

Bonus Chapter

Optimizing Your Flex Applications

In This Chapter

- ▶ Creating your application's SWF file
 - ▶ Dividing your application into Runtime Shared Libraries or Flex modules
 - ▶ Taking advantage of persistent framework caching
 - ▶ Avoiding memory leaks
 - ▶ Finding out about common performance and memory-management tips
-

As highly optimized as Flex is, you have the responsibility as an application developer to keep an eye on the performance, memory management, and size of your Flex application. This chapter covers common tips and tricks that you can use to make sure you're creating a lean and mean Flex application. You find out how to make the *SWF file*, the output of your Flex application, as small as possible for production purposes and how to make your application use the least amount of memory and run as fast as possible.



Keep performance optimizations, memory management, and size in mind when you start building your Flex application so that you don't have to diagnose problems at the end of your development cycle.

Making Your SWF File As Small As Possible

When you compile and run your Flex application, the Flex compiler produces a SWF file. This file contains the Flex framework code, your application code,

your embedded assets and fonts, and some “glue” code that the Flash Player needs to run your full Flex application. You want to make sure that the SWF file you produce for deployment of your application is as small as possible so end users don’t have to deal with long download times and long startup times for the application.

Depending on the type of Flex development you’re doing, you can create three different types of SWF files:



- ✓ **A development SWF file:** You can create this type of SWF file by choosing the File⇨Export⇨Release Build option. The SWF file doesn’t contain extra information, such as debugging- or accessibility-related code. Instead, this type of SWF file is the most appropriate one for production deployment use.

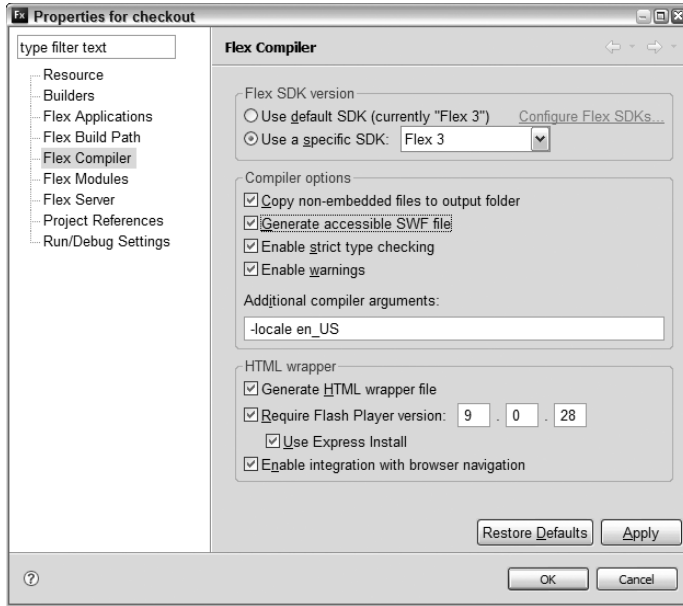
The other two types of SWF files that you can produce are larger. So, make absolutely sure that you have a compelling reason to publish one of those large types of SWF files, rather than a development SWF file, for deployment purposes.

- ✓ **A debug SWF file:** The second type of SWF file that you can produce is a debug SWF file. The Flex compiler generates this type of SWF file when you debug a Flex application in Flex Builder, either by clicking the Debug icon in the main toolbar or by choosing Run⇨Debug Application. A debug SWF file contains extra debugging-specific information (such as the line numbers of code file) and thus has a larger size. Don’t deploy a debug SWF file for production purposes.

- ✓ **An accessibility-enabled SWF file:** This SWF file is larger because it includes extra information that a screen-reader needs to interact correctly with the Flex application. By default, Flex applications aren’t designed to allow a screen-reader to interact with them. A *screen-reader* is a software application that interprets the contents of an application running on your computer. You have to opt in to this choice, which creates a larger, but smarter, SWF file that has screen-reader capabilities.

To create an accessibility-enabled SWF file, select the project in the Flex Navigator, choose File⇨Properties to open the Properties dialog box for your project, select Flex Compiler in the left pane, and then select the Generate SWF File check box on the right, as shown in Figure BC-1. If you’re building a Flex application that requires full accessibility, enable this option — but be aware that the resulting SWF file is large.

