



**MEAP Edition
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=455>

Part 1: Getting started

Introduction

- 1. Beginning with Java**
- 2. Getting Rich with Flex**

Part 2: Integrating with the server side

- 3. Connecting to Web Services**
- 4. Refactoring to Mate**
- 5. Leveraging BlazeDS**
- 6. Flex messaging on JMS**

Part 3: Advanced Flex topics

- 7. Securing your Flex application**
- 8. Creating custom Flex controls with Degrafa**
- 9. Desktop 2.0 with Adobe AIR**
- 10. Flex on Grails**

2

Getting Rich with Flex

In this chapter

- Create a Flex project using archetype
- Create a Flex front-end for the sample application

In the last chapter we introduced you to AppFuse and created our sample issue tracking application. Now it's time to begin the task of creating the Flex front-end for our sample application. We'll start off by incrementally building up the view layer introducing you to a few of the pertinent concepts of Flex as we go. This chapter is not meant to be a comprehensive guide to the Flex framework by any stretch of the imagination, however you should still be able to follow along without too much trouble. If you want a more in depth look at the Flex framework check out the title Flex in Action.

2.1 Generate the application structure

The first thing we need to do before we get started is to create our Flex application. Since we'll be using the Flex Mojos Maven plugin we'll be creating the application in a similar manner as we did for the AppFuse portion of the application. We've taken the liberty to create a Maven archetype to minimize the amount of manual work we would need to do in order to create the project structure.

FNA (FNA is Not AppFuse)

Some folks at Adobe Consulting have started a new project up at Google Code called FNA or FNA is Not AppFuse (<http://code.google.com/p/fna/>). The FNA project has similar goals to that of AppFuse in that they are making an attempt at creating a framework that enables developers a way to jumpstart their RIA applications with Flex and Java. We

have decided against using this framework for this book, but feel like this project does have potential and warrants a look at if you are starting a new project.

So let's get started shall we. Open up a command prompt and navigate to the root directory of our application. Enter the following command to create our Flex application.

```
$ mvn archetype-create -DarchetypeGroupId=org.foj \
-DarchetypeArtifactId=flex-mojos-archetype \
-DarchetypeVersion=1.0-SNAPSHOT \
-DgroupId=org.foj \
-DartifactId=flex-bugs-ria \
-DremoteRepositories=http://flexonjava.googlecode.com/svn/repository
```

This will create a Flex project that slightly resembles a standard Maven project. Since this is not a Java project, there is a slight variation on the project structure. The sources for our Flex application will go in the `src/main/flex` folder and the tests in `src/test/flex` folder. Maven will also modify our main project `pom.xml` and add this project as a module as shown in Listing 2.1.

Listing 2.1 Parent pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
...
<modules>
<module>flex-bugs-ria</module>
<module>flex-bugs-web</module>
</modules>
...
</project>
```

Now that our project has been created we need to configure the Flex Mojos plugins for both the Flex project and the AppFuse project.

2.2 Configuration for the flex-bugs-ria module

For this application we chose to use the Flex Mojos plugin to leverage the powerful dependency management facilities of Maven as well as avoiding having to write yet another Ant build script.

Listing 2.2 The pom.xml for our Flex application

```
<?xml version="1.0"?>
<project>
<parent>
<artifactId>flex-bugs</artifactId> #1
<groupId>org.foj</groupId> #1
<version>1.0-SNAPSHOT</version> #1
</parent> #1
<modelVersion>4.0.0</modelVersion>
<groupId>org.foj</groupId> #2
<artifactId>flex-bugs-ria</artifactId> #3
```

```

<packaging>swf</packaging> #4
<version>1.0-SNAPSHOT</version> #5

<properties> #6
  <flexmojos.version>3.2.0</flexmojos.version> #6
</properties> #6

<build>
  <sourceDirectory>src/main/flex</sourceDirectory> #7
  <testSourceDirectory>src/test/flex</testSourceDirectory> #7

  <finalName>flex-bugs-ria</finalName> #8
  <plugins> #9
    <plugin> #9
      <groupId>org.sonatype.flexmojos</groupId> #9
      <artifactId>flexmojos-maven-plugin</artifactId> #9
      <version>${flexmojos.version}</version> #9
      <extensions>>true</extensions>
      <configuration>
        <targetPlayer>10.0.0</targetPlayer>
        <locales>
          <locale>en_US</locale>
        </locales>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>com.adobe.flex</groupId>
          <artifactId>compiler</artifactId>
          <version>4.0.0.7219</version>
          <type>pom</type>
        </dependency>
      </dependencies> #9
    </plugin> #9
  </plugins> #9
</build>

<repositories> #10
  <repository> #10
    <id>flexmojos-repository</id> #10
    <url>http://repository.sonatype.org/content/groups/public/</url> #10
  </repository> #10
</repositories> #10

<pluginRepositories> #10
  <pluginRepository> #10
    <id>flexmojos-repository</id> #10
    <url>http://repository.sonatype.org/content/groups/public/</url> #10
  </pluginRepository> #10
</pluginRepositories> #10

<dependencies> #11
  <dependency>
    <groupId>com.adobe.flex.framework</groupId>
    <artifactId>flex-framework</artifactId>
    <version>4.0.0.7219</version>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>com.adobe.flex.framework</groupId>

```

