting in their TV rooms. Many have cable or satellite boxes, which are essentially purpose-built PCs. We have game consoles like the PlayStation and Xbox 360, which are powerful computers in their own right. Not surprisingly, many tech-savvy consumers even have dedicated PCs attached to their TVs, often including CableCard units to allow them access to the channels they're paying for.

One of the available accessories for the Xbox 360 is the Media Remote (or the older Universal Media Remote). This remote control looks very much like the ones you may use for your television set, Blu-ray Disc player, or other media device. Apple has a similar but more Spartan remote named the Apple Remote.

Finally, many current keyboards include dedicated media keys for volume control, play/pause, and more. I use a Microsoft Natural Ergonomic Keyboard 4000,² which includes dedicated keys for volume, mute, and play/pause. Collectively, I consider these functions “media commands.”

Because one of the single most popular uses for Silverlight on the PC is watching video, especially through commercial providers like Netflix, support for the remote control and media commands is a no-brainer.

B.4.1 The `MediaCommand` event

Like most input events as covered in chapter 9, the `MediaCommand` event is exposed by the base `UIElement` class. Behind the scenes, Silverlight maps into a single `MediaCommand` enumeration the various input information from the various types of media input devices. Table B.9 shows the supported media commands.

Table B.9 Supported media commands

Command	Typical use
Play	Begin or continue playing the media.
Pause	Temporarily pause the media.
TogglePlayPause	Many keyboards and remote controls have a single button for play and pause and simply toggle between functions each time you press the button.
Stop	Stop playing the media, and return to the menu or other appropriate state.
Record	Begin or end recording.

² Who the heck names these things? They must get paid by the character.

Table B.9 Supported media commands (*continued*)

Command	Typical use
FastForward	Move forward through the media timeline.
Rewind	Move backward through the media timeline.
NextTrack	Move to the next track or marker point in the media.
PreviousTrack	Move to the previous track or marker point in the media.
IncreaseVolume	Raise the volume on the device or player.
DecreaseVolume	Lower the volume on the device or player.
ChannelUp	Increment the channel number.
ChannelDown	Decrement the channel number.
MuteVolume	Mute all audio.

There are several other media commands (Menu, Title, Info) that aren't currently raised on the PC or Mac. Those commands are reserved for potential future support on Xbox only.

In addition to the keys and buttons explicitly handled by the `MediaCommand` enumeration, other common functions such as OK/Enter, numeric entry, navigation arrows, and others are handled by their equivalent keyboard keys, also covered in chapter 9.

About media key hooks and hijacking

On the typical modern computer, there are a lot of apps that try to steal the media keys. Typically these applications are already running in the background. On the Mac, that's iTunes. On the PC, you may have iTunes or Zune, Media Player, or something else.

In most cases, the only way to guarantee that you'll get the media commands you want is to have your application run full-screen and preferably out-of-browser.

If you find your app isn't getting the media commands you expect, try running it in full-screen mode. For the 10-foot experience, this approach typically makes sense anyway.

B.4.2 *MediaCommand and the MediaElement*

You aren't restricted to using the built-in `MediaCommand` enumeration and event with a media element. You could, for example, enable the use of these keys to control playing or pausing a game, or controlling a slide show. That said, the most common use is with media playback, so I'll focus our example on that scenario.

The next listing shows the UI of a barebones media application with a video player and a button to enable viewing full-screen. You may use any video source, but I'm

going to use the same 720p video mentioned earlier in this appendix. It shows the first robot I ever built using the Netduino.³

Listing B.3 XAML for the simple video player UI

```
<UserControl x:Class="MediaCommandExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    MediaCommand="OnMediaCommand"
    d:DesignHeight="300"
    d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <MediaElement x:Name="VideoPlayer"
            Source="http://10rem.net/pub/sl5ia/NetduinoRobot_SmallM.wmv" />

        <Button x:Name="FullScreen" Click="FullScreen_Click"
            HorizontalAlignment="Right" VerticalAlignment="Bottom"
            Content="Full Screen" Margin="5" />
    </Grid>
</UserControl>
```

← MediaCommand event

If you run the application with only the markup in the listing (and stub the generated event handlers referenced in the XAML), you'll see an app that plays a video as soon as it has been launched. The next step is to add in the full-screen functionality and the media command handling. Here's how to implement those features.

Listing B.4 MediaCommand handling in the code-behind

```
private void OnMediaCommand(object sender, MediaCommandEventArgs e)
{
    Debug.WriteLine(e.MediaCommand.ToString());

    switch (e.MediaCommand)
    {
        case System.Windows.Media.MediaCommand.Play:
            VideoPlayer.Play();
            break;

        case System.Windows.Media.MediaCommand.Pause:
            if (VideoPlayer.CanPause)
                VideoPlayer.Pause();
            break;

        case System.Windows.Media.MediaCommand.TogglePlayPause:
            if (VideoPlayer.CurrentState == MediaElementState.Paused ||
                VideoPlayer.CurrentState == MediaElementState.Stopped)
                VideoPlayer.Play();
    }
}
```

³ I, for one, welcome our new .NET Micro Framework–powered robot overlords.

```

        else if (VideoPlayer.CurrentState == MediaElementState.Playing)
            if (VideoPlayer.CanPause)
                VideoPlayer.Pause();
            break;

        case System.Windows.Media.MediaCommand.IncreaseVolume:
            VideoPlayer.Volume += .1;
            break;

        case System.Windows.Media.MediaCommand.DecreaseVolume:
            VideoPlayer.Volume -= .1;
            break;
    }
}

private void FullScreen_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Host.Content.IsFullScreen = true;
}

```

The listing uses the `MediaElement` methods and properties you’ve learned about earlier in this section combined with a small selection of the new `MediaCommand` enumeration. The end result is a media player that can be controlled entirely via your media keyboard or remote.

As mentioned in the sidebar, if you run into problems with the commands being hijacked by something else, be sure to hit the “full screen” button on the page.

Remote control and media keyboard support goes a long way toward making the 10-foot experience usable in Silverlight. Along with `MediaCommand` support, Silverlight 5 introduced another key media playback feature: trick play.

B.5 Supporting variable-speed playback

One of the more frustrating things about video learning is the problem of pace. Most of us can absorb what a speaker is saying even if they were to say it 1.5 or 2 times as fast as they talk on the video. I know that I, for one, tend to speak more slowly and clearly in video so I don’t lose anyone, especially beginners. For accomplished developers, it can seem like I’m channeling Mister Rogers on Valium. The solution, of course, is to speed up the video.

One unfortunate side effect of speeding up video is the speaker starts to sound like Simon Chipmunk (or maybe Alvin, but let’s face it, it’s probably Simon). The ability to understand the spoken audio on the video decreases as the pitch goes up or down, outside the typical human speech range.

A solution to this problem is called variable speed playback, or “trick play.” Trick play allows you to alter the speed of the video while maintaining the pitch of the audio track.

Trick play is a feature of the variable rate playback support. Although you can set a playback rate from -32x to +32x normal speed, you’ll only get pitch correction if the values are within the 0.5x to 2x range. Outside that range, the capability functions as a normal rewind or fast-forward feature.

B.5.1 Fast-forward and rewind

Just about every media device invented includes the ability to fast-forward or rewind the media. In the past, you had to implement this capability by seeking via an offset from the current position. That still works, but there's another way to accomplish this as part of the `Play` functionality.

Let's first look at the fast-forward and rewind functionality. The next listing shows the XAML for a simple media player with several buttons and a progress bar. Let's concentrate on the Play, Fast-Forward, and Rewind buttons and the progress bar for the moment.

Listing B.5 Media player XAML UI

```
<UserControl x:Class="VariableSpeedPlayback.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.Resources>
            <Style TargetType="Button">
                <Setter Property="Margin" Value="3" />
            </Style>
        </Grid.Resources>
        <MediaElement x:Name="VideoPlayer" AutoPlay="False"
            Source="http://10rem.net/pub/sl5ia/NetduinoRobot_SmallM.wmv"
            MediaOpened="VideoPlayer_MediaOpened"
            MediaEnded="VideoPlayer_MediaEnded"
            RateChanged="VideoPlayer_RateChanged" />
        <Grid HorizontalAlignment="Stretch" VerticalAlignment="Bottom"
            Background="#77000000" Margin="5">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <TextBlock x:Name="CurrentRateDisplay" Grid.Row="0"
                Foreground="White" FontSize="12" />

            <ProgressBar Grid.Row="0" x:Name="MediaPosition"
                Height="6" Margin="30,0,0,0"
                HorizontalAlignment="Stretch" />
            <StackPanel Orientation="Horizontal" Grid.Row="1"
                HorizontalAlignment="Center">
                <Button x:Name="Rewind" Click="Rewind_Click"
                    Content="&lt;&lt;" />
                <Button x:Name="PlayPause" Click="PlayPause_Click"
                    Content="> ||" />
                <Button x:Name="FastForward" Click="FastForward_Click"
                    Content="&gt;&gt;" />
                <Button x:Name="FastPlay05X" Click="FastPlay05X_Click" />
            </StackPanel>
        </Grid>
    </Grid>
</UserControl>
```

```

        Content="0.5 x" />
        <Button x:Name="FastPlay10X" Click="FastPlay10X_Click"
            Content="1.0 x" />
        <Button x:Name="FastPlay15X" Click="FastPlay15X_Click"
            Content="1.5 x" />
        <Button x:Name="FastPlay20X" Click="FastPlay20X_Click"
            Content="2.0 x" />
    </StackPanel>
</Grid>
</Grid>
</UserControl>

```

The listing includes all the elements required to demonstrate variable speed playback. The keyword is *playback*. Rewind and Fast-Forward are implemented as play operations—just at different speeds. At first that may not seem intuitive, especially if you expected some sort of dedicated *Rewind* or *FastForward* method on the *MediaElement*. But on closer inspection they make perfect sense as you want the video to play during those operations.

In addition to the UI, you'll add in a bit of code-behind, shown next, to support play/pause and the fast-forward and rewind functionality that's the real meat of this section.

Listing B.6 Code-behind for the variable-speed playback media player

```

public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void EnsureVideoPlaying()
    {
        if (VideoPlayer.CurrentState == MediaElementState.Paused ||
            VideoPlayer.CurrentState == MediaElementState.Stopped)
            VideoPlayer.Play();

        if (!_scrubTimer.IsEnabled)
            _scrubTimer.Start();
    }

    private void PlayPause_Click(object sender, RoutedEventArgs e)
    {
        if (VideoPlayer.PlaybackRate > 2.0 || VideoPlayer.PlaybackRate < 0.5)
        {
            VideoPlayer.PlaybackRate = 1.0;
            EnsureVideoPlaying();
        }
        else if (VideoPlayer.CurrentState == MediaElementState.Paused ||
            VideoPlayer.CurrentState == MediaElementState.Stopped)
        {
            VideoPlayer.Play();
        }
    }
}

```

← **Reset rate**

```

        else if (VideoPlayer.CanPause)
            VideoPlayer.Pause();
    }

    private void Rewind_Click(object sender, RoutedEventArgs e)
    {
        VideoPlayer.PlaybackRate = -5;
        EnsureVideoPlaying();
    }

    private void FastForward_Click(object sender, RoutedEventArgs e)
    {
        VideoPlayer.PlaybackRate = 5;
        EnsureVideoPlaying();
    }

    private void FastPlay05X_Click(object sender, RoutedEventArgs e){}
    private void FastPlay10X_Click(object sender, RoutedEventArgs e){}
    private void FastPlay15X_Click(object sender, RoutedEventArgs e){}
    private void FastPlay20X_Click(object sender, RoutedEventArgs e){}

    private DispatcherTimer _scrubTimer = new DispatcherTimer();

    private void VideoPlayer_MediaEnded(
        object sender, RoutedEventArgs e)
    {
        VideoPlayer.Position = TimeSpan.FromSeconds(0.0);
        VideoPlayer.PlaybackRate = 1.0;
    }

    private void VideoPlayer_RateChanged(object sender,
        RateChangedRoutedEventArgs e)
    {
        CurrentRateDisplay.Text = e.NewRate.ToString();
    }

    private void VideoPlayer_MediaOpened(object sender, RoutedEventArgs e)
    {
        MediaPosition.Minimum = 0.0;
        MediaPosition.Maximum =
            VideoPlayer.NaturalDuration.TimeSpan.TotalMilliseconds;

        _scrubTimer.Interval = TimeSpan.FromSeconds(0.25);
        _scrubTimer.Tick += (s, ea) =>
        {
            MediaPosition.Value =
                VideoPlayer.Position.TotalMilliseconds;
        };

        _scrubTimer.Start();
    }
}

```

← **Rewind**

← **Fast-forward**

← **Display rate using RateChanged**

← **Current position**

Note that you'll need to stub out the `FastPlayRRX_Click` event handlers as shown in the listing so the example will compile.

Introduced in the previous listing is the `RateChanged` event of the `MediaElement`. This event is fired whenever the playback rate of the `MediaElement` is changed. Typically this will be caused by your own code, but it's possible you won't be able to change the rate because the media doesn't support it, or you've already hit the minimum or maximum value or have tried to exceed it. A best practice is to use this event for any actions that depend on the actual playback rate.

Of interest is the `ProgressBar`-based scrub⁴ bar that's used to show the current position in the media. This is helpful when working with the fast-forward and rewind features. I implemented this element using a `DispatcherTimer`, which reports the current progress of the video every quarter of a second. This is required because you can't (and wouldn't want to) bind anything to the `Position` property of the `MediaElement` due to the very high frequency with which it may change.

Using listing B.6, you should be able to fast-forward or rewind the video. Depending on your video card and the speed you've chosen, the playback during those operations may be choppy, so look at the scrub bar to see how your choice has affected the position.

So far you've worked with silent forwarding and rewinding. What happens when you want to hear the audio? In that case, you'll need to use trick play.

B.5.2 Using trick play

I like to play around with synthesizers and computer-generated audio, and even built a simple algorithm-generated audio synthesizer in Silverlight 3. Part of the requirement for any real work there is at least a basic understanding of how the frequency of an audio waveform corresponds to what you perceive as pitch.

AN AUDIO PRIMER

All sounds are made of waves with a specific height (amplitude) and a specific distance from the start to end of the wave, called the wavelength. The distance can also be thought of in terms of how many full waveform cycles will fit within a single second; this is called the frequency of the wave.

Let's take a simple sine wave. Figure B.3 shows a single cycle as well as a number of cycles over a time period.

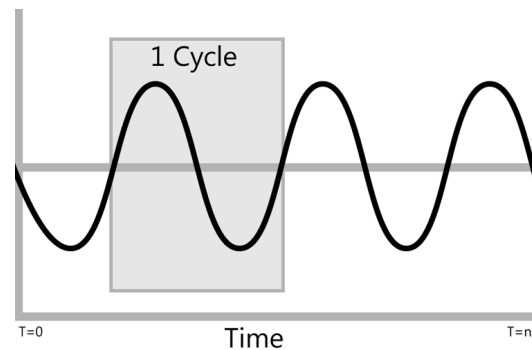


Figure B.3 A sound wave showing a frequency of three cycles over one time unit. A single cycle is highlighted in the gray box.

⁴ Not technically a scrub bar because you can't actually scrub. But the scrub bar is the progress bar-like thing that shows your current position in the video. Scrubbing is the process of using the thumb on that bar to move backward and/or forward through the video.

In figure B.3, you can see that the frequency over the theoretical time period is three. You can see three complete cycles in the image. If $T=n$ was the one-second mark, you'd say the frequency is 3 hertz, or 3Hz. If you were to compress that waveform (or remove vertical slices as is done with digital audio) along the time axis in order to shorten the wavelength, you might see something like figure B.4.

This figure shows a version of the sound wave that completes five cycles in the time period. This sound will sound higher in pitch than the previous wave. It won't sound exactly 1.5 times higher in pitch as you might assume, because the pitch is a logarithmic scale, doubling for each octave in pitch. For example, assuming note A4 is 440.00Hz (some assume 440.1, but that's not important here) the wavelength and frequency values for the next octave of notes are shown in table B.10.

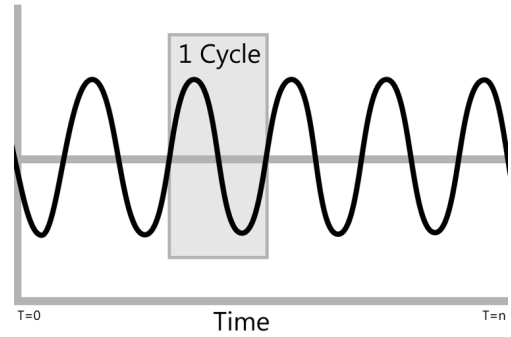


Figure B.4 The same sound wave, sped up so it has a faster frequency (five cycles per time unit), and therefore a higher perceived pitch

Table B.10 Notes from A4 to A5, showing their frequency (cycles per second) and the physical size of the sound wave

Note name	Frequency in Hz	Wavelength in cm
A4	440.00	78.4
A# 4 / Bb4	466.16	74
B4	493.88	69.9
C5	523.25	65.9
C#5 / Db5	554.37	62.2
D5	587.33	58.7
D#5 / Eb5	622.25	55.4
E5	659.26	52.3
F5	698.46	49.4
F#5 / Gb5	739.99	46.6
G5	783.99	44.0
G#5 / Ab5	830.61	41.5
A5	880.00	39.2

You can see that to go up one octave on a keyboard (12 notes), the frequency doubles. So, if you were to speed up a video 2x, you could expect the sound in the video to not only double in speed but also go up a complete octave.

All of this is to show that if you speed up an audio wave, completing more cycles in the same amount of time and shortening the wavelength, the pitch of the audio will increase. If you slow the same wave down, you'll hear a lower pitch. That's why we get chipmunk and demon effects from speeding up or slowing down audio.

Unlike the old analog effect you got from playing a 45 at 33 1/3 RPM or vice versa, digital technologies handle this speeding up or slowing down by removing or extending/adding samples. When speeding up audio, think of it more like removing vertical slices out of the wave so that you can fit it into the required time period. The fewer samples you start with, the fewer samples you have to play with, and the more digital or robotic the result will sound.

TRICK PLAY

Technologies like trick play in Silverlight manage this addition or removal of slices in such a way as to maintain the original frequency or pitch of the audio as it would be delivered. The technology has to do its best to make sure that it removes appropriate parts of the waves without losing the high and low points that characterize the sound and set the amplitude and perceived volume. There are also other considerations; typical audio is made up of a complex combination of many frequencies resulting in speech and other sounds.

Now that you fully understand the problem, you can appreciate the complexity of solving it. Silverlight handles all this sound manipulation behind the scenes whenever you play audio from between .5x to 2.0x its normal speed. Outside of those ranges, Silverlight does nothing with the audio because it simply isn't worth the computing power to try to make those sound reasonable. You'll find that, for most speech, 1.5x is great and 2.0x is fine if you're really paying attention; 0.5x is great for slow motion with intelligible audio.

That said, I'll let you decide for yourself. The next bit of code builds on the previous listings by adding in support for 0.5x, 1.0x, 1.5x, and 2.0x audio and video playback rates.

Listing B.7 Supporting trick play

```
private void FastPlay05X_Click(object sender, RoutedEventArgs e)
{
    MediaPlayer.PlaybackRate = 0.5;
    EnsureVideoPlaying();
}

private void FastPlay10X_Click(object sender, RoutedEventArgs e)
{
    MediaPlayer.PlaybackRate = 1.0;
    EnsureVideoPlaying();
}
```

← Normal speed

```
private void FastPlay15X_Click(object sender, RoutedEventArgs e)
{
    MediaPlayer.PlaybackRate = 1.5;
    EnsureVideoPlaying();
}

private void FastPlay20X_Click(object sender, RoutedEventArgs e)
{
    MediaPlayer.PlaybackRate = 2.0;
    EnsureVideoPlaying();
}
```

Run the example and click the 2.0x button. Notice how the audio is fast but intelligible. Now try 1.5x. At that speed, it's barely perceptible, but you save 15 minutes off an hour-long video! That time adds up to real savings.

Fast-forward and rewind are expected features of any media player. Implementing those features is easy to do with the `PlaybackRate` property. You can support any value from -32x to +32x, enabling you to add a wide range of forward and reverse capabilities.

Trick play is a real bonus feature in Silverlight media. Now that I've used it, I can't imagine watching tutorial videos without it. I only wish our Silverlight-based standardized ethics and other video training at Microsoft had this built in. Presumably it will in the near future.

B.6 Using protected content

The interactive playback features within Silverlight can be used to give your users an engaging media experience. Features like the remote control and variable speed playback are important to supporting the big media guys. Those same companies need to be able control who has access to this experience using purchases, subscriptions, or other mechanisms. To enable this, Silverlight has built-in support for a client-access Digital Rights Management (DRM) technology known as PlayReady for Silverlight.

PlayReady for Silverlight is a content-access technology that enables you to protect your media assets. These assets may be requested from a Silverlight application through a `MediaElement` instance. This control's `Source` property can be used to request protected content from a hosting server. In this section, you'll see how Silverlight uses PlayReady technology. This overview includes requesting protected content, retrieving PlayReady components, and unlocking protected content.

B.6.1 Requesting protected content

A Silverlight application can request protected content, which may be in the form of a protected stream or media file. This item can be requested through the `Source` property of a `MediaElement`, so it's safe to say that there's no difference on the client side between requesting protected and unprotected content. In fact, Silverlight doesn't know if content is protected until it's downloaded. This download happens naturally when a request is made, as shown in figure B.5.

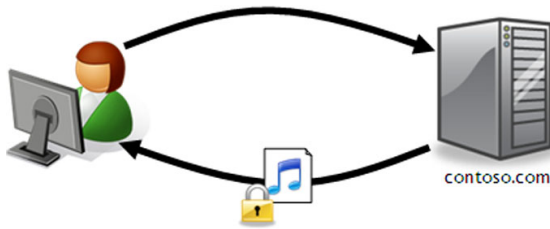


Figure B.5 A user requests protected content from a server. This content is downloaded, in encrypted format, to the Silverlight application.

This figure shows the general idea of requesting protected content from a fictional domain. After this request is made, the server will send an encrypted version of the protected file back to the Silverlight application. This file will have a special header that tells the Silverlight runtime that it's a protected file. This header will provide the location of the licensing server to Silverlight. But before the licensing server can be reached, Silverlight must ensure that the user has the necessary PlayReady components installed.

B.6.2 Retrieving the PlayReady components

By default, Silverlight has the infrastructure for PlayReady, but the PlayReady components aren't installed along with the Silverlight runtime. Instead, they're automatically downloaded and installed when a user requests a protected item. During this one-time installation process, Silverlight goes to the Microsoft website and grabs the necessary components.

Figure B.6 shows how the content access components are retrieved. These components may be customized for a user's machine, solely for the sake of ensuring a robust licensing experience. The user's machine is sometimes referred to as an *individualized DRM client*. This process happens automatically behind the scenes—you don't have to do a thing. Even after the PlayReady components have been installed, the content is still locked. To unlock this content, a request must be made to the licensing server.

B.6.3 Unlocking protected content

Once a protected item has been downloaded to your Silverlight application, it's still encrypted. This encryption can only be unlocked by a key sent from a licensing server, so if you try to play an encrypted file, Silverlight will search the encrypted file's header for the location of a licensing server. Silverlight will use this location to automatically request a key from the licensing server to decrypt the protected content (figure B.7).



Figure B.6 The process of installing the content access components. This one-time process happens the first time a user attempts to use a protected item. Future attempts to access protected content won't go through the process of downloading and installing PlayReady.



Figure B.7 The media content in this figure is locked until a key is retrieved from the licensing server. This server can implement custom logic through the PlayReady SDK.

When a licensing server retrieves a request for a key, it can either accept or deny the request. The licensing server can be used to implement some custom logic to make that decision. This custom logic must be implemented using the server-side PlayReady SDK. Unfortunately, this SDK is outside the scope of this book, but you can probably imagine how it could be used in a key request.

Figure B.7 shows what the request for a content-access key looks like. If this request is accepted, the licensing server will return a key which will unlock the protected content and begin playing it within the requesting `MediaElement`. If the request is denied, a key won't be returned. Instead, the requesting `MediaElement` will raise a `MediaFailed` event.

Silverlight has built-in support for the PlayReady content-access technology, which works behind the scenes to retrieve and unlock protected content—audio and video. One of the easiest ways to use PlayReady DRM and support HD video is to use the Microsoft Media Platform Player Framework.

B.7 Using the Microsoft Media Platform Player Framework

The Microsoft Media Platform: Player Framework (formerly known as the Silverlight Media Framework, or SMF)⁵ and now generally referred to as the MMP Player Framework or “that smooth streaming stuff,” is Microsoft’s open source, scalable, and customizable media player for IIS Smooth Streaming. Like IIS Smooth Streaming itself, its history dates to the Olympics video player and massive amounts of high-quality, protected video that needed to be served up in real time during the event. It has since evolved into an excellent multipurpose media player. Despite the technology rename, you’ll find the majority of the namespaces, templates, and elements still use the acronym SMF.

If you’re building an HD media player, evaluating this framework should be at the top of your task list. Key features of the framework include:

- Support for IIS Smooth Streaming with bit rate monitoring, as well as progressive download and Windows Media Streaming
- Modular, supporting plug-ins
- Support for popular ad standards
- Full styling support
- Stereoscopic 3D support
- A fully functional player right out of the box

⁵ There we go again with the huge product names.

The framework supports much more than that, of course, but those are the top compelling features. It has multiple points of extensibility, and if those aren't enough, full source code is provided.

In this section, you'll first look at what it takes to get the appropriate libraries for the MMP Player, then build a simple player that supports IIS Smooth Streaming.

B.7.1 Using the player libraries

You can get the MMP Player version 2.5 at <http://smf.codeplex.com>. The downloads include the binaries and the full source code as well as a convenient MSI package. Also, as with other CodePlex projects, you can browse the full source code right on the site or download it as part of a release. Be sure to get the latest version, which as of this writing is version 2.5. Don't bother with the older version 1. Significant changes were made after the first version.

If you use the MSI installer, the IIS Smooth Streaming Client libraries are installed automatically. If you're doing a manual install, first download and install the IIS Smooth Streaming Client player SDK using the Web Platform Installer at www.iis.net/download/smoothclient. If the WebPI (Web Platform Installer) doesn't work for you, there's a link right below it for downloading the MSI directly.

Next, download the MMP Player Framework v2.5 release (or the latest release available at the time you're reading this) and install that on your machine. If you use the MSI to install the libraries for Silverlight, you can skip this next step. You could use NuGet as well, if you'd like, but you won't get the convenient project templates if you go that route or the manual installation route.

If manually installing the libraries, unzip them and place them in a common location (but not a system folder such as Program Files) that you'll easily find from within Visual Studio. Be sure to unblock the files individually per this KB article so you can use them: <http://go.microsoft.com/fwlink/?LinkId=179545>. Figure B.8 shows the dialog with the Unblock button.

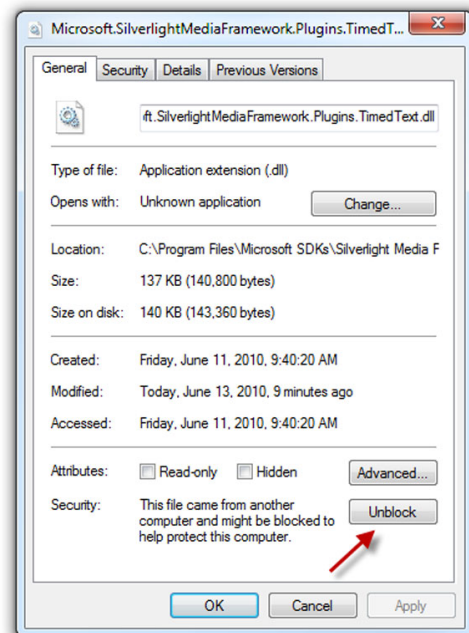


Figure B.8 Unblocking an internet-downloaded DLL in order to be able to reference it from within a Visual Studio project. You only need to do this if you manually install the bits.

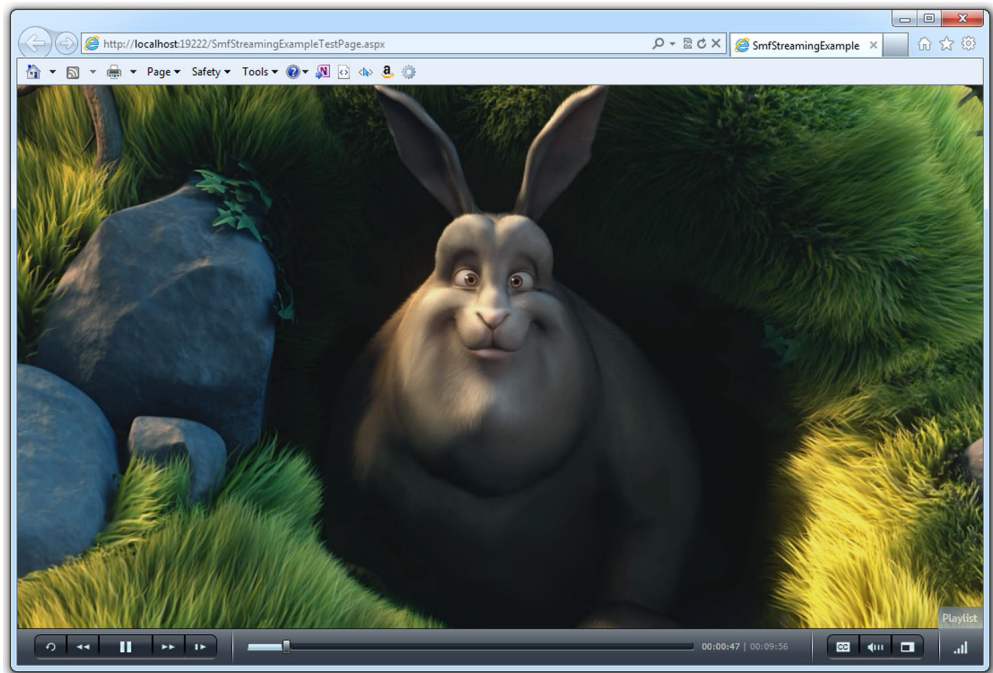


Figure B.9 The default SMF media player with Big Buck Bunny, an IIS Smooth Streaming video, loaded. From the looks of things, the video isn't the only thing that's loaded here.

B.7.2 Creating the player

Once you have everything installed and unblocked, creating a complete media player experience is as simple as starting from a SMF template or referencing the SMF DLLs and creating an instance of the player in XAML. Figure B.9 shows the default player appearance.

Listing B.8 shows how to instantiate the player from XAML. There are a few key namespaces to keep in mind for Smooth Streaming projects. Under the `Microsoft.SilverlightMediaFramework` namespace are the `.Core`, `.Plugins`, and `.Utilities` namespaces and their associated assemblies. Be sure to reference them for all types of SMF projects. For regular Smooth Streaming, there's the `Microsoft.Web.Media.SmoothStreaming.dll` assembly. For progressive download projects, use the `Microsoft.SilverlightMediaFramework.Plugins.Progressive.dll` assembly instead.

Listing B.8 Instantiating the SMF Player from XAML

```
<UserControl x:Class="SmfStreamingExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400"
```



```

xmlns:smf="http://schemas.microsoft.com/smf/2010/xaml/player"
xmlns:media="clr-namespace:Microsoft.SilverlightMediaFramework.Core.Media;
➡ assembly=Microsoft.SilverlightMediaFramework.Core">

```

```

<Grid x:Name="LayoutRoot"
      Background="White">
  <smf:SMFPlayer>
    <smf:SMFPlayer.Playlist>
      <media:PlaylistItem DeliveryMethod="AdaptiveStreaming"
        MediaSource="...long url to video manifest..." />
    </smf:SMFPlayer.Playlist>
  </smf:SMFPlayer>
</Grid>
</UserControl>

```

← MMP Player

The MMP Player requires two namespaces to be included. The first, `smf`, is for the player itself. The second, `media`, is for the playlists and features related to the media supported in the player. Due to the flexibility of the player, loading media takes a few more lines than the usual `MediaElement`. In particular, the player supports a playlist with one or more playlist items queued in it. Each playlist item includes a single piece of media with a specified delivery method. The valid values for `DeliveryMethod` are shown in table B.11.

Table B.11 Possible values for `DeliveryMethod` for the SMF player

Value	Description
NotSpecified	The default value. This will attempt to use the first media plug-in loaded. Because this can be unreliable in players that support more than one type of media delivery method, always specify one of the following values.
AdaptiveStreaming	The player will use IIS Smooth Streaming.
ProgressiveDownload	The player will use a progressive download approach for playing the media. This approach requires no server-side support.
Streaming	The player will use Windows Media Streaming to play the media.

It's important to know that the delivery methods supported are entirely controlled by what plug-ins you package with your Silverlight application. If you leave out the Progressive Download plug-in, for example, your player won't support that delivery method. Though not required, the easiest way to manage this is to figure out what you want in advance and use the appropriate project template.

The Microsoft Media Platform Player Framework is an excellent way to get a fully functional and feature-rich player up and running in a minimum amount of time. It's perfect for traditional video and audio.

B.8 Summary

Silverlight has excellent support for media, especially HD video. When I wrote this chapter for the first edition, HTML was just starting to catch up with the basic HD

video approach and didn't yet support the Smooth Streaming or, in most cases, trick play features. My, how far we've come in a couple years! That said, plug-ins like Silverlight still provide a superior media experience if the idea of installing a plug-in isn't an issue with your customers.

When creating a media player in Silverlight, you have two main choices: create the player from scratch using the `MediaElement` or use the Microsoft Media Platform Player framework to get you 90 percent of the way there and simply style and/or augment as necessary. In all cases, you'll be able to work with playlists and protected (DRM) media.

