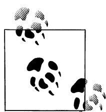
small Flash modules and link application libraries that are made with the same version of the Flex SDK. You'll also investigate the advantages of *runtime shared libraries* (RSLs), including how to use them with modules and how to optimize them.

Planning for Modularization

After deciding to use the module and library approach, you need to carefully review all of your application resources besides the ActionScript or MXML code—namely, images, sounds, and movies (*.swf* files)—to decide if you really need to embed them. The rule of thumb for images is that unless they must be displayed on the first page they should not be embedded. It is almost never worthwhile to embed any sizable sound or movie *.swf* file in a Flex application, as you can gain much better control of their execution by using streaming and starting playback when just enough data has been loaded.



Embedded images required in your RIA should be located in a separate Flash Builder project and loaded as an RSL.

The next step is to separate stylesheets and skins into modules. Doing so offers three advantages. First, it separates the work of designers from that of application developers. Second, removing stylesheets and skins from the compilation process significantly reduces rebuild time during development, because the cost of resource *transcoding* (compilation of fonts and styles) is high. Third, keeping skins and stylesheets outside of the modules simplifies initialization and obviates unnecessary reloading and reapplying of CSS, thus making module initialization faster and safer.

Precompile your CSS into a *.swf* file (right-click on the file to see this option) and then load it from the main application using the `StyleManager` class.

RSLs do introduce performance issues, though: they are loaded one by one and thus impose a "roundtrip" effect. Breaking a monolithic application into 10 RSLs results in additional roundtrip requests to the server and as a result slightly increases the initial load time. A solution to this problem is to use smarter loading of multiple RSLs by modifying the source code of the `RSLListLoader` class available in the SDK and placing it in your application (we'll cover this technique later in this chapter). If you use this approach, you must take special care to ensure that any framework libraries other elements depend upon are loaded first.

Another rule of thumb for large applications is to make the first page as light and free of dependencies as possible. In other words, keep the first page super small. Once all the system and CSS RSLs are loaded and the application enters the preinitialize event, you can start loading the rest of the application code. We recommend that you use the portal approach discussed in Chapter 7 as a starting point for any large application, as

it provides a clean break between applications. We'll cover this topic in the section "Optimizing RSL Loading" on page 32.

It Takes Two to Perform

Fast applications are your goal, but how do you get there? On the one hand, the RIA deployed on the server should consist of a number of relatively small *.swf*, *.swc*, and asset files. On the other, ideally, the end users should use fast and reliable network connections. First let's define how fast your RIA should appear, and then we'll look at how quickly the data arrives at the user's machine.

The major difference between an internal enterprise application and a consumer-facing RIA is that the former runs on fast and reliable networks at predictable speeds, while the latter runs in a Wild West with unknown bandwidth. You have to set proper expectations for your RIA's download speed from the start. To do that, you need a *service level agreement* (SLA).

The idea behind an SLA is that your project's stakeholders agree to and sign off on acceptable delivery speeds for your application and its data. For instance, if your application will run on a 15Mbps intranet, the main page of the application should appear in less than about 7 seconds. If yours is a consumer-facing application, you can reasonably expect that the users should have network connections with at least 768 Kbps of bandwidth. To put yourself into their shoes, you should run tests emulating transfers at this speed; for example, you can use the HTTP proxy/monitor *Charles* (see "Troubleshooting with Charles" in Chapter 4), or a hardware network emulator. To keep down the initial response time, you'll need to make sure that the initially downloadable portion of your application is smaller than 1MB.

To simulate WAN problems, we use several Linux boxes (both virtual and physical ones). Setting up a testing environment can be tedious, though, and you may consider using a simple portable appliance that will turn the simulation of a slow environment into a trivial task. One such portable, inexpensive, and easy to use network simulator is Mini Maxwell.

After an enterprise application has been downloaded, often it starts transferring serious amounts of data from the server. That data should arrive quickly and safe and sound. RIA applications are extremely susceptible to network problems. Even a small probability of lost or misdelivered packages becomes significant when multiplied by the sheer number of small data packages involved. Lost, duplicate, and reordered packages combined with high latency and low bandwidth can cause significant issues for applications fully tested *only* on reliable intranets and then released into the wild of unreliable WAN communications.

Purposely increasing (with software or hardware) the simulated latency up to a realistic 200 ms and the package loss to an admittedly unrealistic 10% will quickly expose any problems in your application's error-handling code. It will also give you a quick feel for the robustness of the code. You can then check whether duplicate or out-of-sequence packages will adversely affect your application, as described in Chapter 5.

While consulting with one of our customers, a foreign exchange trading company, we had to enhance the endpoints in the RTMP protocols to ensure that out-of-sequence messages and network congestion were dealt with properly. The exact remedies you may need to apply in such situations will depend on the communication protocols used by your RIA.

Obviously, with SOAP Web Services and similar high-level protocols you have very loosely bound communications, making implementation of a quality of service (QoS) layer impossible. As the web browsers limit the number of simultaneous `HTTPRequest`s per domain, the latency can cause performance slowdown and timeouts. Missing communication packages escalate the issue with connection starving even further.

If you use one of the AMF implementations for data-intensive applications, they will perform a lot faster (the *.swf* arrival time remains the same). With AMF, the latency is less of a problem as Flex automatically batches server requests together. Implementing symmetrical checkpoints on both the client and server *endpoints* allows processing of lost and duplicate packages, but the lost packages remain a problem as they cause request timeouts.

The robustness of a RIA improves if you move from HTTP/SOAP/REST to either the RTMP protocol or the BlazeDS long-polling AMF. Keeping connections open and using two-way sockets are ideal for high-performance and reliable protocols. Comparing them to `HTTPRequest`s is like comparing a multilane highway with traffic flowing in each direction to a single-track dirt road.

These days, more and more enterprise applications are being built using always-connected protocols for tasks ranging from regular RPC to module loading implementing streaming (the same way movies are streamed). As these protocols evolve, we're likely to see more open source products that provide transparent implementations using a mixture of protocols. Meanwhile, we can mix protocols using Flex techniques such as configuring the fallback channels.

Application Startup and Preloaders

Perceived performance is as important as actual performance. While a large Flex application loads, users may experience unpleasant delays. Rather than frustrating them with inactivity, a useful approach is to give the users something productive to work on. This can be a main window of your application, or just a logon view; the point is that

this very first view should be extremely lightweight and arrive on the user's machine even before the Flex frameworks and the rest of the application code start downloading. Giving users the ability to start working quickly with partially loaded code creates the perception of your application loading rapidly.

In this section you'll learn how to create and load a rapidly arriving logon screen to get the user occupied immediately. Here are the four challenges you face:

- The logon screen has to be very lightweight. It must be under 50 KB, so using classes from Flex frameworks is out of the question.
- The application shouldn't be able to remove the logon window once it's loaded, as the user must log in first.
- If the user completes the login before the application has fully loaded, the standard progress bar has to appear.
- The application should be able to reuse the same lightweight logon window if the user decides to log out at any time during the session.

What Happens in Flash Player Before a Flex Application Is Loaded

`SystemManager` is a main manager that controls the application window; creates and parents the `Application` instance, pop-ups, and cursors; manages the classes in the `ApplicationDomain` container (see the Flex language reference at *http://livedocs.adobe.com/flex/gumbo/langref/*); and more. `SystemManager` is the first class that Flash Player instantiates in the first frame of your application (modules and subapplications have their own `SystemManager` classes). `SystemManger` is responsible for loading all RSLs, which will be discussed later in this chapter.

Hanging off of a `stage` object, `SystemManger` stores the size and position of the main application window and keeps track of its children, such as floating pop-ups and modal windows. Using `SystemManager`, you can access embedded fonts, styles, and the `document` object. `SystemManager` also controls application domains, which are used to partition classes by security domains.

If you're developing custom visual components (descendents of the `UIComponent` class), keep in mind that initially such components are not connected to any display list and `SystemManager=null`. Only after the first call of `addChild()` is `SystemManager` assigned to them. You should not access `SystemManager` from the constructor of your component, because at that point it can still be `null`.

In general, when the `Application` object is created, the process is:

1. The `Application` object instantiates.
2. The `Application` dispatches the `FlexEvent.PREINITIALIZE` event at the beginning of the initialization process.
3. At this point each of the `Application` child components is constructed, and each component's `createChildren()` method is called.

4. The `Application` dispatches the `FlexEvent.INITIALIZE` event, which indicates that all of the application's components have been initialized.
5. The `Application` dispatches the `FlexEvent.CREATION_COMPLETE` event.
6. `SystemManager` removes the `Preloader` object.
7. Flash Player dispatches the `FlexEvent.APPLICATION_COMPLETE` event.

In most cases, you should use the MXML tag `<mx:Application>` to create the `Application` object, but if you need to do this in ActionScript, do not create components in the constructor. Instead, override `createChildren()` to ensure proper timing in the initialization process.

Unlike Flash movies that consist of multiple frames being displayed over a timeline, Flex *.swf* files utilize only two frames. The `SystemManager`, `Preloader`, `DownloadProgressBar`, and a handful of other helper classes live in the first frame. The rest of the Flex framework, your application code, and embedded assets like fonts and images reside in the second frame.

When Flash Player initially starts downloading your *.swf*, as soon as enough bytes come for the first frame it instantiates a `SystemManager`, which creates an instance of `Preloader`. This instance monitors the application download and initialization and in turn creates a `DownloadProgressBar`.

When all bytes for the first frame are in, `SystemManager` sends the `FlexEvent.ENTER_FRAME` for the second frame, and then renders other events.

Dissecting LightweightPreloader.swf

The sample application that will demonstrate how these challenges are resolved is located in the Eclipse Dynamic Web Project called *lightweight-preloader*. This application is deployed on the server. Note that the interactive login window (Figure 8-1) arrives from the server very quickly, while the large application *.swf* file continues downloading; this process may or may not have completed before the user enters his credentials and hits the Login button.

The login view was created in Photoshop and then saved as an image. Figure 8-2 depicts the directory structure of the Flash Builder project *lightweight-preloader*. As you can see, the *assets* directory contains the image file *logon.psd* created in Photoshop, and a lighter *.png* version. At this point, any Flash developer can open this file in the Flash Professional IDE, add a couple of text fields and a button, and save it as a Flash movie. This window can be saved in binary formats (*.fla* and *.swf*), but we've exported the file into a program written in ActionScript. The generated ActionScript code may not be pretty, and you might want to manually edit it, as we have done. The final version of this code (class `LightweightPreloader`) is shown later, in Example 8-2.

The text elements shown in Figure 8-1 are not Flex components, so they are not represented in the directory structure. Example 8-1 shows the code loading the *logon.png* file.

Figure 8-1. Login view of lightweight preloader

Example 8-1. The background of the login view

```
package com.farata.preloading{
    import flash.display.Bitmap;

    [Embed(source="assets/logon.png")]
    public class PanelBackground extends Bitmap
    {
        public function PanelBackground ()
        {
            smoothing = true;
        }
    }
}
```

The *logon.png* image file is 21 KB, and you can reduce its size further by lowering the resolution of the image. The ActionScript class `LightweightPreloader` that uses the `PanelBackground` class adds another 6 KB, bringing the total size of the precompiled *LightweightPreloader.swf* to a mere 27 KB. The Flex class `Preloader` will load this file in parallel with the larger *MainApplication.swf* file.

The total size of the `LightweightPreloader` class is 326 lines of code. Most of this code was exported from Flash Pro, but some additional coding was needed. It's no doubt tempting to use Flex to create such a simple view in a dozen of lines of code, but keeping down the size of the very first preloaded *.swf* file is a lot more important than minimizing the amount of manual coding required. This is the only case where we advocate manual coding rather than taking advantage of the automation offered by Flex!

Using Flash Catalyst to generate the `LightweightPreloader` code from a Photoshop image is also not advisable in this case, because Flash Catalyst uses Flex framework objects, which would substantially increase the size of the *LightweightPreloader.swf* file.

Figure 8-2. Flash Builder project lightweight-preloader

The code for the `LightweightPreloader` class is shown in Example 8-2. Note that the imports section doesn't include any of the classes from the Flex framework.

