FLEX ON JAVA

Bernerd Allmon
Jeremy Anderson

FOREWORD BY JAMES WARD

MANNING

Flex on Java

BERNERD ALLMON
JEREMY ANDERSON

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
180 Broad St.
Suite 1323
Stamford, CT 06901
Email: orders@manning.com

Manning Publications Co.
180 Broad St.
Suite 1323
Stamford, CT 06901

Development Editors:	Nermina Pascal-Miller
	Sebastian Stirling
Copyeditor:	Betsey Henkels
Typesetters:	Dennis Dalinnik
Cover designer:	Marija Tudor

ISBN 9781933988795
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 16 15 14 13 12 11 10

contents

foreword

Every ten to fifteen years there is a radical software paradigm shift. Many of us experienced the last shift as the web gained momentum and software was rebuilt for a new and game-changing deployment model. Today we are in the midst of another great software paradigm shift. User and business needs now require software to be more usable, extensible, and portable.

Rich Internet Applications (RIAs) represent a new generation of software. As the name implies, RIAs provide users with a rich and interactive experience. At the same time, they offer the ease of deployment that made early web applications so successful. RIAs are the future of software because they combine the strengths of the web deployment model with the full client capabilities of thick-client software.

Since 2004 Adobe Flex has been the most prolific toolkit for building RIAs. A primary advantage of Flex for RIA development is that it integrates easily with any backend technology. By providing native XML, SOAP, and Data Remoting capabilities, Flex enables developers to build rich new UIs on top of existing services. For Java developers this combination is especially compelling because many Java systems have already embraced service-oriented architectures with SOAP Web Services, Spring, or one of numerous other technologies.

The union of Java on the backend and Flex on the frontend is so powerful that hundreds of thousands of developers have already embraced this new paradigm to create better software. You've probably picked up this book because you want to do the same tiing—build better software. We all want to build software that we are proud of—software that users will love. *Flex on Java* will teach you how to do just that.

There are many aspects to building great software. If software looks sexy but does not perform well or is not maintainable, its value is diminished. What I love about this book is that it teaches a holistic approach to building great software with Flex and Java. As you would expect, you will learn how to create rich data visualizations. But, just as importantly, you will learn how to efficiently move data between the client and server using BlazeDS, set up unit tests, add security to an application, and more. These are the problems we have to solve in real software. Knowing how to address these fundamental issues frees us to focus on what matters most—creating software that users will love.

These are exciting times for software developers! Today we are building the next generation of software—a generation that will be remembered as the first to be usable, beautiful, and truly helpful. *Flex on Java* empowers you to create that future! I look forward to seeing the future that YOU build with Flex and Java.

JAMES WARD
TECHNICAL EVANGELIST FOR FLEX AT ADOBE
www.jamesward.com

preface

If you'd asked me a few years ago if I'd ever write a book, I would have laughed at the thought. All through high school and college I loathed writing anything more than a short answer, and when it came to writing papers, I was usually one of the people asking about the minimum length required for a passing grade. Now here we are, thousands of words and hundreds of pages later, and BJ and I have survived writing our first book, twice.

So how did I go from absolutely loathing writing to being willing to dedicate so many nights and weekends to writing this book? Since the first 1.0 release of the Flex framework, I've been a fan. I discovered Flex while I was distaining HTML/JavaScript and browser compatibility issues. I was trying to prototype a form-heavy application with complex business rules and validation, struggling with goofy layout issues and JavaScript errors, and was looking for a better solution. Although it's possible to make rich web applications using HTML and JavaScript, it's easy to make ugly ones. Most of the nice AJAX frameworks we take for granted today didn't exist at the time, and many developers had absolutely no idea what AJAX was.

One night, while searching for an alternative, I ran across this excellent framework that allowed you to write Flash-based applications using a declarative syntax and a prototyping scripting language similar to JavaScript, without the cross browser issues because it all ran in the Flash Player. So I picked up a copy of *Developing Rich Clients with Macromedia Flex* by Steven Webster and Alistair McLeod and immediately fell in love with the Flex framework. There was only one problem: it was expensive. It was going to be a hard sell for any but the largest projects.

Flex effectively dropped off my radar as billable projects took precedence, and I didn't have the time or desire to work on any side projects. Then in 2007 Adobe announced that it would open source the Flex framework and portions of the LiveCycle Data Services server components as an open source project of its own called BlazeDS. When I heard this announcement I figured it was time to start learning Flex again. I had discovered a self-published book called *Flexible Rails*, about integrating Flex with Ruby on Rails, and because I was already learning Ruby on Rails, this book was a good choice. So I purchased the PDF and a few short weeks later the author, Peter Armstrong, announced that Manning Publications was going to publish the book.

Being a Java developer for most of my professional career, I began to think about the lack of good books on integrating Flex with a server-side backend. There was a plethora of Flex books available on the market; however most were written from the perspective of a Flash developer and used techniques that would make any seasoned Java developer cringe. Few discussed connecting to either LiveCycleDS or BlazeDS. So I proposed the idea of writing a book on Flex from a web developer perspective to Michael Stephens at Manning.

The book took many shapes. At one time, we contemplated writing a book on both Java and .NET with Flex; we finally settled on an early version of what you now hold in your hands. The main premise of the book is that you can add a Flex frontend to an existing application. The first version of this book attempted to use an existing open source Java web application as its sample application. When we were about two-thirds of the way through the book, we realized that the sample application wasn't working as intended. We'd planned to have a sample application that would be more than just a throwaway. We wanted the readers to develop an application that would incorporate techniques they could apply to their everyday work, but not something so complex that it would distract readers from what we were trying to accomplish—demonstrating Flex and Java.

After much reflection and discussion on the sample application, we decided to scrap it and use AppFuse, an open source platform for quickly building Java web applications, as the basis for our application. This allowed us to construct a sample application in just one short chapter. AppFuse provided many functions out of the box that we would have otherwise had to spend time discussing and setting up. Unfortunately this also meant that much of what we had already written had to be changed, but I feel this was a necessary change for the better.

Here we are, two years later, with a final product that I can proudly say I helped to write. I hope that you enjoy this book and that it helps you in your journey of integrating Flex into your everyday work.

JEREMY ANDERSON

acknowledgments

This book would not have been possible if not for the hard work of many people.

We'd like to thank everyone at Manning, especially Marjan Bace and Michael Stephens, for giving us the opportunity to write—and rewrite—the book. Without the hard work and dedication of our development editors, Nermina Pascal-Miller and Sebastian Stirling, this book would not have happened. Thanks for your guidance and encouragement. Thanks to Karen Tegtmeyer, our review editor, for arranging the peer reviews and to Stephen Hong, for helping to market and promote the book. The rest of Manning's staff was patient and supportive and provided us with invaluable assistance to help make this book a success, in particular our copyeditor Betsey Henkels and our proofreader Elizabeth Martin.

Special thanks go to Richard Dammkoehler for doing such an excellent job in his technical review of the final manuscript shortly before it went to press.

We'd also like to express our thanks to Michael Kimsal for publishing a two-part article based on the Grails chapter of this book in the excellent *GroovyMag*.

Thank you to Matt Raible for giving us AppFuse, which allowed us to build a sample application in just one short chapter, and for promoting our book in his travels.

Thanks to the Adobe team for giving the Flex framework, BlazeDS, and more to the open source community.

Thanks to Marvin Froeder (also known as Velo) for his hard work in developing the FlexMojos Maven plugin, which we use extensively throughout the book.

Thank you to Giacomo "Peldi" Guilizzoni, founder of the outstanding Balsamiq Mockups tool (a fine example of an Adobe AIR application) for providing us with

licenses. Many of the mockups you see as illustrations throughout this book were created using this tool.

Thanks to the following reviewers who read the manuscript at different stages of its development and contributed invaluable feedback: Jeremy Flowers, Sopan Shewale, Rick Evans, Christophe Bunn, Phil Hanna, Nikolaos Kaintantzis, John Griffin, Doug Warren, Brian Curnow, and Peter Pavlovich. Thanks also to everyone who contributed on the MEAP forum.

Last but not least, thank you to James Ward for contributing the foreword to our book.

Jeremy Anderson

I'd like to thank God for blessing me with the talent necessary to write this book.

Second only to God is my wife Karla, who had the patience to see me through this and keep me on task. To my children, Emily and Isaac, thanks for allowing Daddy to hide in his basement office to write. Without their support, understanding, and sacrifice, this book would not have been possible.

Next, big thanks to my coauthor and partner in crime, BJ. If he hadn't joined me on this venture, there might not have been a *Flex on Java*.

Thanks to everyone at Pillar Technology, especially Gary Gentry, Bob Meyers, Chris Beale, Patrick Welsh, Matt VanVleet, Rich Dammkoehler—and everyone else who provided support.

Thank you to Carl Erickson and everyone at Atomic Object for helping me solidify my interpretation of the Passive View pattern in the sample application.

BJ Allmon

Thanks to my God and Father in heaven who is the author of life. Your love endures forever.

I'm humbled by the patience and the love demonstrated to me by my wonderful bride Sarah and our kids Hannah, Zacharee, Elliot, and Jennessee. Thank you for allowing me to steal precious time away to work on this book. I love you so much!

Without my coauthor Jeremy Anderson's talent and thoughtfulness, I wouldn't have been able to contribute to this project. Thank you Jeremy for your hard work in leading this project!

Others who have helped me along in my journey include Bob Myers, Christopher Judd, Kevin Smith, Charlie Close, Gary Gentry, Richard Dammkoehler, Matt VanVleet, Randy Thomas, and Dan Wiebe; the many talented developers at Pillar Technology Group including Mark Flickinger, Ankur Gupta, Beth Seabloom, and Shawn Steinbrunner; the wonderful staff at Mettler-Toledo; the entire staff at Click4Care; the Delaware City Vineyard, Vineyard Church Delaware County, Vineyard Columbus; and my extended family.

about this book

There are many books available that are purposed for teaching technology topics inside and out. These books are necessary for understanding how to use a technology correctly but many times are not meant to teach you what a normal day of a development would look like using that technology. This book was written to demonstrate practical development with two powerhouse technologies, Flex on Java. It will guide you in building your own applications that scale for real-world business needs, leaving you feeling equipped with the fundamentals that are pertinent to the software feature or task at hand.

Throughout the book, the fundamentals of building testable and rich UIs that communicate with a powerful server side are brought together in bite-sized chunks. The topic of building a robust Flex client that sits on top of a Java server-side application will be discussed throughout as it pertains to the integration of the two and passing data back and forth.

Along with the main topic of integrating Flex with Java, topics such as Maven, Spring integration, adding security and personalization, charting, messaging, AIR desktop applications, logging, continuous integration, AppFuse, and even Flex on Grails will be demonstrated.

Who should read this book

This book is geared toward developers with a need for creating rich applications, on a budget, with Flex 4 and Java. All the tools we use for our examples are open source or free and very proven.

This book assumes familiarity with software development in general, specifically Flex and Java. Though it was written with the intent to teach integration techniques for Flex and Java, not language fundamentals, it was done so to make it easy for even Flex or Java beginners to get rolling quickly with both.

How the book is organized

Flex on Java is made up of three parts:

- Part 1 Getting started
- Part 2 Strengthening the backend
- Part 3 Going above and beyond

We start off by introducing the two technologies and building a sample Java application you can play with. We go on to build a Flex client for the Java application that ties into some Java web services. Part 1 covers the first four chapters.

In chapters 5 and 6 we dive deeper into backend integration with Java on the server side. Part 2 introduces topics that allow Flex to connect to Java through object remoting, logging, and messaging. Using the Spring Framework for Flex integration is very powerful and we demonstrate how that can be done.

Part 3, chapters 7-11, covers topics that are off the beaten path, such as security and personalization, building graphs, desktop development with AIR, unit testing, and building a Flex and Grails application.

Code conventions

All source code in listings or in text is in a `fixed-width font like this` to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing. At times, only the important segments of a code listing are displayed on the page. The source code for all of the examples in full can be downloaded from the publisher's website at www.manning.com/FlexonJava.

Author Online

The purchase of *Flex on Java* includes free access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and other users. To access and subscribe to the forum, point your browser to www.manning.com/FlexonJava. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of partlcipatlon on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questlons, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

JEREMY ANDERSON is a software developer for Pillar Technology Group, an Agile consulting firm in the Michigan and Ohio Valley region. He is a self-proclaimed autodidact, constantly tinkering with cutting edge technologies such as Groovy, Grails, and Flex. He's been developing web-based applications on the JVM in one shape or another for over five years. When he's not sitting behind a keyboard hacking away at code, you can usually find him out on the single-track on his bike or sometimes even on foot. He sometimes has time to update his blog http://blog.code-adept.com.

BJ ALLMON is a software developer for Pillar Technology Group. He enjoys participating in local user groups and conferences and becoming a more seasoned software practitioner. When he is not dabbling in software development he can be found spending time with his family of six and playing the six-string.

about the cover illustration

The figure on the cover of *Flex on Java* is a “Soldier.” The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book’s table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book...two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor did not have on his person the substantial amount of cash that was required for the purchase and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person’s trust in one of us. It recalls something that might have happened a long time ago.

The pictures from the Ottoman collection, like the other illustrations that appear on our covers, bring to life the richness and variety of dress customs of two centuries ago. They recall the sense of isolation and distance of that period—and of every other historic period except our own hyperkinetic present. Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

Part 1

Laying the foundation

Part 1 lays the foundation of *Flex on Java* by touring Flex, Java, and other supporting technologies that you will use throughout the book.

In these first four chapters, you will build a sample Java web application to use with numerous code examples throughout the book. The Java web application is built using a service-oriented architecture (SOA).

After making a whirlwind tour of Flex (chapter 1), your first step (chapter 2) is to create a Java application that will expose web services you will later use to connect to them from a Flex client.

In chapter 3, you will create a Flex application and words like AppFuse, Flex Mojos, and FNA enter your vocabulary.

The focus of chapter 4 is to continue building the client-side application and you will begin building a Flex client that will connect to the Java web services that you will choose to expose.

1 Some Flex with your Java?

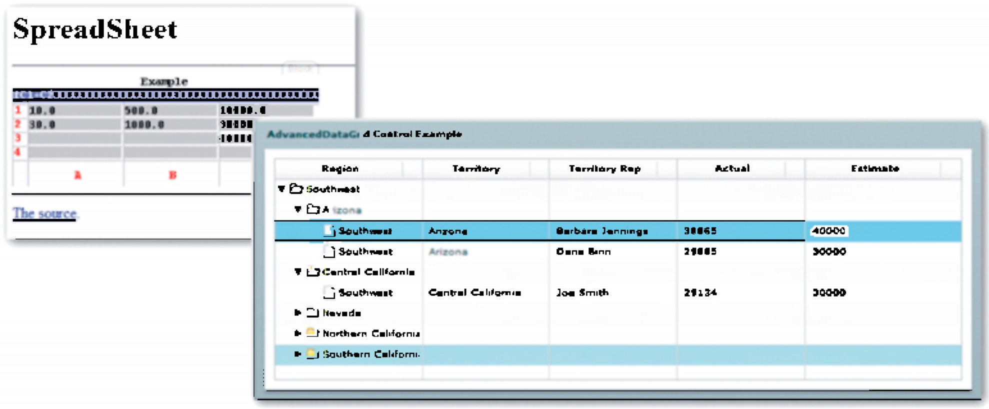
This chapter covers

- A brief history of Java and Flex
- A whirlwind tour of Flex
- MXML and ActionScript

In 1995, Sun introduced the first Java platform and gave birth to the applet which allowed Java applications to run inside the browser with rich functionality and all the benefits of the Java framework, including connecting to the server side. The applet became hugely popular for a couple of years before its popularity waned mainly because of problems surrounding the browser plugin.

Macromedia embraced the idea of having a dedicated runtime environment for the browser, like the Java applet, and in 1997 released the Flash Player. Adobe has since taken over the rights to the Macromedia suite of products and helped to evolve what is now the Flex framework and development API.

Building features in an applet from scratch or even with other rich implementations can be expensive compared to the simplicity of using the Flex framework. Figure 1.1 displays a simple Java applet data grid next to a Flex data grid. The Flex data grid right out of the box not only looks better than the applet, it's


SpreadSheet

Example

	A	B	
1	10.0	500.0	10000.0
2	30.0	1000.0	[illegible]
3			[illegible]
4			

The source

AdvancedDataGrid Control Example

Region	Territory	Territory Rep	Actual	Estimate
▼ Southwest				
▼ Arizona				
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
▼ Central California				
Southwest	Central California	Joe Smith	29134	30000
▶ Nevada				
▶ Northern California				
▶ Southern California				

Figure 1.1 Comparing a Java applet DataGrid (left) to a Flex Advanced DataGrid

much more functional with much less code overhead. The Flex `DataGrid` and `AdvancedDataGrid` components provide built-in support for tasks such as sorting, dragging columns, row highlighting, data nesting, and styling.

The Flash runtime provides lightweight graphics and animation capabilities in manageable file sizes, making the player hugely successful across OSs and browser platforms. The Flash runtime allows for rich applications to have true stateful experiences and a high level of security.

NOTE A stateful experience with Flex means that the client (Flex) will manage or remember everything it needs to without having to: submit to the server side, update and manage a session or request through HTTP, and refresh the client side with updated data after a submit with data from the session or request.

In general, Java developers have successfully leveraged the principles of object-oriented programming (OOP) to build extremely stable, testable, and extensible applications. Flex has become a rich internet application (RIA) solution for Java developers because it not only bridges the gap between a solid server side and a great UI, it is also built on top of OOP principles such as encapsulation, inheritance, and polymorphism.

These advantages benefit other technologies besides pure Java, as you'll see in chapter 11, when we demonstrate Flex integration with Grails, one of the hottest web development platforms. We'll build a simple contact management system and learn how to get rolling with Groovy and Grails development. Integrating Flex with Grails is in many respects easier than integrating Flex with Java.

Flex development is now bolstered by many of the benefits of Java-like frameworks for performing unit testing, functional testing, and continuous integration. The combination of Java and the Flex Software Development Kit (SDK) allows developers new to the business to start building applications immediately.

A thriving Flex open source community can offer Java developers GUI components as simple and as complex as required. Most of these custom components extend stock Flex objects found in the Flex SDK. Adobe made Flex 3 open source, and it now has numerous community resources. The Flex SDK with Adobe's built-in charting components is still commercial.

We have chosen Flex 4 with Java because of the duality of a rich and stateful client in conjunction with a powerful server side. Also, Java is broadly used in the mainstream and is the existing server-side platform for many Flex migration projects. Although there are alternative ways for doing RIA development, Flex will most likely prove to be the superior RIA framework because of the simplicity and testability it provides to developers. We're now ready to discuss some of Flex framework features.

1.1 A whirlwind tour of Flex

It's time to take a peek at the components we'll use throughout this book. We won't go into too much detail about the components as that is beyond the scope of this book. Instead, we'll focus on the usage of components and framework in real-world development.

1.1.1 MXML and ActionScript

At the heart of every Flex application you'll find a combination of MXML files (XML files with the .mxml extension) and ActionScript classes. These two components are the basic building blocks of the Flex framework. The Flex compiler takes these files and creates a small web format (SWF) file, which is executed in the Flash Player.

MXML

MXML is an XML-based markup language similar to HTML/XHTML. The MXML syntax, used to declaratively define your application, has numerous tags for common UI objects, such as text input fields, radio buttons, and drop-down lists. It also has many UI components and layout components that are common in rich client development, such as menu bars, tabbed panels, data grids, and navigational trees. In addition, it's possible to build custom components that extend existing ones or produce something completely different like the flow visualization chart. Figure 1.2, which was made with Degrafa, shows this function.

In chapter 8 we'll be covering the Degrafa drawing API for Flex to create a pie chart for a sample application.

ActionScript

ActionScript, and more specifically ActionScript 3.0, is a dynamic scripting language based on the ECMAScript Language Specification, Third Edition. It is composed of the language specification and the Flash Player API. It is similar to JavaScript in syntax, so it should look familiar to any experienced web developer. Unlike JavaScript, ActionScript is compiled into byte code before being executed, instead of being parsed and interpreted at runtime.

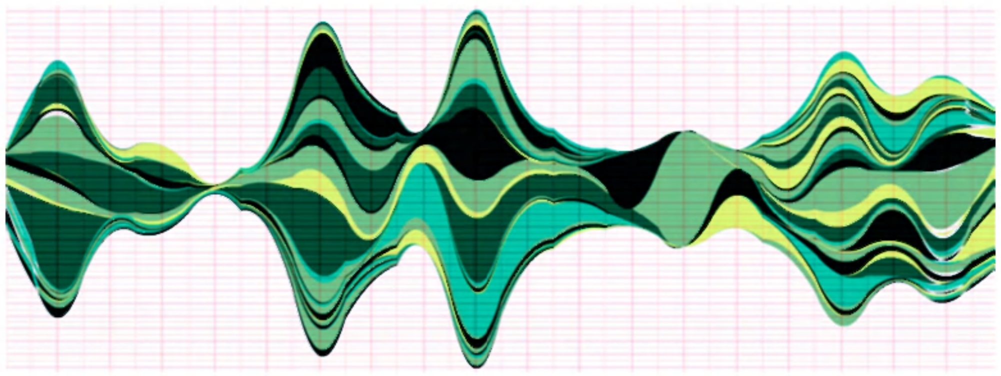


Figure 1.2 Flow visualization chart

> **NOTE** Dozens of user controls, powered by ActlonScript, are available with Flex out of the box. As we demonstrate later in this book, existing components can be extended to create your own custom components. Because your application will always run inside the Flash Player, you don't have to worry about cross-browser compatibility issues either. In chapter 8 we'll go over how to utilize custom components in your Flex applications using ActionScript for the purpose of reuse.

ActionScript is a dynamically typed language similar to Python or Groovy and does its type checking at runtime instead of at compile time. You have the option of directing the compiler to perform type checking at compile time by enabling strict mode on the compiler, but this is not a good substitute for a comprehensive set of unit tests.

The Flex SDK and Flash Player are the two key elements in making a Flex application come to life.

1.1.2 *The Flex SDK*

The Flex 4 SDK comes in two flavors: the Free Adobe Flex SDK and the Open Source Flex SDK. Both contain everything you need for developing, optimizing, and debugging Flex applications. The SDKs include the ActionScript and MXML compilers, tools for creating JavaDoc-like documentation, and the Flex Framework. The only difference between the two is that the Free Adobe Flex SDK contains additional components that enhance the Flex application, such as tools for advanced font encoding, tools for packaging Adobe Integrated Runtime (AIR) applications, and the Flash Player. These extra components are not open source but have been made available by Adobe. To learn more about the Flex SDK downloads visit http://opensource.adobe.com/wiki/display/flexsdk/downloads.

1.1.3 *Flash Player 10*

Of course none of this could be possible without the Flash Player 10 runtime. It is the heart and soul of every Flex applicatlon. Although Flash Player itself is not open source, it has been free since its inception and can be found on nearly every computer in the world. Flash Player gives your Flex applications the ability to execute in the same manner and look the same no matter what browser your application runs in. Because your Flex application runs inside Flash Player, you do not have to be concerned with cross browser issues.

Adobe contributes ActionScript engine source to Mozilla

In November 2006, Adobe contributed the source code for its ActionScript virtual machine to the Mozilla foundation, spawning the Tamarin project. Tamarin will support ECMAScript Edition 3 and be integrated into the SpiderMonkey project, Mozilla's next generation JavaScripting engine to be included with future versions of Mozilla (https://developer.mozilla.org/en/Tamarin).

Because of the widely popular Flash Player and a powerful open source SDK, Flex is a great fit for Java developers building rich clients.

Now comes the part we've all been waiting patiently for—we're going to create a "Hello World!" styled application in Flex.

1.2 Creating an application in Flex

Let's start by modeling our directory structure, shown in figure 1.3, after the Maven default project structure. This will prove useful because we're using Maven to build the FlexBugs sample application in chapter 2.

The sample application can be placed in a project directory such as C:\dev\projects for Windows, or /home/<YOUR_USERNAME>/development for Linux.

The main source code location for our "Hello World!" sample application will be contained in the src/main/flex folder. Because ActionScript follows a pattern similar to Java for packages, if your ActionScript class belongs to the `com.example` package, the source for this class will be contained in the src/main/flex/com/example folder. The src/main/resources folder should contain any resources that belong to the application but are not compiled with the sources. For example, any configuration files or message bundles belong in the resources folder. The src/test/flex and src/test/resources folders are identical to the src/main/flex and src/main/resources folders respectively, except these folders are for the test code of the application.

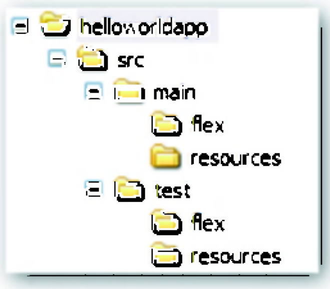
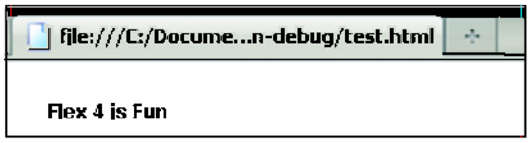


Figure 1.3 The folder structure for our "Hello World!" application is formatted for the Maven build convention.

For the purpose of introducing Flex code, listing 1.1 demonstrates a trivial example of a simple Flex application. We're going to create a single .mxml file that will print the words "Flex 4 is Fun" as seen in figure 1.4.



Flex 4 is Fun

Figure 1.4
A simple Flex application

As you can see there's nothing fancy happening here and the code presented in listing 1.1 is also simple.

Listing 1.1 Main.mxml

```
<?xml version="1.0" encoding="utf-8"?>                     ❶ XML document declaration
<s:Application                                              
    xmlns:fx="http://ns.adobe.com/mxml/2009"                ❷ MXML application root
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

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

    <fx:Style>
        .helloText {                                  ❸ Flex css style support
            padding-top:25px;
            padding-left:25px;
            font-weight: bold;
            color: haloBlue;
        }
    </fx:Style>

    <s:RichText styleName="helloText">
        <s:text>                                          ❹ RichText element
            Flex 4 is Fun
        </s:text>
    </s:RichText>
</s:Application>
```

If you've ever done any web development, this should look familiar. MXML, like XHTML, is nothing more than XML. At the beginning of the file you'll see the standard XML declaration ❶. Next you'll see the root node with the defined xml namespaces ❷, which in most Flex applications will be within the `<mx:Application>` element. In chapter 3, we'll evolve this example by creating a nice Flex application. In chapter 9, where we talk more about Adobe AIR, we'll explain that the root node for an AIR application is typically `<mx:WindowedApplication>`. Inside the `<mx:Application>` element is a style element ❸ and a single `RichText` element ❹ with its text attribute set to "Flex 4 is Fun".

1.3 Finding the right tools and patterns

As this book's title suggests, *Flex on Java* will focus on Flex integration with Java. With that in mind, we diligently searched for the perfect ingredients to equip you with the right tools for real-world scenarios without overcomplicating things.

We're not focusing on teaching Java, but we build a simple Java web application in chapter 2. We chose frameworks that should ease the Java learning curve necessary to get a sample application up and running fast. We won't dive deep into every part of Flex development. Instead, *Flex on Java* will provide you with simple yet powerful examples of integrating the two technologies.

Therefore, we'll be tackling in detail the topics outlined in the sections that follow.

1.3.1 Building a Flex interface

Often developers begin building naked UI components without being connected to a backend server and database. A disconnected Flex client can take advantage of mock data and allow developers to easily prototype the UI without the complexity of external dependencies.

This approach is demonstrated in chapter 3 where we create the beginnings of a rich UI for the FlexBugs sample application shown in figure 1.5.

1.3.2 Integrating with web services

What may seem unfamiliar to you in this chapter is that when we connect to the server side we'll be modeling our application using the Model-View-Presenter (MVP) design pattern instead of employing the typical approach of using the `mxml` tag element that doesn't scale well for most applications. Using the MVP approach, all web services calls

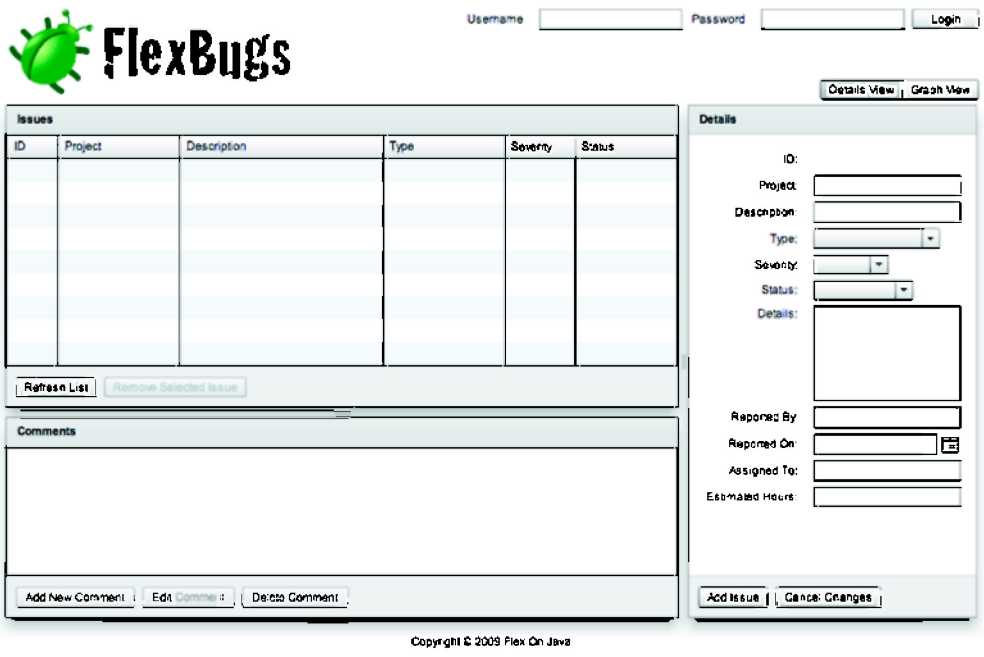


Figure 1.5 The FlexBugs sample Flex application

will be wrapped in a Model object written in ActionScript. This makes changing the implementation less painful.

Connecting to server-side services, event dispatching, and event handling are all things that are critical to any Flex business application and are built into the core of the framework. That's why in chapter 4 we'll demonstrate connecting to the Java server side and how to leverage the powerful Flex API for connecting to web services. Plenty of literature, including the Flex online documentation, covers the typical approach of connecting to services. In this book, we'll create clean interfaces and views as we build a well-designed Flex client that will scale on demand.

1.3.3 *Integration with BlazeDS, logging, and messaging*

Until fairly recently if you wanted to connect your data-driven application to a Java-based backend, your choices were fairly limited. You could expose your Java services as XML web services, either as SOAP-based or RESTful and connect to them this way, write your own custom marshaler/unmarshaler based on the Adobe AMF protocol, or pony up big bucks for a license for Adobe's LiveCycle Data Services. With the release of Flex 3, Adobe decided to spawn BlazeDS, an open source project that contains much of the functionality specific to connecting Flex to a Java-based service.

Integrating Flex with Java is what this book is all about. That's why chapter 5 will demonstrate further how to connect a Flex client more directly to the server side using the open source BlazeDS framework. BlazeDS provides a mechanism for allowing Flex to call methods on Java objects through binary serialization with the Action Message Format or AMF. This is much faster than what's possible with web services or XML/HTTP(s) because it uses real objects and doesn't have to marshal XML.

Because logging is a critical component of any application and development environment, chapter 5 also covers BlazeDS logging in detail.

Real-time messaging is an important feature in most enterprise applications. Chapter 6 will demonstrate how to develop Flex applications that take advantage of simple polling; in chapter 11 we'll discuss how to connect using Java Message Service (JMS). The Flex framework provides an API that enables distributed communication with the server side or communication between clients that is loosely coupled and asynchronous. Flex has both a powerful and simple API for handling messaging.

1.3.4 *Securing Flex applications*

Many technical books skip security, but it's always the elephant in the living room when gathering requirements for an application. It's important to understand the security issues of Flex both to decrease risk and to minimize the cost of what can be one of the most expensive features in any business application. Chapter 7 will demonstrate building authentication, authorization, and personalization with Spring Security (Acegi) integration.

1.3.5 Creating custom Flex controls with Degrafa

Custom components are first class citizens inside the Flex framework, and many ActionScript frameworks take advantage of this. One that really stands out is Degrafa, which is a declarative graphics framework that provides a suite of graphics classes. Degrafa is open source and can be used to build robust charts, tools, and other graphical elements with less effort and complexity than other frameworks.

We'll create a custom pie chart component that will be appropriately tied into our sample application in chapter 8. While we're at it we'll also demonstrate creating a DataGrid `ItemRenderer` and perform dynamic object creation.

1.3.6 Desktop 2.0 with Adobe AIR

A business sometimes can't live with the web alone, and when a desktop client is the best way to go, Flex goes beyond the web with Adobe AIR. A Flex application can easily be ported to a desktop environment with AIR. This is especially easy for well-designed applications that allow for optimal reuse of code. Chapter 9 will demonstrate how to allow a Flex application to live in two worlds—by demonstrating how to package and distribute an AIR application.

1.3.7 Flex and Grails

The Groovy programming language is the first dynamic scripting language to be adopted as a standard through the Java Community Process (JCP). The specification for Groovy can be found under Java Specification Request (JSR) 241 at http://jcp.org/en/jsr/detail?id=241. Groovy supports powerful features that can be found in other languages like Python, Ruby, and Smalltalk; Grails is a full development stack for Groovy that provides a rapid development environment for both server side and the web that resembles and rivals the Ruby on Rails (RoR) framework. With Flex and Grails you can quickly create a dynamic application that runs on the JVM with much less effort. Flex integration with Groovy and Grails is covered in chapter 11.

1.4 Summary

Whether you're experienced or inexperienced in building web or desktop applications with Flex on Java, this book will teach you how to integrate Flex on Java quickly and effectively. Combining Flex with Java allows developers to provide rich UIs with robust server-side technologies and does so with minimal effort and cost. Flex and Java are proven technologies and have continued to be improved over time and used by many companies around the world.

In chapter 2, we lay a foundation by building a sample Java web application for use throughout the majority of the examples in the rest of this book. This should be useful if you don't already have an application or need assistance in getting started with setting up a Flex on Java development environment. Setup-type chapters can feel slightly mechanical because of the downloading and installation they cover. We've tried to keep it interesting and painless while using popular development environment frameworks that the majority of you will be pleased to see utilized.

2 Beginning with Java

This chapter covers

- Generating the application structure with Maven
- Building Java server-side domain objects and services
- Building a simple JSP UI

We'll begin by creating a Java application that will expose web services so we can later connect to them from a Flex client. We have attempted to avoid tying the book to a specific sample application by focusing more on the concepts and techniques of using various frameworks and tools. This should allow you to pick a topic in the book that interests you and get rolling on it. We'll demonstrate many topics by using an application built in this chapter called FlexBugs. If you want to follow the samples in the book you can download the full code listings on the book's website at http://manning.com/allmon. You could also replace the application contents with something that's more meaningful to you by changing the domain objects to manage whatever you want, like contacts or movie favorites.

Throughout the book, especially in this chapter, we leverage a few Java frameworks that help to lighten the amount of work required to build a fully functional web application. This chapter is a bit mechanical because we need to set up a development environment. A few downloads and installs must take place if you choose to use our samples. Feel free to browse through this chapter and skip what you already know.

The Java frameworks used will help keep development to a minimum while creating a sample application to work with for integration purposes. This will allow us to focus on teaching and demonstrating how to build synergy between Flex and Java.

We're building a Java application first as a basis for work in chapter 3, but you can start with Flex in chapter 3 if you'd like or move around the book as convenient. The Java application comes first because we expect most readers to be refactoring existing applications to include a Flex client and this will give you something to play with quickly.

We'll start by generating the project structure with Apache Maven, a convention-over-configuration project management framework. Maven will build the application for us and speed up the development process. After we have a project structure generated, we'll start building the server-side components while leveraging MySQL for the database.

For the Java server-side pieces, we'll start with creating plain old Java objects (POJOs), Data Access Objects (DAOs), and service objects that will be exposed to a web tier.

Let's write a simple Java server-side application using the AppFuse framework. AppFuse was created by Matt Raible of Raible Designs to simplify the construction of Java web applications through convention. Using AppFuse on the server side will allow us to focus on the integration of Flex with Java creating simple domain and service Java objects.

2.1 Working with AppFuse

Because the layers of architecture and complexity can make approaching the building of a Java web application a bit daunting, AppFuse is a great technology choice because it simplifies dealing with the layers and delivering value faster.

AppFuse allows a Java developer to quickly start focusing on business domain concerns. A typical Java application will be POJO-driven and wired together through Spring, the open source dependency injection (DI) framework. The DI design pattern helps to build applications with loosely coupled components making your application more flexible and testable. In addition, AppFuse comes stocked with Maven integration to make things even easier. Let's get things rolling by installing Maven.

2.2 Generating the application structure with Maven

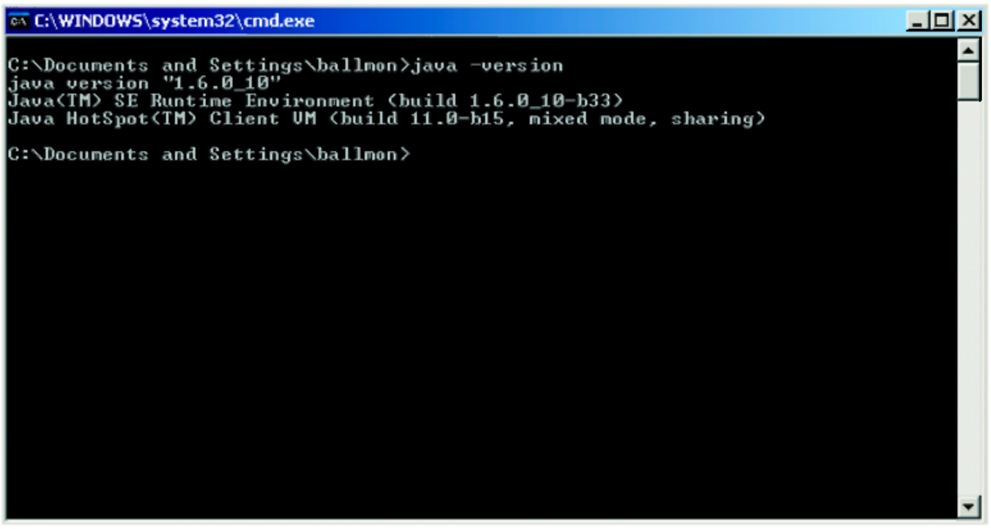
To pigeonhole Maven by calling it a build system doesn't do it justice. Apache Maven is a software project management and comprehension tool. What exactly does that mean? At the core of every Maven project is a project object model, more affectionately known as the POM, and from this POM Maven can build our application, generate reports, generate documentation, and more, all from a single description of the project. To learn more about Maven check out the Apache Maven project site at http://maven.apache.org or download the free ebook from Sonatype at http://www.sonatype.com/book.

Before moving ahead with Maven, be sure you have the Java Development Kit (JDK) version 1.5 or greater properly installed. You can follow the next section for that or skip it if you're ready to go. After you install the JDK, be sure to install the MySQL database as well. You'll need MySQL installed before generating the project with the AppFuse Maven archetype.

2.2.1 Download and install the JDK

To run any Java server-side environment, you must install and configure the JDK. Download and install JDK 1.5+ from the Sun website at http://java.sun.com/javase/downloads/index.jsp. Refer to the Java documentation for instructions on how to install Java on your specific platform. Set up an environment variable for JAVA_HOME that points to the JDK directory. It's also helpful to add the JDK's bin directory to the path. Open a command prompt and type in the Java version to verify that Java is installed correctly. The version information of the configured JDK should be presented as shown in figure 2.1.

After Java is configured you can move on to setting up the open source MySQL database.



```
C:\Documents and Settings\ballmon>java -version
java version "1.6.0_10"
Java(TM) SE Runtime Environment (build 1.6.0_10-b33)
Java HotSpot(TM) Client VM (build 11.0-b15, mixed mode, sharing)

C:\Documents and Settings\ballmon>
```

Figure 2.1 Verify that Java is set up correctly by checking the version

2.2.2 *Download and install MySQL*

To demonstrate database integration and persistence you'll use MySQL, which is an open source database that is extremely lightweight. Download and install MySQL 5.x or higher from the MySQL website at http://dev.mysql.com/downloads/mysql.

Here you'll set up a database for the FlexBugs sample application. After you have MySQL installed pull up the command prompt and log in to MySQL using the root account, then create the flexbugs database as shown in figure 2.2. Using the command `mysql -u root -p` will instruct MySQL to log in to the local host instance of MySQL using the root account. It will ask for the password. Please record the admin account's user and password for later reference. Creating the database is as simple as executing the command `create database flexbugs`.

Let's move on to installing Maven to create the project structure, manage the dependencies, and build the application.

2.2.3 *Download and install Maven*

Maven can be downloaded at http://maven.apache.org/download.html. Be sure to download version 2.0.9 or above. After Maven is downloaded you should set up an M2_HOME environment variable that points to the directory where Maven was installed. The M2_HOME/bin directory will need to be set onto the path as well or exported for any UNIX platform. For more assistance on installing or configuring Maven refer to the Maven documentation at http://www.sonatype.com/books/mvnex-books/reference/installation-sect-maven-install.html.

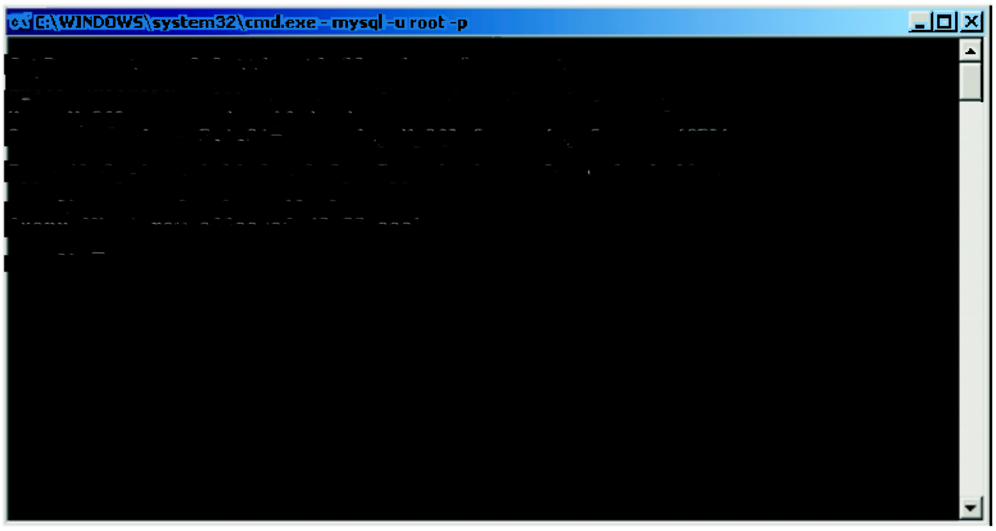


Figure 2.2 Using the MySQL commands to log into the database instance and create the flexbugs database

2.2.4 *Create a Maven multimodule project*

We're going to create a Maven multimodule project called FlexBugs. A multimodule project could be configured manually by creating a top-level *super POM*, adding projects under the super POM directory, and editing the super POM to include the modules with the modules element. We're going to use a technique that exploits a little known feature of the archetype:create plugin, and the Maven site archetype to kick-start the project.

Creating a multimodule project has many benefits, the two most important being (1) the ability to build every artifact in a project with a simple mvn compile command and (2) if you are using either the Maven eclipse:eclipse plugin or the idea:idea plugin, you can enter this command at the root of the project, and it will generate all the project files for all of the contained modules.

First you'll generate the top-level project using the `maven-archetype-site-simple` archetype:

```
mvn archetype:create
 -DgroupId=org.foj
 -DartifactId=flex-bugs
 -DarchetypeArtifactId=maven-archetype-site-simple
```

This generates a Maven project with the directory structure as shown in figure 2.3.

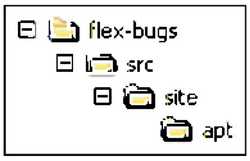


Figure 2.3 The generated top-level Maven project

The project generated is the minimum project setup necessary to generate site documentation. The index.apt file is the main index page for the site, and is written in the Almost Plain Text (APT) format, which is a wiki-like format. You can also generate a more complete site project using the `maven-archetype-site` archetype like this:

```
mvn archetype:create
 -DgroupId=[Java:the project's group id]
 -DartifactId=[Java:the project's artifact id]
 -DarchetypeArtifactId=maven-archetype-site
```

This will generate a project structure similar to figure 2.4.

After you have generated the site project, edit the pom.xml created from the site archetype plugin. Make sure that the packaging type is set to `pom`. We've left sections out (denoted by ...) to be brief.

