

# FLEX3 IN ACTION

SAMPLE CHAPTER

Tariq Ahmed

with Jon Hirschi and Faisal Abid

FOREWORD BY Ryan Stewart





***Flex 3 in Action***

by Tariq Ahmed  
with Jon Hirschi  
and Faisal Abid

Chapter 23

Copyright 2009 Manning Publications

# *brief contents*

---

## **PART 1 APPLICATION BASICS ..... 1**

- 1 ■ Introduction to Flex 3
- 2 ■ Getting started 21
- 3 ■ Working with ActionScript 43
- 4 ■ Layout and containers 69
- 5 ■ Displaying forms and capturing user input 96
- 6 ■ Validating user input 113
- 7 ■ Formatting data 139
- 8 ■ DataGrids, lists, and trees 155
- 9 ■ List customization 184

## **PART 2 APPLICATION FLOW AND STRUCTURE .....213**

- 10 ■ Events 215
- 11 ■ Application navigation 233
- 12 ■ Introduction to pop-ups 263
- 13 ■ View states 281
- 14 ■ Working with data services 294

- 15 ■ Working with XML 316
- 16 ■ Objects and classes 340
- 17 ■ Custom components 355
- 18 ■ Advanced reusability in Flex 380

**PART 3 THE FINISHING TOUCHES .....391**

- 19 ■ Customizing the experience 393
- 20 ■ Working with effects 420
- 21 ■ Drag-and-drop 440
- 22 ■ Charting 462
- 23 ■ Debugging and testing 484
- 24 ■ Wrapping up a project 506

# *Debugging and testing*

---

- This chapter covers:
- Debugging applications using the Flex debugger as well as third-party tools
- Using test automation tools to find bugs and fix them before they go live
- Deploying your applications

This chapter's purpose is to give you a sampling of all the weaponry available both out of the box and in the form of third-party tools, to help speed up the process of developing in Flex from a tooling perspective. Flex Builder comes with a full-blown debugger that allows you to step through the code and watch the value of variables in real time. And new to Flex 3 is the Profiler, which lets you analyze how memory and resources are consumed.

Beyond what comes out of the box, the Flex and Flash ecosystem provides tools from open source and low-cost solutions to full-blown enterprise test-automation packages that will help speed up your debugging and testing. You may not get into the more advanced tooling for a while, but from the start you'll need simple ways to figure out why things aren't working. Let's begin by discussing how you can debug your Flex applications.

## 23.1 Debugging

You can use debugging at multiple levels. We'll review each approach, starting with the easiest.

### 23.1.1 Using the Flash Debug Player

When you install Flex Builder, it automatically installs the Flash Debug Player for you; this Debug Player allows for diagnostic logging, and you can connect external tools to it for debugging purposes.

You can download the latest player at Adobe's website via <http://www.adobe.com/support/flashplayer/downloads.html>. Look in the Debugger Versions section.

To test which version you currently have, visit [http://kb.adobe.com/selfservice/viewContent.do?externalId=tn\\_15507](http://kb.adobe.com/selfservice/viewContent.do?externalId=tn_15507). Be sure the Debug Player item says Yes.

You should already have the Debug Player, but sometimes other software you install may try to install the production version of the Flash Player as part of its installation. In any case, if you're not running the Debug Player, go ahead and get it installed.

Once you've verified that you have the Debug Player, you need to configure logging.

### 23.1.2 Configuring logging

To get logging going, you need to set up the Flash Debug Player's configuration file. If there isn't one, create a text file called `mm.cfg` and place it in the appropriate directory based on your operating system (see table 23.1).

**Table 23.1** Location of the Flash Player's configuration file

Operating system	Path
MAC OS X	/Library/Application Support/Macromedia
Windows 95/98/ME	%HOMEDRIVE%\%HOMEPATH%
Windows 2000/XP	C:\Documents and Settings\[username]
Windows Vista	C:\Users\[username]
Linux	/home/username

Inside this file are three configuration properties to define, as listed in table 23.2.

**Table 23.2** Flash Player's configuration-file settings

Configuration property	Type	Description
<code>TraceOutputFileEnable</code>	Boolean	0 for false (default), 1 for true. Turns logging on or off.
<code>ErrorReportingEnable</code>	Boolean	0 for false (default), 1 for true. Turns logging of error messages on or off.

**Table 23.2** Flash Player's configuration-file settings (*continued*)

Configuration property	Type	Description
MaxWarnings	Number	Maximum number of warnings to record. Default is 100; use 0 for unlimited.

You'll want to set up your mm.cfg file to look like this:

```
TraceOutputFileEnable=1
ErrorReportingEnable=1
MaxWarnings=500
```

With logging now enabled, details will be stored in a log file called flashlog.txt whose location is operating-system specific (see table 23.3).

**Table 23.3** Output location of the Flash Player's log file

Operating system	Path
MAC OS X	/Users/[username]/Library/Preferences/Macromedia/Flash Player/Logs/
Windows 95/98/ME/2000/XP	C:\Documents and Settings\[username]\Application Data\Macromedia\Flash Player\Logs
Windows Vista	C:\Users\[username]\AppData\Roaming\Macromedia\Flash Player\Logs
Linux	/home/[username]/.macromedia/Flash_Player/Logs/

With the configuration good to go, you can now use it via the `trace()` function.

### 23.1.3 Using the `trace()` function

`trace()` is a simple global function that is available from anywhere in your code. It lets you record text to the flashlog.txt log file. Now that you have mm.cfg set up, you can use this function, as listing 23.1 demonstrates.