Figure BC-1:
You can use Flex Builder to create accessible Flex applications.



Modularizing Flex Applications

The SWF file that the Flex compiler creates when you launch your Flex application contains the Flex framework code, as well as your application-specific code and assets. For larger applications, you can end up with a quite large SWF file. Don't worry — Flex provides a way to modularize your Flex application so that the application pulls in certain chunks of code at runtime so that the SWF file is relatively small.

You can modularize your Flex applications in two ways — by using either Runtime Shared Libraries (RSLs) or Flex modules:

- ✓ A *Flex RSL* is just a SWF file that contains code that can be shared as a library between different application SWF files. We discuss how to create a Flex RSL, specifically how to take advantage of a new caching mechanism in Flex 3 and Flash Player 9, in the upcoming “Persistent Framework Caching” section.
- ✓ A *Flex module* is a SWF file that contains some code that can be loaded in to and unloaded from an application at runtime. Flex modules are



powerful. They let you decrease the size of your application SWF file by partitioning code into separate modular SWF files that the application can load in to and unload from your application while it is running. You can split your application into *modules*, or separate pieces, and the main application can load those modules on an as-needed basis.

By using Flex modules in your application, you can decrease the download time and startup time of your application because you reduce the application's size.

The following sections show you how to create a Flex module for your Flex application. For this particular example, you create a standard e-commerce Flex application that you use to buy and sell candy. You can modularize the main application and create a checkout module that loads in when users finish selecting their candy and want to place the order. When a user places an order, the checkout module then loads in to the main candy-selling application.

Creating modules in Flex Builder

In Flex Builder, follow these steps to create a module:

1. Choose File⇒New⇒MXML Module to create a new Flex module.

You can use the MXML Module option to create a checkout module for the example candy-selling Flex application. The New MXML Module dialog box, shown in Figure BC-2, appears.

2. In the top part of the dialog box, select the parent folder for the module.

Normally, you should place the module in the same root folder as the application so that relative paths for external and embedded assets are the same. This means the default setting is acceptable to choose.

3. In the Filename box, enter the name of the module.

4. In the Width and Height boxes, enter the module's dimensions as percentages. From the Layout drop-down list, select the kind of layout you want it to have.

Because a module is basically a mini-application that the main application pulls in on an as-needed basis, the module follows the same layout rules as a Flex application. In the example checkout module, create a module that has a vertical layout and 100-percent width and height so that its size matches the main candy-selling Flex application.

5. In the Module SWF Size area, determine whether you want to optimize your module so that the SWF files don't duplicate the Flex classes that the main application and the module share.

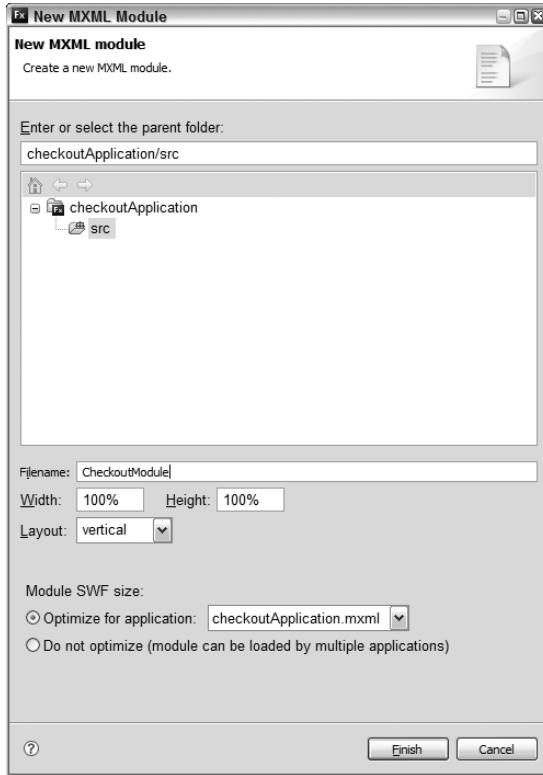


Figure BC-2:
The New MXML Module dialog box helps you create a new module.