```

    <artifactId>playerglobal</artifactId>
    <version>4.0.0.7219</version>
    <classifier>10</classifier>
    <type>swc</type>
  </dependency>
  <dependency>
    <groupId>org.sonatype.flexmojos</groupId>
    <artifactId>flexmojos-unitytest-support</artifactId>
    <version>${flexmojos.version}</version>
    <type>swc</type>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

#11

- 1 Define the parent pom
- 2 This module's groupId
- 3 This module's artifactId
- 4 Packaging type
- 5 This module's version
- 6 The Flex-Mojos version
- 7 Specify the source and test source directories
- 8 The final name for our artifact
- 9 The flex-mojos-plugin
- 10 The Flex Mojos repository at Sonatype
- 11 The Flex and unit testing dependencies

Listing 2.2 shows the resulting `pom.xml` for our Flex application after we generate the project using the `mvn archetype:create` command shown in section 2.1. Since this is a part of a multi-module project, the `pom.xml` lists the parent module's pom as being the parent for this project (#1). Next you'll see the values that you specified for the `groupId` (#2) and `artifactId` (#3) defined, as well as the packaging type (#4) of `swf` since this project is our Flex application and will be compiled to an `swf` file.

NOTE: FILE ASSOCIATION

You may need to associate the SWF file with the Standalone Flash Player or else your build may time out trying to run the flexunit tests. Because flexunit requires the Flash runtime in order to run its test suites, Maven will try to execute the resulting SWF file for your test suite using the default application for SWF files.

Next comes the project's version (#5), which is set to `1.0-SNAPSHOT`. The archetype also defines a common property (#6) for the Flex Mojos version so that we can be sure that the plugin (#9) and any dependencies (#11) defined for the Flex Mojos are using the same version. We're also overriding the version for the Flex compiler here to compile it with the Flex 4 compiler and target version of Flash Player. Since this is not your typical Maven project, the `pom.xml` defines the source and test-source directory locations (#7). The `pom.xml` also defines the repository and plugin repository locations (#10) for the Flex Mojos plugins and dependencies since they don't exist in the central Maven repository.

2.3 Configuring Maven for the flex-bugs-web module

Now that we've taken care of configuring Maven to build our Flex application, we need to make some minor modifications to the `pom.xml` for the `flex-bugs-web` module in order to get our Flex application to be copied over to the appropriate place in our web application. In order to accomplish this task we'll make use of the `maven-dependency-plugin`.

Listing 2.3 Configuring the maven-dependency-plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId> #1
        <executions>
          <execution>
            <id>unpack-config</id> #2
            <goals> #2
              <goal>unpack-dependencies</goal> #2
            </goals> #2
            <phase>generate-resources</phase> #2
            <configuration>
              <outputDirectory>
                ${project.build.directory}/${project.build.finalName}/WEB-INF/flex #3
              </outputDirectory> #3
              <includeGroupIds>${project.groupId}</includeGroupIds> #3
              <includeClassifiers>resources</includeClassifiers> #3
              <excludeTransitive>>true</excludeTransitive> #3
              <excludeTypes>jar,swf</excludeTypes> #3
            </configuration> #3
          </execution>
          <execution> #4
            <id>copy-swf</id> #4
            <phase>process-classes</phase> #4
            <goals> #4
              <goal>copy-dependencies</goal> #4
            </goals> #4
            <configuration> #5
              <stripVersion>>true</stripVersion> #5
              <outputDirectory>
                ${project.build.directory}/${project.build.finalName} #5
              </outputDirectory> #5
              <includeTypes>swf</includeTypes> #5
            </configuration> #5
          </execution> #5
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

```
...  
</project>  
1. The maven-dependency-plugin  
2. The unpack-config execution  
3. The configuration for the unpack-config execution  
4. The copy-swf execution  
5. The configuration for the copy-swf execution
```

Listing 2.3 shows the plugin configuration (#1) that is needed in order for our web application to properly resolve the dependency for our Flex application and ensure that the SWF file gets placed in the proper place.

First we define an execution that we'll call "unpack-config" (#2) and tell Maven to execute this during the "generate-resources" phase of the build, and call the "unpack-dependencies" goal on this plugin. Then in the configuration (#3) we tell the plugin to limit the scope of what gets affected by this execution to artifacts with the same groupId as our project and only artifacts of type resources. This will get utilized later as we create a common project for all of the configuration files for BlazeDS that need to be shared between both the web application and the Flex application.

The Maven build lifecycle

Maven follows the Convention over Configuration paradigm in many aspects, and the build lifecycle is one of them. The folks who designed Maven realized that there are many common steps in every build and developed the concept of the build lifecycle around it. The default lifecycle flows through the following build phases (in order).

```
validate  
compile  
test  
package  
integration-test  
verify  
install  
deploy
```

For more information on Maven and the build lifecycle you can read the Maven Definitive Guide, a free e-book written by Eric Redmond available at <http://www.sonatype.com/books/maven-book/reference/>.

The second execution that we define (#4) is named "copy-swf", and it will do exactly that. We configure this execution to run during the "process-classes" phase of the build lifecycle and execute the "copy-dependencies" goal to copy the SWF file from our Flex

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=455>

project into the `target` folder in order to be placed in the proper location before Maven creates the final WAR file (#5). Next let's take a look at how to create an HTML wrapper, or in this case a JSP wrapper for our Flex application.

2.4 Adding a wrapper for our SWF

Adobe provides many different templates for you to follow for creating HTML wrapper files for your SWF, located in the `/templates` directory of your Flex SDK installation. They include everything from the very basic no frills wrapper, to wrappers that include functionality to detect whether or not the client has the correct version of the Flash Player installed and whether or not to support deep linking and history for your application.