When you're in doubt, I highly recommend going with the MMP Player; it's extremely flexible and supports just about everything you need.

As of this writing, the MMP Player doesn't support the new variable speed playback and trick play features of Silverlight. I expect the player framework to catch up and add those features in the near future, if not already by the time you read this.

Another new feature in Silverlight 5 that's not yet in the MMP Player framework is `MediaCommand` support for remote controls and media keys on newer keyboards. That makes it possible to implement a 10-foot experience with Silverlight. You saw in this appendix that the code to support `MediaCommand` is simple, so it could easily be added to any current player or any player based on the MMP Player framework.

appendix C: *Raw media, webcam, and microphone*

In appendix B you learned how to play audio and video using the `MediaElement` and the enhanced tools that build on it. Typically, the `MediaElement` pulls from an MP3, WMA, or H.264 file on the web; it can also be streamed media in real time.

In web and desktop applications, audio and video don't always come from some known URL. Sometimes they come from within—within the code, that is. Silverlight enables you to create media using just algorithms that generate the audio samples and the video frames. I've used this to create a synthesizer, an animated (but slow) Mandelbrot fractal movie, and even a Commodore 64 emulator.

The Silverlight team originally planned on raw media support not as a way to create computer-generated media, but as a way to enable you to create your own decoders for media types not natively supported in Silverlight. That doesn't mean it's limited to that. No, it's much more powerful. The `MediaStreamSource` API is your hook into the media pipeline, and as long as you give it bits in a known format, the sky is the limit.

Another interesting use of media is as sound effects for kiosk applications and games. Silverlight has struggled with low-latency sound in the past. The addition of the `SoundEffect` and `SoundEffectInstance` classes from XNA have enabled you to have near real-time sound that responds quickly to user events or other code without the ceremony and, quite frankly, lag that the `MediaElement` approach uses.

One final place where media can come from is you and the camera staring at you whenever you sit down at your computer. It's getting hard to find laptops without integrated webcams, and many people have USB webcams on their desktop computers. Silverlight lets you take advantage of that peripheral to capture audio

and video and even still-frame snapshots. This has been put to good use in HR applications and in social media apps like the Facebook client.

I find the raw media approach exciting and extremely flexible, so let's start there.

C.1 Working with raw media

Silverlight has a strong but finite set of codecs it natively supports for audio and video playback. If you wanted to use a format not natively supported, such as the WAV audio file format or the AVI video format, that wasn't an option until the `MediaStreamSource` API was added.

The `MediaStreamSource` API was included in Silverlight 2, but that version required you to transcode into one of the WMV, WMA, or MP3 formats natively supported by Silverlight. In Silverlight 3, the MSS API was augmented to support raw media formats where you sent the raw pixels or audio samples directly through the rest of the pipeline. This made its use much easier, as it required knowledge only of the format you wanted to decode. For the same reason, it ran faster, because an extra potentially CPU-intensive encoding step was avoided.

The `MediaStreamSource` API supports simultaneous video and audio streams. In this section, you'll learn about creating raw video as well as raw audio. In both cases, you'll use algorithmically derived data to drive the raw media pipeline.

C.1.1 A custom `MediaStreamSource` class

To implement your own custom stream source, derive a class from `MediaStreamSource`. As the name suggests, this class will be used as the source for a `MediaElement` on the page. Table C.1 shows that `MediaStreamSource` has several methods that you must override in your implementation in order for your custom source to work properly.

Table C.1 `MediaStreamSource` virtual methodsAppC.fm

Method	Description
<code>SeekAsync</code>	Sets the next position to be used in <code>GetSampleAsync</code> . Call <code>ReportSeekCompleted</code> when done.
<code>GetDiagnosticAsync</code>	Used to return diagnostic information. This method can be a no-op because it's not critical. If used, call <code>ReportGetDiagnosticCompleted</code> when done.
<code>SwitchMediaStreamAsync</code>	Used to change between configured media streams. This method can be a no-op because it's not critical. If used, call <code>ReportSwitchMediaStreamCompleted</code> when done.
<code>GetSampleAsync</code>	Required. Get the next sample and return it using <code>ReportGetSampleCompleted</code> . If there's any delay, call <code>ReportGetSampleProgress</code> to indicate buffering.
<code>OpenMediaAsync</code>	Required. Set up the metadata for the media and call <code>ReportOpenMediaCompleted</code> .
<code>CloseMedia</code>	Any shutdown and cleanup code should go here.

One thing you'll notice about the functions is that many of them are asynchronous. The pattern followed in those methods is to perform the processing, then call a `ReportComplete` method, the name of which varies by task, when finished.

The asynchronous nature of the API helps keep performance up and stops your code from slowing down media playback.

The following listing shows the skeleton of a `MediaStreamSource` implementation, including the methods I just described. You'll continue to build on this throughout the remaining raw media sections.

Listing C.1 The basic `MediaStreamSource` structure

```
public class CustomSource : MediaStreamSource
{
    private long _currentTime = 0;
    protected override void SeekAsync(long seekToTime)
    {
        _currentTime = seekToTime;
        ReportSeekCompleted(seekToTime);
    }
    protected override void GetDiagnosticAsync(
        MediaStreamSourceDiagnosticKind diagnosticKind)
    {
        throw new NotImplementedException();
    }
    protected override void SwitchMediaStreamAsync(
        MediaStreamDescription mediaStreamDescription)
    {
        throw new NotImplementedException();
    }
    protected override void GetSampleAsync(
        MediaStreamType mediaStreamType)
    {
        if (mediaStreamType == MediaStreamType.Audio)
            GetAudioSample();
        else if (mediaStreamType == MediaStreamType.Video)
            GetVideoSample();
    }
    protected override void OpenMediaAsync() { }

    protected override void CloseMedia() { }
    private void GetAudioSample() { }
    private void GetVideoSample() { }
}
```

No-op methods

GetSampleAsync

The most important methods for this scenario are the `OpenMediaAsync` method and the two `GetAudioSample` and `GetVideoSample` methods used to send the next sample to the runtime. Those two methods are called from the `GetSampleAsync` method whenever an audio or video sample is requested, but they're not part of the required interface.

Once you have the `CustomSource` class created, you'll need to use it as the source for a `MediaElement` on a Silverlight page. The next example shows how to wire this up using XAML for the UI and C# code for the wire-up.

Listing C.2 Using a custom `MediaStreamSource` class

XAML:

```
<Grid x:Name="LayoutRoot" Background="White">
  <MediaElement x:Name="MediaPlayer"
    AutoPlay="True"
    Stretch="Uniform"
    Margin="10" />
</Grid>
```

C#:

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }
    CustomSource _mediaSource = new CustomSource();
    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        MediaPlayer.SetSource(_mediaSource);
    }
}
```

Custom
MediaStreamSource

In this listing, you first create a `MediaElement` that will span the size of the page, and assign the `CustomSource` instance to the source property using the `SetSource` method of the `MediaElement`. Once that's completed, the `MediaElement` is set to play and will start requesting samples from the `CustomSource` class.

Right now, your `CustomSource` class doesn't return any samples, so running the application would show nothing. You'll modify the class to return both video and audio, starting with video.

C.1.2 Creating raw video

Being able to create video from raw bits is pretty exciting—it opens up all sorts of scenarios, from bitmap-based animation to custom video codecs. I first played with raw video when I created my Silverlight Commodore 64 emulator. I tried a few different video presentation approaches before I settled on generating the video display in real time as a 50 fps (frames per second) `MediaStreamSource` video at Commodore-accurate 320 x 200.

For this video example, you're going to generate white noise, much like you see on an analog TV when the signal is lost. When complete, the application will look like figure C.1. If you lived in the United States prior to cable TV, this is what you saw after the national anthem finished playing. If the Commodore 64 references didn't clue you in, this surely does. Yes, I'm old.

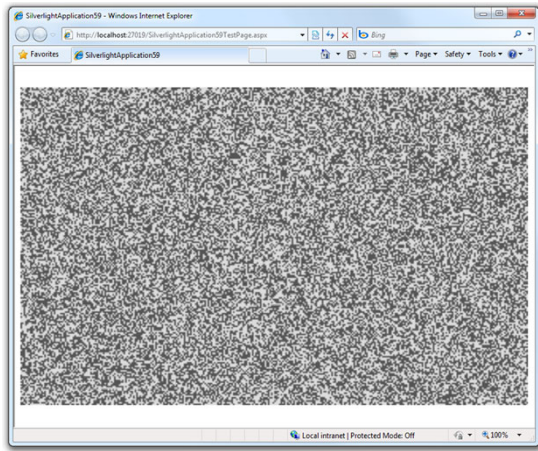


Figure C.1 The completed white noise video generator. When I was a boy, I used to imagine I was watching an epic ant battle from high overhead. Well, until I saw *Poltergeist*, which forever changed the nature of white noise on the TV.

Let's start with the logic required to set up the video stream and follow up with the code that returns the individual frame samples.

SETTING UP THE VIDEO STREAM

When creating raw video, the first step is to set up the video stream parameters. The parameters include things such as the height and width of the frame, the number of frames per second, and the actual video format.

Silverlight supports a number of video formats, each identified by a *FourCC code*. FourCC is a standard four-character code that's used to uniquely identify video formats. In addition to all the existing formats (for example, H264 for h.264 video), two new formats were added specifically for use raw media and the `MediaStreamSource` API. Those are listed in table C.2.

Table C.2 Supported raw media FourCC codes in Silverlight

FourCC code	Description
RGBA	Raw, uncompressed RGB pixels with an alpha component. Silverlight currently ignores the alpha component during processing.
YV12	YUV 12. This is a common media output format used in many codecs.

In the example in this section, you'll use the RGBA format to push raw pixels without any special processing or encoding. It's the easiest format to use, requiring no algorithm other than providing a single pixel with a single color. Here is the video setup code for your simple white noise generator.

Listing C.3 Setting up the video stream

```
private int _frameTime = 0;
private const int _frameWidth = 320, _frameHeight = 200;
private const int _framePixelSize = 4;
```

```

private const int _frameBufferSize =
    _frameHeight * _frameWidth * _framePixelSize;
private const int _frameStreamSize = _frameBufferSize * 100;
private MemoryStream _frameStream = new MemoryStream(_frameStreamSize);
private MediaStreamDescription _videoDesc;
private void PrepareVideo()
{
    _frameTime = (int)TimeSpan.FromSeconds((double)1/30).Ticks;
    var streamAttributes =
        new Dictionary<MediaStreamAttributeKeys, string>();
    streamAttributes[MediaStreamAttributeKeys.VideoFourCC] = "RGBA";
    streamAttributes[MediaStreamAttributeKeys.Height] =
        _frameHeight.ToString();
    streamAttributes[MediaStreamAttributeKeys.Width] =
        _frameWidth.ToString();
    _videoDesc = new MediaStreamDescription(
        MediaStreamType.Video, streamAttributes);
}
protected override void OpenMediaAsync()
{
    var sourceAttributes =
        new Dictionary<MediaSourceAttributeKeys, string>();
    var availableStreams = new List<MediaStreamDescription>();

    PrepareVideo();

    availableStreams.Add(_videoDesc);
    sourceAttributes[MediaSourceAttributeKeys.Duration] =
        TimeSpan.FromSeconds(0).Ticks.ToString(
            CultureInfo.InvariantCulture);
    sourceAttributes[MediaSourceAttributeKeys.CanSeek] =
        false.ToString();
    ReportOpenMediaCompleted(
        sourceAttributes, availableStreams);
}

```

30 frames per second

Available streams

0 is infinite time

The listing shows two functions: `OpenMediaAsync` and `PrepareVideo`. They've been broken up that way because `OpenMediaAsync` will also need to support audio later in this section.

When the class is wired up to a `MediaElement`, Silverlight will first call the `OpenMediaAsync` function. In that function, you need to tell Silverlight what streams are available using a `MediaStreamDescription` instance, a single video stream in this case. Then you need to set up a source attribute for the duration of the video, infinite in this case, and another to indicate whether or not you allow seeking. You take that information and pass it into the `ReportOpenMediaCompleted` method to tell Silverlight you're ready.

The `PrepareVideo` method sets up some variables that will be used when you generate the samples. First, you identify the amount of time per frame. This can vary over the course of the video, but it'll be easier on the developer if you pick a constant frame rate. Then set up a dictionary of attributes that identifies the format of the

video, RGBA in this example, and the dimensions of each frame. Like the Commodore 64 emulator, this example uses 320x200 for the video frame dimensions. Finally, that's all packed into a `MediaStreamDescription` to be used when you start generating frames.

Once the video stream is set up, the next thing to do is to start pumping out frames to be displayed.

RETURNING THE SAMPLE

The main purpose of a `MediaStreamSource` implementation is to return samples. In the case of video, a sample is one complete frame, ready to be displayed. This listing shows the `GetVideoSample` function, called by `GetSampleAsync`.

Listing C.4 Returning the video frame sample

```
private int _frameStreamOffset = 0;
private Dictionary<MediaSampleAttributeKeys, string> _emptySampleDict =
    new Dictionary<MediaSampleAttributeKeys, string>();
private Random _random = new Random();
private byte[] _frameBuffer = new byte[_frameBufferSize];
private void GetVideoSample()
{
    if (_frameStreamOffset + _frameBufferSize > _frameStreamSize)
    {
        _frameStream.Seek(0, SeekOrigin.Begin);
        _frameStreamOffset = 0;
    }

    for (int i = 0; i < _frameBufferSize; i += _framePixelSize)
    {
        if (_random.Next(0, 2) > 0)
        {
            _frameBuffer[i] = _frameBuffer[i + 1] =
                _frameBuffer[i + 2] = 0x55;
        }
        else
        {
            _frameBuffer[i] = _frameBuffer[i + 1] =
                _frameBuffer[i + 2] = 0xDD;
        }
        _frameBuffer[i + 3] = 0xFF;
    }

    _frameStream.Write(_frameBuffer, 0, _frameBufferSize);

    var msSamp = new MediaStreamSample(
        _videoDesc, _frameStream, _frameStreamOffset,
        _frameBufferSize, _currentTime, _emptySampleDict);
    _currentTime += _frameTime;
    _frameStreamOffset += _frameBufferSize;

    ReportGetSampleCompleted(msSamp);
}
```

← Rewind when at end

← Alpha value
0xFF = Opaque

The `GetVideoSample` function first checks to see whether you're approaching the end of the allocated video buffer. If so, it rewinds to the beginning of the buffer. This is an important check to make, because you don't want to allocate a complete stream for every frame, but a stream can't be boundless in size.

Once that's done, you loop through the buffer, moving 4 bytes at a time (the size of a single pixel in the buffer) and generate a random pixel value. The pixel will either be almost white (0xDD, 0xDD, 0xDD) or dark gray (0x55, 0x55, 0x55). When playing with the sample, I found that pure black and white was far too harsh, and these two slightly gray values looked more natural. Though not obvious here, when setting the pixel values you need to do so in Blue, Green, Red, Alpha (BGRA) order.

The next step is to write the buffer to the stream. In this simple example, you could've written the bytes directly to the stream and eliminated the buffer. But in anything more complex than this, you're likely to have at least two buffers (a read-from and a write-to buffer), and even more likely to have a queue of frame buffers used for preloading the individual frames. Consider it a good practice.

Once the stream is populated, create the media stream sample, increment your time counters, and call `ReportGetSampleCompleted` to return the sample to Silverlight.

One interesting note is how sample time is used rather than frame numbers. The use of a time for each frame allows Silverlight to drop frames when it starts to lag behind. This was a main reason I chose `MediaStreamSource` over other approaches in the Silverlight C64 emulator. When the user's machine is busy, or in case it's too slow to run the emulator at full frame rate, I continue to chug along and let Silverlight skip frames it doesn't have time to show. This helps keep everything in sync time-wise, which is crucial when you're also creating audio.

C.1.3 Creating raw audio

In the previous section, you created a white noise video generator. Let's take that all the way and add in white noise audio. Surprisingly, audio is somewhat more complex to set up than video. This is due to the number of options available to you: audio can have different sample bit sizes, be mono or stereo, have different sample rates, and more.

All this information is stored in a class known as `WaveFormatEx`. In order to fit the listing into this book, I'm going to use a greatly simplified, but still functional, version of this class. This is a well-known, existing class. The implementation is not mine (so don't blame me for not using `StringBuilder`!).¹ Create this as a separate class file in your project:

Listing C.5 A simplified `WaveFormatEx` structure

```
public class WaveFormatEx
{
    public short FormatTag { get; set; }
    public short Channels { get; set; }
```

¹ Yeah, I know it's lazy of me not to go in and "fix" the class code.

```

public int SamplesPerSec { get; set; }
public int AvgBytesPerSec { get; set; }
public short BlockAlign { get; set; }
public short BitsPerSample { get; set; }
public short Size { get; set; }
public const uint SizeOf = 18;
public byte[] ext { get; set; }
public const Int16 FormatPCM = 1;
public string ToHexString()

{
    string s = "";
    s += ToLittleEndianString(string.Format("{0:X4}", FormatTag));
    s += ToLittleEndianString(string.Format("{0:X4}", Channels));
    s += ToLittleEndianString(string.Format("{0:X8}", SamplesPerSec));
    s += ToLittleEndianString(string.Format("{0:X8}", AvgBytesPerSec));
    s += ToLittleEndianString(string.Format("{0:X4}", BlockAlign));
    s += ToLittleEndianString(string.Format("{0:X4}", BitsPerSample));
    s += ToLittleEndianString(string.Format("{0:X4}", Size));
    return s;
}

public static string ToLittleEndianString(string bigEndianString)
{
    if (bigEndianString == null) { return ""; }
    char[] be = bigEndianString.ToCharArray();
    if (be.Length % 2 != 0) { return ""; }
    int i, ai, bi, ci, di;
    char a, b, c, d;
    for (i = 0; i < be.Length / 2; i += 2)
    {
        ai = i; bi = i + 1;
        ci = be.Length - 2 - i;
        di = be.Length - 1 - i;
        a = be[ai]; b = be[bi]; c = be[ci]; d = be[di];
        be[ci] = a; be[di] = b; be[ai] = c; be[bi] = d;
    }
    return new string(be);
}

public Int64 AudioDurationFromBufferSize(
    UInt32 cbAudioDataSize)
{
    if (AvgBytesPerSec == 0) return 0;
    return (Int64)(cbAudioDataSize * 10000000 / AvgBytesPerSec);
}
}

```

Main output function

Utility functions

The `WaveFormatEx` class is simply a way to specify the format to be used for Pulse-code Modulation (PCM) wave data in Silverlight. It's a standard structure, forming the header of the WAV file format, which is why you get oddities such as the big-to-little-endian format conversions. The class-based version here includes a single helper utility function `AudioDurationFromBufferSize`, which will be used when you output the PCM samples.

There are more complete implementations of `WaveFormatEx` to be found on the web, including one in my Silverlight Synthesizer project at <http://10rem.net>. Those implementations typically include a validation function that makes sure all the chosen options are correct.

With that class in place, let's move on to the actual stream setup.

SETTING UP THE WAV MEDIA SOURCE

The first step in setting up the sound source is to modify the `OpenMediaAsync` function. That function currently includes a call to `PrepareVideo` followed by adding the video stream description to the list of available streams. Modify that code so that it also includes the audio description information as shown here:

```
...
PrepareVideo();
PrepareAudio();
availableStreams.Add(_videoDesc);
availableStreams.Add(_audioDesc);
...
```

Once those changes are in place, you'll add the `PrepareAudio` function to the class. The `PrepareAudio` function is the logical equivalent to the `PrepareVideo` function; it sets up the format information for Silverlight to use when reading our samples. Here is the code for that function and its required class member variables and constants.

Listing C.6 The `PrepareAudio` function

```
private WaveFormatEx _waveFormat = new WaveFormatEx();
private MediaStreamDescription _audioDesc;
private const int _audioBitsPerSample = 16;
private const int _audioChannels = 2;
private const int _audioSampleRate = 44100;
private void PrepareAudio()
{
    int ByteRate = _audioSampleRate * _audioChannels *
        (_audioBitsPerSample / 8);

    _waveFormat = new WaveFormatEx();
    _waveFormat.BitsPerSample = _audioBitsPerSample;
    _waveFormat.AvgBytesPerSec = (int)ByteRate;
    _waveFormat.Channels = _audioChannels;
    _waveFormat.BlockAlign =
        (short)(_audioChannels * (_audioBitsPerSample / 8));

    _waveFormat.ext = null;
    _waveFormat.FormatTag = WaveFormatEx.FormatPCM;
    _waveFormat.SamplesPerSec = _audioSampleRate;
    _waveFormat.Size = 0;

    Dictionary<MediaStreamAttributeKeys, string> streamAttributes =
        new Dictionary<MediaStreamAttributeKeys, string>();
    streamAttributes[MediaStreamAttributeKeys.CodecPrivateData] =
        _waveFormat.ToHexString();
}
```

← **WaveFormatEx**

← **Must be zero**

```

    _audioDesc = new MediaStreamDescription(
        MediaStreamType.Audio, streamAttributes);
}

```

The most important parts of the listing are the constants controlling the sample format. For this example, you're generating 16-bit (2-byte) samples, in two channels (stereo sound), at a sample rate of 44,100 samples per second: CD-quality audio.

Once those constants are established, they're used to figure out almost everything else, including the number of bytes per second and the block alignment. In our case, the bytes per second is 44,100 (sample rate) * 2 (16 bit sample size) * 2 (channels) or 176,400 bytes per second. That's a lot of data.²

Once the `WaveFormatEx` structure is filled out with this information, set it as the Codec Private Data using its little-endian hex string format as required. That format is independent of your actual processor architecture. Then create the audio description from that data, to be used when reporting samples back to Silverlight.

CREATING SOUND SAMPLES

The final step is to output the audio samples. This requires generating the individual samples and returning them in chunks of predefined size. You'll use a random number generator to generate the noise, much as you did with video. Here is how to fill a buffer with audio and return those samples to Silverlight.

Listing C.7 Outputting audio samples

```

private long _currentAudioTimeStamp = 0;
private const int _audioBufferSize = 256;
private const int _audioStreamSize = _audioBufferSize * 100;
private byte[] _audioBuffer = new byte[_audioBufferSize];
private MemoryStream _audioStream = new MemoryStream(_audioStreamSize);
private int _audioStreamOffset = 0;
private double _volume = 0.5;
private void GetAudioSample()
{
    if (_audioStreamOffset + _audioBufferSize > _audioStreamSize)
    {
        _audioStream.Seek(0, SeekOrigin.Begin);
        _audioStreamOffset = 0;
    }
    for (int i = 0; i < _audioBufferSize;
        i += _audioBitsPerSample / 8)
    {
        short sample =
            (short)( _random.Next((int)short.MinValue,
                (int)short.MaxValue) * _volume);
    }
}

```

Internal buffer size

Sample randomizer

² This is a number to play with if you hit performance problems. Using 8-bit samples instead of 16-bit halves the amount of data to 88,200 bytes, and mono 8-bit audio at 22.5 khz results in an easier 22,000 bytes. On the other side, 48 khz stereo audio at 24 bits, typical in higher-end audio, results in a huge 288,000 bytes per second without even considering surround sound. I've seen digital hardware synthesizers that claim to use 192 khz 24-bit samples, resulting in 1,152,000 bytes per second. More bytes means more work for the computer and more memory pressure.

```

        _audioBuffer[i] = (byte)(sample & 0xFF00);
        _audioBuffer[i + 1] = (byte)(sample & 0x00FF);
    }
    _audioStream.Write(_audioBuffer, 0, _audioBufferSize);
    MediaStreamSample msSamp = new MediaStreamSample(
        _audioDesc, _audioStream, _audioStreamOffset, _audioBufferSize,
        _currentAudioTimeStamp, _emptySampleDict);
    _currentAudioTimeStamp +=
        _waveFormat.AudioDurationFromBufferSize((uint)_audioBufferSize);
    _audioStream = new MemoryStream(_audioStreamSize);
    ReportGetSampleCompleted(msSamp);
}

```

The process for generating the white noise audio sample is similar to generating the frames of video. But instead of having a fixed-width x height buffer you must fill, you can generate as long or as short a sample as you want. This is controlled by the audio buffer size set in code. In general, you want this number to be as low as possible; larger numbers typically introduce latency as well as skipped video frames—the system is too busy generating audio to show the video frame. But set the number too low, and the audio will stutter. If you find the white noise stuttering on your machine, up the buffer to 512 or so and see how that works for you.

TIP To help with latency, you can also play with the `AudioBufferLength` property of the `MediaStreamSource` class. In most cases, you won't be able to get that below 30 ms or so, but that value is itself very hardware-dependent. That property is my own contribution to the class; I was the only one insane enough to be writing a Silverlight-based audio synthesizer at the time. I ran into problem after problem with the triple-buffering (my buffer, plus the Silverlight MSS buffer, plus the underlying DirectX buffer), to the point where all audio was delayed by about 2-3 seconds. The team worked with me to identify where the issues were and then added this knob into the base class to help tweak for latency-sensitive applications like mine.

Once the buffer size is established, perform the same stream overrun check that you did for video, and for the same reasons. Then, loop through the buffer, 2 bytes (16 bits) at a time, and generate a white noise sample. Once the sample is generated, get the 2 bytes from it using a little bit masking, and write those bytes into the buffer. Once the buffer is filled, it's copied into the stream and the sample response built. After incrementing the time counters, the last step is to report the sample to Silverlight.

If you run the application at this point, you should have a short delay while the startup code is executed and the Silverlight internal buffers are filled, followed by simultaneous audio and video white noise. On the surface, this may not seem impressive. But when you consider that the video and audio is completely computer generated, it's considerably more interesting.

Raw audio and video also allow you to display any type of media for which you can write a managed code decoder. Much of the IIS Smooth Streaming client for Silver-

light covered in appendix B is written using a custom `MediaStreamSource` implementation. Though writing a typically hardware-implemented 1080p HD codec in managed code may not lead to good performance, there are many other popular formats that don't have native Silverlight support but that would benefit from a custom `MediaStreamSource` implementation. Some groups have already done work with codecs for things like the various Ogg formats. How about creating a Silverlight MOD/S3M³ player using this technology? Because you can send raw bits directly into the media pipeline, the limit is just the amount of processing you can cram into managed code while maintaining acceptable performance—well, that, and how much you paid attention in math class.

As much as I like `MediaStreamSource`, it does rely on the Silverlight media pipeline and `MediaElement`. That's fine for traditional audio and video, but it isn't well-suited to sound effects where startup time is critical.

C.2 *Playing low-latency sound*

The `MediaStreamSource` API is an interesting way to create your own sounds. But there's always a bit of a lag between starting the sound and hearing the first bit of noise. The `MediaElement` itself also has this issue when playing “regular” media. When you're playing music or video, the primary use case for `MediaElement`, this isn't a problem. Yet when you want to play a sound when the user taps an element in a touch-screen kiosk app, or when they fire a laser in your newest hit game, it can be a real problem.

In the past you'd load up a number of `MediaElements`, often using a round-robin queue to create a poor-man's sound buffer. This worked well enough, but it was complex, was too UI driven, and required a fair bit of code to manage it properly. You can do better—much better.

Silverlight 5 introduced the XNA⁴ `SoundEffect` and `SoundEffectInstance` classes. These are first-class types that enable the playback of XNA-compatible WAV files without the startup lag you see in the `MediaElement`. In this section, you'll explore both of these classes to better understand how to play back sound, and you'll even do basic manipulation of pitch, looping, and volume.

C.2.1 *Playing sound effects*

The core of low-latency sound in Silverlight revolves around the `SoundEffect` class. This class, found in the `Microsoft.Framework.Xna` DLL, includes all the functionality required to load and play a compatible sound wave.

Unlike the typical case with the `MediaElement`, you're likely to have your sound effects embedded as resources inside the project itself rather than downloaded from

³ Go Amiga!

⁴ In case you haven't stumbled across it, XNA is a DirectX-based game developer library used to write for Xbox, PC, Windows Phone, and Silverlight 5. As you can imagine, real-time sound is an important part of any serious game.

the web. This ensures as low a load time as possible, because the network doesn't become the bottleneck.

For that reason, the `SoundEffect` class exposes a `FromStream` method that may be used to load bytes from any arbitrary sound stream. Here is the simplest way to load and play a WAV file stored in your Silverlight app.

Listing C.8 Loading a sound effect from a WAV file

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Xna.Framework.Audio;

namespace SoundEffectDemo
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void FireButton_Click(object sender, RoutedEventArgs e)
        {
            var streamInfo =
                Application.GetResourceStream(
                    new Uri("laser_shot.wav", UriKind.RelativeOrAbsolute));

            var effect = SoundEffect.FromStream(streamInfo.Stream);
            effect.Play();
        }
    }
}
```

For the listing to work, be sure to include the WAV file in the root of your project (or change the path appropriately) and set its build action to Content. Doing so ensures that the file becomes part of the XAP and can be found using the path provided in this example.

Be sure to add a reference to the `Microsoft.Xna.Framework` library found in the SDK folder as well. That's where the sound effect functionality, and much of XNA itself, resides.

Finally, you'll need some UI. This listing shows the simple XAML UI for this and the subsequent examples.

Listing C.9 UI for the low-latency sound examples in this appendix

```
<UserControl x:Class="SoundEffectDemo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```



```
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="400">

<Grid x:Name="LayoutRoot" Background="White">
    <Button x:Name="FireButton" Content="Pew Pew!"
        FontSize="30" Width="200" Height="75"
        Click="FireButton_Click" />
</Grid>
</UserControl>
```

Go ahead and run the code now. If you get an error where the stream is null, you likely misspelled the filename or didn't mark the WAV file with a build action of Content in the property pane.

If you get an error when playing or creating the sound effect, you may have picked an incompatible WAV file. XNA is very specific about which formats it allows. The sound effect WAV file must be:

- PCM-encoded WAV
- 8- or 16-bit sample size; no 24-bit floating point support
- Mono or stereo
- 22.5 kHz, 44.1 kHz, or 48 kHz sample rate

Other formats simply aren't supported. If you download a random WAV file off the internet and run into issues, it almost certainly doesn't fit the required format criteria. Luckily, there are any number of freeware, shareware, and commercial apps that can convert the WAV file into a format that works for you. I happen to like Audacity.

SoundEffect and IDisposable

The `SoundEffect` class implements `IDisposable`. Typical best practice for that would seem to indicate that you'd want to wrap the creation of the effect and its playback inside a using statement like this:

```
using (var effect = SoundEffect.FromStream(streamInfo.Stream))
{
    effect.Play();
}
```

But if you do that, the sound effect will never get a chance to play.

If you want to be a good citizen, you'll need to keep track of your sound effects and only dispose them once you're done with them and they've finished playing. In practice, that can be hard to do, but it's usually not impossible.

Format pickiness aside, it doesn't get a whole lot simpler than that. No more `MediaElement` round-robin schemes or other strangeness just to get a real-time sound. But, and I say this in my best 3 a.m. infomercial voice, "That's not all!" Creating a sound effect loads the bytes for the WAV file into the data for that class. What happens when you want to have several instances of the same sound effect but want to be kind to the user's system memory?

C.2.2 Creating and playing instances

Sound effects are often reused in a game. In addition, it's typical to have multiple uses of the same sound effect overlapping. Consider two or more spaceships shooting at each other. Unless they're British, it's unlikely they'll be polite enough to take turns firing. Instead, the shooting will overlap, and so will the sound effects.

Rather than load up separate `SoundEffect` instances, it's more polite to load the WAV file a single time, then use `SoundEffectInstance` classes to create the individual instances.

This shows how to load up a single sound effect and then use instances each time you wish to fire off the sound.

Listing C.10 Using the `SoundEffectInstance` class for WAV reuse

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Xna.Framework.Audio;

namespace SoundEffectDemo
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();

            Loaded += new RoutedEventHandler(MainPage_Loaded);
        }

        private SoundEffect _laserShotEffect;

        void MainPage_Loaded(object sender, RoutedEventArgs e)
        {
            var streamInfo =
                Application.GetResourceStream(
                    new Uri("laser_shot.wav", UriKind.RelativeOrAbsolute));

            _laserShotEffect = SoundEffect.FromStream(streamInfo.Stream);
        }

        private void FireButton_Click(object sender, RoutedEventArgs e)
        {
            var instance = _laserShotEffect.CreateInstance();

            instance.Play();
        }
    }
}
```

← Main effect

← Instance

In the listing, you create the `SoundEffect` as soon as the page loads, but you don't play it at that time. Instead, when the user clicks the fire button, you create a `SoundEffectInstance` and play that instead. Not only is this approach friendlier memory-wise, but if the sound effect load was a relatively slow process like downloading from a

website, you'd only incur that delay when creating the base sound effect, not when creating the instances.

The `SoundEffectInstance` class gives you a lot more than just reuse of a WAV file—it even allows for some sound manipulation.

C.2.3 *Modifying the sound*

The `SoundEffectInstance` class provides properties to enable basic sound manipulation. For example, you can pan the effect, change its pitch and volume, and even loop the sound. This is by no means a full set of editing functions, or the ability to start with raw data and create a fantastic sound using code. There's no built-in reverb or delay or other capabilities. For the typical use, though, you'll find these sufficient to tweak the sound to fit the in-application dynamics of a typical game or touch-screen app.

The `SoundEffect` class doesn't share these properties, but it does have an overload of the `Play` method, which takes all three in at the same time. It's more common to use a `SoundEffectInstance`, so I'll ignore that overload and deal only with the property approach.

PANNING THE EFFECT

One of the simplest ways you can change a sound is to change which side it appears to come from. Normally, sounds come from the aural center. If you want to modify that to perhaps simulate the footsteps of a person walking from the left side of the screen to the right, you could modify panning to move from aural left to aural right. This is achieved by changing the `Pan` property.