Listing 2.1 Packaging of type `pom` indicates a multimodule project

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.foj</groupId>
    <artifactId>flex-bugs</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>                    ❶ Artifact type (jar, war, ear)
     ...
</project>
```

Because you set the packaging type to `pom` ❶, any projects you generate from the root of the project directory will insert themselves into the project by creating an entry into the modules section of the pom.xml for the site.

AppFuse comes stocked with custom Maven archetypes, which allow AppFuse to create different flavors of Java web applications with varying technology stacks. You'll use the Struts 2 Basic archetype for the FlexBugs sample application.

In the root directory of your project that you created previously, type the command in listing 2.2.

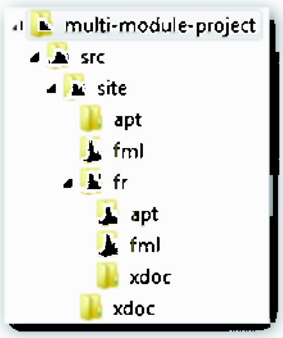


Figure 2.4 A fully dressed up Maven site project

Listing 2.2 Create the `flex-bugs-web` module for the Java server side

```
mvn archetype:create
-DarchetypeGroupId=org.appfuse.archetypes                ❺
-DarchetypeArtifactId=appfuse-basic-struts                   ❷
-DremoteRepositories=http://static.appfuse.org/releases          ❸
-DarchetypeVersion=2.0.2                                     ❶
-DgroupId=org.foj.flex-bugs                            ❺
-DartifactId=flex-bugs-web                          ❷
```

The `appfuse-basic-struts` ❺ archetype isn't a built-in Maven resource. Instead, it's provided through a remote repository ❺. You provide Maven with coordinates to the archetype by also providing the `archetypeGroupId` ❶ and `archetypeVersion` ❶ along with the rest of the required details. The `groupId` ❺ points to the top-level project and the `artifactId` ❺ is the name of the module you are about to create.

After you've executed the command, look inside the top-level pom.xml from the main project. There should now be an entry toward the bottom of the file like the following.

```
...
<modules>
    <module>flex-bugs-web</module>
</modules>
...
```

Executing the command in listing 2.2 should generate the project structure shown in figure 2.5. Don't be concerned with the warnings while creating your project; they are expected. As long as you see BUILD SUCCESSFUL at the end, your project was created successfully.

As you can see from figure 2.5 Maven generated the project structure and added a couple of files for testing.

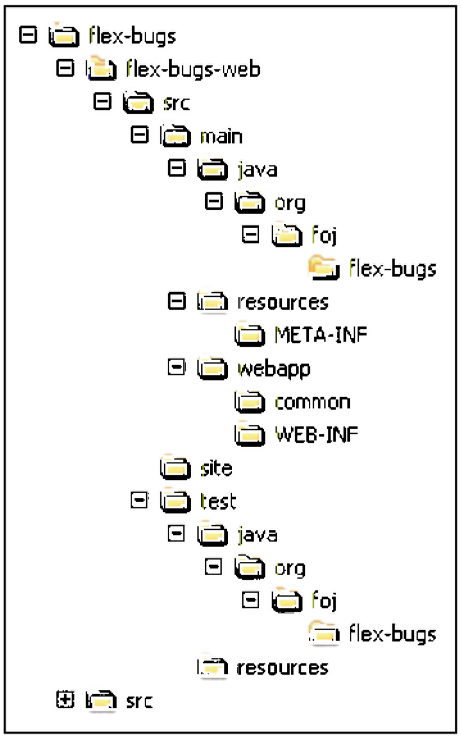


Figure 2.5
Generated module structure using the `appfuse-basic-struts` archetype

2.2.5 *Maven provides a buildable project*

If you look in the `src/main/java/org/foj` package you'll find a source file called App.java, and in the `src/test/java/org/foj` package you'll find a unit test called AppTest.java. Remove both files as you will not need them.

Notice that Maven appears to be building something. In fact, the `flex-bugs-web` POM tries to build a deployable Java Web Archive or *WAR* but will first choke on a configuration issue. If running the `mvn jetty:run-war` command without changing the configuration you'll most likely get this error.

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Error executing database operation: CLEAN_INSERT

Embedded error: Access denied for user 'root'@'localhost' (using password: NO)
[INFO] ------------------------------------------------------------------------
[INFO] For more information, run Maven with the -e switch
```

Let's first edit the POM for the `flex-bugs-web` module. This POM will be located at the root of that module. There's a good deal going but we're going to focus on the piece we need to change. At the bottom you need to specify your MySQL user and password with the values we specified when you set up MySQL earlier. Here's an example:

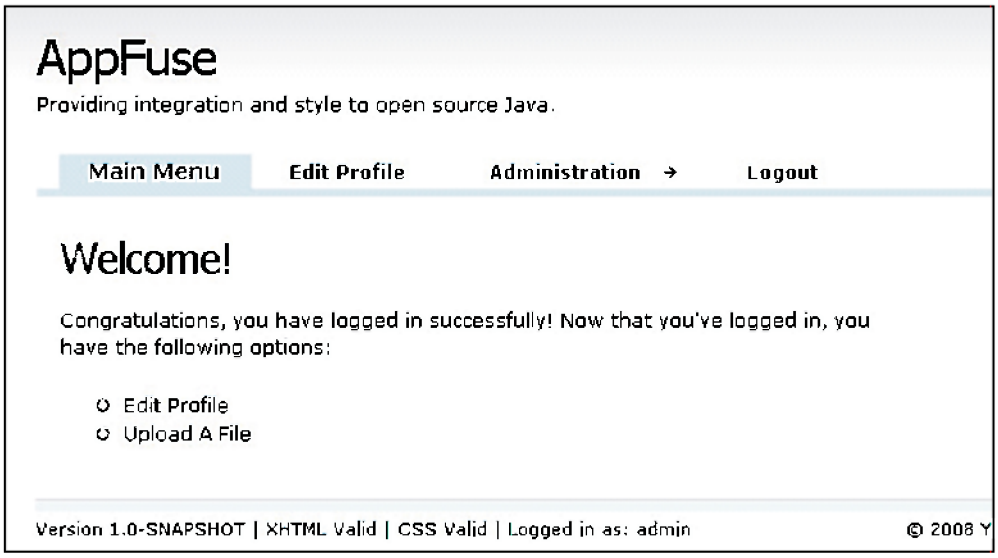
```
...
<jdbc.url><![CDATA[jdbc:mysql://localhost/
➥ flex_bugs_web?createDatabaseIfNotExist=true&amp;useUnicode=true&amp;
➥ characterEncoding=utf-8]]></jdbc.url>
<jdbc.username>root</jdbc.username>
<jdbc.password>java4ever</jdbc.password>
...
```

The Maven archetype we used, brought to us by AppFuse, made it extremely easy to get to this point—far easier than starting from scratch.

2.2.6 Running the FlexBugs web application

Maven equips a developer with the ability to use the application immediately without manually deploying it anywhere. Executing the Maven `jetty:run-war` goal from the `flex-bugs-web` module will gather all the resources, compile all the code and tests, execute the unit tests, generate test reports, build a deployable WAR file, and launch the WAR file in an embedded instance of the popular and lightweight Jetty servlet container. Using the `appfuse-basic-struts` archetype will also generate the default database for us and add configuration files to allow developers to quickly begin developing features.

After you've run the `jetty:run-war` command, you can go to http://localhost:8080/flex-bugs-1.0-SNAPSHOT and log in from there. By default, you can log in to the application using admin for both the username and password. After logging in, you are redirected to the administration panel as seen in figure 2.6. From there you can do basic things like editlng your user profile and managing users.



Welcome!

Congratulations, you have logged in successfully! Now that you've logged in, you have the following options:

- Edit Profile
- Upload A File

Version 1.0-SNAPSHOT | XHTML Valid | CSS Valid | Logged in as: admin © 2008 Y

Figure 2.6 AppFuse default application

The application shows nothing glamorous at this point although everything you see and can do has required a minor setup effort. AppFuse does much under the covers for us from a framework and technology perspective. It's possible that getting a project together with help from Maven saved us a week or more of typical Java development tlme.

Before we start development of the FlexBugs sample applicatlon download the source code at https://flexonjava.googlecode.com/svn/flex-bugs/trunk.

2.3 Build the model objects

A model object is a POJO that is persistable and mapped to the database. In our example we're using AppFuse with the Spring framework and Hibernate to manage performing database operations for objects that are mapped to a database.

Let's start with `Issue.java` as seen in listlng 2.3. For the FlexBugs applicatlon you need something to store issues and comments. An *issue* describes something that needs fixing to meet a requirement. This could be a bug, a new feature, a refactor, or an optimization. A single issue can have many comments so a relatlonship is built between the issue and comment objects.

Listing 2.3 The `Issue` model object

```
package org.foj.model;                                  <-❶ Model Java package

import org.apache.commons.lang.builder.EqualsBuilder;   <-❷ Import declarations
...
                                                        ❸ Java persistence framework
@Entity
public class Issue extends BaseObject implements Serializable {

  private Long id;                              ❺ Class instance variables
  private String project;                       ❹ Issue extends AppFuse BaseObject
  private String description;
  private String type;
  private String severity;
  private String status;
  private String details;
  private String reportedBy;
  private Date reportedOn;
  private String assignedTo;
  private Double estimatedHours;               ❻ Declares database pk relationship

  @Id                                           ❼ Indicates how to generate Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public Long getId() {                         ❽ "getter" method returns Id
    return id;
  }

  public void setId(Long id) {                  ❾ "setter" method sets Id
    this.id = id;
  }

    ...
  @Override                                                        <-┐
  public int hashCode() {                                           ❿ hashCode
    return new HashCodeBuilder(11, 37).append(id).toHashCode();
  }

  @Override                                                    <-┐
  public boolean equals(Object o) {                             ⓫ equals
    if (null == o) return false;
    if (!(o instanceof Issue)) return false;
    if (this == o) return true;

    Issue input = (Issue) o;
    return new EqualsBuilder()
        .append(this.getId(), input.getId())
        .isEquals();

  }
                                                               ⓬ toString provides
  @Override                                                    <-┘ object info
  public String toString() {
    return new ToStringBuilder(this, ToStringStyle.MULTI_LINE_STYLE)
        .append(id)
        .append(project)
        .append(description)
        .toString();
  }
}
```

You'll be storing the model objects in the `org.foj.model` Java package ❸ and will use the AppFuse framework in conjunction with the Spring Framework and Hibernate to simplify our application development. Spring provides DI and more while Hibernate is a database persistence framework that enables object relational mapping framework ❸. The `Id` ❻ and `GeneratedValue` ❼ annotation help to facilitate the persistence by designating a field as a database primary key.

The `Issue` object is a subclass of the AppFuse `BaseObject` ❸ and contains the instance variables ❸ you need to describe an issue. All of the instance variables or *fields* have the getters ❸ and setters ❸ required by the JavaBean specification.

NOTE Extending `BaseObject` requires us to override the `toString` ⓬, `equals` ⓫, and `hashCode` ❿ methods because they're defined as abstract in the `BaseObject` class. To implement these methods we're leveraging the Apache Commons Builder package ❸ for creating the elements for these methods. Whenever you're implementing the `Serializable` interface, it's a good idea to also implement the `equals` and `hashCode` methods and provide a `serialVersionUID` member.

Next you'll create a model object for a comment. The `Comment` will be another persistable object. There can be many comments to a single issue. For the remainder of the code snippets in this chapter we'll use "..." for trivial things like imports and getters and setters of similar objects.

Listing 2.4 The comment model object

```
...

@Entity
public class Comment extends BaseObject implements Serializable {

  private Long id;
  private Issue issue;
  private String author;
  private Date createdDate;
  private String commentText;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  @ManyToOne(fetch = FetchType.EAGER)
  public Issue getIssue() {
    return issue;
  }

  public void setIssue(Issue issue) {
    this.issue = issue;
  }

...

  @Override
  public int hashCode() {
    return new HashCodeBuilder(11, 37).append(id).toHashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (null == o) return false;
    if (!(o instanceof Issue)) return false;
    if (this == o) return true;

    Issue input = (Issue) o;
    return new EqualsBuilder()
        .append(this.getId(), input.getId())
        .isEquals();

  }

  @Override
  public String toString() {
    return new ToStringBuilder(this, ToStringStyle.MULTI_LINE_STYLE)
        .append(id)
        .toString();
  }

}
```

Comment declaration ❶

❷ **Comment has many-to-one relationship with Issue**

There's not much difference between an Issue and a Comment ❶. The class fields are related to comments and there is a many-to-one relationship ❷ with Issue. We've also told Hibernate that we'd like it to eagerly fetch the Issue when returning the Comment. In typical Java web development you would keep the session open to lazily load the Issue object only when it's referred to at runtime, but because your Flex application runs external to the JVM you cannot take advantage of this luxury.

Now that you have your model objects built you can create a set of DAOs. You'll need a DAO for issue and comment.

2.4 Build the DAOs

AppFuse provides generic implementations for DAOs that you can leverage if your DAOs do nothing more than the basic create, retrieve, update, delete (CRUD) operations. Because your IssueDao will do only basic operations, there is no need to define a concrete IssueDao. You can instead use the GenericHibernateDao which you'll see later when you wire the beans in the application context. The CommentDao needs to implement a couple of operations that go beyond the basic CRUD operations so you'll first create an interface for the CommentDao.

Listing 2.5 The CommentDao.java

```
...
public interface CommentDao extends GenericDao<Comment, Long> {
  List<Comment> getCommentsByIssueId(Long issueId);
  void deleteAllCommentsForIssueId(Long issueId);
}
```

The CommentDao has two simple methods, one that returns a list of Comment objects by passing in the issueId argument, and another to delete all of the Comment objects for an issue. The second method facilitates the deleting of Issue objects. Because Comment has a foreign key relationship with Issue, you cannot delete an Issue if any Comments refer to it.

> **NOTE** We defined the relationship between Comment and Issue by annotating the field with a @OneToOne annotation, and could have also defined the reverse of that relationship in the Issue class by including a Set of Comment objects belonging to an Issue. Because we could not lazy load those objects, it would have forced us to eager load the Comment objects into the Set, which would force those Comment objects to eager load their Issue objects. This forces the Issue objects to eager load their Comments, and so on. This usually results in a stack overflow because you've effectively got a circular reference that causes an infinite loop of eager fetching.

Now let's implement the CommentDaoImpl.

Listing 2.6 The CommentDaoImpl.java

```
...
public class CommentDaoImpl extends GenericDaoHibernate<Comment, Long>
    implements CommentDao {

  public CommentDaoImpl() {
    super(Comment.class);
  }

  @Override
  @SuppressWarnings("unchecked")
  public List<Comment> getCommentsByIssueId(Long issueId) {
    return getHibernateTemplate().find
    ➥ ("from Comment where issue_id = ?", issueId);
  }
}
```

Much like the `IssueDaoImpl`, `CommentDaoImpl` extends `GenericDaoHibernate` but implements `CommentDao`. The only interesting thing happening here is that you have a method that returns a list of `Comment` objects by leveraging the Hibernate template and a query. Spring and Hibernate are a wonderful combination and make for clean and intuitive DAOs.

Now that you've constructed the DAOs you can build services.

2.5 Build the services

Now you need to expose services to the web tier. You'll be able to take advantage of these services for the Flex client you'll be building. Again you'll start with interfaces like `IssueManager` in listing 2.7.

Listing 2.7 The IssueManager.java

```
package org.foj.service;

import org.foj.model.Issue;
import javax.jws.WebService;
import java.util.List;

@WebService
public interface IssueManager {            ❶ IssueManager interface declaration with WebService annotation

  List<Issue> getAll();                    ❷ Return all issues
  Issue get(Long id);                      ❸ Get specific issue by its ID
  Issue save(Issue issue);                 ❹ Save issue
  void remove(Long id);                    ❺ Delete issue

}
```

You define the `IssueManager` as a web service by annotating it using `@WebService` ❶. `IssueManager` contains methods defining your basic CRUD operations for reading ❷ and ❸, creating and updating ❹, and deleting ❺. Now let's take a look at the `CommentManager`.

Listing 2.8 The CommentManager.java

```
package org.foj.service;

import org.foj.model.Comment;
import javax.jws.WebService;
import java.util.List;

@WebService                                          ❶ CommentManager interface declaration with WebService annotation
public interface CommentManager  {

  List<Comment> findCommentsByIssueId(Long issueId);      ❷ Get comments for issue Id
  void deleteAllCommentsForIssueId(Long issueId);         ❸ Delete all comments for issue
  Comment get(Long id);
  Comment save(Comment comment);                          ❹ Save comment
  void remove(Long id);                                   ❺ Remove comment

}
```

CommentManager is also a web service ❶ by virtue of it having the @WebService annotation just as with the IssueManager. It contains a method to return a list of Comment objects by providing an issueId ❷. a method for deleting all comments for an issue id ❸, a method for saving a comment ❸ and a method for deleting a comment ❸. Now let's provide implementation for the services like IssueManagerImpl.

Listing 2.9 The IssueManagerImpl.java

```
package org.foj.service.impl;

import org.AppFuse.dao.GenericDao;
import org.foj.model.Issue;
import org.foj.service.IssueManager;
import org.foj.service.CommentManager;
import java.util.List;
import javax.jws.WebService;

@WebService(serviceName = "IssueService",
    endpointInterface = "org.foj.service.IssueManager")       ❶ IssueManagerImpl declaration with WebService annotation
public class IssueManagerImpl implements IssueManager {

  private GenericDao<Issue, Long> issueDao;
  private CommentManager commentManager;

  public IssueManagerImpl() {                                  ❷ Default no arg constructor
  }

  public IssueManagerImpl(GenericDao<Issue, Long> issueDao,
                          CommentManager commentManager) {     ❸ Constructor
    this.issueDao = issueDao;
    this.commentManager = commentManager;
  }

  public List<Issue> getAll() {                                ❹ Method that returns all Issues
    return issueDao.getAll();
  }
  public Issue get(Long id) {                          ❺ Get specific issue
    return issueDao.get(id);
  }

  public Issue save(Issue issue) {                     ❻ Save issue
    return issueDao.save(issue);
  }

  public void remove(Long id) {                        ❼ Remove an issue
    commentManager.deleteAllCommentsForIssueId(id);
    issueDao.remove(id);
  }

}
```

The `IssueManagerImpl` also uses the `@WebService` annotation just as in the interface, but provides the `serviceName` and `endpointInterface` attributes ❶. You provide a default no args constructor ❷ as well as one that will be used by Spring to inject the `IssueDao` and `CommentManager` ❸. Next implement the methods for returning the list of `Issue` objects ❹, returning a specific `Issue` ❺, and saving an `Issue` ❻ by delegating the calls to those methods to the `IssueDao`. The implementation for removing an issue ❼ first deletes any comments for the issue by calling the `CommentManager`, then removes the issue by calling the remove method on the `IssueDao`. Now let's look at the `CommentManager`.

Listing 2.10 The CommentManagerImpl.java

```
package org.foj.service.impl;

import org.foj.dao.CommentDao;
import org.foj.model.Comment;
import org.foj.service.CommentManager;
import java.util.List;
import javax.jws.WebService;

@WebService(serviceName = "CommentService",
            endpointInterface = "org.foj.service.CommentManager")
public class CommentManagerImpl implements CommentManager {
                                        ❶ CommentManagerImpl declaration with WebService annotation
  private CommentDao commentDao;

  public CommentManagerImpl() {                        ❷ Default no args constructor
  }

  public CommentManagerImpl(CommentDao commentDao) {   ❸ Constructor sets injected CommentDao instance to use
    this.commentDao = commentDao;
  }

  public List<Comment> findCommentsByIssueId(Long issueId) {    ❹ Find all comments for issue
    return commentDao.getCommentsByIssueId(issueId);
  }

  public void deleteAllCommentsForIssueId(Long issueId) {       ❺ Delete all comments for issue
    commentDao.deleteAllCommentsForIssueId(issueId);
  }
  public Comment get(Long id) {                                ⊲┐ Get specific
    return commentDao.get(id);                                  6 comment
  }

  public Comment save(Comment comment) {                       ⊲┐
    return commentDao.save(comment);                            7 Save comment
  }

  public void remove(Long id) {                                ⊲┐
    commentDao.remove(id);                                      8 Delete comment
  }
}
```

Like IssueManagerImpl, CommentManagerImpl declares itself to be a WebService ❶. Next using the @WebService annotation and defines its endpoint interface and service name you create a default no args constructor ❷ as well as one that will be used by Spring to inject your CommentDao ❸. You implement the methods to get the Comment objects for an issue ❹, deleting all the Comment objects for an issue ❷, getting a specific Comment ❻, saving a Comment ❼, and deleting a Comment ❽ by delegating to the CommentDAO.

NOTE AppFuse provides GenericManager implementation base classes just as it does for DAOs, but we chose not to use them here because certain Web-Service consumers such as Flex have difficulty dealing with web services that return objects such as ArrayOfAnyType, which is what AppFuse will return if we leverage the GenericManagers. To work around this issue you'll be defining and implementing your CRUD operations for the web services explicitly.

We're now officially done with the server-side objects and can wire things together with the Spring configuration and work on the web tier components.

2.6 Wiring things together with Spring

Spring enables developers to easily connect objects while keeping application components loosely coupled and testable. Notice how we've wired the model, DAO, and service objects together in the following listing. The applicationContext.xml is located in the src\main\webapp\WEB-INF directory, with other configuration files.

Listing 2.11 The applicationContext.xml

```
...
  <!-- Add new DAOs here -->
  <bean id="issueDao" class=                                  ❶ issueDao
  ➥ "org.AppFuse.dao.hibernate.GenericDaoHibernate">
    <constructor-arg value="org.foj.model.Issue"/>
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <bean id="commentDao" class="org.foj.dao.impl.CommentDaoImpl">     ⊲┐
    <property name="sessionFactory" ref="sessionFactory"/>             │
  </bean>                                                   commentDao ❷
  <!-- Add new Managers here -->
  <bean id="issueManager" class="org.foj.service.impl.IssueManagerImpl">
    <constructor-arg ref="issueDao"/>
  </bean>                                                 issueManager ❸

  <bean id="commentManager" class=                   ❹ commentManager
➥ "org.foj.service.impl.CommentManagerImpl">
    <constructor-arg ref="commentDao"/>
  </bean>

...
```

The first bean you define is `GenericDao` for the `issueDao` ❶. The `commentDao` ❷ is defined with your concrete implementation. Next, you create Spring beans for `issue-Manager` ❸ and `commentManager` ❹. The `constructor-arg` element is used to inject the dependencies into the service class constructor.

Now that we've wired things up with Spring let's construct the web tier starting with Struts 2 framework action classes.

2.7 Constructing the web tier

Struts 2 applications implement the Model-View-Controller (MVC) design pattern, which is not to be confused with the MVP design pattern used to develop the Flex application. The pattern encourages separation between the data model, view elements, and controllers that sit between them. The MVC pattern, widely adopted in the Java community, has made its way into other languages and frameworks, like Flex.

2.7.1 Building Struts 2 action classes

You'll start by building controller or *action* classes first, like `IssueAction`.

Listing 2.12 The IssueAction.java

```
package org.foj.action;

import org.AppFuse.webapp.action.BaseAction;
...                                                  ❶ IssueAction extends
public class IssueAction extends BaseAction {          AppFuse BaseAction

  private IssueManager issueManager;
  private CommentManager commentManager;
  private List<Issue> issues;
  private List<Comment> comments;                        Setters for ❷
  private Issue issue;                              IssueManager and
  private Long id;                                  CommentManager

  public void setIssueManager(IssueManager issueManager) {
    this.issueManager = issueManager;
  }

  public void setCommentManager(CommentManager commentManager) {
    this.commentManager = commentManager;
  }

  ...
  public String list() {                                         ❸ Returns list of
    issues = issueManager.getAll();                                Issue objects
    return SUCCESS;
  }

  public String delete() {                                       ❹ Deletes Issue
    issueManager.remove(issue.getId());
    saveMessage(getText("issue.deleted"));

    return SUCCESS;
  }

  public String edit() {                                         ❺ Edits by IssueId
    if (id != null) {
      issue = issueManager.get(id);
    } else {
      issue = new Issue();
    }

    comments = commentManager.findCommentsByIssueId(issue.getId());

    return SUCCESS;
  }

  public String save() throws Exception {                        ❻ Saves Issue
    if (cancel != null) {
      return CANCEL;
    }

    if (delete != null) {
      return delete();
    }

    boolean isNew = (issue.getId() == null);

    issue = issueManager.save(issue);

    String key = isNew ? "issue.added" : "issue.updated";
    saveMessage(getText(key));

    if (!isNew) {
      return INPUT;
    } else {
      return SUCCESS;
    }

  }
}
```

`IssueAction` extends the AppFuse `BaseAction` ❶ that contains many common methods that actions rely on. `IssueAction` has setters for the service objects ❷. These setters will be called by Spring, and their instances will be injected into the action class during runtime. `IssueAction` facilitates controlling communications to the server side from the web tier. It contains the methods for the view pages to list ❸, delete ❹, edit ❺ or, most importantly, save `Issue` objects ❻.

The `CommentAction` object serves the same purpose for the `Comment` object as the `IssueAction` object does for the `Issue` object. All the methods on `CommentAction` are facilitating CRUD for the `Comment` POJO by calling the `commentManager` service. The `CommentAction` class can be downloaded from the website if needed.

Now that the actions are in place, let's work on JSP files to create a simple UI for managing issues.

2.7.2 *Editing the issue menu item*

First you have to modify the menu.jsp to get to the issues list.

Listing 2.13 The menu.jsp

```
...
    <menu:displayMenu name="MainMenu"/>
    <menu:displayMenu name="UserMenu"/>
    <menu:displayMenu name="IssueMenu"/>      ❶ Adding IssueMenu item
    <menu:displayMenu name="AdminMenu"/>        to the JSP view file
    <menu:displayMenu name="Logout"/>
...
```

The menu JSP file reads in the menu xml data. To add the `Issue` menu item you need only add a single line ❶ to this file that is located in the flex-bugs-web/src/main/webapp/common directory. In the following listing you'll provide the xml data for that menu item.

Listing 2.14 The menu-config.xml

```
...
    <Menu name="IssueMenu" title="menu.issue"       ❶ Add Issue menu item
    ➥description="Issues Menu"                       to menu data xml file
        roles="ROLE_ADMIN,ROLE_USER" page="/issues.html">
      <Item name="ViewIssues" title="menu.viewIssues" page="/issues.html"/>
    </Menu>
...
```

By adding to the existing AppFuse plumbing that creates menu items ❶, you quickly gain access to new features. Let's create the IssueList.jsp that will be displayed when you click the issues menu item.

2.7.3 *Adding JSP resources*

The issueList.jsp will display a list of issues and allow you to add or modify existing issues. The issue and comment JSP files will reside in the ../src/main/webapp/WEB-INF/pages directory.

Listing 2.15 The issueList.jsp

```
<%@ include file="/common/taglibs.jsp" %>      ❶ Essential tag
                                                  libraries bundle
<head>
  <title><fmt:message key="issueList.title"/></title>
  <meta content="<fmt:message key='issueList.heading'/>" name="heading"/>
</head>
```

```
<c:set var="buttons">                                    <-❷ Variable holds button data
  <input type="button" style="margin-right: 5px"
         onclick="location.href='<c:url value="editIssue.html"/>'"
         value="<fmt:message key="button.add"/>"/>
  <input type="button" onclick="location.href=
  ➥'<c:url value="/mainMenu.html"/>'"                          ❸ Prints
         value="<fmt:message key="button.done"/>"/>               button data
</c:set>                                                          for display

<c:out value="${buttons}" escapeXml="false"/>                       ❹ Variable
                                                                       represents
<s:set name="issues" value="issues" scope="request"/>                  issues list

<display:table name="issues" class="table" requestURI="" id="issueList"
    export="false" pagesize="25">
  <display:column property="id" sortable="true" href="editIssue.html"
                paramId="id" paramProperty="id" titleKey="issue.id"/>
  <display:column property="project" sortable="true"
     titleKey="issue.project"/>                             Displays nicely
  <display:column property="description" sortable="false"  formatted table ❺
     titleKey="issue.description"/>

  <display:setProperty name="paging.banner.item_name" value="issue"/>
  <display:setProperty name="paging.banner.items_name" value="issues"/>

</display:table>

<c:out value="${buttons}" escapeXml="false"/>              ❻ JavaScript highlights
                                                              table rows
<script type="text/javascript">
  highlightTableRows("issueList");
</script>
```

To make life easier, you include a JSP that in turn includes a bundle of tag libraries ❶ that are useful for the web application. You have button data that will be stored in a variable ❸ and a Java Standard Tag Library (JSTL) tag ❸ that will print the buttons. You create a variable that will hold a list of issues ❸ from the request scope and an HTML table that is formatted using the included display tag library ❸. A little JavaScript is used to highlight rows of data ❻. Now let's have a look at the issue-Form.jsp.

Listing 2.16 The issueForm.jsp

```
<%@ include file="/common/taglibs.jsp" %>

<head>                                                        "s" Struts 2 ❶
  <title><fmt:message key="issueDetail.title"/></title>       form tag in
  <meta content="<fmt:message key='issueDetail.heading'/>"/>       action
</head>

<s:form id="issueForm" action="saveIssue" method="post" validate="true">
  <s:hidden name="issue.id" value="%{issue.id}"/>

  <s:textfield key="issue.project" required=               ❷ Form text
  ➥"true" cssClass="text medium"/>                             input fields
  <s:textfield key="issue.description" required="true"
  ➥cssClass="text medium"/>
  <s:textfield key="issue.type" required="true" cssClass="text medium"/>
  <s:textfield key="issue.severity" required="true" cssClass="text medium"/>
  <s:textfield key="issue.status" required="true" cssClass="text medium"/>
  <s:textarea key="issue.details" required="true" cssClass="text medium"/>

  <li class="buttonBar bottom">                                ❸ CRUD Button bar
    <s:submit cssClass="button" method="save"
    ➥key="button.save" theme="simple"/>
    <c:if test="${not empty issue.id}">
      <s:submit cssClass="button" method="delete"
      ➥key="button.delete" onclick="return confirmDelete('issue')"
              theme="simple"/>
    </c:if>
    <s:submit cssClass="button" method="cancel"
    ➥key="button.cancel" theme="simple"/>
  </li>

</s:form>

<c:if test="${not empty issue.id}">

  <s:form id="commentsForm" action="editComment"                  Comments
  ➥method="post" validate="true">                                 Struts 2 form

    <s:set name="comments" value="comments" scope="request"/>
    <s:hidden name="issue.id" value="%{issue.id}"/>

    <display:table name="comments" class="table"
    ➥requestURI="" id="commentList" export="false" pagesize="25">
      <display:column property="id" sortable="true" href="editComment.html"
                paramId="id" paramProperty="id" titleKey="comment.id"/>
      <display:column property="author"
      ➥sortable="true" titleKey="comment.author"/>
      <display:column property="commentText"
      ➥sortable="false" titleKey="comment.commentText"/>

      <display:setProperty name="paging.banner.item_name" value="comment"/>
      <display:setProperty name="paging.banner.items_name" value="comments"/>

    </display:table>

    <s:submit cssClass="button" key="button.add" theme="simple"/>

  </s:form>

</c:if>

<script type="text/javascript">
  highlightTableRows("commentList");
</script>                                                      JavaScript
                                                               assigns focus
<script type="text/javascript">
  Form.focusFirstElement($("issueForm"));

</script>
```

Obviously, the issueForm.jsp will allow a user to add or edit an issue. If you peek into the included src/main/webapp/common/taglibs.jsp you'll notice that the Struts 2 tag libraries are included and the letter "s" was used for the tag prefix ❶. The Struts 2 `textfield` elements ❷ map to an `Issue` object. The button bar created will contain

Save, Delete, and Cancel buttons ❸. The Delete button will display only if the issue has an id or already exists. Let's keep moving and build the commentForm.jsp.

Listing 2.17 The commentForm.jsp

```
<%@ include file="/common/taglibs.jsp" %>

<head>
  <title><fmt:message key="commentDetail.title"/></title>
  <meta content="<fmt:message key='commentDetail.heading'/>"/>
</head>

<s:form id="commentForm" action="saveComment" method="post" validate="true">
  <s:hidden name="comment.id" value="%{comment.id}"/>
  <s:hidden name="issue.id" value="%{issue.id}"/>
  <s:textfield key="comment.author" required="true" cssClass="text medium"/>
  <s:textfield key="comment.createdDate" required="false"
  ➥cssClass="text medium"/>
  <s:textarea key="comment.commentText" required="false"
  ➥cssClass="text medium"/>

  <li class="buttonBar bottom">
    <s:submit cssClass="button" method="save"
    ➥key="button.save" theme="simple"/>
    <c:if test="${not empty comment.id}">
      <s:submit cssClass="button" method="delete"
      ➥key="button.delete" onclick="return confirmDelete('comment')"
              theme="simple"/>
    </c:if>
    <s:submit cssClass="button" method="cancel"
    ➥key="button.cancel" theme="simple"/>
  </li>

</s:form>
```

As the name suggests, the commentForm.jsp provides a Struts 2 form for updating new or existing comments. When submitted, the form will call the comment manager's `saveComment` method. Now that you have the JSP files in place we'll need to add those properties so that they have real values.

2.7.4 Adding property resources

For the application's messages to be localized, we've leveraged the Java resource bundle framework. Add the properties shown in the following listing to the ApplicationResources.properties file located in the flex-bugs-web/src/main/resources directory.

Listing 2.18 The ApplicationResources.properties

```
# -- menu/link messages --
menu.issue=Issues
menu.viewIssues=View Issues

# -- issue form --
issue.id=Id
issue.project=Project
issue.description=Description
issue.added=Issue has been added successfully.
issue.updated=Issue has been updated successfully.
issue.deleted=Issue has been deleted successfully.

# -- issue list page --
issueList.title=Issue List
issueList.heading=Issues

# -- issue detail page --
issueDetail.title=Issue Detail
issueDetail.heading=Issue Information

# -- comment form --
comment.id=Id
comment.author=Author
comment.issueId=Issue Id
comment.createdDate=Created Date
comment.commentText=Details
comment.added=Comment has been added successfully.
comment.updated=Comment has been updated successfully.
comment.deleted=Comment has been deleted successfully.

# -- issue list page --
commentList.title=Comment List
commentList.heading=Comments

# -- issue detail page --
commentDetail.title=Comment Detail
commentDetail.heading=Comment Information
```

If more language support is needed, add the same properties with the respective translation to the appropriate properties file in the same directory. Now let's wire up the view components with Struts 2.

2.7.5 *Configuring the struts.xml*

To wire up the JSP view components to the controller objects, you can use the struts.xml located in the src/main/resources directory. This listing demonstrates the wiring you need for the issues management.

Listing 2.19 The struts.xml

```
<package>
    ...
 <!-- Add additional actions here -->
    <action name="issues"
    ➥class="org.foj.action.IssueAction" method="list">      ❶ issues action loads issueList.jsp
      <result>/WEB-INF/pages/issueList.jsp</result>
    </action>

    <action name="editIssue"
    ➥class="org.foj.action.IssueAction" method="edit">      ❷ editIssue loads issueForm.jsp
      <result>/WEB-INF/pages/issueForm.jsp</result>
    <result name="error">/WEB-INF/pages/issueList.jsp</result>
  </action>

  <action name="saveIssue"                                  ❸ saveIssue loads
  ➥class="org.foj.action.IssueAction" method="save">           issueForm
    <result name="input">/WEB-INF/pages/issueForm.jsp</result>
    <result name="cancel" type="redirect-action">issues</result>
    <result name="delete" type="redirect-action">issues</result>
    <result name="success" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
  </action>

  <action name="comments" class="org.foj.action.CommentAction"
  ➥method="list">
    <result>/WEB-INF/pages/commentList.jsp</result>
  </action>

  <action name="editComment" class="org.foj.action.CommentAction"
  ➥method="edit">
    <result>/WEB-INF/pages/commentForm.jsp</result>
    <result name="error">/WEB-INF/pages/commentList.jsp</result>
  </action>

  <action name="saveComment" class="org.foj.action.CommentAction"
  ➥method="save">
    <result name="input">/WEB-INF/pages/commentForm.jsp</result>
    <result name="cancel" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
    <result name="delete" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
    <result name="success" type="redirect-action">
      <param name="actionName">editIssue</param>
      <param name="id">${issue.id}</param>
    </result>
  </action>

</package>
```

Struts 2 makes it simple to wire up the view components quickly and make changes. As you can see, the issues action ❶ will load the `issueList.jsp` whenever the `list()` method is invoked. In the same way, `editIssue` ❷ will load the `issueForm.jsp` when the `edit()` method is called and if that doesn't work, it will go back to the list page. The `saveIssue` action ❸ will persist an issue by taking the input from the `issueForm.jsp`.

The remainder of the `IssueAction` is more of the same but pertains to issue comments.

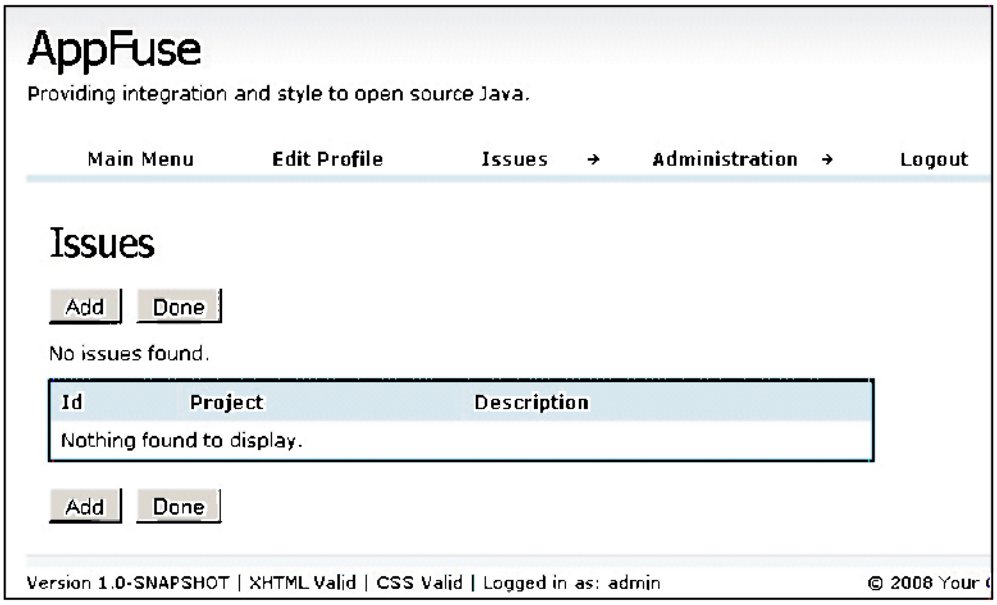
2.7.6 *Configuring Hibernate*

The final step is to configure the POJOs with the Hibernate session factory. That way when the app is loaded into memory, Hibernate recognizes these objects. You do this through the hibernate.cfg.xml located in the src/main/resources directory.

Listing 2.20 The hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
➥Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <mapping class="org.AppFuse.model.User"/>
    <mapping class="org.AppFuse.model.Role"/>
    <mapping class="org.foj.model.Issue"/>        ❶ Adding issue
    <mapping class="org.foj.model.Comment"/>        and comment
  </session-factory>
</hibernate-configuration>
```

In this simple configuration, you create a class mapping ❶ for each of the model objects, `Issue` and `Comment`. Rebuild the application with the `mvn jetty:run-war` command, then refresh your browser. The Issues button should be available as seen in figure 2.7.


AppFuse
Providing integration and style to open source Java.
Main Menu
Edit Profile
Issues →
Administration →
Logout
Issues
Add
Done

No issues found.

Id	Project	Description
Nothing found to display.		

Add
Done
Version 1.0-SNAPSHOT | XHTML Valid | CSS Valid | Logged in as: admin
© 2008 Your

Figure 2.7 The issues list page with the integrated Issues menu button

2.8 Summary

In this chapter we set up a Java web application using the AppFuse framework. AppFuse simplified the plumbing involved in building a typical Java web application by using many popular frameworks, for example Struts 2, Spring, and Hibernate.

In the next chapter we'll start building the rich interface for the sample application in Flex. In the following chapters we'll begin to connect the Flex front end to the Java application using web services and BlazeDS.

3 Getting rich with Flex

This chapter covers

- Creating a Flex project using archetype
- Creating a Flex frontend for the sample application
- Adding a wrapper for an SWF

In chapter 2 we introduced you to AppFuse and created our sample issue tracking application. Now it's time to begin creating the Flex frontend for our sample application. We'll start by incrementally building up the view layer, introducing you to a few of the pertinent concepts of Flex. This chapter is not meant to be a comprehensive guide to the Flex framework by any stretch of the imagination, so you should be able to follow along without much trouble. If you want a more in-depth look at the Flex framework refer to *Flex 4 in Action* by Tariq Ahmed, Dan Orlando, John C. Bland II, and Joel Hooks (to be published by Manning in September 2010).

3.1 Generating the application structure

You need to create a Flex application. Because you'll be using the Flex Mojos Maven plugin you'll be creating the application in a manner similar to what you

FNA (FNA is Not AppFuse)

Folks at Adobe Consulting have started a new project at Google Code called FNA, which stands for FNA is Not AppFuse (http://code.google.com/p/fna-v2/). The FNA project has similar goals to that of AppFuse in that they are attempting to create a framework that enables developers to jump-start their RIA applications with Flex and Java. We have decided against using this framework for this book, but feel the project has potential and warrants a look at if you are starting a new project.

used for the AppFuse portion of the application. We've taken the liberty of creating a Maven archetype to minimize the amount of manual work required to create the project structure.

Let's get started. Open 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. Because this is not a Java project, the project structure varies slightly. 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 here.

Listing 3.1 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 the project has been created you need to configure the Flex Mojos plugins for both the Flex project and the AppFuse project.

3.2 Configuring 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 to avoid writing yet another Ant build script.

Listing 3.2 The pom.xmi for the Flex application

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

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

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

    <finalName>flex-bugs-ria</finalName>
    <plugins>
      <plugin>
        <groupId>org.sonatype.flexmojos</groupId>
        <artifactId>flexmojos-maven-plugin</artifactId>
        <version>${flexmojos.version}</version>
        <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>
      </plugin>
    </plugins>
  </build>
```

1 Define parent pom
2 This module's groupId
3 This module's artifactId
4 Packaging type
5 This module's version
6 Flex-Mojos version
7 Specify source and test source directories
8 Final name for artifact
9 Flex-mojos-plugin

```
  <repositories>
    <repository>
      <id>flexmojos-repository</id>
      <url>http://repository.sonatype.org/content/
      ➥groups/flexgroup/</url>
    </repository>
  </repositories>                                         ⑩ Flex Mojos
  <pluginRepositories>                                       repository at
    <pluginRepository>                                       Sonatype
      <id>flexmojos-repository</id>
      <url>http://repository.sonatype.org/content/
      ➥groups/flexgroup/</url>
    </pluginRepository>
  </pluginRepositories>
  <dependencies>
    <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>                         ⑪ Flex and unit
      <classifier>10</classifier>                              testing
      <type>swc</type>                                         dependencies
    </dependency>
    <dependency>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-unittest-support</artifactId>
      <version>${flexmojos.version}</version>
      <type>swc</type>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Listing 3.2 shows the resulting pom.xml for the Flex application after you generate the project using the `mvn archetype:create` command shown in section 3.1. Because this is part of a multimodule project, the pom.xml lists the parent module's POM as the parent for this project ❶. Next you'll see the values you specified for the `groupId` ❶ and `artifactId` ❶ defined, as well as the `packaging` type ❶ of `swf` because this project is our Flex application and will be compiled to an SWF file.

NOTE You may need to associate the SWF file with the Standalone Flash Player or your build may time out trying to run the `FlexUnit` tests. Because `FlexUnit` requires the Flash runtime 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. You can add a file association in Windows through the Windows Explorer Folder Options and on a Mac by using the file's context sensitive menu. See the documentation athttps://docs.sonatype.org/display/FLEXMOJOS/Running+unit+tests for more information.

Next you set the project's version ❺, which is set to 1.0-SNAPSHOT. We define the final name of our artifact so that the SWF that is generated will not have the version information as part of the filename ❻. The archetype also defines a common property ❻ for the Flex Mojos version so that you can be sure that the plugin ❼ and any dependencies ⓫ defined for the Flex Mojos are using the same version. You're also overriding the version for the Flex compiler here to compile it with the Flex 4 compiler and target version of Flash Player. Because this is not your typical Maven project, the pom.xml defines the source and test-source directory locations ❾. It also defines the repository and plugin repository locations ❿ for the Flex Mojos plugins and dependencies because they don't exist in the central Maven repository.

3.3 Configure Maven for the flex-bugs-web module

Now that you've configured Maven to build the Flex application, you need to make minor modifications to the pom.xml for the `flex-bugs-web` module in order to get the Flex application to be copied over to the appropriate place in the web application. To accomplish this, you'll use the `maven-dependency-plugin`.

Listing 3.3 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">

...
                                              Maven-dependency-plugin ❶
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>unpack-config</id>
            <goals>                               ❷ Unpack-config
              <goal>unpack-dependencies</goal>      execution
            </goals>
            <phase>generate-resources</phase>
            <configuration>
              <outputDirectory>
${project.build.directory}/${project.build.finalName}/WEB-INF/flex
              </outputDirectory>
              <includeGroupIds>${project.groupId}</includeGroupIds>
              <includeClassifiers>resources</includeClassifiers>
              <excludeTransitive>true</excludeTransitive>
              <excludeTypes>jar,swf</excludeTypes>
            </configuration>
          </execution>             Configuration for unpack-config execution ❸
          <execution>
            <id>copy-swf</id>
            <phase>process-classes</phase>                ❹ Copy-swf
            <goals>                                          execution
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <stripVersion>true</stripVersion>
              <outputDirectory>
                ${project.build.directory}/${project.build.finalName}
  </outputDirectory>
              <includeTypes>swf</includeTypes>
            </configuration>
          </execution>
        </executions>
      </plugin>                           Configuration for copy-swf execution ❺
 ...
</project>
```

Listing 3.3 shows the plugin configuration ❶ needed for your web application to properly resolve the dependency for the Flex application and ensure that the SWF file is placed correctly.