You'll almost always choose to optimize the module, so the Optimize for Application radio button is automatically selected.

- 6. If you decided to optimize your module in Step 5, from the drop-down list to the right of the radio button, specify which application you want the module optimized against.**

In this example, you want the checkout module to be optimized against the candy-selling Flex application.

- 7. Click the Finish button.**

Your module is created, and the file opens in Flex Builder's Source mode, containing the `<mx:Module/>` tags.

After you create the checkout module, you need to add all the relevant code so that the checkout module works and users can enter their credit card information and mailing addresses, and submit their orders through the Flex application. Figure BC-3 shows what the sample checkout module looks like. Listing BC-1 contains all the code that dictates how the checkout module should look and behave between the `<mx:Module/>` tags.

The screenshot shows a web form titled "Billing Information" with a light gray border. It contains three input fields: a dropdown menu for "Credit Card" with "Visa" selected, a text input for "Credit Card Number Date", and a date field for "Expiration Date". Below this form are two more sections: "Shipping Information" and "Gift Options, Promotional Codes", both with light gray headers.

Figure BC-3: This check-out view loads in as a Flex module while the application is running.

Listing BC-1: Loading a Flex Module into an Application at Runtime

```
<?xml version="1.0" encoding="utf-8"?><mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical" width="100%" height="100%">
  <mx:Accordion width="400" height="400">
    <mx:Form width="100%" height="100%" label="Billing Information">
      <mx:FormItem label="Credit Card">
        <mx:ComboBox dataProvider="{['Mastercard', 'Visa', 'American Express']}" selectedIndex="1" />
      </mx:FormItem>
      <mx:FormItem label="Credit Card Number Date">
        <mx:TextInput width="80%" />
      </mx:FormItem>
      <mx:FormItem label="Expiration Date">
        <mx:DateField />
      </mx:FormItem>
    </mx:Form>
    <mx:Form label="Shipping Information">
      <!--Shipping Related Code />
    </mx:Form>
    <mx:Form label="Gift Options, Promotional Codes">
      <!--Promotional and Gift option Code. Submit Button />
    </mx:Form>
  </mx:Accordion>
</mx:Module>
```

Loading and unloading modules

After you create the candy checkout module (as we describe in the preceding section), you need to figure out how to load and unload the module at runtime. Remember, you want the module to load when the user navigates to

checkout — so, when the user clicks the Checkout button after selecting the candy that she wants to purchase, you need to load the candy checkout module.

You can load and unload modules in two ways — by using a `ModuleLoader` or the `ModuleManager`. You can most easily load and unload modules by using a `ModuleLoader`, so this example uses that approach. You create a `ModuleLoader` by using the `<mx:ModuleLoader />` MXML tag. The main property that you need to set on the `ModuleLoader` tag is the `url` property, which specifies the location of the module SWF file. The module loads when the `url` attribute is set. So, if you want to load the checkout module when the user clicks the Submit button on the main shopping page, simply set the `url` property of the `ModuleLoader`:

```
<!-- We create the checkout module when the Submit button
      is clicked -->

<mx:ModuleLoader id="checkoutModule" />
<mx:Button label="Submit" click="checkoutModule.url =
      'CheckoutModule.swf' " />
```

Voilà! When the user clicks the Submit button, the checkout module is created, and that module is added to the application in the same location in which the `ModuleLoader` tag lives with respect to the larger application.

Taking Advantage of Persistent Framework Caching

Persistent framework caching is a new feature in Flex 3 and Flash Player 9 that allows the Flash Player to cache a Runtime Shared Library (RSL) of the Flex framework code (which your published application SWF file includes). So, if users have a copy of the Flex framework RSL in their Flash Player cache, they don't need to download the code bits that contain the Flex framework when they download and access your Flex application. This new feature enables you to reduce the size of your Flex application SWF files by many, many kilobytes!