Example 8-2. LightweightPreloader.as

```
package{
    import com.farata.preloading.BitmapLoginButton;
    import com.farata.preloading.ILoginWindow;
    import com.farata.preloading.LoginButtonNormal;
    import com.farata.preloading.LoginEvent;
    import com.farata.preloading.PanelBackground;

    import flash.display.DisplayObject;
    import flash.display.InteractiveObject;
    import flash.display.Sprite;
```

```
    import flash.events.Event;
    import flash.events.FocusEvent;
    import flash.events.IOErrorEvent;
    import flash.events.KeyboardEvent;
    import flash.events.MouseEvent;
    import flash.events.SecurityErrorEvent;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.net.URLVariables;
    import flash.text.TextField;
    import flash.text.TextFieldType;
    import flash.text.TextFormat;
    import flash.text.TextFormatAlign;
    import flash.ui.Keyboard;
    import flash.utils.Dictionary;
    import flash.net.SharedObject;

    public class LightweightPreloader extends Sprite
        implements ILoginWindow{
        public static const loginURL:String = "login";
        public static const LOGIN_INCORRECT_MESSAGE:String =
                                "Failed. Use your myflex.org credentials";
        public static const HTTP_ERROR_MESSAGE:String =
                                "Connection error. Please try again.";

        private var testMode:Boolean = true; // No server data available
        private var loginField:TextField;
        private var passwordField:TextField;
        private var messageField:TextField;
        private var loginButton:DisplayObject;
        public var background:PanelBackground;
        private var focuses:Array = new Array ();
        private var focuseMap:Dictionary = new Dictionary ();
...

        private function doInit ():void{
            background = new PanelBackground ();
            addChild (background);

            loginField = new TextField ();
            addChild (loginField);
            configureTextField (loginField);

            passwordField = new TextField ();
            addChild (passwordField);
            configureTextField (passwordField);
            passwordField.displayAsPassword = true;

            messageField = new TextField ();
            addChild (messageField);
            messageField.type = TextFieldType.DYNAMIC;
            var format:TextFormat = new TextFormat ("_sans", 12, 0xFF0000);
            format.align = TextFormatAlign.CENTER;
            messageField.defaultTextFormat = format;
```

```
            messageField.selectable = false;
            messageField.width = 300;
            messageField.height = 20;

            loginButton = new BitmapLoginButton ();
            addChild (loginButton);
            loginButton.addEventListener (KeyboardEvent.KEY_DOWN,
                                                  onButtonKeyboardPress);
            loginButton.addEventListener (MouseEvent.CLICK, onButtonClick);
            loginButton.addEventListener (FocusEvent.KEY_FOCUS_CHANGE,
                                                  onFocusChange);
            loginButton.addEventListener (FocusEvent.FOCUS_OUT, onFocusOut);
            focuses.push (loginButton);
            focusMap [loginButton] = true;

            var so:SharedObject = SharedObject.getLocal("USER_INFO");
            if (so.size > 0) {
                try {
                    var arr:Array = so.data.now;
                    loginField.text = arr[0];
                    passwordField.text = arr[1];
                }
                catch(error:Error) {
                   // Error processing goes here
                }
            }

            if (stage != null) {
                stage.stageFocusRect = false;
                stage.focus = loginField;
                focus = loginField;
            }
        }
...
        private function doLayout ():void{
            loginField.x = 230;
            loginField.y = 110;

            passwordField.x = 232;
            passwordField.y = 163;

            loginButton.y = 200;
            loginButton.x = (background.width - loginButton.width) / 2;

            messageField.y = 215;
            messageField.x = 65;
        }

        private function onEnterPress (event:KeyboardEvent):void{
            if (event.keyCode == Keyboard.ENTER){
                onLogin ();
            }
        }

        private function onButtonKeyboardPress (event:KeyboardEvent):void{
```

```
            if ((event.keyCode == Keyboard.ENTER) ||
                (event.keyCode == Keyboard.SPACE)) {
                onLogin ();
            }
        }

        private function onButtonClick (event:MouseEvent):void{
            onLogin ();
        }

        private function onLogin ():void {
            if (testMode) {
                onLoginResult ();
            }
            else {
                try{
                    var request:URLRequest = new URLRequest (loginURL);

                    var thisURL:String = loaderInfo.url;
                    if (thisURL.indexOf ("file") < 0) {
                        var variables:URLVariables = new URLVariables ();
                        variables.user = loginField.text;
                        variables.password = passwordField.text;
                        variables.application = "Client Reports";
                        request.data = variables;
                    }

                    var loader:URLLoader = new URLLoader (request);

                    loader.addEventListener (SecurityErrorEvent.SECURITY_ERROR,
                                                              onSecurityError);
                    loader.addEventListener (IOErrorEvent.IO_ERROR, onIOError);
                    loader.addEventListener (Event.COMPLETE, onLoginResult);
                    loader.load (request);
                }
                catch (e:Error) {
                    messageField.text = HTTP_ERROR_MESSAGE;
                }
            }
        }

...
        private function onLoginResult (event:Event = null):void{
            if (testMode) {
                dispatchEvent (new LoginEvent (LoginEvent.ON_LOGIN, "test",
                                                                    null));
            }
            else {
                var loader:URLLoader = URLLoader(event.target);
                if (loader.dataFormat == URLLoaderDataFormat.TEXT) {
                    var response:String = loader.data;
                    var responseXML:XML = new XML (response);
                    var status:String = responseXML.status [0];
                    if (status == "1"){
                        var so:SharedObject =
```

```
                            SharedObject.getLocal("USER_INFO");
                    so.data.now = new Array (loginField.text,
                                             passwordField.text);
                    so.flush();
                    dispatchEvent (new LoginEvent (LoginEvent.ON_LOGIN,
                            "no session available", // No sessionID for now
                            responseXML));
                } else
                    messageField.text = LOGIN_INCORRECT_MESSAGE;
            }
            else {
                messageField.text = HTTP_ERROR_MESSAGE;
            }
        }
    }
  }
}
```

This class extends `flash.display.Sprite`, a very light display node that can have children and display graphics. It adds the image displayed in Figure 8-1 as a background (see the method `doInit()` in Example 8-2):

```
background = new PanelBackground ();
addChild (background);
```

On top of this background, `doInit()` adds a couple of `flash.text.TextField` controls and a subclass of the `flash.display.SimpleButton` shown in Example 8-3.

Example 8-3. BitmapLoginButton.as

```
package com.farata.preloading{
    import flash.display.DisplayObject;
    import flash.display.SimpleButton;

    public class BitmapLoginButton extends SimpleButton{

        public function BitmapLoginButton (){
            super(new LoginButtonNormal (),
                    new LoginButtonOver (),
                    new LoginButtonPress (),
                    new LoginButtonNormal ());
            useHandCursor = false;
        }
    }
}
```

The constructor of the `SimpleButton` takes tiny wrapper classes with images representing different states of the button, as shown in Example 8-4.

Example 8-4. LoginButtonOver.as

```
package com.farata.preloading{
    import flash.display.Bitmap;

    [Embed(source="assets/login_button_over.png")]
```

```
    public class LoginButtonOver extends Bitmap{

        public function LoginButtonOver (){
            smoothing = true;
        }
    }
}
```

This is pretty much it; the graphic portion is taken care of.

The login functionality in a typical Flex application should be initiated from inside the Flex code and not from the HTML wrapper. This will allow you to minimize the vulnerability of the application as you eliminate the step where the user's credentials have to be passed from JavaScript to the embedded *.swf* file.

Example 8-5 contains the authentication code from the `LightweightPreloader` class's `onLogin()` method.

Example 8-5. Authenticating the user from ActionScript

```
var request:URLRequest = new URLRequest (loginURL);
        var thisURL:String = loaderInfo.url;
        if (thisURL.indexOf ("file") < 0) {
            var variables:URLVariables = new URLVariables ();
            variables.user = loginField.text;
            variables.password = passwordField.text;
            variables.application = "Client Reports";
            request.data = variables;
        }

        var loader:URLLoader = new URLLoader (request);

        loader.addEventListener (SecurityErrorEvent.SECURITY_ERROR,
                                                        onSecurityError);
        loader.addEventListener (IOErrorEvent.IO_ERROR, onIOError);
        loader.addEventListener (Event.COMPLETE, onLoginResult);
        loader.load (request);
```

The code in Example 8-5 creates a `URLRequest` object wrapping the values entered in the Flex view. The `URLLoader` makes a request to the specified URL that authenticates the user and returns a piece of XML describing the user's role and any other business-specific authorization parameters provided by your web access management system (e.g., SiteMinder from CA). No sensitive data exchange between JavaScript and the *.swf* file is required.

The function `onLoginResult()` gets the user's data from the server and saves this data as an XML object on the local disk, via a `SharedObject` API providing functionality similar to cookies.

The Main SWF Talks to LightweightPreloader

The main application (*MainApplication.mxml*, presented in Example 8-6) was written with the use of the Flex framework, and it communicates with the external *LightweightPreloader.swf* file via an additional class called `LoginPreloader`, shown in Example 8-7 (note the fourth line in Example 8-6, `preloader="com.farata.preloading.LoginPreloader"`). We've embedded several images into the *MainApplication.mxml* file just to make the *.swf* file extremely heavy (10MB), to illustrate that the login window appears fast and that the main application may continue loading even after the user has entered her login credentials and pressed the Login button.

Example 8-6. MainApplication.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    preloader="com.farata.preloading.LoginPreloader"
    layout="vertical"
    horizontalAlign="center"
    backgroundColor="white"
    verticalAlign="top">
    <mx:Script>
        <![CDATA[
            import mx.containers.TitleWindow;
            import mx.containers.Panel;
            import com.farata.preloading.ILoginWindow;
            import mx.managers.PopUpManager;
            import mx.core.UIComponent;
            import com.farata.preloading.LoginEvent;

            public function set sessionID (value:String):void{
                // Code to store app. Specific session ID goes here.
                trace ("sessionID in Main: " + value);
            }

            public function set loginXML (value:String):void{
                // Code to process authorization XML goes here
                trace ("loginXML in Main: " + value);
            }

            private var loginPanel:Panel;
            private var content:Sprite;

            // Embed several large images just to increase the size
            // of the main SWF to over 10MB.
            // This is done to illustrate fast preloading
            // of the LightweightPreloader login window.
            [Embed(source="com/farata/preloading/assets/Pic1.JPG")]
            public var pic1:Class;
            [Embed(source="com/farata/preloading/assets/Pic2.JPG")]
            public var pic2:Class;
            [Embed(source="com/farata/preloading/assets/Pic3.JPG")]
            public var pic3:Class;
```

```
[Embed(source="com/farata/preloading/assets/Pic4.JPG")]
public var pic4:Class;

private function onLogout ():void {
    var loader:Loader = new Loader ();
    var url:URLRequest = new URLRequest
                           ("LightweightPreloader.swf");
    var context:LoaderContext = new LoaderContext ();
    var applicationDomain:ApplicationDomain =
                           ApplicationDomain.currentDomain;
    context.applicationDomain = applicationDomain;
    loader.load (url, context);
    loader.contentLoaderInfo.addEventListener
                           (Event.COMPLETE,onLoginLoaded);
}

private function onLoginLoaded (event:Event):void {
    content = event.target.content as Sprite;
    var component:UIComponent = new UIComponent ();
    loginPanel = new TitleWindow ();
    loginPanel.title = "Log In";

    component.addChild (content);
    loginPanel.addChild(component);

    PopUpManager.addPopUp(loginPanel, this, true);

    (content as ILoginWindow).activate();

    component.width = content.width;
    component.height = content.height;
    PopUpManager.centerPopUp(loginPanel);

    content.addEventListener (LoginEvent.ON_LOGIN, onLogin);
}

private function onLogin (event:LoginEvent):void {
    (content as ILoginWindow).deactivate();
    PopUpManager.removePopUp (loginPanel);
    loginPanel = null;
    content = null;

    passParamsToApp(event);
    focusManager.activate();
}

private function passParamsToApp (event:LoginEvent):void {
    for (var i:String in event) {
        try {
            this [i] = event [i];
        } catch (e:Error) {
            trace ("There is no parameter " + i +
                         "in " + this + " defined");
        }
    }
```

```
            }
        ]]>
    </mx:Script>
    <mx:ApplicationControlBar width="100%"
        horizontalAlign="right">

        <mx:Button click="onLogout()" label="Log Out" />
    </mx:ApplicationControlBar>
    <mx:VBox
        verticalAlign="middle"
        horizontalAlign="center"
        width="100%"
        height="100%">
        <mx:Panel
            title="Hello"
            paddingLeft="20"
            paddingRight="20"
            paddingTop="10"
            paddingBottom="10">
            <mx:Label text="Application" />
        </mx:Panel>

    </mx:VBox>
</mx:Application>
```

`LoginPreloader` is a subclass of `DownloadProgressBar`. It cares about two things:

- That the loading of the main application is finished and it can be displayed
- That the login request is complete

If the user presses the Login button before the main application (all 10MB, in this case) arrives, the `LoginPreloader` turns itself into a progress bar until the application is fully downloaded and displayed. The `LoginPreloader` therefore acts as a liaison between the main application and the `LightweightPreloader`. Its code is shown in Example 8-7.

Example 8-7. Classes LoginPreloader and UnprotectedDownloadProgressBar

```
package com.farata.preloading{
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.net.URLRequest;
    import flash.system.ApplicationDomain;
    import flash.system.LoaderContext;
    import flash.utils.getDefinitionByName;

    import mx.events.FlexEvent;
    import mx.managers.FocusManager;
    import mx.managers.IFocusManager;
    import mx.managers.IFocusManagerComponent;
    import mx.managers.IFocusManagerContainer;
    import mx.preloaders.DownloadProgressBar;
    import flash.utils.getTimer;
```

```
import flash.utils.Timer;
import flash.events.TimerEvent;

public class LoginPreloader extends DownloadProgressBar {
    private var loginWindow:Sprite;
    private var event:LoginEvent;
    private var loggedIn:Boolean = false;
    private var isLoaded:Boolean = false;
    private var appInited:Boolean = false;
    private var aPreloader:Sprite;
    private var progress:UnprotectedDownloadProgressBar;
    private var _displayTime:int;

    public function LoginPreloader(){
        super();
        _displayTime = getTimer();
        MINIMUM_DISPLAY_TIME = 0;
        var loader:Loader = new Loader ();
        var url:URLRequest = new URLRequest ("LightweightPreloader.swf");
        var context:LoaderContext = new LoaderContext ();
        var applicationDomain:ApplicationDomain =
                                         ApplicationDomain.currentDomain;
        context.applicationDomain = applicationDomain;
        loader.load (url, context);
        loader.contentLoaderInfo.addEventListener (Event.COMPLETE,
                                         onLoginLoaded);
    }

    private function onLoginLoaded (event:Event):void{
        var content:Sprite = event.target.content as Sprite;
        addChild (content);
        loginWindow = content;
        (loginWindow as ILoginWindow).activate();
        content.x = (stage.stageWidth - content.width) / 2;
        content.y = (stage.stageHeight - content.height) / 2;
        content.addEventListener (LoginEvent.ON_LOGIN, onLogin);
    }

    override public function set preloader( preloader:Sprite ):void {
        preloader.addEventListener( FlexEvent.INIT_COMPLETE,
                                               initCompleteHandler );
        aPreloader = preloader;
    }

    private function onLogin (event:LoginEvent):void
    {
        this.event = event;
        loggedIn = true;
        (loginWindow as ILoginWindow).deactivate();
        removeChild (loginWindow);
        if (isLoaded) {
            var anApp:Object = getApplication();
            passParamsToApp();
            (anApp as IFocusManagerContainer).focusManager.activate();
            dispatchEvent( new Event( Event.COMPLETE ) );
```