First define an execution that you'll call `unpack-config` ❷ and tell Maven to execute this during the `generate-resources` phase of the build, and call the `unpack-dependencies` goal on this plugin. In the configuration ❸ you tell the plugin to limit the scope of what is affected by this execution to artifacts with the same `groupId` as your project and to artifacts of type resources. This will be utilized in chapter 5 as you create a common project for all of the configuration files for BlazeDS that need to be shared between the web application and the Flex application.

The Maven build lifecycle

Maven follows the convention over configuration paradigm in the build lifecycle aspect, and many others. 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, read *Maven: The Definitive Guide*, a free ebook by Eric Redmond available at http://www.sonatype.com/products/maven/documentation/book-defguide.

The second execution that you define ❹ is `copy-swf`, and it will do exactly that. You 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 the Flex project into the target folder to be placed in the proper location before Maven creates the final WAR file ❺. Next let's take a look at how to create an HTML wrapper, or in this case a JSP wrapper for our Flex application.

3.4 Adding a wrapper for our SWF

Adobe provides numerous templates for creating HTML wrapper files for your SWF, located in the /templates directory of your Flex SDK installation. There is everything from the basic no frills wrapper, to wrappers that include functionality to detect whether the client has the correct version of the Flash Player installed, and whether to support deep linking and history for your application.

NOTE Normally the HTML wrapper would be an HTML file in your web application. Because 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 flexbugs.jsp, and replace the placeholders in the file with the values shown in listing 3.4.

Listing 3.4 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 into detail about what is contained in flexbugs.jsp because it's likely you won't have to change anything inside of it in the future. Learn more about the HTML templates that Flex provides in the LiveDocs at Adobe's website at http://livedocs.adobe.com/flex/3/html/wrapper_04.html#178239.

3.5 "Hello World!" in Flex

Now that you have the web application configured to properly resolve the Flex application dependency, have placed the resulting SWF file properly, and have the HTML wrapper configured, let's write a "Hello World!" application in Flex 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 the Flex application.

Listing 3.5 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%">                    ❶ Flex Application
                                                   root component
  <s:layout>
    <s:VerticalLayout/>                          ❷ Setting layout
  </s:layout>

  <s:SimpleText text="Hello World!"/>            ❸ SimpleText
</s:Application>                                   component
```

Shown is a minimal "Hello World!" application in Flex. It consists of only the root `Application` component ❶, a layout definition ❷ and a single `SimpleText` component ❸ which has its text property set to `"Hello World!"`

Maven and heap space

You may run into heap space problems when compiling your Flex application with the Flex Mojos, and your build may fail 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
```

To fix this issue, you need only define an environment variable named `MAVEN_OPTS` and set its value to "`-Xmx512m`" adjusting the memory size as needed (https://docs.sonatype.org/display/FLEXMOJOS/FAQ).

To build the 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 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. After it finishes 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. When you see the output shown in the following listing, you know that your applicatIon is running.

Listing 3.6 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.
```

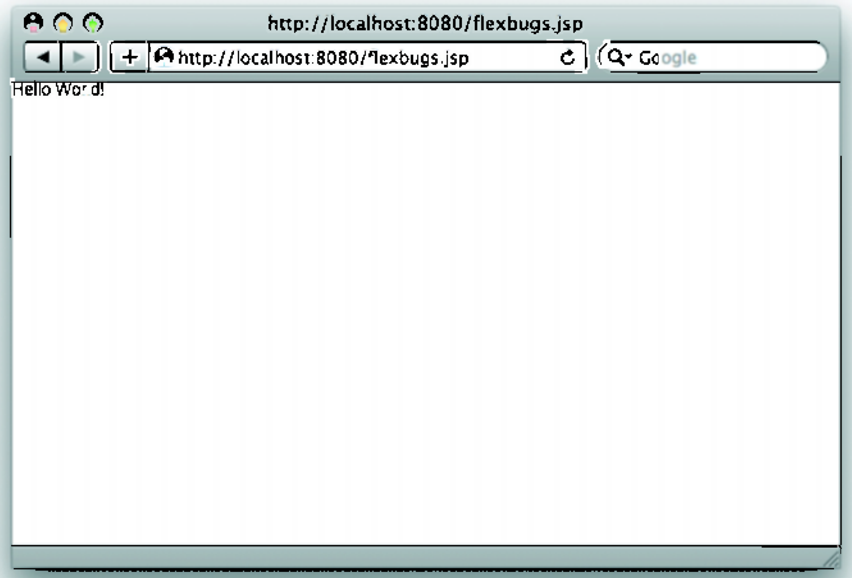
Open your favorite browser and navigate to http://localhost:8080/flexbugs.jsp and you should see a screen similar to figure 3.1.

This sanity check is not terribly exciting, but you can see that the application is configured correctly. Now that you have the obligatory "Hello World!" application out of the way, let's get on with the task of developing the real application.

3.6 Developing the FlexBugs application

To begin developing the application, you should have an idea of what you would like it to look like. Figure 3.2 shows a mockup.

The application is divided into three main areas in a modified master/detail view with a second detail view for any comments on the selected issue. The application can also be divided into header, footer, and main application areas. Defining the application in these terms achieves a couple of objectives. By separating the application into these different pieces, you can create separate mxml files to help keep the code manageable. By breaking up our application into separate mxml files, you can reuse parts of the application.


http://localhost:8080/flexbugs.jsp
http://localhost:8080/flexbugs.jsp
Google

Hello World!

Figure 3.1 Our "Hello World!" application

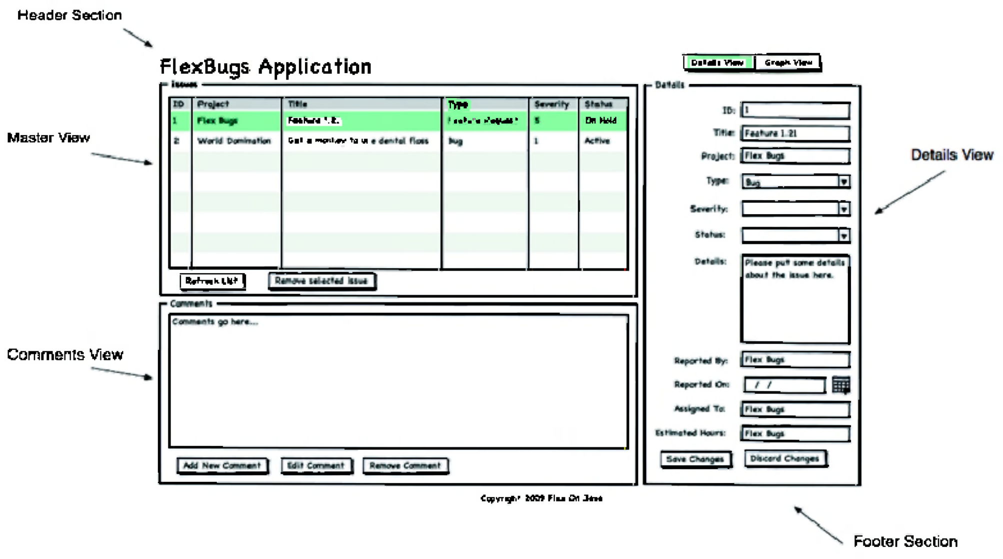


Figure 3.2 A mockup of the FlexBugs application

Next we're going to decompose the application into manageable chunks. 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.

3.6.1 *Introducing ViewStack*

`ViewStack` is a navigation component that allows you to stack a collection of views and selectively display them. Unlike traditional web applications, Flex applications typically don't have many pages. 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 3.3.

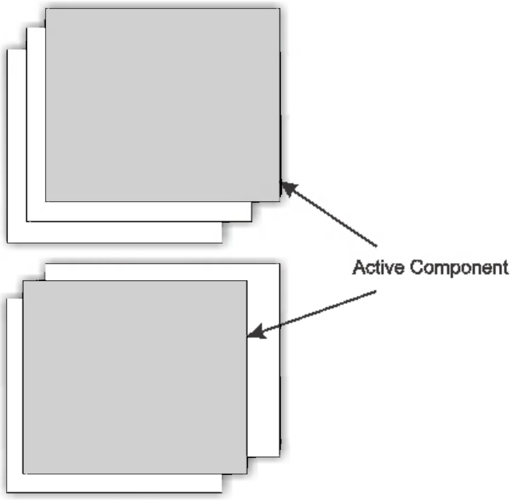


Figure 3.3 How the ViewStack works

You can control the `ViewStack` in a number of ways, the most common of which 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 the figure 3.2 mockup you'll see buttons labeled Details View and Graph View; these are the two views that we'll use of in this application. We'll develop the details view in this chapter and the graph view in chapter 10 when we talk more about graphing components.

Listing 3.7 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/>                                    Define view stack ❶
  </s:layout>

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

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

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

</s:Application>
```

Listing 3.7 shows the Main.mxml after you add the `ViewStack` components to the application. After you remove our `HelloWorld` code, create a `ViewStack` element ❶ and give it an id of `mainViewStack`; this will become important later when you define the `ToggleButtonBar` as the `dataProvider` attribute, for the `ToggleButtonBar` will be set to this `ViewStack` component. You want this `ViewStack` to use up all available horizontal and vertical space so set its width and height to 100%.

Next add two `Canvas` components ❷ and ❸ to the `ViewStack` giving them ids of `view1` and `view2` respectively. These two `Canvas` components will be the two main views the `ToggleButtonBar` will control. The `label` attribute of these two components will be displayed as the text of the two `ToggleButton` controls, so you set those to Details View and Graph View, respectively. Inside these two `Canvas` containers put `Text` components as placeholders, so you can see how the `ToggleButtonBar` will control the two view states in the next section.

3.6.2 *HeaderView*

Before you go much further with building up the application create the header view so you can control the view states that you just created. In the src/main/flex folder create the directory structure shown in figure 3.4 to house the view components.

Figure 3.4 Folder structure for view components

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

Listing 3.8 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">                                    ❶ Component extends Group

  <s:layout>                                             ❷ Define layout
    <s:HorizontalLayout/>
  </s:layout>

  <fx:Script>                                            ❸ Import and declare
    <![CDATA[                                               ViewStack member variable
    import mx.containers.ViewStack;

    public var viewStack:ViewStack;
    ]]>
  </fx:Script>
  <mx:Spacer width="5"/>                                 ❹ Spacers for layout
  <s:SimpleText text="FlexBugs Application"              ❺ Text field for Title
                height="100%"
                fontSize="32"
                fontWeight="bold"
                verticalAlign="middle"/>
  <mx:Spacer width="100%"/>
  <s:VGroup height="100%">
    <mx:Spacer height="100%"/>
    <mx:ToggleButtonBar dataProvider="viewStack"/>       ❻ ToggleButtonBar for
  </s:VGroup>                                               controlling ViewStack
  <mx:Spacer width="5"/>

</s:Group>
```

Listing 3.8 shows the code for the Header.mxml component. There's not much to it. The component itself extends the `Group` component ❶, and defines its layout ❷ as being `HorizontalLayout`, meaning that all the components inside it will be laid out horizontally as opposed to vertically or absolutely. Next you define a public member variable ❸, which will be used to allow the main application to pass in the `ViewStack` that the `ToggleButtonBar` ❻ will control. To do that you create a `Script` block and enclose some ActionScript inside a `CDATA` section, so that any characters that may potentially be parsed as XML are handled correctly.

For the Application Title you have a `SimpleText` component ❼ with a couple of attributes defined on it. The first one is the `text` attribute, which sets the text to be displayed in the application. Flex has support for CSS styles similar to those in web applications. Finally there are `Spacer` elements ❽, which you use to make sure everything is laid out properly. The `Spacer` elements do just what you would expect; they take up space and fill in blank space so that you can effectively lay out components in the application.

Listing 3.9 Adding the header to Main.mxml

```
<?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.*"              ❶ Added namespace
               minWidth="950"                             for view components
               minHeight="600"
               height="100%"
               width="100%">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>                                              ❷ Added header
                                                             to application
  <view:Header viewStack="{mainViewStack}"/>

  <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>
```

The preceding listing shows the updated Main.mxml file that includes the Header.mxml component you just created. First you defined a custom namespace for the view components by adding the code shown at ❶. Next add the 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}` ❷. Now you should be able to build and run the application as outlined earlier, and be presented with a screen that resembles figure 3.5.

When you click the `ToggleButtons` in the upper-right corner, you should see the text in the main part of the application change. Next let's build a simple footer component for the application.

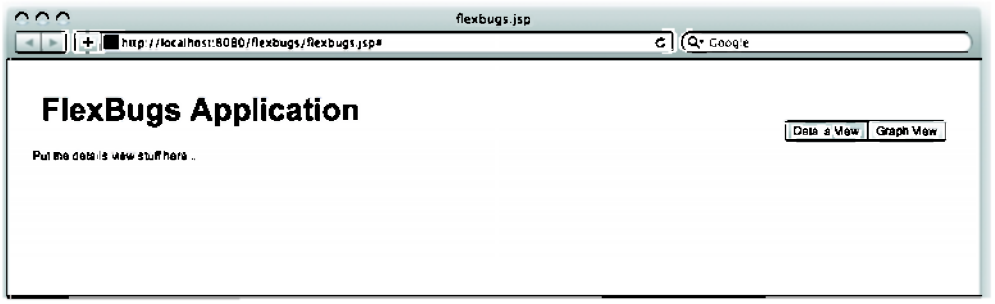

flexbugs.jsp
http://localhost:8080/flexbugs/flexbugs.jsp
Google
FlexBugs Application
Put the details view stuff here...
Data View
Graph View

Figure 3.5 The header view added.

3.6.3 *FooterView*

Inside the same folder where you created the Header.mxml in the previous section, create another file named Footer.mxml. Though it may seem like overkill to separate the 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 3.10 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 &#0169; 2009 Flex On Java"
                height="100%"
                verticalAlign="middle"
                textAlign="center"/>
  <mx:Spacer width="100%"/>

</s:Group>
```

The code in the preceding listing shows the footer file. Like the header, the footer extends the `Group` component. It contains only a single `SimpleText` component, which contains the copyright information, and sets its `textAlign` attribute to `center`. Once again you leverage a couple of `Spacer` elements to assist in layout.

Listing 3.11 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>
                                        ❶ Added footer
  <view:Footer/>                          to Main.mxml

</s:Application>
```

Next you add the footer ❶ to the application much as you did earlier for the header. Near the end of the `Main.mxml`, place the tag for the footer. Next let's move on to creating the `view` component for the master view.

3.6.4 Master view

Now you're getting to the more interesting parts. The master view if you'll recall from figure 3.2 consists of a few components. At the top resides a data grid component with two buttons for adding and removing issues.

Listing 3.12 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%"                                       ❶ Extending
          height="100%">                                       Panel

  <mx:DataGrid id="masterViewDataGrid" width="100%" height="100%">
    <mx:columns>
    <mx:DataGridColumn dataField="id"
                       headerText="ID" width="70"/>       DataGrid ❷
    <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>
                                                       ❸ ControlBar for
  <mx:ControlBar width="100%">                           Add/Delete buttons
    <mx:Button label="Add New Issue"/>
    <mx:Button label="Remove Selected Issue"/>
  </mx:ControlBar>

</mx:Panel>
```

Create a file inside the org/foj/view folder called MasterView.mxml. This will contain the code to create the master view, shown in listing 3.12. This component will be based on the `Panel` component ❶, which is one of the 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.

`DataGrid` ❷ is used to display tabular data, and is one of the fundamental controls used in data-driven applications. The `DataGrid` and its big brother `AdvancedDataGrid` let you edit rows of data within the table cells. For the application, we'll forego that functionality in favor of using a detail view. A `ControlBar` ❸ at the bottom of the `Panel` will contain the buttons for adding and removing issues from the application.

3.6.5 Detail view

Next we're going to develop the detail view, where you will allow the users of the application to modify the data fields for the issues displayed in the master view. `DetailView` has a form containing the fields that can be updated for the issues in the application. Begin just as you did earlier for the master view, by creating a file named DetailView.mxml in the org/foj/view folder. The following listing shows the code we'll be adding to that file.

Listing 3.13 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%">                                 ❶ Extends Panel

  <mx:Form id="issueDetailForm" width="100%">            ❷ Form container
    <mx:FormItem label="ID:" width="100%">
      <mx:Text id="issueId"/>                            ❸ FormItem components
    </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: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>
                                                            ❹ ControlBar
  <mx:ControlBar>                                          <-┘
    <mx:Button id="saveChangesButton" label="Save Changes"/>
    <mx:Button id="cancelChangesButton" label="Cancel Changes"/>
  </mx:ControlBar>

</mx:Panel>
```

Similar to the master view, the details view component will be based off of the `Panel` layout container ❶. Add a `Form` ❷ 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 to submit data to the backend; it's there for aesthetics. Inside the `Form` wrap each input field inside of a `FormItem` ❸ component that provides styling, a label, and layout for each of the form fields. A `ControlBar` ❹ contains the buttons that control saving the data and canceling the edits.

3.6.6 Comments view

The final section of the application we'll lay out is the comments view, which will list any comments added to the issues. First create the CommentsView.mxml file in the org/foj/view folder just as you did for all of the other view components. For this `List` component we're going to display the comment text; you could easily create a custom ItemRenderer for the List items similar to what we'll do in chapter 8 for the custom chart that we'll create.

Listing 3.14 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%"                                  ❶ Extending Panel
          height="100%">

  <mx:List id="commentsList"
          width="100%"                                  ❷ List for comments
          height="100%"
          labelField="commentText">

  </mx:List>
                                                        ❸ ControlBar for buttons
  <mx:ControlBar>
    <mx:Button id="addButton"
               label="Add New Comment"/>
    <mx:Button id="editButton"                          ❹ Buttons for
               label="Edit Comment"/>                     operations on
    <mx:Button id="deleteButton"                          comments
               label="Delete Comment"/>
  </mx:ControlBar>

</mx:Panel>
```

The code for the comments view is based on the Panel component ❶. Next add a List component ❷, which you'll use to contain a list of the comments. At the bottom of the Panel add a ControlBar ❸ to hold the three Button components ❹ we'll define to operate on the comments.

3.7 Laying out the components

Now that you've got all of the components defined, let's add them to the Main.mxml and define the 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 cases it's best to go with either horizontal or vertical layouts, especially if you want your application to resize appropriately. To achieve the layout you want you'll need to nest layout containers as illustrated in figure 3.6.

Figure 3.6 shows the nesting of layout containers that was necessary to duplicate what was shown in the mockup shown in figure 3.1. Listing 3.15 shows the updated Main.mxml with all of the layout components necessary.

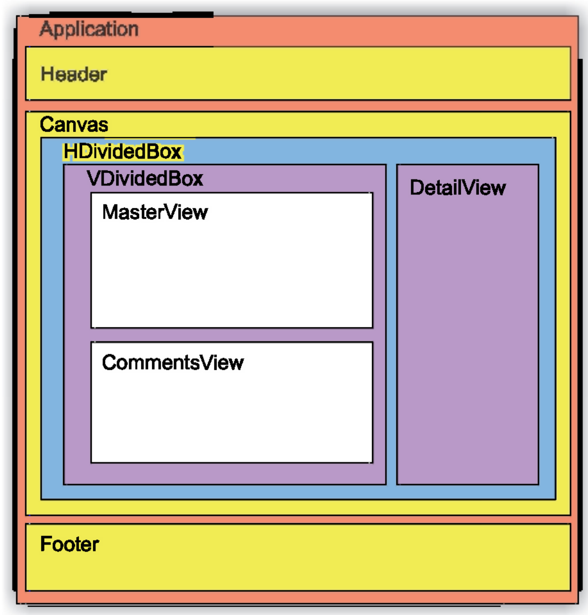


Figure 3.6 Layout for the application

Listing 3.15 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>
    <s:VerticalLayout/>
  </s:layout>

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

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

    <mx:Canvas id="view1" label="Details View">                      ❶ Canvas
      <mx:HDividedBox width="100%" height="100%">                    ❸ HDividedBox
        <mx:VDividedBox width="70%" height="100%">                   ❸ VDividedBox
          <view:MasterView id="masterView" height="60%"/>            ❹ MasterView component
          <view:CommentsView id="commentsView" height="40%"/>        ❸ CommentsView component
        </mx:VDividedBox>
        <view:DetailView id="detailsView" width="30%"/>              ❻ DetailView component
      </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>
```

As illustrated in figure 3.6, you start out with a `Canvas` ❸ container to hold all the nested layout containers for the main `ViewStack`. Inside of this you create an `HDividedBox`, ❸ setting its `width` and `height` to 100%, so it will take up all available space. Within that you place a `VDividedBox` ❸, which will contain the `MasterView` ❸ and the `CommentsView` ❸. Lastly you add the `DetailView` ❸, which will occupy the right side of the `HDividedBox`. The application is almost complete. Next let's create a component to use as a modal popup to edit the comments in the `List` view of the `CommentsView` component.

3.8 Creating a pop-up component

The final component you'll develop in this chapter is a modal pop-up form that you'll wire into the application in the next chapter to add new comments and edit existing ones.

Figure 3.7 is a screenshot of the pop up. Listing 3.16 shows the code for the pop up.

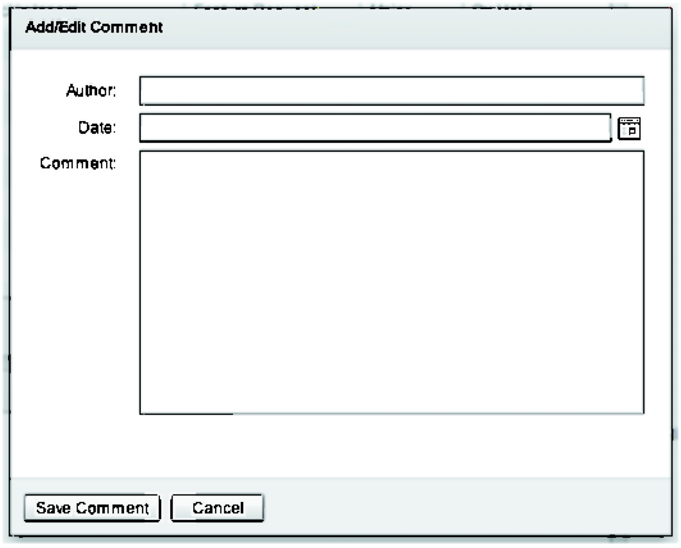


Figure 3.7 Pop up for editing comments

Listing 3.16 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">

  <mx:Form width="100%">
    <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>
    <mx:Button id="saveButton" label="Save Comment"/>
    <mx:Button id="cancelButton" label="Cancel"/>
  </mx:ControlBar>

</mx:TitleWindow>
```

❶ Extends TitleWindow

❷ Add Form

❸ ControlBar

The pop-up window is fairly simple. You define the component to extend the `TitleWindow` component ❶, which you'll typically use for pop ups. Next you add a `Form` and a number of `FormItems` ❷ just as you did earlier for the `DetailView`. The `ControlBar` ❸ added to the window holds the `Button` components for controlling the pop up.

3.9 The finished application

The final design is far from finished, but we're done for now. This is as good a time as any to build the application and see the progress you'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.

From the root of the `flex-bugs-ria` project run the command `mvn install`. The build should complete successfully. If not double-check the code to ensure that it matches the code shown 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 the war file. When the container is up and running, navigate to http://localhost:8080/flexbugs.jsp and you should see something that resembles figure 3.8.

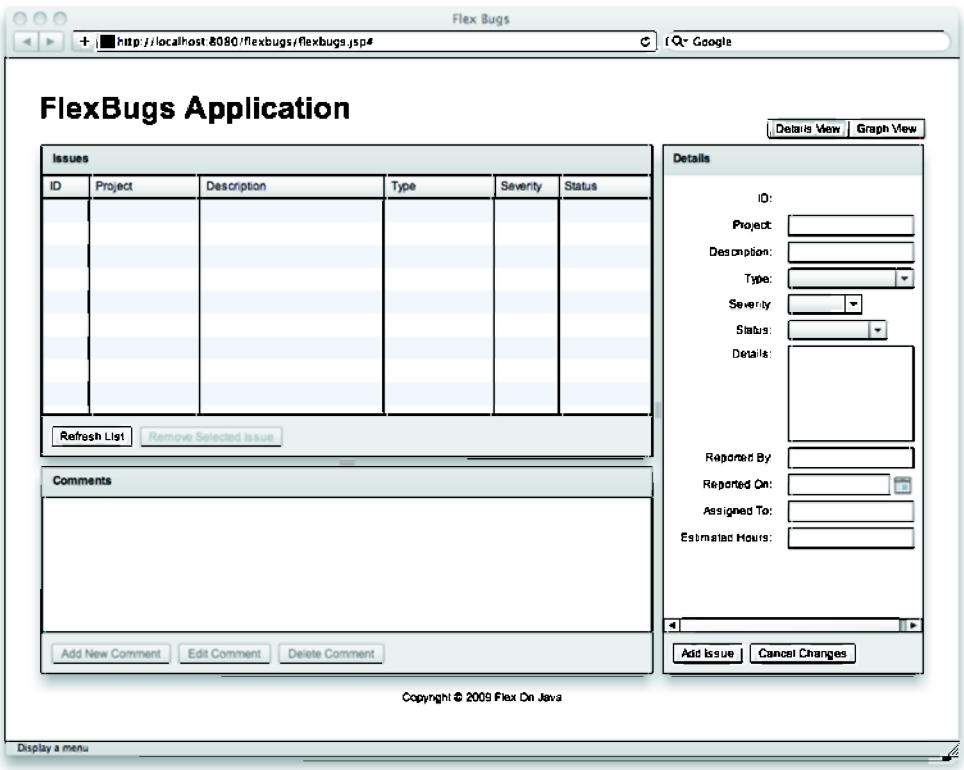


Figure 3.8 The finished application

3.10 Summary

This chapter moved quickly and only briefly introduced many of the components available to you in the Flex framework. You started with an idea of what you wanted the application to look like and decomposed that into several smaller components. By doing that you made the code more manageable, and as you'll see in the next chapter, you made the presentation models and event handling easier to handle. When you had all the pieces built up, you could illustrate basics of layout containers in Flex and put the application together. We're hopeful that you were able to follow along; 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 http://help.adobe.com/en_US/flex/using/index.html.

Now that you've built up the Flex application and laid out the components, where do you go from here? In the next chapter you're going to begin connecting the Flex application to the Java backend you developed in chapter 2 using the `WebService` component. In chapter 5 you'll refactor this to use the BlazeDS framework to talk directly to the Java application.

4 Connecting to web services

This chapter covers

- Model View Presenter
- Event dispatching and handling
- Calling a web service
- Interfaces and views

In chapter 3 you designed the UI for the Flex application, but it won't be useful until it can communicate with a server-side component. In this chapter you're going to continue building up the client-side application. Many Flex books attempt to hide complexity when dealing with client-side code and ActionScript, usually by having you create all your `WebService` components and event handling code in your MXML files. This solution, as many developers agree, does not scale well and quickly shows its warts in any but the most trivial application.

We've decided instead to architect the client-side code in such a manner as to not only scale well, but also isolate your View from Presentation and from external services. This allows the application to be flexible enough to easily refactor and replace one implementation of external service with another with minimal code changes. You could, therefore, painlessly change your application from calling web services to leveraging BlazeDS as you'll see in chapter 5. You'll do this by using MVP.

Because a good portion of the interaction between the Flex framework and the server-side occurs as asynchronous calls, we'll go over events and event handling. We'll cover how to utilize events to facilitate communication between separate sections of the application. We'll also learn how to create the `WebService` component and call the web services you defined in the application in chapter 2. By the end of this chapter you should have a functioning application that communicates with the Java server side.

4.1 Model View Presenter

To build the backend for the Flex application, we're going to make use of a popular GUI architectural pattern called Model View Presenter, and more specifically a variation of that pattern called Passive View. As you may have guessed already, the MVP pattern consists of three main components: a Model, a View, and a Presenter, as shown in figure 4.1. By leveraging the Passive View architecture you are able to create what Michael Feathers describes as a "Humble Dialog Box" (http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf). The primary reason to follow this style of development is that you are able to remove as much logic from the UI as possible

Figure 4.1 shows the relationship among the three components of the MVP pattern. The first element of the MVP triad is the Presenter. As the diagram shows, all information flows from the Presenter to either the Model or View. The Model and View are isolated from each other and should have no knowledge of each other or of the Presenter.

All communication with the Presenter should be done through events. The Presenter is then responsible for maintaining the state of the application, making calls to

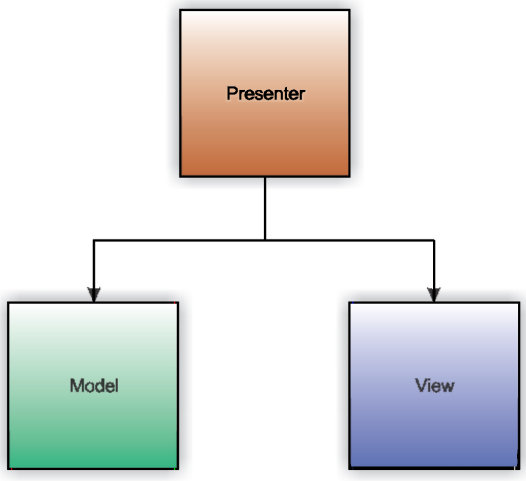


Figure 4.1
The relationship between the Model, View, and Presenter

What about data binding?

For those of you who are familiar with Flex, you'll notice sooner or later that we've decided not to use the built-in data binding functionality that Flex provides. Because we're following the Passive View pattern for this application, we're relying on the Presenter to push any changes to the data being displayed to the view. The main reason is for testability. It's much easier to unit test the presentation behavior if it is in the Presenter than in the View.

the Model and updating the View when necessary. Read more about the MVP pattern and the Passive View pattern at Martin Fowler's blog at (http://martinfowler.com/eaaDev/PassiveScreen.html).

4.2 Web services in Flex

It is fairly common to use web services for application integration, especially when there are disparate platforms involved. Because of the availability of XML parsers for most every common programming language in use today, web services are a prime candidate for applications to share information with each other.

There are two major styles of web services:

- SOAP-based, which are primarily service-oriented
- RESTful, which are primarily resource-oriented

For this application you'll use the `WebService` component to talk to the SOAP-based web services you created in chapter 1. In the example we won't use the `HTTPService` component to connect to RESTful web services, but it's similar enough to using the `WebService` component you should be able to modify the code without too much effort.

Exposing your business functionality via web services has the least impact on your server-side code, as it doesn't couple the application to the clients that will be consuming the service. As you'll see in chapter 5, when we expose a service for consumption using BlazeDS, the remote service is coupled only to clients who can communicate using the AMF binary protocol that BlazeDS uses.

Web services are commonly used to integrate applications especially if they run on different platforms. As it happens, Flex falls into this category because it needs to communicate with the server side, which is most likely not written in ActionScript. With that in mind, the fine folks at Adobe have made integrating a Flex application with a back end using web services a fairly trivial endeavor. Most Flex books show you how to call your web services by creating the service components directly in the MXML files, but because we're striving for reusability, maintainability, and clean code, we're going to encapsulate the `WebService` objects in our model.

4.3 *Dispatching and handling events*

Dispatching and handling events is fundamental to Flex development, and also to development using the Passive View pattern described previously. All interaction with the Presenter is done through dispatching events. In this section we'll discuss how to enable communication between the parts of the application via events.

4.3.1 *Creating a custom event*

Many components in Flex have their own built-in events, such as the click event on a button. If you want to send any data along with your event, such as which row in a DataGrid was selected, you'll need to create your own custom event class that extends the flash.events.Event class.

> **NOTE** Like Java, ActionScript uses packages to logically group related classes and avoid naming conflicts. To declare that a class belongs to a particular package, use the package keyword, followed by the package name with a set of braces containing the code belonging to that package. Then place the class file in a folder structure corresponding to the package name you defined. If your package name is com.example, the class must be located in a com/example folder of your source tree.

To differentiate between events, you can either create a separate subclass for each event that you wish to react to, or create fewer events and use the type attribute to filter out only the events you want to add event listeners for.

Listing 4.1 UIEvent.as

```
package org.foj.event {

import flash.events.Event;

public class UIEvent extends Event {                    ❶ Extending Event

  public static var SELECTED_ISSUE_CHANGED:String =      ❷ Constants for event type
      "selectedIssueChanged";
  public static var REFRESH_ISSUES_BUTTON_CLICKED:String =
      "refreshButtonClicked";
  public static var REMOVE_ISSUE_BUTTON_CLICKED:String =
      "removeButtonClicked";
  public static var SAVE_ISSUE_BUTTON_CLICKED:String =
      "saveIssue";
  public static var SELECTED_ISSUE_SAVED:String =
      "selectedIssueSaved";
  public static var CANCELLED_ISSUE_EDIT:String =
      "cancelledIssueEdit";

  public static var SELECTED_COMMENT_CHANGED:String =
      "selectedCommentChanged";
  public static var ADD_NEW_COMMENT_BUTTON_PRESSED:String =
      "addNewComment";
  public static var EDIT_COMMENT_BUTTON_PRESSED:String =
      "editComment";
  public static var DELETE_COMMENT_BUTTON_PRESSED:String =
      "deleteComment";

  public static var SAVE_COMMENT_BUTTON_PRESSED:String =
      "saveComment";
  public static var CANCEL_EDIT_COMMENT_BUTTON_PRESSED:String =
      "cancelEditComment";                                          ❷ Constants for
  public static var COMMENTS_UPDATED:String =                          event type
      "commentsUpdated";

  public var data : *;                                                  Variable for
                                                                        sending
  public function UIEvent(type : String,                                information
                          bubbles : Boolean = true,                   ❸ with event
                          cancelable : Boolean = false)
  {                                                                 ❹ Constructor
    super(type, bubbles, cancelable);
  }

}
}
```

In listing 4.1 you've created a custom event class called `UIEvent` ❶ and defined constants for the different types of events that will be dispatched ❷. There's also a defined field called `data` ❸ that will accept any type of variable. This will hold any kind of payload you pass with the event. As a last step overload the constructor ❹ and specify default values for whether or not the events should bubble up the chain and be cancelable. Next you'll look at how to dispatch these events.

4.3.2 *Event dispatching*

For the application to react to events, you first have to dispatch them. Because every component in Flex is an ancestor of `UIComponent`, you have the ability to dispatch an event by simply calling the `dispatchEvent` method within your MXML; event bubbling only travels upward toward a component's parent. For the application to work as you expect, you need a mechanism to allow sibling components to intercept events being dispatched to notify you of a button being clicked, or the selected item in a `DataGrid` being changed. To accomplish this you will create an `EventDispatcher-Factory` so that all event dispatching and subscribing happens with the same `Event-Dispatcher` object.

Listing 4.2 EventDispatcherFactory.as

```
package org.foj.event {

import flash.events.EventDispatcher;

public class EventDispatcherFactory{                             ❶ Singleton instance
  private static var _instance:EventDispatcher;                    of EventDispatcher
  public static function getEventDispatcher():EventDispatcher {    <-- Factory method to get EventDispatcher ❷
    if (_instance == null) {
      _instance = new EventDispatcher();
    }
    return _instance;
  }
}
}
```

Listing 4.2 shows the code for the `EventDispatcherFactory`. There's not much to it. It consists of a factory method `getEventDispatcher` ❷, which returns a singleton instance of an `EventDispatcher` ❶. You'll use this instance of `EventDispatcher` to dispatch and subscribe to events. That way any part of the application can dispatch and react to events regardless of where it exists in the application's hierarchy of components and classes. Now that you have the custom events and a way to dispatch them, let's move on to enhancing the application to make it more interactive and useful.

4.4 Creating Issue and Comment transfer objects

To ease handling the responses from calling web services, you need to duplicate the data objects defined in chapter 1 in ActionScript so that you can effectively deal with the data on the client side.

Listing 4.3 Issue.as

```
package org.foj.dto {

public class Issue {

  public var id:int;                        <-- ❶ Public member variables
  public var project:String;
  public var description:String;
  public var type:String;
  public var severity:String;
  public var status:String;
  public var details:String;
  public var reportedBy:String;
  public var reportedOn:Date;
  public var assignedTo:String;
  public var estimatedHours:Number;

  public function Issue(issue:* = null) {   <-- ❷ Convenience constructor
    if (issue != null) {

      this.id = issue.id;
      this.project = issue.project;
      this.description = issue.description;
      this.type = issue.type;
      this.severity = issue.severity;
      this.status = issue.status;
      this.reportedBy = issue.reportedBy;
      this.reportedOn = issue.reportedOn;
      this.assignedTo = issue.assignedTo;
```

```
            this.estimatedHours = issue.estimatedHours;
            this.details = issue.details;
        }
    }
}
}
```

Listing 4.3 shows the Issue data object. There's not much to it. It contains several public member variables ❶, which match the properties in its Java counterpart. Then you define a convenience constructor ❷ for constructing an Issue because the WebService will not return actual Issue objects, but rather construct an ObjectProxy, which is an anonymous dynamic object that contains the same properties as the Issue object. You define a constructor that allows you to create an Issue object from this ObjectProxy.

Listing 4.4 Comment.as

```
package org.foj.dto {

public class Comment {

  public var id:int;                                  ❶ Public member
  public var issue:Issue;                               variables
  public var author:String;
  public var createdDate:Date;
  public var commentText:String;

  public function Comment(comment:* = null) {        ❷ Convenience
    if (comment != null) {                              constructor
      this.id = comment.id;
      this.author = comment.author;
      this.createdDate = comment.createdDate;
      this.commentText = comment.commentText;
      this.issue = new Issue(comment.issue);
    }
  }

}
}
```

Listing 4.4 shows the Comment data object, which looks similar to the Issue data object. It contains a few member variables ❶ and a convenience constructor ❷. Next let's get down to the task of enhancing the UI by applying the MVP pattern to the application.

4.5 Enhancing the master view

The first part of the application that you'll be enhancing is the master view. You'll start by creating the Presenter for the master view. Then you'll create the Issues model, which will encapsulate the interaction with the web services you created in chapter 1. When you're done, update the view to dispatch the necessary events to function properly.

4.5.1 *Creating a Presenter for the master view*

Because the Presenter is responsible for most of what goes on in the application, it will be the first part of the MVP triad that you'll be creating. The Presenter for the master view part of the application needs only a couple of different events to react to. Because the application is stateful by nature, you need to refresh the list of `Issues` whenever any of the issues are changed, that is, an issue is created or removed, whenever an individual issue is updated, or when the user clicks the Refresh Issues button. The Presenter also needs to react when the user clicks the Remove Issue button. The following listing shows the code for the `MasterPresenter` class.

Listing 4.5 MasterPresenter.as

```
package org.foj.presenter {

...

public class MasterPresenter {
                                                           1 View
  private var _view:MasterViewComponent;                          2 Model
  private var _issueModel:SoapIssueModel;

  public function MasterPresenter(view:MasterViewComponent = null) {
    this._issueModel = new SoapIssueModel();                     Constructor 3
    this._view = view;

    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.REFRESH_ISSUES_BUTTON_CLICKED,
                          getIssues);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SELECTED_ISSUE_SAVED,
                          getIssues);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.CANCELLED_ISSUE_EDIT,
                          getIssues);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.REMOVE_ISSUE_BUTTON_CLICKED,
                          removeIssue);
  }                                                     Subscribing to events 4

  private function getIssues(event:UIEvent = null):void {          Getting
    CursorManager.setBusyCursor();                              5 the issues
    _issueModel.getIssues(new AsyncResponder(getIssuesResult,
                                             handleError));
  }

  private function removeIssue(event:UIEvent):void {             Removing
    CursorManager.setBusyCursor();                              6 an issue
    var selectedIssue:Issue = event.data;
    _issueModel.removeIssue(selectedIssue.id,
        new AsyncResponder(removeIssueResult, handleError));
  }                                                          Responding to 7
                                                           getIssues result
  private function getIssuesResult(event:ResultEvent,
                                   token:AsyncToken = null):void {
    CursorManager.removeBusyCursor();
    var issues = new ArrayCollection();

    for each(var item:Object in event.result) {
      var issue:Issue = new Issue(item);
      issues.addItem(issue);
    }

    _view.masterViewDataGrid.dataProvider = issues;
  }

  private function removeIssueResult(event:ResultEvent,
                                     token:AsyncToken = null):void {
    CursorManager.removeBusyCursor();
    _view.masterViewDataGrid.selectedIndex = -1;
    var itemChangedEvent:UIEvent =
        new UIEvent(UIEvent.SELECTED_ISSUE_CHANGED);
    itemChangedEvent.data = new Issue();
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(itemChangedEvent);
    getIssues();
  }

  private function handleError(event:FaultEvent,
                               token:AsyncToken = null):void {
    CursorManager.removeBusyCursor();
    Alert.show(event.message.toString());
  }

}
}
```

Responding to removeIssue result ❽

Responding to fault event ❾

The `MasterPresenter` contains references to the other two pieces of the MVP triad, the `view` ❼ and the `model` ❸. The constructor ❸ takes as an argument its `view`. These actlons are necessary because you'll be constructlng the Presenters in the view components as they're created. The `view` will then pass in a reference to itself when creating the Presenter. You then create an instance of the Model you'll be calling and register a few event listeners for the events that the Presenter needs to react to ❺.

Methods are called as a result of the events being fired by the View. The first method `getIssues` ❺ is called to react to events that require the applicatlon to refresh the `DataGrid`. It first calls the `CursorManager` to change the mouse cursor to the *busy* cursor. Then it calls the Model's `getIssues` method, passing in an `Async-Responder`, which is a way to provide callback methods to the Model to call when a result is returned.

When a `ResultEvent` is fired by the Model, it calls the `getIssuesResult` ❼ callback method, which changes the mouse cursor back to the normal cursor and updates the `DataGrid` by settlng its `dataProvider` property to the collection of `Issues` returned from the Model. As stated earlier, the web service returns a collectlon of `ObjectProxy` objects you need to iterate through to create actual `Issue` objects for the view.

There are also methods to handle when the user clicks on the Remove Issue button ❸ and its callback for a successful return ❽. A simple and generic error handling functlon ❸ returns the cursor to the normal state, and pops up an `Alert` box

WebServices, AsyncToken, and IResponder

In Flex, calls to remote services happen asynchronously, and return an `AsyncToken` object that you can use to add callbacks to be executed when either a `ResultEvent` or `FaultEvent` returns as a result of the remote service call. One way to bind these event handlers to an `AsyncToken` is to add an object that conforms to the `IResponder` interface, such as an `AsyncResponder`, which takes in a callback to call on a `ResultEvent` and one to call if it returns a `FaultEvent` instead in its constructor.

containing the error message returned from the remote call. Obviously you'll want to replace this simplistic error handling when developing real applications, but for the simple example, this will suffice. Now let's create the Model component.

4.5.2 *Creating an Issue Model*

The next component in the MVP triad that you'll implement is the Model. The Model in MVP, sometimes referred to as the Domain Model, represents the core of the business domain and is often responsible for manipulating data in a relational database, or in this case calling an external service that will handle persisting the data.

Listing 4.6 IssueModel.as

```
package org.foj.model {

import mx.rpc.AsyncToken;
import mx.rpc.IResponder;
import mx.rpc.soap.WebService;
import org.foj.dto.Issue;

public class IssueModel {

  private var _issueService:WebService;

  public function IssueModel() {
    _issueService = new WebService();                         ① Create new WebService
    _issueService.wsdl =                                      ② Set its wsdl property
        "http://localhost:8080/services/IssueService?wsdl";
    if (_issueService.canLoadWSDL()) {                        ③ Load the wsdl
      _issueService.loadWSDL();
    }
  }

  public function getIssues(responder:IResponder):void {      ④ getIssues
    var token:AsyncToken = _issueService.getAll();
    token.addResponder(responder);
  }

  public function removeIssue(id:Number, responder:IResponder):void {   ⑤ removeIssues
    var token:AsyncToken = _issueService.remove(id);
    token.addResponder(responder);
  }

}
}
```

Listing 4.6 shows the code for the IssueModel so far. In the constructor, you create a WebService object ❶ and set its wsdl property to http://localhost:8080/services/IssueService?wsdl ❷.

Using WebService destinations

If you prefer not to hardcode the WebService URL inside your Model classes, you can set up the WebService destination metadata in the services-config.xml configuration file. Using this approach enables you to refer to the WebService by its destinationID in order to load the WSDL rather than using a URL to the WSDL.

To use this approach you'll need to replace the sections of code where you set the wsdl property on your WebService objects like _issueService.wsdl with code that sets the destination property to the destination name you configured in the services-config.xml. By leveraging Maven profiles along with its resource filtering you can put property placeholders in your services-config.xml and define different destinations for your web services in the Maven profiles for the different environments. For more information on using Maven profiles and resource filtering, refer to the Maven reference at http://www.sonatype.com/books/mvnref-book/reference/public-book.html. We'll cover working with destinations in chapter 5 when discussing BlazeDS remoting.