**Listing 23.1** Using `trace()` to write information to the Flash log file

```
<mx:Label text="Type in something:"/>
<mx:TextInput id="something"/>
<mx:Button click="trace('Something is currently: ' + something.text)"
  label="Record Something"/>
```

When you type something and click the Record Something button, the text you type shows up in the flashlog.txt file (listing 23.2 shows an example).

**Listing 23.2** Sample Flash log file

```
*** WARNING: Please use gotoAndPlay(1) in the last frame of your Motif
creative if you intend to loop. ***
*** Please use 'stop();' at the end of your Motif creative if you do not
intend for it to loop. ***
```

```

showing end copy
hiding end copy
showing end copy
Something is currently: show this please

```

This is an effective mechanism if you want to record the value of various variables and components.

Viewing this log file may seem cumbersome without the right tool; fortunately, trace-log viewers are available.

### 23.1.4 *Trace-log viewers*

It's one thing to make a log, and it's another to view the log. Using what you have so far, it would be tedious to continually reload `flashlog.txt` to see what it currently looks like.

Fortunately third-party tools are available that do this for you by having a viewing panel that automatically refreshes as the log file changes. You can use any generic log-viewing tool, but here are a few we like.

- *Flash Tracer*—This extension to the Firefox web browser adds a built-in panel to Firefox, which is convenient because you don't need to switch back and forth between two separate applications. Price: free. <https://addons.mozilla.org/en-US/firefox/addon/3469>.
- *ABLogFile Viewer*—This is an open source viewer that is actively maintained. It's generic in nature, but it remembers the last few files opened, so it's easy to get back to the log file. In Windows, the application data directory is hidden, so you may need to navigate to the directory using Explorer and drag and drop the file into the ABLogFile Viewer. Price: free. <http://www.amleth.com/ablogfile/>.
- *Afterthought*—This tool was developed specifically for Flash, but it hasn't been updated to reflect a modification in the current Flash Player that changed where the `flashlog.txt` file is stored; you'll need to edit the options to tell it where to look. This tool is nice because it makes sure the window floats at the front, can filter out warnings, and has handy Clear Screen and Clear Log buttons. Price: free. [http://www.blazepdf.com/downloads/afterthought\\_v2\\_1.zip](http://www.blazepdf.com/downloads/afterthought_v2_1.zip).

These log files obviously log text, but many things in Flex aren't textual in nature, so we'll need a way to convert these objects into strings.

### 23.1.5 *Converting objects to strings*

Whether you're dumping complex data types to a log file or on screen, you must transform them into text. Flex makes this easy: most objects support a `toString()` function that will do the conversion for you. Here's an example:

```

var myNumber:Number = 32;
trace(myNumber.toString());

```



But for more complex objects like `ArrayCollections`, the `ObjectUtil` `ActionScript` class has a `toString()` function that takes a reference of any other object and converts it to a string.

In listing 23.3, the `ObjectUtil` class is used to display all the items in an `ArrayCollection`.

### Listing 23.3 Using `ObjectUtil` to dump an object's properties

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="trace(ObjectUtil.toString(myAC))">

<mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;
    import mx.utils.ObjectUtil;
    public var myAC:ArrayCollection = new ArrayCollection([
      {label:"Jon Hirschi", data:"jhirschi"},
      {label:"Tariq Ahmed", data:"tahmed"},
      {label:"Frank Krul", data:"fkrul"}
    ]);
  ]]>
</mx:Script>
</mx:Application>
```

This code produces something like the log file shown in listing 23.4.

### Listing 23.4 Sample output from `ObjectUtil`

```
(mx.collections::ArrayCollection)#0
  filterFunction = (null)
  length = 3
  list = (mx.collections::ArrayList)#1
    length = 3
    source = (Array)#2
      [0] (Object)#3
        data = "jhirschi"
        label = "Jon Hirschi"
      [1] (Object)#4
        data = "tahmed"
        label = "Tariq Ahmed"
      [2] (Object)#5
        data = "fkrul"
        label = "Frank Krul"
  uid = "70528450-89C6-9120-D5D4-847C1D8C3B70"
  sort = (null)
  source = (Array)#2
```

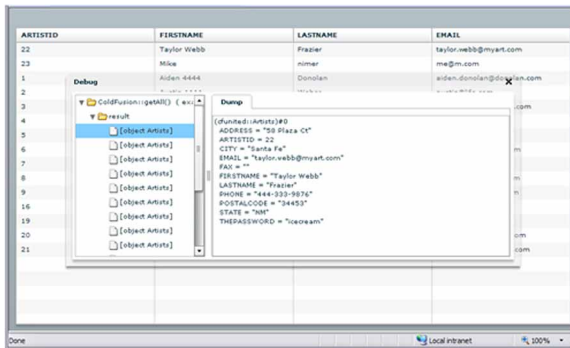
Another popular approach shows the value on screen via the `Alert.show()` function, instead of dumping it to a log file. Here's an example:

```
Alert.show(ObjectUtil.toString(myAC))
```

Using this technique and the Flash log file can get you reasonably far, but to take your game to a new level, you'll want more advanced tooling.

### 23.1.6 *FxSpy*

FxSpy (a.k.a. Flex Spy) is an open source tool that allows you to inspect your components and see what their properties and styles are set to, while allowing you to manipulate many of these items (see figure 23.1).



**Figure 23.1** FxSpy allows you to browse and inspect components and their properties.

This tool is particularly handy when you want to tinker with properties that affect the visual aspects of the application. This can save you time compared to making changes in the code and having to recompile constantly to test those changes.

You can download this free tool at <http://code.google.com/p/fxspy/>.

The tools mentioned so far cater to the front end, but monitoring the back end also becomes vital when you need to know what's going over the wire.

### 23.1.7 *Monitoring network activity*

One of the most challenging things you'll encounter when developing Flex applications is trying to figure out why communications with a back-end data service (such as a web service or remote object) doesn't seem to be working. Yes, you can specify fault handlers to handle exceptions, but that may not be enough.

Flex Builder has a built-in ability to monitor network activity. This is facilitated by adding `<mx:TraceTarget/>` to your application, as shown in listing 23.5.

#### Listing 23.5 Using the TraceTarget component to monitor network activity

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:TraceTarget/>
  ...
  ...
</mx:Application>
```

When you launch your application using the Debug button (see figure 23.2), all network traffic appears in the Console view.