NOTE

Normally the HTML wrapper would be a HTML file in your web application, however since the Sitemesh filter in AppFuse is configured to decorate anything with a `.html` extension, we decided the easiest way to circumvent this filter was to make the wrapper a JSP file.

For this application copy the contents of the `client-side-detection-with-history` folder including the `index.template.html`, `AC_OETags.js` file and the `history` folder from the `/templates` directory of your Flex SDK installation to the `src/main/webapp` directory of the `flex-bugs-web` project. Rename the `index.template.html` file to `flexbugs.jsp`, and replace the placeholders in the file with the values shown in Listing 2.4.

Listing 2.4 HTML Wrapper values

```
{title} -> FlexBugs
{version_major} -> 9
{version_minor} -> 0
{required_revision} -> 28
{width} -> 100%
{height} -> 100%
{application} -> flex-bugs-ria
{bgcolor} -> #869ca7
{swf} -> flex-bugs-ria.swf
```

We aren't going to go into detail about what is contained in the `flexbugs.jsp` since it's likely you won't have to change anything inside of it in the future. You can learn more about the various HTML templates that Flex provides in the LiveDocs at Adobe's website at http://livedocs.adobe.com/flex/3/html/wrapper_04.html#178239.

2.5 "Hello World" in Flex

Now that we have the web application configured to properly resolve our Flex application dependency, place the resulting SWF file in the proper place and have our HTML wrapper configured, let's write a quick "Hello World!" application in Flex just to verify that everything is working as expected. In the `src/main/flex` folder of the `flex-bugs-ria` project create a `Main.mxml` file for our Flex application.

Listing 2.5 Hello World! in Flex

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:view="org.foj.view.*"
  minWidth="950"
  minHeight="600"
  height="100%"
  width="100%">                                     #1

  <s:layout>
    <s:VerticalLayout/>                               #2
  </s:layout>

  <s:SimpleText text="Hello World!"/>                 #3
</s:Application>

```

1. The Flex Application root component**2. Setting the layout****3. A SimpleText component**

Listing 2.5 shows a very minimal "Hello World!" application in Flex. It only consists of the root Application component (#1), a layout definition (#2) and a single SimpleText component (#3) which has its text property set to "Hello World!"

NOTE: MAVEN AND HEAP SPACE

You may run into heap space problems when compiling your Flex application with the Flex Mojos. If your build fails with a message similar to the following:

```

[INFO] -----
[ERROR] FATAL ERROR
[INFO] -----
[INFO] Java heap space
[INFO] -----
[INFO] Trace
java.lang.OutOfMemoryError: Java heap space

```

In order to fix this issue all you need to do is define an environment variable named MAVEN_OPTS and set its value to "-Xmx512m" adjusting the memory size as needed.

To build our Flex application you first need to run `mvn install` from the `flex-bugs-ria` directory. That's it. This will build the `flex-bugs-ria` project, and then deploy the resulting artifact into your local Maven repository so that the `flex-bugs-web` project can include it as a dependency in its `pom.xml`. Once it is finished building you can navigate to the `flex-bugs-web` directory and run the application by typing `mvn jetty:run-war` on the command line. This will start up a Jetty instance and deploy your war inside this instance so that you can visually verify everything is working. Once you see the output shown in Listing 2.6 your application is now running.

Listing 2.6 Console output from the maven-jetty-plugin