```
    }
    else {
        progress = new UnprotectedDownloadProgressBar ();
        progress.isLoaded = appInited;
        progress.minTime = MINIMUM_DISPLAY_TIME - getTimer() +
                                                    _displayTime;
        addChild (progress);

        progress.preloader = aPreloader;
        var xOffset:Number = Math.floor((progress.width -
                             progress.publicBorderRect.width) / 2);
        var yOffset:Number = Math.floor((progress.height -
                             progress.publicBorderRect.height) / 2);
        progress.x = (stage.stageWidth - progress.width) / 2 + xOffset;
        progress.y = (stage.stageHeight - progress.height) / 2 + yOffset;
        progress.addEventListener (Event.COMPLETE, onProgressComplete);
    }
}

private function onProgressComplete (event:Event):void{
    progress.removeEventListener (Event.COMPLETE, onProgressComplete);
    dispatchEvent( new Event( Event.COMPLETE ) );
}

private function initCompleteHandler(event:Event):void{
    appInited = true;
    var elapsedTime:int = getTimer() - _displayTime;

    if (elapsedTime < MINIMUM_DISPLAY_TIME) {
        var timer:Timer = new Timer(MINIMUM_DISPLAY_TIME - elapsedTime, 1);
        timer.addEventListener(TimerEvent.TIMER, flexInitComplete);
        timer.start();
    } else {
        flexInitComplete();
    }
}

private function flexInitComplete( event:Event = null ):void {
    isLoaded = true;
    if (progress) {
        removeChild (progress);
    }
    var anApp:Object = getApplication();
    if (loggedIn) {
        passParamsToApp();
        dispatchEvent( new Event( Event.COMPLETE ) );
    } else {
        (anApp as IFocusManagerContainer).focusManager.deactivate();
    }
}

private function passParamsToApp ():void{
    var anApp:Object = getApplication();
    for (var i:String in event) {
        try {
```

```
                anApp [i] = event [i];
            }
            catch (e:Error) {
                trace ("There is no parameter " + i +
                            "in " + anApp + " defined");
            }
        }
    }

    private function getApplication ():Object{
        return getDefinitionByName
                ("mx.core.Application").application;
    }
  }
}

import mx.preloaders.DownloadProgressBar;
import mx.graphics.RoundedRectangle;
import flash.display.Sprite;

class UnprotectedDownloadProgressBar extends DownloadProgressBar{
    public var isLoaded:Boolean = false;

    public function set minTime (value:int):void{
        if (value > 0) {
            MINIMUM_DISPLAY_TIME = value;
        }
    }

    public function get publicBorderRect ():RoundedRectangle{
        this.backgroundColor = 0xffffff;
        return borderRect;
    }

    public override function set preloader(value:Sprite):void{
        super.preloader = value;
        visible = true;
        if (isLoaded) {
            setProgress (100, 100);
            label = downloadingLabel;
        }
    }
}
```

As soon as the loading of the login window is complete it centers itself on the screen and starts listening to the `LoginEvent.ON_LOGIN`, which is dispatched by `Lightweight Preloader` when the XML with the user's credentials arrives from the server. This XML is nicely packaged inside the `LoginEvent` and saved on the local disk cache under the name `USER_INFO` (see the method `onLoginResult()` in Example 8-2).

Because the `LightweightPreloader` was added as a child of `LoginPreloader` (see `onLo ginLoaded()` in Example 8-8), the latter object will receive all events dispatched by the former.

Example 8-8. The onLoginLoaded() method of LoginPreloader

```
private function onLoginLoaded (event:Event):void{
    var content:Sprite = event.target.content as Sprite;
    addChild (content);
    loginWindow = content;
    (loginWindow as ILoginWindow).activate();
    content.x = (stage.stageWidth - content.width) / 2;
    content.y = (stage.stageHeight - content.height) / 2;
    content.addEventListener (LoginEvent.ON_LOGIN, onLogin);
}
```

This code also stores in `loginWindow` the reference to the login window, which is a subclass of `Sprite`, for reuse in case the user decides to log out. In that event, the login window should be brought back onto the screen; the function `activate()` just puts the focus there.

When the `ON_LOGIN` event arrives, the event handler `onLogin()` shown in Example 8-7 has to figure out whether the download of the main application has completed and if it's ready for use. If it is ready, the application gets activated; otherwise, an instance of the regular progress bar (`UnprotectedDownloadProgressBar`) is created and displayed until the application is ready.

The `Timer` object checks the progress of the download and dispatches the `Event.COMPLETE` to the application from the `flexInitComplete()` handler.

Supporting Logout Functionality

After a successful login, the user will see a screen like Figure 8-3. Besides supporting our custom preloader, the main application (Example 8-6) knows how to reuse the login component in the event that the user decides to log out at any time during the session by pressing the Log Out button.

Even though `LightweightPreloader` (the login component) was intended to be used as the very first visible component of our application, we want to be able reuse its functionality later on too. Hence, `LightweightPreloader` can be used either by the preloader or by a `PopupManager`. The following fragment from the main application does this job when the user clicks the Log Out button:

```
private function onLogout ():void{
    var loader:Loader = new Loader ();
    var url:URLRequest = new URLRequest ("LightweightPreloader.swf");
    var context:LoaderContext = new LoaderContext ();
    var applicationDomain:ApplicationDomain =
                              ApplicationDomain.currentDomain;
    context.applicationDomain = applicationDomain;
    loader.load (url, context);
    loader.contentLoaderInfo.addEventListener (Event.COMPLETE,
                                               onLoginLoaded);
}
```

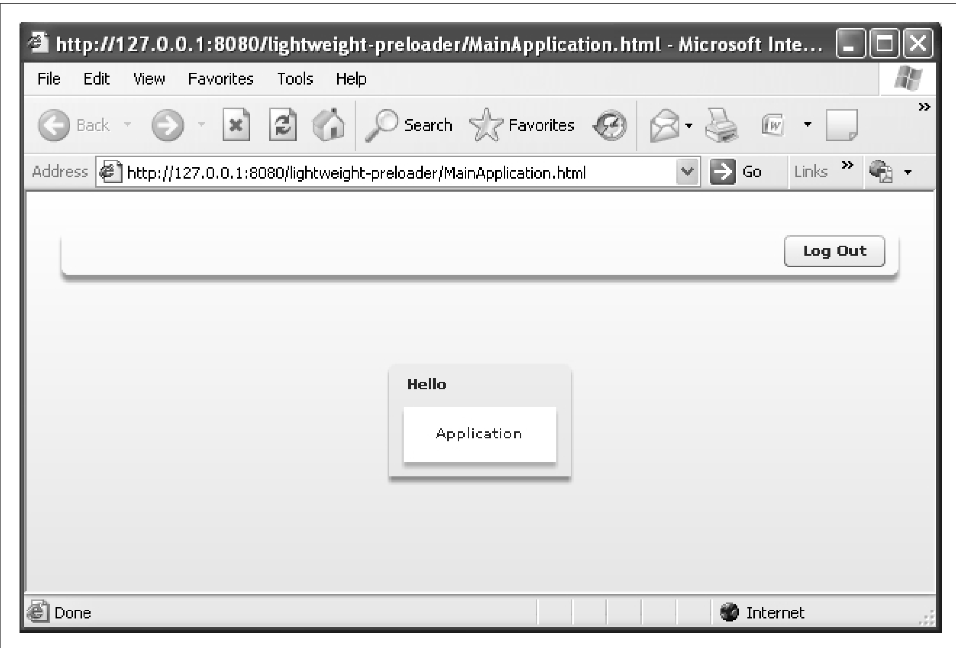
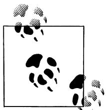


Figure 8-3. The application view after the user has logged in

```
private function onLoginLoaded (event:Event):void{
    content = event.target.content as Sprite;
    var component:UIComponent = new UIComponent ();
    loginPanel = new TitleWindow ();
    loginPanel.title = "Log In";

    component.addChild (content);
    loginPanel.addChild(component);

    PopUpManager.addPopUp(loginPanel, this, true);

    (content as ILoginWindow).activate();

    component.width = content.width;
    component.height = content.height;
    PopUpManager.centerPopUp(loginPanel);

    content.addEventListener (LoginEvent.ON_LOGIN, onLogin);
}
```



The call `PopupManager.addPopup()` is an example of how a Flex application can reuse a Flash component programmed to interact with Flex.

If you have Flash programmers in your team, you can use Flash for creating lightweight components when appropriate. Not only can you create lightweight login windows in Flash, but the entire main application view can be coded this way. As a matter of fact, all static views from your application that mostly contain artwork and don't pull data from the server can be made a lot slimmer if programmed as Flash components.

All communications between `LightweightPreloader`, `LoginPreloader`, and *MainApplication.mxml* are handled by dispatching and listening to the custom event `LoginEvent`, shown in Example 8-9.

Example 8-9. LoginEvent

```
package com.farata.preloading{
    import flash.events.Event;

    public dynamic class LoginEvent extends Event{
        public static const ON_LOGIN:String = "onLogin";

        public function LoginEvent(type:String, sessionID:String, xml:XML){
            super(type);
            this.sessionID = sessionID;
            this.loginXML = xml;
        }
    }
}
```

`LoginEvent` encapsulates the user's session ID (an application-specific session ID that's usually created on application startup and is used for maintaining state on the client) and the data received from the authentication server, represented as XML. This is the slightly nontraditional dynamic event described in the last section of Chapter 4.

The class `LoginPreloader` has a function that extracts the values of the parameters from the custom event and assigns them to the corresponding properties of the application object. If the application didn't have setters such as `sessionID` and `loginXML`, the code in Example 8-10 would throw an exception. If you use the dynamic `Application` object described in Chapter 2, on the other hand, such application properties aren't required. This is a typical situation for dynamically typed languages. The moral is, don't rely on the compiler; do better testing of your application.

Example 8-10. Non-object-oriented method of exchanging data between components

```
private function passParamsToApp (event:LoginEvent):void{
    var anApp:Object = getApplication();
    for (var i:String in event) {
        try{
            anApp [i] = event [i];
```