The `Pan` property can be set to any value from -1 to +1, with -1 being the far left, 0 being the center, and +1 being the far right. The default value is 0 or center. The next listing builds on the previous example to show how to set this property to have the sound come out of the left side of whatever you're listening on.

Listing C.11 Panning the sound effect

```
private void FireButton_Click(object sender, RoutedEventArgs e)
{
    var instance = _laserShotEffect.CreateInstance();

    instance.Pan = -1;

    instance.Play();
}
```

When you run the listing and hit the fire button, you should hear the sound come out of the left side. If it comes out of the right side, your speakers are miswired, or your headphones are on backward.

CHANGING PITCH AND VOLUME

When you're working with sound effects, subtly varying the pitch and volume can help lend a bit of variety to what would otherwise sound too identical. For example, a footstep might have the pitch and volume altered ever so slightly at random points,

making some steps sound quicker or slower than others. In our case, the alteration can be used to indicate different types of laser cannons on our fictional spaceship.

Here's how to play the sound effect at a higher pitch and at 75 percent volume.

Listing C.12 Altering the pitch and volume

```
private void FireButton_Click(object sender, RoutedEventArgs e)
{
    var instance = _laserShotEffect.CreateInstance();

    instance.Volume = 0.75f;
    instance.Pitch = 1.0f;

    instance.Play();
}
```

When you run the listing, you should hear a much higher pitch version of the sound effect you've been using. It'll also have slightly reduced volume, in this case 75 percent of full.

Before moving on to looping sound, let's combine pitch, volume, and pan changes to provide real dynamics to your sound. Let's assume that your fighter alternates between left and right weapons when shooting. Let's add a tiny bit of random pitch and volume variation to make it sound slightly more realistic. This shows the final version with all three effects in place:

Listing C.13 Combining the tweaks to make for a more realistic sound

```
private float _currentPan = -0.5f;
private Random _random = new Random();

private void FireButton_Click(object sender, RoutedEventArgs e)
{
    var instance = _laserShotEffect.CreateInstance();

    instance.Volume = (_random.Next(750, 1000) / 1000.0f) * 1.0f;
    instance.Pitch = (_random.Next(500, 1000) / 1000.0f) * 1.0f;
    instance.Pan = _currentPan;

    instance.Play();

    _currentPan = _currentPan * -1;
}
```

◀ Alternate sides

This example builds on the previous ones. Run it and hit the fire button a number of times. You'll hear a slight variation in volume and pitch, as well as an alternating of left and right side audio. If the difference doesn't jump out at you, you can play with the values to make them more extreme. For example, this will get you some pretty extreme changes:

```
instance.Volume = (_random.Next(100, 1000) / 1000.0f) * 1.0f;
instance.Pitch = (_random.Next(-1000, 1000) / 1000.0f) * 1.0f;
```

You can even change the `_currentPan` value from `-0.5f` to `-1.0f` to have extreme left and right panning.

The last manipulation you'll look at is one of my favorites: looping the sound so it plays continuously.

LOOPING THE SOUND

To loop a sound, you first need to create (or find) a sound effect that will cleanly loop. If your sound effect is optimized for looping, you won't hear clicks, pops, or jumps when the effect restarts. In fact, the `SoundEffectInstance` class has excellent support for lag-free looping of sounds, a killer feature for background sounds effects like engine drones.

The next listing shows a new sound effect, this one optimized for looping. The resulting effect sounds a bit like the engine drone you might expect when piloting, say, a large freighter or fighter.

Listing C.14 Adding in a looped background engine sound

```
private SoundEffect _laserShotEffect;
private SoundEffect _engineEffect;

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    var laserStreamInfo =
        Application.GetResourceStream(
            new Uri("laser_shot.wav", UriKind.RelativeOrAbsolute));

    _laserShotEffect = SoundEffect.FromStream(laserStreamInfo.Stream);

    var engineStreamInfo =
        Application.GetResourceStream(
            new Uri("engine_rumble4.wav", UriKind.RelativeOrAbsolute));

    _engineEffect =
        SoundEffect.FromStream(engineStreamInfo.Stream);

    var engine = _engineEffect.CreateInstance();
    engine.Pitch = -1.0f;
    engine.Volume = 0.5f;
    engine.IsLooped = true;
    engine.Play();
}
```

**Looped and detuned
engine instance**

This example uses a new sound effect WAV named `engine_rumble.wav`. As you did with the other file, you marked this one with a build action of Content. Your loop point in this file isn't perfect, so there's a bit of a hesitation at the end. That's easily corrected with audio editing software or even overlapped and delayed sounds. Run the application and note that you can still fire the weapons without interfering with the background drone.

Wow! You just about have a game. Well...the audio part of it anyway. Combine this with the 3D transform from chapter 7, or the 2D graphics from chapters 24 and 25, and you make the makings of a nice little game engine.

In addition to panning, pitch, volume, and looping, the `SoundEffectInstance` has a number of other useful properties and methods. For example, you can pause and resume the effect instance, something not possible on the `SoundEffect` class.

A best practice is to always use a `SoundEffectInstance` when playing a sound. This will enable you to easily reuse the sound later, as well as tweak it to meet the needs of your application.

The `SoundEffect` and `SoundEffectInstance` classes provide a great way to have low-latency sound in Silverlight. They're also the only built-in way to use WAV files, something `MediaElement` doesn't natively support. In addition to games, you'll find uses for these classes in kiosk applications and any other place where immediate aural feedback is an appropriate part of the application's interface.

So far, you've looked at the `MediaStreamSource` and low-latency sound, two ways to get media into and out of Silverlight. One final way to get video and audio into Silverlight is to use the webcam and microphone APIs. A segment of the API, especially the `VideoSink` and `AudioSink` classes, is conceptually similar to the `MediaStreamSource` code you've completed in the first section but thankfully much simpler.

C.3 Using the webcam

Silverlight 4 introduced the ability to capture media from video capture devices and audio capture devices. Though designed with other devices (such as TV capture cards) in mind, the current implementation handles only webcams and microphones. These devices enable the Silverlight developer to capture raw video and audio data, as well as snapshot stills. Though its first release isn't suitable for conferencing scenarios (there's no built-in compression or encoding), it's excellent for local capture and storage and upload scenarios.

If you've ever tried to use an arbitrary webcam (or microphone) using another technology such as WPF, you'll appreciate how simple the Silverlight team has made this. Not only do you get to avoid DirectShow and similar technologies, the webcam and mic access works cross-platform. As far as device abstraction layers go, this is pretty sweet.

In this section, I'll first explain how to gain access to the webcam and microphone in Silverlight. Then I'll show you how to work with the default webcam and microphones for the platform, including how to capture video and still images. Then, because most machines have more than one audio capture device, and some even more than one video capture device, I'll look at what's required to allow the user to select a specific webcam or microphone.

C.3.1 Gaining access to capture devices

In sandboxed applications, the application must request access to the webcam from a user-initiated event, such as a button click. This is to ensure that a rogue application

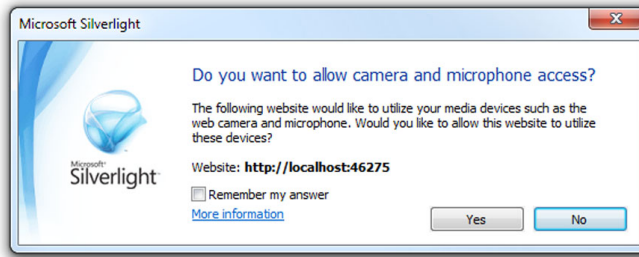


Figure C.2 Webcam and microphone access confirmation dialog

on a website doesn't start photographing you without your consent. The request is explicit, as shown here:

```
if (CaptureDeviceConfiguration.AllowedDeviceAccess ||  
    CaptureDeviceConfiguration.RequestDeviceAccess())  
{ ... }
```

The first check is to see whether the application has already been granted access; this is true if it's running under elevated trust or the user has already allowed access. The second check runs only if the first check is false; it causes the webcam and microphone device access confirmation dialog to be displayed, as shown in figure C.2.

Once the user has confirmed access, you can begin to capture using a specific device or the default devices. Typically, you'll use the default device.

CHANGING THE DEFAULT CAPTURE DEVICE

Silverlight allows the user to set the default webcam and default microphone. This is done by right-clicking on any Silverlight application and selecting the Silverlight menu option. Alternatively, you can open Silverlight from your program shortcut. Once there, select the Webcam/Mic tab and pick from the list of available options. You'll see a preview of the webcam to the left and an audio level meter for the microphone on the right. Figure C.3 shows the configuration dialog.

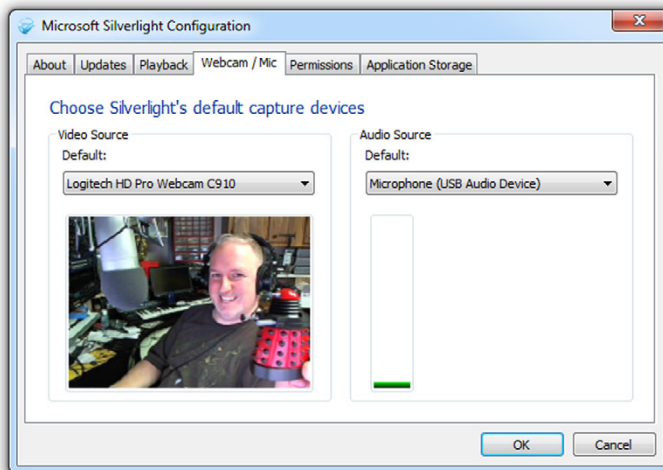


Figure C.3 Silverlight default webcam and microphone tab in the Silverlight settings dialog. Say hello to my little friend. Unfortunately, he doesn't clean desks; he only exterminates.

The settings start out using default capture devices on your machine. You can change it from there. The changes will globally affect all Silverlight applications that use the webcam or microphone.

With the default device set in Silverlight, it's time to write a little code to capture information from the default webcam.

C.3.2 Working with video

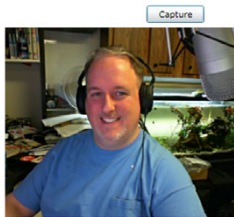
To get the default webcam, you need only call the `GetDefaultVideoCaptureDevice` method of the `CaptureDeviceConfiguration` class. If this method returns `null`, there's no recognized webcam on the machine.

Once you have a capture device, capturing video requires wiring up a capture source and using it as the input source for a `VideoBrush`. The `VideoBrush` is then used to fill a shape, typically a rectangle, on the Silverlight surface.

The next listing shows how to create a simple webcam viewer using the default webcam at a default capture resolution.

Listing C.15 Capturing video using the default capture device

Result:



XAML:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Button x:Name="Capture" Content="Capture"
    Width="75" Height="23" Margin="232,12,0,0"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Click="Capture_Click" />
  <Rectangle x:Name="PresentationSurface"
    Width="376" Height="247" Margin="12,41,0,0"
    HorizontalAlignment="Left"
    VerticalAlignment="Top" />
</Grid>
```

C#:

```
private void Capture_Click(object sender, RoutedEventArgs e)
{
    if (CaptureDeviceConfiguration.AllowedDeviceAccess ||
        CaptureDeviceConfiguration.RequestDeviceAccess())
    {
        var camera =
            CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
        if (camera != null)
        {
```

Default video
capture device




```

var source = new CaptureSource();
source.VideoCaptureDevice = camera;

var videoBrush = new VideoBrush();
videoBrush.Stretch = Stretch.Uniform;
videoBrush.SetSource(source);

PresentationSurface.Fill = videoBrush;
source.Start();
    }
}
}

```

← Start capturing

You first set up the button for the user-initiated video capture and the rectangle to hold the rendered output, both in XAML. In the code, you perform the check to see whether you have access, and request it if not. Then you get the default video capture device and assign it as the capture device for the `CaptureSource`. The video display isn't a `MediaElement`. Instead, you create a `VideoBrush`, set its source to your `CaptureSource`, and paint the rectangle with the output. Finally, you start the capture itself.

This example used the default capture resolution. That's okay for an example, but in a real application, you'll likely want to pick a specific video format based on screen resolution or even the fps.

SETTING THE DESIRED VIDEO FORMAT

Webcams typically support a number of resolutions and video formats. I have a Logitech HD webcam on my PC, and it handles everything from the smallest of postage stamps to 1080p HD video. I also have a Microsoft Cinema HD camera attached that does 720p.⁵ Because the capabilities vary from model to model, you'll need a way to query the webcam to identify its supported video formats and pick between possible capture devices.

The `VideoCaptureDevice` class contains a number of properties. The one of interest to us in this case is the `SupportedFormats` collection. `SupportedFormats` is a collection of `VideoFormat` objects, the properties of which are displayed in table C.3.

Table C.3 The `VideoFormat` class

Member	Description
<code>FramesPerSecond</code>	A floating-point value indicating the number of frames per second.
<code>PixelFormat</code>	Currently, the only valid pixel format is 32 bits per pixel, ARGB.
<code>PixelHeight</code>	The height of the frames in pixels.
<code>PixelWidth</code>	The width of the frames in pixels.
<code>Stride</code>	The number of bytes in a single horizontal line of the frame. Divide this by <code>PixelWidth</code> to know the bytes per pixel, regardless of <code>PixelFormat</code> . A negative stride indicates the image is upside down.

⁵ Hey, I have a 6-core PC (12 if you count hyperthreading). I have to do something to justify that. Two simultaneous HD webcam streams seems to do the trick.

To query the formats for my camera, I inserted this bit of code into the listing at the beginning of this section:

```
foreach (VideoFormat format in camera.SupportedFormats)
    Debug.WriteLine(
        format.PixelWidth + "x" +
        format.PixelHeight + " at " +
        format.FramesPerSecond + " fps " +
        format.PixelFormat.ToString());
```

The resulting list included (among many others) these entries for my Cinema HD:

```
640x480 at 30.00003 fps Unknown
160x120 at 30.00003 fps Unknown
160x120 at 30.00003 fps Unknown
1280x720 at 15.00002 fps Unknown
1280x720 at 15.00002 fps Unknown
960x544 at 30.00003 fps Unknown
960x544 at 30.00003 fps Unknown
800x448 at 30.00003 fps Unknown
800x448 at 30.00003 fps Unknown
800x600 at 30.00003 fps Unknown
...
```

Oddly enough, the pixel format came across as *Unknown* in all cases. Try it with your own webcam and the results will likely vary. Once you see a video format that works for you, you can choose it by assigning it to the *DesiredFormat* property of the *VideoCaptureDevice*. This example uses a LINQ expression to grab the first format with the highest resolution:

```
var format = (from VideoFormat f in camera.SupportedFormats
              orderby f.PixelWidth * f.PixelHeight descending
              select f).FirstOrDefault<VideoFormat>();
if (format != null)
    camera.DesiredFormat = format;
```

That will pick the format with the highest total pixel count. You can modify the statement to pick only the largest width, or the largest size that will fit within a given box, and so forth. Figure C.4 shows the 720p HD version of the webcam shot from the previous listing.

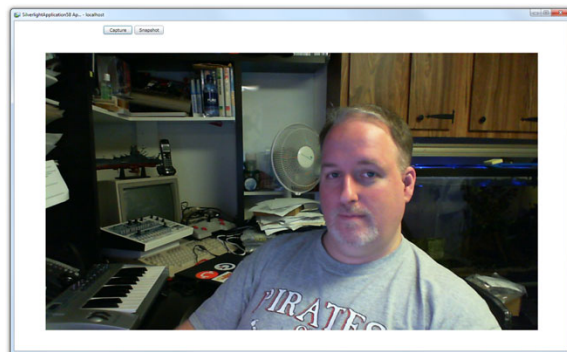


Figure C.4 Webcam screen shot at 720p HD, selected using the *DesiredFormat* property and LINQ. I'm practicing my raised-eyebrow news anchor face. I'll try harder next time. Dig the C128 in the background!

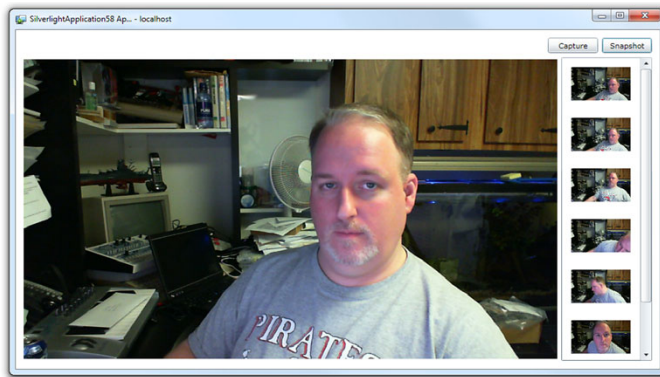


Figure C.5 Capturing the largest video size, plus a series of still photos bound to a `ListBox` on the right. Did I get the anchor look any better? Maybe I need a suit.

One reason you may want to capture at a high resolution is to support the capturing of still images. The Silverlight webcam API allows you to use the webcam as a simple still image camera, returning individual images as `WriteableBitmap` instances.

C.3.3 Capturing still images

Now that you have a reasonably high resolution selected, taking still photos makes much more sense. The Silverlight webcam API supports taking still photos by using an asynchronous capture method. You click a button and call a function, and a few fractions of a second later, the event fires with the image data.

In this section, you'll augment your webcam display application to include a `ListBox` filled with captured still images. Figure C.5 shows the final application.

This listing shows the new XAML required to create the display shown in figure C.5. Note the use of the `DataTemplate` for displaying the bound image information.

Listing C.16 XAML Capturing still images

XAML:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Button x:Name="Capture" Content="Capture"
    Width="75" Height="23" Margin="0,12,93,0"
    HorizontalAlignment="Right" VerticalAlignment="Top"
    Click="Capture_Click" />
  <Button x:Name="TakeSnapshot" Content="Snapshot"
    Height="23" Width="75" Margin="0,12,12,0"
    VerticalAlignment="Top" HorizontalAlignment="Right"
    Click="TakeSnapshot_Click" />
  <Rectangle x:Name="PresentationSurface" Margin="12,41,154,12" />
  <ListBox x:Name="Images" Width="136" Margin="0,41,12,12"
    HorizontalAlignment="Right"
    ScrollViewer.HorizontalScrollBarVisibility="Disabled">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <Image Margin="10" Height="50" Width="100"
```

```

        Source="{Binding}" />
    </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Grid>

```

The XAML creates a second button, for requesting a snapshot, and adds a `ListBox` to hold the images. The `DataTemplate` for the `ListBox` is pretty simple; all it includes is a single image with its source set to be the item bound to it.

Once you have the XAML in place, use the following code to update the code-behind.

Listing C.17 C# code for capturing the still images

```

public MainPage()
{
    InitializeComponent();
    Images.ItemsSource = _images;
}
private CaptureSource _source;
private ObservableCollection<ImageSource> _images =
    new ObservableCollection<ImageSource>();
private void Capture_Click(object sender, RoutedEventArgs e)
{
    if (CaptureDeviceConfiguration.AllowedDeviceAccess ||
        CaptureDeviceConfiguration.RequestDeviceAccess())
    {
        var camera =
            CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
        if (camera != null)
        {
            _source = new CaptureSource();
            var format = (from VideoFormat f in camera.SupportedFormats
                           orderby f.PixelWidth * f.PixelHeight descending
                           select f).FirstOrDefault<VideoFormat>();
            if (format != null)
            {
                camera.DesiredFormat = format;
                _source.VideoCaptureDevice = camera;

                var videoBrush = new VideoBrush();
                videoBrush.Stretch = Stretch.Uniform;
                videoBrush.SetSource(_source);
                PresentationSurface.Fill = videoBrush;

                _source.CaptureImageCompleted += (s, ea) =>
                {
                    _images.Add(ea.Result);
                };
                _source.Start();
            }
        }
    }
}
private void TakeSnapshot_Click(object sender, RoutedEventArgs e)
{
    _source.CaptureImageAsync();
}

```

**CaptureSource
refactored to
class-level**

Live video display

The previous listing builds on your previous code, refactoring some things out to class-level variables and adding new code. In addition to refactoring the `CaptureSource` out to class level, you add a new `ObservableCollection` of `ImageSource` to the class members. This will be used as the items source for the `ListBox` to support the binding of images using the `DataTemplate`.

The majority of the code inside `Capture_Click` is the same as what you've built so far. I included the LINQ method for obtaining the highest resolution, as you saw in previous examples. Toward the end of the method, before starting the webcam capture, you include an event handler to add the captured image to the `ObservableCollection`. This image is a `WriteableBitmap` (covered in chapter 25) so you can do additional manipulation with it if you want. Finally, the button click handler for the snapshot button calls the `CaptureImageAsync` method of the capture source.

With that code in place, your webcam display app can now capture stills alongside displaying the output from the webcam. In theory, you could treat those stills like individual frames in a video, but a better way to access the frame data is to use a custom `VideoSink`.

C.3.4 *Getting the raw video data*

Obviously, capturing still images at random frames is no substitute for being able to get at the raw video bits. Currently, the only way to access the raw video stream is to create your own `VideoSink` class. This is a class that'll take a video capture source and let the capture source push samples to it. It's possible then to get access to the raw bytes for the frames, but they'll be uncompressed. I have to stress that without fast compression, a video conferencing or chat application would be out of the question. Though it's possible to perform this compression from code inside Silverlight, it's not likely to perform well enough to use on a real production application.

Disclaimers aside, let's see how to implement this. The first task is to create the custom `VideoSink` class. The class has no real implementation, as it'd completely depend on what you want to do with the bits. I've seen some examples that write out uncompressed (huge) AVI files, for example.

Listing C.18 A sample `VideoSink` class for capturing raw webcam video

```
public class CustomVideoSink : VideoSink
{
    private long _currentFrame = 0;
    protected override void OnCaptureStarted()
    {
        VideoFrameQueue.Open();
    }
    protected override void OnCaptureStopped()
    {
        VideoFrameQueue.Close();
    }
    protected override void OnFormatChange(VideoFormat videoFormat)
```

```

{
    VideoFrameQueue.VideoFormat = videoFormat;
}
protected override void OnSample(
    long sampleTimeInHundredNanoseconds,
    long frameDurationInHundredNanoseconds,
    byte[] sampleData)
{
    _currentFrame++;
    VideoFrameQueue.Append(
        _currentFrame,
        sampleTimeInHundredNanoseconds,
        frameDurationInHundredNanoseconds,
        sampleData);
    System.Diagnostics.Debug.WriteLine(_currentFrame);
}
}

```

← Capture format

← Append frame to queue

In the listing, `CustomVideoSink` derives from the `VideoSink` class. That class provides four overridable members of interest. The `OnCaptureStarted` and `OnCaptureStopped` methods are used for startup and shutdown code. In those methods, you open and close a fictional `VideoFrameQueue` class. The implementation of that class will vary significantly based on what you intend to do with the raw bytes, so I've left it out of this example.

One other utility method is `OnFormatChanged`. This method is executed when the video format is changed, and it will always fire at least once, at the beginning of the capture. Once you know the video format, you can start doing something useful with the bytes that make up each frame. The `OnSample` method provides those bytes to you.

In the `OnSample` method, you'll almost certainly want to write the bytes and other required information to a queue to be processed. I've represented that with the `VideoFrameQueue` member. The queue would likely have a worker on a background thread that would write the frame to a larger file format, or do some simple encoding/compression as required. If you try to do that all inside this method, you'll run into timing issues.

The last step is to hook your custom video sink in to the processing pipeline. First, in the code-behind of the listing from the start of this chapter, add the following private member variable:

```
private CustomVideoSink _sink = new CustomVideoSink();
```

Then, in the same listing, modify the capture block in the button click event handler to look like this:

Listing C.19 Using a custom `VideoSink` to grab frames

```

var camera =
    CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
if (camera != null)
{

```

```

var source = new CaptureSource();
source.VideoCaptureDevice = camera;
VideoBrush videoBrush = new VideoBrush();
videoBrush.Stretch = Stretch.Uniform;
videoBrush.SetSource(source);
_sink.CaptureSource = source;

PresentationSurface.Fill = videoBrush;
source.Start();
}

```

← Wire up new VideoSink

In this listing, you’ve wired your new `CustomVideoSink` into the existing code. The new line in the event handler assigns the capture source, so the sink is now wired up to the webcam. Note that you can have more than one video sink attached to any capture source, but the processor utilization will rise proportionally.

C.3.5 A note about audio

Video is seldom captured alone. More often than not, you’ll want to capture audio as well. The Silverlight webcam and microphone API supports capturing audio independently, or along with video.

The Silverlight Microphone API is almost identical to the Webcam API, so I’ll leave it out for space reasons. The primary difference is that instead of a `VideoCaptureDevice` you’ll have an `AudioCaptureDevice`. There’s no native way to output the raw audio, so you’ll need to create an `AudioSink` just like you created a `VideoSink` for grabbing video frames. Of course, just as I noted with the `VideoSink`, what you do in the `AudioSink` is going to depend on what your plans are for encoding. The data format that comes from Silverlight is raw PCM audio.

The Silverlight Webcam API is a powerful way to integrate video capture devices into your application. Already I’ve seen some novel uses, including stop-motion animation, image and gesture recognition, and Facebook photo uploading. The API is simple to use, providing you with the device capabilities and a simple way to request access. It works cross-platform and abstracts away all the little details you’d normally need to understand to work with webcam and microphone devices on various machines.

C.4 Summary

One place where Silverlight has always excelled is in the delivery of media. Not only does it have first-class support for the most popular media types, but if Silverlight doesn’t support a media format you want to use, it has a provision for allowing you to create managed codecs, decoding your own format, and sending the raw unencoded bytes to Silverlight. This API is so complete, I’ve even been able to use it to generate video and audio from code, without any original media source files.

While Silverlight has been excellent for media in general, it hasn’t been very good for low-latency or near-real-time audio as would be required in kiosk apps and games. In Silverlight 5, we borrowed a little code from our good friends in the XNA and

Windows Phone teams and implemented the features you need to be able to have super responsive sound effects.

Finally, not all media comes from files or algorithms. Sometimes, media comes in the form of captured video, audio, and still images from an attached webcam and microphone. Silverlight has excellent support for all types of webcams and mics, cross-browser and cross-platform.

In appendix D, you'll look at some of the windowing and navigation elements available to help you structure your applications, media, or other.

appendix D: *Introduction to* *WCF RIA Services*

Data-oriented Silverlight applications are multitier by nature—they have a client, a server with services, and a data store. As you learned in chapter 19, the way Silverlight handles network calls requires setting up asynchronous proxies (or performing raw asynchronous network operations). Sometimes, sharing entities between the client and server is a simple task; sometimes it's not. In general, the amount of code that goes into what could be considered plumbing and standard CRUD methods ends up being a significant portion of the overall source code for the application.

In many organizations, the code that makes up those plumbing and standard operations despite best efforts ends up being duplicated in project after project, which requires repeated testing of what is, essentially, the same code. Reuse is rarely seen, and when it is, it's in relatively trivial things such as logging services or caching. When reuse is enforced, it can be overly cumbersome to use across the suite of applications and difficult to update.

When developing WCF RIA Services (often called *RIA Services* for short), Microsoft realized that most applications built (again, despite best efforts) are actually minisilos from the client through to the database interface, and often through to the database tables themselves. I know from personal experience at many clients around the country that this is true—it's our industry's dirty little secret, despite all the talk about OOP reuse, service-oriented architecture (SOA), and more. Applications have a silo of functionality they use and some minor integration points with other systems using web services. I bring this up to point out that a nongoal of RIA Services is the creation of robust SOA solutions, in the true sense of SOA, not the “we used a service” sense.

Don't get me wrong—this isn't as bad as it sounds. In fact, the developer community has been coming around more to a YAGNI (“You Ain't Gonna Need It”) and

KISS (“Keep It Simple, Stupid”) agile approach these days. Often this is summed up simply as GSD (Get Stuff Done)¹—building only what you need while following good development practices makes it possible for you to deliver in a timely fashion.

WCF RIA Services is a framework and set of tools that attempts to make building modern multitier applications as simple as building classic two-tier client/server applications. WCF RIA Services doesn’t tie you to the single application model, but it’s optimized to support it as the most prevalent application model. I’m talking about building real, scalable, efficient, and easily coded multitier applications that work cleanly from front to back using a minimum amount of ceremonial code. This is accomplished through a framework and set of tools that provide the following benefits:

- Automatic creation of common Create Read Update Delete (CRUD) methods for entities
- Automatic generation and synchronization of service methods and their client-side proxies
- Validation rules and arbitrary business logic methods that are shared between the client and server without duplication of effort
- High-level client-side data source controls that make data manipulation simple
- Integration with ASP.NET security
- Through the project template, an overall application structure you can build on

In addition, when combined with the `DataGrid` and `DataForm` covered in chapter 17, you get automatic UI generation for entities, as well as simple UI wire-up for CRUD operations and validation.

This is all done in a way that allows you to maintain the level of control you want. There are enough extension points to let you hook into processes as well as manage client operations from code rather than the controls if you desire. Although optimized for the full application front-to-back scenario, it’s flexible enough to incorporate other services and even other RIA Services servers into the overall solution. You can even expose your RIA Services service calls and data in a number of ways to allow interoperating with other systems.

Although RIA Services does support other clients such as ASP.NET and jQuery/JavaScript, the functionality is at its strongest when used with Silverlight. Throughout its development, RIA Services was almost exclusively a Silverlight technology, giving back to the framework as techniques and code were developed. Most users of RIA Services are building Silverlight applications. The reason is simple: RIA Services helps solve a problem that’s strongest in Silverlight: how to build multitier data-oriented applications with different but mostly compatible frameworks on the client and server, without native database or ORM access from the client, and perform all requests asynchronously while keeping the footprint down.

¹ Yeah, yeah, so shoot me. I need to keep the text here pretty G-rated, although the gun reference in this footnote probably ruined that.

Our tour of RIA Services will start with a look at the tooling and templates that make it easy to use in Visual Studio. You'll create a project that'll be used in the examples in the rest of the appendix. After that, I'll show you what it takes to expose data to Silverlight, how to consume it from Silverlight both from code and markup, and how to filter, sort, group, and page that data. Of course, there's more to application development than read-only data, so you'll go through the update process to make sure the data can make a full round-trip. Then, you'll learn how to support loose coupling in an otherwise tightly coupled system. This appendix wraps up with a look at where to put business logic, followed by a discussion on securing your applications.

I'm excited about the efficiency that RIA Services brings to the table, so let's get building.

D.1 WCF RIA Services architecture, tooling, and template

WCF RIA Services applications are similar to standard Silverlight applications in that there's both a client application and a "home" server. The server serves up the Silverlight application and contains the services the application is to use. RIA Services works with multiple-server and multiple-client scenarios, but as mentioned in the introduction, the typical scenario is one server per application domain. Figure D.1 captures this typical architecture at a high level.

At first glance, the architecture looks like any other Silverlight application, except for that odd shared bit. That's one of the many things that make RIA Services worth the effort to learn.

RIA Services includes strong support for creating client-side proxies and entities that preserve, with high fidelity, the validation rules and logic written on the server. As a developer, you need to write the code only once, and RIA Services will take care of the rest. I'll cover this in depth later in the appendix.

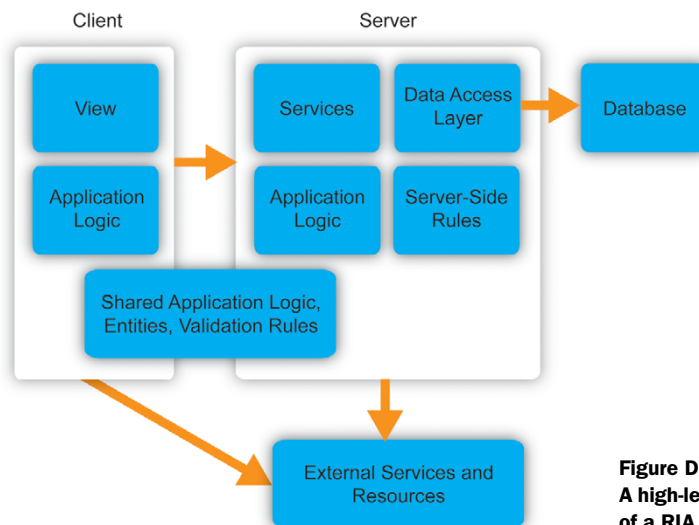


Figure D.1
A high-level view of the architecture
of a RIA Services application

In this section, you'll learn about the tooling that makes RIA Services work. Then, you'll dive right into creating a new project using the Silverlight Business Application template, a WCF RIA Services version of the navigation template covered in chapter 29. You'll build on this project throughout the rest of this appendix and the next.

D.1.1 RIA Services tooling support

Much of what makes WCF RIA Services tick is the magic that happens as part of the build process. When you first create a Silverlight application and select the option Enable WCF RIA Services, you've set up a client-to-server project link. That option puts a single line in the Silverlight .csproj project file:

```
<LinkedServerProject>..\RiaExample.Web\RiaExample.Web.csproj  
➡ </LinkedServerProject>
```

That one line of XML makes possible the autogeneration of the client proxies, types, and more. That also means a Silverlight application can be directly attached to at most one RIA Services server. To get around this limitation, you can create Silverlight class library projects and allow them to link to different servers and use the class libraries in your own project.

If you're curious, check out the obj/Debug folder in your Silverlight project. Alongside the files generated by the base Silverlight tools, you'll find a number of files generated by the RIA Services tooling, to keep track of server references, source files, and more. It's mostly unicorn and rainbow² magic, but it's fun for the curious and perhaps helpful during an odd debugging session.