This feature is considered anemic and underpowered. Fortunately, a number of third-party tools do an amazing job of speeding up the debugging process:

- *ServiceCapture*—One of the best tools out there. Not only does it understand Flash’s native AMF Remote object protocol, but it deciphers web service SOAP and JSON-RPC. Its interactive nature lets you see what parameters were passed to remote functions and what the data looked like coming back in real time. Price: \$35 U.S. <http://kevinlangdon.com/serviceCapture/>.
- *Charles Web Debugging Proxy* (CWDP)—A well-polished product loaded with features. It provides protocol support similar to ServiceCapture, but it also does bandwidth throttling to simulate network connection speed, and you can export the results to a CSV file. It’s available on multiple platforms but requires the Sun Java JDK to be installed. Price: \$50 U.S. <http://xk72.com/charles/index.php>.

Moving onto the heavyweight tool of debugging, let’s look at the Debugger that comes with Flex Builder, a feature that came to maturity with Flex 3.

### 23.1.8 Using the Debugger

If you’ve reached a point where you need to see how things are changing and you also need fine-grained control, the Debugger is the way to go. It helps you isolate where issues are by allowing you to step through your code in a controlled manner and see how variables change as the code executes.

The tool works by launching in Debug mode, setting breakpoints to pause execution, then adding variables to watch.

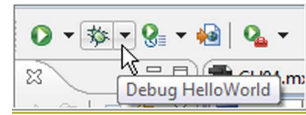
#### LAUNCHING DEBUG MODE

Enabling the Debugger is similar to how you’ve been launching your applications thus far, except in this case you use the Debug button (refer back to figure 23.2).

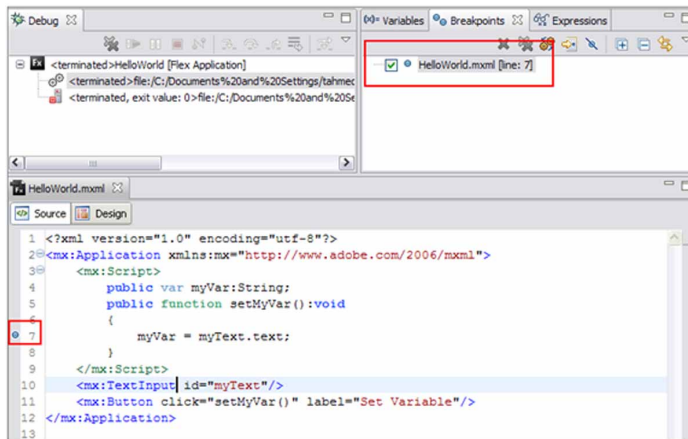
Notice that along with the bin folder, there’s also bin-debug—this is the folder to which Flex Builder publishes a debug version of your application. When you click the Debug button, this version is launched.

If Flex Builder hasn’t done so already, switch to the Debug perspective in Flex Builder by choosing Window > Perspective > Flex Debugging. You’ll see a number of views, including the following:

- *Debug*—Where you are in the application
- *Variables*—An area where you can watch the values of all the variables Flex is managing



**Figure 23.2** You’ll find the Debug button on the Flex Builder toolbar.



**Figure 23.3** Set breakpoints show up in the Breakpoints view.

- *Breakpoints*—Points at which you want the application to stop running so you can pause and evaluate the situation.
- *Expressions*—All the variables you indicated you want to watch

With the Debugger enabled, let's proceed to working with breakpoints.

#### ADDING/REMOVING A BREAKPOINT

You can add a breakpoint anywhere you have ActionScript. For instance, you can add a breakpoint to inline ActionScript in an MXML tag:

```
<mx:Button click="myVar=myText.text"/>
```

You can also place a breakpoint inside an `<mx:Script>` block or in an ActionScript (.as) file.

To do this, follow these steps:

- 1 Using Flex Builder, locate a line of code on which you want to halt execution.
- 2 Right-click in the marker bar (where the line numbers are displayed).
- 3 Select Toggle Breakpoint.

To remove the breakpoint, repeat the same steps.

If all goes well, you should see your breakpoint listed in the Breakpoints view, as shown in figure 23.3.








With the Debugger launched and breakpoints set, you can control the execution of your application.

#### CONTROLLING EXECUTION

After you've set your breakpoints, launch your application in Debug mode. As soon as you hit the line that has a breakpoint, execution will stop and Flex Builder will want your attention.

You can use Debug view to control what happens next by using the functions outlined in table 23.4.

**Table 23.4 Flex Builder's Debugger functions**

Icon	Function	Description
	Resume	Resumes execution of the application
	Suspend	Pauses the application
	Terminate	Terminates the application
	Disconnect	Disconnects the Debugger from the application
	Step Into	Steps into the next line of code
	Step Over	Skips over the next line of code
	Step Return	Continues the current function until it's done

Now that you can control execution, the feature that people generally use the Debugger for is watching variables.