```
        }
        catch (e:Error) {
            trace ("There is no parameter " + i +
                                "in " + anApp + " defined");
            }
        }
    }
```

Example 8-9 and Example 8-10 use dynamic typing because it is appropriate in a special situation: when a Flash *.swf* file may have a bunch of properties in the event it dispatches. The `Application` object, however, may not need all these properties. The `for in` loop in Example 8-10 assigns only those dynamic properties that exist in the `Application` object.

Objects that use strongly typed properties perform better than dynamic ones. For a typical Flex way of exchanging data between components, implement the Mediator design pattern described in Chapter 2.

The sample application with `Preloader` not only demonstrates how to use pure Flash components in a Flex application for improving perceived performance, but also illustrates techniques of mixing and matching Flex and native Flash components.

Just to recap, the main application is written in Flex, the `LightweightPreloader` is a Flash component created in the Flash Professional IDE with some manual modifications of the generated ActionScript code, and the `LoginPreloader` is a manually written reusable ActionScript class that loads the *.swf* file with the Flash login component and removes it when the functionality of this *.swf* is no longer needed.

Using Runtime Shared Libraries

Using a tiny preloader *.swf* can give users the feeling that your application loads quickly, but you should also endeavor to make the main application load as fast as possible. A typical enterprise Flex RIA consists of several *.swf* files (the main application, fonts and styles, modules) as well as several *.swc* libraries (both yours and the Flex framework's). Your goal with these remains the same: ensure that only the minimum portion of the code travels over the network to the end user's machine.

How to Link Flex Libraries

Right-click on a project name in Flash Builder and select the Flex Build Path option, and you'll see a "Library path" panel similar to the one in Figure 8-4. This panel lists only the libraries from the Flex framework (AIR applications have some additional libraries). Both the framework and the necessary libraries must be linked to your project. You set the linkage method for the Flex framework via the "Framework link-

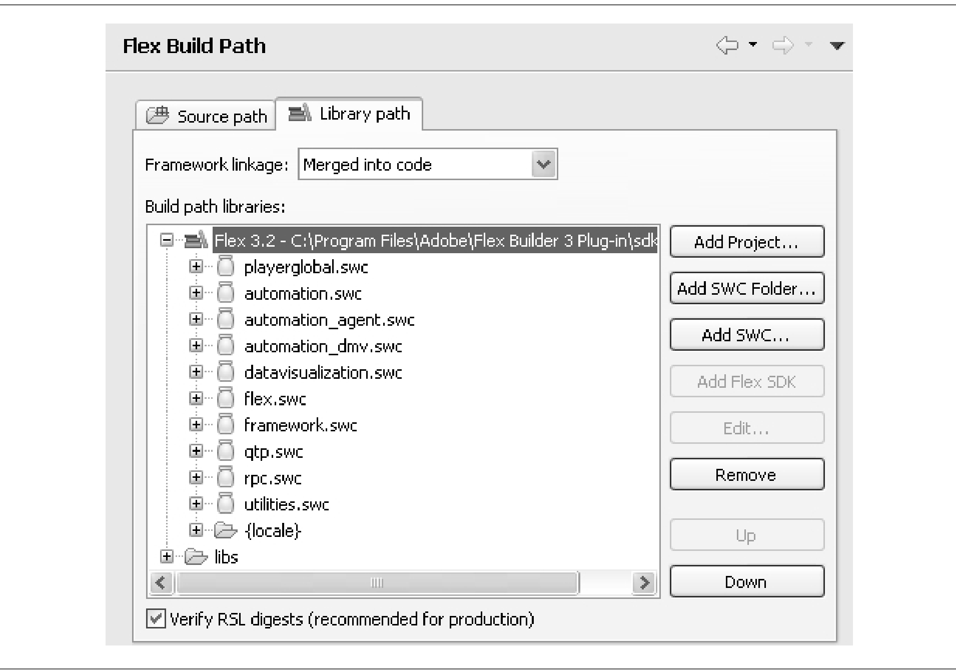


Figure 8-4. The library path of a simple Flex project

age" drop-down menu (more on this in the next section). For now, however, just concentrate on linking the Flex libraries that your project needs for successful compilation and execution. To do this, click on the plus sign by the library name (see Figure 8-4) and double-click on the link method. You can choose one of three linkage methods:

- RSLs
- Merged-in
- External

A typical enterprise application is composed of several Flash Builder projects. The main application must link in the libraries that are absolutely necessary to support the first screen. Optionally, it also can include some common libraries for modules that might be downloaded as a result of a user's interactions with your RIA. Loading common RSL libraries during application startup is not such a good idea, however, if you load modules in the application security domain and not their own subdomains (see Chapter 7). You need to manage your RSLs and ensure that each RSL is loaded only once. This can be done using the singleton `ModuleManager`, as you'll see in the section "Optimizing RSL Loading" on page 32.

Selecting the ***merge-in*** linkage for an application or a module increases the *.swf* size only by the size of the classes from the library that are actually mentioned in the *.swf* file. This requirement has a negative side effect for dynamically created (and, therefore,

not referenced in advance) objects. To have all objects available, you must declare a number of variables of each type that exists in the *.swc* file to ensure that all the classes that are needed (even for code that's loaded later) are included in the *.swf* file.

Chapter 7's "Bootstrapping Libraries as Applications" section described the process that happens once libraries are loaded. If the linker does not find explicit references to some classes from the linkage tree originated by the `Application` or `Module` class, it might omit both necessary supporting classes and not perform some parts of the necessary initialization process. If you are developing large data-driven dynamic applications, using bootstrapped libraries instead of modules is a safer and more reusable solution.

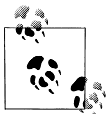
For example, if the code in your application never uses *`SomeGreatClass`* from the library *xyz.swc*, its code will not be included in the *.swf* file during compilation. Hence, if your business code "weighs" 300 KB and the *xyz.swc* is 100 KB, the compiled *.swf* file's size won't reach 400 KB unless each and every class from *xyz.swc* has been used. Merge-in linkage is justifiable only for small applications that won't use most of the framework classes.

Consider a RIA that consists of two Flash Builder projects: the main application (*proj1* at 250 KB) and a Flex module (*proj2* at 50 KB). Both of these projects use classes from the library *xyz.swc*. Chances are that *proj1* and *proj2* will need some of the same and some different classes from *xyz.swc*. What are your options here?

Of course, you can link *xyz.swc* using the merge-in option, in which case each project will include in its *.swf* file only those classes that are needed from *xyz.swc*. As you can guess, some amount of code duplication is unavoidable here: those classes that are needed in both projects will travel to the user's machine twice.

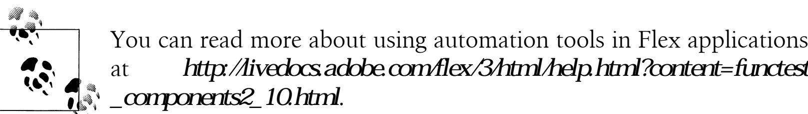
This may be acceptable for simple applications, but in an enterprise application with multiple *.swf* files you should consider a different approach. In *proj1*, specify that *xyz.swc* should be linked as an RSL. As a result, none of its classes will be included in the *.swf*, and the entire library (100 KB) will be downloaded before the `applicationComplete` event is even triggered. In this case, in *proj2* you can safely specify *external* as the linkage type for *xyz.swc*, because you know that by the time this project's *.swf* file is downloaded *xyz.swc* will already be there.

Now assume that the module from *proj2* is not immediately needed on application startup. If you use the RSL approach, the total size of the compiled code that has to exist on the user's machine is 250 KB +100 KB + the size of the Flex framework (500 KB or more). If the user initiates an action that requires the module from *proj2*, yet another 50 KB will be downloaded. (In the next section you'll learn a way to avoid repeatedly downloading 500 KB worth of Flex framework.)



Both the RSL and external linkage approaches imply that libraries will be available in the browser by the time an application or module needs them. The difference between the methods is that when you link a library as an RSL, the compiled *.swf* file contains a list of the libraries and Flash Player loads them. When you use external linkages, on the other hand, the compiled *.swf* doesn't contain mention of any external *.swc* files because it expects that another *.swf* has already loaded them. For more details, refer to the "Bootstrapping Libraries as Applications" section in Chapter 7 or search the Web for "IFlexModuleFactory interface."

As soon as a project is created, you should remove the default libraries that it doesn't need. For example, all libraries containing the automation API in general and *qtp.swc* (which enables support of the QTP testing tool from HP) in particular are not needed unless you are planning to run automated functional tests that record and replay user interactions with your RIA. Even if you are using the automation libraries during development, don't forget to remove them from the production build. Don't be tempted to rely on the libraries' merge-in linking option to limit the included classes; although the merge-in option includes only those objects that are used in the code when Flash Builder builds your project, its linker must still sift through all the libraries to make sure which are needed and which are not. (We'll discuss the linkage options in more detail a bit later.)



You can read more about using automation tools in Flex applications at *http://livedocs.adobe.com/flex/3/html/help.html?content=functest_components2_10.html*.

You'll also want to remove *datavisualization.swc* from the main application. In general, this library has to be linked on the module level. Your enterprise application should consist of a small shell that will be loading modules on an as-needed basis, and this shell definitely doesn't need to link *datavisualization.swc*. Consider as an example a shell application that loads 10 modules, 3 of which use Flex charting classes located in *datavisualization.swc*. In this scenario, you should link *datavisualization.swc* as an RSL. But, you may argue, if I do so and at some point all three charting modules need to be loaded, the data visualization RSL will be loaded three times! This would be correct, but you can get around it by using the optimized way of loading modules described in the "Optimizing RSL Loading" on page 32 section of this chapter.

The Flex Framework RSL

Before your application starts, `SystemManager` downloads (or loads from the local cache) all required RSL libraries and resource bundles (localization support) the application requires.

Choosing "Runtime shared library" from the "Framework linkage" drop-down on "Library path" panel (Figure 8-4) is simple and smart at the same time: deploy the Flex framework as separate from the business *.swf* library, and when the user first downloads the RIA Flash Player (Version 9.0.115 or above) will save the framework library in its own disk cache. It gets better, though; this library is designed to work across different domains, which means that users may get it from any site that was built in Flex and deployed with the Flex framework as an RSL, and not necessarily from *your* website.

These Flex SDK libraries are signed RSLs; their filenames end with *.swz*, and only Adobe can sign them. If you open the *rsls* directory in your Flex or Flash Builder's installation directory, you will find these signed libraries there. The path to the *rsls* directory may look like this:

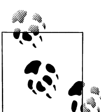
C:\Program Files\Adobe\Flex Builder 3 Plug-in\sdks\3.2.0\frameworks\rsls

At the time of this writing, the following RSLs are located there:

- *framework_3.2.0.3958.swz*
- *datavisualization_3.2.0.3958.swz*
- *rpc_3.2.0.3958.swz*

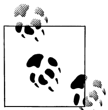
As you see, the filename includes the Flex SDK version number (in this case, 3.2.0) and the build number (3958).

We recommend that you build Flex applications under the assumption that your users will already have or will be forced to install a version of Flash Player not older than 9.0.115. If you can't do this for any reason, include pairs of libraries (*.swz* and corresponding *.swf*) in the build path, such as *rpc_3.2.0.3958.swz* and *rpc_3.2.0.3958.swf*. If the user has a player supporting signed libraries, the *.swz* file will be engaged. Otherwise, the unsigned fallback *.swf* library will be downloaded.



For a detailed description of how to use the Flex framework RSLs, read the Adobe documentation available at *http://livedocs.adobe.com/flex/3/html/help.html?content=rsl_09.html*.

At Farata Systems, we were involved in creating a website for an American branch of Mercedes Benz (*http://www.mbusa.com*). Examining this site with the web-monitoring tool Charles reveals which objects are downloaded to the user's machine.



While measuring the performance of a web application, you should use tools that clearly show you what the application in question is downloading. Charles does a great job of monitoring AMF, and we also like the Web Developer toolbar for the Mozilla browser, available at *http://chrispederick.com/work/web-developer/*. This excellent toolbar allows you with a click of a button to enable or disable the browser's cache, cookies, and pop-up blockers; validate CSS; inspect the DOM; and more.

In Figure 8-5 you can see that a number *.swf* files are being downloaded to the user's machine. (We took this screenshot on a PC with a freshly installed operating system just to ensure that no Flex applications that might have been deployed with signed framework RSLs were being run from this computer.) This website is a large and well-modularized RIA, and the initial download—the main window of the RIA, plus the required shareable libraries for the rest of the application—includes *.swf* files of approximately 162 KB, 95 KB, 52 KB, 165 KB, and 250 KB. The total download is about 730 KB, which is an excellent result for such a sophisticated RIA.

As you can see in Figure 8-5 though, there is one more library that is coming down the pipe: *framework_3.0.0.477.swz*. This Flex framework RSL is pretty heavy—525 KB—but the good news is that it's going to be downloaded only once, whenever the user of this PC runs into a Flex application deployed with this signed RSL.

Figure 8-6 shows what happens the second time we hit mbusa.com (*http://mbusa.com*), after clearing the web browser's cache. As you can see, the *.swf* files are still arriving, but the *.swz* file is not there any longer. The reason for this is that it was saved in the local Flash Player's cache on disk, which is not affected by clearing the web browser's cache—clearing the browser's cache removes the cached RSLs (*.swf* files), but not the signed ones (*.swz*).

The result is a substantial reduction in the initial size of this large RIA: from a total of 1.3 MB down to 730 KB.

For this reason, for large applications we recommend that you *always use signed framework RSLs*. Flex has become a popular development platform for RIAs, driving the adoption of the latest versions of Flash Player. The probability is high that within a year of their release, most cross-domain signed RSLs will exist on your clients' computers.

Starting from Flex 4, Adobe will officially host signed RSLs on its servers, which is an extra help for websites with limited network bandwidth. If you prefer, you won't even have to deploy the *.swz* files on your server. Unofficially, this feature exists even now: select and expand any library with the RSL linkage type, go to edit mode, and click the Add button (see Figure 8-7), and you'll be able to specify the URL where your *.swz* libraries are located.

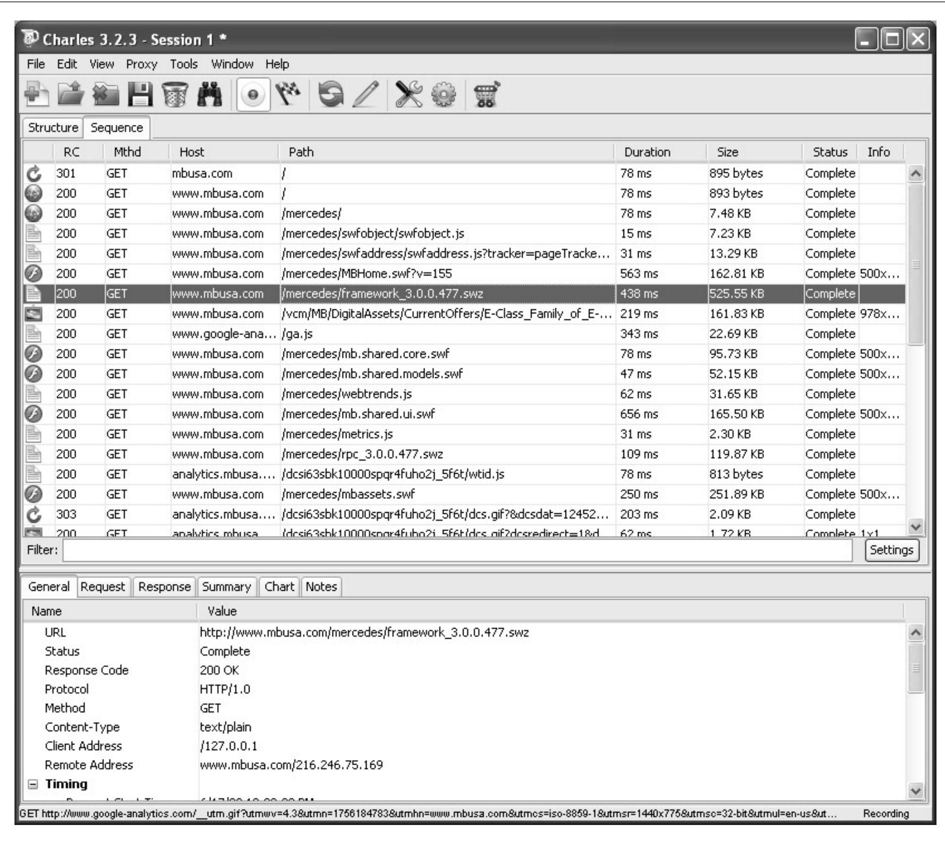


Figure 8-5. Visiting mbusa.com (http://mbusa.com) from a new PC

If the benefits of cached RSLs are so obvious, why not deploy every project with signed libraries? We see three reasons for this:

- There is a remote chance that the user will have a version of Flash Player older than release 9.0.115, the version where signed RSLs were introduced.
- The initial downloadable size of the Flex application is a bit larger if it's deployed with RSLs rather than the merge-in option. At the time of this writing no statistics have been published regarding the penetration of the signed Flex RSLs, and if you wrongly assume that your users will have cached RSLs on their machines, you may find that using the merge-in option would have resulted in a smaller download (say, one 1.1 MB *.swf* as opposed to two files totaling 1.3 MB for a virgin machine). In consumer applications, each 100 KB reduction matters.
- When the merge-in option is used the client's web browser doesn't need to make an extra network call to load the *.swz*; the entire code is located in one *.swf*.

To address these valid concerns, you can:

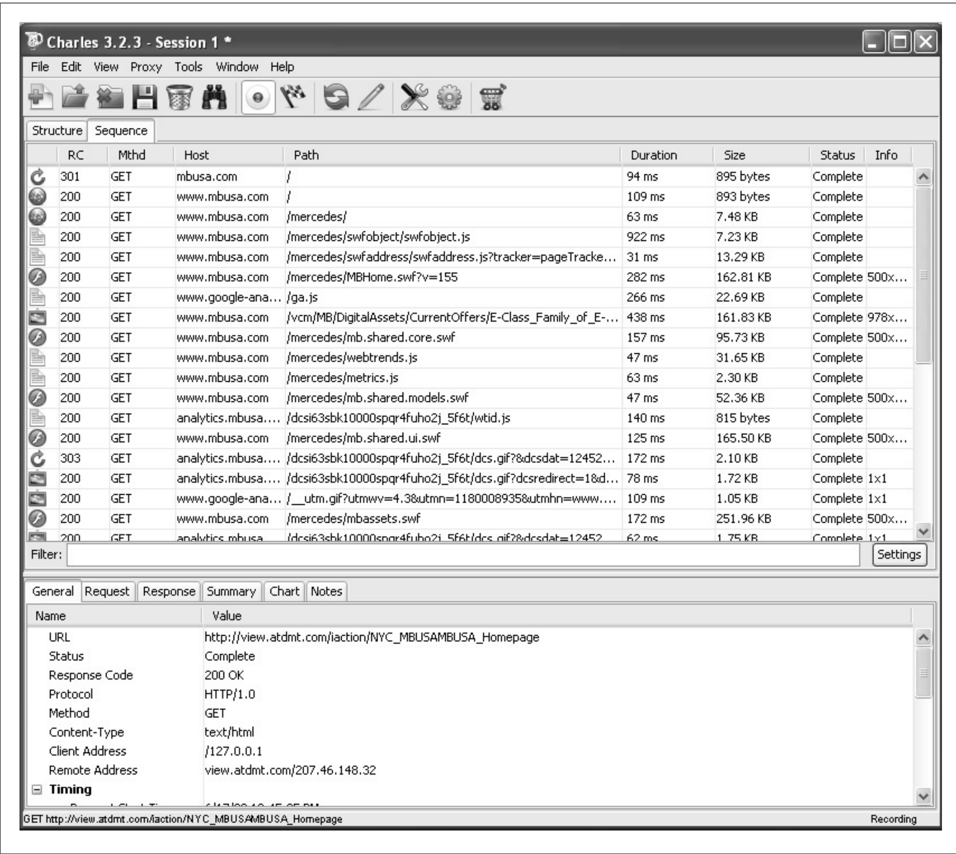


Figure 8-6. Visiting mbusa.com (http://mbusa.com) after the framework RSL has been cached

- Force users to upgrade to the later version of the player, if you're working in a controlled environment. For users that can't upgrade the player, provide fall-back *.swf* files.
- Repackage your RSLs for distribution, including only those classes your application needs. This technique is described in James Ward's blog at *http://www.jamesward.com/blog/2007/02/19/faster-flex-applications-shrink-your-rsls/*.
- Intervene in the load process (keep in mind that the Flex framework is open source and all initialization routines can be studied and modified to your liking).

If you have the luxury of building a new enterprise RIA from scratch rather than trying to fit *.swf* files into an existing HTML/JavaScript website, we recommend that you get into a "portal state of mind." In no time your RIA will grow and begin demanding more and more new features, modules, and functionality. Why not expect this from the get-go? Assume that the application you are about to develop will grow into a portal in a couple of years, and plan ahead by creating a lightweight shell-like Flex application

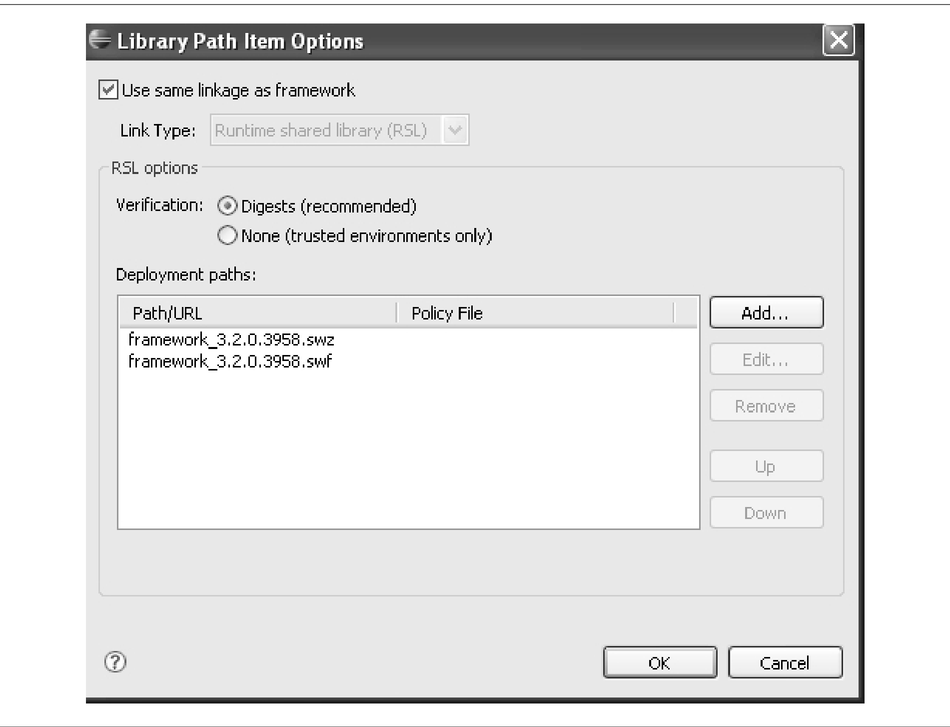


Figure 8-7. Specifying location of RSL libraries

that loads the rest of the modules dynamically. If you start with such a premise, you'll naturally think of the shared resources that have to be placed into RSLs and the rest of the business functionality will be developed as modules and reusable components.

Optimizing Library Linkages with FX2Ant

Deploying a multiproject Flex RIA is yet another step that should be optimized for performance. Flash Builder's Export release build option, however, is not applicable for enterprise applications, which are deployed into production by running a set of scripts from a command line.

Farata's FX2Ant utility (see Chapter 4) is known in the Flex community as a tool for automation of writing Ant build scripts for Flex projects. But there is yet another benefit to using FX2Ant rather than manually writing build scripts: it optimizes the linkage parameters of RSL libraries for multiproject applications.

Another section of the generated Ant scripts removes debug and metadata information from the *.swf*. FX2Ant also optimizes the cases when the modules are using the RSLs. In the build scripts, the module's RSL linkage will be replaced with the external type as the RSLs are guaranteed to be loaded by the main application, thus reducing the number of server calls during the module load procedure. On average, using FX2Ant

scripts reduces the size of generated modules by 10–25%, even compared with release builds produced by Flash Builder. For example, the following code snippet generated by FX2Ant optimizes the size of *all* resources and removes unnecessary metadata (items that are not listed in the `--keep-as3-metadata` option):