The main body of code that's generated falls under the Generated_Code folder on the Silverlight application. This includes a single .g.cs file with all the context and proxy classes, and one or more subfolders with the additional model classes. Because this code is autogenerated, you won't want to change it. But having the source code available is useful when you're trying to understand exactly what RIA Services is doing in the client application.

In the remainder of the appendix, feel free to inspect the .g.cs file and the rest of the code in the Generated_Code folder as you add methods to various server-side classes. After all, you should always be aware of what the tools are doing for you.

Now that you understand the relationship between the web project and the client project, you can create the start of an application using the Silverlight Business Application template.

D.1.2 Creating a project with the template

The Silverlight tools for Visual Studio 2010 include a WCF RIA Services solution template, based on the navigation template discussed in chapter 29. This template is

² If you're really and truly bored and need a break from reading, check out <http://cornify.com/> to add unicorns and rainbows to any website or photo. Warning: 1980s 5th grade girls' Trapper Keeper graphics overload.

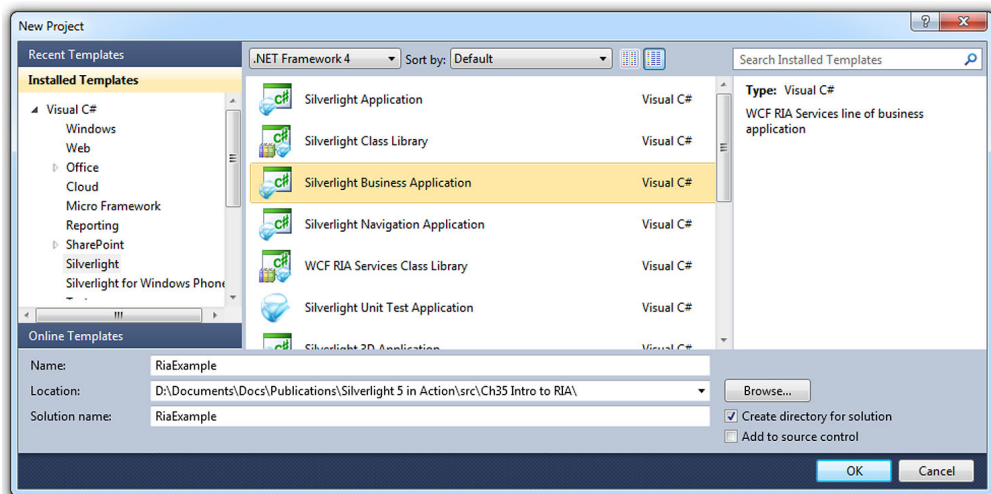


Figure D.2 Creating a WCF RIA Services application using the Silverlight Business Application template

called the Silverlight Business Application template. Although you don't need to use this template to create a RIA Services project (you need to select the Enable WCF RIA Services check box when creating a new Silverlight project as mentioned in the previous section), it does provide a good project structure to start with.

Figure D.2 shows the New Project dialog with this template selected. You'll use this project, *RiaExample*, in the rest of the appendix.

Note that when you create a new WCF RIA Services project, you're not prompted with the usual second New Project dialog, asking whether to create a website or enable WCF RIA Services. In a RIA Services project, both are required.

Despite the fact that they're based on the same original template, the styling steps described in chapter 29 won't work exactly with this template. When you run the application, you'll get something that looks similar to the chapter 29 template but with a few additions. Figure D.3 shows the bare application at runtime.

At runtime, the main difference you'll notice is the addition of the Login button. If you click that, you'll get a `ChildWindow` login/registration prompt. I'll discuss authentication in the next appendix.

The other changes, compared to the navigation application, require a little more digging.

APPLICATION RESOURCES

The Silverlight Business Application template has good support for customization and localization of the strings presented to the user. If you crack open the `Assets\Resources\ApplicationStrings.resx` file, you'll see that you can change key prompts, window titles, and more without altering the XAML.

Although not strictly required, when adding your own pages or prompts a best practice is to place the text in one of the three resource files (*ApplicationStrings*, *Error-*

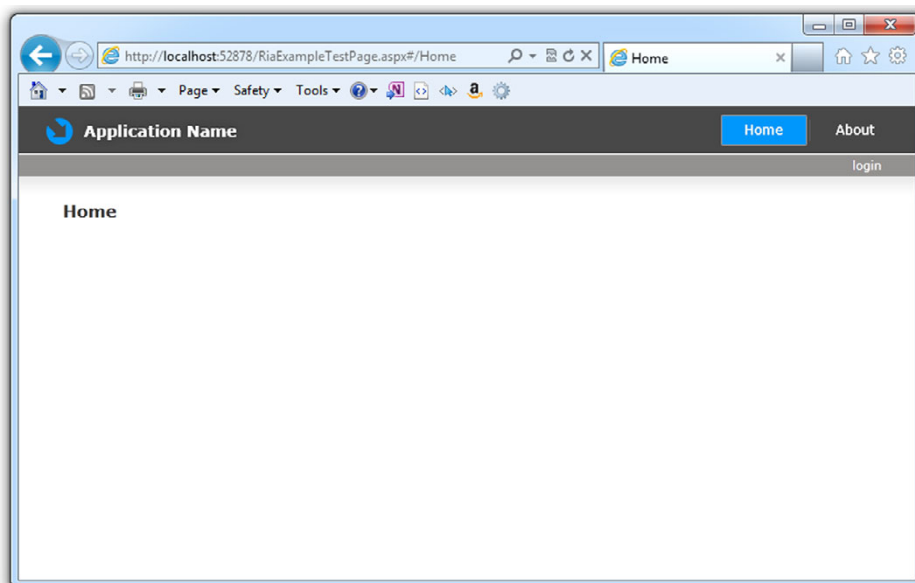


Figure D.3 The application when first run. Note the addition of the Login button on the right as compared to the navigation template shown in chapter 29.

Resources, or *SecurityQuestions*) rather than directly into XAML or code. Of course, you can create your own resource files if the text doesn't logically fit in one of these three.

To test the application resources approach, change the `ApplicationName` property in the `ApplicationStrings.resx` file to something different. I chose "WCF RIA Services Example." Run it, and you'll see the changed name. It doesn't change in the designer right away, but after a build (or build and run), you'll see the title update in the designer as well. In this way, the resource files don't block your design-time experience, yet they add significant opportunity for later configuration and internationalization.

How and why does this work? Open `MainPage.xaml`, and find the `TextBlock` named `ApplicationNameTextBlock`. Its definition looks like this:

```
<TextBlock x:Name="ApplicationNameTextBlock"
           Style="{StaticResource ApplicationNameStyle}"
           Text="{Binding Strings.ApplicationName,
           Source={StaticResource ApplicationResources}}"/>
```

The `TextBlock`'s `Text` value is bound to a property of the generated resource file class `Strings`. The resource property name is the same as that defined in the resource file. I've used traditional resource files before, and it was never this easy to get values into the UI. The power of binding in Silverlight makes using traditional resource files a no-brainer.

OTHER DIFFERENCES

The client project file has a number of other differences when compared to the straight navigation template. As you explore the project structure, you'll see additional

controls (such as the `BusyIndicator`), helper classes, additional views, and more. You'll run across many of them as you create your RIA Services application in the upcoming sections.

WCF RIA Services, especially through the use of the application template, makes it easy to structure a full business application following best practices. The tooling in Visual Studio helps automatically synchronize the client and server, avoiding a cumbersome manual step.

The architecture of WCF RIA Services, although geared toward Silverlight applications, is usable by other application types as well as through the server-side services. Let's leave the client project alone for a moment to concentrate on the server (web) project in order to learn how to expose data to the application.

D.2 Exposing data with the domain service

WCF RIA Services applications are typically used with a database back-end. It's possible to use something other than a database; RIA Services itself doesn't care what type of backing store you use, as long as a base domain service class exists for it.

Other Silverlight applications use a WCF, SOAP, or REST service server-side to access data. Those services, in the case of SOAP and WCF, typically expose methods for retrieving and updating data. They may expose domain methods to perform other functions or calculations as well. REST-based services typically expose a domain model in an entity-centric way, as you learned in chapter 21.

In an RIA Services application, the service to use is a *domain service*. A domain service, which is built on WCF, provides LINQ-based access to domain objects or data, as well as standard method-based service access to additional domain functions. It sits between the database and your client code, combining many of the advantages of the other services with the added bonus that the wire-up with the client happens automatically. The domain services are the heart of a WCF RIA Services application. It's the way your Silverlight client will communicate with server-side resources, get data, update data, and more.

In this section, you'll first create a domain service in the web project, then consume it from the Silverlight client. This section wraps up with an in-depth look at the common domain service methods for retrieving, updating, and deleting data, as well as explaining what it takes to add your own methods to the service.

D.2.1 Creating the domain service

For this project, you'll use the Entity Framework Model and the Adventure Works database. Follow the instructions in appendix A and set up the database, connection, and Entity Framework Model in the existing web project.

TIP At this point, set this appendix aside and go visit appendix A. Come back here once you have the database and entity data model all set up.

Build the project before your next step: adding the domain service. This will ensure that the appropriate metadata is available from the Entity Framework Model. When

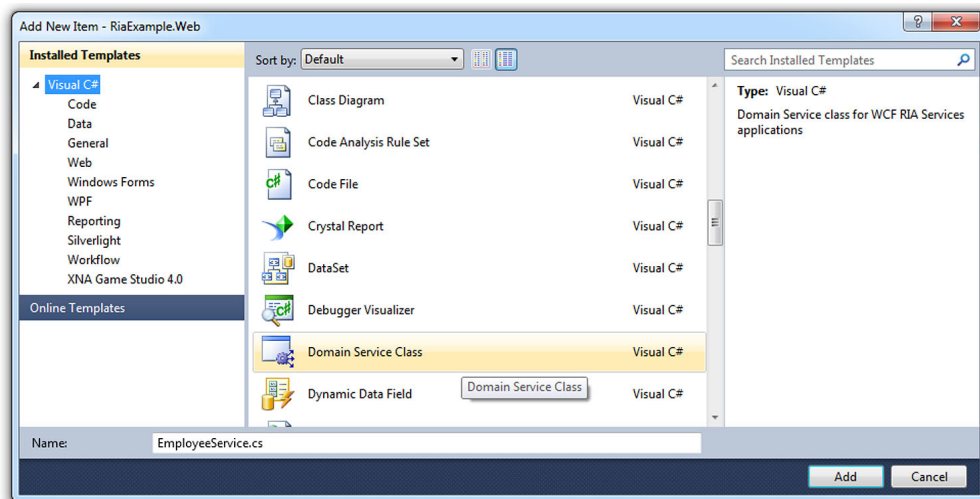


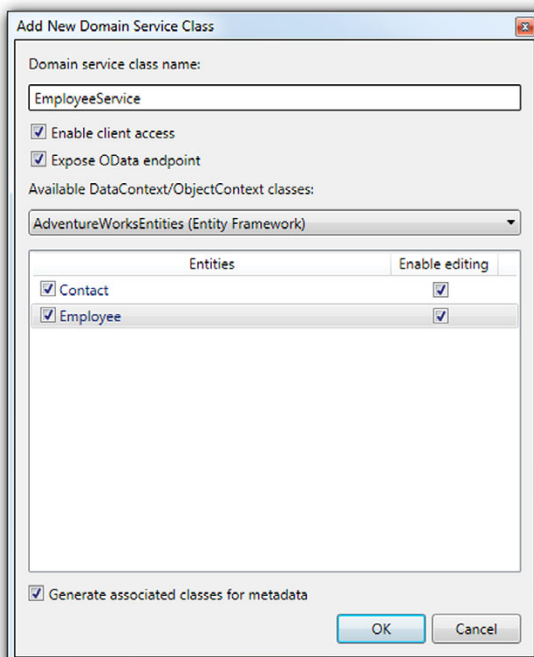
Figure D.4 Creating the `EmployeeService` domain service. You can find the **Domain Service Class** template in the top-level **Visual C#** template list.

that's done, right-click the **Services** folder in the web project (the **Services** folder was created as part of the business application template), and choose **Add New Item**. The item you want to add is the **Domain Service Class**, included in the top-level **Visual C#** template list in the **New Item** dialog. Figure D.4 shows the correct template in use.

Name your domain service `EmployeeService.cs` and click **Add**. You'll then be presented with the RIA Services-specific **Add New Domain Service Class** dialog shown in figure D.5.

This dialog requires careful attention. First, you want to make sure the **Enable Client Access** option is checked. When checked, it allows the domain service to be used by clients such as Silverlight. If unchecked, the service will only be available server-side, which is of no real use to you here.

Figure D.5 The **Add New Domain Service Class** dialog. If your dialog entity list is empty, cancel and build the project. Pay extra special attention to the checked items.



The next option is Expose OData Endpoint. OData is an XML-based data format. For most projects, this is entirely optional, but because I'll be discussing OData in appendix E, it needs to be checked.

The middle of the dialog includes a list of entities from the Entity Framework Model. If this list is empty, you need to cancel the dialog and build the project to generate the required metadata. Select (check) each entity that will be handled from this domain service; typically this is only one entity, or a small number of highly related entities, such as you have in this case. By default, the service handles retrieve operations only; if you want to allow create, update, and delete as you do in the examples in this chapter, ensure that the check box under Enable Editing is selected.

Finally, the Generate Associated Classes for Metadata check box at the bottom of the dialog box is an important option. When selected, this creates a class you can use to provide attribute-based validation and metadata for each of the entities. This class is named `<DomainServiceName>.metadata.cs`.

If all the correct options³ are selected,⁴ when you click OK, the two classes (service and metadata) will be created in the Services folder on the web project. The `EmployeeService` class automatically includes all the appropriate domain service methods to perform CRUD operations on both the selected `Contact` and the `Employee` types.

With the solution set up properly, and having an understanding of its role in a RIA Services application, let's turn to working with the domain service you created in the web project.

D.2.2 Domain service method types

The methods in the domain service have names starting with `Get`, `Insert`, `Update`, and `Delete`. This naming convention allows for automatic wire-up of the operations with the client. This convention-over-configuration approach is common outside the Microsoft developer ecosystem and has made inroads in the web and WCF teams at Microsoft. You'll find it supported in many of the things the WCF team produces as well as things like ASP.NET MVC 4.

Conventions help reduce the ceremony in setting up your services. But conventions don't always work for everyone or in every situation. For instances where you'd rather not go with convention, you can use a series of attributes to make your choices explicit. Table D.1 shows the attributes, conventions, and their descriptions.

³ Double-check!

⁴ No, seriously, triple-check before hitting OK. You can't easily rerun this dialog, and knowing which files and functions to remove/modify to re-create your steps here won't be possible until you understand RIA Services better.

Table D.1 Naming conventions, equivalent attributes, and their purposes

Name prefix	Attribute	Purpose
(Any)	[Query()]	A method that returns data without any side effects. The usual approach is to prefix with <code>Get</code> , but any prefix is fine as long as the function returns an instance of an entity <code>T</code> , an <code>IEnumerable<T></code> , or an <code>IQueryable<T></code> .
Insert, Add, Create	[Insert()]	An operation that inserts a single entity into the data store. The method takes the entity as a parameter.
Update, Change, Modify	[Update()]	An operation that updates a single entity in the data store. The method takes the entity as a parameter.
Delete, Remove	[Delete()]	An operation that deletes an entity in the data store. The method takes the entity as a parameter.
(Any)	[Invoke()]	A business method that must be executed without tracking or deferred execution. It may or may not have side effects. Use only when one of the other method types can't be used.
(Any)	[Update()]	A named update with <code>UsingCustomMethod=true</code> set in the attribute. This is a purpose-built function that performs a specific type of update. An example may be a product discount or firing an employee.

One potential issue with relying just on naming conventions is renaming the function can completely break the application. Unless your team is very comfortable with convention-based programming, use the attributes. In the remainder of this section, I'll go through each of the types of operations on the domain service.

QUERY METHODS

Query methods are methods that return a single entity or a set of entities. The default query method generated by the template returns all instances of the entity in the data store. This allows you to further compose the query on the client with additional criteria to limit the result set.

Query methods may be indicated by convention or attribute, as shown previously. When using the attribute, you have a few options to set. These are shown in table D.2.

Table D.2 `QueryAttribute` members

Member	Description
<code>HasSideEffects</code>	Queries shouldn't typically have side effects that would alter data. If they do, set this property to <code>true</code> so clients can make decisions as to how to use the method. For example, an HTTP client may send a POST instead of a GET.
<code>IsComposable</code>	Set this to <code>true</code> if the query allows composing to add additional criteria.
<code>IsDefault</code>	Set this to <code>true</code> if this query is the default query for the entity type.
<code>ResultLimit</code>	This is the maximum number of results the method should return. Defaults to 0, which indicated unlimited results.

Creating a query method on the service is simple if you follow the naming and method signature conventions and understand LINQ as used by your database provider/ORM (LINQ to Entities and the Entity Framework in our case). For example, here's a query that returns only salaried employees:

```
public IEnumerable<Employee> GetSalariedEmployees()
{
    return from Employee emp in ObjectContext.Employees
           where emp.SalariedFlag == true
           select emp;
}
```

When the solution is compiled, the method is turned into a client-side method named `GetSalariedEmployeesQuery` on the generated `ObjectContext` domain context object (more on the context in a bit). The tooling handles this creation so you don't need to.

TYPES OF QUERY METHODS

Query methods fall into three primary buckets:

- Methods returning a single concrete instance of an entity
- Methods returning a collection or enumerable of zero or more entities
- Methods returning an `IQueryable` of the entity

The first two are easily understood, falling squarely into patterns you've used since functions were first conceptualized in computer science. The third option is a little different and provides real flexibility.

A function with an `IQueryable` return type returns an expression tree. This is a LINQ concept for a generic query that's to be executed by a query provider. The `IQueryable` interface itself inherits from `IEnumerable`, so it also represents the results of that expression tree. Even when you build the LINQ query on the client, the query itself is executed server-side, typically all the way back at the database for a provider such as the Entity Framework.

In effect, this means you can have this query method on the server:

```
public IQueryable<Employee> GetEmployeesSorted()
{
    return from Employee emp in ObjectContext.Employees
           orderby emp.Title, emp.HireDate
           select emp;
}
```

and use it like this on the client:

```
ObjectContext context = new EmployeeContext();
EntityQuery<Employee> query =
    from emp in context.GetEmployeesSortedQuery()
    where emp.SalariedFlag == true
    select emp;
```

Note that the query is composed—the server-side query and the client-side query are combined to return a set of results. The server side by itself handles only the sort. The client side checks the `salaried` flag. These aren't run serially—that is, the client doesn't get the sorted list and then filter it, but the two steps happen together on the server, potentially even being passed into the database after being translated into dynamic SQL.

That's a powerful way to provide prefiltered or presorted data to the client. For example, the query could've taken a parameter to use in the filter or used security to decide which records could be returned to the client. The query execution itself is deferred; it's not executed until the client code first accesses the result data.

TIP One way to force execution of a query, should you want to, is to call `ToList()` (or any other method that forces access to the collection) on the resulting query object.

You'll learn more about using the domain service query methods from the client later in this appendix. Another class of methods the service provides is for data manipulation: insert, update, and delete operations.

INSERT, UPDATE, AND DELETE METHODS

The generated code for the insert, update, and delete methods takes in a single entity and uses the backing data store to perform the appropriate operation. For example, the update code looks like this:

```
public void UpdateEmployee(Employee currentEmployee)
{
    this.ObjectContext.Employees.AttachAsModified(currentEmployee,
        this.ChangeSet.GetOriginal(currentEmployee));
}
```

That tells the server-side object context to add this employee and mark it in the modified state, using the passed-in employee object as the current state and the original object as the last-known state from the data store. The `Attach` and `AttachAsModified` functions are all provided by the Entity Framework. The specific function used for your data provider may vary.

For a given entity, it's unusual to create alternate general insert, update, and delete methods. Doing so would confuse RIA services, not to mention your fellow programmers. There's one exception—the *named update method*.

NAMED UPDATE METHODS

Normally, the update methods are handled automatically based on the state of the data. But you may have situations where you need to provide a custom update method that you'll call directly rather than let Silverlight infer the update operation for a particular entity during the `SubmitChanges` call on the domain context.

To mark an update operation as a named update operation, it needs to have the usual update operation signature and the `Update` attribute with `UsingCustomMethod = true`. Here's an example:

```
[Update(UsingCustomMethod = true)]
public void SpecialCascadedUpdate(Employee emp)
{
    ...
}
```

This approach exists to allow you to handle special cases related to business logic or database complexities. It's still called as part of the batched `SubmitChanges` call. If you want to immediately execute a function, another approach is available.

INVOKE METHODS

CRUD methods are called as part of a batch—the entities have the CRUD operations performed on them but aren't sent to the server for the actual action until the call to `SubmitChanges` is made on the client.

Invoke methods are normal methods you can use to perform some sort of calculation or return a piece of data. They're operations that need to be executed without change tracking or deferred execution. Invoke methods shouldn't be used to load data; that's what query methods are intended for. Returning an entity from an `Invoke` method bypasses the pattern and won't cause the appropriate change tracking and entity generation to occur on the client.

Although the `Invoke` attribute is optional, to be considered an invoke method a method shouldn't take entities as a parameter or return an entity, `IEnumerable`, or `IQueryable` of entities as a result.

A typical `Invoke` method, if there could be such a thing, might look like this listing. Go ahead and add that to the domain service on the server project.

Listing D.1 The `Invoke` method to add to the domain service

```
[Invoke()]
public int CalculateVacationBonus(DateTime hireDate)
{
    int vacationBonus;
    DateTime today = DateTime.Today;
    int yearsInService = today.Year - hireDate.Year;
    if (hireDate.AddYears(yearsInService) > today)
        yearsInService--;
    if (yearsInService < 5)
        vacationBonus = 10;
    else
        vacationBonus = 20;
    return vacationBonus;
}
```

It's a regular business method. Given that it's on the server, you probably have a reason—it may call another web service, or it may hit a database to do a lookup. In this case, it's on the server simply to illustrate the invoke type.

As mentioned, the `Invoke` attribute is optional. When in doubt, add the attribute to make your intentions clear. For normal CRUD methods where the name is sufficiently patterned using the naming conventions, this is usually unnecessary. But I find that `Invoke` methods can be ambiguous at first glance. Speaking of naming conventions, what happens when you want to avoid having them kick in?

IGNORING METHODS DESPITE THE NAME

Some of these operations require the use of attributes, but many are autogenerated via the naming conventions. If you don't want RIA Services to generate a domain method for your service method, apply the `Ignore` attribute to that method, as shown here:

```
[Ignore()]
public void UpdateEmployeeButNotReally(Employee emp)
{
    ...
}
```

With that attribute in place, despite the fact that the method uses the *Update* naming convention and method signature, it won't be generated as an update call on the client. `Ignore` does exactly what you'd expect it to do.

The domain service provides a number of standard method types, many of which are autogenerated from the tooling but may be modified or replaced. Domain services provide CRUD operations in the form of insert, update, delete, and query methods. In addition, arbitrary functionality may be included in invoke methods.

When discussing the `IQueryable` type, I snuck an `EmployeeContext` object into the example. What's that, and what does it provide? That's the subject of the next section.

D.3 Using a domain service from Silverlight

Domain services execute on the server, running under the full .NET 4 Framework. The client-side equivalent of the domain service is the domain context object. Domain context objects provide a proxy for the service methods, as well as change tracking, operation batching, and more. If you think back to our discussion on networking in chapter 20, this is akin to the generation of WCF service proxies for SOAP services.

For each domain service on the server, RIA Services will generate one domain context object on the client. In the case of the `EmployeeService` domain service, the client domain context is named `EmployeeContext`. The domain service may be wired up to Silverlight via RIA Services controls in the UI that go through the context object, or via explicit use of the context object in code. Both approaches have advantages and disadvantages and will impact the overall architecture of your application.

I'll cover both approaches in this section. First, you'll learn how to connect via code—something very useful for your view models and other code-only connection approaches. Then, because it's simply the fastest and easiest way to use RIA Services, you'll look at the XAML approach.

D.3.1 Connecting via code

One way to use the domain service is to reference the client context object from code and execute queries directly against it. Because this is the most traditional way when compared to the usual pattern of working with services and WCF service proxies, let's start with it. First, you'll need a little UI.

You can't test the connection without having something to bind it to. So, it's time for a trusty `DataGrid`. Replace everything else in the `LayoutRoot`, starting with the `ScrollView`, with the XAML for the page layout and `DataGrid`, so the page looks like this:

Listing D.2 New XAML for Views\Home.xaml.cs

```
<navigation:Page x:Class="RiaExample.Home"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:navigation="clr-namespace:System.Windows.Controls;
  ➡ assembly=System.Windows.Controls.Navigation"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480"
  Style="{StaticResource PageStyle}">

  <Grid x:Name="LayoutRoot">
    <Grid Margin="10">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="350" />
      </Grid.ColumnDefinitions>
      <sdk:DataGrid x:Name="EmployeeGrid"
        Grid.Column="0" Margin="5" />
    </Grid>
  </Grid>
</navigation:Page>
```

← DataGrid

The code creates a two-column page, the first column of which includes a `DataGrid`. Be sure to add the `sdk` namespace at the top for the `DataGrid` control.

In the `Views\Home.xaml.cs` code-behind file in the client project, replace the `OnNavigatedTo` method with the `OnNavigatedTo` method in the next listing. Be sure to build the project beforehand so the `EmployeeContext` gets generated.

Listing D.3 Code-behind with new `OnNavigatedTo` method

```
using System.ServiceModel.DomainServices.Client;
using System.Windows.Controls;
using System.Windows.Navigation;
using RiaExample.Web;
using RiaExample.Web.Services;
```

```

namespace RiaExample
{
    public partial class Home : Page
    {
        public Home()
        {
            InitializeComponent();

            this.Title = ApplicationStrings.HomePageTitle;
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            EmployeeContext context = new EmployeeContext();
            EntityQuery<Employee> query = context.GetEmployeesQuery();
            context.Load<Employee>(query);
            EmployeeGrid.ItemsSource = context.Employees;
        }
    }
}

```

← **OnNavigatedTo**

← **Grid binding**

When the page is navigated to, this code automatically loads all the employees and assigns that collection to the `ItemsSource` of a `DataGrid`. The `EmployeeContext` object, in this instance, serves as the proxy for the domain service. Note that though you don't bother to hook up a method to the `Load` method asynchronous return, it's still executed asynchronously, and the results appear through binding.

The query system is flexible: you could change the client-side query to add criteria and a sort if you wanted to. (Be sure to add `using System.Linq;` to the top of the code before you try to compile.) For example, replace the query definition in `OnNavigatedTo` with this:

```

EntityQuery<Employee> query =
    from emp in context.GetEmployeesQuery()
    where emp.SalariedFlag == true
    orderby emp.HireDate
    select emp;

```

This example selects all the employees that are salaried and sorts them by hire date. The query itself is executed on the server, as you learned in the previous section. When using the Entity Framework with SQL Server as you are here, the query is executed all the way back at SQL Server, and only the items matching the query are returned.

When you run the application, you'll get something that looks like figure D.6. Note that I didn't use the sort and filter mentioned earlier; this is using simply `GetEmployeesQuery`.

Connecting to the domain service via code as you've done here allows you to better take advantage of advanced patterns such as MVVM and have complete control over

BirthDate	Contact	ContactID	CurrentFlag	Employee1	Employee2	EmployeeID	Gender	HireDate	LoginID
5/15/1972 12:00:00 AM		1209	<input checked="" type="checkbox"/>	Employee	Employee : 16	1	M	7/31/1996 12:00:00 AM	adventure-
6/3/1977 12:00:00 AM		1030	<input checked="" type="checkbox"/>	Employee	Employee : 6	2	M	2/26/1997 12:00:00 AM	adventure-
12/13/1964 12:00:00 AM		1002	<input checked="" type="checkbox"/>	Employee	Employee : 12	3	M	12/12/1997 12:00:00 AM	adventure-
1/23/1965 12:00:00 AM		1290	<input checked="" type="checkbox"/>	Employee	Employee : 3	4	M	1/5/1998 12:00:00 AM	adventure-
8/29/1949 12:00:00 AM		1009	<input checked="" type="checkbox"/>	Employee	Employee : 262	5	M	1/11/1998 12:00:00 AM	adventure-
4/19/1965 12:00:00 AM		1028	<input checked="" type="checkbox"/>	Employee	Employee : 109	6	M	1/20/1998 12:00:00 AM	adventure-
2/16/1946 12:00:00 AM		1070	<input checked="" type="checkbox"/>	Employee	Employee : 21	7	F	1/26/1998 12:00:00 AM	adventure-
7/6/1946 12:00:00 AM		1071	<input checked="" type="checkbox"/>	Employee	Employee : 185	8	F	2/6/1998 12:00:00 AM	adventure-
10/29/1942 12:00:00 AM		1005	<input checked="" type="checkbox"/>	Employee	Employee : 3	9	F	2/6/1998 12:00:00 AM	adventure-
4/27/1946 12:00:00 AM		1076	<input checked="" type="checkbox"/>	Employee	Employee : 185	10	M	2/7/1998 12:00:00 AM	adventure-
4/11/1949 12:00:00 AM		1006	<input checked="" type="checkbox"/>	Employee	Employee : 3	11	M	2/24/1998 12:00:00 AM	adventure-
9/1/1961 12:00:00 AM		1001	<input checked="" type="checkbox"/>	Employee	Employee : 106	12	F	3/3/1998 12:00:00 AM	adventure-
10/1/1946 12:00:00 AM		1072	<input checked="" type="checkbox"/>	Employee	Employee : 185	13	M	3/5/1998 12:00:00 AM	adventure-
5/3/1946 12:00:00 AM		1067	<input checked="" type="checkbox"/>	Employee	Employee : 21	14	M	3/11/1998 12:00:00 AM	adventure-
8/12/1946 12:00:00 AM		1073	<input checked="" type="checkbox"/>	Employee	Employee : 185	15	M	3/23/1998 12:00:00 AM	adventure-
11/9/1946 12:00:00 AM		1068	<input checked="" type="checkbox"/>	Employee	Employee : 21	16	F	3/30/1998 12:00:00 AM	adventure-
5/6/1946 12:00:00 AM		1074	<input checked="" type="checkbox"/>	Employee	Employee : 185	17	F	4/11/1998 12:00:00 AM	adventure-
9/9/1946 12:00:00 AM		1069	<input checked="" type="checkbox"/>	Employee	Employee : 21	18	M	4/18/1998 12:00:00 AM	adventure-
4/30/1946 12:00:00 AM		1075	<input checked="" type="checkbox"/>	Employee	Employee : 185	19	F	4/29/1998 12:00:00 AM	adventure-
6/15/1967 12:00:00 AM		1129	<input checked="" type="checkbox"/>	Employee	Employee : 173	20	M	1/2/1999 12:00:00 AM	adventure-
12/4/1972 12:00:00 AM		1231	<input checked="" type="checkbox"/>	Employee	Employee : 148	21	M	1/2/1999 12:00:00 AM	adventure-

Figure D.6 The DataGrid populated using the DomainDataSource control in XAML

the execution path. As you get more into code separation patterns, that can be a significant benefit.

TIP I set the `DataGrid.ItemsSource` property via code. There's no reason you couldn't set up a ViewModel (chapter 33) and bind the `ItemsSource` to an exposed `Employees` property. If you go with using the domain context object from code, follow the ViewModel/MVVM pattern when you do it; you'll thank yourself later. We'll cover this approach in detail in appendix F.

I'll cover the domain context class in more detail later in this appendix, primarily in section D.3 when I discuss update functionality.

There's another approach that's easier to use and includes a ton of built-in functionality. Before making up your mind which approach you want to use, look at the `DomainDataSource` control.

D.3.2 Using the DomainDataSource control

The `DomainDataSource` control provides an all-XAML way to interface with the domain service. I've heard this described as a bad thing, akin to wiring your UI directly to your database. I strongly disagree with that assessment, but I do agree that despite the utter simplicity of using the control, there are some drawbacks when it comes to testing, mocking, and application structure.

Before making up your mind that the control is a Bad Thing, let's look at what it can do. After all, some applications will benefit from this approach. Despite how it looks, it's not like you're binding VB3 UI controls directly to tables in an access database;⁵ there are a few layers of abstraction in between.

First, go into the code-behind for Home.xaml.cs and comment out the code in `OnNavigatedTo`. You'll no longer need any of it.

Then, to use the `DomainDataSource` control, you'll need to add a Silverlight assembly reference to the RIA Services SDK assembly `System.Windows.Controls.DomainServices.dll`. When that's done, inside the `LayoutRoot` `Grid` of `/Views/Home.xaml`, add the markup to add the `DomainDataSource` and its required namespace, as shown next.

Listing D.4 Updated Home.xaml with DomainDataSource

```
<navigation:Page x:Class="RiaExample.Home"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:navigation="clr-namespace:System.Windows.Controls;
  ➤ assembly=System.Windows.Controls.Navigation"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  xmlns:riaControls="clr-namespace:System.Windows.Controls;
  ➤ assembly=System.Windows.Controls.DomainServices"
  xmlns:domain="clr-namespace:RiaExample.Web.Services"
  mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480"
  Style="{StaticResource PageStyle}">

  <Grid x:Name="LayoutRoot">
    <riaControls:DomainDataSource x:Name="DataSource"
      AutoLoad="True"
      QueryName="GetEmployees">
      <riaControls:DomainDataSource.DomainContext>
        <domain:EmployeeContext />
      </riaControls:DomainDataSource.DomainContext>
    </riaControls:DomainDataSource>

    <Grid Margin="10">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="350" />
      </Grid.ColumnDefinitions>
      <sdk:DataGrid x:Name="EmployeeGrid"
        ItemsSource="{Binding Data, ElementName=DataSource}"
        Grid.Column="0" Margin="5" />
    </Grid>
  </Grid>
</navigation:Page>
```

Diagram annotations:

- An arrow points from the text **DomainDataSource** to the `<riaControls:DomainDataSource>` element.
- An arrow points from the text **EmployeeContext** to the `<domain:EmployeeContext />` element.