#### **WATCHING VARIABLES**

Watching a variable lets you track the value that the variable is assigned in real time. You can also manually manipulate the value if you want to experiment with how the application would behave using another value.

Whether you've already launched your application or are about to, you can indicate that you want to watch a variable by doing the following:

- 1 Right-click the variable name.
- 2 Select Watch.

Once the variable is added, you should see the variable listed in the Expressions view. Let's try it:

- 1 Set a breakpoint on the line indicated in listing 23.6.

#### **Listing 23.6 Using a breakpoint on the line that sets the variable**

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    public var myVar:String;
    public function setMyVar():void
    {
```

```

        myVar = myText.text;
    }
</mx:Script>
<mx:TextInput id="myText"/>
<mx:Button click="setMyVar()" label="Set Variable"/>
</mx:Application>

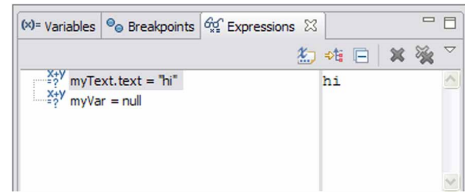
```

← **Set breakpoint here**

- 2 Watch the `myVar` and `myText.text` variables.
- 3 Launch your application in Debug mode.
- 4 Type something in the text box, and click the button. The application will suspend execution.
- 5 Switch to Flex Builder; you should see the variables listed.
- 6 Keep clicking the Step Into button, and watch how the variables change (see figure 23.4).

This process of using a Debugger will be a new experience for those coming from high-level languages like ColdFusion and PHP, because the asynchronous nature of Flex lends itself to the step debugging needed in an event-driven environment. Once you do it a few times, you'll get used to it.

Similar to debugging is the process of testing. Whereas debugging is about fixing problems, testing is about making sure problems don't exist.



**Figure 23.4** Watched variables show their values in the Expressions view.

## 23.2 Testing

It's always a good idea as part of your Systems Development Life Cycle to do as much up-front testing as possible before you go live with the application. ASP, PHP, ColdFusion, and other such languages will tell you the line of code and filename when a crash occurs—this luxury doesn't exist in the land of Flex because of its compiled nature.

After you've finished the debugging phase, testing is a good idea to make sure new features work the way they're supposed to and previous features continue to function. Testing is also an opportunity to fine-tune the application.

### 23.2.1 Types of tests

Three main types of testing are available:

- Profiling
- Unit testing
- Functional testing

Let's define each, starting with profiling.

#### **PROFILING**

*Profiling* is a process whereby you're not testing for proper behavior but rather are looking to see how you can optimize the resource usage of your application. Using

profiling tools, you can identify memory leaks and unnecessary processing overhead and highlight areas that could use optimization.

#### **UNIT TESTING**

*Unit testing* is a way to automate programmatically testing your components, functions, and classes. It's considered a back-end type of testing, because it makes sure other pieces that utilize these back-end pieces receive the proper results and behavior from a programmatic perspective.

#### **FUNCTIONAL TESTING**

*Functional testing* covers the other end of the QA process by focusing on the front-end experience. Functional testing simulates a user running the application and verifies that the behavior and information are accurate. This could be done manually, but that would be prone to human error as well as a potentially huge time investment.

Functional testing works by recording actual use of your application and tracking what was entered and how the application behaved. Each one of these recordings represents a test case for which you've documented the expected results.

It goes beyond checking whether the results shown are right; it also makes sure that various business rules are enforced, that things validate properly, and that the overall experience is the intended result. In addition to automating the testing of new features, you can also use functional testing to verify that existing functionality continues to work properly.

### **23.2.2 Flex Profiler**

New to Flex 3, profiling adds the ability to see where resource bottlenecks (processing and memory utilization) in your application occur so you can isolate and address the issues.

The Profiler works by periodically sampling what's going on in your application. Kind of like time-lapse photography, it takes a snapshot of what things look like at given intervals. In doing so, it's able to determine how many objects currently exist, how much memory they're using, how many functions are being called, and how much time those functions are consuming.

Here's what you should look for:

- *Function call frequency*—How often are you calling functions? Are you making unnecessary calls? See if you can minimize how often heavy-duty functions are called or if they be broken into smaller pieces, only a portion of which are called frequently.
- *Duration of functions*—How long are functions taking to run? Hand in hand with the previous item, you can investigate particular functions to determine the average amount of time a particular function takes to run.
- *Who's calling whom*—Observing this *call stack* and observing the chain of functions calling functions may reveal interesting information.

- *Object count/object allocation*—How many objects have been instantiated? What if you have many instances of the same object? For example, if you can use fewer instances, you'll save on overhead.
- *Object size*—How big are these objects? For objects that have many instances, is their combined size a concern?
- *Garbage collection*—By default, Flex takes care of destroying objects when they're no longer needed. *Garbage collection* is the process of recovering the memory that they once held. It's possible that objects may linger for a while before this process occurs (which causes memory leaks), so you may want to add logic in these cases to explicitly destroy them.