Next you need to call the loadWSDL method ❸ on the WebService so that the WebService will download the WSDL and be able to make calls against the web service. This is a necessary step when defining your WebService in ActionScript as opposed to using the WebService MXML tag. As the final steps, you define the two business methods getIssues ❹ and removeIssue ❺, which call the web service and add the responder callback you passed in from the Presenter as a responder to the AsyncToken. Let's wrap up this MVP triad by refactoring the MasterView.

4.5.3 *Updating the master view*

Now that we've got the Presenter and Model in place, let's update the MasterView to add functionality.

Listing 4.7 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%"
          creationComplete="init()">                 ❶ creationComplete
                                                        event
  <fx:Script>
  <![CDATA[
  import mx.events.ListEvent;
  import org.foj.dto.Issue;
  import org.foj.event.EventDispatcherFactory;
  import org.foj.event.UIEvent;
  import org.foj.presenter.MasterPresenter;

  private var presenter:MasterPresenter;

  private function init():void {                          2 Bootstrapping
    presenter = new MasterPresenter(this);                  Presenter
    refreshList();
  }                                                       3 refreshList

  private function refreshList():void {
    var refreshEvent:UIEvent =
        new UIEvent(UIEvent.REFRESH_ISSUES_BUTTON_CLICKED);
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(refreshEvent);
  }                                               selectedItemChanged 4

  private function selectedItemChanged(event:ListEvent):void {
    var itemChangedEvent:UIEvent =
        new UIEvent(UIEvent.SELECTED_ISSUE_CHANGED);
    itemChangedEvent.data = masterViewDataGrid.selectedItem as Issue;
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(itemChangedEvent);
  }

  private function removeSelectedIssue():void {
    var selectedIssue:Issue =                            5 removeSelectedIssue
        masterViewDataGrid.selectedItem as Issue;
    var removeEvent:UIEvent =
        new UIEvent(UIEvent.REMOVE_ISSUE_BUTTON_CLICKED);
    removeEvent.data = selectedIssue;
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(removeEvent);
  }

]]>
</fx:Script>

<mx:DataGrid id="masterViewDataGrid"
             width="100%"
             height="100%"                                6 datagrid
             itemClick="selectedItemChanged(event);">       itemClick event
  <mx:columns>
    <mx:DataGridColumn dataField="id"
                       headerText="ID" width="50"/>
    <mx:DataGridColumn dataField="project"
                       headerText="Project" width="120"/>
    <mx:DataGridColumn dataField="description" width="200"
                       headerText="Description"/>           7 DataGridColumn
    <mx:DataGridColumn dataField="type"                        bindings
                       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%">
    <mx:Button label="Refresh List" click="refreshList()"/>          ❸ Refresh List click event
    <mx:Button label="Remove Selected Issue"
               click="removeSelectedIssue()"
               enabled="{masterViewDataGrid.selectedItem != null}"/>
  </mx:ControlBar>                                                    Remove Selected Issue click event ❸

</mx:Panel>
```

Listing 4.7 shows the updated MasterView.mxml. The first change is the addition of a handler for the `creationComplete` event of the component ❶. You use this event to help bootstrap the Presenter ❸ and make a call to the `refreshList` ❸ method to populate the `DataGrid` automatically on startup. The `refreshList` method is used to respond to the click event on the Refresh List button as well ❸. Next you define an event handler method called `selectedItemChanged` ❸, which responds to any clicks within the `DataGrid` ❸.

You also define an event handler method called `removeSelectedIssue` ❸ to respond to any clicks of the Remove Selected Issue button ❸. To ensure that this button is enabled only when we have an actively selected row in the `DataGrid`, we've bound the enabled property of the button to the `selectedItem` property of the `DataGrid`. We've bound the `DataGridColumn` items ❸ in the `DataGrid` to the properties defined on the `Issue` data object earlier. This will allow the `DataGrid` to bind the column values to the objects in its data provider.

4.6 *Enhancing the detail view*

The next MVP triad you'll create is for the detail view. Just as you did in the previous section, you'll begin by creating the Presenter, then add the necessary methods to the `IssueModel`, and finish by enhancing the View.

4.6.1 *Creating a DetailPresenter*

The `DetailPresenter` must maintain more state than the `MasterPresenter`. Because the code necessary for the `DetailPresenter` is more complex, we'll break it up into smaller chunks, starting with listing 4.8.

Listing 4.8 DetailPresenter.as

```
package org.foj.presenter {

...

public class DetailPresenter {

  private var _issue:Issue;                 ❶ Selected issue
  private var _view:DetailView;             ❷ View
  private var _issueModel:IssueModel;       ❷ Model

  public function DetailPresenter(view:DetailView) {
    this._issueModel = new SoapIssueModel();
    this._view = view;
    this._issue = new Issue();

    view.issueTypes = new ArrayCollection(
        ["Bug", "Feature Request", "Enhancement"]
        );
    view.severityTypes = new ArrayCollection(
        ["Minor", "Major", "Severe"]
        );
    view.statusTypes = new ArrayCollection(
        ["Open", "In Progress", "On Hold", "Finished"]
        );

    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SELECTED_ISSUE_CHANGED,
                          changedIssue);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SAVE_ISSUE_BUTTON_CLICKED,
                          saveIssue);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.CANCELLED_ISSUE_EDIT,
                          cancelChanges);
  }
...
```

❹ **Initialize components**

❺ **Populate combo boxes**

❻ **Add event listeners**

The `DetailPresenter` not only holds references to the View ❸ and Model ❸, it also needs to hold a reference to the currently selected issue ❶ from the `DataGrid` in the `MasterView`. In the constructor you initialize the Model, set the View to what is being passed in, and create an empty `Issue` object ❸. Next the Presenter sets the possible values in the combo boxes for the issue types, severity, and status ❸ by creating `Array-Collections` with the possible values and setting the `dataProvider` property of the corresponding `ComboBox`. Then you add event listeners ❻ to respond to the events that the `DetailView` will fire, as well as the changed issue event that the `MasterView` fires when the selected item in the `DataGrid` changes. The following listing shows the rest of the code for the `DetailPresenter`.

Listing 4.9 DetailPresenter.as (continued)

```
...
private function get selectedIssue():Issue {
    return this._issue;
  }

  private function set selectedIssue(issue:Issue):void {
    this._issue = issue;
    _view.issueId = issue.id == 0 ? "" : issue.id.toString();
    _view.project = issue.project;
    _view.description = issue.description;
    _view.type = issue.type;
    _view.severity = issue.severity;
    _view.status = issue.status;
```

❶ **Get property for selected issue**

❷ **Set property for selected issue**

```
    _view.details = issue.details;
    _view.reportedBy = issue.reportedBy;
    _view.reportedOn = issue.reportedOn;
    _view.assignedTo = issue.assignedTo;
    _view.estimatedHours = isNaN(issue.estimatedHours) ?
                           "" : issue.estimatedHours.toString();
    _view.addEditLabel = issue.id == 0 ?
                         "Add Issue" : "Save Issue";
  }

  private function saveIssue(event:UIEvent = null):void {      ❸ Save issue
    _issue.project = _view.project;
    _issue.description = _view.description;
    _issue.type = _view.type;
    _issue.severity = _view.severity;
    _issue.status = _view.status;
    _issue.details = _view.details;
    _issue.reportedOn = _view.reportedOn;
    _issue.reportedBy = _view.reportedBy;
    _issue.assignedTo = _view.assignedTo;
    _issue.estimatedHours = !isNaN(Number(_view.estimatedHours)) ?
                            Number(_view.estimatedHours) : null;

    _issueModel.saveIssue(_issue,
        new AsyncResponder(saveIssueResult, handleError));
  }

  private function cancelChanges(event:UIEvent = null):void {   ❹ Cancel changes
    selectedIssue = new Issue();
  }

  private function changedIssue(event:UIEvent):void {          ❺ Selected Issue changed
    selectedIssue = event.data;
  }

  private function saveIssueResult(resultEvent:ResultEvent,     Save issue result ❻
                                   token:AsyncToken = null):void {
    selectedIssue = new Issue(resultEvent.result);
    var event:UIEvent = new UIEvent(UIEvent.SELECTED_ISSUE_SAVED);
    event.data = _issue;
    EventDispatcherFactory.getEventDispatcher().dispatchEvent(event);
  }

  private function handleError(event:FaultEvent,
                               token:AsyncToken = null):void {
    Alert.show(event.message.toString());                       Error handler ❼
  }
 }
}
```

You then create get and set properties ❶, ❷ for the currently selected issue. The get property will return the current issue; the set property will update all the form fields in the view so that they match the currently selected issue.

Next you've defined a method to handle saving the `Issue` ❸, which gets the values from the view and updates the currently selected `Issue` before calling the `saveIssue`

Properties in ActionScript

ActionScript, unlike Java, has first-class support for properties similar to other languages such as C#, Ruby, and Groovy. This gives you the ability to get and set values on a variable as if it were a public member variable on the class, instead of having to do `getXXX()` and `setXXX()` methods as you would in Java.

The syntax for defining get/set properties is as follows:

```
public function get name():String {
  return _name;
}
public function set name(value:String):void {
  _name = value;
}
```

Many developers use the convention of prefixing the private member variables with an underscore (_) so that the compiler can figure out if you are trying to reference the private member variable or the get/set property in the class. One of the advantages of using get/set properties is that it removes the need to define a lot of `getXXX/setXXX` methods that do nothing beyond getting and setting the private variable. Using properties allows you to define your member variables as public. If you need to encapsulate logic when member variables are accessed, you can easily refactor your class to use a get/set property, and none of the classes that use it will have to change as a result.

method on the model. The method that responds to the Cancel Changes button being clicked ❼ sets the selected issue to a brand new `Issue` object. Whenever the `SELECTED_ISSUE_CHANGED` event is fired, the `changedIssue` ❺ method is called and sets the selected `Issue` by using the set property you defined earlier. You create the handlers for the `saveIssueResult` ❻ and the error handler ❼ to handle the result from calling the `IssueModel`.

4.6.2 *Updating the IssueModel*

Now that you've implemented the Presenter for the details view, let's do the same for the new methods in the `IssueModel` to support the updated functionality.

Listing 4.10 IssueModel.as

```
package org.foj.model {

...                                                                  getIssue ❶
  public function getIssue(id:Number, responder:IResponder):void {     <-┘
    var token:AsyncToken = _issueService.get(id);
    token.addResponder(responder);
  }                                                                 saveIssue ❷
  public function saveIssue(issue:Issue, responder:IResponder):void {  <-┘
    var token:AsyncToken = _issueService.save(issue);
```

```
        token.addResponder(responder);
      }

    }

  }
```

Just as before, the methods in the `IssueModel` are fairly simple. The `getIssue` ❶ method takes an `id` parameter to find a specific `Issue` as well as an `IResponder` to add a responder to the token returned from the call to the `WebService`. The `saveIssue` ❷ method takes in an `Issue` object to pass along to the `WebService` to persist, as well as an `IResponder`.

4.6.3 Updating the detail view

The last part of the MVP triad that needs to be updated now is the `DetailView`. Listing 4.11 shows the updated version of the `DetailView.mxml` after you add the necessary methods to dispatch the events to save an `Issue` or cancel the changes.

Listing 4.11 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%"                           ❶ creationComplete
          creationComplete="init()">

  <fx:Script>
  <![CDATA[

    import mx.collections.ArrayCollection;

    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.UIEvent;
    import org.foj.presenter.DetailPresenter;

    private var _presenter:DetailPresenter;          ❷ initialization

    private function init():void {
      _issueType.selectedIndex = -1;
      _issueSeverity.selectedIndex = -1;
      _issueStatus.selectedIndex = -1;
      _presenter = new DetailPresenter(this);
    }                                                      ❸ saveChanges

    private function saveChanges():void {
      var saveEvent:UIEvent = new UIEvent(UIEvent.SAVE_ISSUE_BUTTON_CLICKED);
      EventDispatcherFactory.getEventDispatcher().dispatchEvent(saveEvent);
    }                                                      ❹ cancelChanges

    private function cancelChanges():void {
      var cancelEvent:UIEvent = new UIEvent(UIEvent.CANCELLED_ISSUE_EDIT);
      EventDispatcherFactory.getEventDispatcher().dispatchEvent(cancelEvent);
    }
  ]]>

  </fx:Script>

  <mx:Form id="_issueDetailForm" width="100%">
    ...
  </mx:Form>

  <mx:ControlBar>
    <mx:Button id="_saveChangesButton"
               label="Add Issue"
               click="saveChanges()"/>                ❺ click event on
                                                         saveChangesButton
    <mx:Button id="_cancelChangesButton"
               label="Cancel Changes" click="cancelChanges()"/>
  </mx:ControlBar>
                                                      click event on
</mx:Panel>                                           cancelChangesButton ❻
```

You tell the component to call the `init` ❷ method to initialize the component when the `creationComplete` ❶ event is fired. Inside the `init` method, you bootstrap the Presenter just as you did earlier in the `MasterView`. You also set the `selectedItem` property on the three combo boxes to `-1`, so they don't have a value selected when the component is first created. Then you add an event handler for the click event on the `saveChangesButton` ❺ called `saveChanges` ❺. Inside the `saveChanges` method you dispatch an event signaling that the button has been clicked. As a final step, you add an event handler `cancelChanges` ❺ for the click event on the `cancel-ChangesButton` ❺ and it dispatches an event notifying the application that the button had been clicked.

4.7 Enhancing the comments view

You've almost finished enhancing the sample application. The last part that needs to be updated is the comments view. You'll start by creating an MVP triad just as before. Later you'll add a modal pop up to add new comments and edit existing comments. Let's get started.

4.7.1 Creating a comments presenter

Creating the Presenter for the comments view of the application will be just like creating the last two Presenters. The `CommentsListPresenter` will be responsible for maintaining more state than the previous Presenters. It will need to maintain a reference to not only the currently selected issue, but also the currently selected comment, to properly persist the `Comment` objects. Because the code for the `Comments-ListPresenter` is rather lengthy, we'll break it up. The following listing shows the first section in which you define which events this Presenter will listen for and a couple of get/set properties.

Listing 4.12 CommentsListPresenter.as

```
package org.foj.presenter {

...

public class CommentsListPresenter {

  private var _selectedIssue:Issue;                          ❶ Maintaining state
  private var _selectedComment:Comment;
  private var _commentModel:CommentModel;
  private var _view:CommentsView;
  private var _popl:EditCommentForm;

  public function CommentsListPresenter(view:CommentsView = null) {
    this._commentModel = new CommentModel();
    this._view = view;                                       Constructor ❷

    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SELECTED_ISSUE_CHANGED,    ❸ Event listeners
                          changeSelectedIssue);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.DELETE_COMMENT_BUTTON_PRESSED,
                          removeComment);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SELECTED_COMMENT_CHANGED,
                          changeSelectedComment);
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.COMMENTS_UPDATED,
                          refreshComments);

  }                                                          ❹ get property for selectedIssue

  private function get selectedIssue():Issue {
    return this._selectedIssue;
  }                                                          ❺ set property for selectedIssue

  private function set selectedIssue(issue:Issue):void {
    this._selectedIssue = issue;
    _view.addCommentButton.enabled = issue.id > 0;
  }                                                          ❻ get property for selectedComment

  private function get selectedComment():Comment {
    return this._selectedComment;
  }

  private function set selectedComment(comment:Comment):void {
    this._selectedComment = comment;
    _view.editCommentButton.enabled = comment != null;       set property for selectedComment ❼
    _view.deleteCommentButton.enabled = comment != null;
  }

...
```

First you define variables to maintain the state ❶. Then you define a constructor ❷ that takes in a reference to the View that this Presenter is created for. Next you initialize the Model, and add a few event listeners ❸ for the Presenter so it can react properly to specific events.

Once again you're leveraging get and set properties in this Presenter, using the `set` property to trigger whether or not certain elements of the view should be enabled. You start by creating a `get` property for the currently selected issue ❸, and in its corresponding `set` property ❺, you determine whether or not to enable the Add Comments button based on whether or not the currently selected issue's `id` property is greater than zero, indicating it has been saved. You do this to make sure that the user can't save a comment for an `Issue` that has not been persisted to the backend previously. You create a `get` property for the currently selected comment ❻ as well as a `set` property for the selected comment ❼. Similar to the last `set` property, you use this opportunity to enable or disable the Edit Comment and Delete Comment buttons based on whether or not there is a comment selected in the `List` component containing the comments.

Listing 4.13 CommentsListPresenter.as (continued)

```
...
  private function changeSelectedIssue(event:UIEvent):void {          <-┐
    selectedIssue = event.data;                                 Handler for |
    selectedComment = null;                          selectedIssue changing ❶
    refreshComments();
  }

  private function changeSelectedComment(event:UIEvent):void {         <-┐
    selectedComment = event.data;                                Handler for |
  }                                                selectedComment changing ❷

  private function removeComment(event:* = null):void {    <-┐
    CursorManager.setBusyCursor();                             ❸ removeComment
    _commentModel.removeComment(selectedComment.id,
        new AsyncResponder(removeCommentResult, handleError));
  }

  private function refreshComments(event:* = null):void {    <-┐ refresh
    CursorManager.setBusyCursor();                             ❹ comments list
    _commentModel.getCommentsForIssueId(selectedIssue.id,
        new AsyncResponder(loadCommentsResult, handleError));
  }

  private function removeCommentResult(event:ResultEvent,
                                       token:AsyncToken = null):void {  <-┐
    CursorManager.removeBusyCursor();                                      |
    refreshComments();                                   removeCommentResult ❺
  }

  private function loadCommentsResult(event:ResultEvent,
                                      token:AsyncToken = null):void {  <-┐
    CursorManager.removeBusyCursor();                                     |
    var comments:ArrayCollection = new ArrayCollection();                 |
    for each (var result:* in event.result) {                             |
      comments.addItem(new Comment(result));             loadCommentsResult ❻
    }
    _view.commentsList.dataProvider = comments;
    _view.commentsList.selectedIndex = -1;
    selectedComment = null;
  }

  private function handleError(event:FaultEvent,
                               token:AsyncToken = null):void {
    CursorManager.removeBusyCursor();                                handleError ❼
    Alert.show("Error occured: " + event.message);
  }
}
}
```

Listing 4.13 shows the rest of the code for the CommentsListPresenter. First you define a method to handle changes to the selected issue ❶, where you set the selected issue from the data passed along in the event. You then set the selected comment to null, and refresh the list of comments to be displayed. When someone clicks on a comment in the List view of the CommentsView, the selectedCommentChanged ❷ method will be called, where you set the selected comment to the comment in the incoming event.

The next method you define handles the user clicking the Remove Comment button ❸. You first tell the CursorManager to set the busy cursor, as you did earlier in the other Presenters. Then you make a call to the CommentModel to remove the selected comment. When the result comes back ❹ from the call to the Comment-Model you remove the busy cursor and tell the Presenter to refresh the list of comments for the view.

To refresh the list of comments you created a functlon called refreshComments ❺, which makes a call to the CommentModel to retrieve a list of comments for a given issue id. When the result comes back from this call ❻, you iterate through the list of ObjectProxy instances that come back from the WebService and use the convenience constructor you defined at the beginning of this chapter to create a list of Comment objects. You then set the dataProvider property of the List component in the view to this list of Comment objects and remove the busy cursor. Last you create a simple error handler ❼ as you did for the two Presenters you created earlier.

4.7.2 Creating a comment model

Other than the method names, the CommentModel will look similar to the IssueModel you created earlier.

Listing 4.14 CommentModel.as

```
package org.foj.model {

import mx.rpc.AsyncToken;
import mx.rpc.IResponder;
import mx.rpc.soap.WebService;

public class CommentModel {

  private var _commentService:WebService;
  public function SoapCommentModel() {                           ❶ Constructor
    _commentService = new WebService();
    _commentService.wsdl =
            "http://localhost:8080/services/CommentService?wsdl";
    if (_commentService.canLoadWSDL()) {
      _commentService.loadWSDL();
    }                                                getCommentsForIssueId ❷
  }
  public function getCommentsForIssueId(issueId:Number,
                                       responder:IResponder):void {
    var token:AsyncToken = _commentService.findCommentsByIssueId(issueId);
    token.addResponder(responder);
  }                                                    ❸ removeComment
  public function removeComment(id:Number,
                               responder:IResponder):void {
    var token:AsyncToken = _commentService.remove(id);
    token.addResponder(responder);
  }
}
}
```

In the constructor ❶ you create a new `WebService` component and set its `wsdl` property to the WSDL for the web service. Then just as you did earlier, you tell the `WebService` to load the WSDL so you can make calls to it. Next you define the two methods needed for the application to function, the `getCommentsForIssueId` ❷ and the `removeComment` ❷ methods.

4.7.3 CommentView

Now that you have the methods implemented in the `CommentModel`, you can update the `CommentsListView` component to enable it to respond to user interaction. Listing 4.15 shows the updated CommentsListView.mxml file. For now, you're going to respond only to clicks in the `List` component containing the comments and the Remove Comment button. In the next section you'll start implementing the modal pop up that will allow you to add and edit comments.

Listing 4.15 CommentsListView.mxml

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

  <fx:Script>
  <![CDATA[

    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.UIEvent;
    import org.foj.presenter.CommentsListPresenter;
    private var _presenter:CommentsListPresenter;

    private function init():void {                         ❺ init
      _presenter = new CommentsListPresenter(this);
    }
                                                           ❸ removeComment
    private function removeComment():void {
      commentsList.selectedIndex = -1;
      var removeCommentEvent:UIEvent =
          new UIEvent(UIEvent.DELETE_COMMENT_BUTTON_PRESSED);
      EventDispatcherFactory.getEventDispatcher()
          .dispatchEvent(removeCommentEvent);          selectedItemChanged ❹
    }

    private function selectedItemChanged(event:Event):void {
      var commentChangedEvent:UIEvent =
          new UIEvent(UIEvent.SELECTED_COMMENT_CHANGED);
      commentChangedEvent.data = commentsList.selectedItem;
      EventDispatcherFactory.getEventDispatcher().dispatchEvent
      ➥ (commentChangedEvent);
    }
  ]]>
  </fx:Script>

  <mx:List id="_commentsList"
          width="100%"
          height="100%"                                ❺ itemClick event
          itemClick="selectedItemChanged(event)"
          labelField="commentText">

  </mx:List>

  <mx:ControlBar>
    <s:Button id="_addCommentButton"
              label="Add New Comment"
              enabled="false"/>
    <s:Button id="_editCommentButton"
              label="Edit Comment"
              enabled="false"/>
    <s:Button id="_deleteCommentButton"
              label="Delete Comment"                ❺ click event
              click="removeComment()"
              enabled="false"/>

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

Again, you use the `creationComplete` event ❺ to call the `init` method ❺ where you bootstrap the Presenter. You then create an event handler named `removeComment` ❺ to handle the user clicking the Delete Comment button ❺. Inside this method you set the `selectedIndex` to -1 so no items are selected in the `List` and dispatch an event notifying the rest of the application that the button has been clicked. Next you define an event handler method `selectedItemChanged` ❺ to handle the user selecting an item in the `List` component ❺.

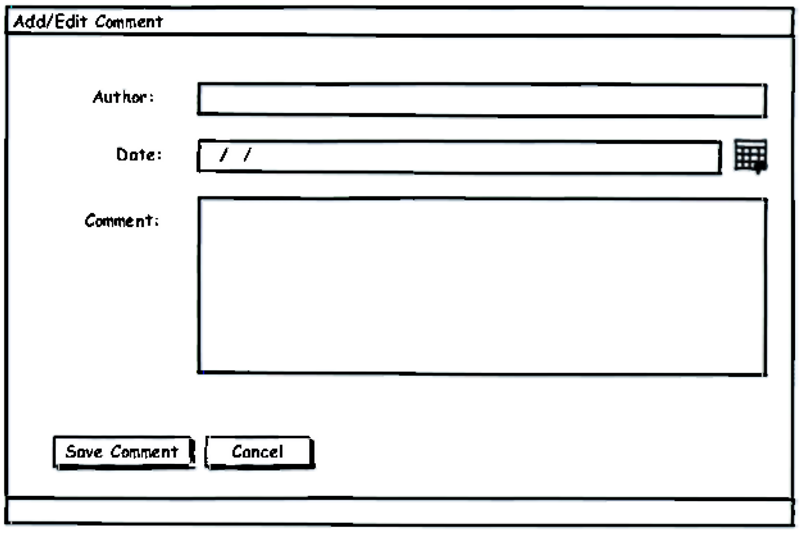


Figure 4.2 Wireframe of pop up for editing comments

4.8 Adding a pop-up form for editing comments

All that's left to complete the sample application is creating a pop up for adding and editing comments. Figure 4.2 shows a wireframe mockup of the pop up. It's a fairly simple pop-up dialog that consists of a text input for the author's name, the date the comment was created, and a text area for entering the comment details.

Now that you see what you want to build, let's get started. The following section illustrates how you build the pop up, starting with updating the Comment Presenter.

4.8.1 Updating the CommentPresenter

First let's add the methods you need to the `CommentPresenter` for the pop-up dialog.

Listing 4.16 Adding a pop up to CommentListPresenter.as

```
private var _popl:EditCommentForm;                               <-1 Pop up component

public function CommentsListPresenter(view:CommentsView = null) {     <-
...
  EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.ADD_NEW_COMMENT_BUTTON_PRESSED,
                          addNewComment);                  Add more
  EventDispatcherFactory.getEventDispatcher()            event listeners 2
        .addEventListener(UIEvent.EDIT_COMMENT_BUTTON_PRESSED,
                          editComment);
  EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.SAVE_COMMENT_BUTTON_PRESSED,
                          saveComment);
  EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.CANCEL_EDIT_COMMENT_BUTTON_PRESSED,
                          cancelEdit);
  EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.COMMENTS_UPDATED,
                          refreshComments);
...
}                                                              ❸ addNewComment
private function addNewComment(event:* = null):void {
    selectedComment = new Comment();
    selectedComment.issue = selectedIssue;
    _popl = PopUpManager.createPopUp((_view as UIComponent).root,
            EditCommentForm, true) as EditCommentForm;
  PopUpManager.centerPopUp(_popl as UIComponent);

}                                                                  ❹ editComment
private function editComment(event:* = null):void {
  _popl = PopUpManager.createPopUp((_view as UIComponent).root,
          EditCommentForm, true) as EditCommentForm;
  PopUpManager.centerPopUp(_popl as UIComponent);
  _popl.author.text = selectedComment.author;
  _popl.commentDate.selectedDate = selectedComment.createdDate;
  _popl.commentText.text = selectedComment.commentText;
}                                                                 ❺ saveComment
private function saveComment(event:* = null):void {
  CursorManager.setBusyCursor();
  selectedComment.author = _popl.author.text;
  selectedComment.createdDate = _popl.commentDate.selectedDate;
  selectedComment.commentText = _popl.commentText.text;

  _commentModel.saveComment(selectedComment,
      new AsyncResponder(saveCommentResult, handleError));
}                                                                     ❻ cancelEdit
private function cancelEdit(event:* = null):void {
  removePopUp();
}

private function saveCommentResult(event:ResultEvent,
                                   token:AsyncToken = null):void {
  CursorManager.removeBusyCursor();
  selectedComment = event.result as Comment;       saveCommentResult handler ❼
    var commentsChangedEvent:UIEvent =
        new UIEvent(UIEvent.COMMENTS_UPDATED);
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(commentsChangedEvent);
    removePopUp();
  }                                                            ❽ removePopUp
  private function removePopUp():void {
    PopUpManager.removePopUp(_popl as UIComponent);
  }
...
```

Change the CommentPresenter to declare a member variable for the pop up itself ❶. Next add event listeners ❷ to respond to user clicks on the buttons on the Comments-View and the pop up you're going to create.

Define an event handler for the Add New Comment ❸ button, set the selected comment to a brand new comment and create the pop up using the `PopupManager`. The call to `createPopUp` takes three parameters: the component that will be the parent of the pop up, pop up class, and whether or not the pop up should be modal. Set the parent for the pop up to be the root application component so that when you call `centerPopUp` on the `PopUpManager` in the next line it will center relative to the whole application instead of to the `CommentView`.

The next event handler, `editComment` ❹ starts out in a manner similar to the `addNewComment` handler. You create a pop up using the `PopUpManager` and center it, then update the fields in the pop up with the values from the currently selected comment. When the user clicks the Save Comment button in the pop up, the `saveComment` ❺ handler is called. You then take all the values from the pop up and update the currently selected comment and make a call to the `CommentModel` to save the comment. When the user clicks the Cancel Changes button on the pop up, the `cancelEdit` ❻ method is called, where you remove the pop up.

When the result comes back from the call to save the comment, the `saveCommentResult` handler ❼ is called. There you update the selected comment with the result from the web service call, then trigger the `List` to update by dispatching an event signaling that the comments have changed and remove the pop up ❻.

4.8.2 Updating the CommentModel

Next you add another method to the `CommentModel` to call the web service and save a comment.

Listing 4.17 CommentModel.as

```
...
public function saveComment(comment:Comment,
                             responder:IResponder):void {
    var token:AsyncToken = _commentService.save(comment);
    token.addResponder(responder);
  }
...
```

Listing 4.17 shows the method you add to the `CommentModel` to save comments. This method is just like every other call we've made to the `WebService`. It delegates the call to the web service and adds a responder to the token so the application can react when the result comes back from the asynchronous call to the web service.

4.8.3 Creating the pop-up component

Now you can create the actual pop up component that you'll use to create new comments as well as edit existing ones. Even though you're instantiating the pop up in ActionScript by making calls to the `PopUpManager`, it's easiest to define this actual component in MXML just as with all the other View components. Listing 4.18 shows the contents of the EditCommentForm.mxml file defining the pop up.

Listing 4.18 EditCommentForm.mxml

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

  <fx:Script>
    <![CDATA[

    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.UIEvent;                                      ❷ saveComment
                                                                         eventHandler
    private function saveComment():void {
      var saveEvent:UIEvent =
          new UIEvent(UIEvent.SAVE_COMMENT_BUTTON_PRESSED);
      EventDispatcherFactory.getEventDispatcher()
          .dispatchEvent(saveEvent);
    }                                                                  ❸ cancelChanges
                                                                         eventHandler
    private function cancelChanges():void {
      var cancelEvent:UIEvent =
          new UIEvent(UIEvent.CANCEL_EDIT_COMMENT_BUTTON_PRESSED);
      EventDispatcherFactory.getEventDispatcher()
          .dispatchEvent(cancelEvent);
    }

    ]]>
    </fx:Script>                                             ❹ Form

  <mx:Form width="100%">
    <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>
    <mx:Button id="_saveButton"                              ❺ ControlBar
               label="Save Comment"
               click="saveComment()"/>
    <mx:Button id="_cancelButton"
               label="Cancel"
               click="cancelChanges()"/>
  </mx:ControlBar>

</mx:TitleWindow>
```

The code for the pop up looks similar to the DetailsView for the issues. The main difference is this component extends the TitleWindow ❶ component instead of the Panel component. Most of the pop up windows you'll create will extend this component. You then define a couple of methods ❷, ❸ to react to the button clicks of the pop up to dispatch UIEvents for the application to listen for. Just as with the DetailView, you use the Form component ❹ for convenient layout of the form fields and labels for the pop up. You then create a ControlBar ❺ to hold the Save Comment and Cancel buttons.

4.8.4 Updating CommentsListView

The only thing left is to update the CommentsListView to dispatch the necessary events for the Add Comment and Edit Comment buttons.

Listing 4.19 Updated CommentListView.mxml

```
...
private function addNewComment():void {                                    ◁┐
  var addNewCommentEvent:UIEvent =                                         ❶ addNewComment
      new UIEvent(UIEvent.ADD_NEW_COMMENT_BUTTON_PRESSED);
  EventDispatcherFactory.getEventDispatcher()
      .dispatchEvent(addNewCommentEvent);
}

private function editComment():void {                                       ◁┐
  var editCommentEvent:UIEvent =                                            ❷ editComment
      new UIEvent(UIEvent.EDIT_COMMENT_BUTTON_PRESSED);
  EventDispatcherFactory.getEventDispatcher()
      .dispatchEvent(editCommentEvent);
}
...
<mx:ControlBar>
    <s:Button id="addCommentButton"
              label="Add New Comment"
              click="addNewComment()"                                 ◁┐
              enabled="false"/>                                         │
    <s:Button id="editCommentButton"                                    │
              label="Edit Comment"                                      │
              click="editComment()"                                   ◁┤
              enabled="false"/>                                         ❸ click handlers
    <s:Button id="deleteCommentButton"                                  │ on the buttons
              label="Delete Comment"                                    │
              click="removeComment()"                                 ◁┘
              enabled="false"/>

  </mx:ControlBar>
```

The first two changes to the CommentsListView are adding methods ❶, ❷ to be called when the Add New Comment and Edit Comment buttons are clicked. These will create and dispatch `UIEvents` like every other view you've defined so far. Then you update the buttons themselves ❸ so that they will call the methods you defined when they are clicked.

4.9 Summary

In this chapter you explored the architectural elements that make up a well-designed, flexible, and maintainable RIA. You learned how to apply the MVP pattern to the code and how following that pattern, you effectively separate the areas of concern into their isolated parts. After you abstracted the calls to external services in the Model, you discovered how easy it would be to potentially replace this portion of the code without affecting any other part of the code.

In the next chapter you'll build upon what you've assembled in this chapter and introduce the Spring BlazeDS Integration framework and BlazeDS remotIng.

Part 2

BlazeDS remoting

In the the first of two chapters in part 2, you'll dive into BlazeDS remoting, learning the basics of BlazeDS and refactoring the example application to take advantage of it. In a nutshell, BlazeDS is a collection of Java components that you can deploy witin your web application for AMF/HTTP communication between Flex and Java. BlazeDS also supports messaging. BlazeDS is basically a subset of the commercial Flex Data Services offering from Adobe, and although you can do many tiings with BlazeDS, certain functions like clustering and advanced binding techniques can only be accomplished witin the full-blown Flex Data Services.

In chapter 6, you'll take the next step in evolving our Flex and Java communications and take advantage of real-time messaging between the two using BlazeDS messaging. The chapter exploits the use of the Flex Messaging API and simple polling to receive updates from the server when changes in the model have occurred.

You will learn how to set up BlazeDS logging, performance benchmarking, and Flex Messaging.

5

BlazeDS remoting and logging

This chapter covers

- Building a BlazeDS Configuration Module
- Flex and Java communication with BlazeDS
- Spring BlazeDS Integration framework
- Logging events and performance statistics

In previous chapters you built a Flex client application using the MVP design pattern and connected it to Java through web services. Now you're going to move from XML/HTTP web services and try remoting.

For Action Message Format communication, or AMF, Flex provides the `RemoteObject` component that uses Adobe's own AMF binary protocol to communicate with the server. This means you'll need a process running on the server side, such as BlazeDS, that understands how to serialize and deserialize the AMF protocol.

In the next section we introduce you to BlazeDS and begin integrating with our sample application. We'll be using the FlexBugs sample application but it would be equally useful to wire up your own slice of functionality to see it work for your own purposes.

5.1 *Introducing BlazeDS*

A Flex client typically runs in a web browser or through the AIR desktop application platform. BlazeDS comes into the technology stack for use when the client needs to communicate with a Java application on the server that is commonly powered by a service-oriented architecture (SOA). Figure 5.1 shows the basic anatomy of a BlazeDS application framework.

In a nutshell, BlazeDS is a collection of Java components that you can deploy with your web application for AMF/HTTP communication between Flex and Java. BlazeDS also supports messaging. BlazeDS is basically a subset of the commercial Flex Data Services offering from Adobe, and although you can do many things with BlazeDS, certain functions like clustering and advanced binding techniques can only be accomplished with the full-blown Flex Data Services.

NOTE In Adobe's words, BlazeDS is the server-based Java remoting and web messaging technology that enables developers to easily connect to back-end distributed data and push data in real-time to Adobe® Flex® and Adobe AIR^{TM} applications for more responsive rich Internet application (RIA) experiences.

You may be asking, "Why would I use BlazeDS instead of web services?" One of the biggest advantages to using the AMF protocol for communication is performance. Adobe's James Ward put together a nice application called BlazeBench (http://www.jamesward.com/wordpress/2007/12/12/blazebench-why-you-want-amf-and-blazeds/), which benchmarks methods of communicating to the server side from a web application, including Flex, and displays the different aspects of the transaction. Figure 5.2 shows a sample output from BlazeBench.

It may be difficult to see in the screenshot, but the total execution time for an operation using Flex AMF3 is by far quicker for round-trip time to the server, and

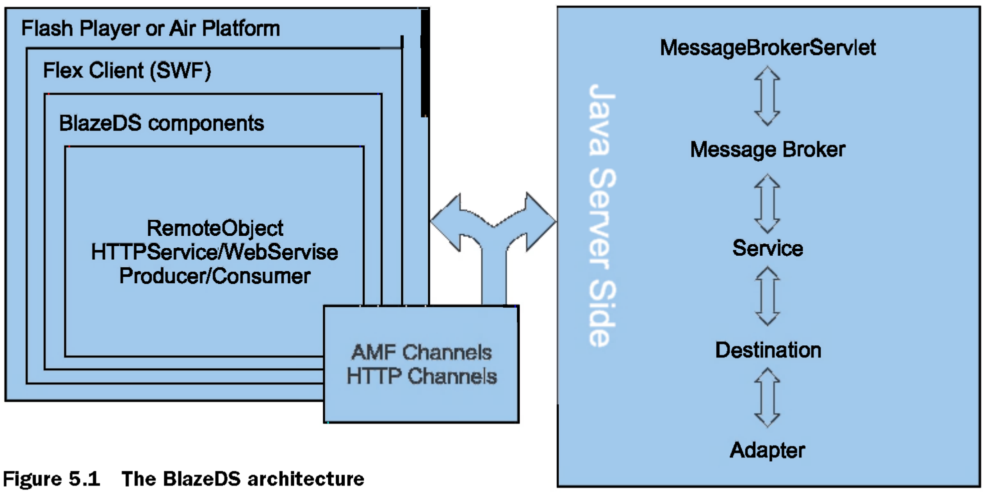


Figure 5.1 The BlazeDS architecture

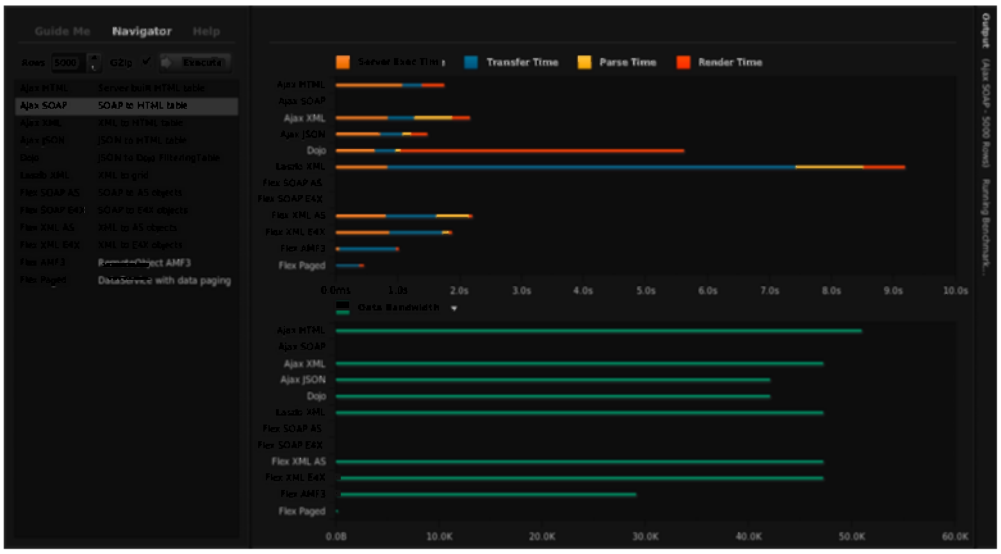

Guide Me
Navigator
Help
Rows 5000
GZip
Execute
Ajax HTML Server built HTML table
Ajax SOAP SOAP to HTML table
Ajax XML XML to HTML table
Ajax JSON JSON to HTML table
Dojo JSON to Dojo FilteringTable
Laszlo XML XML to grid
Flex SOAP AS SOAP to AS objects
Flex SOAP E4X SOAP to E4X objects
Flex XML AS XML to AS objects
Flex XML E4X XML to E4X objects
Flex AMF3 RemoteObject AMF3
Flex Paged DataService with data paging
Server Exec Time
Transfer Time
Parse Time
Render Time
0.0ms 1.0s 2.0s 3.0s 4.0s 5.0s 6.0s 7.0s 8.0s 9.0s 10.0s
Data Bandwidth
0.0B 10.0K 20.0K 30.0K 40.0K 50.0K 60.0K
Output (Ajax SOAP - 5000 Rows) Running Benchmark...

Figure 5.2 Comparing performances of various methods of transferring data from the server side to the client

it has the smallest data bandwidth for the transmission as well as the shortest parse tlme.

5.2 *Getting BlazeDS*

You're in luck if you are using Maven to build your web application. After much persistence from the development community, there are now BlazeDS JARs in the main Maven repository. Add the dependencies shown in listlng 5.1 in your pom.xml to include the BlazeDS libraries in your application and they are downloaded for you.

Listing 5.1 Adding BlazeDS to your pom.xml

```
<dependency>
    <groupId>com.adobe.blazeds</groupId>
       <artifactId>blazeds-core</artifactId>
       <version>3.2.0.3978</version>
   </dependency>
   <dependency>
       <groupId>com.adobe.blazeds</groupId>
       <artifactId>blazeds-common</artifactId>
       <version>3.2.0.3978</version>
   </dependency>
   <dependency>
       <groupId>com.adobe.blazeds</groupId>
       <artifactId>blazeds-proxy</artifactId>
       <version>3.2.0.3978</version>
   </dependency>
```

```
<dependency>
    <groupId>com.adobe.blazeds</groupId>
    <artifactId>blazeds-remoting</artifactId>
    <version>3.2.0.3978</version>
</dependency>
<dependency>
    <groupId>org.springframework.flex</groupId>
    <artifactId>spring-flex</artifactId>
    <version>1.0.0.RELEASE</version>
</dependency>
```

We've also included the Spring-Flex artifact because you're going to use this framework for communicating with server-side destinations. The Spring BlazeDS Integration reduces the complexity in configuring BlazeDS.

For those not using Maven, you have to jump through a couple of hoops to get the proper libraries in your application. Adobe doesn't provide the JAR files directly on the BlazeDS downloads page (http://opensource.adobe.com/wiki/display/blazeds/Release+Builds); it provides only the binary distribution as a WAR file, a turnkey solution which contains a ready-to-use Tomcat instance with sample applications, and a source download. The easiest way to get the JAR is to get the WAR distribution and extract them out of the WEB-INF/lib directory. Thankfully everything you need to add to your web application is there including all the dependencies. Figure 5.3 shows the libraries contained in the WEB-INF/lib directory.

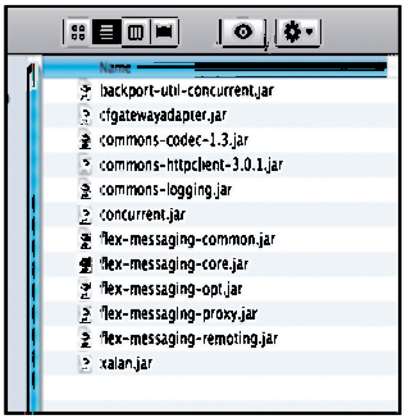

Name
backport-util-concurrent.jar
cfgatewayadapter.jar
commons-codec-1.3.jar
commons-httpclient-3.0.1.jar
commons-logging.jar
concurrent.jar
flex-messaging-common.jar
flex-messaging-core.jar
flex-messaging-opt.jar
flex-messaging-proxy.jar
flex-messaging-remoting.jar
xalan.jar

Figure 5.3 BlazeDS libraries and their dependencies

Now let's get BlazeDS connected up in a more modular way with Maven.

5.3 *Building a BlazeDS configuration Maven module*

To share configuration dependencies between the Java web application and the Flex client you're going to create a BlazeDS configuration module. The directory structure will look like the one in figure 5.4. You can create this module at the same level as the `flex-bugs-web` module and the `flex-bugs-ria` module.

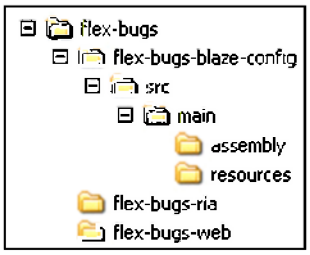

flex-bugs
flex-bugs-blaze-config
src
main
assembly
resources
flex-bugs-ria
flex-bugs-web

Figure 5.4 BlazeDS configuration module project structure

As you can see the only thing needed is the resources directory for the BlazeDS configuration files and the POM descriptor and a Maven assembly. Remember that the project structure can be generated much as was demonstrated in chapters 2 and 3. If you don't generate them with the Maven archetype you must specify in the top-level POM (flex-bugs) a new module, and you must associate the BlazeDS configuration as a dependency for the other projects.

Next modify the module's POM descriptor file.

Listing 5.2 BlazeDS configuration module POM

```
<?xml version="1.0"?>
<project>
  <parent>
    <artifactId>flex-bugs</artifactId>
    <groupId>org.foj</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.foj</groupId>
  <artifactId>flex-bugs-blaze-config</artifactId>      ❶ Coordinates indicate
  <packaging>pom</packaging>                              POM packaging
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>     Using Maven
        <executions>                                    ❷ assembly plugin
          <execution>
            <id>Package BlazeDS configuration</id>
            <goals>
              <goal>single</goal>
            </goals>                                  ❶ Assembly happens
            <phase>package</phase>                       during package phase
            <configuration>
              <descriptors>
                <descriptor>src/main/assembly/resources.xml</descriptor>
              </descriptors>
            </configuration>                             Path to assembly
          </execution>                                     instructions ❶
        </executions>
      </plugin>
    </plugins>
  </build>

</dependencies>

</project>
```

Because you created this module only to share its resources between the client and server modules you can specify the packaging type of `pom` ❶ and specify that you've chosen to use the `maven-assembly-plugin` ❶ to zip up resources ❶ during the package phase ❶. This POM basically instructs Maven to collect into a zip file the resources you need to share.

Listing 5.3 shows the contents of the resources.xml descriptor file found in the src/main/assembly folder of the `flex-bugs-blaze-config` module.

Maven assembly install and deploy

The Maven framework automatically installs and deploys anything built through the context of an assembly. For installation, Maven installs the artifact(s) into your local repository located typically in the .m2 directory, which is usually found in your home directory. Maven deploys artifacts if the distribution management is specified with snapshot or release repositories defined. For deployment, a valid Maven repository, like Sonatype's own *Nexus* repository, is required.

Listing 5.3 resources.xml

```
<assembly>
  <id>resources</id>
  <formats>
    <format>zip</format>                                ❶ Assembly format
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>src/main/resources</directory>
      <outputDirectory></outputDirectory>               ❷ To include in assembly
    </fileSet>
  </fileSets>
</assembly>
```

This assembly instructs Maven to create a zip file ❶ with the files listed in the `fileset` ❷. That is where you'll be adding our two BlazeDS configuration files that need to be shared across modules.

5.3.1 *Configuring BlazeDS*

BlazeDS uses a couple of key configuration files that by convention are put in a flex folder under WEB-INF. You could put them wherever you like, but by placing them under the WEB-INF folder, you make them accessible only to your web application, and you can feel safe knowing any sensitive information, such as passwords in your proxy-config.xml, are safe from prying eyes.

The first file we'll look at is services-config.xml.

Listing 5.4 services-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>

  <services>                                            ❶ Import the
    <service-include file-path="proxy-config.xml"/>       proxy-config.xml

    <default-channels>
      <channel ref="my-amf"/>
    </default-channels>

  </services>
  <channels>                                                  ❷ Channel definition

    <channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
      <endpoint
        url="http://{server.name}:{server.port}/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>          Endpoint url with
    </channel-definition>                                        property names ❸

  </channels>

</services-config>
```

This is how the basic services-config.xml file should look for BlazeDS. There are a couple of sections to note here. First you tell the services-config.xml file to import the proxy-config.xml ❶ and where to find it. Next in the channels definition ❷ you define an AMF channel and point it at a servlet that you'll define in the web.xml later.