The markup in the listing sets up a new `DomainDataSource` control, tells it to automatically call the query when loaded, and sets the query name to the one that loads the employee information from the domain service. For this to work, you'll also need to set up both the `riaControls` and `domain` namespaces in the same XAML file.

⁵ I see the old VB3/4/5/6 VCR data-binding control in my nightmares from time to time. It's up there with the one about having a physics final today but having skipped the class all semester to spend time MUDDing.

The first namespace, `riaControls`, defines the location for the `DomainDataSource` control. The second, `domain`, defines the location for the generated domain context class: the client-side proxy for the domain service on the server. That proxy is the magic that links the UI to the domain service class created earlier.

TIP If you're curious about where the client-side proxy is defined and what it looks like, select the Silverlight project and, from the Project menu, select Show All Files. Scroll down, and you'll see a `Generated_Code` folder. In that folder, you'll find a number of interesting files, but the one that contains the proxies and entity definitions is `<projectname>.Web.g.cs`.

Because you're no longer setting the data source via code, the `DataGrid` needs to be bound to the new data source in XAML. The `DomainDataSource` object exposes its assigned context object's data through the `Data` property.

What you've changed on the grid is the `ItemsSource`. The markup here binds the `DataGrid` to the `Data` property of the `DomainDataSource`. Because the `DataGrid` instance is set up by default to autogenerate columns and show all data, you'll end up with an application that looks like the example in figure D.6 when run; the UI hasn't changed, just the way you get data on the client.

The `DomainDataSource` is easy to use. Although "Look Ma, no code!" isn't the most important reason to pick one approach over another (and in some cases can be a reason *not* to pick an approach), the domain data source is powerful and flexible enough to make it a real contender for how you connect to your domain service.

One other reason I like the `DomainDataSource` control is because both the team and the community are working to come up with better approaches that allow using that control with a ViewModel directly. Yep, using your ViewModel while still taking advantage of most of the coolness of the `DomainDataSource` is on everyone's radar.

The `DomainDataSource` and the underlying domain context objects support updating as well as querying, of course. But before you look at that, it's worth exploring some of the most compelling reasons to use the `DomainDataSource` control: filtering, sorting, grouping, and paging.

D.4 Filtering, sorting, grouping, and paging

User interfaces used to be simple to design because user expectations were so low. Character-mode terminals, difficult-to-memorize commands, and complex keystrokes that required keyboard function key overlays⁶ were the norm at one point, with some approaches persisting even into the GUI era.

As applications gained more chrome functionality, things such as sorting and grouping became expected functionality. In the mid-1990s, I developed applications in Visual Basic, and the users assumed they could do things like sort grids using

⁶ During the 1980s and '90s, there was a robust market for keyboard overlays for WordPerfect, WordStar, Lotus 123, and others. Most used the function keys in normal, Shift, Alt, and Control modes, all for different commands.

column headers, drag to rearrange, and so forth. Unfortunately, these assumptions didn't come out until user-acceptance testing.

These days, anything that helps meet the bar for base application functionality (for business applications, this is typically defined by what Microsoft Windows or Microsoft Office does in similar situations) is something I appreciate.

One reason I appreciate the `DomainDataSource` control is how well it integrates with other client-side controls to allow for filtering, sorting, grouping, and paging of the data. Any of those features, done right and done well, can amount to a fair bit of code and a testing burden.

Consider that you want to ensure they execute server-side for the best performance. You also have to handle the always-troublesome paging algorithms. What happens when users add a new item to a paged set? What happens when they sort? Fortunately, the RIA Services team has made intelligent decisions about behavior in each of these scenarios and implemented them into the code base.

You'll progressively add each of these capabilities—filtering, sorting, grouping, and paging—to the `DomainDataSource`-based version of your code, starting with filtering.

D.4.1 Filtering

Microsoft Excel and Microsoft SharePoint have brought filtering of table- or grid-based data up to the level of basic functionality for most applications. Proper filtering that performs efficiently isn't a huge effort, but it's a chunk of code that has to be maintained and tested. Having filtering support built in, so that all you need to provide is a filtering UI, is a huge benefit to most applications.

The first step is to create a basic single-field filter UI. Modify the controls in the home page XAML to replace the `DataGrid` with the three controls shown next.

Listing D.5 Updated `DataGrid` and filter controls

```
<TextBlock Height="23" Width="84" Margin="6,10,0,0"
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Text="Title Contains" />
<TextBox x:Name="FilterText"
    Height="23" Margin="96,6,5,0"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Top" />
<sdk:DataGrid x:Name="EmployeeGrid"
    Grid.Column="0" Margin="0 40 5 5"
    ItemsSource="{Binding Data, ElementName=DataSource}" />
```

← Filter

Be sure to leave the layout grid and other page elements intact. The markup in the listing creates space (via the margins on the `DataGrid`) and fills it with a `TextBlock` and `TextBox` that you'll use to gather filter information from the user. Although the `DomainDataSource` controls are smart enough to be able to filter on any column using a number of different operators, you'll go with a straight `contains` filter on a single field to keep things simple.

The next step is to wire the filter `TextBox`, named `FilterText`, to the data source and specify what field it'll operate on. Before you do that, let's look at how filtering is implemented on the `DomainDataSource` class.

FILTER DESCRIPTORS EXPLAINED

Filtering is implemented via two properties on the `DomainDataSource`. The first is the `FilterOperator`, which can be `And` or `Or` and controls how the filter descriptors are combined. The second is the collection of `FilterDescriptor` objects named, appropriately, `FilterDescriptors`.

Filter descriptors are discrete filter instructions that may be combined to produce an effective filter for a query. Conceptually, they're applied like a `where` clause in SQL, although the actual implementation is ultimately up to how the provider implements the composed where functionality in a LINQ query. Table D.3 shows the properties of the `FilterDescriptor` class.

Table D.3 Properties of the `FilterDescriptor` class

Property	Description
<code>IgnoredValue</code>	The value to be used for something like <code>(all)</code> , where you don't want any value appended to the filter.
<code>IsCaseSensitive</code>	If true, the filter is case-sensitive for string values. How this works depends on settings in the data store used by the domain service.
<code>Operator</code>	A <code>FilterOperator</code> that explains the relationship between <code>PropertyPath</code> and <code>Value</code> . Supported values are shown in table D.4.
<code>PropertyPath</code>	The path to the data item to be evaluated against the <code>Value</code> property. This is the property of your entity.
<code>Value</code>	The value to use for the filter condition.

Of the properties listed in the table, the relationship defined by the combination of the `PropertyPath`, `Operator`, and `Value` properties is the most interesting and the most relevant to filtering. A number of operators are supported, each of which is described in table D.4.

Table D.4 Values for the `Operator` property of the `FilterDescriptor`

Value	Description
<code>IsLessThan</code>	The data value must be smaller than the filter value.
<code>IsLessThanOrEqualTo</code>	The data value must be smaller than or equal to the filter value.
<code>IsEqualTo</code>	The data value must be equal to the filter value.
<code>IsNotEqualTo</code>	The data value must be different from the filter value.
<code>IsGreaterThanOrEqualTo</code>	The data value must be larger than or equal to the filter value.

Table D.4 Values for the Operator property of the FilterDescriptor (continued)

Value	Description
IsGreaterThan	The data value must be larger than the filter value.
StartsWith	The data value must start with the filter value (strings only).
EndsWith	The data value must end with the filter value (strings only).
Contains	The data value must contain the filter value (strings only).
IsContainedIn	The data value must be contained in the filter value (strings only).

It may seem somewhat redundant to list the descriptions for each of these values given their names, but there are three important bits of information to get from table D.4:

- A quite comprehensive set of filter operators is available.
- The order of the statement, read left to right, is *Property Operator Value*.
- Some of the operators make sense only on strings, because they perform sub-string operations.

The reason for the lengthy member names is twofold: you can't have operators like `>=` in XAML without ugly and unreadable escaping like `>=`, and you need an enumeration to set the property in XAML or from code. Primarily, the list is optimized for using from XAML.

Because it's optimized for XAML, you'd think the properties would all support binding—and you'd be right. It's possible to build a complete filter expression using filters created using binding, meaning you can provide the user with a drop-down list of fields, a drop-down list of operators, and a `TextBox` for the value. All five listed properties of the `FilterDescriptor` class are dependency properties that support binding.

USING FILTER DESCRIPTORS WITH THE DOMAINDATASOURCE

Despite the binding flexibility, you'll implement a simple filter where only the filter value itself is bound. You already have the `TextBox` for the value in place, so the next step is to add the associated `FilterDescriptor` to the `DomainDataSource`. The next listing shows the updated `DomainDataSource` filter, including the descriptor and `FilterOperator`.

Listing D.6 Updated DomainDataSource with the filter in place

```
<riaControls:DomainDataSource x:Name="DataSource"
    AutoLoad="True"
    FilterOperator="And"
    QueryName="GetEmployees">
    <riaControls:DomainDataSource.DomainContext>
        <domain:EmployeeContext />
    </riaControls:DomainDataSource.DomainContext>
```

← Filter group operator

```

<riaControls:DomainDataSource.FilterDescriptors>
  <riaControls:FilterDescriptor
    PropertyPath="Title"
    Operator="Contains"
    Value="{Binding Text, ElementName=FilterText}" />
</riaControls:DomainDataSource.FilterDescriptors>
</riaControls:DomainDataSource>

```

← Filter descriptor

The markup in the listing augments the `DomainDataSource` to add a `FilterDescriptor`. That `FilterDescriptor` targets the `Title` property of the `Employee` entity and checks to see that it contains (using the `Contains` operator) the current value in the `Text` property of the `FilterText` field on the same page. If you had more than one filter, the `FilterOperator` would come into play. For now, leave it as `And`.

When run, you'll have an experience like that shown in figure D.7.

Type in the *Title Contains* field, and pause for a second or two or tab off the field. The pause or loss focus will kick in the filter, executing the query on the server and displaying the results in the grid.

By adding only a few lines of XAML, you were able to add property-value filtering (which also works with sorting, grouping, and paging, as you'll see in the next sections) without having to wire up anything at the database level or even the service level. This makes sense. Like all the other features in this section, filtering should be a given for an application; there's little point in implementing the same tired old filtering code again and again. The same goes for sorting, the next topic.

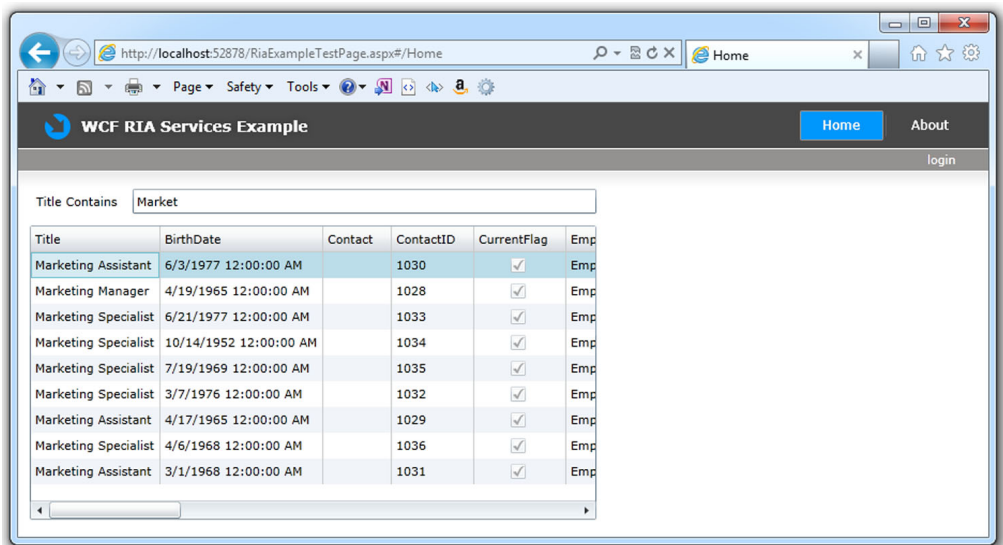


Figure D.7 Filtering the results to those that contain *Market* in the title. This was done entirely with the `DomainDataSource` and a little in-XAML binding.

D.4.2 Sorting

You may have already noticed that the `DataGrid`, when wired to the `DomainDataSource` (or any other `ICollectionView` or even `IList`), provides automatic sorting capabilities when you click column headers. The `DomainDataSource` also provides a way to perform a default sort on the data using `SortDescriptor` objects. For instance, to have the data sorted by `Title` and `HireDate` by default, you can add the XAML from the following listing to the inside of the `DomainDataSource` markup on `Home.xaml`.

Listing D.7 Updated DomainDataSource with both filtering and sorting

```
<riaControls:DomainDataSource x:Name="DataSource"
    AutoLoad="True"
    FilterOperator="And"
    QueryName="GetEmployees">
    <riaControls:DomainDataSource.DomainContext>
        <domain:EmployeeContext />
    </riaControls:DomainDataSource.DomainContext>
    <riaControls:DomainDataSource.FilterDescriptors>
        <riaControls:FilterDescriptor PropertyPath="Title"
            Operator="Contains"
            Value="{Binding Text, ElementName=FilterText}" />
    </riaControls:DomainDataSource.FilterDescriptors>
    <riaControls:DomainDataSource.SortDescriptors>
        <riaControls:SortDescriptor Direction="Ascending"
            PropertyPath="Title" />
        <riaControls:SortDescriptor Direction="Ascending"
            PropertyPath="HireDate" />
    </riaControls:DomainDataSource.SortDescriptors>
</riaControls:DomainDataSource>
```

Sort

When you run the application, you'll see that the `DataGrid` control itself isn't ignorant of the specified sort. In most applications, when you sort queries in the database, the client has no idea the data was sorted. With the `DomainDataSource`, the `DataGrid` is aware and indicates as much in the column headers. See figure D.8 for the proof in the column headers.

You can also sort server-side as part of the query code, as you've seen earlier in this chapter. In either case, sorting is recommended for grouping and required for paging.

Title ▲	HireDate ▲	BirthDa
Accountant	3/22/1999 12:00:00 AM	2/4/199
Accountant	4/9/1999 12:00:00 AM	8/1/199
Accounts Manager	3/3/1999 12:00:00 AM	8/8/199
Accounts Payable Specialist	3/15/1999 12:00:00 AM	3/18/19
Accounts Pavaable Specialist	4/2/1999 12:00:00 AM	4/9/199

Figure D.8
Data sorted with the `DomainDataSource`. Note the sort indicators in the column headers.

D.4.3 Grouping

Supporting grouping is as easy as sorting. Following the trend you've seen so far, grouping is also accomplished through a collection of descriptors. In this case, the descriptors are `GroupDescriptor` objects. For example, if you want to group on `Title`, you add the following tiny bit of XAML to the `DomainDataSource` markup, just as you did the Sort section:

```
<riaControls:DomainDataSource.GroupDescriptors>
  <riaControls:GroupDescriptor PropertyPath="Title" />
</riaControls:DomainDataSource.GroupDescriptors>
```

This relies on the previous sort for the grouping to make any sense. As expected, this integrates nicely with the `DataGrid`. Figure D.9 shows the `DataGrid` control with the new grouping in place.

With the grouping in place, you can still sort using the column headers, but the sort happens within the defined grouping.

The final and perhaps most interesting of the features is the support for paging.

D.4.4 Paging

There currently exist three main UI paradigms for dealing with a large number of records. You can preload everything and allow scrolling, you can implement an *infinite*

Title ▲	HireDate ▲	BirthDate	Contact	Co
Title: Accountant (2 items)				
Accountant	3/22/1999 12:00:00 AM	2/4/1966 12:00:00 AM		12
Accountant	4/9/1999 12:00:00 AM	8/1/1969 12:00:00 AM		12
Title: Accounts Manager (1 item)				
Accounts Manager	3/3/1999 12:00:00 AM	8/8/1973 12:00:00 AM		12
Title: Accounts Payable Specialist (2 items)				
Accounts Payable Specialist	3/15/1999 12:00:00 AM	3/18/1967 12:00:00 AM		12
Accounts Payable Specialist	4/2/1999 12:00:00 AM	4/9/1969 12:00:00 AM		12
Title: Accounts Receivable Specialist (3 items)				
Accounts Receivable Specialist	1/19/1999 12:00:00 AM	4/7/1966 12:00:00 AM		12
Accounts Receivable Specialist	2/7/1999 12:00:00 AM	3/26/1966 12:00:00 AM		12
Accounts Receivable Specialist	2/25/1999 12:00:00 AM	10/22/1974 12:00:00 AM		12
Title: Application Specialist (4 items)				
Application Specialist	1/24/1999 12:00:00 AM	3/3/1975 12:00:00 AM		12
Application Specialist	2/12/1999 12:00:00 AM	7/28/1971 12:00:00 AM		12
Application Specialist	3/7/1999 12:00:00 AM	4/14/1978 12:00:00 AM		12
Application Specialist	3/20/1999 12:00:00 AM	6/19/1968 12:00:00 AM		12
Title: Assistant to the Chief Financial Officer (1 item)				
Assistant to the Chief Financial Officer	2/13/1999 12:00:00 AM	7/23/1954 12:00:00 AM		12

Figure D.9 The `DataGrid` with grouping, courtesy of the `DomainDataSource` control

scroll that performs lazy fetching of additional data (a good example is the Bing image search), or you can use data paging. When the web started to define how we built applications, data paging became the most common way to deal with large volumes of data. After all, if it's good enough for Google, it must be good enough for your application, right?

I've never been a fan of paging, but it certainly has some advantages when it comes to getting a lot of information in front of a user while reducing network traffic and database load.

When you're building RIA Services applications, paging is accomplished with a combination of two items:

- The `PageSize` and `LoadSize` in the `DomainDataSource`
- A `DataPager` control

The `PageSize` property of the `DomainDataSource` controls how many items appear on a single page. The `LoadSize` property controls how many items the `DomainDataSource` loads into memory at one time. For example, if you have a `PageSize` of 15 and a `LoadSize` of 30, every other page will cause a network hit to the server to get the next 30 items. Because RIA Services doesn't know the usage pattern of your application, these two knobs are left entirely up to you.

The next listing shows the updated `Home.xaml` markup, including the `DomainDataSource` page settings and the new `DataPager` control. For this example, you'll set `PageSize` to 15 and `LoadSize` to 30. The next thing to do is to add a `DataPager` control (easiest if dragged onto the surface or markup) and change the margins on the `DataGrid` to make room at the bottom.

Listing D.8 Updated `Home.xaml` with paging

```
<navigation:Page x:Class="RiaExample.Home"
...
mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480"
Style="{StaticResource PageStyle}">

<Grid x:Name="LayoutRoot">
    <riaControls:DomainDataSource x:Name="DataSource"
        PageSize="15" LoadSize="30"
        AutoLoad="True"
        FilterOperator="And" QueryName="GetEmployees">
        ...
    </riaControls:DomainDataSource>

    <Grid Margin="10">
        ...
        <sdk:DataGrid x:Name="EmployeeGrid"
            Grid.Column="0" Margin="0 40 5 40"
            ItemsSource="{Binding Data, ElementName=DataSource}" />

        <sdk:DataPager Grid.ColumnSpan="2"
            Source="{Binding Data, ElementName=DataSource}"
```

PageSize and LoadSize

←

DataPager

←

```

HorizontalAlignment="Stretch"
VerticalAlignment="Bottom" />

</Grid>
</Grid>
</navigation:Page>

```

In the listing I've left out the majority of the markup, which is identical to the previous listings. With all of the markup in place, run the application and navigate through the pages. The application, still with sorting and grouping in place, should look like figure D.10.

For you to make the most of the `DataPager`, its source must be set to an `IEnumerable` that implements the `IPagedCollectionView` interface, an example of which is, as you probably guessed, the `DomainDataSource` control. The data must also be sorted, either via the query on the server or via sorting specified in the `DomainDataSource`. If the data isn't presorted, you'll get an exception at runtime.

DATAPAGER PROPERTIES

The `DataPager` is a fully templatable control, supporting the lookless model XAML is famous for. In addition, the `DataPager` has a number of properties that control its behavior and appearance.

In addition to helpful utility properties such as `CanMoveToFirstPage` and `CanMoveToNextPage`, the `DataPager` includes a `DisplayMode` property that's used to control which buttons and boxes are shown in the UI. Table D.5 shows the different values this property can be set to.

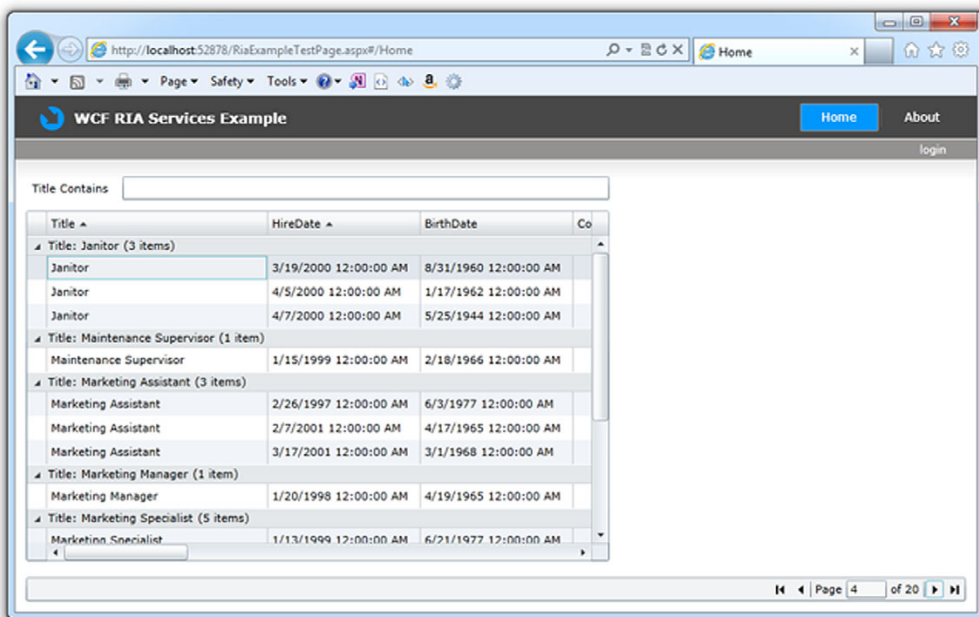

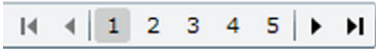

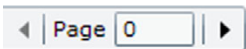
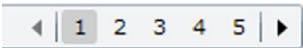


Figure D.10 The `DataPager` in use with a page size of 15 and a load size of 30

As you can see, the control provides a number of paging interfaces, covering the gamut typically seen in applications and on the web. For the ones that show page numbers, you can use the `NumericButtonCount` property to control how many numbers are displayed. In addition, you can use the `AutoEllipsis` property to display an ellipsis, rather than a number, to indicate more pages.

Table D.5 `DisplayMode` property values and their associated UI

Property	Runtime appearance
<code>FirstLastNumeric</code>	
<code>FirstLastPreviousNext</code>	
<code>FirstLastPreviousNextNumeric</code>	
<code>Numeric</code>	
<code>PreviousNext</code>	
<code>PreviousNextNumeric</code>	

The `DomainDataSource` control makes it easy to add common data-browsing capabilities—filtering, sorting, grouping, and paging—to your applications. Combined, these are high-value, high-effort development tasks in most applications. Having the functionality built in saves you from having to reinvent the wheel or tell your customer “no” when the feature is requested (or worse, assumed).

So far, everything you’ve done has been with read-only data. Real applications typically need to update data as well.

D.5 Updating data

Most data-oriented applications have to do more than read data; they need to perform inserts, updates, and deletes as well. In the discussion about the domain service methods, I touched on the three data modification methods that begin with the prefixes `Insert`, `Update`, and `Delete`.

WCF RIA Services makes updating data as easy as retrieval. The domain service methods are trim, and they’re autogenerated for the usual cases. The client-side domain context methods (which I’ll cover in D.5.2) that provide access to those services are also autogenerated.

In this section, you'll start with creating a UI using the `DataForm` that allows you to update the data in the domain service. You'll then look at the client-side counterpart of the domain service: the domain context. Finally, you'll go through how the entity class and its buddy class with validation and display metadata work together to make it easier to have a robust and feature-rich data container on the client.

D.5.1 Using the `DataForm` UI

The `DataForm`, like the `DataGrid`, is extremely powerful when matched up with WCF RIA Services and the `DomainDataSource` control. The `DataForm`, in fact, was originally part of WCF RIA Services before it was pulled out and made part of the Silverlight Toolkit. The `DataForm` is covered in full in chapter 17, so I won't repeat that content here. But you'll use it to provide the update UI for the entities in this application.

The right side of the page has been empty so far. You've been leaving room for the `DataForm`. The next listing shows the chunk of markup with the `DataForm` added, right after the `DataGrid` element and before the `DataPager` element. Be sure to add the proper `DataForm` namespace to the top of the XAML file as well:

```
xmlns:dataForm="clr-namespace:System.Windows.Controls;
    ➤ assembly=System.Windows.Controls.Data.DataForm.Toolkit"
```

Listing D.9 Adding the `DataForm` element to `Home.xaml`

```
<sdk:DataGrid x:Name="EmployeeGrid"
    Grid.Column="0" Margin="0 40 5 40"
    ItemsSource="{Binding Data, ElementName=DataSource}" />

<dataForm:DataForm Grid.Column="1" Margin="5 40 0 40"
    ItemsSource="{Binding Data, ElementName=DataSource}"
    CurrentItem=
    ➤ "{Binding SelectedItem, ElementName=EmployeeGrid, Mode=TwoWay}" />

<sdk:DataPager Grid.ColumnSpan="2"
    Source="{Binding Data, ElementName=DataSource}"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Bottom" />
```

← **DataForm**

This listing sets up a `DataForm` that uses the same `ItemsSource` as the `DataGrid`, so it's also bound to the `DomainDataSource` control. The `CurrentItem` property is bound to the `DataGrid`'s selected item, keeping the form in sync with what's shown in the `DataGrid`. Note that the binding is two-way, so the `DataForm` navigation controls can also be used to change the selected item in the grid. Figure D.11 shows the application with the new addition.

Navigate around using the grid and the navigation buttons at the upper right. When you're sure it's all working, you'll wire up the save functionality.

SAVING CHANGES

To submit the changes to the server, you need to have a button wired up to the `SubmitChangesCommand` of the `DomainDataSource`. That command does the equivalent

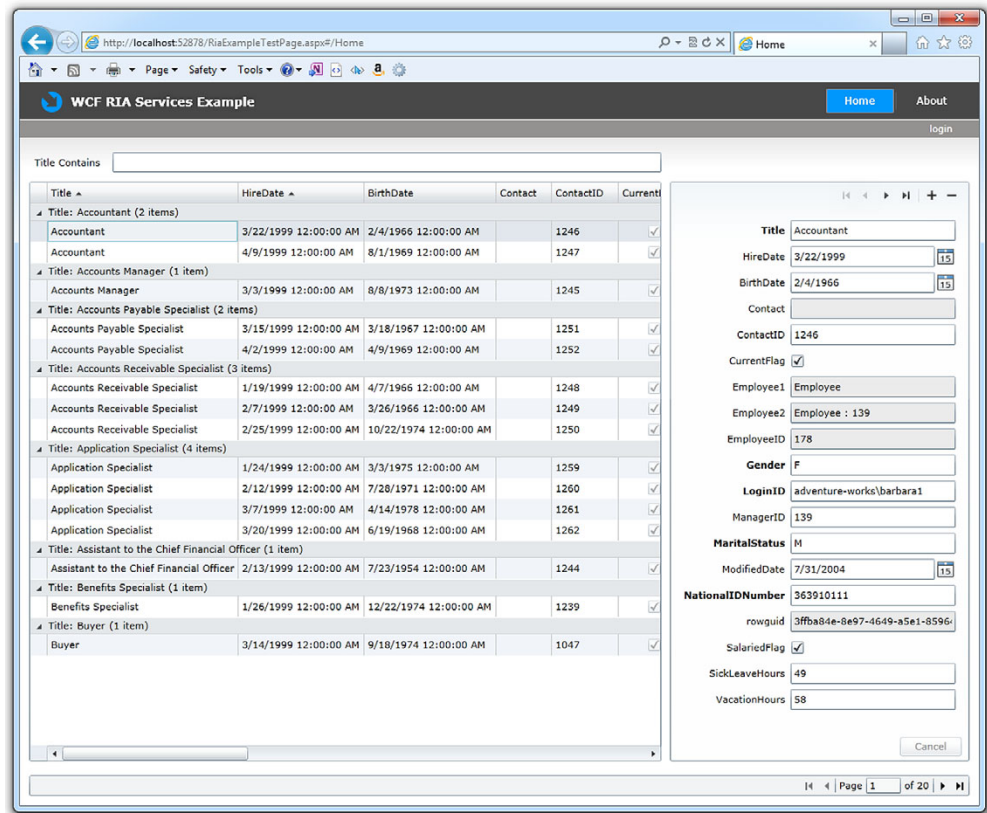


Figure D.11 The application with the details DataForm on the right, populated from the selected grid item. Row navigation works from both the grid and the DataForm.

of calling `SubmitChanges` on the domain context from code. Place the button shown in this listing immediately below the `DataForm` markup.

Listing D.10 The `SubmitChanges` button markup for saving

```
...
<sdk:DataPager Grid.ColumnSpan="2" ... />

<Button x:Name="SubmitChanges"
        Grid.Column="1" Margin="5"
        HorizontalAlignment="Right"
        VerticalAlignment="Top"
        Height="25" Width="120"
        Command="{Binding SubmitChangesCommand, ElementName=DataSource}"
        Content="Submit Changes" />
```

The code adds a Submit Changes button at the upper right on the screen. Modify some data and give it a try.

In theory, you have a fully working application at this point; you can perform CRUD⁷ operations using the UI. Use the + button to add a new record and the – button to delete the current record. When you’re finished, click the new Submit Changes button to call the `SubmitChanges` function behind the scenes. This function, like almost everything else in the `DomainDataSource` control, relies on the generated domain context object. In this case, it’s the `EmployeeContext`.

D.5.2 The domain context

One of the types of classes that’s generated based on the domain service is a client-side domain context. The domain context is 1:1 with the domain service. In your solution, for example, you have an `EmployeeService` domain service and an `EmployeeContext` domain context.

In addition to the previously seen query methods, the domain context has a number of properties and methods. The most commonly useful are shown in table D.6, using `Employee` as the example.

Table D.6 Properties and methods of the generated domain context class

Member	Description
<code>CalculateVacationBonus</code> method	The method generated from your server-side <code>Invoke</code> operation
<code>EntityContainer</code> property	Internal, but important for holding the actual entities and tracking insert and delete operations
<code>HasChanges</code> property	True if the domain context is tracking any entities with changes (updates, inserts, deletes)
<code>IsLoading</code> property	True if the domain context is loading data
<code>IsSubmitting</code> property	True if the domain context is submitting changes
<code>RejectChanges</code> method	Rejects all pending changes and reverts objects to their unedited state
<code>SubmitChanges</code> method	Sends all pending change operations to the domain service for processing

INVOKE OPERATIONS

In this example, a client-side invoke operation was created for the `CalculateVacationBonus` function you added to the domain service. Because all network calls in Silverlight are asynchronous, you can’t call the function and get the result. Instead, you need to set up a callback. For example, the next listing includes the client-side code to

⁷ Note that due to the relationship with the `Contact` object and other relationships, deletes and inserts currently fail. Updates work fine. You’ll take care of that later in this appendix.

call the `CalculateVacationBonus` function and do something useful with the results. Add it to the code-behind `Home.xaml.cs`.

Listing D.11 Calling an invoke operation from the client

```
private void CalculateBonus()
{
    var context = DataSource.DomainContext as EmployeeContext;
    var emp = EmployeeGrid.SelectedItem as Employee;
    if (emp != null)
    {
        DateTime hireDate = new DateTime(2002, 05, 16);
        var invokeOp = context.CalculateVacationBonus(
            hireDate, OnInvokeCompleted, emp);
    }
}

private void OnInvokeCompleted(InvokeOperation<int> invokeOp)
{
    if (invokeOp.HasError)
    {
        MessageBox.Show(invokeOp.Error.Message);
        invokeOp.MarkErrorAsHandled();
    }
    else
    {
        Employee emp = invokeOp.UserState as Employee;
        if (emp != null)
        {
            emp.VacationHours += (short)invokeOp.Value;
        }
    }
}

private void CalculateBonusButton_Click(object sender, RoutedEventArgs e)
{
    CalculateBonus();
}
```

Get Context

Execute Invoke operation

Once the code is added, update the XAML UI with a new Calculate Bonus button placed right under the Submit Changes button in the markup:

```
<Button x:Name="CalculateBonusButton"
        Grid.Column="1"
        Margin="5 5 130 5"
        HorizontalAlignment="Right"
        VerticalAlignment="Top"
        Height="25"
        Width="120"
        Click="CalculateBonus_Click"
        Content="Calc Bonus" />
```

This code, from the code-behind for `Home.xaml`, shows how to call an invoke method that modifies the data on the client. Note the parameters to the `CalculateVacationBonus` client-side method. On the server, the method took only a single parameter. On

the client, it takes that same parameter, plus a callback and a data item. In this case, the data item is the `Employee` you're working with. You use that because you need access to the `Employee` inside the callback method.

The callback method executes when the asynchronous call has completed. The single parameter for the callback is an `InvokeOperation` object with a number of properties, including the `UserState` and error information.