Using the Profiler is similar to using the Debugger: you launch the tool from Flex Builder.

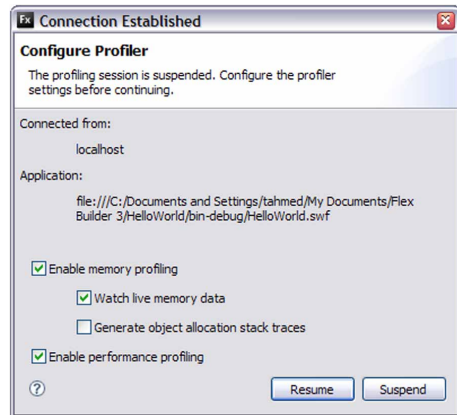
#### LAUNCHING THE PROFILER

You'll find the Profile button on the toolbar in Flex Builder, beside the Debug icon.

If you click it, the window shown in figure 23.5 opens, where you can select what you want to watch.

The settings in the dialog box are:

- *Connected From*—The server you're connected to (or localhost, if it's the same computer).
- *Application*—Path to the Flex application's SWF file.
- *Enable Memory Profiling*—Enables the tracking of memory usage.
- *Watch Live Memory Data*—Memory tracking will occur if Enable Memory Profiling is selected. Watch Live Memory Data indicates whether you want to watch it in the Live Objects view.
- *Generate Object Allocation Stack Traces*—Lets you track how objects are created.
- *Enable Performance Profiling*—Lets you watch where processing time is spent.



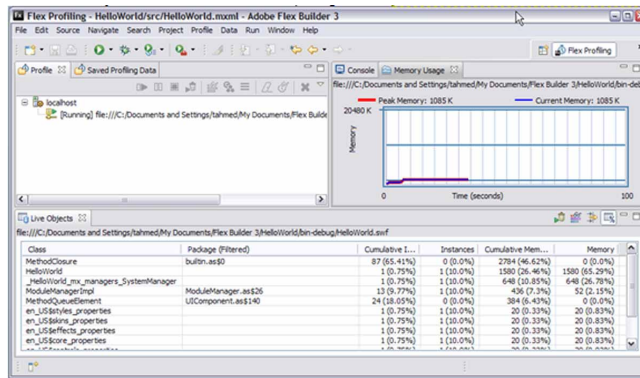
**Figure 23.5** Launching the Profiler prompts you with configuration options.

You could use all of these options all the time; the only difference is how much profiling data is generated and the performance impact while profiling is running.

Launch the Profiler to see it in action, as shown in figure 23.6.

The Profiler functions similarly to the Debugger. After you've launched it, switch back to Flex Builder to control the application's execution.





**Figure 23.6** The Profiler provides aggregate statistics on memory and CPU usage.



**CONTROLLING EXECUTION**

Once the Profiler is running, you'll see your application listed in the Profile view. Select the application, and a bunch of icons will light up and give you options. Table 23.5 describes these options, which are execution-control functionality.

**Table 23.5** Flex Builder's Profiler functions

Icon	Function	Description
	Resume	Resumes execution of the application.
	Suspend	Pauses the application.
	Terminate	Terminates the application.
	Run Garbage Collection	Forces garbage collection to run now versus waiting for it to happen. This recovers memory from objects that are no longer in use.
	Take Memory Snapshot	Takes a snapshot of your memory usage and breaks it down by how that memory is used. Use this feature to take snapshots whenever you suspect major memory operations occur, and later evaluate these snapshots.
	Find Loitering Objects	Goes hand in hand with Take Memory Snapshot in that this feature can compare the differences between snapshots.
	View Allocation Trace	Available after selecting two memory snapshots. Shows which functions were called from one snapshot to another and how much memory was utilized in the process.
	Reset Performance Data	Resets the recorded information.

**Table 23.5 Flex Builder's Profiler functions (continued)**

Icon	Function	Description
	Capture Performance Profile	Used in conjunction with Reset Performance Data. After clicking Reset Performance Data, click this button to snapshot a number of function calls and how long they took to run. You can repeat this process as often as you like.
	Delete	Deletes an item you've selected (for example, a memory snapshot).

The Profiler is the only tool available for Flex that allows this kind of testing, and it does a great job. It gives you deep insight into how much memory and resources are being consumed and where that consumption occurs.

Next we'll discuss unit testing, which automates the testing of objects in order to verify that they behave as expected. The Flex community has produced a number of unit-testing frameworks that can assist you.

### 23.2.3 FlexUnit (unit testing)

If you're familiar with Java, you may have encountered a tool called JUnit—FlexUnit is a Flex spin on that. It's a unit-testing framework for Flex and ActionScript 3, created collaboratively by a few Adobe and non-Adobe developers. This tool is free and is available at <http://opensource.adobe.com/wiki/display/flexunit/>.

You create an application called a *test runner*, which executes ActionScript functions you create to conduct tests on various functions and classes. Although the project has documentation, we'll give you a crash course on it, starting with setting it up.

#### SETTING UP FOR FLEXUNIT

A test runner is a small application that uses FlexUnit's `TestSuite` class to run through all your tests. You could add it to your main project MXML file, but because a GUI component is involved it's easier to make it a simple application on its own. This can be a new project or an auxiliary application to your main one. Follow these steps:

- 1 Download the latest zip file from the FlexUnit site: <http://opensource.adobe.com/wiki/display/flexunit/>.
- 2 Unzip the file to a location such as `c:\flexunit`.
- 3 Open your project in Flex Builder.
- 4 Right-click the project's top-level folder, and select Properties.
- 5 Click the Library Path tab.
- 6 Click the Add SWC button.
- 7 Navigate to the FlexUnit directory, and under `bin`, select the `flexunit.swc` file.
- 8 The next step is to create a test-runner application to execute your tests.

#### CREATING A TEST RUNNER

A test runner can add any number of test-case collections and run through them. FlexUnit calls these test-case collections a *test suite*, which adds a level of conve-

nience when you're working with a lot of test cases. Listing 23.7 shows an example of a test-runner application.

**Listing 23.7** Setting up a FlexUnit test runner

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns=""
    xmlns:flexunit="flexunit.flexui.*" >
    <mx:Script>
        <![CDATA[
            import flexunit.framework.TestSuite;

            private function runTests():void
            {
                var ts:TestSuite = new TestSuite();
                ts.addTest( myTest.suite() );

                testRunner.test = ts;
                testRunner.startTest();
            }

        ]]>
    </mx:Script>
    <mx:Button label="Begin Tests" click="runTests()" />
    <flexunit:TestRunnerBase id="testRunner"
        width="100%" height="100%" />
</mx:Application>
```