```
<unzip src="${DOCUMENTS}/PortalLib/bin/PortalLib.swc"
dest="${build.dir}">
<patternset>
<include name="library.swf"/>
</patternset>
</unzip>
<java jar="${flex.sdk.dir}/lib/optimizer.jar" fork="true"
failonerror="true">
<jvmarg line="-ea -DAS3 -DAVMPLUS -
Dflexlib='${flex.sdk.dir}/frameworks' -Xms32m -Xmx384m -
Dsun.io.useCanonCaches=false"/>
<arg line="'${build.dir}/library.swf' --output
'${build.dir}/PortalLib.swf' --keep-as3-
metadata='Bindable,Managed,ChangeEvent,NonCommittingChangeEv
ent,Transient' "/>
</java>
<delete file="${build.dir}/library.swf"/>
```

Consider the example of a RIA that consists of two Flash Builder projects, in each of which the developer has specified the library *xyz.swc* with a link type of *RSL*. The script generated by FX2Ant will keep RSL as the linkage type for the *xyz.swc* in the main project but will change the linkage for this library to *external* in the second one.

You might try shaving off another 10–20% of the unused framework code by repackaging the framework *.swc* files to keep only those used in your application and modules, but for large applications it's seldom worth the effort.

Optimizing RSL Loading

Optimizing the loading of RSLs is an important step in optimizing your projects. Consider an application with 10 modules, 3 of which use *datavisualization.swc* as an RSL. To avoid redundant loading, you'll want to make sure that `SystemManager`—the keystone of any Flex application, which gets engaged by the end of the very first application frame and immediately starts loading RSLs—behaves as a singleton.

The sample application that we'll study in this section is an improved version of the project from Chapter 7's "Sample Flex Portal" section. This section's source code includes the following Flash Builder projects: *OptimizedPortal*, *FeedModule*, *ChartsModule*, and *PortalLib*.

Creating Modules with Test Harness

Once again, here's our main principle of building enterprise Flex applications: you should aim to create a lightweight shell application that loads modules when they're needed. This approach leads to the creation of modularized and better-performing

RIAs. To be productive, though, developers working on particular modules need to be able to quickly perform unit and functional tests on their modules, without depending too much on the modules their teammates are working on.

The *FeedsModule* project is an Eclipse Dynamic Web Project with its own "server-side" *WebContent* directory. This project also includes a very simple application, *TestHarness.mxml*, that includes two modules: `GoogleFinancialNews` and `YahooFinancialNews`. Mary is responsible for development of these two modules, which will later become part of a larger *OptimizedPortal*. Whereas in Chapter 7 our focus was on creating a portal for integrating various applications, here we are building it as a shell for hosting multiple modules.

To avoid issues caused by merging module and application stylesheets, we recommend having only one CSS file at the application level. This may also save you some grief trying to figure out why modules are not being fully unloaded as the description of the `unload()` function promises; merged cascading style sheets may create strong references that won't allow the garbage collector to reclaim memory on module unloads.

You should also avoid linking into the module's byte code the information from the *services-config.xml* file that comes with BlazeDS/LCDS. If you specify separate *services-config.xml* files in the compiler option of the module's project, the contents of those files (configured destinations and channels) will get sucked into the compiled *.swf*.

In our team, all developers must submit their modules fully tested and with minimal efforts made to configure the portal environment. Example 8-11 lists the application that Mary uses for testing, while Figure 8-8 shows the results.

Example 8-11. TestHarness.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical">
    <mx:Label text="google"/>
    <mx:ModuleLoader id="mod1"
            creationComplete="mod1.loadModule('GoogleFinancialNews.swf')"
            width="800" height="300"
            applicationDomain="{ApplicationDomain.currentDomain}"
            ready="mod2.loadModule('YahooFinancialNews.swf')"/>
    <mx:Label text="yahoo"/>
    <mx:ModuleLoader  id="mod2" width="800" height="300"
            applicationDomain="{ApplicationDomain.currentDomain}"/>
</mx:Application>
```

Each of the modules in *TestHarness* has the ability to load yet another module: `ChartModule`. This is done by switching to the view state `ChartState` and calling the function `loadChartSWF()`. Example 8-12 depicts the code of the module `YahooFinancialNews`.

Example 8-12. The module YahooFinancialNews

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
       horizontalGap="0" verticalGap="0"  width="100%" height="100%"
```

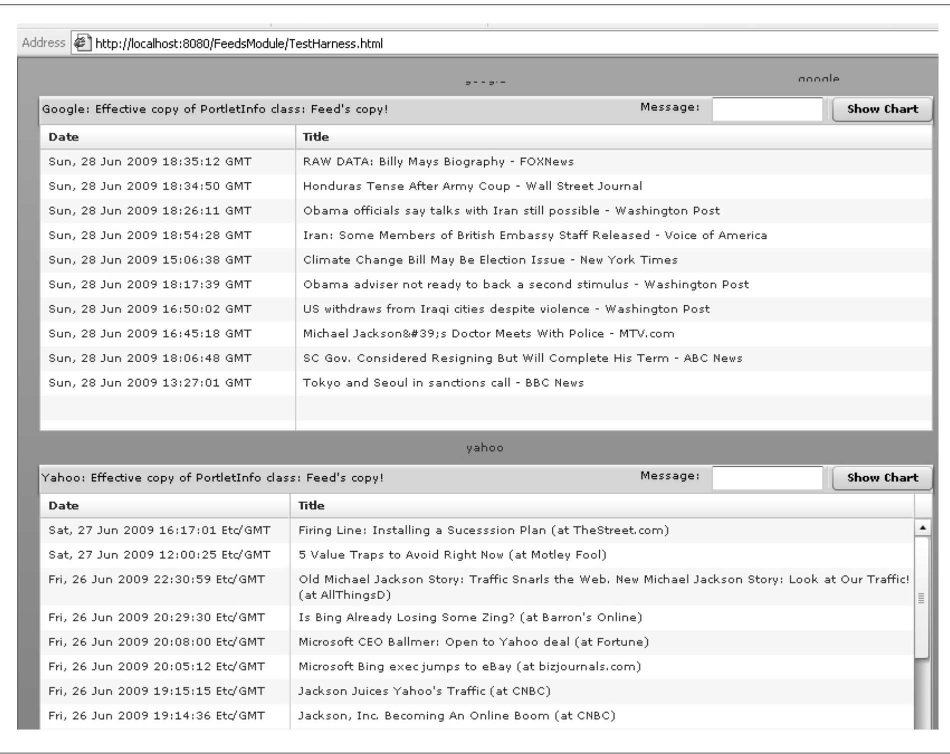


Figure 8-8. Running TestHarness.mxml

```
        paddingBottom="0" paddingLeft="0" paddingRight="0" paddingTop="0"
        backgroundColor="white"
    >

    <mx:states>
        <mx:State name="ChartState">
            <mx:RemoveChild target="{newsGrid}"/>
            <mx:AddChild relativeTo="{header}" position="after">
                <mx:ModuleLoader id="chart_swf"
                        applicationDomain="{ApplicationDomain.currentDomain}"
                        creationComplete="loadChartSWF()" width="100%"
                        height="100%"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:HBox id="header"
            width="100%" height="25"
            backgroundColor="#ffffff" backgroundAlpha="0.8"
            verticalAlign="middle" color="black">

        <mx:Label htmlText="Yahoo: Effective copy of PortletInfo class:
                {PortletInfo.INFO}"/>
        <mx:HBox width="100%" horizontalAlign="right" horizontalGap="3"
```