```
...
2009-03-28 19:31:14.206::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 3 seconds.
```

Open up your favorite browser and navigate to `http://localhost:8080/flexbugs.jsp` and you should see a screen similar to Figure 2.1.

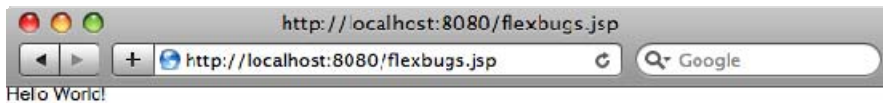


Figure 2.1 Our "Hello World!" application

Not terribly exciting, however it's nice sanity check to see that our application is configured correctly. Now that we have the obligatory "Hello World" application out of the way, let's get on with the task of developing the real application.

2.6 Developing the FlexBugs application

In order to begin developing our application, we should at least have some sort of idea of what we would like it to look like. Figure 2.2 shows a mockup of what our application should look like.

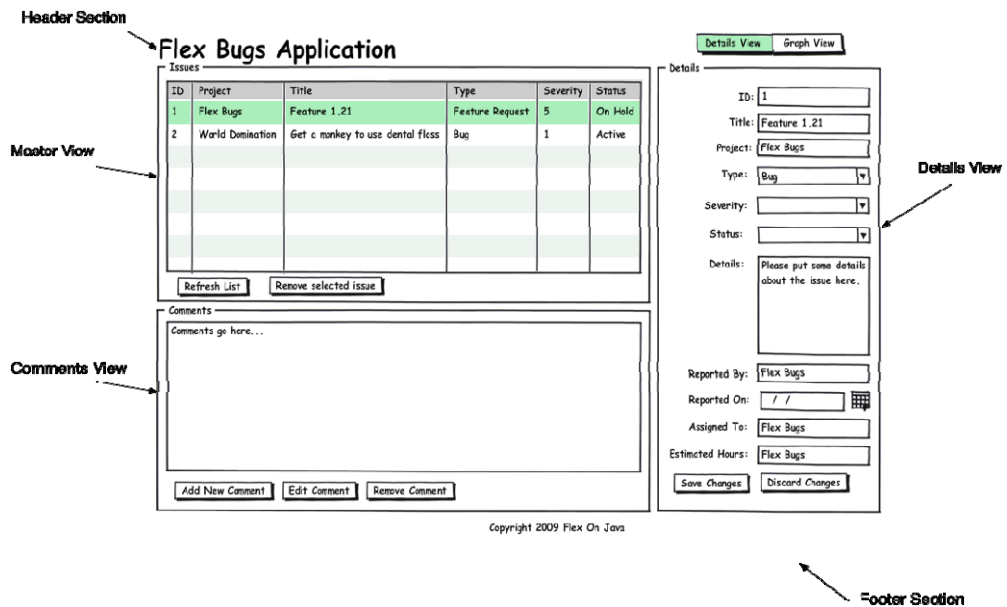


Figure 2.2 - A mockup of the Flex Bugs application

The application is broken up into three main areas of concern in a sort of modified Master/Detail view with a second detail view for any comments that exist for the selected issue. Our application can also be broken up into a header, footer and main application area. Defining our application in these terms achieves a couple of different objectives. First, by separating our application into these different pieces, we can create separate MXML files to help keep our code manageable. Another benefit is that by breaking up our application into separate MXML files, we have the ability to reuse parts of our application if the need arises.

Next we're going to disseminate our application into manageable chunks and we'll start by introducing some of the container and navigation components we'll be using to build this application starting with the `ViewStack` navigation component.

2.6.1 Introducing the `ViewStack`

The `ViewStack` component is a navigation component that allows you to stack up a collection of views and selectively display them. Unlike traditional web applications, Flex applications typically don't have many pages that make up the application. A Flex application typically has one application that will change its view state depending on which part of the application is active. The `ViewStack` is one of the Flex components that allows you to do this by bringing the active view to the foreground and hiding the inactive views in the background as shown in Figure 2.3.

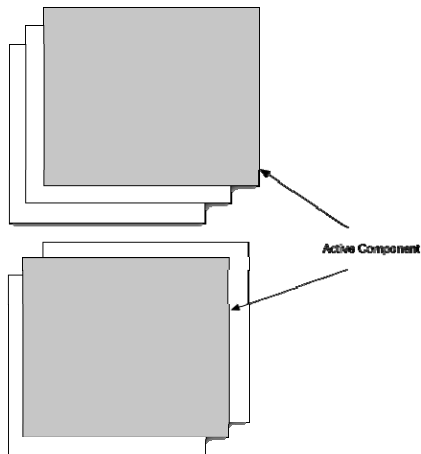


Figure 2.3 How the ViewStack works

You can control the ViewStack in a number of ways; the most common is to utilize one of the navigation components such as the `LinkBar`, `ButtonBar`, or `ToggleButtonBar`. For the FlexBugs application we'll leverage the `ToggleButtonBar` to facilitate switching the view state. In the top right corner of our mockup you'll see two buttons labeled "Details View" and "Graph View", these are the two views that we'll make use of in this application. We'll develop the "Details View" in this chapter and come back to develop the "Graph View" in Chapter 10 when we talk more about graphing components.

Listing 2.7 Defining the ViewStacks

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  minWidth="950"
  minHeight="600"
  height="100%"
  width="100%">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <mx:ViewStack id="mainViewStack" width="100%" height="100%"> #1
    <mx:Canvas id="view1" label="Details View"> #2
      <mx:Text text="Put the details view stuff here..."/> #2
    </mx:Canvas> #2

    <mx:Canvas id="view2" label="Graph View"> #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=455>

```

        <mx:Text text="Put some graphs here..." />           #3
    </mx:Canvas>                                           #3
</mx:ViewStack>

</s:Application>
1. Define our view stack
2. Add the first view component
3. Add the second view component

```

Listing 2.7 shows our Main.mxml after we add the ViewStack components to our application. After we remove our HelloWorld code, create a ViewStack element (#1) and give it an id of "mainViewStack", this will become important later as we define our ToggleButtonBar, as the dataProvider attribute for the ToggleButtonBar will be set to this ViewStack component. We want this ViewStack to use up all available horizontal and vertical space so we set its width and height to 100%. Next we add two Canvas components (#2 and #3) to our ViewStack giving them ids of "view1" and "view2" respectively. These two Canvas components will be the two main views our ToggleButtonBar will control. The label attribute of these two components will be what gets displayed as the text of the two ToggleButton controls so we set those to "Details View" and "Graph View", respectively. Inside of these two Canvas containers we simply put some Text components as placeholders, so that we can see how the demonstrate how the ToggleButtonBar will control our two view states in the next section.

2.6.2 HeaderView

Before we go much further with building up the application we need to create our header view so that we can control our view states that we just created. In the src/main/flex folder create the directory structure shown in Figure 2.4 to house our view components.

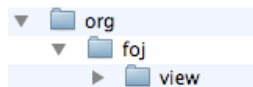


Figure 2.4 Folder structure for view components

ActionScript and Flex follow a similar packaging structure as Java so we will leverage that aspect in order to keep our source files organized similar to how we would do in a Java project. Inside of the view folder create a new file called Header.mxml, where we will put the code for the header for our application.

Listing 2.8 Header.mxml

```

<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  width="100%"
  height="60">

```

#1