**Add FlexUnit namespace**

**Import TestSuite class**

**Call addTest() function as needed to add TestSuites**

**Execute tests**

**Add FlexUnit MXML visual component**

Now all you need to do is create the test cases.

### CREATING A TEST CASE

The last piece involves scripting your test cases by creating ActionScript classes that return a test suite containing all the necessary tests. You can make one big test suite with many test cases, or you can make many test suites with fewer test cases—whatever is easier for you.

In this test-runner example, you invoke a test suite called `myTest`, shown in listing 23.8.

**Listing 23.8** `myTest.as`: Sample FlexUnit test suite with a number of test cases

```
package {

    import flexunit.framework.TestCase;
    import flexunit.framework.TestSuite;

    public class myTest extends TestCase {

        public function myTest( methodName:String ) {
            super( methodName );
        }

        public static function suite():TestSuite {
            var ts:TestSuite = new TestSuite();
        }
    }
}
```

```

        ts.addTest( new myTest( "testMilesToKm" ) );
        ts.addTest( new myTest( "testKmToMiles" ) );
        return ts;
    }

    public function testMilesToKm():void {
        var mph:Number = 65;
        var kmh:Number = speedConverter.toKmh( mph );
        assertTrue( "We want to see 104km/h", kmh == 104 );
    }

    public function testKmToMiles():void {
        var kmh:Number = 104;
        var mph:Number = speedConverter.toMph( kmh );
        assertTrue( "We want to see 65mph", mph==65 );
    }
}
}
}

```

This code tests an ActionScript class called `speedConverter` (see listing 23.9), whose purpose is to convert speed values from km/h to mph and vice versa.

#### Listing 23.9 `speedConverter.as`: the test suite tests this code

```

package
{
    public class speedConverter
    {
        public static function toKmh(i:Number):Number
        {
            return (i*1.6);
        }
        public static function toMph(i:Number):Number
        {
            return (i/1.6);
        }
    }
}

```

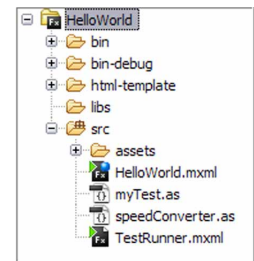
The test runner application has been created, and the test cases are ready to go. All that's left is to execute them.

#### EXECUTING THE TESTS

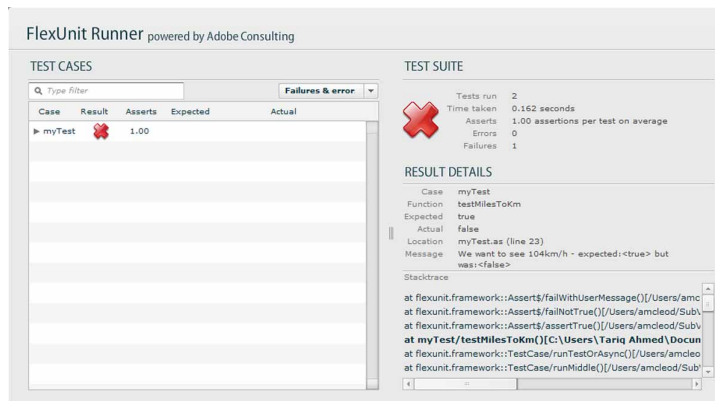
A number of files are involved, so let's recap by looking at figure 23.7.

The test runner won't run automatically because it's not the project's main MXML file. Don't fret! You can easily work around this by loading the `TestRunner.xml` file, then clicking the Run/Launch application button.

In this example, you begin the tests with the click of a button.



**Figure 23.7** Files involved in the FlexUnit test case



**Figure 23.8 FlexUnit's display outputs all issues it encounters.**

But if things don't go well, the items that failed appear on the right with a dump of the diagnostics to go with them (see figure 23.8). Clicking a failure shows a stack trace that describes the situation when the test failed.

Another unit-testing framework available to Flex is Fluint, which works in the same manner.

### 23.2.4 Fluint (unit testing)

Originally known as the oddly named dpUInt (that's not a typo), Fluint is the relative newcomer on the block; it was created by Digital Primates ([www.digitalprimates.net](http://www.digitalprimates.net)) and released as an open source project. It's free and available at <http://code.google.com/p/fluint>.

It's similar to FlexUnit, but it's considered a unit and integration test framework (which explains the name). It goes beyond what FlexUnit can do by adding robust asynchronous support, test sequencing, XML output, test UI components, and Adobe Integrated Runtime (AIR) support.

*Test methods* are the functions that actually perform a test. They're contained in test cases, which in turn are used by test suites.

Like FlexUnit, Fluint has documentation, but we'll briefly run down how to use it.