In this method, you check for an error. If there's no error, you cast the `UserState` back to an `Employee` object, check it for `null`, then use the function return value (the calculated bonus) and add that to the existing vacation hours. That object is marked as `HasChanges = true` on the entity. The entity is then eligible for the `SubmitChanges` call.

SUBMITCHANGES

Referring back to table D.6, you'll notice that no `Insert`, `Update`, or `Delete` methods were generated. Instead, those are called via `SubmitChanges`.

`SubmitChanges` is an asynchronous batching operation. It handles sending all method calls, with the exception of `Invoke` and `Query` operations, to the server.

When you insert new items or delete existing items, those operations occur only on the client. When you call `SubmitChanges`, it loops through the entities on the client and sends to the server those entities that require a persistence operation, calling the appropriate operation for each entity.

To cancel all pending changes for the domain context, call the `RejectChanges` method. It reverts entities to their previous state, removes any newly inserted items, and reinstates any deleted items.

The domain context is the client-side proxy for the domain service, as well as the container within which all instances of a given entity reside. It provides an interface for invoke operations and query operations, as well as an implicit interface to the insert, update, and delete operations through the `SubmitChanges` method.

The entity classes `Employee` and `Contact` both inherit from a common client-side base class that provides much of the required change-tracking and other plumbing functionality. This class is named, appropriately enough, `Entity`.

D.5.3 The Entity class

Each client-side entity you work with, `Employee` and `Contact` in this example, derives from the `Entity` base class. This class provides a number of important change-tracking properties and methods. Table D.7 shows the most important public members of the `Entity` base class.

Table D.7 Important public members of the `Entity` class

Member	Description
<code>EntityState</code>	The data state of this entity: Detached, Unmodified, Modified, New, or Deleted
<code>HasChanges</code>	Indicates that this entity has changed since the last time it was saved

Table D.7 Important public members of the Entity class (continued)

Member	Description
HasValidationErrors	Indicates that this entity has failed validation
ValidationErrors	Returns a collection of validation errors
GetOriginal	Returns an instance of the unchanged entity from cache

Your derivations of the `Entity` class (the `Contact` class and the `Employee` class) also include all the individual properties that correspond to the fields coming from the database. Because this code was generated by the tools and not shared with the server, the properties have `INotifyPropertyChanged` and several other events injected into them. In this way, your otherwise plain classes on the server can support binding and events on the client. To give you an idea of the robustness of the properties set up, this listing shows the `Gender` property for the `Employee`.

Listing D.12 The generated client-side `Employee` Entity property `Gender`

```
[DataMember()]
[Required()]
[StringLength(1)]
public string Gender
{
    get
    {
        return this._gender;
    }
    set
    {
        if ((this._gender != value))
        {
            this.OnGenderChanging(value);
            this.RaiseDataMemberChanging("Gender");
            this.ValidateProperty("Gender", value);
            this._gender = value;
            this.RaiseDataMemberChanged("Gender");
            this.OnGenderChanged();
        }
    }
}
```

Validation attributes

The setter for the property includes a number of calls to generated methods. Those methods perform validation and take care of `INotifyPropertyChanged` notification as well as raise information events, such as `DataMemberChanging` and `DataMemberChanged`.

In this example, the `OnGenderChanging` and `OnGenderChanged` methods are partial methods that you can implement in a buddy class on the client, should you wish. A *buddy class* is a partial class you create to augment an existing partial class. In this way, you can modify the behavior of the class without introducing an inherited class.

Note the use of attributes to tell the UI that this is a required field with a maximum length of 1. This information was automatically inferred from the entity model on the server at code-generation time. For that reason, changes to the database will require updates to the EDMX model and automatic downstream updates here.

In addition to the validation and display attributes described in chapters 17 and 18 and shown in this example, a number of other attributes are used in the entity. You'll see how to use the special validation and display metadata attributes in the next section, but in the meantime table D.8 shows some of the helper attributes you'll likely run across.

Table D.8 Interesting attributes on the Employee Entity

Attribute	Description
<code>DataMember</code>	Indicates that this property should be serialized by WCF and is part of the data contract.
<code>Association</code>	Specifies that the property is part of a relationship, such as a foreign key. You'll find this on the nested entities such as <code>Contact</code> .
<code>XmlIgnore</code>	Indicates that this property shouldn't be serialized. Useful on nested entities.
<code>RoundtripOriginal</code>	Sends the object back to the server with its original value when the object is updated, even though this property hasn't changed.
<code>Key</code>	Indicates that this field is part of the primary key.

Seeing the attributes in place provides a little insight into how Silverlight keeps track of various properties. For example, you now know how the client knows that a certain field is the primary key for the entity.

Although the `Entity` class provides extensibility points on the client, it's rare for an application to use them for validation or anything remotely like a business function. Extensions provided on the client can't be used back at the server and so can become a disconnect between the two models. To keep the two in sync, the RIA Services team provided a server-side model for extending the entity: metadata.

D.5.4 Using validation and display metadata

When you first created the domain service on the server, the wizard offered an option to generate the associated metadata class. This metadata class is a partial class that exists on the server and relates to a single entity. If you open the `EmployeeService.metadata.cs` file in the server project, you'll see both the `Contact` and `Employee` partial classes.

These partial classes include nested classes with the same public properties that are also defined in the entity classes. Those are just placeholders, providing a location where you can define metadata to control the display and validation of the fields.

But wait—why am I covering metadata in this section? Because this metadata is useful only if the client understands it. Silverlight and parts of ASP.NET are currently the

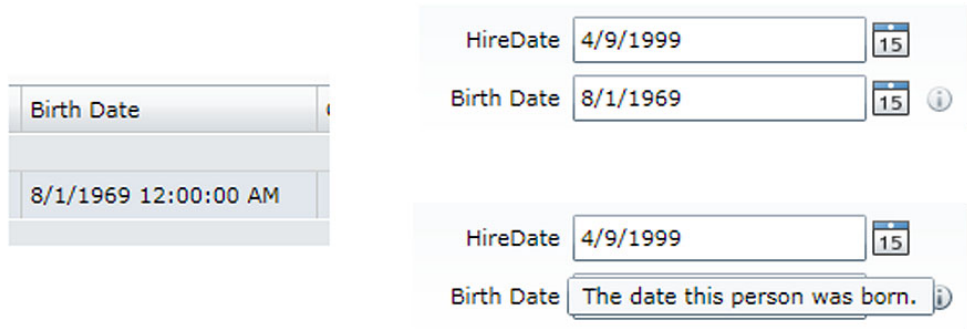


Figure D.12 The `Display` annotation in use on the `DataGrid` on the left and the `DataForm` on the right. At the lower right is the `Description` property in a tooltip.

only clients that can make sense of attribute-based annotation metadata for validation and display.

CONTROLLING DISPLAY

The `DataForm` labels and the `DataGrid` column headers have that ugly PascalCase text formatting. It'd be nicer to introduce actual spaces to make the fields more human-readable. You may even want to provide some tooltip descriptive information for certain fields.

In the `EmployeeService.metadata.cs` file on the server, scroll down to the `Employee` partial class and the nested `EmployeeMetadata` class within it. Find the `BirthDate` field and add this attribute:

```
[Display(Name="Birth Date",
        Description="The date this person was born.")]
public DateTime BirthDate { get; set; }
```

That says to use the string “Birth Date” for column headers and field labels, and if a tooltip or other description approach is available, use this description. Figure D.12 shows how this looks at runtime.

As you learned in chapters 17 and 18, annotations can be used for more than display. One of the more powerful uses is for validation.

ADDING VALIDATION

You get data type validation and the inferred validation (string length, required, and so forth) from the database for free. But you'll typically want to add your own validation to make the UI more bulletproof.

In the `EmployeeService.metadata.cs` file, scroll down to the `Employee` partial class and the nested `EmployeeMetadata` class within it. Find the `Gender` field, and add this attribute:

```
[RegularExpression(" [MmFf]",
        ErrorMessage="Specify (M)ale or (F)emale, please")]
public string Gender { get; set; }
```

Run the application, and attempt to type something else into the `Gender` field. The regular expression restricts the valid input choices to M, m, F, and f. The metadata entered on the server was automatically carried over to the client. If you open the `RiaExample.Web.g.cs` file on the client and navigate to the generated `Gender` property, you'll see the addition of the new attribute:

```
[DataMember()]  
[RegularExpression("[MFmf]",  
                    ErrorMessage="Specify (M)ale or (F)emale, please")]  
[Required()]  
[StringLength(1)]  
public string Gender  
...
```

The `StringLength`, `Required`, and `DataMember` attributes were previously there as part of the inferred metadata coming from the data model.

Annotation for display and validation is a nice, easy way to add significant robustness to your classes. Because the information goes into metadata buddy classes, you don't have to worry about the autogeneration process stepping on them.

D.6 Summary

I hope I've given you a taste of what WCF RIA Services can help you accomplish. Despite the depth of this appendix, I've just scratched the surface and have even more to cover in the next appendix. RIA Services supports advanced patterns such as the Presentation Model, as well as non-Silverlight clients. It also supports transactions and concurrency schemes with conflict resolution, and it supports composed entities where master-detail relationships can be saved in one chunk. There are many more attributes that can be used, as well as variations on the domain services.

In the next appendix, you'll continue your exploration of WCF RIA Services with a focus on advanced things you can do with it, including exposing data to other clients, incorporating business logic, and decoupling your entity model from your presentation model.

appendix E: *WCF RIA endpoints, security, business logic, and decoupling*

In appendix D, you were introduced to WCF RIA Services and the business application template. You went all the way from an empty application to one supporting basic CRUD operations. In this appendix, I'll expand on that example to include several other optional but important topics.

A typical Silverlight application doesn't exist in a vacuum; it needs to share its data with other types of clients, perhaps on the web, perhaps on the desktop. Despite the ease of doing so, the very last thing developers want to do is lock their application in a hard silo from which no data escapes. One of the first things you'll look at in this appendix is how to prevent that by exposing other endpoints from your domain service.

Editing of data wouldn't be complete unless you also considered ways to integrate security authentication and authorization into the mix. You need to be able to control who can update data, who can delete data, and who can access data. All of these things are possible with RIA Services and ASP.NET.

With all these great end-to-end features, it can be easy to tightly couple your UI to entities defined by your Object Relational Mapping tool or library (ORM). Although that's not always a bad thing (simple table maintenance apps are fine, for example), it often is. You'll look at the presentation model support in RIA Services and learn how to decouple your entities to avoid this problem.

Finally, one of RIA Service's highly touted benefits is the ability to share business logic between the client and server without having to duplicate it. The appendix wraps up with a look at how to do that and explains how to handle general code sharing between the tiers and even between different clients.

By the end of this appendix, you'll understand a number of important ways to augment your RIA Services applications beyond simple CRUD operations. Let's start with how to break the application data out of the silo.

E.1 Exposing data to other clients through endpoints

Every client-exposed domain service is also a WCF service. The full address of the WCF service is the web server plus the full namespace, with all dots replaced by dashes, plus `svc`. For example, for `EmployeeService`, in the `RiaExample.Web.Services` namespace, the full URL is

```
http://localhost:<port>/RiaExample-Web-Services-EmployeeService.svc
```

If you start the project and then replace the URL with that (so the ASP.NET site is still running), you'll get the normal WCF service page. Unlike an ASMX SOAP service, you can't execute the service methods from this page (which is good for preventing curious end users from running services directly).

You can use the Add Service Reference menu option from any WCF-aware project type (WPF, Windows Forms, ASP.NET, or even console) and use the service directly. You won't get the rich metadata and client-side validation provided by a native RIA Services client, but you'll be able to access the data and queries, as well as any defined domain methods in the service.

In addition to the Add Service Reference approach, which should be your first option if supported in your client, several other possible endpoints are supported. Each of the endpoints—OData, JSON and SOAP—were picked because of their broad industry and application support, as well as the different types of interfaces they expose. OData is very data-oriented and semi-RESTful in nature; JSON is very structural, like XML, and has broad support for HTML clients; and SOAP is method-oriented and is the traditional business application choice. I'll cover each throughout the rest of this section.

RIA Services and HTML clients

Although WCF RIA Services was initially created as a Silverlight-only technology, interest in it has caused RIA to move to other client types as well. In addition to the OData support in this section, WCF RIA Services has full support for HTML clients using jQuery. Those clients can use the same types of rules, async operations, and more that Silverlight clients can, albeit with a slightly different JavaScript-based client API. By extension, WCF RIA Services should be able to easily support Windows 8 JavaScript Metro applications.

E.1.1 Exposing an OData endpoint

RIA Services can expose a read-only OData endpoint for use by any application that can speak the OData/AtomPub protocol. When creating the domain service, you were

offered the option to expose an OData endpoint. For the example in appendix D, you did that (or I hope you did!). Selecting that option did two things:

- Added a `system.serviceModel/domainServices/endpoints` name of `OData` to the `web.config` file in the ASP.NET project.
- Added `IsDefaultQuery` to the retrieve methods in the domain service class

On my machine, the `web.config` change made by the tooling looks like this:

```
<system.serviceModel>
  <domainServices>
    <endpoints>
      <add name="OData"
        type="System.ServiceModel.DomainServices.Hosting.ODataEndpointFactory,
        System.ServiceModel.DomainServices.Hosting.OData, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
    </endpoints>
  </domainServices>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
    multipleSiteBindingsEnabled="true" />
</system.serviceModel>
```

If you have an entry like that in your `web.config` you made the correct selections when creating the project. Because the name added is `OData`, the service name has `/OData` appended to it. In this case, the service name is

`http://localhost:<port>/RiaExample-Web-Services-EmployeeService.svc/OData`

If you want to see metadata about the service (the OData rough equivalent of SOAP WSDL), you can append `/metadata` to the endpoint name. For this service, it's as follows (with ... substituting for the local host and port):

`http://.../RiaExample-Web-Services-EmployeeService.svc/OData/metadata`

To access the root entities sets exposed by the domain service, you append `Set` to the name of the entity so `Employee` becomes `EmployeeSet`. Then, append that to the OData endpoint URL, as shown here:

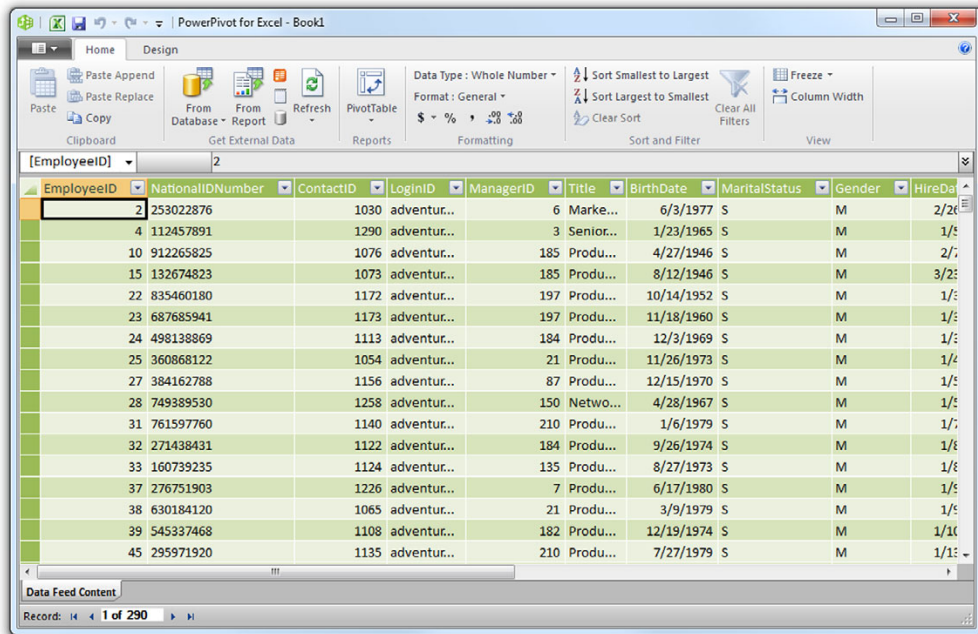
`http://.../RiaExample-Web-Services-EmployeeService.svc/OData/ContactSet`
`http://.../RiaExample-Web-Services-EmployeeService.svc/OData/EmployeeSet`

Currently, accessing a single entity by ID isn't supported in the OData endpoint. With a full OData endpoint, you'd be able to do something like this:

`http://.../RiaExample-Web-Services-EmployeeService.svc/OData/EmployeeSet(1)`
 (NOTE: this is not supported)

You can easily test the OData endpoint in Microsoft PowerPivot¹ for Excel 2010 by selecting the From Data Feeds option while the application is running, and providing the full `EmployeeSet` or `ContactSet` URL. When executed, the `EmployeeSet` query returns the results directly into PowerPivot, as seen in figure E.1.

¹ You can download Microsoft PowerPivot for Excel 2010 from <http://powerpivot.com>.



EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title	BirthDate	MaritalStatus	Gender	HireDate
2	253022876	1030	adventur...	6	Marke...	6/3/1977	S	M	2/20/2002
4	112457891	1290	adventur...	3	Senior...	1/23/1965	S	M	1/1/2002
10	912265825	1076	adventur...	185	Produ...	4/27/1946	S	M	2/1/2002
15	132674823	1073	adventur...	185	Produ...	8/12/1946	S	M	3/2/2002
22	835460180	1172	adventur...	197	Produ...	10/14/1952	S	M	1/1/2002
23	687685941	1173	adventur...	197	Produ...	11/18/1960	S	M	1/1/2002
24	498138869	1113	adventur...	184	Produ...	12/3/1969	S	M	1/1/2002
25	360868122	1054	adventur...	21	Produ...	11/26/1973	S	M	1/1/2002
27	384162788	1156	adventur...	87	Produ...	12/15/1970	S	M	1/1/2002
28	749389530	1258	adventur...	150	Netwo...	4/28/1967	S	M	1/1/2002
31	761597760	1140	adventur...	210	Produ...	1/6/1979	S	M	1/1/2002
32	271438431	1122	adventur...	184	Produ...	9/26/1974	S	M	1/1/2002
33	160739235	1124	adventur...	135	Produ...	8/27/1973	S	M	1/1/2002
37	276751903	1226	adventur...	7	Produ...	6/17/1980	S	M	1/1/2002
38	630184120	1065	adventur...	21	Produ...	3/9/1979	S	M	1/1/2002
39	545337468	1108	adventur...	182	Produ...	12/19/1974	S	M	1/1/2002
45	295971920	1135	adventur...	210	Produ...	7/27/1979	S	M	1/1/2002

Figure E.1 Data from the WCF RIA Services OData endpoint, loaded into PowerPivot for Excel 2010. PowerPivot is a C# .NET Office add-in application.

OData endpoints are good for querying data on the web or using tools such as PowerPivot. Although OData could be used for Ajax applications, you'll be better served using the native JSON endpoint.

E.1.2 Exposing a JSON endpoint

Both the JSON and SOAP endpoints require the use of assemblies in the RIA Services toolkit, which can be installed, like all other Silverlight tools, using the Microsoft Web Platform Installer.² If you performed a default Silverlight tools installation with RIA Services, you have the toolkit installed. If you don't have a toolkit folder under the Program Files\Microsoft SDKs\RIA Services 1.0\ folder, you can manually install the toolkit from <http://silverlight.net/getstarted/riaservices/>.

From the web project, you'll need to add an assembly reference to the `Microsoft.ServiceModel.DomainServices.Hosting` assembly in the RIA Services toolkit. The DLL is typically located in your RIA Services SDK folder; you can find it by browsing to it. Figure E.2 shows the Add Reference dialog with the correct assembly selected. Note that it starts with "Microsoft.ServiceModel," not "System.ServiceModel." If you're not careful, it's easy to pick the wrong assembly.

² You can download the Web Platform Installer from <http://bit.ly/WebPI>.

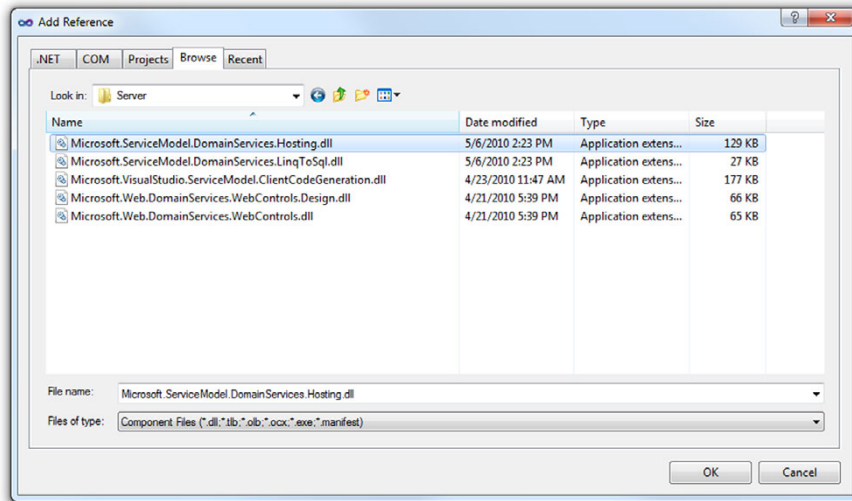


Figure E.2 The Add Reference dialog with the correct assembly selected to allow exposing JSON and SOAP endpoints

When the project reference is set, you'll need to modify the web.config file to add the new JSON endpoint. In the domainServices\endpoints section, where the OData endpoint also lives, add the XML from the next listing, taking particular note of the line continuations (those need to all be on single lines).

Listing E.1 XML Entry to add to the web.config

```
<add name="JSON"
      type="Microsoft.ServiceModel.DomainServices.Hosting.JsonEndpointFactory,
      ↪ Microsoft.ServiceModel.DomainServices.Hosting, Version=4.0.0.0,
      ↪ Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
```

That configuration entry sets up a WCF endpoint using the factory included in the RIA Services toolkit DLL. When it's configured, the root URL will be, as it was in the OData case, the service name with `<endpoint>`:

```
http://localhost:<port>/RiaExample-Web-Services-EmployeeService.svc/JSON
```

That's not the complete URL, just the root with the endpoint name. You can call the endpoint anything you want, as long as you use the same endpoint name in the configuration file and in the URL. By convention, you use the return type—JSON. To perform a query, use `Get<EntityName>s` as the format. For example:

```
http://.../RiaExample-Web-Services-EmployeeService.svc/Json/GetEmployees
```

Note that if you call that URL using Internet Explorer 8, you'll get a download error. If you use IE9 or later, you'll be able to open the results in a text editor. If you use Google Chrome, or another browser or JSON tool, you'll be able to see the text of the

JSON content in-browser. If you have nothing handy and don't have a current browser, create this simple HTML file (see the following listing) in the web project and select View in Browser. I called mine Test-JsonEndpoint.html and used a little jQuery to handle the Ajax call.

Listing E.2 Testing the JSON endpoint from JavaScript using jQuery

```
<html>
<head>
<title>Awesome JSON Endpoint Test</title>
<script src="http://ajax.microsoft.com/ajax/jquery/jquery-1.6.4.min.js"
    type="text/javascript">
</script>
</head>
<body>
<button type="button" onclick="query()">Query</button>
<div id="results">
</div>
<script type="text/javascript">
    function query() {
        $.ajax({
            type: "GET",
            url: "/RiaExample-Web-Services-EmployeeService.svc/JSON/GetEmployees",
            success: function (data) {
                $("#results").append("<ul>");
                var employees = data.GetEmployeesResult.RootResults;
                $.each(employees, function (i, entity) {
                    $("#results").append("<li>" + entity.EmployeeID +
                        " " + entity.Title + "</li>");
                });
                $("#results").append("</ul>");
                alert("Data received");
            }
        });
    }
</script>
</body>
</html>
```

← Note path

The example HTML page in the listing shows how to test the retrieve method of the JSON endpoint for your RIA Services domain service class. Using the `EmployeeID` and `Title`, it creates a single list item for each employee returned in the query and displays an alert when the query returns. Note the path used to get to the root of the results: it's the name of the query with `Result` appended, plus the name `.RootResults`. This is consistent for any RIA Services JSON `get` call.

jQuery makes the service call and processing simple. If you haven't yet been exposed to jQuery, definitely check it out. jQuery has been the one thing that makes JavaScript and DOM manipulation almost tolerable for me. It's a great library for handling on-page work, and it interacts nicely with Silverlight.

The JSON endpoint also supports updating data. For space and relevance reasons, I won't create a full update UI here, but the code is similar to any other JSON Ajax call using a POST.

JSON is great for Ajax applications, but the format itself can be limiting. Although not as rich as the WCF native formats, another widely understood format is SOAP.

E.1.3 Exposing a SOAP endpoint

Like JSON endpoints, SOAP endpoints are updatable services exposed using a service endpoint definition in the web.config file. The entry to add for SOAP is shown next.

Listing E.3 The web.config entry for a SOAP endpoint

```
<add name="Soap"
type="Microsoft.ServiceModel.DomainServices.Hosting.SoapXmlEndpointFactory,
    Microsoft.ServiceModel.DomainServices.Hosting, Version=4.0.0.0,
    Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
```

This requires the same assembly reference the JSON example required. Note that the public key token and other assembly information are identical as well.

Unlike the JSON approach, the SOAP endpoint ends up working right at the root service level. For example, to get the Web Services Description Language (WSDL) for the SOAP service in your solution, hit this URL with the browser:

```
http://localhost:<port>/RiaExample-Web-Services-EmployeeService.svc?wsdl
```

You don't need to add `/Soap` to the URL; requesting the WSDL implies that.

To fully utilize the SOAP client, you'll need to add a service reference from another project and generate the client. As was the case in the JSON version, the service is read/write but doesn't expose the entity metadata to the client.

WCF RIA Services has built-in support for Silverlight and JavaScript, but other clients need to be able to at least access the data. For those situations, RIA Services includes the ability to have OData, JSON, and SOAP endpoints. You don't get the complete RIA package with all of its time-saving goodies, but you get at least as much as you would if you'd used a vanilla web service and handled validation and other features manually.

One of those goodies you get with WCF RIA Services is support for authentication and authorization. You'll look at that next.

E.2 Authentication and authorization

Authentication is the process of identifying a user. *Authorization* is the process of granting the user access to parts of the system. Business applications almost always require some form of authentication and typically lock down critical functions using an authorization scheme. It's a rare system indeed where every user has complete access to every function. But until RIA Services came along to help with this, implementing security in Silverlight applications was a difficult process at best.

WCF RIA Services authentication is built on ASP.NET authentication and membership. I won't go into great detail on how to configure ASP.NET, but any tutorial on ASP.NET membership and authentication configuration will apply here.

The Silverlight Business Application template includes much of the authentication infrastructure built in. Normally, you'd have to add in the authentication domain service and the appropriate entity classes. Fortunately, those are all there, just waiting to be activated.

In this section, you'll explore the authentication and authorization capabilities of ASP.NET, surfaced through WCF RIA Services. You'll examine the UI and services that the template provides and that build on the RIA Services libraries. Throughout, you'll look at both Forms-based authentication and Windows authentication.

E.2.1 Authentication

Authenticating users usually involves getting their username and some sort of secret password (or PIN or biometric data), and comparing the pair against data stored in the database. Figure E.3 shows the built-in Login dialog that comes with the Silverlight Business Application template.

The dialog is wired up to the `AuthenticationService` on the website, which in turn uses ASP.NET membership. It also includes an appropriate validation display for incorrect username and password combinations. Figure E.4 shows this view.

There are two ways of validating this information in a RIA Services application: Forms-based authentication and Windows authentication.

FORMS-BASED AUTHENTICATION

Forms-based authentication (FBA) is cookie-based authentication in ASP.NET. Almost any ASP.NET website with an on-page login form is using a type of Forms-based authentication. Rather than relying on system tokens and security credentials provided by the OS, each site or application can store user information in a database. For the vast majority of applications running outside the firewall, this is the way security is handled.

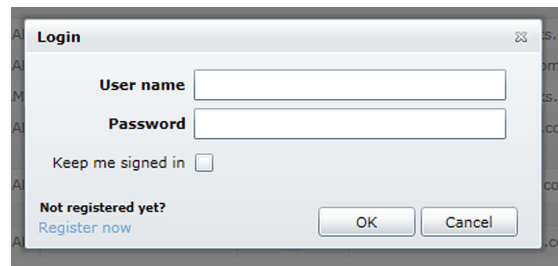


Figure E.3 The Login dialog in the Silverlight Business Application template. Note the registration link on the left.

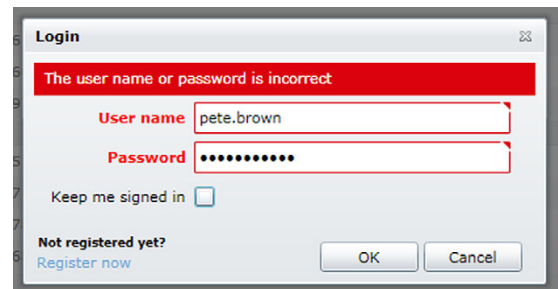


Figure E.4 The Login dialog when an incorrect password was entered

To configure the users and roles for an application using FBA, you'll use the ASP.NET application configuration site. This site writes to the aspnetdb database (or other database if so configured) where the membership data is stored. More often than not, this database is located in the App_Data folder on the ASP.NET site.

To configure this application, select the web project and choose the ASP.NET Configuration option from the Project menu. Figure E.5 shows the menu you'll see.

You can create new users through the administration site. If you don't have any defined, you'll want to create one now.

In addition, the Silverlight Business Application template includes a self-service registration UI (which you can disable if you desire) for allowing self-registration of users. This form, shown in figure E.6, is wired up through the *UserRegistrationService* on the server.

Configuring the site and application to use FBA is a two-step process. The first step is to open the web.config file and ensure that the authentication mode is set to Forms, as shown next.

Listing E.4 Web.Config Entry for forms authentication

```
<system.web>
...
<authentication mode="Forms">
  <forms name=".RiaExample_ASPXAUTH" timeout="2880" />
</authentication>
...
</system.web>
```

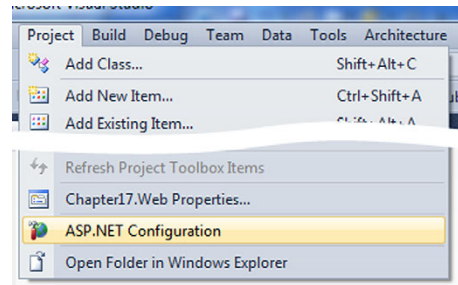


Figure E.5 The Project menu showing the ASP.NET Configuration option selected. This is the option used to configure the authentication database. If you don't see it, make sure the right project is selected.

Figure E.6 The Register dialog in the Silverlight Business Application template. For most business applications, you'll secure or eliminate this dialog.

The second step is to open App.xaml.cs in the Silverlight project and check the constructor to ensure the `Authentication` property of the web context is set to `FormsAuthentication`, as shown in the next listing.

Listing E.5 Required Forms Authentication setting in app.xaml.cs

```
public App()
{
    InitializeComponent();

    WebContext webContext = new WebContext();
    webContext.Authentication = new FormsAuthentication();
    this.ApplicationLifetimeObjects.Add(webContext);
}
```

Forms Authentication ←

With those two options set and a user created, you're ready to try out the application. Try logging in via the link on the main page. You'll see the UI change to indicate your login name rather than the login button, and the credential information itself will be available throughout the application.

NOTE If you're running this with an earlier release of RIA Services, such as the RIA Services SP2 RC, the business application template has a bug that disables everything once you log in. You might say that it's a super secure system, except it has the side effect of doing exactly the opposite of what you'd want. There's an easy one line of code fix for this, though, described in Jeff Handley's blog post: <http://bit.ly/BATDisabled>. In the downloadable source code for this section, I already applied the fix.

Although FBA is the most common form of authentication, we can't forget good old Windows authentication.

WINDOWS AUTHENTICATION

Windows authentication relies on the Windows OS and security infrastructure to provide the appropriate authentication scheme and tokens. For behind-the-firewall systems, Windows authentication is usually the better approach because there's no separate login process. Instead, the Silverlight application participates in single sign-on (SSO) along with other applications on the client.

To configure the application to use Windows authentication, first set the authentication mode in web.config:

```
<authentication mode="Windows" />
```

Then, in the App.xaml.cs file, modify the constructor to set the authentication to Windows:

```
webContext.Authentication = new WindowsAuthentication();
```

The business application template has startup logic that attempts to automatically resolve the credentials of the signed-in user. You'll find with `WindowsAuthentication`

that a second or two after the application launches, you're greeted with your credentials in the upper-right corner. No Login dialog required!

REQUIRING AUTHENTICATION

Regardless of which approach you use (forms or windows), you can require authentication from code or via attributes. On a domain service, it's easy to mark a single method as requiring a valid user account by applying the `RequiresAuthentication` attribute, as shown next.

Listing E.6 Modification to `InsertEmployee` to require authentication

```
[Insert]
[RequiresAuthentication]
public void InsertEmployee(EmployeePresentationModel employeePM)
{
    ...
}
```

The code in the domain service in the asp.net project ensures that only authenticated users are able to use the `InsertEmployee` service method. The `Insert` attribute is optional of course, because you properly named this method. Technically, the use of the `RequiresAuthentication` attribute falls under authorization because you're granting access based on security. But the authorization system is even more powerful than this.

E.2.2 Authorization

When you authorize users, you're granting them permission to perform an action. Authorization comes in many forms: client-side code can check to see whether users are authenticated, as well as whether they're members of a specific role. Server-side code or attributes can grant access to individual service methods.

The usual approach when working with authorization in ASP.NET and in RIA Services is to use role-based authorization. This is especially useful with Forms-based authentication, because the roles can be configured using the same ASP.NET administration application.