It's also possible to use properties for the channel definition that transform into their real values when the application starts. It's common to do this for the endpoint `url` attribute ❸. Using the `{server.name}` and `{server.port}` properties allows the application to be ported without having to hard code different static values each time it needs to be deployed in a different environment. Refer to the BlazeDS documentation at http://opensource.adobe.com/wiki/display/blazeds/Developer+Documentation if you need more information on how to tweak the configuration.

Flash Player Security and RPC

Whenever you attempt to receive data from a remote service, Flash Player checks the domain name of the remote service location and compares it to the domain name of your Flex application's location. If the two do not match, the data cannot be loaded from the remote service. The easiest way to remedy this situation is to host your Flex application on the same domain as your remote services. If this is not possible you have a few options. You can create a crossdomain.xml file to allow Flash applications outside of your domain to use the remote services. You can create a servlet in your web application to act as a proxy for the Flash application, or you can configure BlazeDS to do this by using the proxy-config.xml.

The last configuration file we'll examine for now is proxy-config.xml.

Listing 5.5 proxy-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<service id="proxy-service"
         class="flex.messaging.services.HTTPProxyService">

  <properties>
    <connection-manager>
      <max-total-connections>100</max-total-connections>
      <default-max-connections-per-host>2</default-max-connections-per-host>
    </connection-manager>
    <allow-lax-ssl>true</allow-lax-ssl>
  </properties>

  <adapters>
    <adapter-definition id="http-proxy"
       class="flex.messaging.services.http.HTTPProxyAdapter"
       default="true"/>
    <adapter-definition id="soap-proxy"
       class="flex.messaging.services.http.SOAPProxyAdapter"/>
  </adapters>

  <default-channels>
    <channel ref="my-amf"/>
  </default-channels>

  <destination id="DefaultHTTP">
  </destination>

</service>
```

In the proxy-config.xml file you can configure any server-side proxies you need to leverage, if, for instance, you're communicating with web services outside your domain and have no control over placing a crossdomain.xml file on the server.

With BlazeDS configured and ready to go, you can now get on with the tasks of modifying some of your services to communicate via AMF.

Where's the remoting-config.xml?

Great question! The BlazeDS examples in this book take advantage of a brand new Spring BlazeDS Integration framework brought to us by SpringSource. Spring BlazeDS Integration allows you to set up remoting destinations by annotating a Java class and specific methods that need to be exposed to Flex through remoting. You can still choose to hand-wire the XML instead of using the annotations but we recommend the use of the Spring BlazeDS Integration annotations and will demonstrate them in this chapter.

Visit the BlazeDS developer documentation if your project requires the remoting-config.xml at http://opensource.adobe.com/wiki/display/blazeds/Developer+Documentation

In the next sections we're going to start off simple by showing you how to connect to a POJO on the server side and get more complicated and leverage the powerful Spring Framework to communicate with a fully injected Spring bean.

ADD THE BLAZEDS CONFIG MODULE TO THE TOP-LEVEL POM

To configure the top-level POM, open and edit the pom.xml file in the flex-bugs directory and add the `flex-bugs-blaze-config` module.

Listing 5.6 Adding BlazeDS config module to the top-level pom.xml

```
<modules>
    <module>flex-bugs-blaze-config</module>
    <module>flex-bugs-ria</module>
    <module>flex-bugs-web</module>
</modules>
```

Next associate the dependency with the `flex-bugs-web` module and `flex-bugs-ria` module.

UPDATE THE FLEX AND JAVA WEB MODULES

Now you can add the configuration module to the other projects as a dependency by editing their respective POM files for the flex-bugs-web/pom.xml.

Listing 5.7 Adding the BlazeDS config dependency to the Java web module

```
<dependencies>
    <dependency>                                                BlazeDS configuration
        <groupId>org.foj</groupId>                              module dependency
        <artifactId>flex-bugs-blaze-config</artifactId>     ❶   declaration
        <version>1.0-SNAPSHOT</version>
        <classifier>resources</classifier>                  ❷   Use resources classifier
        <scope>provided</scope>                                 and provided scope
        <type>zip</type>                        Type element is type
    </dependency>                               of artifact produced
    . . .                                   ❸   by assembly
</ dependencies>
```

As you can see, you add the new dependency to the `dependencies` element ❶ in each module's `pom` and you are done. Because the scope is provided, it will not add the resources ❷ to the WAR. You had to specify the type ❸ because it's a zip and not the default type. You'll be getting them from the Flex module by editing the POM descriptor in the `flex-bugs-ria` module.

Listing 5.8 Adding the BlazeDS config dependency to the Flex module

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>              Maven-dependency-
  <executions>                                              ❶   plugin definition
    <execution>
      <id>unpack-config</id>                            ❷   Invoke unpack-
      <goals>                                               dependencies goal
        <goal>unpack-dependencies</goal>
      </goals>                                              ❸   Execute goal during
      <phase>generate-resources</phase>                         generate-resources phase
      <configuration>
        <outputDirectory>${project.build.directory}/        ❹   Output directory
➥ generated-resources</outputDirectory>                         to unpack zip file
        <includeArtifactIds>flex-bugs-blaze-config</includeArtifactIds>
        <includeGroupIds>${project.groupId}</includeGroupIds>
        <excludeTransitive>true</excludeTransitive>
      </configuration>                                          Include flex-bugs-
    </execution>                                                blaze-config artifact ❺
  </executions>
</plugin>
```

There are many ways to share configurations, and as you see, you can use the maven-dependency-plugin ❶. You execute the unpack-dependencies goal ❷ during the generate-resources phase ❸ and must choose an output directory ❹ in order to unzip the file where you want. You may have noticed that you used the built-in maven property ${project.build.directory}, which equates to the location of the target directory. Now you must indicate what to unpack. This is where you specify the flex-bugs-blaze-config artifact ❺.

After the configuration is completed for the shared configuration approach, you can start integrating Flex with Java through the use of BlazeDS AMF remoting!

5.4 *Exposing Java services to Flex remoting*

In this section you'll demonstrate how to configure and expose Java services to Flex through the use of BlazeDS and Spring annotations. First you'll modify the web module configuration, then modify the Java code. For demonstration purposes you'll use the FlexBugs sample application.

5.4.1 *Web module configuration updates*

First you'll modify the flex-bugs-web module's configuration to enable it for remoting. You'll start with the applicationContext.xml file located in the /flex-bugs-web/src/main/webapp/WEB-INF directory.

Listing 5.9 ApplicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>                Update schema namespace ❶
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/spring-security-
       ➥2.0.4.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-
       ➥2.5.xsd"
    default-lazy-init="true">                         ❷ Enable annotations

    <context:annotation-config />                    ❸ Use component
    <context:component-scan base-package="org.foj" />    scanner

  <!-- Add new DAOs here -->
  <bean id="issueDao" class="org.appfuse.dao.hibernate.GenericDaoHibernate">
    <constructor-arg value="org.foj.model.Issue"/>
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <bean id="commentDao" class="org.foj.dao.impl.CommentDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
  <!-- Add new Managers here -->
  <bean id="issueService" class="org.foj.service.impl.IssueManagerImpl">
    <constructor-arg ref="issueDao"/>
    <constructor-arg ref="commentService"/>
  </bean>

  <bean id="commentService" class="org.foj.service.impl.CommentManagerImpl">
    <constructor-arg ref="commentDao"/>
  </bean>

  <!-- Add new Actions here -->
</beans>
```

To take advantage of the Spring BlazeDS Integration you must add the appropriate schema changes and namespace elements ❶. Because you'll be using annotations to expose your services to BlazeDS remoting destinations, you must enable annotations ❷ and use the component scanner ❸ to help Spring find the annotations you're going to define.

Next, by convention, you must create a flex-spring-servlet.xml configuration file, as seen in listing 5.10, because the framework depends on it.

NOTE The naming of the flex-spring-servlet.xml file is important because the framework scans for it by convention, and the application will fail to start up properly without it.

Later, you'll use the flex-spring-servlet.xml file to help configure the messaging service.

Listing 5.10 flex-spring-servlet.xml plumbing

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:flex="http://www.springframework.org/schema/flex"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">

</beans>
```

Last, you need to add a new servlet definition to the web.xml so that your Flex application can communicate with the server-side application. Listing 5.11 shows the changes that need to be made to the web.xml of our web application module.

Listing 5.11 web.xml

```
<servlet>
    <servlet-name>flex-spring</servlet-name>                  <-❶ Servlet name
    <servlet-class>
           org.springframework.web.servlet.DispatcherServlet  <-┐
    </servlet-class>                                           ❷ Servlet class
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>flex-spring</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>        <-- Servlet mapping
</servlet-mapping>                                      3 definition
```

This is a typical Java web application servlet mapping where you define a servlet name ❶ and a servlet class ❷ to dispatch actions to whenever a URL pattern ❸ match is found.

Now let's begin working with code by modifying the server-side classes.

5.4.2 *Expose AMF remoting destinations*

Connecting to POJOs on the server side is trivial after you've got BlazeDS configured properly. To illustrate connecting to a POJO from Flex, we're going to use the Flex-Bugs application by exposing the `IssueManager` interface to Flex remoting using the Spring BlazeDS Integration framework.

UPDATES TO ISSUE CODE

This is the code for the `IssueManagerImpl` that we'll be connecting to.

Listing 5.12 IssueManager.java

```
package org.foj.service;

import org.foj.model.Issue;
import org.springframework.flex.remoting.RemotingDestination;      <-
import org.springframework.flex.remoting.RemotingInclude;

import javax.jws.WebService;                              Import Spring BlazeDS
                                                          Integration classes 1
@WebService
@RemotingDestination(channels = {"my-amf"})          <-- RemotingDestination exposes
public interface IssueManager {                       2 issueService to Flex remoting

  @RemotingInclude                                   <-- RemoteInclude
  java.util.List<Issue> getAll();                     3 annotation

  @RemotingInclude
  Issue get(Long id);

  @RemotingInclude
  Issue save(Issue issue);

  @RemotingInclude
  void remove(Long id);

}
```

As you can see, not much had to change to expose the `IssueService` methods to Flex. The first step was to import the Spring BlazeDS Integration classes needed ❶. The Spring classes help by supplying the `@RemotingDestination` and `@Remoting-Include` annotations.

Next, you added the `@RemotingDestination` "my-amf" ❷. The `@Remoting-Destination` must match the one defined in the services-config.xml BlazeDS configuration file. This annotation exposes this class as a destination for any Java class that implements the `IssueManager` interface. This means that the `IssueManagerImpl` Java class, the class that implements the `IssueManager`, will be exposed as an AMF remote service and not just XML/HTTP.

The Spring BlazeDS Integration framework removed the need for having to define this metadata through the remoting-config.xml configuration file. You don't use a remoting-config.xml file with the BlazeDS Integration framework.

You can now annotate each method that you need to expose with the `@RemotingInclude` ❸ annotation and you are done! The `@RemotingInclude` annotation exposes the method over the `@RemotingDestination`.

If needed, the Spring BlazeDS Integration framework's remoting package also provides a `@RemotingExclude` annotation for intentionally excluding methods. Now let's move on to making changes to the Flex client to take advantage of the exposed AMF service.

UPDATES TO COMMENTS CODE

Now that you have everytining in place for the issue code, you'll need to make the same types of changes to the `CommentManager` interface. The following listing shows the `CommentManager.java` class changes.

Listing 5.13 CommentManager.java

```
package org.foj.service;

import org.foj.model.Comment;
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingInclude;

import javax.jws.WebService;
import java.util.List;

@WebService
@RemotingDestination(channels = {"my-amf"})
public interface CommentManager {

  @RemotingInclude
  List<Comment> findCommentsByIssueId(Long issueId);

  @RemotingInclude
  void deleteAllCommentsForIssueId(Long issueId);

  @RemotingInclude
  Comment get(Long id);

  @RemotingInclude
  Comment save(Comment comment);

  @RemotingInclude
  void remove(Long id);

}
```

As you can see you didn't need to change any of the application code to expose the `CommentManager` to remoting. All you had to do was add the necessary imports, include the `@RemotingDestination`, and add the appropriate `@RemotingInclude` annotations for the methods you need exposed.

Now you have a Java server side that is set up for BlazeDS remoting. You tackled much of the Java configuration through Spring's robust Flex BlazeDS Integration framework and its extremely useful annotations.

5.5 Connecting to Java with BlazeDS

With the Java remoting services ready to go you can begin the work on the Flex client to connect to the exposed service objects and methods. Let's start with the `Issue` object.

Listing 5.14 Issue.as

```
package org.foj.dto {

[RemoteClass(alias="org.foj.model.Issue")]                    ❶ RemoteClass annotation
public class Issue {
...
```

That wasn't complicated. You only had to associate this object with the server side equivalent by using the `RemoteClass` annotation ❶ which you need because it allows you to map domain objects between Flex and Java by specifying the Java package and class name.

TIP Without correctly specifying the `RemoteClass`, any attempts at persisting `Issue` objects will fail because BlazeDS will not know what server-side object the Flex object is bound to.

Next you need to modify the `IssueModel` class to call the POJO.

Listing 5.15 IssueModel.as

```
package org.foj.model {

import mx.rpc.AsyncToken;
import mx.rpc.IResponder;
import mx.rpc.events.FaultEvent;
import mx.rpc.events.ResultEvent;
import mx.rpc.remoting.RemoteObject;
import org.foj.dto.Issue;

public class IssueModel {

  private var _issueService:RemoteObject;                     ❶ Create RemoteObject

  public function IssueModel() {
    _issueService = new RemoteObject();
    _issueService.destination = "issueService";              ❷ Set destination
  }

  public function getIssues(responder:IResponder):void {
    var asyncToken:AsyncToken = _issueService.getAll();
    asyncToken.addResponder(responder);
  }

...

}
```

Again a few things have to change, and yet only minor ones are required to take advantage of the performance increases of BlazeDS remoting. In the `IssueModel` you are now using the Flex `RemoteObject` ❶ for communication to the `issueService` instead of the `WebService` object. When you construct a new `IssueModel` you set the `RemoteObject` destination ❷ as the `issueService` you defined earlier in our applicationConfig.xml. That's it. Really!

NOTE Names of the `RemoteObject` instance variables map to the bean name in the Spring applicationContext.xml file. Be sure these names are correct!

With all of that out of the way you should be able to run your application and see that it's now populating the `DataGrid` with data from our `issueService` through BlazeDS rather than the web services you created in chapter 4.

5.6 Logging

Since the inception of programming, developers have always found ways to enable applications to communicate what they are doing. This comes in all different forms, from logging basic information to fatal exceptions during the application runtime. With the FlexBugs sample application, you need to log the Java server-side events and the BlazeDS-specific events, and gather performance statistics.

Building the FlexBugs application with AppFuse simplified this task by configuring the popular log4j framework for logging Java application messages. Now you must inform the log4j framework that you want to log even more for BlazeDS.

5.6.1 BlazeDS logging

Adding BlazeDS into the mix was beneficial. But adding any library or framework to benefit the application presents new challenges; you must collect information from the library to better understand what is happening when you use it.

Thankfully, SpringSource recognized this with the Spring BlazeDS Integration framework and added a new logging object to its core package. The `CommonsLogging-Target` class allows for automatic logging of events from BlazeDS into the application's existing logging configuration.

CONFIGURE LOG4J FOR BLAZEDS FILE LOGGING

You have to configure an application to print messages to the console or even a log file in a couple of places. The first place, as seen in listing 5.16, is the log4j.xml configuration file found in the `flex-bugs-web` module, which can be found in the flex-bugs-web/src/main/resources directory.

Listing 5.16 log4j.xml

```
<appender name="FILE" class="org.apache.log4j.RollingFileAppender">     ❶ log4j rolling file appender
  <param name="File" value="flexbugs.log"/>                             ❷ Log file name
  <layout class="org.apache.log4j.PatternLayout">                       ❸ Log file layout definition
```

```
    <param name="ConversionPattern"
           value="%d [flex-bugs-web] %p [%t] %c{1}.%M(%L) | %m%n"/>     <─┐
  </layout>                                                  ConversionPattern │
</appender>                                                   configuration ❹

<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">      <─┐
  <layout class="org.apache.log4j.PatternLayout">             ConsoleAppender │
    <param name="ConversionPattern"                             configuation ❺
           value="%d [flex-bugs-web] %p [%t] %c{1}.%M(%L) | %m%n"/>
  </layout>
</appender>                                          ❻ blazeds logger
                                                          definition
<logger name="flexbugs">                          <─┘              ❼ Logging level
  <level value="DEBUG" />                                      <─┘
  <appender-ref ref="FILE" />                                  <─┐
    <appender-ref ref="CONSOLE" />                 <─┐            ❽ File appender
</logger>                                          ❾ Console appender
```

As seen in listing 5.16 the log4j configuration can be trivial and yet extremely robust. Log4j itself allows for outputting to both the console and to a file. In the example you specified a `RollingFileAppender` ❸ so you can keep log messages and continue to add to that file as you use the application.

You must specify the log file name as flexbugs.log ❸. Log4j requires the file parameter and doesn't provide a default. It's also possible to specify a path to the file in this parameter. It will place the flexbugs.log file at the root of the flex-bugs-web directory.

Log4j also enables various layout styles. By choosing to use the `PatternLayout` ❸ you can specify what you want your messages to look like and what information should be in the log message. You do this by defining the `ConversionPattern` parameter ❸. There are other patterns to choose from but we chose this one to give us additional flexibility. The first thing to notice about the log message pattern is that it will prefix each message with `[flex-bugs-web]`. It then assigns other parameters for printing the message. For more on log4j configuration go tohttp://www.scribd.com/doc/7679107/ Log4j-Quick-Reference.

You also include an appender for the console ❸. Console appending is especially helpful during software development. Writing the logs to a file is valuable not just for development but also for production environments.

Finally, you need to create the logger itself. Here you created a logger named `blazeds` ❸. You configure the logger to output at the `DEBUG` level ❸ and append the messages to both our `RollingFileAppender` and the `ConsoleAppender` by their respective reference names `FILE` ❸ and `CONSOLE` ❸. Log4j is ready to create a log file for your application.

CONFIGURE BLAZEDS TO OUTPUT TO LOG4J

Setting up the logging of BlazeDS is a two-step process. Now that you have a logger available specifically for BlazeDS you can configure BlazeDS to work with log4j. Listing 5.17 shows the necessary changes to the services-config.xml file, located in the directory.

Listing 5.17 services-config.xml

```
   ...
<logging>
  <target class="org.springframework.flex.core.CommonsLoggingTarget"     ❶ Logger class
          level="All">                                                  ❷ Logging level
    <properties>
      <categoryPrefix>flexbugs</categoryPrefix>                        ❸ Category prefix
    </properties>
  </target>
</logging>
...
```

The BlazeDS configuration is simple. You specify the Spring framework's `CommonsLoggingTarget` class ❶, the logging level you desire ❷, and the `categoryPrefix` which points to the log4j.xml logger named `flexbugs` ❸. The `CommonsLoggingTarget` can be configured to output more—or less—information from BlazeDS events through its properties and logging categories by using filters.

BlazeDS/LCDS logging categories

Several categories are available for helping to filter out specific logging messages. Logging can be performance-intensive so it's wise to log only what you absolutely need to provide the best support for the application. For more on the categories available or to know what you can filter out of the logs, see the API documentation for the `CommonsLoggingTarget` at http://static.springsource.org/spring-flex/docs/1.0.x/javadoc-api/org/springframework/flex/core/CommonsLoggingTarget.html.

Now that logging has been configured for BlazeDS it's time to see the results.

VIEW THE FLEXBUGS LOG FILE

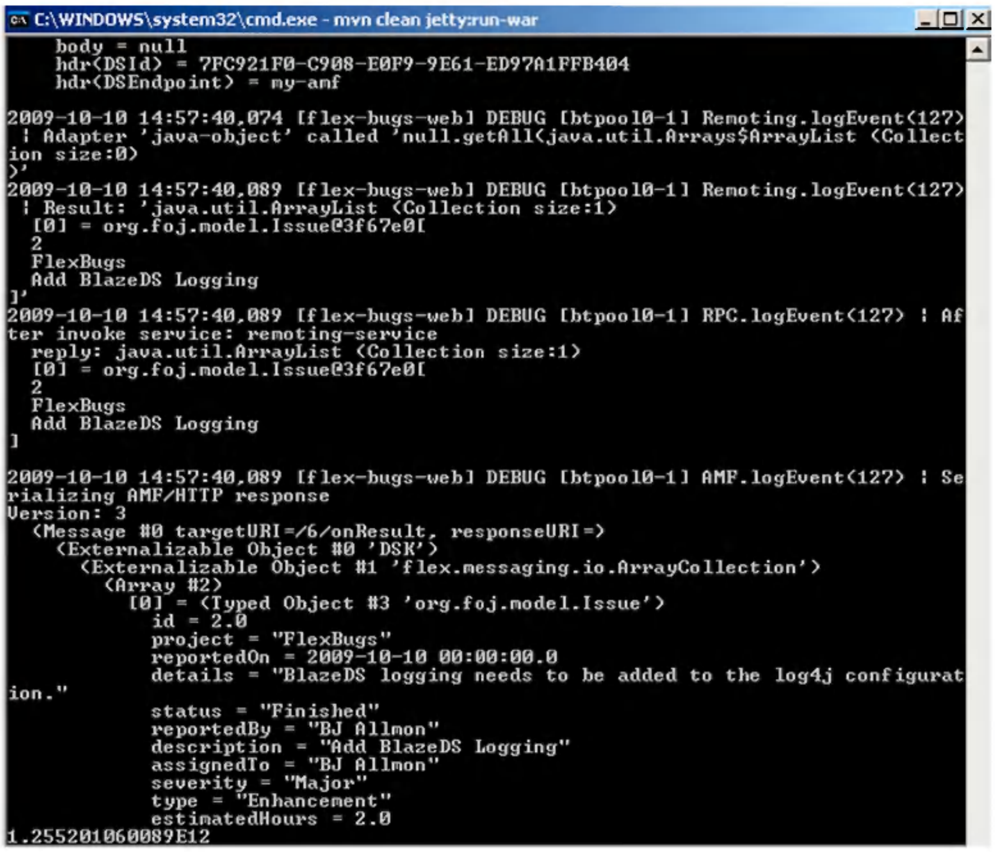
If you fire up your sample application you'll find a log file available. If you're using the FlexBugs samples and run the command `mvn clean jetty:run-war` on the `flex-bugs-web` module, after performing an `mvn clean install` on the `flex-bugs-config` module, you'll find the log file in the flex-bugs-web directory.

Figure 5.5 shows the logging messages going to the console.

Toward the bottom of figure 5.5 you'll find an AMF log event showing an issue completed for adding the BlazeDS logging to the log4j configuration. In that event you can see that the issue was serialized through AMF/HTTP with all the issue details. If you open the log file you'll find the same results and logging of the BlazeDS events.

5.6.2 *Built-in BlazeDS benchmarking*

One nice feature of BlazeDS is its built-in ability to acquire performance results on its processing of messages. In this example you'll configure the BlazeDS services-config.xml to retrieve the message times and size. These measurements are great for development but shouldn't be enabled for production because they cause performance degradation. This may seem counterproductive but it's not. It's good to assure



```
C:\WINDOWS\system32\cmd.exe - mvn clean jetty:run-war
    body = null
    hdr(DSId) = 7FC921F0-C908-E0F9-9E61-ED97A1FFB404
    hdr(DSEndpoint) = my-amf

2009-10-10 14:57:40,074 [flex-bugs-web] DEBUG [btpool0-1] Remoting.logEvent(127)
 | Adapter 'java-object' called 'null.getAll(java.util.Arrays$ArrayList (Collect
ion size:0)
)'
2009-10-10 14:57:40,089 [flex-bugs-web] DEBUG [btpool0-1] Remoting.logEvent(127)
 | Result: 'java.util.ArrayList (Collection size:1)
  [0] = org.foj.model.Issue@3f67e0[
  2
  FlexBugs
  Add BlazeDS Logging
]'
2009-10-10 14:57:40,089 [flex-bugs-web] DEBUG [btpool0-1] RPC.logEvent(127) | Af
ter invoke service: remoting-service
  reply: java.util.ArrayList (Collection size:1)
  [0] = org.foj.model.Issue@3f67e0[
  2
  FlexBugs
  Add BlazeDS Logging
]

2009-10-10 14:57:40,089 [flex-bugs-web] DEBUG [btpool0-1] AMF.logEvent(127) | Se
rializing AMF/HTTP response
Version: 3
  (Message #0 targetURI=/6/onResult, responseURI=)
    (Externalizable Object #0 'DSK')
      (Externalizable Object #1 'flex.messaging.io.ArrayCollection')
        (Array #2)
          [0] = (Typed Object #3 'org.foj.model.Issue')
            id = 2.0
            project = "FlexBugs"
            reportedOn = 2009-10-10 00:00:00.0
            details = "BlazeDS logging needs to be added to the log4j configurat
ion."
            status = "Finished"
            reportedBy = "BJ Allmon"
            description = "Add BlazeDS Logging"
            assignedTo = "BJ Allmon"
            severity = "Major"
            type = "Enhancement"
            estimatedHours = 2.0
1.255201060089E12
```

Figure 5.5 Console log of BlazeDS events

that you have the best performance possible for your production environments at all times to keep the users delighted.

To enable BlazeDS performance collection, open up the services-config.xml for editing. Listing 5.18 shows the changes to the `my-amf` configuration.

Logging affects performance

A developer must choose wisely the messages that are logged in a production environment. You shouldn't keep log messages in an application that were only troubleshooting messages or sloppy in loop count logging that spits out a bunch of unimportant things for every element in a collection of objects. To put it simply, logging affects performance, especially when writing out to file IO. Make sure that debugging is not enabled in production unless it's absolutely necessary. Also be sure to only log INFO type messages when they are helpful or required.

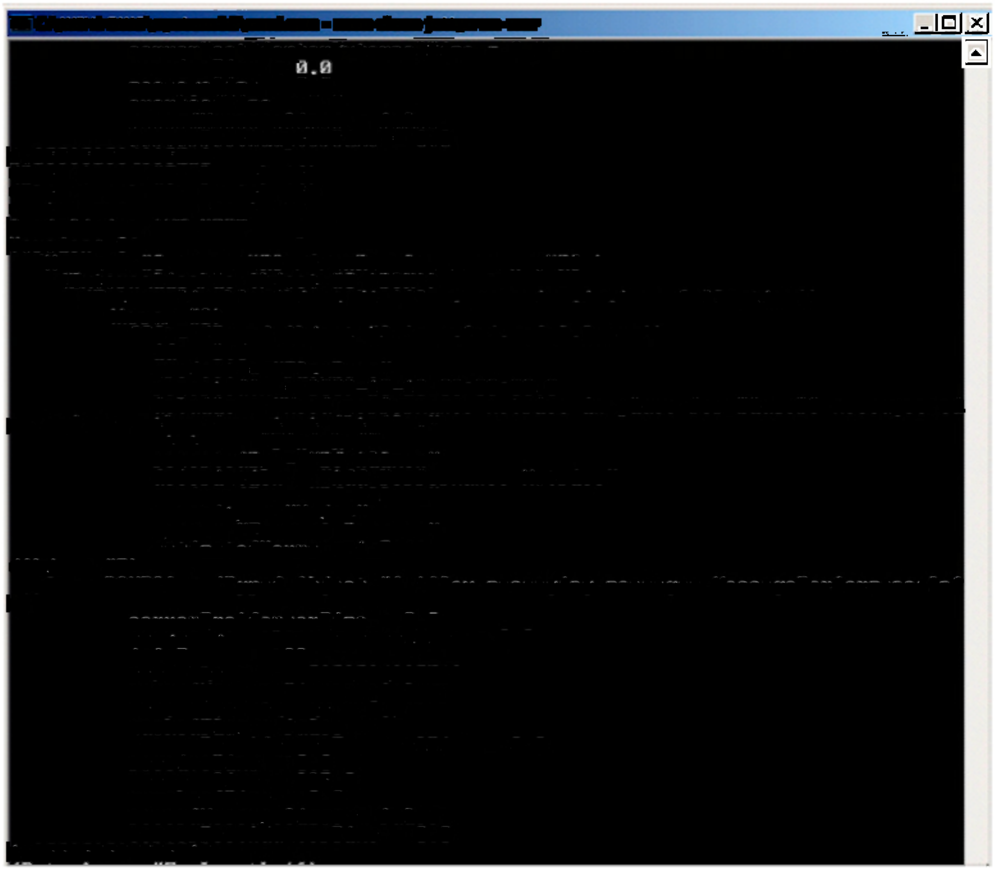
Listing 5.18 services-config.xml

```
  <channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/
    ➥{context.root}/messagebroker/amf"
            class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
      <record-message-times>true</record-message-times>      ❶
   <record-message-sizes>true</record-message-sizes>         ❷
 </properties>
  </channel-definition>
```

❶ Message times configuration

❷ Message sizes configuration

In listing 5.18 you enabled the performance tracking properties for recording botn message times and sizes. It's a simple Boolean tnat can be set to false for production environment configuration. Adding these properties allows for the automatic logging of this information. This is a great way to gain insight into the objects you're sending back and fortn between the client and server. If you were to create a new issue in Flex-Bugs you would see something like the display in figure 5.6.



```
                    0.0
```

Figure 5.6 BlazeDS `MessagePerfomanceInfo` in action

As seen in figure 5.6 you now have an additional log message directly under the insertion of a new issue for the project FlexBugs. The BlazeDS `MessagePerformanceUtils` class shows the `sendTime` and the `messageSize` that you defined in the services-config.xml. You can see that the `sendTime` is equal to 1.25ms and a message size of 835.0 bytes.

In general, the `flex.messages.messaging.MessagePerformanceUtils` class provides metrics that describe the size and timing of a message sent from a client to the server and its server response message. Performance information can be captured when the `record-message-times` and `record-message-sizes` are configured to `TRUE`. From there it's possible to set up more performance-gathering metrics in the client. Configuring the client would mean creating a response acknowledgment or message handler.

NOTE The BlazeDS development guide instructs developers to not activate more than one performance measurement at a time because of the extra performance hit that each measurement adds to the application runtime. Use them wisely. If you truly have performance issues you may consider other means for performance testing like an open source tool or commercial alternative. Using a separate tool for performance testing would allow the BlazeDS configuration to stay true to its production environment configuration during the performance testing.

For more on how to configure message performance logging through a message event handler refer to the BlazeDS development guide at http://livedocs.adobe.com/blazeds/1/blazeds_devguide/help.html?content=mpi_3.html.

5.7 Summary

In this chapter you refactored out the XML/HTTP web service implementation and replaced it with BlazeDS AMF remoting. The web service implementations had the least impact on the server side but required more work to be done on the client side, because we're not receiving first class objects in the responses. When you switched to using BlazeDS to communicate with the server side you gained in performance without increasing the complexity of the application.

Using the Spring BlazeDS Integration framework reduced the amount of configuration typically required to set up remoting destinations. It also gave you a handful of useful annotations and other features that reduced the complexity of the application even more. As you'll continue to see throughout the remainder of the book, the Spring BlazeDS Integration simplifies the code and makes building robust Flex and Java applications much easier.

You also configured logging to be sure that you can capture those important application activities in a log file. Setting up the logging for BlazeDS events was a simple task using both log4j and the Spring BlazeDS Integration framework logger class.

BlazeDS also enables the gathering of performance measurements on a channel service. With that you were able to pump performance stats on message size and time to your log file.

In the next chapter you'll take the next step in evolving our Flex and Java communications and take advantage of real-time messaging between the two using BlazeDS messaging.

6 Flex messaging

This chapter covers

- Setting up BlazeDS for messaging
- Using the Flex Messaging API
- Creating a polling channel

The Flex Messaging API, bundled with BlazeDS, provides asynchronous messaging, which you can use to create a better user experience by enabling your application to refresh itself in real time whenever anyone using the application makes any changes. The BlazeDS `MessageService` allows bidirectional communication between Flex clients and the server side.

In general, you want to use messaging to notify the client of changes. This will fire off an event to refresh the FlexBugs issues list. Figure 6.1 demonstrates this use of messaging.

This chapter will exploit the use of the Flex Messaging API and simple polling to receive updates from the server when changes in the model have occurred. Changes will cause an event to be dispatched that will refresh the master view and ultimately the list of issues.

The details of messaging operations depend on your needs and the style of underlying messaging architecture you've chosen; for example, client-to-client,

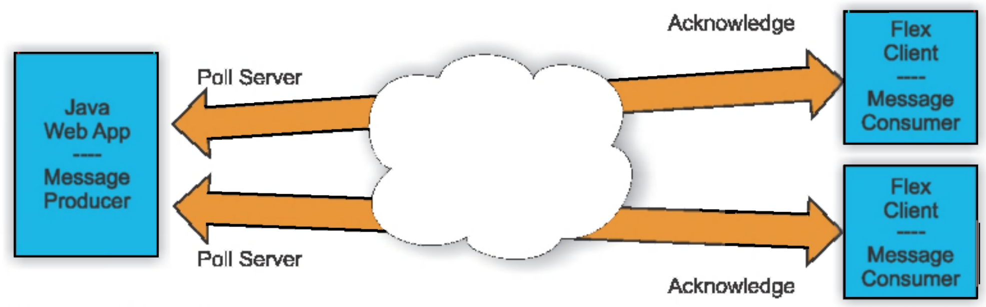


Figure 6.1 Simple polling

JMS, Flex to POJO, or JavaBean messaging. On top of that, it's possible to configure the client to perform simple or long polling or even streaming. All these scenarios for messaging are useful and have accompanying benefits and consequences. It's always best to start with the simple approach before moving to the complex.

6.1 Setting up BlazeDS for messaging

Little configuratIon is needed to set up a simple messaging architecture that allows a Flex client to subscribe to a server-side component. It's also good to use a messaging API that is agnostlc to the underlying messaging architecture. This makes it easy to start with a simple server side, say with POJOs, and expand to something like JMS if necessary.

6.1.1 Modifying the services-config.xml

You first modify the services-config.xml file. If working in the FlexBugs sample application, it is found in the `blaze-config` module. The services-config.xml file is located in the src/main/resources directory.

You want to prepare the server side with a new channel definition for polling as seen in listing 6.1. For brevity we've excluded elements of the file that have already been discussed and are not going to change.

Listing 6.1 Adding the polling channel-definition to services-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
...
  <channels>
...
    <channel-definition id="my-polling-amf"
                        class="mx.messaging.channels.AMFChannel">
      <endpoint url="http://{server.name}:{server.port}/{context.root}/
      ➥messagebroker/amfpolling"
                class="flex.messaging.endpoints.AMFEndpoint"/>
      <properties>
        <polling-enabled>true</polling-enabled>
```

❶ Channel-definition declaration

❷ Endpoint url

❸ Polling is enabled

```
        <polling-interval-seconds>4</polling-interval-seconds>          ◁┐
      </properties>                                                      │
    </channel-definition>                                Polling interval ❹

  </channels>
...
</services-config>
```

All you need to do to set up the channel definition is specify an `id` ❶ and `endpoint` ❷, and enable polling ❸. The channel definition for the `my-polling-amf` channel ❶ will be used by Flex when contacting the server.

The endpoint element ❷ provides a URL that must be unique from other endpoints, and the endpoint class for the server. The endpoint is used by the channel service to do its business in regards to client-side and server-side communication.

Properties defining the channel are nested in a properties element. The channel definition contains behaviors that allow it to be configured in a variety of ways. In our example, you have simple polling enabled ❸ and the polling interval ❹ set to poll at every 4 seconds. Simple polling is generally less efficient than long polling because it continues to ping the server at each specified interval and receives acknowledgments even when there are no changes, and the acknowledgments are empty. Long polling allows the client to ping the server; the server keeps the request and returns an acknowledgment when there is a message.

Polling performance

When selecting a polling mechanism, consider which solution would require the least amount of overhead. Both simple and long polling can be server-side intensive if there are numerous messages being passed back and forth and an abundance of users. Even though long polling is generally more efficient than simple, it's possible that an application with many users with frequent changes could cause more client and server friction and be less responsive than controlling polling with the simple approach. If long polling is an option, streaming should be a consideration as well. Streaming is similar but keeps a connection open instead of opening and closing one between each transmission.

The changes to the services-config.xml were the only changes necessary for the configuration module to add a polling channel. Now let's move on to the webapp module changes.

6.1.2 Updating the webapp server-side module

Now that you've tackled the channel definition in the services-config.xml file you can move on to modifying the webapp. We'll start with the flex-spring-servlet.xml and applicationContext.xml changes, then move on to changes in the Java code.

MODIFY THE FLEX-SPRING-SERVLET.XML

You start by editing the flex-spring-servlet.xml found in the src/main/webapp/WEB-INF directory of the `flex-bugs-web` module. Here you will be adding a new Spring-managed `MessageBroker`, `MessageService`, and `MessageDestination` for your new polling channel.

Listing 6.2 Adding `MessageBroker`, `MessageService`, and `MessageDestination`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:flex="http://www.springframework.org/schema/flex"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
     http://www.springframework.org/schema/flex
     http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">

  <flex:message-broker>                                          
    <flex:remoting-service default-channels="my-amf"/>    ❶ Added message-broker
      <flex:message-service default-channels="my-polling-amf" />
    <flex:secured/>
  </flex:message-broker>                              Added message-service ❷

  <flex:remoting-destination ref="userDao"/>
                                                      ❸ Added message-destination
  <flex:message-destination id="flexMessage"/>

</beans>
```

In the listing you added the Spring `message-service` ❷ that points to the `my-polling-amf` channel you defined in the services-config.xml. Because you are using a Spring-managed `MessageBroker` ❷, you can specify the message destlnatlon ❸ by adding the element and giving it the id you want to refer to in the server-side implementation. You will use the message destlnatlon to send messages from the server-side objects.

MODIFY THE APPLICATIONCONTEXT.XML

The last bit of configuratlon detail is found in the Spring applicatlonContext.xml. This file is also located in the src/main/webapp/WEB-INF directory of the `flex-bugs-web` module. The code listed here shows the two Spring beans you need to define in the applicatlonContext.xml.

```
<bean id="defaultMessageTemplate"
       class="org.springframework.flex.messaging.MessageTemplate" />

<bean id="issueService" class="org.foj.service.impl.IssueManagerImpl">
    <constructor-arg ref="issueDao"/>
    <constructor-arg ref="commentService"/>
    <constructor-arg ref="defaultMessageTemplate"/>
</bean>
```

So that you can push messages to the message destlnatlon from your Java objects, Spring has given us the `MessageTemplate` helper class. Inject an instance of the

MessageTemplate into the issueService and use it in the IssueManagerImpl. Because the MessageTemplate is configured as a Spring bean, it autodetects the MessageBroker. Otherwise, it would need to be configured.

MODIFY THE ISSUEMANAGERIMPL

Now that you have the Spring configuration squared away, you can take advantage of the injected MessageTemplate object by adding it to the IssueManagerImpl class as seen in listing 6.3.

Listing 6.3 Adding the injected MessageTemplate

```
...
import org.springframework.flex.messaging.MessageTemplate;

@WebService(serviceName = "IssueService",
➥endpointInterface = "org.foj.service.IssueManager")
public class IssueManagerImpl implements IssueManager {
  private GenericDao<Issue, Long> issueDao;
  private CommentManager commentManager;
  private MessageTemplate messageTemplate;          ❶ MessageTemplate field

  public IssueManagerImpl(GenericDao<Issue, Long> issueDao,
                          CommentManager commentManager,
                          MessageTemplate messageTemplate)
  {                                                  ❷ Constructor arguments
    this.issueDao = issueDao;
    this.commentManager = commentManager;
    this.messageTemplate = messageTemplate;
  }                                                  ❸ Set messageTemplate
 ...

  public Issue save(Issue issue) {

    String messageBody = "Issue was saved";
    messageTemplate.send("flexMessage", messageBody);
    return issueDao.save(issue);                     ❹ Updated save method
  }

  public void remove(Long id) {
    commentManager.deleteAllCommentsForIssueId(id);

    String messageBody = "Issue was removed";        ❺ Updated remove method
    messageTemplate.send("flexMessage", messageBody);
    issueDao.remove(id);
  }

}
```

To take advantage of the Spring MessageTemplate, you first create a private field to store the MessageTemplate instance ❶ that you inject. Because you are injecting through the constructor ❷ you set the private field ❸ to equal the injected instance.

In the example you will use the MessageTemplate to send the client a text message notification. This message will let the client know that a refresh is needed. You wire up the notification to the save ❹ and remove ❺ methods and pass a message to the client indicating that the issue was either saved or removed.

Now that all the necessary changes are in place for simple messaging with Spring and Flex, it's time to update the Flex client.

6.2 Modifying the client for messaging

The Flex client changes are extremely simple. The use of the MVP design pattern isolates the changes to only two locations. We're going to create a `ChannelSetFactory` to leverage the power of the `ChannelSet` object for communicating with BlazeDS, and we'll modify the `IssueModel` object to receive the issue model updates from the polling channel.

6.2.1 Creating a ChannelSetFactory

Similar to the approach you took earlier with the `EventFactory`, you create a singleton instance of a `ChannelSet` and retrieve it so that all your components are using the same `ChannelSet`. Using a `ChannelSet` allows you to leverage communication channels such as `AMFChannel`, `HTTPChannel`, `StreamingAMFChannel`, and `StreamingHTTPChannel` at the same time, creating redundancy and fail-safes that help guarantee your data gets transmitted. To do this you'll create a `ChannelSetFactory` in the `flex-bugs-lib` module.