#### SETTING UP FOR FLUINT

Fluint also uses a test runner to execute tests—you can make a separate project to facilitate this or run it as a secondary application to your main project. In this example, you'll run it from an existing project:

- 1 Download the latest SWC file from <http://code.google.com/p/fluint/downloads>.
- 2 Open your project in Flex Builder.
- 3 Right-click the project's top-level folder, and select Properties.
- 4 Click the Library Path tab.
- 5 Click the Add SWC button.
- 6 Navigate to the directory to which you downloaded the SWC file, and select the file.

With Fluint installed, let's look at how to create the test runner.

### CREATING A TEST RUNNER

The test runner is a mini Flex application that loads all the test cases and runs through them. Here are the steps to create Fluint test runner:

- 1 Download the sample test runner from <http://code.google.com/p/Fluint/downloads/list>.
- 2 Save the `SampleTestRunnerFlex.mxml` file to your project's folder.
- 3 Open it, and modify the line that calls `suiteArray.push()`, as shown in listing 23.10.

#### Listing 23.10 Setting up a Fluint test runner

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:fluint="http://www.digitalprimates.net/2008/fluint"
  layout="absolute"
  creationComplete="startTestProcess(event)"
  width="100%" height="100%">

  <mx:Script>
    <![CDATA[
      import
      net.digitalprimates.fluint.unitTests.frameworkSuite.FrameworkSuite;

      protected function startTestProcess( event:Event ) : void
      {
        var suiteArray:Array = new Array();
        suiteArray.push( new testSuite() );
        testRunner.startTests( suiteArray );
      }
    ]]>
  </mx:Script>

  <fluint:TestResultDisplay width="100%" height="100%" />
  <fluint:TestRunner id="testRunner"/>
</mx:Application>
```

← Add test suites

Now the test runner needs the actual tests to execute against.

### CREATING TEST CASES

You can create as many test-case files as you want; these are ActionScript classes, each of which has functions/methods that perform some kind of test. Each function's name must start with lowercase test.

Fluint automatically searches for all functions that start with the word test, so you don't need to explicitly call these functions yourself. Listing 23.11 shows an example test case that tests the `speedConverter` class from the `FlexUnit` example.

**Listing 23.11** testCase.as: Creating Fluint test cases

```
package
{
    import net.digitalprimates.fluint.tests.TestCase;

    public class testCase
    extends net.digitalprimates.fluint.tests.TestCase
    {
        public function testMilesToKm():void
        {
            var mph:Number = 65;
            var kmh:Number = speedConverter.toKmh( mph );
            assertEquals( 104,kmh);
        }

        public function testKmToMiles():void
        {
            var kmh:Number =104;
            var mph:Number = speedConverter.toMph( kmh );
            assertEquals( 65,mph);
        }
    }
}
```

← **Create all test cases**

You have the test runner, which is the foundation; you have the test cases coded; and now you just need to add the test cases to test suites.

#### CREATING TEST SUITES

A Fluint test suite does one simple thing: it adds all the test cases that are needed as part of the test suite. It too is an ActionScript class.

If you look back at listing 23.10, when creating the test runner, you added a test suite called testSuite:

```
suiteArray.push( new testSuite() );
```

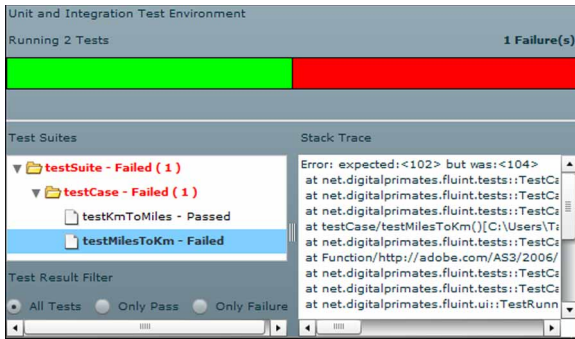
That file looks something like listing 23.12.

**Listing 23.12** testSuite.as: Adding test cases to the Fluint test suite

```
package
{
    import net.digitalprimates.fluint.tests.TestSuite;