ROLE-BASED AUTHORIZATION

Although you could enable access to individual features on a user-by-user basis, role-based authorization is by far the most common way to grant access. In this model, users belong to roles, such as Manager, Administrator, or HR, and individual permissions are granted to the roles.

To enable roles in the RIA Services application, ensure that the `roleManager` entry in `web.config` is set to true, as shown in the following listing.

Listing E.7 The required `roleManager` entry in `web.config`

```
<roleManager enabled="true">
  <providers>
    <clear />
```

← Make sure this is "true"


```

<add name="AspNetSqlRoleProvider"
      type="System.Web.Security.SqlRoleProvider"
      connectionStringName="ApplicationServices" applicationName="/" />
<add name="AspNetWindowsTokenRoleProvider"
      type="System.Web.Security.WindowsTokenRoleProvider"
      applicationName="/" />
</providers>
</roleManager>

```

The actual contents of the setting will be specific to the implementation. The main thing to confirm is that there's a `roleManager` entry and that it's enabled. When that setting is confirmed and you've created some users and added them to appropriate roles using the ASP.NET configuration application, you can start to modify the application to look for those roles. The easiest and most powerful check you can make is on the service methods on the domain service. This is done via the `RequiresRole` attribute, as shown next.

Listing E.8 Securing the service by requiring authentication and the Manager role

```

[Query(IsDefault = true)]
[RequiresAuthentication]
public IQueryable<Employee> GetEmployees() { ... }

[Insert]
[RequiresRole("Manager")]
public void InsertEmployee(Employee employee) { ... }

[Update]
[RequiresRole("Manager")]
public void UpdateEmployee(Employee currentEmployee) { ... }

[Delete]
[RequiresRole("Manager")]
public void DeleteEmployee(Employee employee) { ... }

```

The listing shows how to secure the three data modification functions and restrict access to the `GetEmployees` method to those who are authenticated.

The `RequiresRole` attribute takes in one or more role names as strings. When the client attempts to access the service method, the server consults the security tokens provided and checks to see whether the user has the correct role. If the user isn't a member of that role, the service call results in an exception, which you must trap on the client.

You must handle this exception and gracefully inform the user that access isn't allowed. When using the `DomainDataSource` control, you do this in the `LoadedData` event, as shown next. The first line goes in the constructor for the page.

Listing E.9 Handling the Insufficient Permissions exception in Home.xaml.cs

```

this.DataSource.LoadedData += new
    EventHandler<LoadedDataEventArgs>(DataSource_LoadedData);
...

```

```
private void DataSource_LoadedData(object sender, LoadedDataEventArgs e)
{
    if (e.HasError)
    {
        if (e.Error is DomainOperationException &&
            e.Error.Message.Contains("denied"))
        {
            MessageBox.Show("Insufficient permissions for operation. Nyah!");
            e.MarkErrorAsHandled();
        }
    }
}
```

In the case of the `EmployeeContactService`, you require the Manager role for the `Insert`, `Update`, and `Delete` methods and `RequiresAuthentication` for the query method. For methods tagged with `RequiresRole`, you don't need to also add `RequiresAuthentication`; it's assumed in the role check.

Run the application and try to use the functions. You'll need to log in before you can do anything. Once you log in, click on the About page and navigate back to the main page to load the data by calling the query method with the new credentials.³

The second way to check for authorization is to use the client-side `WebContext` object. This is useful to enable/disable menu options and buttons, as well as to perform client-side checks. Don't rely on this as your only security check, though, because you always want the server to be secured. The following listing shows how to check on the client.

Listing E.10 Checking for permissions on the client side

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (WebContext.Current.User.IsInRole("Manager"))
    {
        SubmitChanges.Visibility = Visibility.Visible;
    }
    else
    {
        SubmitChanges.Visibility = Visibility.Collapsed;
    }
}
```

← Role

I put this in the `OnNavigatedTo` handler in `Home.xaml.cs`, but that's not the best place. Instead, you'll want to reevaluate any UI changes like this whenever the user logs in or logs out.

Run the application with this change and you'll notice that the Submit button is no longer visible (unless you left yourself logged in). Log in with a user in the Manager role, then click the About page and back to home to reload, and you'll see both the data and the button.

WCF RIA Services makes it easy to integrate authentication and authorization into your own application. Because it builds on ASP.NET membership and security, you

³ You're not going to actually make me apologize for that horrible UI hack, are you? After all, it's coding like that that makes you have to restart applications to apply config changes, reboot to reload drivers, etc. I'm in good company. Well, in some sort of company anyway.

know it's using a well-known and time-tested approach, which is already supported by the community. The ability to support Windows authentication as well as Forms-based authentication covers the vast majority of authentication schemes for applications in the wild. It won't cover them all out of the box (certificate-based authentication with smart cards comes to mind), but it wasn't meant to. If you can get ASP.NET to recognize the security model, though, there's a really good chance you can get WCF RIA Services to use it.

So far in this appendix, we've looked at functional features of WCF RIA Services. In the remainder of the appendix, we'll look at features of RIA Services that help with architectural decisions, such as decoupling and using the MVVM pattern. First, let's look at WCF RIA Services support for presentation models.

E.3 *Loose coupling: using presentation models*

So far, you've created a tight coupling between your database and the UI due to bringing the data structure through from back to front. RIA Services enables you to create entities that combine data from multiple entities in the data access layer—for example, combining the `Contact` and `Employee` classes into a single logical entity.

When using a presentation model, you can respond to changes in the database or database model by changing only how the presentation model aggregates that data. Also, you can simplify the client code by designing a model that aggregates only those fields that are relevant to users of the client.

Although conceptually similar, the presentation model here shouldn't be confused with the Presentation Model pattern. The pattern shares some similar goals and approaches, but the RIA Services approach is more server-centric.

I consider the presentation model to be one of the most important additions to WCF RIA Services in terms of making it work with best practices and patterns such as MVVM. As great as RIA Services is without it, it always bothered me that the data model was logically coupling the UI to the services to the data access layer to the database. Change one, and they all have to change—not a good situation to be in.

Ideally, you'd have a good object-persistence mapper that would flatten objects and relationships and handle all this for you, along with the knowledge to use it. That alone would eliminate most uses of the presentation model approach, including the example I'll include in this appendix. In many cases, developers don't have this available to them, or don't have the knowledge required to set up an existing one, or perhaps are further constrained by other business or environmental factors.

The presentation model approach is also good for combining data from multiple sources. You can create a single entity that's composed of fields from multiple databases.

In all of these cases, the presentation model approach can be a huge help.

In this section, you'll take the employee service and model you've been working with and convert (more correctly, rewrite) it to introduce a presentation model. I'll show you how to query data, update data, and insert data using this new model.

E.3.1 Creating the employee presentation model

You've been unable to perform insert and update operations on the `Employee` class so far because it's tied to the `Contact` class. This relationship is purely a database thing. It makes little or no sense from an end-user perspective; they're logically part of the same entity. This is a common scenario, because you tend to factor out things such as contact information, address information, and more in the database, and it always causes no end of annoyances at the UI level.

You have two goals in creating an employee presentation model:

- Expose the contact information as first-class fields of a logical employee entity
- Limit the other fields that are returned to the client

The first step in creating a presentation model is to create a class named `EmployeePresentationModel` on the asp.net server project. Create this class in the server-side Models folder, using the following code

Listing E.11 The `EmployeePresentationModel` class

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RiaExample.Web.Models
{
    public class EmployeePresentationModel
    {
        [Key]
        [Display(AutoGenerateField = false)]
        public int EmployeeID { get; set; }
        [Required]
        public string NationalIDNumber { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        [Required]
        public bool NameStyle { get; set; }
        [Display(AutoGenerateField = false)]
        public int ContactID { get; set; }
        [Display(Name = "Email Address")]
        public string EmailAddress { get; set; }
        [Required]
        public int EmailPromotion { get; set; }
        public string Phone { get; set; }
        [Required]
        public string Title { get; set; }
        [Display(Name = "Birth Date")]
        public DateTime BirthDate { get; set; }
        [Required]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }
        [Required]
        public string LoginID { get; set; }
    }
}
```

← Don't show key
by default

```

[Required]
public string MaritalStatus { get; set; }
[Required]
[StringLength(1)]
[RegularExpression("[MFmf]",
    ErrorMessage = "Specify (M)ale or (F)emale, please")]
public string Gender { get; set; }
[Required]
public bool SalariedFlag { get; set; }
[Required]
public int VacationHours { get; set; }
[Required]
public int SickLeaveHours { get; set; }
[Required]
public bool CurrentFlag { get; set; }
}
}

```

← Length, Required, and Regex

In the listing you create an aggregate `Employee` class that includes fields from both the `Employee` and `Contact` classes you've been using so far. Also, because the metadata is no longer inferred from the database or read using the metadata buddy class you created earlier, you add a minimum amount of metadata to ensure that required fields are marked as such and to make a few of the names easier to read. I've also stopped autogeneration of UI columns/fields for the keys and added basic validation, a good example of which is the `Gender` property. Note that all of this is right in line with the class; when using presentation models, you don't need to separate metadata from entities.

In your own classes, you'll need to make sure that you account for required fields. If you can't infer their values when performing an insert or update operation, you'll need to include them in the class so users can input their values.

This new class now abstracts you from the database. If the structure of the database changes, you can change the query and update operations—the UI won't be affected (assuming it's a structural change, not a change in what defines an employee).

The next step is getting this information down to the client. To do that, you'll need to create at least one query operation and wire it through all the way to the `DomainDataSource` you've been using.

E.3.2 Supporting query operations

The presentation model approach requires a completely new domain service and new query and update operations. The new domain service class will no longer be directly based on the `LinqToEntitiesDomainService` base class but will instead be based directly on the `DomainService` base class.

For lack of a better name, I called the domain service `EmployeeContactService`, because it aggregates both the `Employee` and `Contact` entities. Create a new class file with this name and place it in the Services folder on the server project. Next is the code for this service.

Listing E.12 The EmployeeContactService in the Services folder

```

using System;
using System.Linq;
using System.ServiceModel.DomainServices.Hosting;
using System.ServiceModel.DomainServices.Server;
using RiaExample.Web.Models;

namespace RiaExample.Web.Services
{
    [EnableClientAccess]
    public class EmployeeContactService : DomainService
    {
        private AdventureWorks_DataEntities _context =
            new AdventureWorks_DataEntities();

        [RequiresAuthentication]
        public IQueryable<EmployeePresentationModel> GetEmployees()
        {
            return from e in _context.Employees
                   orderby e.Title, e.HireDate
                   select new EmployeePresentationModel()
                   {
                       BirthDate = e.BirthDate,
                       ContactID = e.ContactID,
                       CurrentFlag = e.CurrentFlag,
                       EmailAddress = e.Contact.EmailAddress,
                       EmailPromotion = e.Contact.EmailPromotion,
                       EmployeeID = e.EmployeeID,
                       FirstName = e.Contact.FirstName,
                       LastName = e.Contact.LastName,
                       NameStyle = e.Contact.NameStyle,
                       NationalIDNumber = e.NationalIDNumber,
                       Phone = e.Contact.Phone,
                       SalariedFlag = e.SalariedFlag,
                       SickLeaveHours = (int)e.SickLeaveHours,
                       Title = e.Title,
                       HireDate = e.HireDate,
                       Gender = e.Gender,
                       VacationHours = (int)e.VacationHours
                   };
        }
    }
}

```

← Context

← Build presentation model

The main code in the function performs a standard mapping of properties from two entities to one other. Note that even with the custom methods, you're still able to return `IQueryable` and to allow composition on the client.

WIRING UP TO THE UI

Because you have the same query name as you used in the `EmployeeService` domain service, to use the new service from the UI, you need to make only one change:

change the `DomainContext` property of the `DomainDataSource` in `Home.xaml` to point to the `EmployeeContactContext` as shown next.

Listing E.13 Updated DomainDataSource in Home.xaml

```
<riaControls:DomainDataSource x:Name="DataSource"
    PageSize="15" LoadSize="30"
    AutoLoad="True" FilterOperator="And"
    QueryName="GetEmployees"
    LoadedData="DataSource_LoadedData">
  <riaControls:DomainDataSource.DomainContext>
    <domain:EmployeeContactContext />
  </riaControls:DomainDataSource.DomainContext>
  ...
</riaControls:DomainDataSource>
```

← New service's
domain context

Be sure to build before making this change; otherwise, the `EmployeeContactContext` class won't exist on the client. Note that you didn't have to update any service references or add a new service reference—the WCF RIA Services tooling took care of that for you. That alone is worth the price of admission.

When you run the application, you'll see something like figure E.7. The new UI has fewer fields and looks a lot better than what you had before.

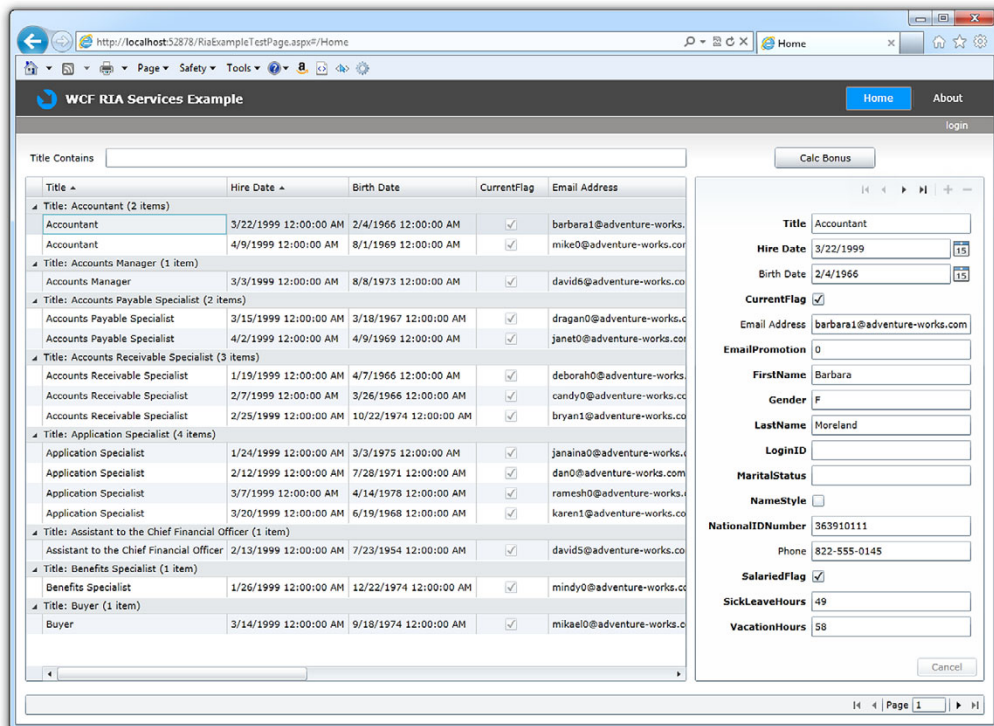


Figure E.7 The UI using the new `EmployeePresentationModel` class. Note how you have fields from the contact object now available to the UI.

You have a lot fewer fields in the UI now. Some, like Birth Date, which have had the `Display` attribute applied, have better labels and column headers. You could set the display name for the remaining ones, and the display order, as well using the same attribute. For space reasons, I didn't include the attributes in the listings here.

The presentation model approach certainly works in this situation. It's not meant just for flattening objects, although you can use it for that. It also shines in situations where you need to do joins in LINQ and combine the results into a single logical object.

Retrieval is fine for a demo, but the real test comes when you need to use this information in an update operation.

E.3.3 Supporting update operations

To perform an update using the presentation model approach, you'll need to map from the presentation model class back to the entities used in the backing store. Essentially, you're doing the reverse of what you did in the query operation.

This listing shows how to map from the presentation model back to the database entities. Place this code in the `EmployeeContactService` domain service in the ASP.NET project.

Listing E.14 The `UpdateEmployee` method and support functions

```
private void MapEmployee(Employee emp,
    EmployeePresentationModel employeePM)
{
    emp.BirthDate = employeePM.BirthDate;
    emp.CurrentFlag = employeePM.CurrentFlag;
    emp.Contact.EmailAddress = employeePM.EmailAddress;
    emp.Contact.EmailPromotion = employeePM.EmailPromotion;
    emp.Contact.FirstName = employeePM.FirstName;
    emp.Contact.LastName = employeePM.LastName;
    emp.Contact.NameStyle = employeePM.NameStyle;
    emp.Contact.Phone = employeePM.Phone;
    emp.NationalIDNumber = employeePM.NationalIDNumber;
    emp.SalariedFlag = employeePM.SalariedFlag;
    emp.SickLeaveHours = (short)employeePM.SickLeaveHours;
    emp.Title = employeePM.Title;
    emp.HireDate = employeePM.HireDate;
    emp.Gender = employeePM.Gender;
    emp.VacationHours = (short)employeePM.VacationHours;
    emp.MaritalStatus = employeePM.MaritalStatus;
    emp.LoginID = employeePM.LoginID;
}

[Update]
[RequiresRole("Manager")]
public void UpdateEmployee(EmployeePresentationModel employeePM)
{
    Employee emp = _context.Employees
        .Where(e => e.EmployeeID == employeePM.EmployeeID)
```

← Get current
persisted entity


```

        .FirstOrDefault();

    MapEmployee(emp, employeePM);
    EmployeePresentationModel original =
        this.ChangeSet
            .GetOriginal<EmployeePresentationModel>(employeePM);

    if (original.CurrentFlag != employeePM.CurrentFlag ||
        original.EmailAddress != employeePM.EmailAddress ||
        original.EmailPromotion != employeePM.EmailPromotion ||
        original.FirstName != employeePM.FirstName ||
        original.LastName != employeePM.LastName ||
        original.NameStyle != employeePM.NameStyle ||
        original.Phone != employeePM.Phone)
    {
        emp.Contact.ModifiedDate = DateTime.Now;
    }
    _context.SaveChanges();
}

```

← Check for changes

← Save to database

You're definitely in manual-plumbing land at this point. Of course, if you want to have separation between two layers, you'll have some mapping. Here, the mapping is in a reusable function so the `Insert` method can use it. Note how you check the original employee object to see if there were any changes before setting the modified date for the `Contact` object. You'll want to do the same for the `Employee` object; I left that out for space considerations.

The code you write in this function will be dependent on your choice of data access layer. The code in listing E.14 works well with the Entity Framework objects.

The next type of operation you'll need to support is the insertion of new objects. This one can get tricky due to the creation of dependent entities and the generation of keys.

E.3.4 Supporting insert operations

Update operations are easy, because you often don't have to worry much about entity relationships or foreign keys. `Insert` operations usually have a few extra steps to perform in addition to the mapping.

The next listing shows the `InsertEmployee` function. This function makes use of the `MapEmployee` function from the previous listing.

Listing E.15 The `InsertEmployee` function

```

[Insert]
[RequiresRole("Manager")]
public void InsertEmployee(EmployeePresentationModel employeePM)
{
    Contact contact = _context.Contacts.CreateObject();
    Employee emp = _context.Employees.CreateObject();
    emp.Contact = contact;
    MapEmployee(emp, employeePM);
    contact.ModifiedDate = DateTime.Now;
}

```

```

contact.rowguid = Guid.NewGuid();
contact.PasswordHash = "Adventure";
contact.PasswordSalt = "xyzzzy";
emp.ModifiedDate = DateTime.Now;
emp.rowguid = Guid.NewGuid();

_context.Contacts.AddObject(contact);
_context.Employees.AddObject(emp);
_context.SaveChanges();
}

```

Add new and save

The function in the listing creates the `Contact` and `Employee` data entities and sets the contact to be the contact for the `Employee`. It then calls the `MapEmployee` function from the previous listing to map the presentation model properties to the data entity properties. The next step is to set a few fields; the password-related fields here are dummies, but the modified date fields are correct. The last step before saving changes is to add the `Contact` and `Employee` to the entity sets. Finally, with a call to `SaveChanges`, the information all goes in the database.

I've included the query, update, and insert methods. I left out delete as an exercise for the reader. This is a simple function to build following the pattern established by the code included here.

The `Invoke` method `CalculateVacationBonus` transfers over nicely. Simply copy that from the `EmployeeService` to the `EmployeeContactService` and you'll have completed the methods required to make the UI function.

The presentation model approach allows you to continue to benefit from WCF RIA Services while also benefitting from the increased decoupling of the layers. Although the database-through-UI coupling won't be a problem for many applications, for anything expected to survive into a maintenance mode, it can be a real pain.

The presentation model approach isn't without its issues. You have to write more CRUD operation code, including mapping. This code has a habit of getting out of sync; it's also a great place to find typos and copy-paste errors. When using this approach, I highly recommend building tests around your mapping functions and keeping them up to date.

So far, you've seen normal CRUD operations and simple validation. I threw in one business function for calculating a vacation bonus, but otherwise you haven't seen any real business logic. The next section covers how to include this critical code in a RIA Services application.

E.4 Business logic

A business application without business logic is just a forms-over-data maintenance application. Although apps like that are easy to build using WCF RIA Services, they're not the usual case.

Business logic usually consists of discrete functions that implement discrete rules. Some may come in the form of validation, others may look like calculated fields, and still others may be helper methods that return a current piece of data from an external system.

There are several places where you can put logic in your code. I've tried to capture some general guidelines in table E.1.

Table E.1 Where to put your business logicAppE.fm

Type	Location
Data validation	Attributes on metadata or entities.
Field validation rule	Noncritical: custom validators. Critical: code in domain methods on the domain service. Prevent persistence if criteria aren't met.
External data access	Domain methods on the server calling out to web services. Services classes on the client, if the result won't be required for server-side validation. Shared code services proxy or shared binary.
Calculated field	If self-contained within the entity, as an additional property of the entity. If requires integration with other data or services, as a method on the domain service or shared code or a binary file.
General calculation or business logic	As a method on the domain service if a server round-trip is okay or required. As a method in shared code or a binary file if needed on the client and server with local calculation for speed.
On insert/update logic	In the Insert/Update method in the domain service.
Reusable logic shared between projects	Domain service. Shared code or binary file.
Anything else	Shared code or binary file.

You've already seen how to write methods on the domain service. In this section, you'll learn how to place logic in entities as well as how to share logic or code between the client and server.

NOTE For this section, you'll go back to your generated `EmployeeService` classes. That means you'll need to change the `DomainDataSource.DomainContext` in the `Home.xaml` markup to use the correct `EmployeeContext` rather than the `EmployeeContactContext`. It's a simple change; you may leave the server-side classes alone.

E.4.1 Business logic in entities

When a calculated field is part of the business logic for your application, one place you can place it is directly in the entity. This makes sense if the data required for the calculation exists entirely in the entity itself. If the data is external, consider making the calculation a service that you call to get the results.

A reasonable type of calculation might be, for example, one to take into account your start date and how many vacation hours you have when deciding if you can go in

the hole to take a longer vacation than you would've been allowed to take if going strictly by the book. Do too much of it, and you end up as an indentured servant. Do it too soon in your career, and you're just asking for trouble.

Going back to the original generated classes (not the presentation model classes), add the function in the following listing to the `Employee` class using a new file named `Employee.shared.cs` stored in the `\Models\Shared` folder on the web project.

Listing E.16 An example business method on the `Entity` class

```
using System;
namespace RiaExample.Web
{
    public partial class Employee
    {
        public int AllowedOverdraftVacationHours
        {
            get
            {
                DateTime today = DateTime.Today;
                int yearsInService = today.Year - HireDate.Year;
                if (HireDate.AddYears(yearsInService) > today)
                    yearsInService--;
                if (yearsInService < 1)
                    return 0;
                else if (yearsInService < 5)
                    return 20;
                else
                    return 40;
            }
        }
    }
}
```

The code performs a simple calculation and becomes part of the `Employee` class due to the use of the `partial` keyword. You may then call that method either on the client or the server, as it'll be available in both places. The key thing to note is that it's using information already available as part of the `Employee` class. I don't recommend this approach if you need to access data outside of the `Employee` class; there are other options, as you'll see next.

E.4.2 Sharing code

So far, you've put all the business logic into methods of the domain service or used it as a property of the entity. The domain service is a great place to put logic you want accessible to the client or server but executed only on the server. For methods that match, including them on the entity class is a great idea. Sharing code and controlling where it executes is important.

SHARED SOURCE FILES

In the previous example, you saw how the code went into a file with the `.shared.cs` extension. That naming is a convention understood by RIA Services. Anything with a

.shared.cs name is copied to the client on build as part of the code-generation process. As long as you keep the namespaces clean, this provides an easy way to share classes between the tiers.

LINKED SOURCE FILES

Visual Studio has long had the capability to link source files from one project to another. As long as the contained source code (including namespace-using statements) is compatible across both projects, it'll work fine.

This is source-level sharing and isn't limited to RIA Services applications. I've used it with WCF applications and also when dual-targeting Silverlight and WPF. Consider one project the master and add the file to it. Then, choose Add Existing Item in the other project, navigate to the source, and click the Add drop-down button so you can add a link. As my favorite black-helmeted villain would say, "All too easy."

SHARED BINARIES

Silverlight 4 along with .NET 4 introduced another option for sharing: .NET 4 applications can add references to a Silverlight class library, as long as that class library uses only certain namespaces. But this approach also works outside of RIA Services.

I've never been a big fan of this approach, because it feels a little dirty to me. But for this type of use, it should be perfectly acceptable. Keep in mind that you can only reference certain libraries when using these types of projects.

PORTABLE CLASS LIBRARIES

Visual Studio 2010 SP1 expanded on shared binaries with the official introduction of the Portable Class Library (PCL) project. This type of project can be shared across .NET 4+, Silverlight 5+, Windows Phone 7, and Xbox 360 and Windows 8 Metro, based on the profiles you select. As with shared binaries, you must keep your references down to a minimum. For more information on how this works and what's supported, see this MSDN article: <http://bit.ly/NetPCL>.

I've slowly been converting to this approach, because it has great platform support and there's no need to duplicate projects. It's not quite as flexible as the linked source approach, but it's certainly easier on developers and less error prone.

Regardless of which approach you use, sharing your code between client and server, or even between different types of clients, is an excellent idea. The fewer copies of a function you have, the less likely they are to get out of sync.

E.5 *Summary*

WCF RIA Services was developed for Silverlight, but the architects understood well the need to keep the data from being locked up behind some sort of proprietary service. For that reason, they made sure that domain services could be exposed in a multitude of formats consumable by other clients. OData, JSON, and SOAP, covered in this appendix, are the most popular formats on the web, and all are supported by WCF RIA Services.

In this and the previous appendix, you made use of the `DomainDataSource` control. One unfortunate side effect of all that was a tight coupling of the entities from

the database all the way through to the UI. Although it requires extra effort, WCF RIA Services has an answer for this coupling in the presentation model approach. In that, you have to create entities and the domain service from scratch, but once done, everything else “just works.”

What about business logic? You can put your business logic inside invoke methods on a domain service, as methods added to the partial class for the entity, and as shared code that can be downloaded to the client as part of the build process. Not to mention that you can use the standard Silverlight-supported approaches of shared source or shared binaries.

I’ve yet to see a serious business application that didn’t include authentication and authorization of some sort. No one wants to leave data-oriented applications open for anyone to mess around with. Silverlight and WCF RIA Services can take advantage of the security models in ASP.NET, building on proven technologies and knowledge you may already possess.

All this combines to be an intense and robust platform for building business applications—and it’s only at version 1!

appendix F: *WCF RIA Services* *and MVVM*

So far, you've relied heavily on the UI components, specifically the `DomainDataSource`, to integrate with WCF RIA Services. Without a doubt, that's the most efficient way to get started, and it's often enough for smaller or less complex applications.

That said, I spent a whole appendix espousing the benefits of MVVM, and I'm not about to do a complete about-face just because we're using RIA Services. In previous versions of RIA Services, such as the one I covered in the previous edition, using MVVM meant losing many of the features that made the `DomainDataSource` so useful. Specifically, you had to reimplement sorting, paging, grouping, filtering and more. Any one of those features could be challenging if not downright onerous to implement yourself. Combined, it's more work than most people would consider reasonable.

Luckily, in the latest version of the product, the RIA Services team included several key components to make building MVVM applications possible, without losing many of the benefits provided by the `DomainDataSource` control. The new `DomainCollectionView` and the classes it uses are the heart of a RIA Services MVVM implementation.

This appendix will cover how to integrate MVVM into your RIA Services application. I won't go nuts with MVVM but will instead focus on the core concepts that have the most impact (and the most potential difficulties) when working with RIA Services. You'll start by creating a view model, which you'll move your integration code to. Then, I'll introduce the `DomainCollectionView` class, which is what makes MVVM possible with RIA Services. Then, you'll learn how to use the `DomainCollectionView` and its related classes for filtering, grouping, sorting, and paging, but not without first dealing with that pesky login refresh problem. The appendix will wrap up with

using the domain context to call the `invoke` operation on the server to calculate the vacation bonus, as well as submitting changes to the database.

The examples and information here will build on what you learned in appendixes C, D, and E, as well as chapter 33 on MVVM, so be sure to have read that material before proceeding.

F.1 Moving to a view model

The first step in updating the example application is to provide a view model class for your view to interact with. In the Silverlight client project, create a new folder named `ViewModels`. Inside it, place a class named `EmployeeViewModel.cs`.

You're going to use the Command pattern rather than event handlers for your button actions. As discussed in chapter 33, this helps decrease coupling and makes the view model more easily testable.

In support of this, bring in the `ViewModelCommand` class from chapter 33 (section 33.4.1) and place it in the `ViewModels` folder in the client project. That folder should then have two files: the view model itself and the command class.

The view model will need a number of public properties and methods to support the `Home.xaml` view. I've described them in table F.1.

Table F.1 `EmployeeViewModel` members

Public member	Description
Constructor	Initialize the various types, handle the security check, and if all is well, load the initial set of data.
<code>CanLoad</code>	For our example, this always returns true if the user is authenticated. You may want to have additional criteria in here, maybe even a role check. Keep it in sync with the server-side role requirements, however.
<code>Employees</code>	An <code>ICollectionView</code> property that exposes the data collection, as well as filtering, sorting, and other useful functions. The main source for binding to our list of employees.
<code>SelectedEmployee</code>	The employee currently selected onscreen. This was previously handled entirely in XAML using <code>UIElement</code> binding between the <code>DataGrid</code> and <code>DataForm</code> .
<code>SubmitChangesCommand</code> , <code>CanSubmitChanges</code> , <code>SubmitChanges</code>	Implements the command pattern for submitting changes (saving) to the database.
<code>AddVacationBonusCommand</code> , <code>CanAddVacationBonus</code> , <code>AddVacationBonus</code>	Implements the command pattern for calculating the bonus and applying it to the selected employee.
<code>FilterCommand</code> , <code>CanFilter</code> , <code>ApplyFilter</code>	Implements the command pattern for filtering the data.
<code>FilterString</code>	Bound to the filter <code>TextBox</code> on the page, this property holds the text to filter by.

I'll cover most of those members in later parts of this section when I discuss the specific functionality they implement. The initial view model is much simpler, with only the code shown here.