This new Flash Player cache is different from a browser cache because Flash Player manages it alone — which means you can't clear it or add anything to it. The Flash Player cache serves as a cache for Adobe components, and the Flex framework is cached as an Adobe component. The first time an application user downloads a Flex application that's configured to use the Flex framework as an RSL, the user's Flash Player cache receives a permanent copy of that RSL. The next time that user downloads a similarly configured Flex application, the Flash Player recognizes that the RSL is already cached and uses a copy of the Flex framework from the cache. The Flash Player han-

dles the security associated with persistent framework caching. The Flash Player verifies the Flex framework RSL so that only certified code is used. Configuring your Flex applications to use the framework RSL can reduce the size of your Flex SWF file by anywhere from 90K to 500K.

Follow these steps to create an RSL that encapsulates all the Flex framework code in Flex Builder so that your application can take advantage of the new persistent framework caching feature:

1. Select the project in the Flex Navigator and choose File⇨Properties.

The Project Properties dialog box for your target project appears.

2. Select the Flex Build Path option in the left pane and then click the Library Path tab in the right pane, as shown in Figure BC-4.

Next, you change the option for how code is linked in to your application.

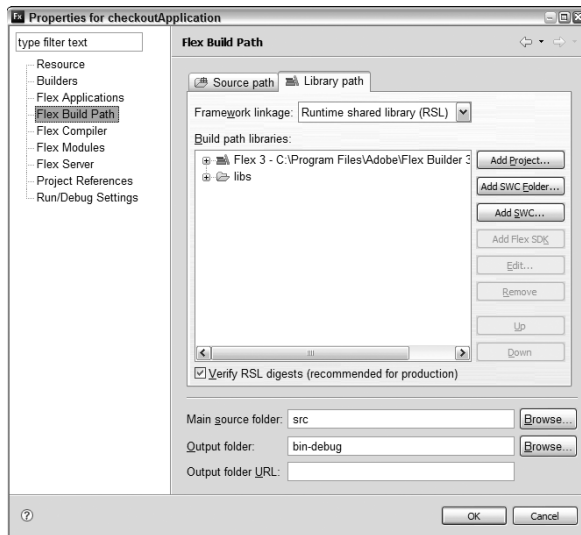


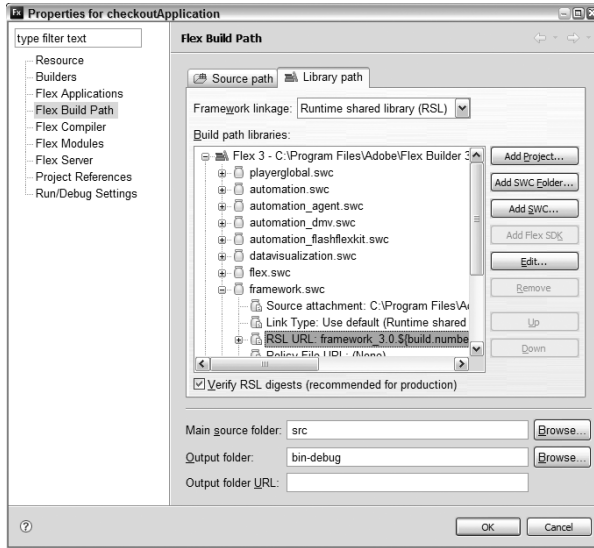
Figure BC-4:
Create an RSL for the Flex framework code so that it's cached in the Flash Player.

3. Select Runtime Shared Libraries (RSL) from the Framework Linkage drop-down list, if it isn't already selected.

4. Expand the Flex 3 folder and navigate to the framework.swc entry. Expand that entry and select the RSL URL entry.

Your dialog box should now look like Figure BC-5.

Figure BC-5:
Select the RSL URL in Build Path Libraries to create the Flex framework RSL.



5. Click the Edit button to create the RSL.

When you click the Edit button, the Library Path Item Options dialog box appears, which lets you configure the RSL. You probably don't have to change many of the Flex Builder default settings. The preset values in this dialog box, such as the Digests settings, ensure that the correct framework RSL is linked in.

6. In the Deployment Paths area, select the path of the RSL code that you want linked in to your application. Select the option that ends in .swf, rather than .swz, so that the framework code is linked in as a SWF file, not as a SWZ file. To learn more about the difference between a SWZ file and a SWF file, refer to the Flex documentation.