    public class testSuite
    extends net.digitalprimates.fluint.tests.TestSuite
    {
        public function testSuite()
        {
            addTestCase( new testCase() );
        }
    }
}
```

← **Add all cases to test**



**Figure 23.9** Fluint’s output highlights cases that passed and failed.

Now you’re ready to rock. Test cases are grouped into test suites, and the test runner executes the test suites.

### EXECUTING THE TESTS

A number of files and pieces were mentioned in setting up this example.

All you have to do is launch the `SampleTestRunnerFlex` application. Its output is shown in figure 23.9.

At the top is a color chart representing each test—green for pass and red for fail. On the left is a folder structure of your test suites and each of their test cases. On the right is diagnostic information about any failures. And at the bottom left is a filter to display only passes, only failures, or all the tests.

We’ve covered profiling and unit testing so far, which help you evaluate the inner workings of your application. The last type of test we’ll discuss is functional testing, which focuses on testing the interface from a user’s perspective.

### 23.2.5 *FunFX (functional testing)*

FunFX is an open source initiative to make functional testing available to all. It’s closer to a framework (which lets people make test-automation tools) than a functional test automation tool in itself. It’s free; the initiative’s website is <http://funfx.rubyforge.org/>, and you can download FunFX from <http://files.rubyforge.com/bytemark.co.uk/funfx/>.

FunFX has the potential to save you time in your functional testing endeavors. It’s written in Ruby for its scripting abilities, but buyer beware—this is a new project, so it isn’t a simple plug-and-play situation.

### 23.2.6 *RIATest (functional testing)*

Considering the competition, whose prices are in the thousands, RIATest is a low-cost solution that can save you a lot of time in your QA cycle. The product works by embedding an agent during compile time or by using a runtime loader. It costs \$499 U.S. and is available at <http://www.riatest.com>.

We like its simple and straightforward approach:



- Compile your application with the `RIATest` module embedded in it, or use its loader feature to dynamically wrap the agent around your application.
- Record your dry runs through the application (optionally, you can hand-write the test scripts).
- Play back your scripts to verify the results.

Key features include:

- An Action Recorder simplifies the process of recording actions into a human-readable script.
- Syntax highlighting makes those human-readable scripts even easier to read via color coding.
- The engine comes with its own scripting language, which is modeled after `ActionScript`. This gives you a lot of control while using a language you already know.
- A Component Inspector allows you to mouse over a UI component and inspect its properties.
- Built-in script debugging helps you figure out problems with your test scripts.

`RIATest` has a lot of momentum behind it. What's interesting is that it has a public mechanism for submitting enhancement requests and bug reports—making it community driven.

### **23.2.7 HP QuickTest Pro (functional testing)**

QuickTest Pro (a.k.a. QTP) is an industrial-strength product from HP (via acquisition of Mercury Interactive). It's powerful and versatile—hence the price tag: \$6,000–\$9,000 U.S. (pricing may vary depending on licensing options). Go to <http://www.hp.com> and search for *QuickTest Professional*. QTP was the first functional-testing tool available for Flex, and for a long time it was the only choice.

Some key features are:

- Defect replication
- Keyword-driven test case development
- The ability to update multiple test scripts quickly using a shared object repository
- Easy to use, with minimal training required
- Simple interface

Test cases are in the context of business workflows (versus a purely technical perspective), which makes QTP easy for QA users to relate to.

Test-case creation is advanced: for example, contextual and conditional test cases can go down various branches depending on the results; you can query your database to find data that is in a current state, then perform tests on that data; and so on.

### **23.2.8 IBM Rational Functional Tester (functional testing)**

Not many Flex developers are aware that Rational Functional Tester exists, and in particular that it supports Flex. This Enterprise-class product is specifically aimed at the Java and .NET communities, and it supports testing of all sorts of applications from standalone to the web-based variety. It costs \$5,400–\$10,500 (pricing may vary depending on licensing options) and is available via <http://www-306.ibm.com/software/awd-tools/tester/functional/index.html>.

Some key features are:

- Data-driven as well as keyword-driven test cases
- Supports test scripts written in many languages including Java and Visual Basic
- Version control
- Dynamic data validation

The Rational Functional Tester is a world-class product that has a world-class price. It's an extremely robust piece of software that even allows for test automation of dumb terminal clients, all the way to Siebel and SAP application testing.

## **23.3 Summary**

In Flex's relatively short life, testing and debugging used to be weak points, but the ecosystem has come a long way in a short amount of time with a variety of tooling and advanced capabilities.

When it comes to debugging, you can do simple things to get insight into what's going on. In addition, the Flex Debugger is a robust tool that gives you tactile control over isolating issues.

Ancillary to debugging is profiling, and the Flex Profiler (new to Flex 3) gives you an advanced tool for understanding the internals of your application. Use it to find out how your application is consuming memory and processing resources.

Making sure new bugs aren't introduced in the first place is what testing is all about. Testing tools range from free, open source projects to pricey, enterprise-strength tools from commercial vendors.

After you've tested and debugged your software, the last stage in the development process is polishing and wrapping up the project. The next chapter teaches you how to prepare for deployment.

# FLEX 3 IN ACTION

Tariq Ahmed with Jon Hirshi and Faisal Abid

FOREWORD BY Ryan Stewart, Adobe Community Evangelist



Adobe's Flex 3 is a complete platform for rich web development. It combines an easy-to-use development environment (Flex Builder) with an elegant programming model based on JavaScript. Flex 3 provides enterprise-quality data and server components that integrate with Java, PHP, and Rails. And now that the major components of Flex are open-source, the cost is right: free!

*Flex 3 in Action* starts with a concise overview of Flex 3 and of ActionScript. It then introduces every significant Flex component in a friendly and highly pragmatic way. The well constructed and well explained examples focus your attention on key properties, methods, and events related to each tag or class. The book covers events in Flex and provides tips on debugging event logic. Readers will find their knowledge of JavaScript or ActionScript to be helpful, but not required. And no previous experience with Flex is assumed.

## What's Inside

- How to build on your existing web development skills
- Interactive forms, drag-and-drop, data-driven features
- New Flex 3 features like the Profiler, AdvancedDataGrid, and Refactoring.
- How to share code across multiple projects

## About the Authors

A Manager of Product Development at Amcom Technology, Tariq Ahmed is well known as the creator of the Community Flex (CFLEX.Net) site. Tariq and coauthor Jon Hirschi, an Adobe Flex Community Expert, initiated numerous Flex-based projects at eBay. Faisal Abid has worked for Buzzspot and RazorCom.

For online access to the authors, code samples, a free ebook, and a free ebook of the FLEX 4 edition go to [www.manning.com/FLEX3inAction](http://www.manning.com/FLEX3inAction)

"The code examples are its strength—plentiful for almost every topic in the book."

—Andrew Grothe  
Triware Technologies, Inc.

"Easy enough for the newbie, detailed enough for the veteran."

—Ken Brueck  
Move Networks

"It's my user-friendly tutorial and reference."

—Christophe Bunn  
Kitry S.A.S.

"An impressive amount of Flex content in a single volume."

—Charlie Grier  
Amcom Technology

ISBN-13: 978-1933988740  
ISBN-10: 1933988746



54999



MANNING

US/CAN \$49.99 [INCLUDING EBOOK]