```
            verticalGap="0">
        <mx:Label text="Message: "/>
        <mx:TextInput id="textInput" text="{_messageText}" width="100"/>
        <mx:VRule height="20"/>
        <mx:Button label="{currentState == 'ChartState' ? 'Show Feed' :
                'Show Chart'}" click="currentState = (currentState ==
                'ChartState' ? '' : 'ChartState')"/>
    </mx:HBox>
    <mx:filters>
        <mx:DropShadowFilter angle="90" distance="2"/>
    </mx:filters>
</mx:HBox>

<mx:DataGrid id="newsGrid" width="100%" height="100%"
        dataProvider="{newsFeed.lastResult.channel.item}"
        variableRowHeight="true"
        dragEnabled="true" creationComplete="onCreationComplete()">
    <mx:columns>
        <mx:Array>
            <mx:DataGridColumn headerText="Date" dataField="pubDate"
                    width="80"/>
            <mx:DataGridColumn headerText="Title" dataField="title"
                    wordWrap="true" width="200"/>
        </mx:Array>
    </mx:columns>
</mx:DataGrid>

<mx:HTTPService id="newsFeed" useProxy="true"
        destination="YahooFinancialNews" concurrency="last"
        resultFormat="e4x" fault="onFault(event)"/>

<mx:Script>
    <![CDATA[
    import mx.managers.PopUpManager;
    import com.farata.portal.Message;
    import com.farata.portal.events.BroadcastMessageEvent;
    import mx.controls.Alert;
    import mx.rpc.events.*;

    [Bindable]
    private var _messageText:String;

    private function onCreationComplete():void {
        var bridge:IEventDispatcher = systemManager.loaderInfo.sharedEvents;
        bridge.addEventListener(
                BroadcastMessageEvent.BROADCAST_MESSAGE_TO_PORTLETS,
                messageBroadcasted );
        newsFeed.send({s:"YHOO"});
    }
    private function loadChartSWF():void{
        chart_swf.loadModule("/ChartsModule/ChartModule.swf");
    }

    private function messageBroadcasted( event:Event ):void{
        var newEvent:BroadcastMessageEvent =
```

Figure 8-9. Switching to chart view

```
                BroadcastMessageEvent.unmarshal( event );
            var message:Message = newEvent.message;
            _messageText = message.messageBody;
        }

        private function onFault(event:FaultEvent):void {
            Alert.show( event.toString() );
            mx.controls.Alert.show(
                    "Destination:" + event.currentTarget.destination + "\n" +
                    "Fault code:" + event.fault.faultCode + "\n" +
                    "Detail:" + event.fault.faultDetail, "News feed failure"
            );
        }
        ]]>
    </mx:Script>
</mx:Module>
```

Click the application's Show Chart button to make sure that loading one module from the other works fine and that they properly pick the destination from the main application's *services-config.xml*. Figure 8-9 shows the expected result.

Because you want to have a test harness that will allow you to run and test these modules outside of the main portal, we'll do a trick that will link the *TestHarness* application

with the one and only *services-config.xml* of the main portal project. Example 8-13 lists the file named *TestHarness-config.xml* located in the *FeedsModule* project.

Example 8-13. TestHarness-config.xml

```
<flex-config>
  <compiler>
    <services>
      c:/farata/oreilly/OptimizedPortal/WebContent/WEB-INF/flex/services-config.xml
    </services>
  </compiler>
</flex-config>
```

The very fact that a project has a file with the same name as the main application but with the suffix *-config* will make the Flex compiler use it as configuration file that redirects to the real *services-config.xml*. (Remember, you need to replace `c:/farata/oreilly` with the actual location of the workspace of the *OptimizedPortal* project.)

Open the class *TextHarness_FlexInit_generated.as* in the generated folder of the *FeedModule* project, and you'll see a section taken from the portal project. A fragment of this section is shown here:

```
ServerConfig.xml =
<services>
    <service id="remoting-service">
        <destination id="AnnualGenerator">
            <channels>
                <channel ref="my-amf"/>
            </channels>
        </destination>
        <destination id="QuoterDataGenerator">
            <channels>
                <channel ref="my-amf"/>
            </channels>
        </destination>
    </service>
    ...
    <channels>
        <channel id="my-rtmp" type="mx.messaging.channels.RTMPChannel">
            <endpoint uri="rtmp://{server.name}:58010"/>
            <properties>
            </properties>
        </channel>
     </channels>
</services>;
```

Essentially, here's what's happening: while building the *FeedsModule* project, the Flex compiler determines that it has two modules and one application and that it, therefore, must build three *.swf* files. It checks if `TestHarness`, `GoogleFinancialNews`, and `YahooFinancialNews` have their own configuration files. `TestHarness` has one, so the compiler uses it in addition to *flex-config.xml* from Flex SDK. `GoogleFinancialNews` and `YahooFinancialNews` do not have their own configuration files, so for them the compiler just uses the parameters listed in *flex-config.xml*.

What did we achieve here? We've created a small project that can be used for testing and debugging the modules without the need to have its own server-side configuration filefile configuration (*services-config.xml*). If you have worked on a large modularized Flex application, chances are that you've run into conflicts caused by destinations having the same names but pointing to different classes—such classes were likely created by different programmers and are located in multiple modules' *services-config.xml* files. With this approach, you won't run into such situations.

In the next section, you'll learn how to make your modules go easy on network bandwidth.

Creating a Shell Application with a Custom RSL Loader

Mary, the application developer, knows how to test her modules, and she'd really appreciate it if she didn't have to coordinate her testing with other developers who might link the same RSLs to their modules. Is it possible to make the application a little smarter, so it won't load a particular RSL with the second module if it's already downloaded it with the first one?

To avoid duplication in modules, the Flex framework offers a singleton class called `ModuleManager` (see Chapter 7). However, it falls short when it comes to RSLs. Luckily, the Flex framework is open source, and we'll show you how to fix this shortcoming yourself. First, let's take a closer look at the problem.

As you'll remember, the singleton `SystemManager` is the starting class that controls the loading of the rest of the application's objects. Our sample application is a portal located in the Flash Builder project *OptimizedPortal*. Adding the compiler option `-keep` allows you to see the generated ActionScript code for the project. The main point of interest is the class declaration in the file *_OptimizedPortal_mx_managers_SystemManager-generated.as*, located in the *generated* folder (Example 8-14).

Example 8-14. Generated system manager for OptimizedPortal

```
package{

import flash.text.Font;
import flash.text.TextFormat;
import flash.system.ApplicationDomain;
import flash.utils.getDefinitionByName;
import mx.core.IFlexModule;
import mx.core.IFlexModuleFactory;

import mx.managers.SystemManager;

[ResourceBundle("collections")]
[ResourceBundle("containers")]
[ResourceBundle("controls")]
[ResourceBundle("core")]
[ResourceBundle("effects")]
[ResourceBundle("logging")]
```

```
[ResourceBundle("messaging")]
[ResourceBundle("skins")]
[ResourceBundle("styles")]
public class _OptimizedPortal_mx_managers_SystemManager
    extends mx.managers.SystemManager
    implements IFlexModuleFactory{
    // Cause the CrossDomainRSLItem class to be linked into this application.
    import mx.core.CrossDomainRSLItem; CrossDomainRSLItem;

    public function _OptimizedPortal_mx_managers_SystemManager(){
        super();
    }

    override public function create(... params):Object{
        if (params.length > 0 && !(params[0] is String))
            return super.create.apply(this, params);

        var mainClassName:String = params.length == 0 ?
                        "OptimizedPortal" : String(params[0]);
        var mainClass:Class = Class(getDefinitionByName(mainClassName));
        if (!mainClass)
            return null;

        var instance:Object = new mainClass();
        if (instance is IFlexModule)
            (IFlexModule(instance)).moduleFactory = this;
        return instance;
    }

    override public function info():Object{
        return {
            cdRsls: [{"rsls":["datavisualization_3.3.0.4852.swz"],
"policyFiles":[""]
,"digests":["6557145de8b1b668bc50fd0350f191ac33e0c33d9402db900159c51a02c62ed6"],
"types":["SHA-256"],
"isSigned":[true]
},
{"rsls":["framework_3.2.0.3958.swz","framework_3.2.0.3958.swf"],
"policyFiles":["",""]
,"digests":["1c04c61346a1fa3139a37d860ed92632aa13decf4c17903367141677aac966f4","1c0
4c61346a1fa3139a37d860ed92632aa13decf4c17903367141677aac966f4"],
"types":["SHA-256","SHA-256"],
"isSigned":[true,false]
},
{"rsls":["rpc_3.3.0.4852.swz"],
"policyFiles":[""]
,"digests":["f7536ef0d78a77b889eebe98bf96ba5321a1fde00fa0fd8cd6ee099befb1b159"],
"types":["SHA-256"],
"isSigned":[true]
}]
,
            compiledLocales: [ "en_US" ],
            compiledResourceBundleNames: [ "collections", "containers", "controls",
"core", "effects", "logging", "messaging", "skins", "styles" ],
            currentDomain: ApplicationDomain.currentDomain,
```

```
            layout: "vertical",
            mainClassName: "OptimizedPortal",
            mixins: [ "_OptimizedPortal_FlexInit",
"_richTextEditorTextAreaStyleStyle", "_ControlBarStyle", "_alertButtonStyleStyle",
"_SWFLoaderStyle", "_textAreaVScrollBarStyleStyle", "_headerDateTextStyle",
"_globalStyle", "_ListBaseStyle", "_HorizontalListStyle", "_todayStyleStyle",
"_windowStylesStyle", "_ApplicationStyle", "_ToolTipStyle", "_CursorManagerStyle",
"_opaquePanelStyle", "_TextInputStyle", "_errorTipStyle", "_dateFieldPopupStyle",
"_dataGridStylesStyle", "_popUpMenuStyle", "_headerDragProxyStyleStyle",
"_activeTabStyleStyle", "_PanelStyle", "_DragManagerStyle", "_ContainerStyle",
"_windowStatusStyle", "_ScrollBarStyle", "_swatchPanelTextFieldStyle",
"_textAreaHScrollBarStyleStyle", "_plainStyle", "_activeButtonStyleStyle",
"_advancedDataGridStylesStyle", "_comboDropdownStyle", "_ButtonStyle",
"_weekDayStyleStyle", "_linkButtonStyleStyle",
"_com_farata_portal_PortalCanvasWatcherSetupUtil",
"_com_farata_portal_controls_SendMessageWatcherSetupUtil" ],
            rsls: [{url: "flex.swf", size: -1}, {url: "utilities.swf", size: -1},
{url: "fds.swf", size: -1}, {url: "PortalLib.swf", size: -1}]
        }
    }
}

}
```

Skim through the code in Example 8-14, and you'll see that all the information required by the linker is there. The Flex code generator has created a system manager for the *OptimizedPortal* application that's directly inherited from `mx.managers.SystemManager`, which doesn't give you any hooks for injecting the new functionality in a kosher way: whatever you put in the class will be removed by the code generators during the next compilation. The good news is that the Flex SDK is open source, and you are allowed to do surgery on its code (and even submit your changes to be considered for inclusion in upcoming releases of Flex).

The goal here is to change the behavior of the `SystemManager` so it won't load duplicate instances of the same RSL if more than one module links to it. (Remember *datavisualization.swc*, used in 3 out of 10 modules?)

Scalpel, please!

The *flex_src* directory of the project *OptimizedPortal* includes a package called *mx.core*, which includes two classes: `RSLItem` and `RSLListLoader`. These are the classes from the Adobe Flex SDK that we altered. The class `RSLListLoader` sequentially loads all required libraries. The relevant fragment of this class is shown in Example 8-15.

Example 8-15. Modified Flex SDK class RSLListLoader

```
////////////////////////////////////////////////////////////////////////////////
//
//  ADOBE SYSTEMS INCORPORATED
//  Copyright 2007 Adobe Systems Incorporated
//  All Rights Reserved.
//
//  NOTICE: Adobe permits you to use, modify, and distribute this file
```

```
//  in accordance with the terms of the license agreement accompanying it.
//
////////////////////////////////////////////////////////////////////////////////

package mx.core{

import flash.events.IEventDispatcher;
import flash.events.Event;
import flash.utils.Dictionary;

[ExcludeClass]

/**
 *  @private
 *  Utility class for loading a list of RSLs.
 *
 *  <p>A list of cross-domain RSLs and a list of regular RSLs
 *  can be loaded using this utility.</p>
 */

public class RSLListLoader{
/**
    *  Constructor.
    *
    *  @param rslList Array of RSLs to load.
    *  Each entry in the array is of type RSLItem or CdRSLItem.
    *  The RSLs will be loaded from index 0 to the end of the array.
    */
    public function RSLListLoader(rslList:Array)
    {
        super();
        this.rslList = rslList;
    }

    /**
     *  @private
     *  The index of the RSL being loaded.
     */
    private var currentIndex:int = 0;

    public static var  loadedRSLs:Dictionary = new Dictionary();
    /**
     *  @private
     *  The list of RSLs to load.
     *  Each entry is of type RSLNode or CdRSLNode.
     */
    private var rslList:Array = [];

    ...
    /**
     *  Increments the current index and loads the next RSL.
     */
    private function loadNext():void{
        if (!isDone()){
            currentIndex++;
```

```
            // Load the current RSL
            if (currentIndex < rslList.length){
                // Skip the loading process if the RSL has been loaded already
                if (loadedRSLs[(rslList[currentIndex] as RSLItem).url] == null){
                    rslList[currentIndex].load(chainedProgressHandler,
                            listCompleteHandler, listIOErrorHandler,
                            listSecurityErrorHandler, chainedRSLErrorHandler);
                    loadedRSLs[(rslList[currentIndex] as RSLItem).url] = true;
                } else {
                    loadNext(); // Skip already loaded RSLs
                }
            }
        }
    }
    ...
}
}
```

Example 8-15 adds only a few lines to the class:

```
public static var  loadedRSLs:Dictionary = new Dictionary();
...
if (loadedRSLs[(rslList[currentIndex] as RSLItem).url] == null){
...
loadedRSLs[(rslList[currentIndex] as RSLItem).url] = true;
...
loadNext(); // Skip already loaded RSLs
```

but these have a dramatic effect on the RSL loading process.

The `static` dictionary `loadedRSL` keeps track of already loaded RSLs (the `url` property of the `RSLItem`), and if a particular RSL that's about to be loaded already exists there, the `RSLListLoader` doesn't bother loading it. This prevents loading of duplicate RSLs and will substantially reduce the download time of some enterprise Flex applications.

In the class `RSLItem`, we've changed the access level of the property `url` from `protected` to `public`:

```
public var url:String; // PATCHED - was protected
```

We recommend that you avoid using the keyword `protected`. For more details, read the blog post at *http://tinyurl.com/m6sp32*.

Because the source code of our versions of `RSLItem` and `RSLListLoader` is included in the project, they will be merged into the *.swf* file and have precedence over the original classes with the same names provided in the Flex SDK libraries. "Flex open sourcing in action!" could have made a nice subtitle for this section. The very fact that the Flex SDK is open source gives us a chance to improve its functionality in any enterprise application, and even to submit proposals for changes to Adobe.

Moving on, let's run the *OptimizedPortal* application that uses our modified `RSLLoader` class. The code for this application is presented in Example 8-16, but we won't give a detailed explanation of this code because in the context of this chapter what's happening under the hood when the modules are being loaded is more important.

Example 8-16. OptimizedPortal.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application layout="vertical"
        xmlns:mx="http://www.adobe.com/2006/mxml"
        xmlns:fx="http://www.faratasystems.com/2009/portal" >

    <mx:Style source="styles.css"/>
    <mx:Button id="b" label="Add module" click="addModule()"/>
    <mx:Script>
        <![CDATA[
        import mx.core.UIComponent;
        import mx.modules.Module;
        import mx.events.ModuleEvent;
        import mx.modules.ModuleManager;
        import mx.modules.IModuleInfo;

        private var _moduleInfo:IModuleInfo;

        private function addModule() : void {
            // create the module - note, we're not loading it yet
            moduleInfo =
                    ModuleManager.getModule("/FeedsModule/YahooFinancialNews.swf");
            // add some listeners
            _moduleInfo.addEventListener(ModuleEvent.READY, onModuleReady, false,
                                                                    0, true);
            _moduleInfo.addEventListener(ModuleEvent.SETUP, onModuleSetup, false,
                                                                    0, true);
            _moduleInfo.addEventListener(ModuleEvent.UNLOAD, onModuleUnload, false,
                                                                    0, true);
           _moduleInfo.addEventListener(ModuleEvent.PROGRESS, onModuleProgress,
                                                                    false, 0, true);
           // load the module
           _moduleInfo.load();
        }

        /**
         * The handlers for the module loading events
         **/
        private function onModuleProgress (e:ModuleEvent) : void {
            trace("ModuleEvent.PROGRESS received: " + e.bytesLoaded + " of " +
                    e.bytesTotal + " loaded.");
        }

        private function onModuleSetup (e:ModuleEvent) : void {
            trace("ModuleEvent.SETUP received");
            // cast the currentTarget
            var moduleInfo:IModuleInfo = e.currentTarget as IModuleInfo;
```

```
            trace("Calling IModuleInfo.factory.info ()");
            // grab the info and display information about it
            var info:Object = moduleInfo.factory.info();
            for (var each:String in info) {
                trace("     " + each + " = " + info[each]);
            }
        }

        private function onModuleReady (e:ModuleEvent):void {
            trace("ModuleEvent.READY received");
            // cast the currentTarget
            var moduleInfo:IModuleInfo = e.currentTarget as IModuleInfo;
            // add an instance of the module's class to the display list
            trace("Calling IModuleInfo.factory.create ()");
            this.addChild( moduleInfo.factory.create () as UIComponent);
            trace("module instance created and added to Display List");
        }
        private function onModuleUnload (e:ModuleEvent) : void {
            trace("ModuleEvent.UNLOAD received");
        }
        ]]>

    </mx:Script>
    <fx:PortalCanvas width="100%" height="100%">
        <fx:navItems>
            <fx:NavigationItem>
                <fx:PortletConfig title="Complete Application" isModule="true"
                        preferredHeight="400" preferredWidth="850">
                    <fx:props>
                        <mx:Object
                                url="/FeedsModule/GoogleFinancialNews.swf"/>
                    </fx:props>
                </fx:PortletConfig>
            </fx:NavigationItem>
            <fx:NavigationItem>
                <fx:PortletConfig title="Just a Module" isModule="true"
                        preferredHeight="400" preferredWidth="850">
                    <fx:props>
                        <mx:Object
                                url="/FeedsModule/YahooFinancialNews.swf"/>
                    </fx:props>
                </fx:PortletConfig>
            </fx:NavigationItem>
        </fx:navItems>
    </fx:PortalCanvas>
</mx:Application>
```

Example 8-16 uses a number of tags from our library *PortalLib*, which is linked to the *OptimizedPortal* project (its source code comes with this chapter's sample code). Here are some very brief descriptions of these components:

`PortletConfig`
: A bunch of public variables: `portletId`, `title`, `preferredWidth`, `showMaximized`, `isSingleton`, `props`, and `content` (which is a `DisplayObject`)

Figure 8-10. The main view of the OptimizedPortal application

`NavItem`
: A component with a getter and setter for a label, a tooltip, an icon, and an associated portlet of type `PortletConfig`

`PortletConfig`
: Describes the future portlet window

`NavIgationItem`
: Describes an icon on the portal desktop that can be clicked to create an instance of that window

For the next experiment, we'll clear the browser's cache and start Charles to monitor the loading process.

Flash Builder has a known issue: it sorts the libraries in the project's build path into alphabetical order, which may produce hard-to-explain runtime errors in some cases. Thus, before running the *OptimizedPortal* application Flash Builder opens its project build path and ensures that *datavisualization.swc* is listed after *utilities.swc*; otherwise, you might see an error about `TweenEffect`.

Running the *OptimizedPortal* application displays the main view shown in Figure 8-10 *really fast*, which is one of the most important goals of any RIA.

In Figure 8-11, Charles shows what *.swf* files have been loaded so far: *OptimizedPortal.swf*, *flex.swf*, *utilities.swf*, *fds.swf*, and *PortalLib.swf*.

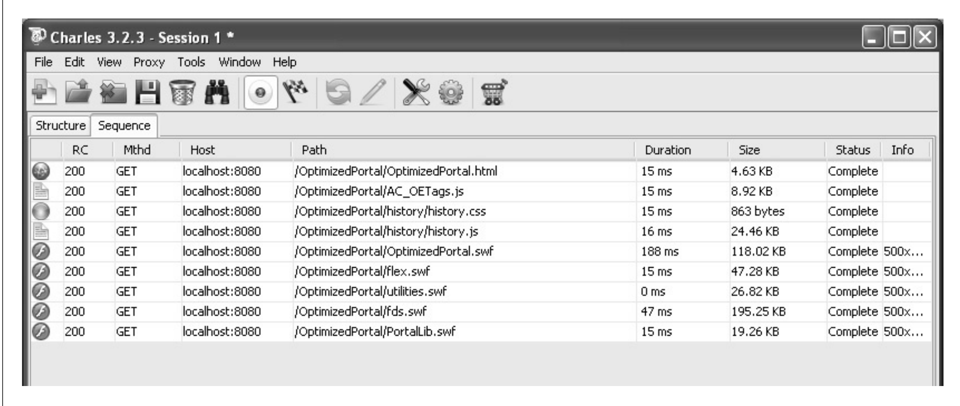


Figure 8-11. Charles shows initial downloads

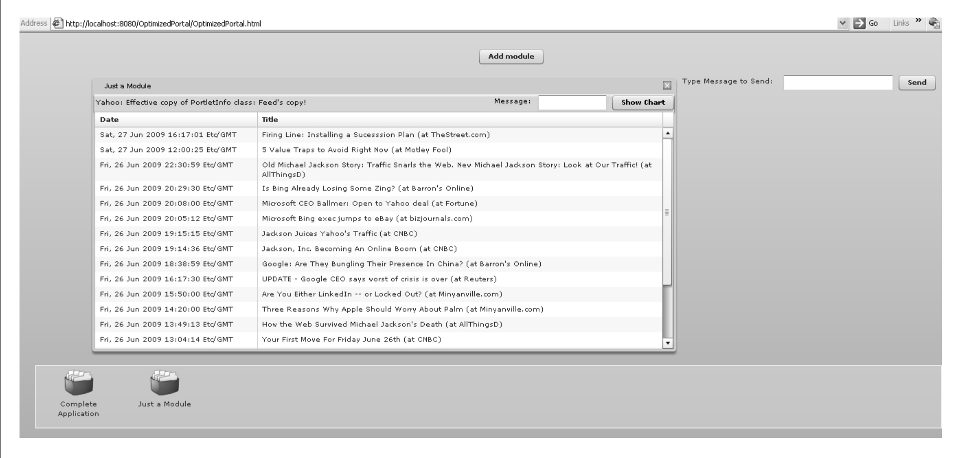


Figure 8-12. Loading the YahooFinancialNews module

Dragging the “Just a Module” icon from the bottom bar to the empty area of the application loads the module and populates it with the data, as you can see in Figure 8-12.

In Figure 8-13, Charles shows that two more *.swf* files were loaded: *YahooFinancialNews.swf* and *datavisualization_3.3.0.4852.swf*.

For this experiment we didn’t use a signed *datavisualization.swz*, because the goal here is to demonstrate the fact that the even though the *datavisualization* library is linked as an RSL to more than one module, it’ll get loaded only once.

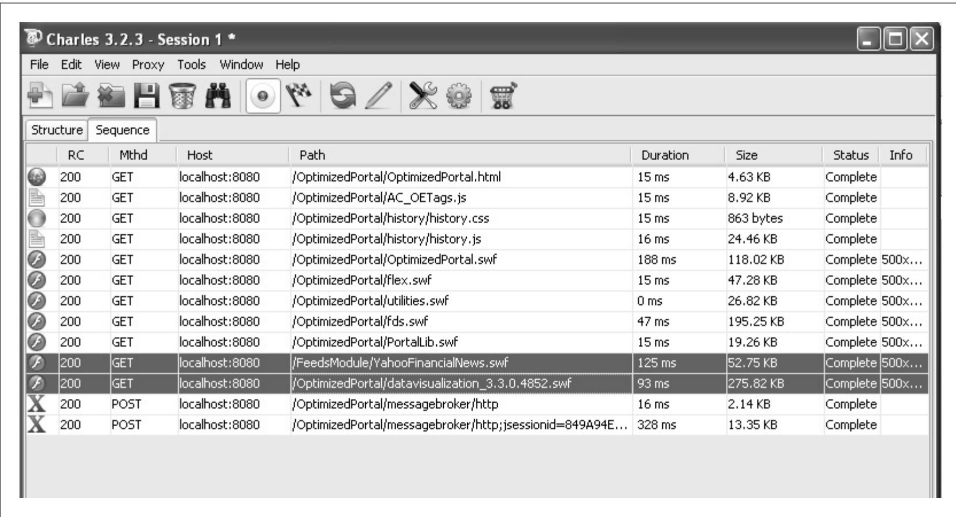


Figure 8-13. YahooFinancialNews comes with datavisualization.swf

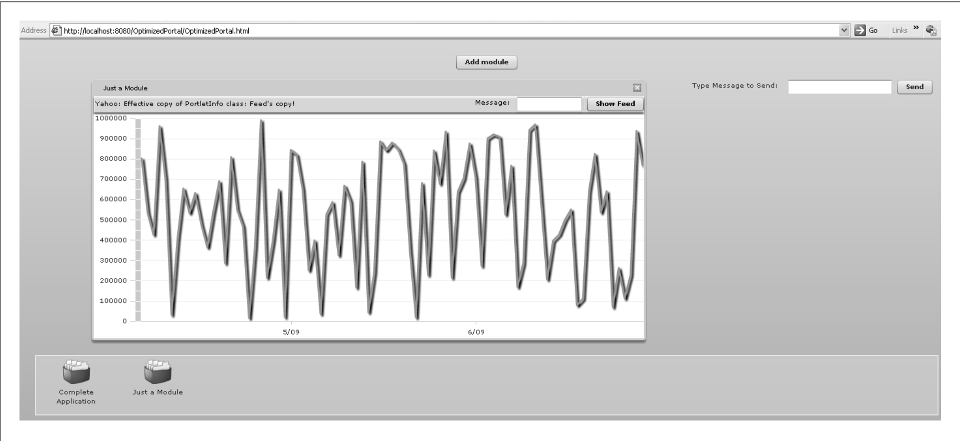


Figure 8-14. Loading the ChartModule

When the user switches to chart view by clicking the Show Chart button (Figure 8-9) yet another module, *ChartModule*, will be loaded. This module also has the *datavisualization* RSL in its build path. The *OptimizedPortal* view looks like Figure 8-14.

Charles shows the results in Figure 8-15.

As you can see, *ChartModule.swf* has been downloaded, but its *datavisualization* RSL has not because it was already downloaded by the module `YahooFinancialNews`—proof that you can do smarter RSL loading to improve your portal's performance.