Listing F.1 The initial `EmployeeViewModel` class

```
using System;
using System.ComponentModel;
using System.Linq;
using System.ServiceModel.DomainServices.Client;
using System.ServiceModel.DomainServices.Client.ApplicationServices;
using System.Windows;
using System.Windows.Data;
using Microsoft.Windows.Data.DomainServices;
using RiaExample.Web;
using RiaExample.Web.Services;

namespace RiaExample.ViewModels
{
    public class EmployeeViewModel : INotifyPropertyChanged
    {
        private EmployeeContext _context =
            new EmployeeContext();

        public EmployeeViewModel()
        {
        }

        private bool _canLoad = false;
        public bool CanLoad
        {
            get { return _canLoad; }
            set { _canLoad = value; NotifyPropertyChanged("CanLoad"); }
        }

        private Employee _selectedEmployee;
        public Employee SelectedEmployee
        {
            get { return _selectedEmployee; }
            set
            {
                _selectedEmployee = value;
                NotifyPropertyChanged("SelectedEmployee");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        protected void NotifyPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
                PropertyChanged(this,
                    new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

← RIA Services domain context

Usual INPC stuff

By the end of this appendix, the view model will be considerably larger. Let's tackle it piece by piece, starting with the most important thing: the `DomainCollectionView`.

Alternately, rather than implement `INotifyPropertyChanged` and include the event-raising code here, you can use the base `ViewModel` class from the MVVM chapter.

F.2 Introducing the `DomainCollectionView`

The `DomainCollectionView` is a concrete implementation of the `ICollectionView` interface. This is a powerful interface that supports grouping, sorting, filtering, as well as collection changed notification and enumeration via `IEnumerable`. It also supports `MoveFirst/Last/Position` type functionality to make paging easier. Typically, it's a wrapper or view over collection data, but as an interface, the specific implementation is left to the concrete class.

The `DomainCollectionView` was released as part of the WCF RIA Services SDK. The SDK is a collection of libraries provided out-of-band by the RIA Services team—that is, they can be updated separately from the main Silverlight and RIA Services installers, and therefore are also installed separately.

There are three main participants in the process of getting data to the client in RIA Services: the collection view, its loader, and the source to be populated with data. Figure F.1¹ shows how these parts fit in to the overall solution.

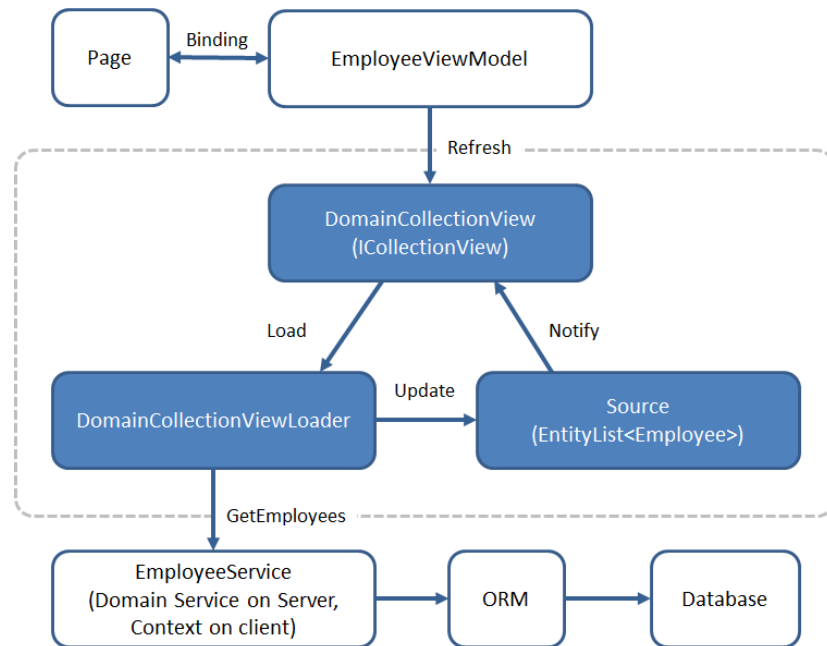


Figure F.1 The `DomainCollectionView`, `DomainCollectionViewLoader`, and `EntityList` in the context of the solution

¹ Huge thanks to Kyle McClellan on the WCF team for creating the initial diagram that heavily inspired this one. If you're interested in RIA Services, his is definitely a blog to subscribe to: <http://blogs.msdn.com/b/kylemc>.

The `DomainCollectionView` implements a number of well-known interfaces that allow it to act as a collection, participate in binding notification, and support paging, enumeration, and editing. It's the primary class with which the binding system will interface.

The `DomainCollectionViewLoader` is the class responsible for loading data into the `DomainCollectionView`. Typically you're going to call methods on the context, but this could also simply load in hard-coded mock data for testing purposes. Decoupling the loader from the collection view provides this flexibility and makes testing easier.

The `Source` is the collection data itself. It only needs be an `IEnumerable`, but in most RIA Services cases, it will be an `EntityList<T>` associated with the domain context.

The `DomainCollectionView`, along with the loader and its source, provide in code approximately the same functionality you had in XAML with the `DomainDataSource` but in an MVVM-friendly package. As you'll see throughout the rest of this section, only the client code needs to change; the server code can remain the same.

F.2.1 Updating the view model with the `DomainCollectionView`

In the Silverlight client project, add a reference to the RIA Services Toolkit assembly `Microsoft.Windows.Data.DomainServices`. If you don't have the toolkit installed (it will show up in your Add/Remove Programs and will be dated September 2011 or later), you may install it from here: <http://bit.ly/RiaServices>. (Remember, bit.ly links are case-sensitive; the lowercase version of that link, created by someone else, goes to a different location.)

Once installed, update the `EmployeeViewModel` to add this code.

Listing F.2 `EmployeeViewModel` updates

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    private EmployeeContext _context = new EmployeeContext();

    private DomainCollectionView<Employee> _view;
    private DomainCollectionViewLoader<Employee> _loader;
    private EntityList<Employee> _source;

    public EmployeeViewModel()
    {
        _source = new EntityList<Employee>(_context.Employees);
        _loader = new DomainCollectionViewLoader<Employee>(
            LoadEmployees, OnEmployeesLoaded);

        _view = new DomainCollectionView<Employee>(_loader, _source);
    }

    ... selected employee ...

    private void RefreshData()
    {
        using (_view.DeferRefresh())
        {
        }
    }
}
```

View, loader,
and source

Load actions

Wait...what?

```

    }

    private bool _canLoad = false;
    public bool CanLoad
    {
        get { return _canLoad; }
        set { _canLoad = value; NotifyPropertyChanged("CanLoad"); }
    }

    private LoadOperation<Employee> LoadEmployees()
    {
        var query = _context.GetEmployeesQuery();

        return _context.Load<Employee>(query);
    }

    private void OnEmployeesLoaded(LoadOperation<Employee> op)
    {
        if (op.HasError)
        {
            op.MarkErrorAsHandled();
        }
        else if (!op.IsCanceled)
        {
            _source.Source = op.Entities;

            _view.SetTotalItemCount(op.TotalEntityCount);

            if (op.TotalEntityCount > 0)
            {
                SelectedEmployee = op.Entities.First<Employee>();
            }
        }
    }

    public ICollectionView Employees
    {
        get { return _view; }
    }
    ...
}

```

← **Load Employees**

← **Error here**

← **DomainCollectionView**

Security checks aside (you'll take care of that in a moment), the view model now has all the code required to load data from the domain data service. The actual integration with the service is through the `_context` member variable. That code lives in the `LoadEmployees` method, which builds a query and returns it to the caller (the loader in this case). The loader then executes the query and fires off the `OnEmployeesLoaded` callback inside which you can do error checks, update the selected employee, and generally do whatever you need to with the returned data.

But how does the `LoadEmployees` method ever get called? If you look back at the constructor, you can see that `DomainCollectionViewLoader` constructor takes in two parameters: the action to be used for loading, and the callback function to call when it's complete.

The `DomainCollectionViewLoader` gets called when you call `Refresh` or `Defer-Refresh` on the `DomainCollectionView`. The strange `using` statement on

`DeferRefresh` is there to help scope the changes you wish to apply. If you won't be otherwise modifying the view, you can use `Refresh` instead. If you have multiple changes you wish to be batched up before you refresh (for example, a sort and a group and a filter), enclose them within the `using` statement. When `Dispose` is called on the result, the query will execute. It's an ingenious if not entirely intuitive use of the `Dispose` pattern.

Before you wire up the UI, there's one nagging problem you should address: there's no automatic refresh of the data when you log in. Let's fix that now.

F.3 Fixing the login refresh problem

Many of the domain service methods require authentication and/or the user to be in a specific role. Up until now, we've had to log in, click another page, then click back to force the data to load. That's just bad coding. Fixing the login refresh doesn't require MVVM by any means, but it's nice and easy to do here, so let's fix it. Listing F.3 shows the changes to the view model required to address this issue.

Listing F.3 ViewModel code to eliminate the login problem

```
public EmployeeViewModel()
{
    ...

    if (WebContext.Current.User.IsAuthenticated)
    {
        CanLoad = true;
        RefreshData();
    }

    WebContext.Current.Authentication.LoggedIn += new
        EventHandler<AuthenticationEventArgs>(Authentication_LoggedIn);

    WebContext.Current.Authentication.LoggedOut += new
        EventHandler<AuthenticationEventArgs>(Authentication_LoggedOut);
}

void Authentication_LoggedOut(object sender, AuthenticationEventArgs e)
{
    CanLoad = false;
}

void Authentication_LoggedIn(object sender, AuthenticationEventArgs e)
{
    CanLoad = true;
    RefreshData();
}
```

Existing constructor code

Could clear data here too

Refresh on login

The `WebContext` class supplies an `Authentication` object, which includes two useful events: `LoggedIn` and `LoggedOut`. You handle those events in the view model along with two other things:

- 1 You set `CanLoad` to `true` if logged in, `false` if logged out. The XAML view can use this property to enable/disable UI elements.

- 2 You call `RefreshData` when the user logs in. You could also clear the data when they log out.

You also check the value of the `User.IsAuthenticated` property to see if it's true when the view model is constructed. If the value is true, the user was already logged in when the view model was instantiated. Those simple changes get rid of the requirement to have to switch pages after login. This could also have been implemented at the page level, or in a reusable service shared by all view models.

With the login refresh issue out of the way, it's time to update the XAML UI and see how the application works.

F.4 Integrating the XAML UI

You've created the view model and provided loading capabilities as well as the security check. But you haven't yet linked it up to the view.

The first step in updating the UI is to set the data context of the page to the view model. You can do this either in XAML or in the code-behind. You can also use injection if that's your preferred approach. This listing shows how to set the data context from the code-behind for the page.

Listing F.4 Setting the data context in Home.xaml.cs

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DataContext = new EmployeeViewModel();
}
```

While you're in the code-behind, be sure to comment out the code inside the `DataSource_LoadedData` event (or remove the event altogether). You can also remove the `CalculateBonusButton_Click` handler as well as any other button click handlers you may have added. You'll replace those with command bindings.

Speaking of bindings: most of the binding statements in the XAML have changed, plus you added a Filter button to make filtering easier. Rather than try to point out all the deltas, I've included the full set of markup here.

Listing F.5 Updated Home.xaml markup

```
<navigation:Page x:Class="RiaExample.Home"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:navigation="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls.Navigation"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:dataForm="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls.Data.DataForm.Toolkit"
    mc:Ignorable="d"
    d:DesignWidth="640" d:DesignHeight="480"
    Style="{StaticResource PageStyle}">
    <Grid x:Name="LayoutRoot">
```

```

<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="350" />
  </Grid.ColumnDefinitions>
  <TextBlock Height="23" Width="84" Margin="6,10,0,0"
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Text="Title Contains"/>
  <TextBox x:Name="FilterText"
    Height="23" Margin="96,6,51,0"
    Text="{Binding FilterString, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Top" />
  <Button x:Name="Filter" Height="23" Width="40"
    Margin="0,6,5,0" Content="Filter"
    Command="{Binding FilterCommand}"
    HorizontalAlignment="Right"
    VerticalAlignment="Top"/>
  <sdk:DataGrid x:Name="EmployeeGrid"
    Grid.Column="0" Margin="0 40 5 40"
    IsEnabled="{Binding CanLoad}"
    SelectedItem="{Binding SelectedEmployee, Mode=TwoWay}"
    ItemsSource="{Binding Employees}" />
  <dataForm:DataForm Grid.Column="1"
    Margin="5 40 0 40"
    IsEnabled="{Binding CanLoad}"
    ItemsSource="{Binding Employees}"
    CurrentItem="{Binding SelectedEmployee, Mode=TwoWay}" />
  <sdk:DataPager Grid.ColumnSpan="2"
    IsEnabled="{Binding CanLoad}"
    Source="{Binding Employees}"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Bottom" />
  <Button x:Name="SubmitChanges"
    Grid.Column="1"
    Margin="5" Height="25" Width="120"
    HorizontalAlignment="Right"
    VerticalAlignment="Top"
    Command="{Binding SubmitChangesCommand}"
    Content="Submit Changes" />
  <Button x:Name="CalculateBonusButton"
    Grid.Column="1" Margin="5 5 130 5"
    HorizontalAlignment="Right"
    VerticalAlignment="Top"
    Height="25" Width="120"
    Command="{Binding AddVacationBonusCommand}"
    Content="Calc Bonus" />
</Grid>
</navigation:Page>

```

Filter string

New filter button

Employees collection view

The new markup now binds each of the elements to data on the view model rather than to other elements in XAML. This gives you more control over what happens and when in your code. One example of that is the filter: rather than let the `DomainDataSource` handle it whenever the text changes, you now have a button and a command wired up as well as the actual text in the view model. Let's look at how to use that now.

Each button is now wired up to a command object, which handles enabling/disabling the button when the function isn't available. In addition, the `DataGrid`, `DataForm`, and `DataPager` have their `IsEnabled` properties bound to the `CanLoad` property of the view model. These help prevent having an active UI when there's nothing the user can do, as in the case of when they aren't yet logged in.

At this point, you can now run the sample and see that there's data being returned. Because of the login check, simply run the app, log in, and verify that the data comes back without your having to do anything else. All working? Good. (The buttons are all enabled but nonfunctional because you haven't yet created their command objects.) Now let's add in the sorting, paging, and grouping functionality.

F.5 Sorting, paging, and grouping

The main reason I liked the `DomainDataSource` was that it was the only way to get paging, sorting, and grouping without writing a bunch of extra code. That's powerful stuff. Now, the `DomainCollectionView` provides the same type of functionality to your MVVM (or code-behind) applications.

In the previous version of the application, you sorted the data on the `Title` and `HireDate` fields by default. You then grouped on `Title`, and paged based on that. Just as was the case before, paging requires that you have an active sort in place—you can't page unsorted data.

F.5.1 Sorting

Using the `DomainCollectionView`, you can add sorting, paging, and grouping using code equivalents of the markup you had with the `DomainDataSource`. You'll start with the most common operation: sorting. Here are the updates to make to the view model to enable sorting.

Listing F.6 Updated `RefreshData` and `LoadEmployees` methods to enable sorting

```
private LoadOperation<Employee> LoadEmployees()
{
    var query = _context.GetEmployeesQuery();

    return _context.Load<Employee>(query.SortBy(_view));
}
private void RefreshData()
{
    using (_view.DeferRefresh())
    {
        _view.SortDescriptions.Clear();

        _view.SortDescriptions.Add(
            new SortDescription("Title",
                               ListSortDirection.Ascending));
        _view.SortDescriptions.Add(
            new SortDescription("HireDate",
                               ListSortDirection.Ascending));
    }
}
```

SortBy

Clear old sort

New sort criteria

This listing added two chunks of code to enable sorting. The first enables sorting in the `LoadEmployees` method by passing in `query.SortBy(_view)` rather than just `query`. The second, in the `RefreshData` method, adds the sort descriptions to the domain collection view. These descriptions are just like the ones you had in XAML—they specify a property name and a sort direction. The two modifications together sort the data.

F.5.2 Paging

With the data sorted, you can now page it to reduce the amount of data loaded on the client at any one time. As previously mentioned, paging requires a sorted set. If you don't sort the data first, you'll get an exception telling you that `Skip` can't be called on an unsorted set. This makes sense because data isn't guaranteed to come back in any particular order if it isn't sorted, and so a query couldn't guarantee that the paged data would make any sense.

This listing adds paging support to the sample application.

Listing F.7 Adding paging support to the view model

```
private LoadOperation<Employee> LoadEmployees()
{
    var query = _context.GetEmployeesQuery();

    return
        _context.Load<Employee>(query.SortAndPageBy(_view));
}
private void RefreshData()
{
    using (_view.DeferRefresh())
    {
        ... sort descriptions as before ...

        _view.PageSize = 20;
        _view.MoveToFirstPage();
    }
}
```

← **SortAndPageBy**

Set page size and move to first

There are two important changes here. First, you change the `SortBy` method to `SortAndPageBy`. This enables paging during the load operation. Second, you set the page size inside the `DeferRefresh` block. You also move to the first page, but that isn't a key function. But if you change sorting or paging values, you'll likely want to do that.

The final feature you'll look at in this section is grouping.

TIP If you run the sample application and no data appears, put a breakpoint in the `op.HasError` block in the `OnEmployeesLoaded` method of the view model. In that, inspect the `op.Error` property to see what error is returned. If you inspect the property using the property inspector rather than code, you'll need to look two `base` levels down before you'll find the `Error` property.

F.5.3 Grouping

Take a look at your email client. There's a pretty good chance that you're using some sort of grouping in the display, perhaps by day as is the default in Microsoft Outlook.

Grouping is an important and often overlooked way to make data more scannable for the user.

Once the data is sorted, it's a simple matter to enable grouping—a single change is required.

Listing F.8 Enabling grouping in the RefreshData method

```
private void RefreshData()
{
    using (_view.DeferRefresh())
    {
        ... sort descriptions as before ...

        _view.GroupDescriptions.Clear();
        _view.GroupDescriptions.Add(
            new PropertyGroupDescription("Title"));

        ... paging support as before ...
    }
}
```

← Group by the Title field

Run the application again. Now you have paging, sorting, and grouping again just as you did with the `DomainDataSource` but in an MVVM-appropriate way. Figure F.2 shows the application with sorting, paging, and grouping all working.

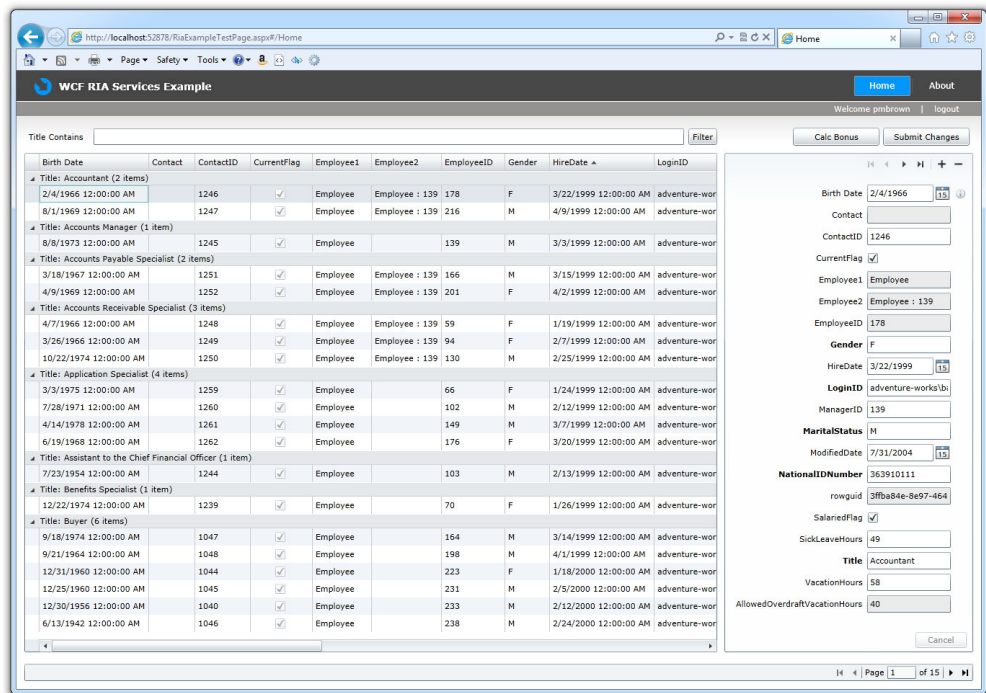


Figure F.2 The sample application with sorting, paging, and grouping all in place in an MVVM-friendly way.

In truth, getting these features working from code was almost as simple as specifying it in XAML, so I wouldn't hesitate to use this in other RIA Services applications. The new `DomainCollectionView` class and the `ICollectionView` interface, as well as the support classes such as `DomainCollectionViewLoader`, provide the flexibility an MVVM implementation backed by RIA Services requires.

Sorting, paging, and grouping are definitely important parts of the `DomainCollectionView` and help bring parity with the `DomainDataSource` you used in appendixes D and E. But you've only scratched the surface of what these objects can do. And if they don't do everything you require, you can implement your own loaders or even your own `ICollectionView` alternate. The RIA Services team isn't big on publishing documentation, so if you want to learn more about these features, you'll find the forums at Silverlight.net and blogs such as Kyle's (mentioned previously) indispensable.

You're not quite done yet. Although I've covered the majority of the functionality from the code-behind and XAML version of the sample application, you still haven't wired up any of the buttons to do things like filtering, calculating the bonus, or saving changes. I'll look at filtering next.

F.6 Filtering

The `DomainDataSource` control provided easy support for filtering data. All you needed to do was create a filter descriptor and bind it to the `TextBox` on the form. In the new model the filtering is slightly more complex but it's more flexible.

When working in code, the filter can be anything you'd like, implemented in the way that makes the most sense for your application. In most cases, you'll use one or more `Contains` statements appended to the LINQ query in `LoadEmployees`. You're filtering on a single field, `Title`, so you'll use a single `Contains` statement in this example.

Of course, you could use any other LINQ method or operator you'd like. For a string value, you could use `StartsWith` or `EndsWith`, for example. You could even implement your own Soundex-type algorithm and use that instead.

In this section you'll wire up command support for the filter button using the `ViewModelCommand` for the first time in this sample. You'll then implement the filter in the data loading code. The UI is already bound to the correct properties, so there's nothing you'll need to update in XAML.

F.6.1 Applying the Command pattern

The first thing to do is to wire the Filter button to the `ViewModel`. You'll use the Command pattern to bind the button to the view model. As you learned in chapter 33, using the Command pattern helps decouple the UI from the code and automatically

handles enabling/disabling the button based on the ability to execute the action. This shows how to do that using the `ViewModelCommand`:

Listing F.9 Command pattern support for filtering

```
private ViewModelCommand _filterCommand = null;
public ViewModelCommand FilterCommand
{
    get
    {
        if (_filterCommand == null)
        {
            _filterCommand = new ViewModelCommand
            (
                p => ApplyFilter(),
                p => CanFilter
            );
        }
        return _filterCommand;
    }
}

public bool CanFilter
{
    get { return CanLoad && !String.IsNullOrEmpty(FilterString); }
}

public void ApplyFilter()
{
    RefreshData();
}

private string _filterString;
public string FilterString
{
    get { return _filterString; }
    set
    {
        _filterString = value;
        NotifyPropertyChanged("FilterString");
        NotifyCanFilterChanged();
    }
}

protected void NotifyCanFilterChanged()
{
    NotifyPropertyChanged("CanFilter");
    FilterCommand.OnCanExecuteChanged();
}
```

CanFilter

Reload data

CanFilter may change

Update command

The `FilterText` `TextBox` on the page is bound to the `FilterString` property in the view model using `TwoWay` binding. Whenever the user updates the text onscreen, the view model is changed to reflect that.

When the user clicks the Filter button, the bound command calls the `ApplyFilter` method. This method simply calls `RefreshData` to reload the data with the new filter in place. (You'll implement the updated `RefreshData` shortly.)

If you run the sample at this point, the first obvious change you'll see is that the Filter button is disabled before you log in and before you type anything in the filter box. This is a result of the `CanFilter` property and the call to `OnCanExecuteChanged` in the command. Once you load the data, the filter button is still disabled. That's because `CanFilter` is a calculated field, based on the value of `CanLoad`. When you change `CanLoad`, there's no automatic property change notification for `CanFilter`. To fix this, you need to address the change in `CanLoad`, as shown next.

Listing F.10 Updating `CanLoad`, which also updates `CanFilter`

```
private bool _canLoad = false;
public bool CanLoad
{
    get { return _canLoad; }
    set
    {
        _canLoad = value;
        NotifyPropertyChanged("CanLoad");
        NotifyCanFilterChanged();
    }
}
```

← Notify of potential
CanFilter change

The change to `CanLoad` wraps up all the infrastructure changes required to support filtering. The one thing you haven't yet done is the actual filtering of data.

F.6.2 Filtering the data

The filtering happens as part of the query in `LoadEmployees`. To keep the code and markup short and simple, you'll continue to filter on only one field, although you could certainly provide an updated UI and view model with more choices.

The next listing shows the updated `LoadEmployees` view model method with the filter implemented as part of the query.

Listing F.11 Updated filter code in the `LoadEmployees` method

```
private LoadOperation<Employee> LoadEmployees()
{
    var query = _context.GetEmployeesQuery();

    if (!string.IsNullOrEmpty(FilterString))
    {
        query = query.Where(e =>
            e.Title.Contains(FilterString));
    }

    return _context.Load<Employee>(query.SortAndPageBy(_view));
}
```

← Filter operation

The example shows the updated `LoadEmployees` method. The new code first checks to see if the `FilterString` is valid. If so, it adds a `Where` clause to the query to perform a `Contains` operation on the `Title` property using the `FilterString` as the parameter. The rest of the method remains the same.

The filter operation is the first one to use the Command pattern. There are a couple others that need to be wired up as well. This appendix wraps up with their implementation.

F.7 Domain context invoke and update operations

You've come a long way from the `DomainDataSource` control on XAML with supporting code-behind to the view model approach supported by the `DomainCollectionView` and `DomainCollectionViewLoader` classes, as well as the all-important context.

The only operation where you've used the context so far is simple loading of data, whether it be straight or filtered. But the context is your bridge to the server and, as such, contains a lot more functionality.

In this section, you'll first implement the invoke operation for calculating the vacation bonus. You may recall that this is a server-side call that's kicked off from a button on the client. It'll need command object support in the view model, as well as integration with the context to call the invoke method. Next, you'll submit the changes to the context, which will then pass them up to the server to save your entries to the database.

F.7.1 Calling the invoke operation

The domain service in the ASP.NET project contains a method for calculating the vacation bonus to apply to an employee's compensation. This method, `CalculateVacationBonus`, is an invoke operation. That means it's not a recognized CRUD operation; it's a standard service method. Therefore, to call this method you need to go through the client-side context object explicitly rather than rely on it being called automatically as part of a retrieve or update operation.

The first step in wiring up this method is to wire the button to the view model using the same Command pattern you used for the filter functionality. This listing shows most of the code required to wire those up.

Listing F.12 Command pattern support for calculating vacation bonus

```
private ViewModelCommand _addVacationBonusCommand = null;
public ViewModelCommand AddVacationBonusCommand
{
    get
    {
        if (_addVacationBonusCommand == null)
        {
            _addVacationBonusCommand = new ViewModelCommand
            (
                p => AddVacationBonus(),
                p => CanAddVacationBonus
            );
        }
        return _addVacationBonusCommand;
    }
}
```

← Execute functionality

```
public bool CanAddVacationBonus
{
    get { return SelectedEmployee != null; }
}
```

Everything you see in the listing is following the same pattern established by the filter command method. But the command's execute functionality, `AddVacationBonus`, isn't present in this listing. I've broken that out into a separate listing because that's where all the interesting stuff happens. This listing delivers the goods:

Listing F.13 Calling the invoke method from the command

```
public void AddVacationBonus()
{
    if (SelectedEmployee != null)
    {
        DateTime hireDate = SelectedEmployee.HireDate;
        var invokeOp = _context.CalculateVacationBonus(
            hireDate, OnCalculateBonusInvokeCompleted,
            SelectedEmployee);
    }
}

private void OnCalculateBonusInvokeCompleted(
    InvokeOperation<int> invokeOp)
{
    if (invokeOp.HasError)
    {
        invokeOp.MarkErrorAsHandled();
    }
    else
    {
        Employee emp = invokeOp.UserState as Employee;
        if (emp != null)
        {
            emp.VacationHours += (short)invokeOp.Value;
        }
    }
}
```

← Hire date

← Better error handling needed

← Apply bonus

The listing has two methods. The first, `AddVacationBonus`, is called by the command's execute method. This code in turn gets the hire date and passes that in to the `CalculateVacationBonus` method of the context object.

The code for this method is identical to the code you had in the code-behind version, except there's no error message displayed in a `MessageBox` when there's an error. Most MVVM toolkits include their own mechanisms for displaying UI messages, so the implementation will vary. Nevertheless, make sure you do something besides just make the error as handled.²

Before you can say you've completed this functionality, there's one last bit of housekeeping to do. `AddVacationBonusCommand` relies on the selected employee being valid; it disables the button if there's no selected employee. When the

² That's my favorite type of error handling: "On Error Resume Next" and forget about it. Fly error, be free!

`SelectedEmployee` property is set, the command isn't notified—the same problem you had with the filtering command. To correct this, update the `SelectedEmployee` setter to notify the command of the potential change.

Listing F.14 Updated `SelectedEmployee` setter

```
private Employee _selectedEmployee;
public Employee SelectedEmployee
{
    get { return _selectedEmployee; }
    set
    {
        _selectedEmployee = value;
        NotifyPropertyChanged("SelectedEmployee");
        AddVacationBonusCommand.OnCanExecuteChanged();
    }
}
```

← Notify command of change

That's all there is to moving the vacation bonus functionality over. If you run the application at this point and select a row in the grid, you should be able to update the vacation time based on the hire date. Click the Calc Bonus button and watch the `VacationHours` field for changes.

Calling invoke operations from a view model requires little effort versus using code-behind. The only real changes you need to make are to (optionally) use the Command pattern and refactor any UI display to use the notification system of your toolkit of choice.

The next and final step is to enable that Submit Changes button so you can save the newly added bonus. I know I'd be disappointed if my vacation bonus only lasted as long as the user had this screen up.

F.7.2 Saving changes

In the XAML and code-behind version of this sample application, the Submit Changes button was wired to a command exposed by the `DomainDataSource` control. This was all accomplished in XAML, without any code.

As much as I like XAML and the power of binding, I've never been much of a fan of "Look, Ma, no code!" being a measure of success, especially when it's eliminating only a tiny bit of code as you'll see here.

In the remainder of this section, you'll wire up the Submit Changes button to a command on the view model and implement the code in the view model to send the changes through the context and up to the server.

First, just as you did twice before, you need to hook up the button. Here's how.

Listing F.15 Command pattern implementation for Submit Changes

```
private ViewModelCommand _submitChangesCommand = null;
public ViewModelCommand SubmitChangesCommand
{
    get
    {
```



```

        if (_submitChangesCommand == null)
        {
            _submitChangesCommand = new ViewModelCommand
            (
                p => SubmitChanges(),
                p => CanSubmitChanges
            );
        }
        return _submitChangesCommand;
    }
}
public bool CanSubmitChanges
{
    get
    {
        return WebContext.Current.User.IsInRole("Manager") &&
            _context.HasChanges;
    }
}

public void SubmitChanges()
{
    _context.SubmitChanges();
}

```

← Manager check

← Submit changes

Submitting changes to the domain service is surprisingly easy. All you need to do is wire up the command and call `SubmitChanges` on the context object. That's it. It's not much more complex than wiring up the whole thing in XAML as you did before.

To provide the same button enabling the XAML method provided, you want to enable change saving only when there are actually changes to save. The domain context object exposes a `HasChanges` property. It also exposes a `PropertyChanged` event like any good INPC (`INotifyPropertyChanged`)-implementing class. In the constructor, you'll hook the `PropertyChanged` event and see if `HasChanges` has changed. If so, you'll notify the command object, as shown in this listing.

Listing F.16 Updating the command when `HasChanges` changes

```

public EmployeeViewModel()
{
    _context.PropertyChanged +=
        new PropertyChangedEventHandler(OnContextPropertyChanged);
}

void OnContextPropertyChanged(object sender,
    PropertyChangedEventArgs e)
{
    if (e.PropertyName == "HasChanges")
        SubmitChangesCommand.OnCanExecuteChanged();
}

```

Notify command

If you want to be really complete, you'll need to call the `OnCanExecuteChanged` in the login/logout event handlers as well. That's a simple change I'll leave to you.

Run the application and make some changes to the data. May I suggest that you use the Calc Bonus button as one way to make those changes? Notice how the Submit Changes button lights up once you apply the changes to the field. Now click that button

to commit the changes. Notice how the button is disabled again? Refresh the page and verify that the changes were actually saved. They were? Excellent. We're done here.

Event handler shortcuts

When wiring up event handlers, I typically use either the default visual studio syntax, or the lambda expression syntax. Yet for those cases where you want to create a separate event handler method as you have in this appendix, the “new WhateverEventHandler” part is entirely optional. For example, instead of writing this:

```
_context.PropertyChanged +=  
    new PropertyChangedEventHandler(OnContextPropertyChanged);
```

you could simply write:

```
_context.PropertyChanged += OnContextPropertyChanged;
```

The advantages to that are minimal, but I just ran into a case on some interim Windows 8 Runtime stuff where not using the event handler type would've been helpful, because the name had changed.

Calling `Invoke` operations from the view model is just as easy as it was from code-behind. You followed the Command pattern in this appendix, but as discussed in chapter 33, that's not a hard and fast requirement for MVVM—use it if it makes sense in your application.

Saving data via the domain service is even easier. The client-side context object provides a `SubmitChanges` method that sends to the server all the modifications: creates, updates, and deletes; and calls the appropriate server-side methods associated with those modifications.

F.8 Summary

Deciding to use WCF RIA Services no longer precludes the use of the MVVM pattern. The `DomainCollectionView` and its related classes, plus the domain context, provide all of the functionality you had with the `DomainDataSource` in XAML.

In this appendix you used the MVVM skills you learned in chapter 33 and applied them to the RIA Services application built in appendixes D and E. You didn't look at any particular MVVM toolkit nor did the application incorporate IoC or other helpful but external patterns. You did use the Command pattern because that's one of the patterns that tends to be hard to escape as you get deeper into MVVM; it's simply too easy and useful. Unlike chapter 33, there were no additional service abstractions in this code. RIA Services provides enough encapsulation of the domain methods for me. You, of course, are free to use whatever patterns or additional methods you want. The key thing to understand is how to use the `DomainCollectionView` and the loader and context classes to accomplish your goals.

Silverlight 5 IN ACTION

Pete Brown



This hands-on guide explores Silverlight from the ground up, covering every feature in rich, practical detail. It is readable and the coverage is comprehensive. You'll master networking, MVVM, and more, with dozens of code samples you can use in Visual Studio or the free Visual Web Developer Express.

Silverlight 5 in Action teaches you how to build desktop-quality applications you can deploy on the web. Beginners will appreciate the progression from simple examples to full applications that employ good design and coding practices. Seasoned .NET developers will love how the sample code embraces and extends what they already know.

What's Inside

- 2D and 3D graphics and animation
- Business application services, rules, and validation
- The MVVM pattern and testing
- 5 free appendixes (200 pages) available online

A background in C# or VB.NET is helpful, but no knowledge of Silverlight or XAML is required. This book is a revised edition of *Silverlight 4 in Action*.

Pete Brown leads Microsoft's Silverlight/XAML Developer Community team. He's been a Silverlight MVP, an INETA speaker, and successful RIA Architect.

To download a free eBook plus additional content in PDF, ePub and Kindle formats, owners of this book should visit manning.com/Silverlight5inAction

“As entertaining as it is educational. Pete covers Silverlight 5 like no one else!”

—Joe Suchy, ATC Transportation

“The go-to source for real Silverlight 5 answers and examples.”

—Dave Campbell, WynApse

“Hands down THE best reference for everything Silverlight.”

—Michael Crump, Telerik

“A must-have book for every Silverlight developer.”

—Dave Davis
BlueMetal Architects Inc.

ISBN 13: 978-1-61729-031-2
ISBN 10: 1-61729-031-9



9 781617 129031