```

<s:layout>
  <s:HorizontalLayout/> #2
</s:layout>

<fx:Script>
  <![CDATA[ #3
    import mx.containers.ViewStack;

    public var viewStack:ViewStack;
  ]]>
</fx:Script>
<mx:Spacer width="5"/> #4
<s:SimpleText text="Flex Bugs Application" #5
  height="100%"
  fontSize="32"
  fontWeight="bold"
  verticalAlign="middle"/> #5
<mx:Spacer width="100%"/> #4
<s:VGroup height="100%">
  <mx:Spacer height="100%"/> #4
  <mx:ToggleBarButton dataProvider="viewStack"/> #6
</s:VGroup>
<mx:Spacer width="5"/> #4
</s:Group>

```

1. Component extends Group
2. Define the layout
3. Import and declare a ViewStack member variable
4. Spacers for layout
5. Text field for our Title
6. ToggleBarButton for controlling ViewStack

Listing 2.8 shows the code for our `Header.mxml` component. There's really not a whole lot to it. The component itself extends the `Group` component (#1), and defines its layout (#2) as being `HorizontalLayout`, meaning that all of the components inside of it will be laid out horizontally as opposed to vertically or absolutely. Next we define a public member variable (#3), which will be used to allow our main application to pass in the `ViewStack` that the `ToggleBarButton` (#6) will control. In order to do that we create a `Script` block and enclosing some `ActionScript` inside of a `CDATA` section, so that any characters that may potentially be parsed as XML are handled correctly.

For the Application Title we have a `SimpleText` component (#5) with a couple of attributes defined on it. The first one of these attributes is the `text` attribute, which simply sets the text to be displayed in our application. Flex has support for CSS styles similar to what you may be used to working with in web applications, and we'll make use of this in Chapter XX to change the appearance of our application. Finally there are a few `Spacer` elements (#4), which we use to make sure everything is laid out properly. The `Spacer` elements do just what you would expect, they simply take up space and are used to fill in blank space so that you can effectively lay out components in your application.

Listing 2.9 Adding the Header to Main.mxml

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=455>

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:view="org.foj.view.*" #1
  minWidth="950"
  minHeight="600"
  height="100%"
  width="100%">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <view:Header viewStack="{mainViewStack}"/> #2

  <mx:ViewStack id="mainViewStack" width="100%" height="100%">

    <mx:Canvas id="view1" label="Details View">
      <mx:Text text="Put the details view stuff here..."/>
    </mx:Canvas>

    <mx:Canvas id="view2" label="Graph View">
      <mx:Text text="Put some graphs here..."/>
    </mx:Canvas>
  </mx:ViewStack>

</s:Application>

```

1. added namespace for the view components
2. added header to application

Listing 2.9 now shows our updated `Main.mxml` file that now includes our `Header.mxml` component we just created. First we defined a custom namespace for our view components by adding the code shown at (#1). Next we add our custom component to the `Main.mxml` by using this custom namespace prefix and pass in a reference to the `ViewStack` component by using the binding expression `"{mainViewStack}"` (#2).

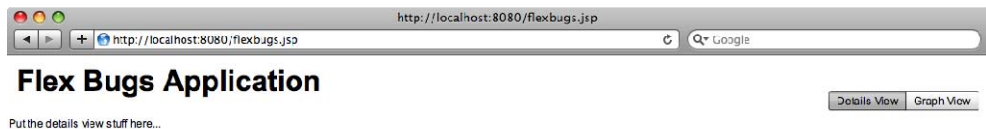


Figure 2.5 The Header view added.

Now you should be able to build and run the application as outlined earlier, and be presented with something that resembles Figure 2.5. When you click on the `ToggleButton`s in the upper right hand corner, you should see the text in the main part of the application change. Next let's build a very simple footer component for our application.

2.6.3 FooterView

Inside the same folder where we just created the `Header.mxml` in the previous section, create another file named `Footer.mxml`. Though it may seem like a little bit of overkill to separate our footer into its own MXML file, we'll do it anyway just to get you in the habit of systematically breaking your Flex application into smaller, more manageable pieces.

Listing 2.10 Footer.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  width="100%"
  height="40">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <mx:Spacer width="100%"/>
  <s:SimpleText text="Copyright © 2009 Flex On Java"
    height="100%"
    verticalAlign="middle"
    textAlign="center"/>
  <mx:Spacer width="100%"/>

</s:Group>
```

The code in Listing 2.10 shows our footer file, and is rather unremarkable. Similar to how we did the header earlier, our footer extends the `Group` component. It only contains a single `SimpleText` component, which contains our copyright information, and sets its `textAlign` attribute to "center". Once again we leverage a couple of `Spacer` elements to assist in layout.

Listing 2.11 Main.mxml updated with footer

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:view="org.foj.view.*"
  minWidth="950"
  minHeight="600"
  height="100%"
  width="100%">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  ...
  </mx:ViewStack>

  <view:Footer/> #1
```

```
</s:Application>
```

1. Added the footer to the Main.mxml

Next we add the footer to our application much like we did earlier for the header. Near the end of the Main.mxml, place the tag for the footer as shown in Listing 2.11. Next let's move on to creating the view component for our master view.

2.6.4 MasterView

Now we start to get to the more interesting parts. The master view if you'll recall from Figure 2.2 consists of a few different components. At the top of the master view reside a data grid component, and two buttons at the bottom of the data grid for adding and removing issues.