Listing 6.4 ChannelSetFactory.as

```
package org.foj.model {
import mx.messaging.ChannelSet;
import mx.messaging.channels.AMFChannel;

public class ChannelSetFactory {

  private static var _messagingChannelSet:ChannelSet;        ❶ Singleton instance

  public function ChannelSetFactory() {
  }

  public static function getMessagingChannel():ChannelSet {     ❷ Static factory method
    if (_messagingChannelSet == null) {
      var pollingChannel:AMFChannel = new AMFChannel("my-polling-amf",
          "http://localhost:8080/flexbugs/messagebroker/amfpolling");
      _messagingChannelSet = new ChannelSet();
      _messagingChannelSet.addChannel(pollingChannel);         AMFChannel ❸
    }

    return _messagingChannelSet;
  }

}
}
```

The code for the `ChannelSetFactory` is similar to the `EventDispatcherFactory`. Start by declaring your private instance variable to hold the `ChannelSet` ❶, then define a static factory method that the rest of the application will use to retrieve the ChannelSet ❷. Inside this method you check to see if the instance has been instantiated; if not create one and assign your AMFChannel to it ❸. All that's left is to return the ChannelSet to the caller. Next let's look at modifying the IssueModel to leverage this ChannelSet.

6.2.2 *Changing the IssueModel*

The final task to perform, in your sample messaging design, is to fire off an event when the client receives a push notification message which should refresh the issue master view DataGrid. You do this by adding a Consumer of the messaging channel.

Listing 6.5 The IssueModel message Consumer

```
package org.foj.model {

...

import mx.messaging.events.MessageEvent;                     ❶ Import Flex
                                                               MessageEvent class
public class IssueModel {

  private var _issueService:RemoteObject;

  public function IssueModel() {

    var defaultChannelSet:ChannelSet = ChannelSetFactory.getDefaultChannel();

    _issueService = new RemoteObject();
    _issueService.destination = "issueService";
    _issueService.channelSet = defaultChannelSet;

   var messagingChannelSet:ChannelSet                         ❷ Get messaging
     = ChannelSetFactory.getMessagingChannel();                  ChannelSet

    var consumer:Consumer = new Consumer();
    consumer.destination = "flexMessage";
    consumer.channelSet = messagingChannelSet;
    consumer.addEventListener(MessageEvent.MESSAGE, messageHandler);
    consumer.subscribe();

  }                                                           Flex Consumer ❸

  private function messageHandler(event:MessageEvent):void{
    var eventDispatcher:EventDispatcher =
              EventDispatcherFactory.getEventDispatcher();

    var refreshEvent:UIEvent = new                            messageHandler ❸
     UIEvent(UIEvent.REFRESH_ISSUES_BUTTON_CLICKED);
    eventDispatcher.dispatchEvent(refreshEvent);
  }

...

 }
}
```

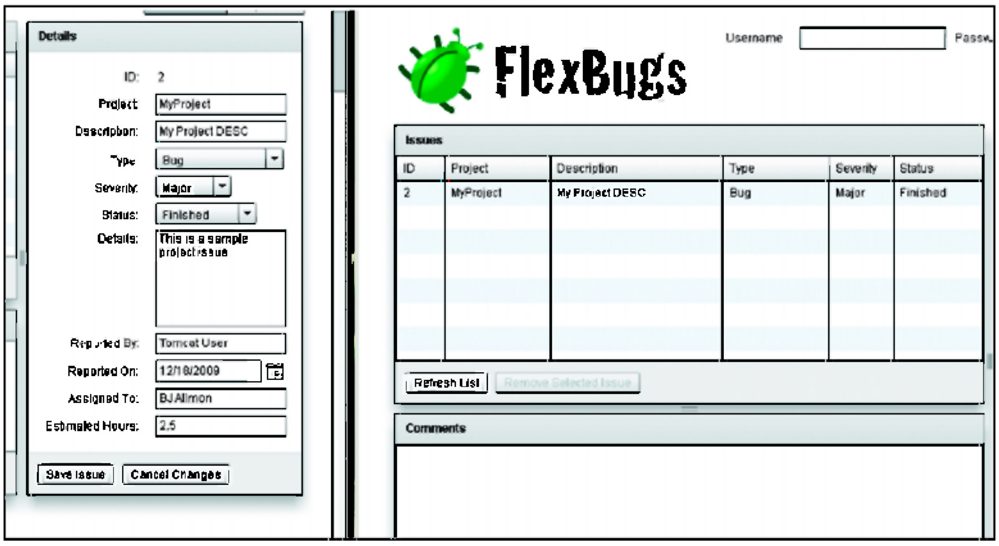
In the `IssueModel` you receive notification of changes through the built-in Flex Messaging API. After importing the `MessageEvent` class ❶, you need to get an instance of the messaging AMF `ChannelSet` from the `ChannelSetFactory` ❷.

Next you need to create a Flex `Consumer` ❸ and establish its properties. You configured the consumer's destination to be the `flexMessage` destination, defined in the `flex-spring-servlet.xml`. The consumer's `ChannelSet` was set to the one you got from the `ChannelSetFactory`.

The event listener is an important piece. It allows you to invoke a method when a consumer receives a message from the server. You declared that you wanted the event to be the `MessageEvent` type `Message` and that the method to call is `messageHandler` ❹. Finally, you subscribe to the server-side message destination by calling `consumer.subscribe()`.

Now that your Flex client is set up to subscribe to a messaging destination, you can see it in action. This is possible by opening up two different browser instances; make changes in one while keeping an eye on the other as in figure 6.2 where the browser snippet on the left shows the issue form that persisted the issue and the browser on the right automatically gets the update.

Changes should be noticeable in the issues `DataGrid` as changes occur in the application to the issues. The changes will occur within 4 seconds, as specified in the channel definition. That's all there is to setting up a simple messaging solution with Flex!



ID:	2
Project:	MyProject
Description:	My Project DESC
Type:	Bug
Severity:	Major
Status:	Finished
Details:	This is a sample project issue
Reported By:	Tomcat User
Reported On:	12/18/2009
Assigned To:	BJ Allmon
Estimated Hours:	2.5

ID	Project	Description	Type	Severity	Status
2	MyProject	My Project DESC	Bug	Major	Finished

Figure 6.2 Trying out the messaging using two browsers

6.3 Summary

In this chapter you configured BlazeDS, Spring, and a Flex client for a simple messaging architecture. You changed certain Java service methods so that they would notify the client of changes and made changes to the IssueModel to listen as a consumer. Using the Flex messaging API you used a Consumer to subscribe to the messaging channel service and received dynamic updates for the Issues master view. You will take messaging a step farther by configuring JMS when discussing Flex with Grails.

In the next chapter you will take a peek at securing and personalizing an application. The Spring security framework will be used with its annotation to provide great flexibility with the least amount of complexity.

Part 3

The joys of Flex on Java

Part 3, chapters 7 through 11, goes beyond what you'd find in most books on Flex and Java. You will cover topics like security and personalization because most applications need to implement a security strategy, and personalization usually comes next.

You will also cover charting with Degrafa, an open source Flex drawing API, and adding a chart to the example application.

One feature that sets the Flex framework apart from other web frameworks is its ability to run as a native desktop application. With this in mind, you will refactor the example application to include a deployment to the desktop version of the application.

Because writing code without tests is irresponsible, these final chapters demonstrate how to test your Flex application with FlexUnit. We not only provide basics on test-driven development (TDD), we break down the example application's structure and show you how to maintain good coverage across the entire application.

The last chapter covers Flex and Grails development. This is an exciting combination of technologies. You will quickly construct a new example application with Grails and build a Flex frontend for it.

7 Securing and personalizing your application

This chapter covers

- Authentication
- Authorization
- Personalization

In the past couple of chapters you've experienced some of the great integration features that BlazeDS gives us. Now you're going to take integration one step further as you strengthen your application and add security features.

You'll leverage the existing security infrastructure provided by AppFuse and not have to spend precious time on the particulars of setting up a Lightweight Directory Access Protocol (LDAP) server, authenticating against Active Directory, and creating Access Control Lists (ACLs). There are plenty of resources on the web that cover these advanced topics, which are beyond the scope of our goals for this chapter. The information you cover in this chapter should be sufficient for about 90% of the applications that you'll encounter.

You'll take an iterative approach to adding security to the sample application, starting by adding simple login and logout functionality, allowing the application to authenticate using the same mechanism that AppFuse uses internally. You'll build upon that by adding security constraints to the services and lock down the destructive method calls to only users belonging to specific roles. You'll also learn about an often-overlooked aspect of security, personalization. Before getting started, let's cover basic concepts of the Spring Security framework.

7.1 Authentication

The simplest form of security that you can add to the application is to allow a user to enter his user name and password and authenticate using the server side. For many applications this is usually sufficient. There are many strategies for authenticating users, from rolling your own authentication methods and exposing them as remote services for the application to use, to implementing basic authentication measures using the web server, or in your case using a security framework like Spring Security.

AppFuse provides, out of the box, a simple authentication mechanism based on user information being stored in the database, so you don't need to go through the trouble of setting up an authentication provider. The big advantage to leveraging a framework like Spring Security is that you can change out the mechanism for authentication from a database table to an LDAP server without having to change anything in your Flex application. So now let's get started with implementing the components you need to allow users to authenticate using your application.

7.1.1 Modifying the ChannelSetFactory

Even though all the remote components that you use to communicate with external services have methods supporting sending credentials for authentication, this is not an ideal solution because the application would need to hold the username and password, in order for you to manually pass them along with every remote method call you made. Not only is this tedious, it's error prone. Instead you'll leverage the `ChannelSet` to authenticate the user to the server side and maintain a session until you either close the application or log out.

To do this you'll modify the `ChannelSetFactory` that you created in the last chapter as shown in listing 7.1

Channels and ChannelSets

Flex uses different methods of communicating with BlazeDS on the server side depending on the type of communication: `AMFChannel`, `HTTPChannel`, `StreamingAMFChannel`, and `StreamingHTTPChannel`. These channels encapsulate the behavior of connecting and maintaining communications with the BlazeDS components on the server side. You may recall that when setting up BlazeDS you configured an `AMFChannel` in the services-config.xml file providing a means of connecting to BlazeDS using your `RemoteObject` components. The Flex remoting components such as `RemoteObject` allow you to assign a set of these channels for use by the components, giving you fallback and failover behavior out of the box, as well as allowing users to authenticate to the server side.

Listing 7.1 ChannelSetFactory.as

```
package org.foj.model {
import mx.messaging.ChannelSet;
import mx.messaging.channels.AMFChannel;

public class ChannelSetFactory {                                    ❶ Singleton
                                                                      instance
  private static var _defaultChannelSet:ChannelSet;              <┘
  private static var _messagingChannelSet:ChannelSet;

  public function ChannelSetFactory() {
  }                                                                   ❷ Static factory
                                                                        method
  public static function getDefaultChannel():ChannelSet {          <┘
    if (_defaultChannelSet == null) {
      var channel:AMFChannel = new AMFChannel("my-amf",                <┐ AMF
          "http://localhost:8080/flexbugs/messagebroker/amf");          ❸ Channel
      _defaultChannelSet = new ChannelSet();
      _defaultChannelSet.addChannel(channel);
    }

    return _defaultChannelSet;
  }

  public static function getMessagingChannel():ChannelSet {
    if (_messagingChannelSet == null) {
      var pollingChannel:AMFChannel = new AMFChannel("my-polling-amf",
          "http://localhost:8080/flexbugs/messagebroker/amfpolling");
      _messagingChannelSet = new ChannelSet();
      _messagingChannelSet.addChannel(pollingChannel);
    }

    return _messagingChannelSet;
  }

}
}
```

You start by declaring a private instance variable to hold the `ChannelSet` ❶. Then you add another statlc factory method that the rest of the application will use to retrieve the `ChannelSet` ❷. Inside this method you check to see if the instance has been instantiated, and if not create one and assign your `AMFChannel` to it ❸. Then return the `ChannelSet` to the caller.

Next let's create the custom event class for your login panel to use.

7.1.2 Creating a UserEvent

The `UserEvent` class should look familiar; it's another custom event class just like the one you created for the rest of the events that your application uses. You're creatlng this one separate from the other event class, because it is a separate area of concern from UI events. This will make it easier to extract the login panel to a separate library later, by keeping all of the login functlonality separate from the rest of the applicatlon logic.

Listing 7.2 UserEvent.as

```
package org.foj.event {
import flash.events.Event;

public class UserEvent extends Event{

  public static var LOGIN_BUTTON_PRESSED:String =
      "loginButtonPressed";
  public static var LOGOUT_BUTTON_PRESSED:String =
      "logoutButtonPressed";
  public static const USER_LOGGED_IN:String =
      "userLoggedIn";
  public static const CURRENT_USER_UPDATED:String =
      "currentUserUpdated";

  public var data : *;

  public function UserEvent(type : String,
                            bubbles : Boolean = true,
                            cancelable : Boolean = false)
  {
    super(type, bubbles, cancelable);
  }

}
}
```

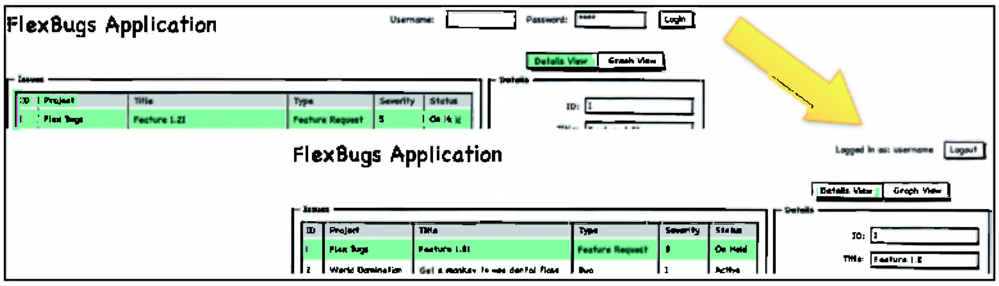
❶ Event type constants
❷ Data field
❸ Overloaded constructor
❹ Call to super

Listing 7.2 shows your custom `UserEvent`. You begin just as you did in the last event class by defining constants to describe what events you'll be using this class for ❶. Next you define a `data` member variable to carry any data with the event if necessary ❷. Next you declare an overloaded constructor ❸ and inside this is a call to the base constructor in the `Event` class ❶. Now that you've defined the custom event, let's put it to use and start to create the login panel.

7.1.3 Creating a login panel

There are many ways to implement the login functionality. You can create a separate login screen, a pop up, or implement it inline. Because you're not going to attempt to prevent users from accessing and browsing the application unless they're logged in, having a separate login screen that takes users away from the application doesn't make sense. Neither does blocking the application with a modal dialog forcing them to login, so you're going with the simplest possible solution, which is to present the user with a couple of fields in the upper-right side of the application for entering the username and password.

The login panel will use a `ViewStack` just as your main application does, so that as users log in, the look of the login panel will change to reflect that the user has logged in successfully as shown in figure 7.1. When the user logs out, the view will change back to the initial logged out state.



ID	Project	Title	Type	Severity	Status
1	Flex Bugs	Feature 1.21	Feature Request	5	On Hold

ID	Project	Title	Type	Severity	Status
1	Flex Bugs	Feature 1.81	Feature Request	8	On Hold
2	World Domination	Get a monkey to use dental floss	Bug	1	Active

Figure 7.1 Logging in to the application

Listing 7.3 LoginPanel.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"
         creationComplete="init()">

  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[
    import mx.messaging.ChannelSet;

    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.UserEvent;
    import org.foj.model.ChannelSetFactory;
    import org.foj.presenter.LoginPresenter;

    private var presenter:LoginPresenter;

    private function init():void {
      presenter = new LoginPresenter(this);
    }

    private function login():void {
      var loginEvent:UserEvent =
          new UserEvent(UserEvent.LOGIN_BUTTON_PRESSED);
      EventDispatcherFactory.getEventDispatcher()
          .dispatchEvent(loginEvent);
    }

    private function logout():void {
      var logoutEvent:UserEvent =
          new UserEvent(UserEvent.LOGOUT_BUTTON_PRESSED);
      EventDispatcherFactory.getEventDispatcher()
          .dispatchEvent(logoutEvent);
    }

    ]]>
  </fx:Script>
```

1. Extends Group
2. Set layout for component
3. Presenter for login component
4. Init method
5. Event handler for the login button
5. Event handler for the logout button

```
  <mx:Spacer width="100%"/>
                                                          ❼ View stack
  <mx:ViewStack id="loginStack"
                paddingTop="5"
                paddingBottom="5">
    <mx:HBox id="loggedOut">
      <mx:FormItem label="Username">                        ❽ LoggedOut view
        <mx:TextInput id="usernameInput" width="150"/>
      </mx:FormItem>
      <mx:FormItem label="Password">
        <mx:TextInput id="passwordInput" width="150"
     displayAsPassword="true"/>
      </mx:FormItem>
      <mx:Button id="loginLink" label="Login" click="login()"/>

    </mx:HBox>
    <mx:HBox id="loggedIn">
      <mx:Spacer width="100%"/>                              ❸ LoggedIn view
      <mx:Text id="loginLabel" text="Logged in as: "/>
      <mx:Text id="userName"/>
      <mx:Button id="logoutLink" label="Logout" click="logout()"/>
    </mx:HBox>

  </mx:ViewStack>
</s:Group>
```

Listing 7.3 shows the code for our LoginPanel. As with most of our other components, this panel will extend the Group component ❸, and you define its layout to be horizontal ❸. Next you define a private member variable for your Presenter ❸ and initialize it in your init method ❸ passing in a reference to the panel in the constructor. Then you define a couple of event handlers for the login button being clicked ❸ as well as the logout button being clicked ❻. These two event handlers follow the same pattern that you established in chapter 3. They create the appropriate UserEvent and dispatch it for your Presenter, and anyone else who is concerned can react.

You define the visual components starting with defining a ViewStack ❸ that will allow you to switch between the states that are possible for the login panel. Inside of the ViewStack component you create two HBox components, one for the loggedOut view state ❻ and one for the loggedIn view state ❸. Now you need to create the Presenter for the LoginPanel.

7.1.4 Creating a login Presenter

The LoginPresenter class starts out simply. It follows the same general pattern that all of the previous Presenters have, only having to react to a couple of button presses and not having to maintain any kind of state just yet.

Listing 7.4 LoginPresenter.as

```
package org.foj.presenter {

...

public class LoginPresenter {
    private var _view:LoginPanel;                                    ❶ View
    private var _model:LoginModel;                                   ❷ Model

    public function LoginPresenter(view:LoginPanel) {
      this._view = view;
      this._model = new LoginModel();                                ❸ Add event
                                                                       listener for
      EventDispatcherFactory.getEventDispatcher()                      login
          .addEventListener(UserEvent.LOGIN_BUTTON_PRESSED,
          login);
      EventDispatcherFactory.getEventDispatcher()                    ❹ Add event
          .addEventListener(UserEvent.LOGOUT_BUTTON_PRESSED,           listener for
          logout);                                                     logout
    }

    private function login(event:UserEvent = null):void {           ❺ Event
      var responder:IResponder = new AsyncResponder(                   handler for
          loginResult, handleError);                                   login event
      _model.login(_view.usernameInput.text,
          _view.passwordInput.text, responder);
    }

    private function logout(event:UserEvent = null):void {          ❻ Event
      var responder:IResponder = new AsyncResponder(                   handler for
          logoutResult, handleError);                                  logout event

      _model.logout(responder);
    }

    private function loginResult(event:ResultEvent,                  ❼ Event
                                 token:Object = null):void {           handler for
      _view.loginStack.selectedChild = _view.loggedIn;                 login result
      _view.passwordInput.text = "";

      Alert.show("You have logged in as: " + event.result.name);
    }

    private function logoutResult(event:ResultEvent,                 ❽ Event
                                  token:Object = null):void {          handler for
      _view.loginStack.selectedChild = _view.loggedOut;                logout result
      Alert.show("You have successfully logged out");
    }

    private function handleError(event:FaultEvent,
                                 token:AsyncToken = null):void {     ❾ Error
      CursorManager.removeBusyCursor();                                handler
      Alert.show(event.fault.faultString);
    }

  }
}
```

You start your Presenter by defining the _view ❶ and _model ❷ instance variables. In the constructor you assign the view reference passed in to your instance variable and create a new instance of your model. Then you add an event listener for LOGIN_BUTTON_PRESSED event ❸, assigning the login handler ❺, and an event listener for the LOGOUT_BUTTON_PRESSED event ❹, assigning the logout handler ❻ to it.

Inside the login handler, you create an AsyncResponder to pass into the model, pointlng it to the loginResult handler method ❼ and the error handler ❾. Then you call the login method on your model passing in the username, password, and the responder. Inside the logout handler, you similarly create a responder pointing to the logoutResult ❽ and error handler methods, then call the logout method on your model passing in the responder.

When the login method from the model responds, your login result handler is invoked, and switches the current view state of the login panel, blanks out the password box contents so that the user will be forced to type in a password the next tlme she tries to login, and displays an Alert statlng that she has logged in successfully. On the other side of the equation, when the logout method from the model responds, the logout result handler is called, which switches the view stack back to the initlal logged out state, and alerts the user that she has logged out.

7.1.5 Creating a login manager

Now that the Presenter is created, you need to implement the model so that you can authentlcate users against the application. The following listlng shows the LoginModel for the LoginPanel.

Listing 7.5 LoginModel.as

```
package org.foj.model {
import mx.messaging.ChannelSet;
import mx.rpc.AsyncToken;
import mx.rpc.IResponder;
import mx.rpc.remoting.RemoteObject;

public class LoginModel {

  private var _defaultChannelSet:ChannelSet;                 Get default ❶
                                                              ChannelSet
  public function LoginModel() {
    _defaultChannelSet = ChannelSetFactory.getDefaultChannel();
  }

  public function login(username:String,
                        password:String,                      ❷ Login
                        responder:IResponder):void {            method

    if (!_defaultChannelSet.authenticated) {
      var token:AsyncToken = _defaultChannelSet.login(username, password);
      token.addResponder(responder);
    }
  }
                                                              ❸ Logout
  public function logout(responder:IResponder):void {           method
    var token:AsyncToken = _defaultChannelSet.logout();
    token.addResponder(responder);
  }
}
}
```

The model for the `LoginPanel` is simple to start with. Inside the constructor for the model, you get the default `ChannelSet` from your `ChannelSetFactory` ❶, and assign it to an instance variable. Inside of the `login` method ❷, you first check to see if the user is already logged in, as this sometimes results in an error condition, and BlazeDS will complain about the `ChannelSet` being already authenticated. If the user is not logged in, it will then call the `login` method on the `ChannelSet`. The `logout` method ❸ calls `logout` on the `ChannelSet`, not having to worry about checking to see if the user is currently logged in.

7.1.6 *Updating the header*

Now you need to update your Header.mxml to include the newly created `LoginPanel`.

Listing 7.6 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"
         xmlns:view="org.foj.view.*"
         width="100%"
         height="100">

  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

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

    public var viewStack:ViewStack;
    ]]>
  </fx:Script>

  <mx:Spacer width="5"/>
  <s:SimpleText text="Flex Bugs Application"
                height="100%"
                fontSize="32"
                fontWeight="bold"
                verticalAlign="middle"/>
  <mx:Spacer width="100%"/>
  <s:VGroup height="100%">
    <view:LoginPanel height="100%"/>                    ❶ LoginPanel
    <s:HGroup id="toggleButtonPanel" width="100%">
      <mx:Spacer width="100%"/>
      <mx:ToggleButtonBar dataProvider="viewStack"/>
    </s:HGroup>
  </s:VGroup>
  <mx:Spacer width="5"/>

</s:Group>
```

The changes necessary to add the login panel to the header are trivial. You've made a few minor tweaks to the layout and spacing components and only had to add one line of XML to add the `LoginPanel` ❶ to the header. When that is done, all that's left is to configure BlazeDS to enable security.

7.1.7 Enabling security for Flex

In order to enable security for our application, you need to make a small modification to the flex-spring-servlet.xml context file.

Listing 7.7 flex-spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:flex="http://www.springframework.org/schema/flex"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
         http://www.springframework.org/schema/flex
         http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">

  <flex:message-broker>
    <flex:secured/>
  </flex:message-broker>

</beans>
```

With all of that in place, it's time to run the application and log in. By default, two users are created in AppFuse, a regular user with the `ROLE_USER` authority and an administrator with the `ROLE_ADMIN` authority. The regular user's username is *user* with a password of *user*, and in case you hadn't guessed, the administrator's username is *admin* with a password of *admin*. After you build and run the application you should be presented with something that looks like figure 7.2.

Log in with the user and user account, and you'll be presented with an Alert window telling you that you've successfully logged in. If you mistype the password, you should be presented with an error message stating that your credentials were

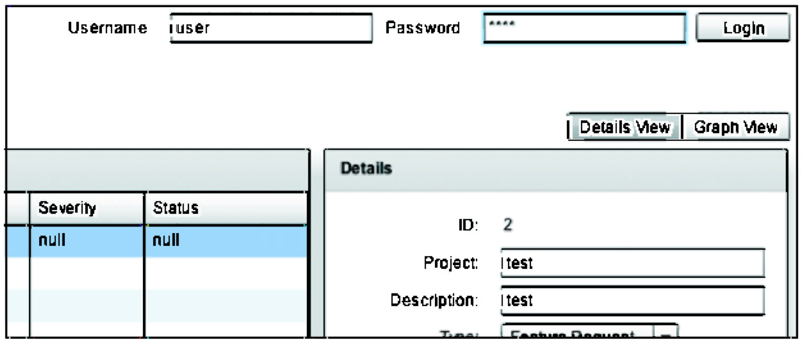


Figure 7.2 The login panel in action

incorrect. This was the first piece of functionality you wanted to implement in this chapter. Now let's move on to adding authorization to the application.

7.2 Authorization

Sometimes authenticating the users isn't enough. You might want to be able to selectively secure specific parts of the application and not others. You can allow users who are not logged in to be able to browse the list of open bugs and comments, but restrict logging new bugs to authenticated users. This is where authorization comes into play.

Many application frameworks do not have this level of support, so you end up having to put crosscutting concerns such as authorization throughout the code, surrounding certain functionality with checks to see if the user is authenticated or has permissions to access that particular piece of functionality. In this area the Spring Security framework shines.

Spring Security leverages aspect-oriented programming (AOP) to allow you to apply complex rules for who should be able to execute method calls. You can even get such fine-grained control that you can allow logged in users to add new issues and comments and make changes to existing issues and comments, but restrict deleting to administrators. We won't be discussing AOP much in this book, so if you'd like to learn more about how to use AOP with Spring, check out Craig Walls' excellent book *Spring in Action*. The third edition is to be published in December 2010. (http://manning.com/walls4/).

7.2.1 Flex Spring Security primer

The Spring BlazeDS Integration provides several different methods of securing the application. You'll start with the broadest control and work your way to more and more fine-grained control. The first method you could use is to secure an entire AMF channel using the flex-spring-servlet.xml as shown here, but this would not be desirable because you lose all ability to define different levels of security for different parts of the application.

```
<flex:message-broker>
    <flex:secured>
        <flex:secured-channel channel="my-amf" access="ROLE_USER" />
 </flex:secured>
</flex:message-broker>
```

The next method is to secure the application by adding security constraints directly to the destinations by configuring the destination in a remoting-config.xml as shown here.

```
<destination id="issueService">
    ...
    <security>
        <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
                <role>ROLE_USER</role>
            </roles>
        </security-constraint>
    </security>
</destination>
```

This would apply a security constraint for the issueService, on a system-wide level, allowing only users in the role ROLE_USER access to this destination. This method of securing an application, while simple, will not work for what you're trying to do with the application for a couple of reasons. This would effectively cut off any ability to do method-level authorization for the application. In addition, you are not defining the destinations using a remoting-config.xml. You're allowing the Spring BlazeDS Integration to dynamically configure the remoting destinations for you.

Moving on to a more finely grained security, you can set up a global method interceptor using AOP pointcut syntax to define a pattern of methods you want to secure. The following code shows an example of this taken from the security.xml file provided by AppFuse.

```
<global-method-security>
    <protect-pointcut
      expression=" execution(* *..service.UserManager.getUsers(..))"
      access="ROLE_ADMIN"/>
</global-method-security>
```

This will constrain any call to a method named getUsers contained in a class or interface named UserManager whose package name ends with "service". As you can see this method allows you to apply very complex patterns to define which methods get secured.

```
<bean id="issueService" class="org.foj.service.impl.IssueManagerImpl">
    <constructor-arg ref="issueDao"/>
    <constructor-arg ref="commentService"/>
    <security:intercept-methods>
        <security:protect method="save"
            access="ROLE_USER,ROLE_ADMIN" />
        <security:protect method="remove*"
            access="ROLE_ADMIN" />
    </security:intercept-methods>
  </bean>
```

The code shows how to secure methods at the bean level in XML by using the Spring application context. This gives you more localized control over the security than the global method illustrated previously, keeping the security constraints closer to the code that it affects.

```
public interface IssueManager {

  ...

  @Secured({"ROLE_USER", "ROLE_ADMIN"})
  @RemotingInclude
  Issue save(Issue issue);
  @Secured({"ROLE_ADMIN"})
  @RemotingInclude
  void remove(Long id);

}
```

The code shows the final method of securing the business methods by using the `@Secured` annotation. This is the most localized method of security and the method you're going to use in upcoming examples. This keeps the declared security constraints right with the code you're trying to secure. The other advantage to using the annotations over the other methods is that if people decide to override what you've defined in the code, they can override it using the XML configuration.

7.2.2 Spring Integration Security

In order for the Flex application to be able to take advantage of using Spring Security for authorization, you first need to add the Spring Integration Security dependency to the pom.xml in the `flex-bugs-web` project.

Listing 7.8 Spring Integration Security dependency

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-security</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>
```

By adding this dependency, BlazeDS will use a special `LoginCommand` object on the server side that enables the `ChannelSet` login and logout functionality to integrate with Spring Security's authorization mechanisms by returning the username and any authorities that the user has in the result event. It also does error translating on the server side to translate security exceptions that may occur into their BlazeDS security exception equivalent. Now the error can be reported back to the Flex client instead of returning an HTTP 403 status code, which will break the application. You can read more about the Spring Integration Security in the docs for the Spring BlazeDS Integration at http://static.springsource.org/spring-flex/docs/1.0.x/reference/html/.

7.2.3 @Secured annotations

There are a number of methods you can use to secure your application; we have chosen to continue along the path of using annotations where possible to declare which interface methods we want to secure and what security constraints we want to place on them. We'll start by adding the necessary annotations to the `IssueManager`.

Listing 7.9 IssueManger.java

```
package org.foj.service;

import org.foj.model.Issue;
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.security.annotation.Secured;

import javax.jws.WebService;

@WebService
@RemotingDestination(channels = {"my-amf"})
public interface IssueManager {

  @RemotingInclude
  java.util.List<Issue> getAll();

  @RemotingInclude
  Issue get(Long id);                                    ❶ save method
  @Secured({"ROLE_USER", "ROLE_ADMIN"})                    annotation

  @RemotingInclude
  Issue save(Issue issue);

  @Secured({"ROLE_ADMIN"})                                 remove method
  @RemotingInclude                                       ❷ annotation
  void remove(Long id);

}
```

You've added an `@Secured` annotation to the save method ❶ and declared that only users having `ROLE_USER` and `ROLE_ADMIN` privileges can add and update issues. You defined both roles since your roles are not overlapping, meaning that `ROLE_ADMIN` is not a superset of `ROLE_USER` but rather separate altogether. Next you declare the `remove` method to be usable only by those who have the `ROLE_ADMIN` granted to them ❷. Next you add the necessary annotatlons to the `CommentManager`.

Listing 7.10 CommentManager.java

```
package org.foj.service;

import org.foj.model.Comment;
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.security.annotation.Secured;

import javax.jws.WebService;
import java.util.List;

@WebService
@RemotingDestination(channels = {"my-amf"})
public interface CommentManager {

  @RemotingInclude
  List<Comment> findCommentsByIssueId(Long issueId);

  @Secured({"ROLE_ADMIN"})                                deleteAllCommentsForIssueId
  @RemotingInclude                                      ❶ method annotation
  void deleteAllCommentsForIssueId(Long issueId);

  @RemotingInclude
  Comment get(Long id);
  @Secured({"ROLE_USER", "ROLE_ADMIN"})                  ❷ save method
  @RemotingInclude                                          annotation
  Comment save(Comment comment);

  @Secured({"ROLE_ADMIN"})                               ❸ remove method
  @RemotingInclude                                          annotation
  void remove(Long id);

}
```

Similar to the IssueManager annotatlons, you start by declaring that only members of ROLE_ADMIN can call the deleteAllCommentsForIssueId ❶ and remove ❸ methods. Then you declare that the save method can be called by users with either ROLE_USER or ROLE_ADMIN privileges ❷. Notice that you never specified any security constraints for the get methods of your services. This ensures that anyone can call them whether or not they're authentlcated to the application. Now that you've declared the security with annotatlons, you need to enable support for these annotations in Spring Security.

7.2.4 Overriding default security settings

AppFuse has Spring Security installed and configured out of the box, so to extend upon that you need to extract its security.xml configuratlon and put that in the flex-bugs-web project in the WEB-INF folder alongside the other Spring configuration files. To do this you copy the security.xml that is included in the appfuse-web-common.war that basically is overlaid on top of the flex-bugs-web.war when you build this project.

Listing 7.11 security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"

  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/
      spring-security-2.0.1.xsd">
                                                         ❶ Security by
  <http auto-config="true" lowercase-comparisons="false">   url pattern
    <intercept-url pattern="/admin/*" access="ROLE_ADMIN"/>
    <intercept-url pattern="/passwordHint.html*"
                   access="ROLE_ANONYMOUS,ROLE_ADMIN,ROLE_USER"/>
    <intercept-url pattern="/signup.html*"
                   access="ROLE_ANONYMOUS,ROLE_ADMIN,ROLE_USER"/>
    <intercept-url pattern="/a4j.res/*.html*"
                   access="ROLE_ANONYMOUS,ROLE_ADMIN,ROLE_USER"/>
    <form-login login-page="/login.jsp"
                authentication-failure-url="/login.jsp?error=true"
                login-processing-url="/j_security_check"/>
    <remember-me user-service-ref="userDao"
                 key="e37f4b31-0c45-11dd-bd0b-0800200c9a66"/>
  </http>

  <authentication-provider user-service-ref="userDao">
    <password-encoder ref="passwordEncoder"/>
  </authentication-provider>

  <global-method-security
        secured-annotations="enabled" jsr250-annotations="enabled">
    <protect-pointcut
        expression="execution(* *..service.UserManager.getUsers(..))"
        access="ROLE_ADMIN"/>
    <protect-pointcut
        expression="execution(* *..service.UserManager.removeUser(..))"
        access="ROLE_ADMIN"/>
  </global-method-security>
</beans:beans>
```

❷ Authentication provider

❸ Enable security annotations

Listing 7.11 shows the modified security.xml, which for the most part is identical to the security.xml included in AppFuse. The first section ❶, which is configured by AppFuse, dictates which URL patterns to intercept and pass through the security filters. You don't need to specify anything in this section for the Flex application as the Spring Integration Security handles all the filtering for the Flex and BlazeDS interactions. The second section ❷ defines the authentication provider, which in this case is using the default userDao provided by AppFuse. If you wanted to change that method of authentication to an LDAP server, for instance, you would configure it there.

In the last segment ❸ you added a couple of attributes to the global-method-security tag to set the secured-annotations to enabled and the jsr250-annotations to enabled as well.

7.2.5 Updating IssueModel and CommentModel

Now all that's left to enable authorization to work in the Flex application is to modify the models to use the ChannelSetFactory you created earlier. The following listing shows the changes for the IssueModel.

Listing 7.12 IssueModel.as

```
package org.foj.model {

...

public class IssueModel {

  private var _issueService:RemoteObject;

  public function IssueModel() {

    var defaultChannelSet:ChannelSet =
        ChannelSetFactory.getDefaultChannel();

    _issueService = new RemoteObject();
    _issueService.destination = "issueService";
```

❶ Get ChannelSet

```
      _issueService.channelSet = defaultChannelSet;        Set channelSet
    }                                                      on service ❷
    ...
  }
}
```

As you can see the impact of adding this functionality to the Flex application is minimal. First you get the default `ChannelSet` from the `ChannelSetFactory` ❶, just as you did earlier for the `LoginModel`, and you set the `channelSet` property on the `RemoteObject` ❷ to the `defaultChannelSet`. The following listing shows the same changes for the `CommentModel`.

Listing 7.13 CommentModel.as

```
package org.foj.model {

...

public class CommentModel {

  private var _commentService:RemoteObject;

  public function CommentModel() {
    var defaultChannelSet:ChannelSet =                     ❶ Get ChannelSet
        ChannelSetFactory.getDefaultChannel();

    _commentService = new RemoteObject();
    _commentService.destination = "commentService";
    _commentService.channelSet = defaultChannelSet;        Set channelSet
  }                                                        on service ❷

  ...
}
}
```

Now you can build and run the application and if you try to save or delete an item without being logged in with the proper authorities granted, you will be presented with an error message like that in figure 7.3

Again, the error handling is simple, displaying the fault message in an Alert box. Notice that the error message accurately describes the error. The last security enhancement you're going to make to the application is to add personalization.

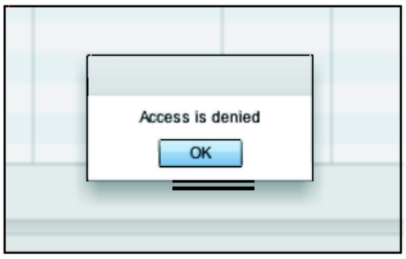

Access is denied

Figure 7.3 Error when trying to execute a method without authorization

7.3 Personalization

When you think about leveraging a security framework in the Flex application, you probably don't think about personalization, even though the two are closely related. Recall earlier when we discussed the Spring Integration Security module, you saw that one of the things returned in the `ResultEvent` is the username of the person who was authenticating.

Now that the user has authentlcated to the application, you can use the informa-tlon returned from the login process to get more informatlon such as the user's first and last names. Why not just use the username that the user typed into the login box you ask? Simple. If you wait for the login method to return successfully and take the value returned instead, you are guaranteed that the user has first authenticated, and that the username returned in the `ResultEvent` is the correct username for that user.

7.3.1 Adding the UserService to the LoginModel

Because the `ResultEvent` returned from logging in and out of the `ChannelSet` only includes the user name, you'll have to use another method to get the user's full name from the applicatlon. Fortunately AppFuse exposes this functlonality in the `userDao` object; you still have to tell BlazeDS to expose this to the Flex applicatlon. So you need to add the following line to the flex-spring-servlet.xml in the src/main/webapp/WEB-INF folder of the `flex-bugs-web` project:

```
<flex:remoting-destination ref="userDao"/>
```

This one line of xml configuratlon accomplishes the same thing you did in chapter 5 with the `@RemotingDestination` annotations, the difference being that you don't have to crack open the AppFuse source code to expose this Spring bean as a remote service to Flex. Now let's add this service to the `LoginModel`.

Listing 7.14 LoginModel.as

```
package org.foj.model {
import mx.messaging.ChannelSet;
import mx.rpc.AsyncToken;
import mx.rpc.IResponder;
import mx.rpc.remoting.RemoteObject;

public class LoginModel {                                   ❶ userService

  private var _userService:RemoteObject;                 <┘
  private var _defaultChannelSet:ChannelSet;

  public function LoginModel() {
    _defaultChannelSet = ChannelSetFactory.getDefaultChannel();

    _userService = new RemoteObject();                     <┐ Creating
    _userService.destination = "userDao";                  ❷ RemoteObject
    _userService.channelSet = _defaultChannelSet;
  }

  ...                                                      ❸ getUserDetails
  public function getUserDetails(username:String,         <┘ method
                                 responder:IResponder):void {
    var token:AsyncToken = _userService.loadUserByUsername(username);
    token.addResponder(responder);
  }

}
}
```

First you define an instance variable for the userService ❶ and create a new instance of it in the constructor ❷. As with other remote objects, you set its destlnatlon to userDao because that is what the Spring bean is defined as, and you set the channel-Set to the defaultChannelSet that you got from the ChannelSetFactory. As a final step you define a method called getUserDetails ❸, which takes as its first argument the username to look up and a responder to call back to when the service returns. Now let's update the LoginPresenter to use this new functlonality.

7.3.2 Updating the LoginPresenter

You need to make a couple of minor modificatlons to the LoginPresenter to use the method you just created in the LoginModel.

Listing 7.15 LoginPresenter.as

```
package org.foj.presenter {

...
public class LoginPresenter {

  private var _view:LoginPanel;
  private var _model:LoginModel;

  public function LoginPresenter(view:LoginPanel) {

    ...

    EventDispatcherFactory.getEventDispatcher()                ❶ Event listener for
        .addEventListener(UserEvent.USER_LOGGED_IN,              user logged in
        getUserDetails);

  }

  ...                                                            ❷ Event handler
                                                                   for user logged
  private function loginResult(event:ResultEvent,                  in result
                               token:Object = null):void {
    _view.loginStack.selectedChild = _view.loggedIn;
    _view.passwordInput.text = "";

    var userLoggedInEvent:UserEvent =
        new UserEvent(UserEvent.USER_LOGGED_IN);
    userLoggedInEvent.data = event.result.name;
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(userLoggedInEvent);                       ❸ Event handler
  }                                                                for user logged
                                                                   out result
  private function logoutResult(event:ResultEvent,
                                token:Object = null):void {
    _view.userName.text = "";
    _view.loginStack.selectedChild = _view.loggedOut;
    Alert.show("You have successfully logged out");

    var userChangedEvent:UserEvent =
        new UserEvent(UserEvent.CURRENT_USER_UPDATED);
    userChangedEvent.data = null;
    EventDispatcherFactory.getEventDispatcher()
```

```
        .dispatchEvent(userChangedEvent);                     Event handler for ❹
  }                                                              user logged in
  private function getUserDetails(event:UserEvent = null):void {            <┘
    var responder:IResponder = new AsyncResponder(
        getUserDetailsResult, handleError);

    _model.getUserDetails(event.data, responder);             ❺ Handler for
  }                                                              userService
  private function getUserDetailsResult(event:ResultEvent,    <┘  result
                                      token:Object = null):void {
    var user = event.result;
    _view.userName.text = user.fullName;
    Alert.show("Welcome Back " + user.fullName);

    var userChangedEvent:UserEvent =
        new UserEvent(UserEvent.CURRENT_USER_UPDATED);
    userChangedEvent.data = user;
    EventDispatcherFactory.getEventDispatcher()
        .dispatchEvent(userChangedEvent);
  }

  ...

}
}
```

The first change you make is to add a listener for the `USER_LOGGED_IN` event ❶ so that you know when you need to update the user details. Next you make a slight modification to the method that is called when the result comes back from the call to log in on the `IssueModel` ❷. Here you take the username from the event result and create a new `UserEvent` of type `USER_LOGGED_IN`, and you set the event's `data` property to the username. This allows you to pass along the username to whoever would want to respond to a user logging in. You do something similar in the logged out result handler ❸ except you create a `CURRENT_USER_UPDATED` event to notify other parts of the application that the currently logged in user has been changed, and set its `data` property to `null`.

Next you define the event handler for the `USER_LOGGED_IN` event listener you defined in the constructor ❹. Inside this method you create an `AsyncResponder` object and make a call to the `getUserDetails` method on the `IssueModel` you defined in the previous section. When the result from this call comes back, it is handled by the `getUserDetailsResult` method ❺ where you update the view to show the current user's full name, and you create a new `CURRENT_USER_UPDATED` event setting its `data` property to the `User` object that came back in the `ResultEvent`.

Now let's look at updating the `DetailPresenter` and `CommentsListPresenter`, which will be listening for the `CURRENT_USER_UPDATED` event that you just added.

7.3.3 Updating the DetailPresenter and CommentsListPresenter

Now that the `LoginPresenter` is broadcasting an event that notifies the rest of the application that the currently logged in user has changed, you can add event listeners to the other parts of the application that may want to listen for that. You'll start with the `DetailPresenter` so that you can default the text in the author field for any new issues to the currently logged in user. The following listing shows the changes needed to add this to the `DetailPresenter`.

Listing 7.16 DetailPresenter

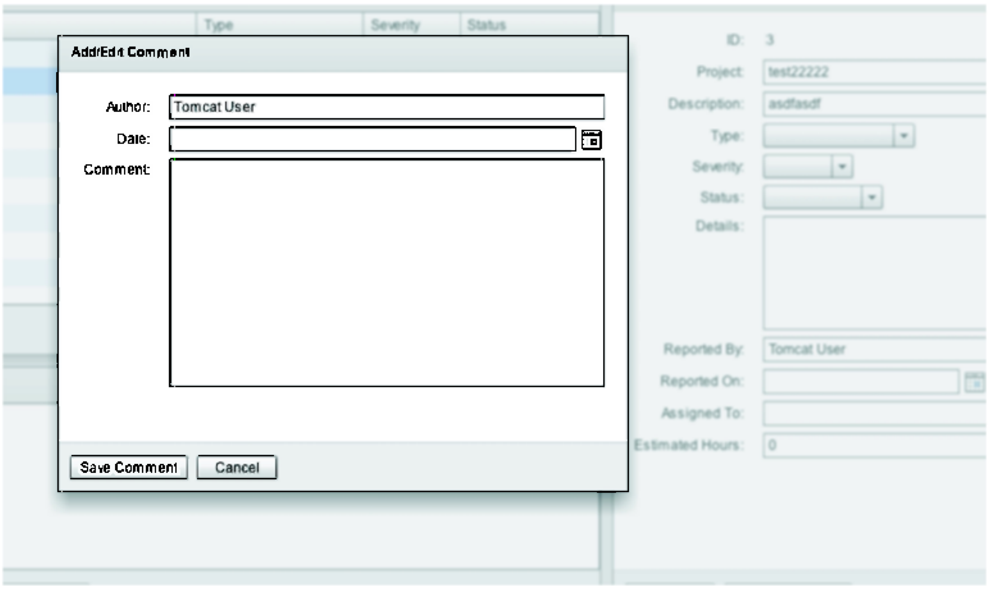
```
package org.foj.presenter {

...

public class DetailPresenter {

  private var _currentUser:User;                          1 Instance variable for current user
  private var _issue:Issue;
  private var _view:DetailView;
  private var _issueModel:IssueModel;

  public function DetailPresenter(view:DetailView) {

...