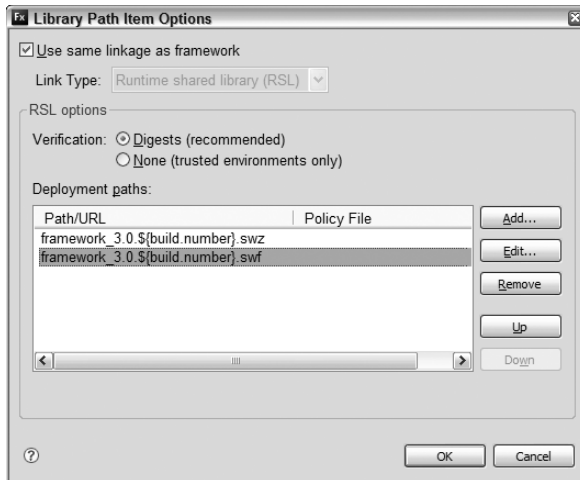
The dialog box should look like Figure BC-6.

7. Click OK to close the Library Path Item Options dialog box.

8. Click OK again to close the Project Properties dialog box.

You need to clean and rebuild your project. (To find out more about cleaning and rebuilding a project, check out Chapter 4.) When you next run the application, the Flex compiler produces a much smaller SWF file than it did before you created a framework RSL.

Figure BC-6:
Create a Flex framework RSL as either a SWF file or a SWZ file.



Diagnosing Memory Leaks

When you use Flex, you can definitely inject memory leaks into your application code accidentally. A *memory leak* occurs when you create some code that uses up more and more memory while the application runs. Memory leaks can slow down your application, and they can eventually cause the application to crash.

Diagnosing memory leaks in a Flex application can be difficult, but certain tools, such as the Flex Profiler, can make the process easier. The Flex Profiler also helps you diagnose performance issues in a Flex application. To diagnose memory leaks by using the Flex Profiler, run the profiler in memory mode and interact with your application. While you run through your application, watch the displayed information for increased memory consumption that never goes away. Constantly increasing memory consumption may be a sign that some code is allocating more and more system memory without ever releasing it, which is the most common type of memory leak. For more on how to use the Flex Profiler, check out Chapter 5.

Two of the most common causes of memory leaks are

- ✓ The use of `Dictionary` objects
- ✓ The creation of event listeners that don't get garbage collected



Garbage collection is the native process built into the Flash Player that deletes objects not in use to free up memory. A simple algorithm determines whether an object is in use (and thus not available to be garbage collected) or not in use (and thus ready to be cleaned up). Determining whether an object should be garbage collected requires determining whether any other objects keep a reference to that object. If so, other objects depend on the object, so it isn't garbage collected.

A common memory leak occurs when an object keeps around an unnecessary reference, preventing that object from being garbage collected, even though it should be cleaned up. You most often have this kind of memory leak when you create `Dictionary` ActionScript objects or Flex event listeners that have strong, rather than weak, references.

ActionScript Dictionary objects

An ActionScript `Dictionary` object allows you to create a map of objects, called *keys*, in which a value can be stored in the `Dictionary` object and looked up for retrieval based on the key's identity. It's basically a grab bag in which you can store objects or data for later retrieval and use.

You can create an ActionScript `Dictionary` object so that if the only reference to the key object is the `Dictionary` object, the key object can be garbage collected. This functionality is determined by the `weakKeys` parameter that's passed in when you create a new `Dictionary` object. By default, the `weakKeys` parameter is `false`, meaning the key object isn't garbage collected, even if the only reference to it is the `Dictionary` object. To avoid memory leaks, and if you know that you can clean up objects whose only reference is the `Dictionary` object, toggle that parameter to `true` when you create a new `Dictionary` object as demonstrated in the following line of code:

```
public var dict:Dictionary = new Dictionary(true);
```

To find out more about ActionScript `Dictionary` objects, refer to the online Flex documentation.