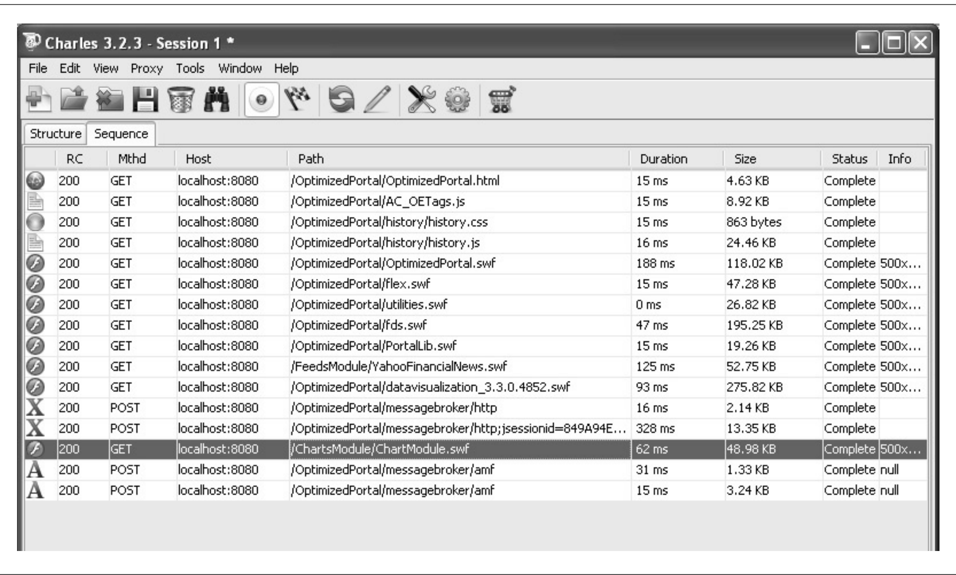


Figure 8-15. ChartModule comes without datavisualization

In this experiment we've been using *datavisualization.swc* as a guinea-pig RSL, but you can and should apply the same technique for any business-specific RSL that your application might use.

A Grab Bag of Useful Habits

This section discusses three areas that may seriously affect the performance of your application: memory leaks, Flash Builder's Profiler, and the Just-in-Time compiler. At the end of this section you'll find a checklist of items that can help you in planning performance-tuning tasks.

Dealing with Memory Leaks

Wikipedia defines a *memory leak* as "a particular type of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed. This condition is normally the result of a bug in a program that prevents it from freeing up memory that it no longer needs."

Flash Player offers help in dealing with memory leaks. A special process called the *garbage collector* (GC) runs periodically and removes from memory any objects that the Flex application is no longer using. It counts all references to each object in memory, and when an object's reference count goes down to zero, that object is removed from memory.

In some cases, two objects have references to each other, but neither of them is referred to anywhere else. In this case the reference count never becomes zero, so the objects will never be collected. Flash Player tries to identify such situations by running a slower method called "mark and sweep."

You should write code that nullifies non-local reference variables that point to objects that are not needed (`myGreatObj=null;`); likewise, if you call `addChild()` don't forget about `removeChild()`, and if you call `addEventListener()` remember to call `removeEventListener()`.

The `addEventListener()` function has three optional arguments. Setting the last one to `true` indicates that so-called "weak references" should be used with this listener, meaning that if this object has only weak references pointing to it, the GC can remove it from memory.

Your main target in optimization of memory consumption should be unloading of unneeded data. Following these recommendations will prevent RAM from becoming littered with unneeded objects.

Closures

In some cases there is not much you can do about memory leaks: some object instances will get stuck in memory and gradually degrade the performance of your application.

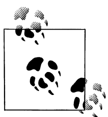
A *closure*—that is, an object representing an anonymous function—will never be garbage-collected. Here's an example:

```
myButton.addEventListener("click",
        function (evt:MouseEvent){//do something});
```

With syntax such as this, the object that represents the handler function gets attached to the stage as a global object. You can't use syntax like `removeEventListener("click", myHandlerFunction)` here because the closure used as an event handler doesn't have a name. Things get even worse because all objects created inside this closure won't be garbage-collected either.

Be careful with closures. Don't use them just because it's faster to create an anonymous in-place function than it would be to declare a named one. Unless you need to have an independent function that remembers some variables' values from its surrounding context, don't use closures, as they may result in memory leaks.

You can't use weak references with the listeners that use closures, as they won't have references to the function object and will be garbage-collected.



If you add a listener to the `Timer` object, use weak references; otherwise, Flash Player will keep the reference to this object for as long as the timer is running.

Opportunistic garbage collection

Flash Player's mechanisms for allocating and deallocating memory are browser-specific, so the GC will work differently depending on the web browser in which your Flex application is running.

How do you identify that you have memory leaks? If you can measure available heap memory before and after the GC runs, you can make a conclusion about the memory leaks. But this brings us to the next question: "How can you force garbage collection?"

There is a trick with the `LocalConnection` object that can be used to request immediate garbage collection. If your program creates two instances of the `LocalConnection` object using the same name in the `connect()` call, Flash Player will initiate the GC process:

```
var conn1:LocalConnection = new localConnection();
var conn2:LocalConnection = new localConnection();
conn1.connect("MyConnection");
conn1.connect("MyConnection");
```

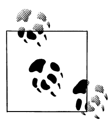
It's not typical, but you can use the `LocalConnection` object to send and receive data in a single *.swf* (for example, to communicate between modules of the same Flex application).

Some web browsers force garbage collection on their own. For example, in Internet Explorer minimizing the browser window initiates garbage collection.

If you know that all of your users will have Flash Player 9.0.115 or newer, you may use the following API to cause GC: `flash.system.System.gc()`.

JIT Benefits and Implications

The Flex compiler is actually a set of subcompilers that convert your ActionScript and MXML code into different formats. For example, besides *mxmlc* and *compc*, there is a precompiler that extracts the information from the precompiled ActionScript byte code (ABC). ABC is the format that Flash Player runs, but the story doesn't end there.



You can read more about compilers at *http://opensource.adobe.com/wiki/display/flexsdk/Flex+3+Compiler+Design*.

Most of the performance advances in the current version of AS3 as compared to AS2 are based on its *Just-in-Time* (JIT) compiler, which is built into Flash Player. During the *.swf* load process, a special *byte code verifier* analyzes the code to ensure that it is valid for execution: it performs validation of code branches, verifies types and linkages, does early binding, and validates constants.

The results of the analysis are used to produce a *machine-independent representation* (MIR) of the code that the JIT compiler can use to efficiently produce performance-optimized machine-dependent code. Unlike Flash VM code that is a classic *stack machine*, the MIR is more like a parsed execution path prepared for easy register optimization. The MIR compiler does not process the entire class, though; rather, it takes an opportunistic approach and optimizes one function at a time (a much simpler and faster task). For example, this is how the source code of an ActionScript function is transformed into assembly code for the x86 Intel processor:

- In ActionScript 3:

  ```
  function (x:int):int {
  return x+10
  }
  ```

- In ABC:

  ```
  getlocal 1
  pushint 10
  add
  returnvalue
  ```

- In MIR:

  ```
  @1 arg +8// argv
  @2 load [@1+4]
  @3 imm 10
  @4 add (@2,@3)
  @5 ret @4 // @4:eax
  ```

- In x86:

  ```
  mov eax,(eap+8)
  mov eax,(eax+4)
  add eax,10
  ret
  ```

x86 code can execute 10 to 100 times faster than ABC code, a gain that easily justifies including the extra JIT compilation step. In addition, the JIT process eliminates dead code, optimizes common expressions, and folds constants. On the machine code generation side, it adds optimized use of registers for local variables and instruction selection.

You need to help realize these benefits by carefully coding *critical* loops (avoid over-optimization!). For example, consider the following loop:

```
for (var i:int =0; I < array.length; i++) {
    if( array[i] == SomeClass.SOMECONSTANT)...
```

It can be optimized to produce very efficient machine code by removing calculations and references to other classes, thus keeping all references local and optimized:

```
var someConstant:String = SomeClass.SOMECONSTANT;
var len:int = array.length;

for (var i :int = 0; I < len; i++) {
    if (array[i] == someConstant)
```

JIT is great at providing machine code performance for heavy calculations, but it has to work with data types the CPU handles natively. At the very least, in order to make JIT effective you should typecast to strong data types whenever possible. The cost of typecasting and fixed property access is lower than the cost of a lookup even for a single property.

JIT only works on class methods. As a result, all other class constructs, variable initialization on the class level, and constructors are processed in interpreter mode. You have to make a conscious effort to defer initialization from constructors to a later time so JIT can have a chance to improve performance.

Using the Flash Builder Profiler

The Flash Builder Profiler monitors memory consumption and execution time. However, it monitors very specific execution aspects based on information available inside the virtual machine, which currently reports incomplete data. For example, memory consumption reported by the profiler and by the OS will differ greatly because the profiler fails to account for the following:

- Object definitions and static variables
- Memory used by the JIT compiler
- Unfilled areas of the 4 KB memory pages resulting from deallocated objects

More importantly, when showing memory used by object instances it reports the size used by object itself and not by any sub-objects. For example, if you are looking at employee records, the profiler will report the records to be of the same size regardless of the sizes of the last and first names. Only the sizes of the properties pointing to the string values are reported within the object; the actual memory used by the strings is reported separately, and it's impossible to quantify as belonging to specific employee records.

The second problem is that in any sizable application, with deferred garbage collection there are a lot of issues to do with comparing memory snapshots. Finding holding references as opposed to circular ones is a tedious task, which hopefully will be simplified in the next version of the tool.

As a result, it is usually impractical to check for memory leaks on the large application level. Most applications incorporate memory usage statistics like `System.totalMemory` into their logging facilities to give developers an idea of possible memory issues during

the development process. However, a much more interesting approach is to use the Flash Builder Profiler as a monitoring tool while developing individual modules. If using this approach, you'll need to invoke `System.gc()` prior to taking memory snapshots so irrelevant objects won't sneak into your performance analysis.

The profiler offers a great deal of information for performance analysis. It reveals the execution times of every function, as well as cumulative times, and (most importantly) it provides insights into the true cost of excessive binding, initialization and rendering costs, and computational times. You will not be able to see the time spent in handling communications, loading code, JIT compilation, and data parsing, but you will at least be able to measure direct costs related not to design issues but to the coding techniques in use.

Performance Checklist

While planning to improve the performance of your RIA, consider the five categories outlined here.

Startup time

To reduce startup time:

- Use preloaders to quickly display either functional elements (e.g., a logon screen) or some business-related news.
- Design with modularization and optimization of *.swf* files in mind (remove debugger and metadata information).
- Use RSLs and signed framework libraries.
- Minimize the initially displayed UI.
- Externalize (don't embed) large images and unnecessary resources.
- Process large images to make them smaller for the Web.

UI performance

To improve user interface performance at startup:

- Minimize usage of containers within containers (especially inside data grids). Most UI performance issues are derived from container measurement and layout code.
- Defer object creation and initialization (don't do it in constructors). If you postpone creation of UI controls up to the moment they become visible, you'll have better performance. If you do not update the UI every time one of the properties changes but instead process them together (`commitProperties()`), you'll most likely be able to execute common code sections responsible for rendering once instead of multiple times.

- For some containers in your application, use `creationPolicy=queued` for deferred container creation and enhanced perceived initialization performance.
- Provide adaptive user-controlled duration of effects. Although nice cinematographic effects are fine during the application's introduction, users should be able to control their timing and disable them.
- Minimize updating of CSS in the runtime. If you need to set styles based on data, do it early—preferably in the initialization stage and not in the `creationComplete` event, as this will minimize the number of lookups required.
- Validate performance of data-bound controls (such as `List`-based controls) for scrolling and manipulation (sorting, filtering, etc.) early in development and with maximum data sets. Do not use the Flex `Repeater` component with sizable data sets.
- Use the `cacheAsBitmap` property for fixed-size objects, but not for resizable and changeable objects.

I/O performance

To speed up I/O operations:

- Use AMF rather than Web Services and XML-based protocols, especially for large (over 1 KB) result sets.
- Use strong data types with AMF on both sides for the best performance and memory usage.
- Use data streaming to push real-time information to the client. If you have a choice of how to send data from the server to clients, select them in the following order: RTMP, AMF streaming, long polling.
- Use lazy loading of data, especially with hierarchical data sets.
- Try to optimize a legacy data feed; compress it on a proxy server at least, and provide an AMF wrapper if possible.

Memory utilization

To use memory more efficiently:

- Use strongly typed variables whenever possible, especially when you have a large number of instances.
- Avoid using the XML format.
- Provide usage-based classes for non-embedded resources. For example, when you build a photo album application, you'll want to cache more than one screenful of images to make scrolling faster (i.e., so you won't have to reload already scrolled images). The amount of memory that would be required and common sense, however, should prevent you from keeping ***all*** images in the cache.

- Avoid unnecessary bindings (e.g., bindings used for initialization), as they produce tons of generated code and live objects. Provide initialization through your code when it is needed and has a minimal performance impact.
- Identify and minimize memory leaks using the Flash Builder Profiler.

Code execution performance

For better performance, you can make your code JIT-compliant by:

- Minimizing references to other classes
- Using strong data types
- Using local variables to optimize data access
- Keeping code out of initialization routines and constructors

Additional code performance tips are:

- For applications working with large amount of data, consider using the `Vector` data type (for Flash Player 10 and above) rather than `Array`.
- Binding slows startup, as it requires initialization of supporting classes and heavily pollutes memory; keep it to a minimum.

Summary

In this chapter you learned how to create a small no-Flex logon (or similar) window that gets downloaded very quickly to the user's computer, while the rest of the Flex code is still in transit.

You now know how to create any application as a mini-portal with a light main application that loads light modules that:

- Don't have the information from *services-config.xml* engraved into their bodies
- Can be tested by a developer with no dependency on the work of other members of the team

You won't think twice when it comes time to modify the code of even such a sacred cow as `SystemManager` to feed your needs. (Well, you *should* think twice, but don't get too scared if the source code of the Flex framework requires some surgery.) If your version of modified Flex SDK looks better to you than the original, submit it as a patch to be considered for inclusion in a future Flex build; the website is *http://opensource.adobe.com/wiki/display/flexsdk/Submitting+a+Patch*.

While developing your enterprise RIA, keep a copy of the "Performance Checklist" on page 53 section of this chapter handy and refer to it continuously from the very beginning of the project.

If you've tried all techniques you knew to minimize the size of a particular *.swf* file and you are still not satisfied by its size, as a last resort, create an ActionScript project in Flash Builder and rewrite the module without using MXML. This might help.