    EventDispatcherFactory.getEventDispatcher()           2 Event listener for current user changing
        .addEventListener(UserEvent.CURRENT_USER_UPDATED,
        currentUserChanged);
  }

  ...

  private function cancelChanges(event:UIEvent = null):void {
    selectedIssue = new Issue();
    selectedIssue.reportedBy = _currentUser.fullName;     3 Default author field
    var event:UIEvent = new UIEvent(UIEvent.SELECTED_ISSUE_CHANGED);
    event.data = selectedIssue;
    EventDispatcherFactory.getEventDispatcher().dispatchEvent(event);
  }

  ...                                                     Event handler for currentUserChanged 4

  private function currentUserChanged(event:UserEvent):void {
    _currentUser = event.data;
    if (selectedIssue.id == 0) {
      _view.issueReportedBy.text = _currentUser == null ?
                                  "" :_currentUser.fullName;
    }
  }

  ...
}
}
```

First you add an instance variable for the `DetailPresenter` to be able to cache the current user in 1. Next you add an event listener for the `CURRENT_USER_UPDATED` event 2 and point it at the `currentUserChanged` event handler. Inside the `cancel-Changes` event handler, which is invoked each time the user wants to clear out the changes made in the detail form and creates a new issue, you set the default for the `reportedBy` property to the current user's full name 3. Last you implement an event listener for the current user changing ❹, where you set the instance variable of current user to the user that was passed along in the event. Then you update the `reported by` field on the view if the currently displayed issue is not an existlng issue. The following listlng shows the changes for the `CommentsListPresenter`.

Listing 7.17 CommentsListPresenter

```
package org.foj.presenter {

...

public class CommentsListPresenter {                              ❶ Instance variable
                                                                     for current user
  private var _currentUser:User;
  private var _selectedIssue:Issue;
  private var _selectedComment:Comment;
  private var _commentModel:CommentModel;
  private var _view:CommentsView;
  private var _popl:EditCommentForm;

  public function CommentsListPresenter(view:CommentsView = null) {

    ...

    EventDispatcherFactory.getEventDispatcher()                   Event listener for
        .addEventListener(UserEvent.CURRENT_USER_UPDATED,       ❷ user changed
        currentUserChanged);
  }

  ...

  private function addNewComment(event:* = null):void {
    selectedComment = new Comment();
    selectedComment.issue = selectedIssue;

    _popl = PopUpManager.createPopUp((_view as UIComponent).root,
        EditCommentForm, true) as EditCommentForm;
    _popl.author.text = _currentUser.fullName;                    Default
    PopUpManager.centerPopUp(_popl as UIComponent);             ❸ author field
  }

  ...

  private function currentUserChanged(event:UserEvent):void {
    _currentUser = event.data;
  }                                                          Event handler for
                                                             user changed event ❹
  ...

}
}
```

As with the `DetailPresenter`, you start by defining an instance variable to allow this Presenter to cache the currently logged in user ❷. After that, you add an event listener for the `CURRENT_USER_UPDATED` event ❷ pointlng it to the `currentUserChanged` method ❹. where you set the current user to the user passed in with the event. Inside the `addNewComment` method you default the text of the `Author` field in the comment pop up to be the current user's full name ❸.



Add/Edit Comment

Author: Tomcat User

Date:

Comment:

Save Comment Cancel

ID: 3

Project: test22222

Description: asdfasdf

Type:

Severity:

Status:

Details:

Reported By: Tomcat User

Reported On:

Assigned To:

Estimated Hours: 0

Figure 7.4 Add new comment defaults Author to currently logged in user

Now build and run the application. If you log in using the Tomcat User username of `user` and password of `user` and try to add a new comment to an issue, you should see that the `Author` field is now defaulting to the logged in user's full name as shown in figure 7.4. When you log out you should see that the fields are no longer being defaulted to anything.

7.4 Summary

When you started this chapter, you had an application which anybody could have added critical issues to and removed important issues from with no controls to prevent such destructive behavior. Over the course of the chapter you added measures to control access.

You started by adding the simplest of security constraints to the application, allowing the users to authenticate to the server side, although this doesn't do you much good until you can control what the users do to the data. After you added the authorization constraints, you had fine-grained control over who could modify and delete the data. This is important, because before Spring Security was developed, the most common way to declaratively define security constraints this way at the method level was to use EJBs and container-managed security, which would probably have made this tight integration difficult to create.

Even in the fairly simple example you have been able to define three different levels of security for the application with minimal intrusion to the existing web application. The ability to declaratively define security constraints on the business methods by using either the `@Secured` annotations or using AOP pointcut syntax is extremely powerful. This means that you no longer have to litter the source code with `if` statements checking if users are logged in and whether or not they have the right level of access to perform the operation.

The last thing you did in the journey through this chapter was to add functionality that doesn't do much for securing the application, but will likely delight the users for the simple fact that the application is personalized to them. They no longer have to type their names into the author fields when adding new issues and comments, and can see that they are logged in, and whom they are logged in as.

In the next chapter you're going to continue enhancing the application by adding data visualization to put a bird's-eye view on the data. You're going to tackle creating a pie chart component from scratch using the Degrafa framework.

Charting with Degrafa

8

This chapter covers

- Introducing the Degrafa drawing library
- Creating a custom PieChart component
- Creating an ItemRenderer for a DataGrid
- Dynamic object creation

Now that your application is secured and communicating with the server side, it's time to add data visualization components and enable a bird's-eye view on the data. Adding data visualization components to your application allows people to see at a glance the open project issues and the number of bugs versus the number of feature requests, without having to manually count them in the data grid in the master view. The data could be visualized in many ways—we'll only scratch the surface in this chapter.

Adobe provides data visualization components, but only when you purchase a license for the professional version of the Flash Builder IDE. Because our goal is to do Flex development using only free and open source technologies, we've decided to create our own visualization components—besides, it's more fun.

8.1 Drawing in Flex

Flex and Flash provide powerful drawing libraries that we could leverage to create our custom graph components, but we're going to leverage an open source graphics library called Degrafa. Using Degrafa gives us the ability to declaratively build our graphing components rather than having to deal with the complex calculations involved in drawing pie chart slices as illustrated in listing 8.1, which shows an example ActionScript class specifically for drawing a pie chart slice found at http://www.adobe.com/devnet/flash/artlcles/adv_draw_methods.html. Notlce how much trigonometry is involved in creating something as simple as a pie chart slice from scratch.

Listing 8.1 Example of drawing in ActionScript

```
/*-------------------------------------------------------------
    mc.drawWedge is a method for drawing pie shaped
    wedges. Very useful for creating charts. Special
    thanks to: Robert Penner, Eric Mueller and Michael
    Hurwicz for their contributions.
-------------------------------------------------------------*/
MovieClip.prototype.drawWedge = function(x, y, startAngle, arc,
➥radius, yRadius) {
    ==============
    // mc.drawWedge() - by Ric Ewing (ric@formequalsfunction.com) -
    ➥version 1.3 - 6.12.2002
    //
    // x, y = center point of the wedge.
    // startAngle = starting angle in degrees.
    // arc = sweep of the wedge. Negative values draw clockwise.
    // radius = radius of wedge. If [optional] yRadius is defined,
    ➥then radius is the x radius.
    // yRadius = [optional] y radius for wedge.
    // ==============
    // Thanks to: Robert Penner, Eric Mueller and Michael Hurwicz
    ➥for their contributions.
    // ==============
    if (arguments.length<5) {
        return;
    }
    // move to x,y position
    this.moveTo(x, y);
    // if yRadius is undefined, yRadius = radius
    if (yRadius == undefined) {
        yRadius = radius;
    }
    // Init vars
    var segAngle, theta, angle, angleMid, segs, ax, ay, bx, by, cx, cy;
    // limit sweep to reasonable numbers
    if (Math.abs(arc)>360) {
        arc = 360;
    }
    // Flash uses 8 segments per circle, to match that, draw in a maximum
    // of 45 degree segments. First calculate how many segments are needed
    // for our arc.
```

```
        segs = Math.ceil(Math.abs(arc)/45);
        // Now calculate the sweep of each segment.
        segAngle = arc/segs;
        // The math requires radians rather than degrees. To convert from degrees
        // use the formula (degrees/180)*Math.PI to get radians.
           theta = -(segAngle/180)*Math.PI;
        // convert angle startAngle to radians
        angle = -(startAngle/180)*Math.PI;
        // draw the curve in segments no larger than 45 degrees.
        if (segs>0) {
            // draw a line from the center to the start of the curve
            ax = x+Math.cos(startAngle/180*Math.PI)*radius;
            ay = y+Math.sin(-startAngle/180*Math.PI)*yRadius;
            this.lineTo(ax, ay);
            // Loop for drawing curve segments
            for (var i = 0; i<segs; i++) {
                angle += theta;
                angleMid = angle-(theta/2);
                bx = x+Math.cos(angle)*radius;
                by = y+Math.sin(angle)*yRadius;
                cx = x+Math.cos(angleMid)*(radius/Math.cos(theta/2));
                cy = y+Math.sin(angleMid)*(yRadius/Math.cos(theta/2));
                this.curveTo(cx, cy, bx, by);
            }
            // close the wedge by drawing a line to the center
            this.lineTo(x, y);
        }
    };
```

Adobe has released the specifications for its declarative graphics library, called FXG. It appears that the Degrafa team has collaborated with the Adobe team to create this specification, but the FXG functionality is only a subset of what is available from the Degrafa library. This may be a library to keep your eye on as it's being developed.

8.2 Common Degrafa concepts

Before diving into developing the component, let's familiarize ourselves with some of the terms and concepts that we'll see as we work through this example.

- *Surface*—This is the base component for everything you'll do in Degrafa. All other Degrafa components will be composed within a `Surface`.
- *GeometryGroup*—After the `Surface`, this is the next level of composition. The `GeometryGroup` tag allows you to group Degrafa components to compose an object.
- *Stroke*—`Stroke` is the object that is used to define the look of an object's outline, in terms of color, thickness, and style. Degrafa provides different `Stroke` objects for your use depending on the style of stroke you want: `SolidStroke`, `LinearGradient`, and `RadialGradient`.
- *Fill*—`Fill` refers to the appearance of the bounded area of a graphical component. Degrafa provides the following fills: `SolidFill`, `LinearGradient`, `RadialGradient`, `BitmapFill`, `BlendFill`, and `ComplexFill`.

- *Shapes*—Degrafa supports drawing many different shapes out of the box, such as `Circle`, `Ellipse`, `RegularRectangle`, `RoundedRectangle`, `Polygon`, and more. For irregular shapes, Degrafa also has an extensive library of auto shapes and enables defining any shape you'd like by providing a Scalable Vector Graphics (SVG) path.
- *Repeaters*—This gives you the ability to repeat a shape any number of times on the surface.

Much like other Flex components, the Degrafa components are considered either container components, meaning they will contain other Degrafa components, or graphical elements. Figure 8.1 shows the relationship of the common components.

More than a single chapter would be needed to cover all the features that Degrafa offers, especially when it comes to skinning and the advanced CSS functions you can accomplish with this powerful framework. We're only going to scratch the surface; to learn more about Degrafa, you can start with the Foundation section of the documentation at http://www.degrafa.org/samples/foundation.html.

8.3 Creating a pie chart for fun and profit

Now that we have some of the basic concepts, let's get on with the task of creating a custom pie chart component. We were inspired by a blog posting by Derrick Grigg titled appropriately enough *Degrafa Pie Chart*, which can be found at http://www.dgrigg.com/post.cfm/04/15/2008/Degrafa-Pie-Chart. After we decomposed it and removed some of the extra visual effects such as tweening and gradients, it barely

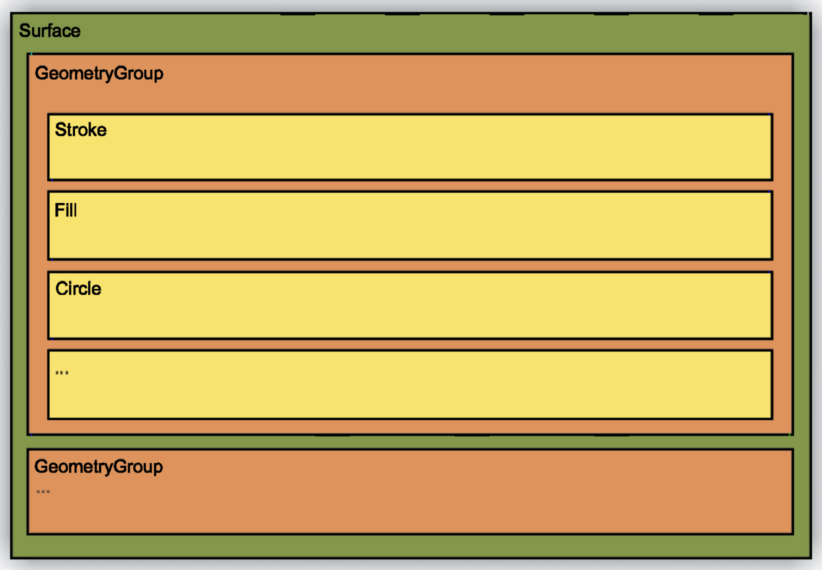
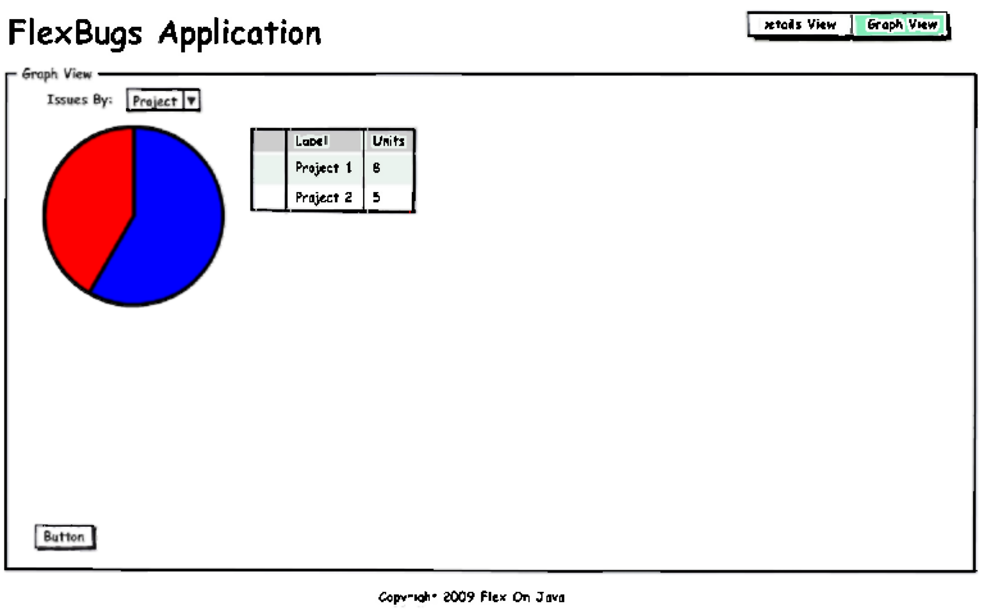


Figure 8.1 Relationship of Degrafa components



	Label	Units
	Project 1	8
	Project 2	5

Figure 8.2 Mock-up of the Graph View

resembles what we started with. Figure 8.2 shows a mock-up of the chart we'll be developing in this chapter.

Recall in chapter 2 you created a separate view to contain your data visualizatIon. For this example you'll be developing only a single pie chart component, but some of the concepts illustrated here could potentially be applied to creatIng any number of chartIng components.

The component you'll develop is a combinatIon of a pie chart and a data grid, which will serve the purpose of a legend for the pie chart. Without this it may be difficult for someone looking at the chart to differentIate between data points on the graph. The pie chart will consist of the pie chart itself and another component for each of the slices that make up the chart. You'll also develop a simple custom `ItemRenderer` for the chart legend to draw a simple box inside one of the cells in the data grid.

You'll also be adding a label and a combo box to the `GraphView` to allow the user to change the data the chart shows. By changing the value of the combo box the user can show how many issues there are by project, type, status, or severity.

8.3.1 New custom event

We're going to create a new custom event for our pie chart. The reason we're creatIng a new one rather than continuing to use the `UIEvent` we created earlier is that if we ever wanted to put more than one pie chart component into our applicatIon, we'd need to be able to distInguish which component fired the event.

Listing 8.2 PieChartEvent.as

```
package org.foj.event {
import flash.events.Event;

public class PieChartEvent extends Event{

  public static const DATA_PROVIDER_UPDATED:String =
  "dataProviderUpdated";

  public var data:*;
  public var id:*;                                          ❶ id property

  public function PieChartEvent(type : String,
                        bubbles : Boolean = true,
                        cancelable : Boolean = false)
  {
    super(type, bubbles, cancelable);
  }
}
}
```

This event differs from the one created previously in the addition of an `id` ❶ property. This is done so that the presenter can decide whether or not it needs to react to the event. With the new event created, you can move on to creating the component itself.

8.3.2 *PieChart component*

First you'll develop the view that contains the pie chart and legend. You'll create these view components in a new package, so create a file named PieChart.mxml in the `org.foj.components` package of your project. The following listing shows the first part of the code for the `PieChart` view.

Listing 8.3 PieChart.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"
         xmlns:degrafa="http://www.degrafa.com/2007"        ❶ CreationComplete
         creationComplete="init()">                           handler

  <s:layout>
    <s:HorizontalLayout/>                                   ❷ HorizontalLayout
  </s:layout>

  <fx:Script>
  <![CDATA[
    import mx.collections.ICollectionView;
    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.PieChartEvent;
    import org.foj.presenter.PieChartPresenter;             ❸ Define presenter
                                                              field
    private var _presenter:PieChartPresenter;                 ❹ Define data
    private var _dataProvider:ICollectionView;                  provider field
    private function init():void
    {
      _presenter = new PieChartPresenter(this, id);         <-┐ ❸ Pass component's
    }                                                            id to presenter

    public function set dataProvider(dataProvider:ICollectionView):void { <-┐
      var refreshEvent:PieChartEvent =                                      │
          new PieChartEvent(PieChartEvent.DATA_PROVIDER_UPDATED);           │
      refreshEvent.id = id;                                 Set property for │
      refreshEvent.data = dataProvider;                        data provider ❺

     EventDispatcherFactory.getEventDispatcher().dispatchEvent(refreshEvent);
    }
  ]]>
  </fx:Script>

...

</s:Group>
```

The code is similar to what you developed in chapter 3 when you created all the MVP components. You set the `creationComplete` event ❸ to call the `init` method. Next you set the layout of your component to use `HorizontalLayout` ❸. Then you declare a couple of private member variables for the data provider and its presenter ❸, ❸. Inside the `init` method you bootstrap your presenter ❸. Last you create a set property ❸ for the data provider where you create an event to notlfy the presenter that the data provider was updated. Listing 8.4 shows the rest of your pie chart component.

Listing 8.4 PieChart.mxml (continued)

```
<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"
         xmlns:degrafa="http://www.degrafa.com/2007"
         creationComplete="init()">
...
  </fx:Script>                                          ❶ Spacer to help lay out
                                                        <-┘ the component
  <mx:Spacer width="10"/>

  <degrafa:Surface id="pieSurface"
                   width="200" height="200">              <-❸ Degrafa surface

    <degrafa:GeometryGroup id="pieGroup">                 <-┐ Geometry group
                                                            ❸ containing pie chart
      <degrafa:filters>                                 <-┐
        <mx:DropShadowFilter color="0x000000"             │
                             alpha="0.5"/>                │ Drop shadow
      </degrafa:filters>                                  ❹ for pie chart

    </degrafa:GeometryGroup>

  </degrafa:Surface>
  <mx:DataGrid id="legendDataGrid">               <-┐ Legend for
    <mx:columns>                                    ❺ pie chart
```

```
      <mx:DataGridColumn width="40"
                         sortable="false"                    ❻ Custom ItemRenderer
                      itemRenderer="org.foj.components.PieLegendRenderer"/>
      <mx:DataGridColumn dataField="label"
                         headerText="Label"/>
      <mx:DataGridColumn dataField="units"
                         headerText="Units"/>
    </mx:columns>
  </mx:DataGrid>

</s:Group>
```

First you added a spacer ❶ to the component to put a bit of padding between your pie chart and its surrounding components. Next you added a Degrafa `Surface` component ❷ and a `GeometryGroup` ❸ to hold the rest of the Degrafa components necessary for the pie chart component. The `GeometryGroup` is the component to which you'll add your pie chart slices when you create them. You've also added a `DropShadowFilter` ❹ to the `GeometryGroup` to add a bit of visual flair to the pie chart. Last you defined the `DataGrid` component ❺ for your chart legend, with a custom `ItemRenderer` ❻ to display the color that corresponds to the data in the chart, which you'll create in a bit.

8.3.3 PieChartSlice

Now that we've defined the pie chart component, let's move on to defining the slices that will make up the pie chart. The pie chart slice is a rather simple component. We probably could have created the pie chart slice programmatically in ActionScript, however this approach allows us to define sensible defaults declaratively in MXML, adding behavior as well.

Listing 8.5 PieChartSlice.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<degrafa:GeometryGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
                       xmlns:degrafa="http://www.degrafa.com/2007"
                       height="400"
                       width="400">                          ❶ Extends GeometryGroup
  <fx:Script>
    <![CDATA[

    public function refresh():void                          ❷ Your pie chart slice
    {
      this.graphics.clear();
      this.arc.preDraw();
      this.arc.draw(graphics, null);
    }

    ]]>
</fx:Script>

  <degrafa:EllipticalArc                                     ❸ Adds EllipticalArc
      id="arc"
      width="200"
      height="200"
      closureType="pie"/>
</degrafa:GeometryGroup>
```

The pie chart slice will extend from `GeometryGroup` ❶ instead of the standard Flex `Group` component that you extended in chapter 2. Next you define a `refresh` method ❷ to abstract behavior away from your presenter. Last you add an `EllipticalArc` component to the component ❸ and set default values such as its `width`, `height`, and most importantly `closureType` property, which you set to *pie*.

8.3.4 Custom ItemRenderer

The next component you're going to create is the custom `ItemRenderer` for the pie chart legend. This simple component will draw a colored box in the data grid cell to correspond with the colors of the pie chart.

Listing 8.6 PieLegendRenderer.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:HBox xmlns:fx="http://ns.adobe.com/mxml/2009"
         xmlns:s="library://ns.adobe.com/flex/spark"
         xmlns:mx="library://ns.adobe.com/flex/halo"                ❶ Extends HBox
         xmlns:degrafa="http://www.degrafa.com/2007">

  <mx:Spacer width="2"/>                                            ❷ Spacer to
  <degrafa:Surface>                                                   help align box
    <degrafa:GeometryGroup>
      <degrafa:fill>                                                 ❸ SolidFill for
        <degrafa:SolidFill id="fill">                                  rectangle
          <degrafa:color>{data.legend}"</degrafa:color>
        </degrafa:SolidFill>                                          ❹ Color of
      </degrafa:fill>                                                   legend item
      <degrafa:RegularRectangle                        ❺ Rectangle
          width="20"
          height="20"
          fill="{fill}"/>
    </degrafa:GeometryGroup>
  </degrafa:Surface>

</mx:HBox>
```

The `ItemRenderer` is extending `HBox` ❶ because all `ItemRenderer` objects for the `DataGrid` component must be halo components. Add a `Spacer` component ❷ to help align the rectangle the way you want it. The `SolidFill` component ❸ defines the fill color for the `RegularRectangle` ❺. Now that you've finished creating all the visual components for the pie chart, let's move on to creatlng the Presenter. An implicitly defined variable is available to the `ItemRenderer` named `data`, which corresponds to the item in the `dataProvider` that you're rendering. Use this implicit variable to set the color of the `Fill` object ❸, which is contained in the legend property of the object.

8.3.5 Presenter for the PieChart

Now that all the visual components are created for the pie chart, let's create the Presenter. The Presenter for the pie chart becomes more involved than any of your previous ones, but it shouldn't be hard to follow.

Listing 8.7 PieChartPresenter.as

```
package org.foj.presenter {

import com.degrafa.paint.SolidFill;
import org.foj.components.PieChart;
import mx.collections.ArrayCollection;
import org.foj.components.PieChartSlice;
import org.foj.event.EventDispatcherFactory;
import org.foj.event.PieChartEvent;
import org.foj.model.PieChartModel;

public class PieChartPresenter {

  private var _view:PieChart;
  private var _model:PieChartModel;
  private var _id:String;
  private var _dataProvider:ArrayCollection;

  public function PieChartPresenter(view:PieChart, id:String) {
    this._view = view;
    this._model = new PieChartModel();
    this._id = id;

    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(PieChartEvent.DATA_PROVIDER_UPDATED,
        refreshData);
  }

  public function set dataProvider(data:ArrayCollection):void {
    _view.legendDataGrid.dataProvider = data;
    this._dataProvider = data;
  }

  public function get dataProvider():ArrayCollection {
    return this._dataProvider;
  }

  private function refreshData(event:PieChartEvent = null):void {
    if (event.id == _id) {
      changeData(event.data);
    }
  }

  private function changeData(data:ArrayCollection):void {
    dataProvider = data;
    createSlices();
  }

  private function createSlices():void {
    while (dataProvider.length > _view.pieGroup.numChildren) {
```

1 Private member variables

2 Constructor

3 Set property for dataProvider

4 Event handler

5 Create slices method

```
      _view.pieGroup.addChild(new PieChartSlice());
    }
    setLegendColors();
    redrawSlices();
  }
                                                                         ❻ Set legend
  private function setLegendColors():void {                          <┘    colors on data
    for (var i:int = 0; i < dataProvider.length; i++)
    {
      dataProvider.getItemAt(i).legend = _model.getLegendColorForIndex(i);
    }
  }
                                                                          ❼ Redraw slices
  private function redrawSlices():void {                             <┘     after update
    var currentAngle:Number = 0;
    var totalUnits:Number = _model.getTotalUnits(_dataProvider);

    for (var i:int = 0; i < _view.pieGroup.numChildren; i++) {
      var slice:PieChartSlice = _view.pieGroup.getChildAt(i)
as PieChartSlice;
      var legendColor:Number = _model.getLegendColorForIndex(i);
      var arc:Number = i < dataProvider.length ?
                       _model.getAngleForItem(
dataProvider[i].units, totalUnits) : 0;

      // workaround for weird display if only one arc and it's 360 degrees
      arc = arc < 360 ? arc : 359.99;                                  <┐

      redrawSlice(slice, currentAngle, arc, legendColor);               ❽ Workaround
      currentAngle += arc;
    }
    _view.pieGroup.draw(null, null);
  }

  private function redrawSlice(slice:PieChartSlice,                    <┐
                               startAngle:Number,                       ❾ Redraw slice
                               arc:Number,
                               color:Number):void {
    slice.arc.fill = new SolidFill(color, 1);
    slice.arc.startAngle = startAngle;
    slice.arc.arc = arc;
    slice.refresh();
  }

}
}
```

You first define private member variables to hold onto references to the `view` and the `model`, the `id` of the component this Presenter belongs to, and the `dataProvider` for your pie chart ❶. The constructor ❷ for this Presenter not only takes in a reference to the `view`, but also is used to bootstrap the `id` for the `view` component because there may be multlple pie charts contained in the application. You also define an event listener for the `dataProvider` being updated in the view component. Next you define a pair of get and set propertles ❸ for the `dataProvider` you leverage to update the `dataProvider` property of the legend data grid whenever the `data-Provider` for the pie chart is updated.

You then define the event handler method for the event that is fired whenever the `dataProvider` for the view is updated ❹. Inside this method you check to see if the `id` of the component firing the event is the same as the `id` that created this Presenter. That way if there are multiple pie chart components, this method can determine whether or not it needs to react.

The `createSlices` ❺ method checks to see if the data provider has more elements contained in it than there are pie chart slices in your pie chart. If there are more elements in the data provider, it will create more pie chart slices. In the `set-LegendColors` ❻ method you iterate through the items in the `dataProvider` and set the `legend` property of the item to the corresponding color, which you'll get from the pie chart model class.

After all of that, refresh your pie chart with a call to the `redrawSlices` ❼ method. This will iterate over the pie chart slices and update the data values, such as the start angle of the slice and its arc. You iterate over the pie chart slices instead of the data provider because there may be more slices than items in the `dataProvider`, and this will draw the extra slices with an arc of 0. There is also a little workaround ❺ for when there is only a single slice and its arc is 360, which sets its arc to 359.99 so that it would draw correctly. After all of the data for the slice is updated, it is passed into the `redrawSlice` ❾ method to tell the slice to redraw itself.

8.3.6 Model for the PieChart

Now you only have one piece of the MVP triad to complete for your pie chart. Even though the pie chart doesn't need to call out to any remote services, you've stlll refactored a couple of methods that could be considered business logic rather than presentatlon logic and have no need to maintain any kind of state. The following listing shows the code for the pie chart model.

Listing 8.8 PieChartModel.as

```
package org.foj.model {
import mx.collections.ICollectionView;

public class PieChartModel {

  private var colors:Array = [                  <┐ Array of colors
    0x468966,                                    ❶ for legend
    0xFFB03B,
    0xFFF0A5,
    0x999574,
    0x007D9F,
    0x8E2800,
    0x8E28F4,
    0x0528F4,
    0xF42105,
    0x0CF405
  ];
  public function getLegendColorForIndex(index:Number):Number {   <-
    return colors[index];                                Convenience method
  }                                                       for getting color ❷

  public function getAngleForItem(units:Number,
    totalUnits:Number):Number {                          <-  Calculate angle
    return ((((units / totalUnits) * 100) * 360) / 100) ;   ❸ for item
  }

  public function getTotalUnits(dataProvider:ICollectionView):Number {   <-
    var total:Number = 0;
                                                     Calculate total number
    for each(var item:Object in dataProvider) {        of items for pie chart ❹
      total += item.units;
    }

    return total;
  }
}
}
```

An array of 10 different hex values ❶ corresponds to the colors you want the pie chart to use for its data points. This number could easily be increased should the need arise for more data points in your graphs; for this example this number should suffice. Next you define a convenience method ❷ for getting the color value for a specific index. The method `getAngleForItem` ❸ takes care of the calculation for determining the size of the angle for an item based on the total number of items contained within the pie chart and the number of items passed in. The last method you define ❹ in your model iterates through the data set passed in and returns back the total number of items for the pie chart.

8.4 Adding your pie chart to the application

That takes care of all of the pieces you need for the pie chart component. Next you're going to need to make changes to the `GraphView` components in order to support it. For the example you'll add only a single instance of this pie chart that you'll be able to change the data it's visualizing with a combo box. You could just as easily display each of these visualizations for the data separately. We chose this approach because it's an easy way to illustrate the event handling working for the pie chart components because the data will be updated for the chart each time you select a different reporting point from the combo box.

8.4.1 Updating the GraphView

To add the pie chart to the application, you'll first update the `GraphView`. In chapter 2 you created the `GraphView` with simple placeholder text; now is the time to implement this view. The following listing shows the updated `GraphView` component after you made the changes to add the pie chart.

Listing 8.9 GraphView.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Panel title="Graph View"
         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:components="org.foj.components.*"            ❶ Components namespace declaration
         width="100%"
         height="100%"
         creationComplete="init()">

  <fx:Script>
  <![CDATA[
    import org.foj.event.EventDispatcherFactory;
    import org.foj.event.UIEvent;
    import org.foj.presenter.GraphPresenter;

    private var _presenter:GraphPresenter;

    private function init():void {                            ❷ Bootstrap presenter
      _presenter = new GraphPresenter(this);
      refreshData();
    }

    private function refreshData():void {                     ❸ Refresh button event handler
      var refreshEvent:UIEvent = new UIEvent(UIEvent.REFRESH_GRAPHS);
      refreshEvent.data = groupByComboBox.value;
      EventDispatcherFactory.getEventDispatcher().
      ➥dispatchEvent(refreshEvent);
    }

    private function changeGroupBy(event:Event):void {        ❹ Group by combo box event handler
      var changeEvent:UIEvent = new UIEvent(UIEvent.REFRESH_GRAPHS);
      changeEvent.data = groupByComboBox.value;
      EventDispatcherFactory.getEventDispatcher().dispatchEvent(changeEvent);
    }

  ]]>
  </fx:Script>

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
                                                              Group by combo box ❺
  <mx:HBox>
    <mx:Text text="Issues By: " fontWeight="bold" fontSize="16"/>
    <mx:ComboBox id="groupByComboBox" change="changeGroupBy(event)"/>
  </mx:HBox>

  <components:PieChart id="issuesPieChart"/>                  ❻ Pie chart

  <mx:Spacer height="100%"/>

  <mx:ControlBar>
    <mx:Button id="refreshButton"                             ❼ Refresh button
               label="Refresh Data"
               click="refreshData()"/>
  </mx:ControlBar>

</s:Panel>
```

By now this should be almost second nature. You start by declaring the namespace for your pie chart components ❶, and bootstrap the presenter in the init method ❷. Next define a couple of event handler methods for use when the refresh button is clicked ❸ and when the group by combo box is updated ❹. Add an HBox to contain a label component as well as the ComboBox ❺ to switch data field by which the data is grouped in your pie chart ❻. Last, add a button ❼ to trigger a refresh on the data being displayed in the pie chart.

8.4.2 *Creating a Presenter for the GraphView*

After you've updated the view component for the GraphView, you need to create the Presenter to support it.

Listing 8.10 GraphPresenter.as

```
package org.foj.presenter {
import mx.collections.ArrayCollection;
import mx.controls.Alert;
import mx.managers.CursorManager;
import mx.rpc.AsyncResponder;
import mx.rpc.AsyncToken;
import mx.rpc.events.FaultEvent;
import mx.rpc.events.ResultEvent;
import org.foj.dto.Issue;
import org.foj.event.EventDispatcherFactory;
import org.foj.event.UIEvent;
import org.foj.model.IssueModel;
import org.foj.model.GraphModel;
import org.foj.view.GraphView;

public class GraphPresenter {

  private var _view:GraphView;
  private var _issueModel:IssueModel;
  private var _model:GraphModel;

  private var _groupByValues:ArrayCollection =          ❶ ArrayCollection for
      new mx.collections.ArrayCollection([                 group by combo box
        { label: "Project", data: "project" },
        { label: "Status", data: "status" },
        { label: "Type", data: "type" },
        { label: "Severity", data: "severity" }
      ]);

  public function GraphPresenter(view:GraphView) {      ❷ Constructor
    this._view = view;
    this._issueModel = new IssueModel();                  EventListener for ❸
    this._model = new GraphModel();                      refresh button and
                                                          combo box events
    EventDispatcherFactory.getEventDispatcher()
        .addEventListener(UIEvent.REFRESH_GRAPHS, refreshGraphs);

    _view.groupByComboBox.dataProvider = _groupByValues;
  }                                                       Set dataProvider for
                                                          group by combo box ❹
  private function refreshGraphs(event:UIEvent = null):void {   <-┐
    CursorManager.setBusyCursor();                                 │  Event
    _issueModel.getIssues(new AsyncResponder(getIssuesResult,      │  handler for
                                             handleError));      ❸  refresh event
  }

  private function getIssuesResult(event:ResultEvent,                      <-┐
                                   token:AsyncToken = null):void {           │
    CursorManager.removeBusyCursor();                                        │
    var issues:ArrayCollection = new ArrayCollection();      Event handler   │
                                                             for result from │
    for each(var item:Object in event.result) {                 issue model ❻
      var issue:Issue = new Issue(item);
      issues.addItem(issue);
    }

    _view.issuesPieChart.dataProvider =
        _model.groupCollectionBy(issues,
            _view.groupByComboBox.selectedItem.data);
  }                                                          ❼ Event handler
                                                             │ for errors
  private function handleError(event:FaultEvent,          <-┘
                               token:AsyncToken = null):void {
    CursorManager.removeBusyCursor();
    Alert.show(event.message.toString());
  }

}
}
```

You start by defining an `ArrayCollection` of items for the combo box ❸. This is an example of the power of a dynamic language such as ActionScript. Those familiar with JavaScript may recognize this as JSON notatlon. ActlonScript allows you to create objects like this without any type of `Class` declaratlon, which is great for these one off throwaway objects.

Next you define the constructor ❸, which takes as an argument the `view` component that belongs to this Presenter. Inside this constructor you instantiate an issue model which you use to retrieve the issues from the server side, as well as an instance of the `GraphModel` that you'll create in a moment. You then add an event listener for the `REFRESH_GRAPHS` event ❸ and map it to your handler. The last thing your constructor will do is set the `dataProvider` property of the `groupByComboBox` ❸ so that it has the appropriate display values and data values to functlon correctly.

> **NOTE** The two propertles you define for the objects being passed to your combo box have special meaning to this component. The `label` property is what will be displayed to the user in the combo box. The second property, `data`, will be returned when using the `value` property for the combo box, which is similar to how HTML drop-down boxes work.

You next define an event handler for when the `REFRESH_GRAPHS` event is triggered ❸. Inside this handler you make a call to the issue model to fetch the list of issues in the system. In the result event handler for this call ❻ you make a call to the graph model to aggregate the data and set the pie chart's `dataProvider` property to the result from this call. Last you define a generic error handler as you have in the other Presenters ❼.

8.4.3 *Creating the graph model*

Now the only thing left to do is implement your `GraphModel`. This model will contain only one method that is used to aggregate and format the list of issues retrieved from the issue model into a format that your pie chart can understand.

Listing 8.11 GraphModel

```
package org.foj.model {

import mx.collections.ArrayCollection;

public class GraphModel {
  public function GraphModel() {
  }

  public function groupCollectionBy(allIssues:ArrayCollection,
                                    field:String):ArrayCollection {
    var group:Array = new Array();                                        ❶

    for each(var issue:Object in allIssues) {                             ❷
      if (group[issue[field]] == null) {
        group[issue[field]] = 1;
      } else {
        group[issue[field]]++;
      }
    }

    var result:ArrayCollection = new ArrayCollection();                   ❸
    for (var key:String in group) {                                       ❹
      result.addItem({label: key, units: group[key]});
    }