Listing 2.12 MasterView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  layout="vertical"
  title="Issues"
  width="100%"
  height="100%"> #1

  <mx:DataGrid id="masterViewDataGrid" width="100%" height="100%"> #2
    <mx:columns>
      <mx:DataGridColumn dataField="id"
        headerText="ID" width="70"/>
      <mx:DataGridColumn dataField="project"
        headerText="Project" width="120"/>
      <mx:DataGridColumn dataField="description"
        headerText="Description"/>
      <mx:DataGridColumn dataField="issue-type"
        headerText="Type" width="120"/>
      <mx:DataGridColumn dataField="severity"
        headerText="Severity" width="70"/>
      <mx:DataGridColumn dataField="status"
        headerText="Status" width="100"/>
    </mx:columns>
  </mx:DataGrid>

  <mx:ControlBar width="100%"> #3
    <mx:Button label="Add New Issue"/>
    <mx:Button label="Remove Selected Issue"/>
  </mx:ControlBar>
</mx:Panel>
```

1. Extending Panel

2. The DataGrid

3. Control Bar for Add/Delete buttons

First create a file inside of the `org/foj/view` folder called `MasterView.mxml`. This will contain the code necessary to create our master view, which is shown in Listing 2.12. This component will be based on the `Panel` component, which is one of the various layout containers available in the Flex framework. The `Panel` component provides us with a title bar area where we can provide the group of components contained within a meaningful title much like the `<legend>` tag in HTML, or a group box control, for those more familiar with desktop development, would do. The next component you'll see in the code listing is the `DataGrid` (#2). The `DataGrid` is a powerful component that is used to display tabular data, and is also one of the fundamental controls used in data-driven applications. The `DataGrid` and its bigger brother the `AdvancedDataGrid` give you the ability to actually edit rows of data within the table cells, but for our application we're not going to leverage that functionality in favor of using a detail view. Lastly we have a `ControlBar` (#3) at the bottom of our `Panel` that will contain the buttons for adding and removing issues from the application.

2.6.5 DetailView

The next view we're going to develop is the detail view, where we will allow the users of the application to modify the data fields for the issues that are displayed in the master view. The details view will contain a form containing the various fields that can be updated for the issues in our application. Begin just as we did earlier for the master view, by creating a file named `DetailView.mxml` in the `org/foj/view` folder. Listing 2.13 shows the code we'll be adding to that file.

Listing 2.13 DetailView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
          xmlns:s="library://ns.adobe.com/flex/spark"
          xmlns:mx="library://ns.adobe.com/flex/halo"
          layout="vertical"
          title="Details"
          width="100%"
          height="100%">                                     #1

  <mx:Form id="issueDetailForm" width="100%">                #2
    <mx:FormItem label="ID:" width="100%">                 #3
      <mx:Text id="issueId"/>
    </mx:FormItem>
    <mx:FormItem label="Project:" width="100%">
      <mx:TextInput id="projectName"
                    width="100%"/>
    </mx:FormItem>
    <mx:FormItem label="Description:" width="100%">
      <mx:TextInput id="issueDescription"
                    width="100%"/>
    </mx:FormItem>
    <mx:FormItem label="Type:" width="100%">
      <mx:ComboBox id="issueType"/>
  </mx:Form>
</mx:Panel>
```

```

</mx:FormItem>
<mx:FormItem label="Severity:" width="100%">
  <mx:ComboBox id="issueSeverity" />
</mx:FormItem>
<mx:FormItem label="Status:" width="100%">
  <mx:ComboBox id="issueStatus" />
</mx:FormItem>
<mx:FormItem label="Details:" width="100%">
  <mx:TextArea id="issueDetails"
    width="100%"
    height="100" />
</mx:FormItem>
<mx:FormItem label="Reported By:" width="100%">
  <mx:TextInput id="issueReportedBy"
    width="100%" />
</mx:FormItem>
<mx:FormItem label="Reported On:" width="100%">
  <mx:DateField id="issueReportedOn"
    width="100%" />
</mx:FormItem>
<mx:FormItem label="Assigned To:" width="100%">
  <mx:TextInput id="issueAssignedTo"
    width="100%" />
</mx:FormItem>
<mx:FormItem label="Estimated Hours:" width="100%">
  <mx:TextInput id="issueEstimatedHours"
    width="100%" />
</mx:FormItem>
</mx:Form>

<mx:ControlBar> #4
  <mx:Button id="saveChangesButton" label="Save Changes" />
  <mx:Button id="cancelChangesButton" label="Cancel Changes" />
</mx:ControlBar>

```

```
</mx:Panel>
```

1. Extends Panel
2. The Form container
3. FormItem components
4. ControlBar

Similar to the master view, the details view component will be based off of the Panel layout container (#1). Next we add in a Form (#2) container to help organize the input fields that will be used to ultimately update the issues in the application. Unlike in HTML, the Form container in the Flex framework serves no other purpose than to group form controls on the page. You do not need to wrap fields in a Form tag in order to submit data to the back end; it's simply there for aesthetics. Inside of the Form we wrap each input field inside of a FormItem (#3) component that provides some styling, a label and layout for each of the form fields. Lastly the detail view contains a ControlBar (#4), which again contains the buttons that control saving the data and cancelling the edits.

2.6.6 CommentsView

The final section of the application we'll layout is the comments view which will eventually list any comments that are added to our issues. First create the `CommentsView.mxml` file in the `org/foj/view` folder just as we did for all of the other view components. Initially the comments view will only contain some simple placeholder containers, but later on in chapter XX we'll be developing a custom item renderer to display the comments as well as let us add new comments.