Event listeners with strong references

You can create a Flex event listener so that it's either a strong reference or a weak reference. You identify an event listener as strong or weak to determine whether an object acted upon by that event listener can be garbage collected. You can create only weak event listeners in ActionScript. Event listeners created in MXML are always identified as strong. For the most part, you

want your event listener to be weak referenced so that if an object's only reference is an event listener, the Flash Player Garbage Collector collects that object and frees up its memory to the larger system. Whenever you create a new event listener in ActionScript code by using the `addEventListener` method call, you should set the last parameter to `true` if you want the event listener to be identified as a weak reference. By default, this parameter (`useWeakReference`) is `false`. To help prevent memory leaks, toggle this property to `true` as demonstrated in the following line of code:

```
myButton.addEventListener("click", clickHandler, false, 0, true);
```

Performance Optimization Tips

Thinking about performance while you write your application code always creates a better running application. Thinking about performance while you write your Flex application prevents finding a last-minute performance flaw at the end of your product cycle — or worse, after the application has shipped!

When you're writing your application, watch out for certain coding practices that might negatively affect the application's performance. You can find many of these tips and tricks in a white paper in the Adobe Flex Developer Center, an online resource that Adobe hosts, which offers technical articles and white papers. (See Chapter 20 for more on the Flex Developer Center.) Adobe published a white paper during the Flex 1.0 timeframe that describes common performance flaws and how to avoid them when you write Flex application code. Adobe updates this white paper with each revision of Flex. Look over this white paper (at www.adobe.com/devnet/flex/articles/client_perf.html) if you want to find out how to avoid performance and memory pitfalls.

In the following sections, we summarize a few of the key tips that the Flex performance whitepaper describes (but read that white paper for a full, detailed list of tips and tricks).

Make your startup code mean and lean

Be careful about what code you invoke when you start an application or a component. Make any code that you add to the `initialize` event handler or `creationComplete` event handler (both events are dispatched at the beginning of the application and component lifecycle) as lean and mean as possible.

The most expensive call in the Flex framework is the `setStyle` ActionScript method. When your code, or the Flex framework code, calls the `setStyle` method on any object, a tree of calls is produced so that the style that's set on a parent object is propagated down to all necessary child objects. Sometimes, this resulting tree can be quite large, so the call can take a long time to execute. If you find yourself calling the `setStyle` method frequently in your application code, or you're setting numerous styles at startup of the application or after a component has been created, look into other ways of setting styles, such as using CSS style sheets or creating `<mx:Style/>` blocks in your application code. For the lowdown on setting styles in your application, turn to Chapter 18.

Convert relative layout containers into absolute positioning containers

Flex offers a great set of relative layout containers, such as `HBox`, `VBox`, `Form`, and `Panel`, which handle the measurement and positioning of child objects automatically. Although these containers can make laying out your application much more convenient, they're not the fastest containers offered in the Flex framework. The relative layout containers must do two passes over the child controls to size those controls and then position them. In absolute positioning containers, such as `Canvas`, `Application`, `Panel`, or `TitleWindow`, the `layout` attribute is set to a value of `absolute`. In the absolute positioning containers, explicit widths, heights, and `x` and `y` positions are automatically set on child elements, so those containers avoid your having to make the first measurement pass.

If your application performance is sluggish, you may want to look into converting some of your relative layout containers into absolute positioning containers and specifying `x` and `y` values. To preserve the reflowing and resizing nature that the relative layout containers offer, you can add layout constraints to your absolute positioning containers. To find out more about containers and layout constraints, check out Chapter 10.

Reevaluate your layout hierarchy

If you nest relative layout containers, your application may be slower than necessary. When you nest containers that don't need to be nested, you just add to the number of objects that the parent application has to create, size, and position, which adds to the startup expense of your application. If you have many relative layout tags contained within each other, reevaluate your layouts. Do you really need all those tags, such as `HBox`, `VBox`, and `Panel`? Did you set the `layout` attribute of `Panel` and `TitleWindow` containers to `vertical`, `horizontal`, or `absolute` so that you don't need to add a first

child that's an HBox, VBox, or Canvas container? Asking yourself these questions while you create and refactor layouts can help you create speedy and skinny Flex applications.

You can visualize your layout container hierarchy by using the Show Surrounding Containers feature in Flex Builder Design view. This feature helps you visualize the containers in your Flex application and decide whether you're using too many containers. When you build more Flex applications, you can get the hang of limiting container usage to only those containers that you absolutely need. Reducing the number of containers definitely improves the startup performance of your application.