Listing 2.14 CommentsView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  layout="vertical"
  title="Comments"
  width="100%"
  height="100%">                                     #1

  <mx:List id="commentsList"                            #2
    width="100%"
    height="100%"
    labelField="commentText">

  </mx:List>

  <mx:ControlBar>                                     #3
    <mx:Button id="addButton"                          #4
      label="Add New Comment" />
    <mx:Button id="editButton"                         #4
      label="Edit Comment" />
    <mx:Button id="deleteButton"                      #4
      label="Delete Comment" />
  </mx:ControlBar>

</mx:Panel>
```

1. Extending Panel
2. List for the comments.
3. ControlBar for buttons
4. Buttons for operations on comments

Listing 2.14 shows the code for the comments view. Just like we did in the other main pieces of our application, the comments view is based on the `Panel` component (#1). Next we add a `List` component (#2), which we'll use to contain a list of our comments. At the bottom of the `Panel` we add a `ControlBar` (#3) to hold the 3 `Button` components (#4) we'll define to operate on the comments.

2.7 Laying out the components

Now that we've got all of the components defined, let's add them to the `Main.mxml` and define our overall application layout. In Flex, all components can be laid out within other containers generally in one of three ways, horizontally, vertically, or absolutely. In most

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=455>

cases it will be desirable to go with either horizontal or vertical layouts, especially if you want your application to resize appropriately. In order to achieve the layout you ultimately want you often times will need to nest layout containers within one another as illustrated in Figure 2.6.

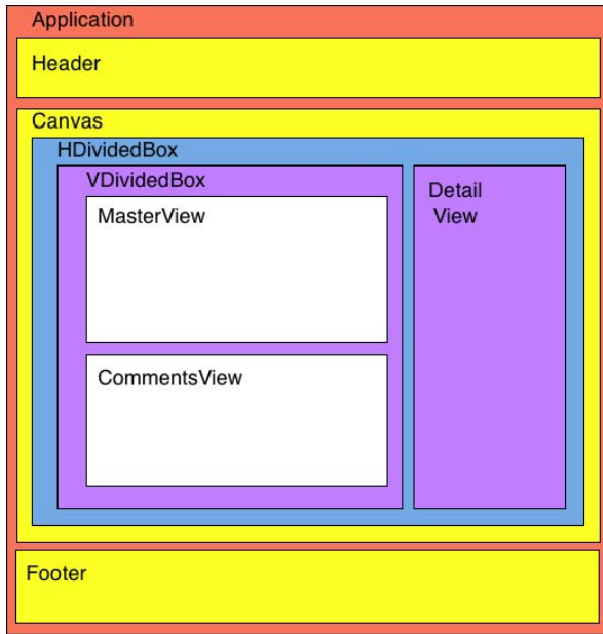


Figure 2.6 The layout for our application

The diagram in Figure 2.6 shows the nesting of layout containers that was necessary in order to duplicate what was shown in our mockup shown in Figure 2.1 earlier. Listing 2.15 shows our updated `Main.mxml` with all of the layout components necessary.

Listing 2.15 Laying out the application

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:view="org.foj.view.*"
  minWidth="950"
  minHeight="600"
  height="100%"
  width="100%">

  <s:layout>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=455>

```

    <s:VerticalLayout/>
</s:layout>

<view:Header viewStack="{mainViewStack}"/>

<mx:ViewStack id="mainViewStack" width="100%" height="100%">

    <mx:Canvas id="view1" label="Details View"> #1
        <mx:HDividedBox width="100%" height="100%"> #2
            <mx:VDividedBox width="70%" height="100%"> #3
                <view:MasterView id="masterView" height="60%"/> #4
                <view:CommentsView id="commentsView" height="40%"/> #5
            </mx:VDividedBox>
            <view:DetailView id="detailsView" width="30%"/> #6
        </mx:HDividedBox>
    </mx:Canvas>

    <mx:Canvas id="view2" label="Graph View">
        <mx:Panel title="Graph View" width="100%" height="100%">
            <mx:Text text="Put some graphs here..."/>
        </mx:Panel>
    </mx:Canvas>
</mx:ViewStack>

<view:Footer/>

</s:Application>

```

Just as was shown in the diagram in Figure 2.6, we start out with a Canvas (#1) container to hold all of the nested layout containers for the main ViewStack. Inside of this we create an HDividedBox (#2) setting its width and height to 100%, meaning that we want it to take up all available space. Within that we place a VDividedBox (#3), which will contain the MasterView (#4) and the CommentsView (#5). Lastly we add the DetailView (#6), which will occupy the right hand side of the HDividedBox. Now our application is almost complete. Next let's create a component which we'll use as a modal popup to edit the comments that are in the List view of our CommentsView component.

2.8 Creating a popup component

The final component we'll develop in this chapter is a modal popup form that we'll wire into the application in the next chapter for adding new comments and editing existing comments.

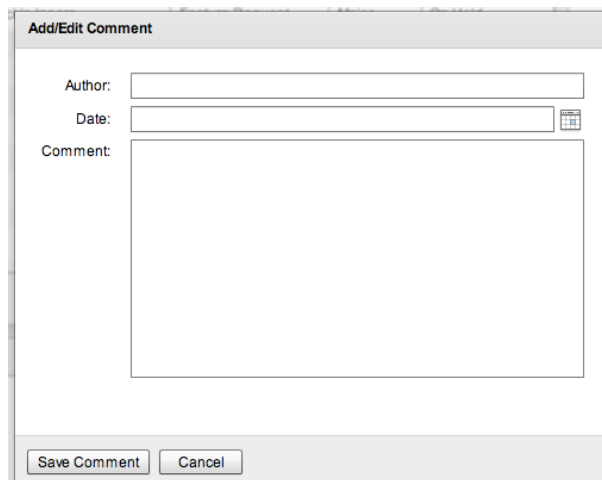


Figure 2.7 Our popup for editing comments

Figure 2.7 is a screenshot showing what our popup will end up looking like. Listing 2.16 shows the code for our popup.

Listing 2.16 EditCommentForm.mxml

```

<?xml version="1.0" ?>
<mx:TitleWindow
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  title="Add/Edit Comment"
  height="400"
  width="500">                                     #1

  <mx:Form width="100%">                             #2
    <mx:FormItem label="Author:" width="100%">
      <mx:TextInput id="author"
        width="100%" />
    </mx:FormItem>
    <mx:FormItem label="Date:" width="100%">
      <mx:DateField id="commentDate"
        width="100%"
        formatString="MM/DD/YYYY" />
    </mx:FormItem>
    <mx:FormItem label="Comment:" width="100%">
      <mx:TextArea id="commentText"
        width="100%"
        height="200" />
    </mx:FormItem>
  </mx:Form>

  <mx:ControlBar>                                   #3

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=455>

```

    <mx:Button id="saveButton" label="Save Comment" />
    <mx:Button id="cancelButton" label="Cancel" />
</mx:ControlBar>

```

```

</mx:TitleWindow>

```

1. Extends TitleWindow
2. Add a Form
3. Control Bar

Our popup window is fairly simple. We define our component to extend the TitleWindow component (#1), which is the component you'll typically use for popups. Next we add a Form and a bunch of FormItems (#2) just like we did earlier for the DetailView. Finally we add a ControlBar (#3) to the window to hold the Button components for controlling the popup.

2.9 The finished application

Well, the application is far from finished, but we're done for now. Now is as good a time as any to build our application and see the progress we've made. The application won't be functional until the end of the next chapter, but it's always reassuring to see what the finished product will look like anyway.

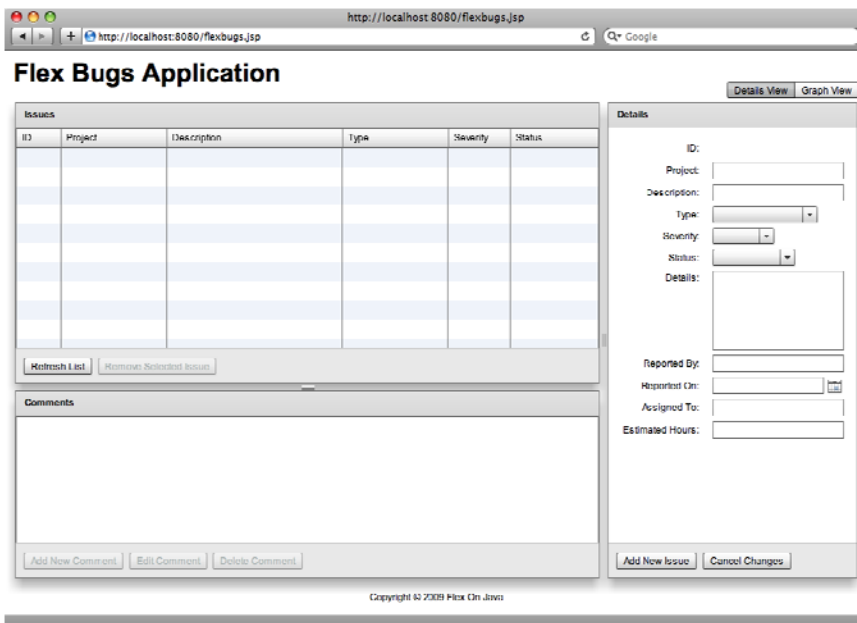


Figure 2.8 The finished application

From the root of the `flex-bugs-ria` project run the command `mvn install`. The build should complete successfully, if not go back through and double-check your code to ensure that it matches the code in the listings. Next inside the root of the `flex-bugs-web` project run the `mvn jetty:run-war` command to start up the embedded Jetty web container and deploy your war file. Once the container is up and running navigate to `http://localhost:8080/flexbugs.jsp` and you should see something that resembles Figure 2.8.

2.9 Summary

This chapter moved quickly and only briefly introduced many of the components available to you in the Flex framework. We started out with an idea of what we wanted our application to look like and decomposed that into several smaller components. By doing that we not only made our code more manageable, but as you'll see in the next chapter, it will also make our presentation models and event handling more manageable. Once we had all of our pieces built up, we were able to illustrate some of the basics of layout containers in Flex and put our application together. We're hopeful that you were able to follow along, however if you feel like you need more information about layouts and the components of the Flex framework, a good place to start would be the LiveDocs at Adobe's site (<http://livedocs.adobe.com/flex/3/html/help.html>).

Now that we've built up our Flex application and layed out our components, where do we go from here? In the next chapter we're going to begin connecting our Flex application to the Java back end we developed in Chapter 1 using the `WebService` component as well as the `HTTPService` component. Later on in Chapter 6 we'll refactor this to use the BlazeDS framework to talk directly to our Java application.