    return result;
  }
}
}
```

❶ **Array to hold intermediate groupings**

❷ **Iterate over items and create associative array**

❸ **ArrayCollection for results**

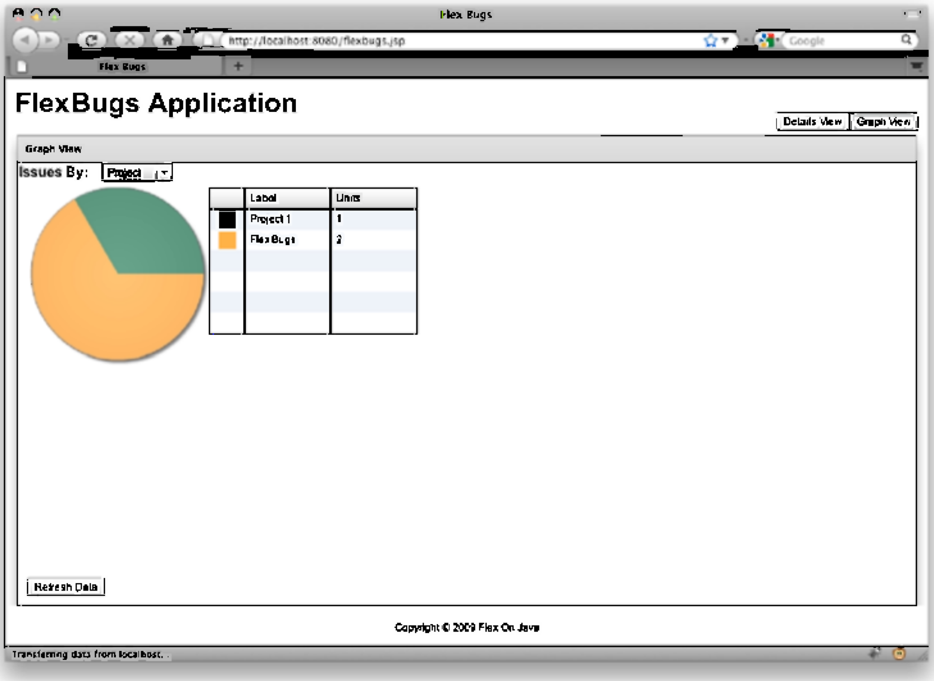
❹ **Iterate over intermediate results and build real results**

The implementation for this method is fairly simplistic. First you create an array ❶ that will be used to help group and count the items passed in by the property name in the field variable. Next you iterate over the collection of issues ❷. If the array does not contain an entry already with the field name, add it and set its value to 1. Otherwise increment the value for that field name. Now create an `ArrayCollection` that will contain the real results that you need to return from this method ❸. Iterate over the intermediate results ❹ creating a dynamic object containing the two properties that your pie chart needs to properly display the data: `label` and `units`. Last you return the result `ArrayCollection`.

NOTE Concerning dynamically adding propertles, recall that in your `PieChartPresenter` you are setting a property called `legend` on the results from this functlon call although you never define the objects being returned as having a `legend` property. This is another example of how powerful, and sometlmes dangerous, a dynamic language such as ActlonScript can be. Because you never defined a property named `legend`, ActionScript interpreted this to mean add a property to this object named `legend`, which it did, and happily continued on its merry way. If you fat fingered that property name somewhere else in your code it would make for a difficult bug to track down.

With all of the components implemented, build and deploy the RIA module to the local repository by issuing an `mvn install` command on the command line within this module's root folder. Then change directories to the web module and run the command `mvn jetty:run-war` to start up the Jetty container so you can see your hard work in action.

Keep in mind that you may want to add the `-Dmaven.test.skip=true` optlon to that command so Maven won't blow away any issues that you had entered into the database. After Jetty starts up, if you don't have any issues created in the applicatlon, add a few, then click the Graph View button in the upper-right corner and you should see something similar to figure 8.3.



	Label	Units
■	Project 1	1
🟧	FlexBugs	2

Figure 8.3 The finished GraphView

Next change the selected value in the combo box at the top of the page and watch your graph and legend update to display the data grouped by a different data point. If you click back to the `DetailsView` and add or remove issues you can come back to this view and click the Refresh Data button at the bottom of the screen and it should update your pie chart with the new data from the database.

8.5 Beyond the example

When it comes down to it, there isn't much that you couldn't visualize using Degrafa and some ingenuity and imagination. There are great examples of ways to visualize your data at the Degrafa site at http://www.degrafa.org/samples/data-visualization.html—everything from bar charts, to gauges, financial data, Gantt charts, even combining visualization with maps. There was even a recent presentation at 360|Flex Indy (http://www.flexjunk.com/2009/05/30/developing-a-smith-chart-using-axiis-and-degrafa/) on creating a Smith Chart using a data visualization framework called Axiis which is built on Degrafa. Figure 8.4 shows more examples of data visualization components created with Degrafa.

The Degrafa possibilities are endless; in fact someone has even created a Growl like component for use in Flex applications (http://lukesh.wordpress.com/2009/04/04/rawr-flexgrowl-component-available/). Figure 8.5 shows an example of this component.

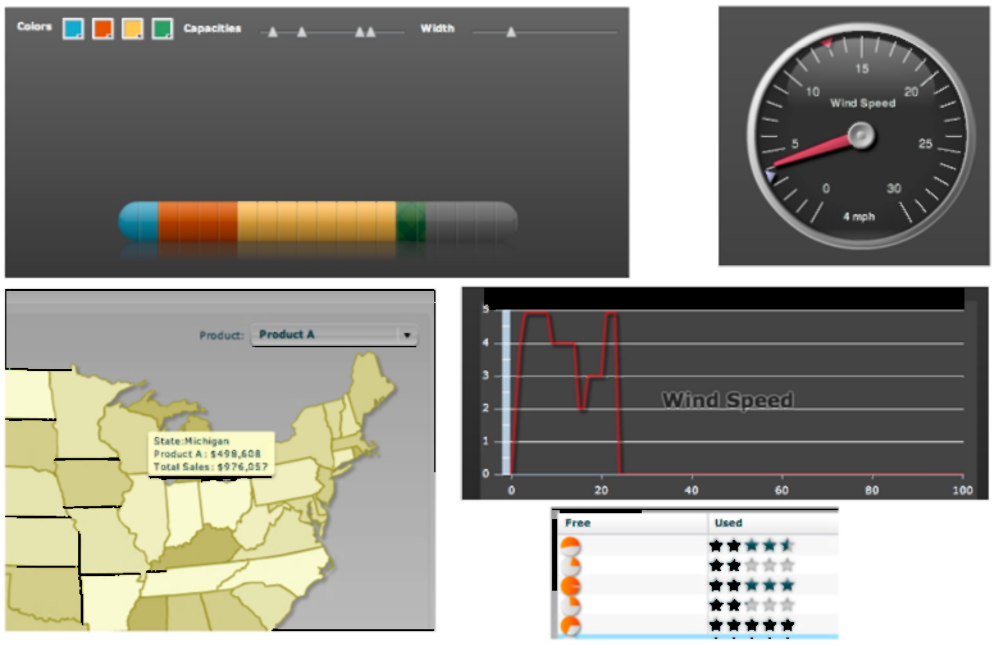
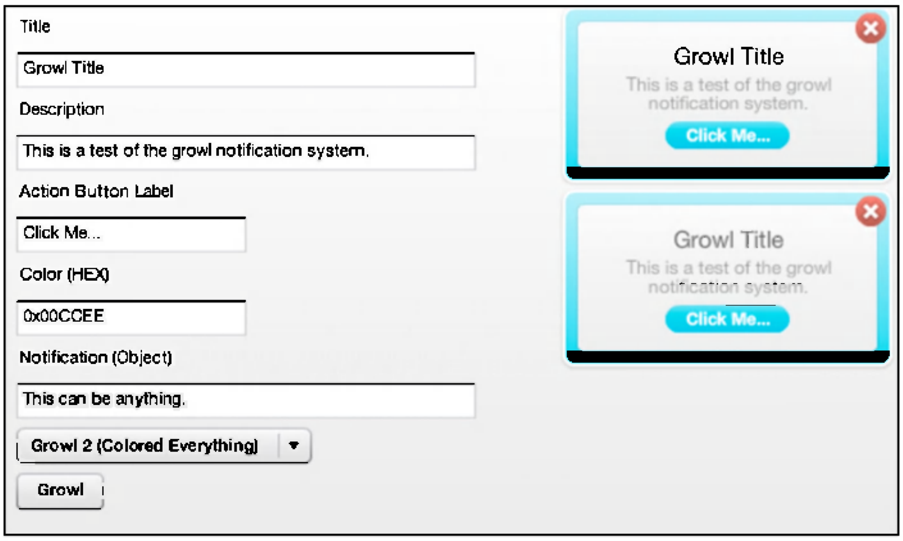


Figure 8.4 More examples of visualization components created with Degrafa



Title

Growl Title

Description

This is a test of the growl notification system.

Action Button Label

Click Me...

Color (HEX)

0x00CCEE

Notification (Object)

This can be anything.

Growl 2 (Colored Everything)

Growl

Growl Title

This is a test of the growl notification system.

Click Me...

Growl Title

This is a test of the growl notification system.

Click Me...

Figure 8.5 A Growl like component for Flex

You can do amazing things with Degrafa, whether skinning your application and component to look like an iPhone application, or creating a rich user experience such as Autodesk's Project Dragonfly shown in figure 8.6.

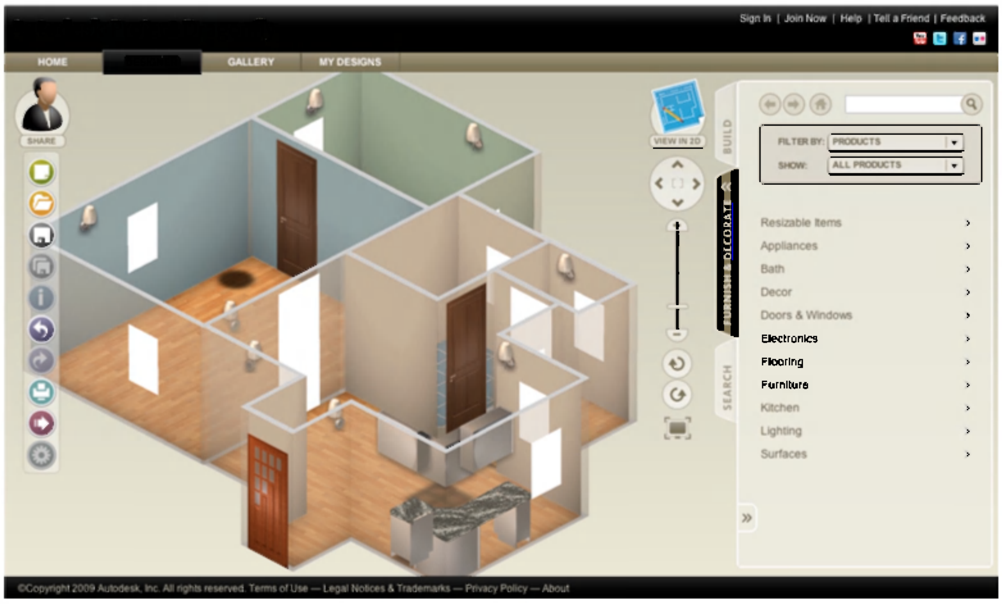


Figure 8.6 Autodesk's Project Dragonfly interactive home design software

The Autodesk example, which can be found at http://www.homestyler.com/, is one of our favorite examples of what is possible with using the Degrafa library. In the Project Dragonfly app, you can do anything from defining your floor plan to moving furniture, doors, countertops, and other features interactively in either a 2D top-down view or a full 3D view that can be zoomed and rotated. This is a great example of an immersive rich internet experience; the kind of functionality you would never expect to be available from a web application. You're limited only by your imagination.

8.6 Summary

In this chapter we've introduced the Degrafa framework and created a custom pie chart component. We've also introduced creating a custom `ItemRenderer` for displaying something other than simple text inside of a `DataGrid` cell. And if that weren't enough we also demonstrated some of the dynamic features of the ActionScript language by creating ad hoc objects on the fly and adding properties to those objects at runtime.

We've also built this component in such a way that you could now easily refactor it into a separate reusable library. It wouldn't take much effort to extract the components you created in this chapter into its own module and add them as a dependency to any project.

Desktop 2.0 with AIR

9

This chapter covers

- Creating a common SWC library
- Making an AIR version of a Flex application
- Creating a key for signing the application
- Packaging the application for distribution
- Distributing the AIR application via the web

Although your Flex application is rich and engaging, some power users will want a version that they can download and install locally as a desktop application. Adobe AIR allows you to easily accomplish this otherwise daunting task. RIAs blur the line between traditional web applications and desktop client applications, especially when you can convert the Flex application to a desktop application running in the Adobe AIR runtime by changing only a couple of lines of code. Yes, you heard that right—only a couple of lines. What other framework allows you to reuse this much of the client code, going from a purely web-based experience to a desktop application?

You do want to be able to use the Flex version of your application, so you're going to refactor all the code that will be common between the Flex application and the AIR application that you're going to build into a separate Shockwave

Component (SWC) project. Then you can declare a dependency on this SWC library in both the AIR project and the Flex project.

After you have the common library project created, create a new Maven project for your AIR application, which will in turn create an SWF file that you'll use to package and deploy your AIR application. Then you'll have to create one more project that will contain all of the resources necessary to package your AIR application, such as the certificate for signing the application and any assets that the application will need, such as the icons the OS will use to display the application.

Last, we'll look at what it takes to distribute the application via a web page and how to let users update the version of their application quickly and painlessly. Let's get on with the show.

9.1 Creating a common library

Because the only part of your application that will differ between the Flex version and the AIR version will be the main MXML file, you can refactor all the common classes out into a separate library. That way you can avoid duplicating classes across the two projects.

You're also going to extract the main contents of the old Main.mxml into a view component so that the new Main.mxml contains only this new view component. This will allow you to change the application structure and content in a single place and enable you to build both the Flex version of the application and the AIR version without having to change either of the Main.mxml files in the applications. Figure 9.1

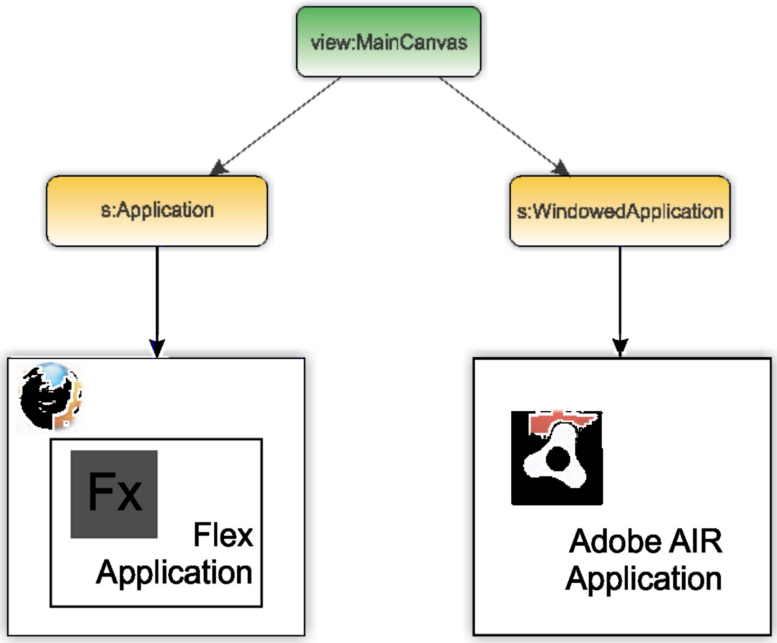


Figure 9.1 High-level view of desired architecture

shows at a high level how the Flex application and AIR application relate to the view component you're going to extract. You first need to create a new project to contain the common elements of the application.

9.1.1 Creating an SWC project

To create your SWC project, you'll use the same archetype that you used to create your Flex application in chapter 2. Open a command line and change the current directory to the root of the FlexBugs project hierarchy, and type the following command.

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

This will create a new folder in the FlexBugs directory named flex-bugs-lib, which contains the new project.

Listing 9.1 `flex-bug-lib` pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
➥http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns=
➥"http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>flex-bugs</artifactId>
    <groupId>org.foj</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>org.foj</groupId>
  <artifactId>flex-bugs-lib</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>swc</packaging>
  <name>FlexBugs common library</name>

  <build>
    <sourceDirectory>src/main/flex</sourceDirectory>
    <testSourceDirectory>src/test/flex</testSourceDirectory>
    <finalName>flex-bugs-lib</finalName>
    <plugins>
      <plugin>
        <groupId>org.sonatype.flexmojos</groupId>
        <artifactId>flexmojos-maven-plugin</artifactId>
        <version>${flexmojos.version}</version>
        <extensions>true</extensions>
        <dependencies>
          <dependency>
            <groupId>com.adobe.flex</groupId>
            <artifactId>compiler</artifactId>
            <version>4.0.0.7219</version>
            <type>pom</type>
          </dependency>
        </dependencies>
        <configuration>
          <targetPlayer>10.0.0</targetPlayer>
          <locales>
            <locale>en_US</locale>
          </locales>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>flexmojos-repository</id>
      <url>http://repository.sonatype.org/content/groups/public/</url>
    </repository>
    <repository>
      <id>flexonjava-repository</id>
      <url>http://flexonjava.googlecode.com/svn/repository</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>flexmojos-repository</id>
      <url>http://repository.sonatype.org/content/groups/public/</url>
    </pluginRepository>
  </pluginRepositories>
  <dependencies>
    <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>
      <type>swc</type>
      <classifier>10</classifier>
    </dependency>
    <dependency>
      <groupId>org.degrafa</groupId>                        ❶ Degrafa
      <artifactId>degrafa</artifactId>                        dependency
      <version>Beta3.1</version>
      <type>swc</type>
    </dependency>
  </dependencies>
  <properties>
    <flexmojos.version>3.2.0</flexmojos.version>
  </properties>
</project>
```

By using the archetype, the only modificatlon you need to make to the pom.xml is to add the dependency on Degrafa ❶. Everything else is configured for you automatically by the archetype.

9.1.2 Extracting common classes

Now that the project has been created, let's look at which classes need to be moved. As we stated at the beginning of this chapter, the only file that is going to be different between your Flex application and your AIR version of the applicatlon is the Main.mxml application file at the root of your source directory. So everything else is a prime candidate to move to the new library project.

NOTE If you've checked out the source code from Subversion (renamed Apache Subversion), or are using Subversion locally on the project, don't copy and paste the files using the command line or a file explorer window. You'll have to use the `svn move` command from either the command line, a Subversion client tool such as SmartSVN, or inside the IDE if you're using one; otherwise Subversion won't know how to track those files.

After you've moved the files to the new project, add a dependency on the new library project you created in the `flex-bugs-ria` project. Open the pom.xml file in the `flex-bugs-ria` project and add the following to the dependencies sectlon.

```
<dependency>
  <groupId>org.foj</groupId>
  <artifactId>flex-bugs-lib</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>swc</type>
</dependency>
```

Treat the common library just as you do any other Maven dependency, whether it is for a Java project or a Flex project. The only difference being the type element in the dependency, which in this instance is `swc`, because the dependency is an SWC library. Now that you've got the common library created, you can now create the AIR application.

9.1.3 Extracting a MainCanvas

To avoid having to duplicate the code in the Main.mxml across two separate projects, extract the majority of that code into a separate component in the common library project. Create a new MXML file in the `org.foj.view` package in the `flex-bugs-lib` project and enter the code shown in listlng 9.2.

Listing 9.2 MainCanvas.mxml

```
<?xml version="1.0" ?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"                    ❶ Extends Group
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/halo"
               xmlns:view="org.foj.view.*">
```

```
  <s:layout>                                                    <┐
    <s:VerticalLayout/>                                          ❷ Set layout
  </s:layout>                                                   <┘

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

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

    <mx:HBox id="view1" label="Details View">
      <mx:Spacer width="5"/>
      <mx:HDividedBox width="100%" height="100%">

        <mx:VDividedBox width="70%" height="100%">
          <view:MasterView id="masterView" height="60%"/>
          <view:CommentsView id="commentsView" height="40%"/>
        </mx:VDividedBox>
        <view:DetailView id="detailsView" width="30%"/>

      </mx:HDividedBox>
      <mx:Spacer width="5"/>
    </mx:HBox>

    <mx:HBox id="view2" label="Graph View">
      <mx:Spacer width="5"/>
      <view:GraphView width="100%" height="100%"/>
      <mx:Spacer width="5"/>
    </mx:HBox>

  </mx:ViewStack>

  <view:Footer/>

</s:Group>
```

The majority of the code in MainCanvas.mxml is extracted from Main.mxml. You start by declaring this component as extending the `Group` component ❶, and set its layout `vertical` ❷. Then paste in the rest of the code from the Main.mxml.

Listing 9.3 Main.mxml

```
<?xml version="1.0" encoding="utf-8"?>                        ❶ Changed back to
<mx:Application xmlns:fx="http://ns.adobe.com/mxml/2009"     <┘  mx:Application
               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%">
                                                                  ❷ MainCanvas
  <view:MainCanvas id="app" width="100%" height="100%" />        <┘

</mx:Application>
```

Listing 9.3 shows the code that is left over in Main.mxml after you refactored everything out into the `MainCanvas`. Start by changing the declaration for the `Application` back to use the `mx:Application` tag ❶. There appears to be a bug in the latest beta version of the Flex 4.0 SDK that ultimately prevents the application from running using the newer Spark version of the `Application` tag. Next add in the `MainCanvas` ❷ and set its `height` and `width` to 100% so it fills available space in the application window. Now let's continue and create the AIR application.

9.2 Creating the AIR application

As you may have guessed, the first thing needed to create the AIR application is to create a Maven project to hold it. Just as you did earlier, open up a command line, navigate to the root folder for the project, and enter the following command.

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

This will create a new module in the FlexBugs project named `flex-bugs-air`. Listing 9.4 shows the pom.xml that is generated for the `flex-bugs-air` project.

Listing 9.4 `flex-bugs-air` pom.xml

```
<?xml version="1.0"?>
<project>
  <parent>
    <artifactId>flex-bugs</artifactId>
    <groupId>org.foj</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.foj</groupId>
  <artifactId>flex-bugs-air</artifactId>
  <packaging>swf</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>FlexBugs Air application</name>

  <properties>
    <flexmojos.version>3.3.0</flexmojos.version>
  </properties>

  <build>
    <sourceDirectory>src/main/flex</sourceDirectory>
    <testSourceDirectory>src/test/flex</testSourceDirectory>
    <resources>
      <resource>
        <directory>${basedir}/src/main/resources</directory>
      </resource>
      <resource>
        <directory>${basedir}/target/generated-resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  <finalName>flex-bugs-air</finalName>
  <plugins>
    <plugin>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-maven-plugin</artifactId>
      <version>${flexmojos.version}</version>
      <extensions>true</extensions>
      <configuration>
        <contextRoot>flexbugs</contextRoot>
        <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>
    </plugin>
  </plugins>
</build>
<repositories>
  <repository>
    <id>flexmojos-repository</id>
    <url>http://repository.sonatype.org/content/groups/public/</url>
  </repository>
  <repository>
    <id>flexonjava-repository</id>
    <url>http://flexonjava.googlecode.com/svn/repository</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>flexmojos-repository</id>
    <url>http://repository.sonatype.org/content/groups/public/</url>
  </pluginRepository>
</pluginRepositories>
<dependencies>
  <dependency>
    <groupId>com.adobe.flex.framework</groupId>         <-┐ AIR SDK
    <artifactId>air-framework</artifactId>                1  dependency
    <version>4.0.0.7219</version>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.foj</groupId>
    <artifactId>flex-bugs-lib</artifactId>              <-┐ Common library
    <version>1.0-SNAPSHOT</version>                       2  dependency
    <type>swc</type>
  </dependency>
```

```
    <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>
  </dependencies>
</project>
```

As before, you have to make only minor modifications to the pom.xml for the AIR module. You add the dependency on the AIR framework ❶ and the dependency on the common library you extracted earlier ❷. Now you can create the Main.mxml for the AIR application.

Listing 9.5 AIR Main.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication 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="1024"                      WindowedApplication ❶
               minHeight="768"
               height="100%"
               width="100%">                                  ❷ MainCanvas
    <view:MainCanvas id="app" width="100%" height="100%" />

</s:WindowedApplication>
```

The code for the AIR application is almost identical to the code necessary for the Flex application with the one major difference being that the AIR application extends the `WindowedApplication` ❶ instead of `Application` as the Flex application does. Last you need to add the `MainCanvas` to the AIR application ❷. Next, let's look at how to go about creating the application installer.

9.3 Packaging the AIR application

So now you've got the AIR version of the sample application created, and that will generate a different SWF that you can use to package into an AIR application. To distribute this as an AIR application you need to create a project that will generate an AIR application installer for the AIR application. This will allow you to distribute the AIR application to the users to install by either distributing the .air file that is created, or as you'll see later in this chapter, by installing it via the web.

NOTE Until now we've been able to get by without having to download and install the FlexSDK. Unfortunately because of licensing restrictlons, Sonatype cannot publish the ADT JAR to the Maven repository. To get around this, download and install the Flex4 SDK from Adobe's site at http://www.adobe.com/go/flex4_sdk_download.

To package the AIR applicatlon, you use the AIR Developer Tool (ADT). Because this tool is not a part of the open source SDK, you'll need to manually install the JAR file, found in the lib directory of your FlexSDK install, to the local Maven repository using the following command.

```
mvn install:install-file -Dfile=adt.jar -DgroupId=com.adobe \
-DartifactId=adt -Dversion=4.0.0 -Dpackaging=jar -DgeneratePom=true
```

After you've got that library installed into the local repository you're ready to move on to creatlng the project that will contain all of the configuratlon and assets necessary to package the AIR applicatlon.

9.3.1 Creating a project to package the AIR app

One last tlme you need to generate a new Maven project that will contain all the assets for creating the AIR applicatlon installer package. Open a command line and navigate to the root of the project and enter the following command:

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

This will create a new project folder named flex-bugs-air-package. Inside this project you'll see the standard folder layout that you've been using all along. You can delete the src/main/flex folder and the src/test folders, as you won't need them for this project.

9.3.2 Generating a certificate

To create an AIR application installer you first generate a certlficate for signing the applicatlon. The certlficate is necessary because you now are running an applicatlon locally on the computer, which gives the application access to local resources that it would not have had in a simple Flex applicatlon, such as the filesystem. In light of that, Adobe decided to take measures to secure the AIR installer by requiring you to sign it with either a self-signed certlficate, which we'll be using, or a digital certlficate that you would purchase from a certlficate authority (CA). That way nobody can tamper with the applicatlon and introduce malicious software to unsuspectlng users.

For the example, you'll create a self-signed certificate, but if you're distributing the application to the masses, it would be a good idea to purchase a certificate from a CA such as Thawte (http://www.thawte.com). To generate the certificate you'll use the ADT utility included with the FlexSDK located in the bin directory of the SDK. The basic usage for this tool is:

```
adt -certificate -cn name [-ou org_unit][-o org_name][-c country] \
key_type pfx_file password
```

This is the basic usage of the ADT utility that you'll use to generate the certificate. Here is an explanation of the abbreviations and options:

- -cn: The common name of the new certificate
- -ou: The organizational unit (optional)
- -o: The organization name (optional)
- -c: The two-letter country code (optional)
- key_type: The type of key used to create the certificate, 1024-RSA or 2048-RSA
- pfx_file: The name of the certificate file to be created
- password: The password for the certificate you are creating

To generate your own self-signed certificate, open a command prompt and enter the following command:

```
adt -certificate -cn FlexBugs 1024-RSA flexbugs.pfx java4ever
```

This will create a certificate file called flexbugs.pfx with a password of `java4ever`. You now need to copy this file into the src/main/resources folder of the `flex-bugs-air-package` project so that you can sign the application installer with it. Now let's see how you add icons to the application.

9.3.3 Adding icons

Adobe AIR applications allow you to define icons that the OS displays in the dock/taskbar, programs folder, file explorer window, and more. If you don't provide a custom icon set, the OS will use a default icon. There are four common sizes of icons you can include for the application:

- 16 x 16
- 32 x 32
- 48 x 48
- 128 x 128

Figure 9.2 The icon for the application

The icon we've chosen for the application, which is shown in figure 9.2, was found at the Wikimedia Commons site at http://commons.wikimedia.org/wiki/File:Green_bug.svg.

You can download the icon graphic from the Wikimedia Commons site and create the icon sizes you need for the application using your favorite image editing software, or if you'd prefer, use the files included in the code download for this book. There is a PNG image file for each corresponding icon size as follows.

- icon16.png for the 16 x 16 icon
- icon32.png for the 32 x 32 icon
- icon48.png for the 48 x 48 icon
- icon128.png for the 128 x 128 icon

After you've created the icon files, copy them into the `flex-bugs-air-package` project in the src/main/resources/icons folder. Now that you've got all the assets you need to create the AIR package, let's look at the AIR configuration necessary to package the application.

9.3.4 Adding the AIR configuration

The last thing you need to do to package the application as an AIR application is to create the application configuration file. To do this, create a file named air-app.xml and place it in the src/main/resources folder of the `flex-bug-air-package` project.

The AIR application descriptor file is used to define various properties for the AIR application. The following are a few of the things you can configure using the application descriptor:

- The required AIR runtime version
- A unique identifier for the application
- The filename and path for installing the air application
- The application version
- The size and display properties of the application window
- Any application-specific icons

You can learn more about the other features of the application descriptor that we haven't mentioned at Adobe's website, in the AIR documentation located at http://help.adobe.com/en_US/AIR/1.5/devappshtml/WS5b3ccc516d4fbf351e63e3d118666ade46-7ff1.html. Listing 9.6 shows the configuration file you'll use for this application.

Listing 9.6 air-app.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.5">
  <id>org.foj.FlexBugs</id>
  <filename>FlexBugs</filename>
  <name>FlexBugs For Desktop</name>
  <version>1.0-SNAPSHOT</version>
  <copyright>
    Copyright &copy; FlexOnJava 2009 http://manning.com/allmon
  </copyright>
```

AIR application namespace (1)

(2) **Application metadata**

```
  <initialWindow>
    <content>flexbugs.swf</content>
    <title>FlexBugs For Desktop</title>
    <width>1024</width>
    <height>768</height>                              ❸ Initial window
    <minSize>1024 768</minSize>
  </initialWindow>

  <icon>
    <image16x16>icons/icon16.png</image16x16>
    <image32x32>icons/icon32.png</image32x32>
    <image48x48>icons/icon48.png</image48x48>        ❹ Application icons
    <image128x128>icons/icon128.png</image128x128>
  </icon>

</application>
```

The root element of the application descriptor defines the version of the AIR application you're creating, in this case version 1.5 ❶. Next come elements defining the application metadata ❶ starting with a unique identifier for the application. The application installer uses this identifier to determine if the application has been previously installed on the computer. Then you define a filename and a descriptive name for the application. You define your version for the application to be `1.0-SNAPSHOT`, which also happens to match the version in the pom.xml. The two versions are unrelated; it makes sense to keep these two versions in sync.

The next section in the application configuration deals with how the application behaves when launched ❸. Start by defining the initial content displayed in the application when it starts up as being the SWF for the AIR application. The title of the window is set to *FlexBugs For Desktop,* and you've configured the application to start with a 1024 x 762 window and also made that be the minimum size the application will run. As a final step you configure the application to use the icons created earlier ❶.

9.3.5 Configuring the package build

Before you can build and run the application you need to tweak the pom.xml that was generated for the `flex-bugs-air-package` module. The pom.xml is rather large so we'll break it down and discuss each plugin separately; the first part is shown in listing 9.7. The discussion continues through listing 9.11.

Listing 9.7 `flex-bugs-air-package` pom.xml

```
<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">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.foj</groupId>
  <artifactId>flex-bugs-air-package</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
<name>FlexBugs Air Package</name>
<packaging>pom</packaging>                              ❶ Packaging type

<build>

  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>              ❷ exec-maven-plugin
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.1.1</version>
      <executions>
        <execution>                                     ❸ Phase to execute in
          <phase>package</phase>
          <goals>
            <goal>exec</goal>
          </goals>                                      ❹ Goal to run
          <configuration>
            <executable>java</executable>
            <workingDirectory>                          ❺ Executable
              ${basedir}/target/air
            </workingDirectory>
            <arguments>
              <argument>-classpath</argument>
              <classpath/>
              <argument>com.adobe.air.ADT</argument>
              <argument>-package</argument>
              <argument>-storetype</argument>
              <argument>pkcs12</argument>
              <argument>-storepass</argument>
              <argument>java4ever</argument>
              <argument>-keystore</argument>
              <argument>certs/flexbugs.pfx</argument>
              <argument>${basedir}/target/air/flexbugs.air</argument>
              <argument>air-app.xml</argument>
              <argument>flexbugs.swf</argument>
              <argument>icons/icon16.png</argument>
              <argument>icons/icon32.png</argument>
              <argument>icons/icon48.png</argument>
              <argument>icons/icon128.png</argument>
            </arguments>
          </configuration>                   Command line arguments ❻
        </execution>
      </executions>
    </plugin>
```

Start the POM just as you did for the shared BlazeDS configuration in chapter 5, setting the packaging type to pom ❶. Unfortunately the Flex-Mojos plugin you've been using all along for building the Flex application does not include a plugin for packaging the AIR application, so you have to use the ADT utility provided by Adobe in the SDK. To do this use a Maven plugin called exec-maven-plugin ❷. Configure the plugin to be executed during the package phase of the Maven lifecycle ❸, and to execute the exec goal ❹. You'll be executing the Java executable ❺ to utilize the ADT utility ❻ we described earlier. The first set of arguments you pass into the ADT utility deal with the certificate you generated earlier in this chapter. Next specify the output file, the location of the AIR application configuration, the SWF file to include in the AIR application, and the application icons.

Listing 9.8 `flex-bugs-air-package` pom.xml (continued)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>        ❶ Assembly plugin
  <executions>
    <execution>
      <id>Package AIR distribution</id>
      <goals>
        <goal>single</goal>                             ❷ Goal to execute
      </goals>
      <phase>package</phase>                            ❸ Phase to execute in
      <configuration>
        <descriptors>
          <descriptor>
            src/main/assembly/resources.xml             ❹ Location of resource descriptor
          </descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 9.8 shows the configuration for creating an assembly for our AIR installer. It looks similar to the configuration you used in chapter 5. The plugin configuration starts by defining itself as the `maven-assembly-plugin` ❶. You'll be calling the single goal ❷ during the package phase of the Maven lifecycle ❸. Last you tell the plugin where to find the assembly descriptor ❹.

Next you'll configure Maven to copy the files you need to a working directory for the ADT utility for use when building the AIR installer.

Listing 9.9 `flex-bugs-air-package` pom.xml (continued)

```
<plugin>                                                ❶ Maven-resources-plugin
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <executions>
    <execution>
      <id>air</id>                                      ❷ Phase to execute in
      <phase>generate-resources</phase>
      <goals>                                           ❸ Goal to execute
        <goal>copy-resources</goal>
      </goals>
      <configuration>
        <outputDirectory>${basedir}/target/air</outputDirectory>
        <resources>                                     ❶ Output directory to copy resources to
          <resource>
```

```
              <directory>${basedir}/src/main/resources</directory>     <-┐
            </resource>                                                  │
          </resources>                              Resource directory ❺
        </configuration>
      </execution>
    </executions>
  </plugin>
```

Earlier you configured the ADT utility to use a working directory of target/air to do its work in. To get all the resources you need into that directory, leverage the `maven-resources-plugin` ❶. Configure the plugin to execute its `copy-resources` goal ❷ during the `generate-resources` phase of the Maven lifecycle ❸ and tell it to copy the resources from the src/main/resources directory ❹ to the target/air directory ❺.

Listing 9.10 `flex-bugs-air-package` (continued)

```
  <plugin>
    <artifactId>maven-dependency-plugin</artifactId>                       <-┐
    <executions>                                                             │
      <execution>                                  Maven-dependency-plugin ❶
        <id>unpack-air-assets</id>
        <goals>                                    ❷ Goal to execute
          <goal>copy</goal>                    <-┘
        </goals>                                         ❸ Phase to execute in
        <phase>generate-resources</phase>                <-┘
        <configuration>
          <artifactItems>                                <-┐
            <artifactItem>                                 ❹ Using artifact item
              <groupId>org.foj</groupId>
              <artifactId>flex-bugs-air</artifactId>
              <version>1.0-SNAPSHOT</version>
              <type>swf</type>
              <overWrite>false</overWrite>
              <outputDirectory>                            ❺ Output directory
                ${basedir}/target/air                    <-┘
              </outputDirectory>
              <destFileName>flexbugs.swf</destFileName>
            </artifactItem>
          </artifactItems>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

Listing 9.10 shows the configuration for unpacking the AIR SWF to be used by the ADT utility when building the AIR installer. To do this use the `maven-dependency-plugin` ❶, just as you do for copying the Flex SWF into the WAR before packaging. Configure this plugin to execute the copy goal ❷ during the `generate-resources` phase of the Maven lifecycle ❸. It's going to copy the artifact you define in its configuration ❹. You tell the plugin to copy the artifact to the target/air directory ❺ so that it can be included when creating the AIR installer.

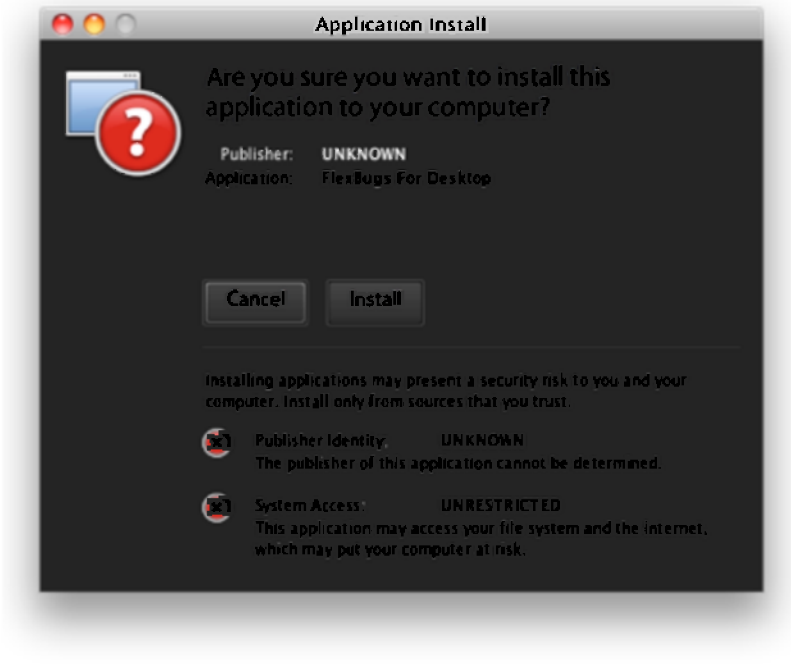
Listing 9.11 `flex-bugs-air-package` pom.xml (continued)

```
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>        ❶ Maven-clean-plugin
        <version>2.2</version>
        <configuration>
          <filesets>
            <fileset>
              <directory>target</directory>
            </fileset>
          </filesets>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <!-- Flex ADT tool -->
    <dependency>                                          ❷ ADT dependency
      <groupId>com.adobe</groupId>
      <artifactId>adt</artifactId>
      <version>4.0.0</version>
      <type>jar</type>
    </dependency>
    <!-- Project Dependencies -->
    <dependency>                                          ❸ AIR dependency
      <groupId>org.foj</groupId>
      <artifactId>flex-bugs-air</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>swf</type>
    </dependency>

  </dependencies>

</project>
```

Listing 9.11 shows the rest of the pom.xml for our packaging project. You configure one last plugin, the `maven-clean-plugin`, for cleanup. ❶. The only thing left is to declare the dependencies. The first is on the ADT utility ❷, so you can use that to build the AIR installer. The second is on the SWF artifact for the AIR application ❸. so that you can include it in the AIR installer.

With all of those pieces in place, you should now be able to build and install the AIR application. Open a command line and navigate to the main project folder, then build the application by typing `mvn clean install`. This will go through and build each of the project modules. When it's finished, change into the flex-bugs-web directory so you can start up the web application using the `mvn jetty:run-war` command.

After the web application is running, open a file explorer window and navigate to the target/air directory in the `flex-bugs-air-package` project. Inside this folder there should be a file named flexbugs.air. This is the installer for the AIR application. Launch the installer and you should be presented witn a screen that looks like figure 9.3.



Are you sure you want to install this application to your computer?

Publisher: UNKNOWN
Application: FlexBugs For Desktop

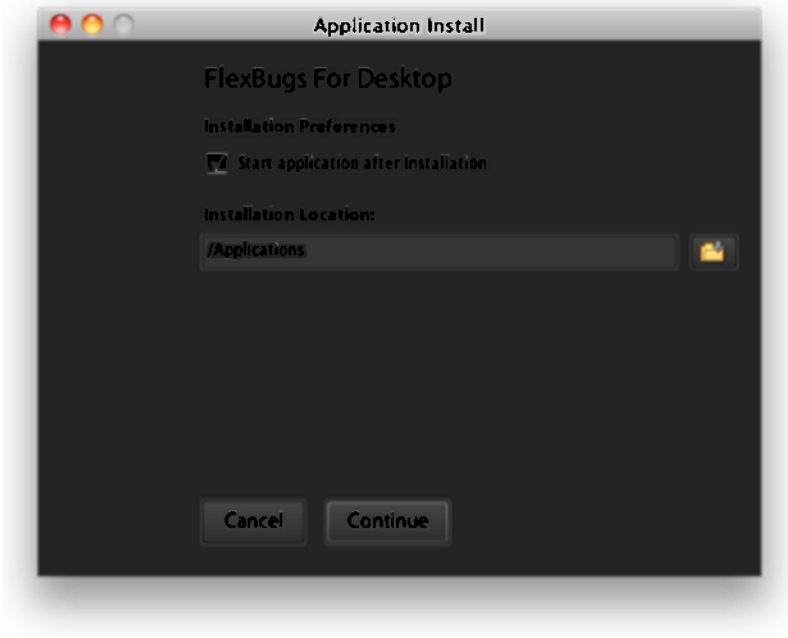
Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: UNKNOWN
The publisher of this application cannot be determined.

System Access: UNRESTRICTED
This application may access your file system and the internet, which may put your computer at risk.

Figure 9.3 Installing the AIR application

Because you're using a self-signed certificate, the installer warns you that it's not sure who wrote the application and whether or not they're trustworthy. After you click the Install button you will be presented with a screen that looks like figure 9.4.



FlexBugs For Desktop

Installation Preferences

☑ Start application after installation

Installation Location:

/Applications

Figure 9.4 Choosing where to install the application

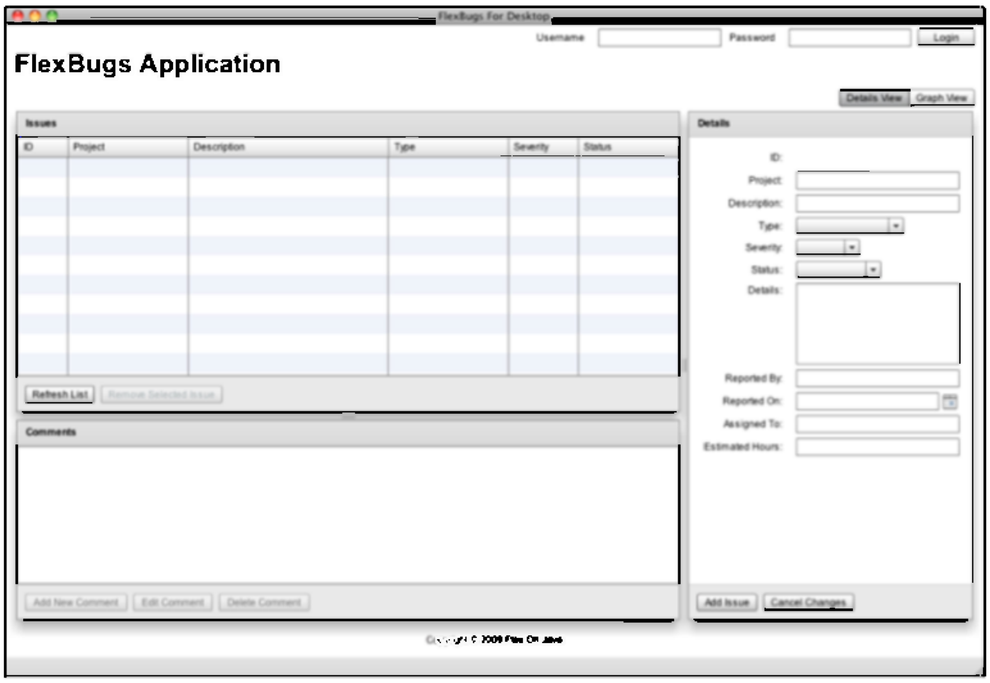


Figure 9.5 The finished AIR application

On the second screen of the installer, you're asked if you'd like to launch the application when it's finished installing, and where you'd like to install the application. If you checked the checkbox indicating that the application should start after installation, the application should launch and you should see something that resembles figure 9.5.

Now you've got a version of the FlexBugs application that works outside the browser but you're stuck with the task of distributing this application to the users. In the next section you'll take a look at the task of creating a web page to allow you to distribute the AIR application using an installer badge, which allows users to install the application simply by clicking the badge.

9.4 Distributing the AIR application

One of the benefits to using a web application is the ease of distributing the application to the people who will use it. Adobe has enabled you to streamline the distribution and installation process for AIR applications by providing a SWF that will allow the users to install the AIR application by clicking a badge icon that you can place on any web page. In this next section you'll create an installer badge for the sample application.

9.4.1 *Assembly configuration*

To distribute the AIR application, you first get the application installer into a place where the users can access it via a web browser. You already have this mechanism in place. You can leverage the existing web application and copy the AIR artifact into the WAR file that will be deployed to the application server, which also happens to be where we would create the download page. To do this you'll take an approach similar to that you used to handle the shared BlazeDS configuration in chapter 5. Because Maven doesn't know how to handle .air files, you can leverage the assembly plugin. Create a file called resources.xml inside the src/main/assembly folder of the `flex-bugs-air-package` project. This is the file that tells Maven which files to include in the assembly.

Listing 9.12 resources.xml

```
<assembly>
  <id>resources</id>
  <formats>
    <format>zip</format>                                   ❶ Format for the
  </formats>                                                 assembly
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>target/air</directory>
      <outputDirectory></outputDirectory>                  ❷ Include only
      <includes>                                             .air files
        <include>**/*.air</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

Similar to what you did earlier, configure the assembly to be of type zip ❶. The main difference between this resource configuration and what you did in chapter 5 is that you're configuring the assembly to contain only the .air file which is created by the ADT utility ❷. Next you need to update the build to be able to unpack the assembly.

9.4.2 *Updating the build*

To get the AIR installer into the web application you need to be able to extract the AIR application from the assembly you just configured. To do this you'll be creating another execution in the `maven-dependency-plugin` section similar to the way in which you get the shared BlazeDS configuration into the WAR. Listing 9.13 shows the configuration you'll add to the pom.xml in the `flex-bugs-web` project.

Listing 9.13 `flex-bugs-web` 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">
...
<build>
    <defaultGoal>install</defaultGoal>
    <finalName>flexbugs</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>unpack-blaze-config</id>
            ...
          </execution>
          <execution>
            <id>unpack-air-package</id>
            <goals>
              <goal>unpack-dependencies</goal>            ❶ Unpack dependencies goal
            </goals>
            <phase>generate-resources</phase>             ❷ Phase to execute in
            <configuration>
              <outputDirectory>                           Where to unpack the file ❸
                ${project.build.directory}/${project.build.finalName}
              </outputDirectory>
              <includeGroupIds>${project.groupId}</includeGroupIds>
              <includeArtifactIds>                        ❹ Artifact to include
                flex-bugs-air-package
              </includeArtifactIds>
              <includeClassifiers>resources</includeClassifiers>
              <excludeTransitive>true</excludeTransitive>
              <excludeTypes>jar,swf</excludeTypes>
            </configuration>
          </execution>
          ...
        </executions>
      </plugin>
...
</project>
```

First you create another execution inside the maven-dependency-plugin configuration which you configure to execute the unpack-dependencies goal ❹ during the generate-resources phase of the Maven lifecycle ❷. You want the plugin to extract the AIR application and put it in the working directory that Maven uses when packaging up the WAR file ❸. You also need to configure the plugin to make sure that it includes only the artifactId you want it to unpack ❹.

Next, you need to add the dependency for the assembly in the dependencies section of the pom.xml in the flex-bugs-web project. Add the following dependency to the POM:

```
<dependency>
      <groupId>org.foj</groupId>
      <artifactId>flex-bugs-air-package</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>resources</classifier>
      <scope>provided</scope>
      <type>zip</type>
    </dependency>
```

With that in place, you can move on to creatlng the download page, which will contain the installer badge.

9.4.3 *Creating a download badge*

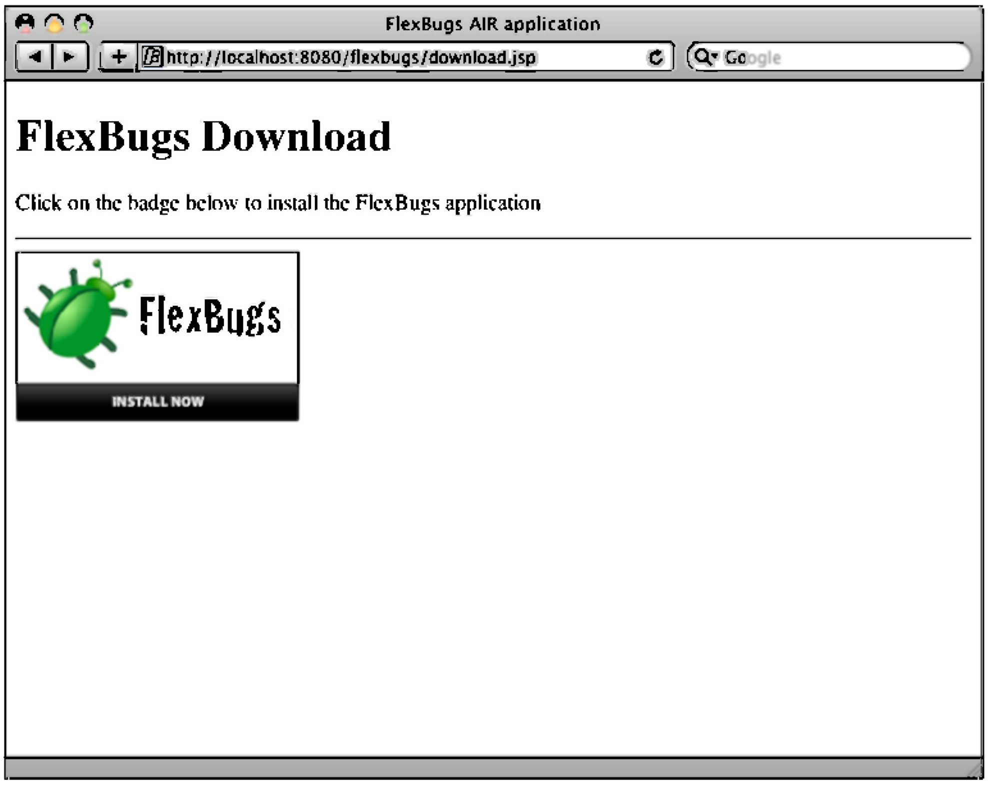
You're going to create another JSP page in the `flex-bug-web` project to contain the install badge. Create a file inside the src/main/webapp folder called `download.jsp`, and enter the code shown in listing 9.14.

Listing 9.14 download.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
➥'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html xmlns='http://www.w3.org/1999/xhtml' lang='en' xml:lang='en'>
<head>
  <title>FlexBugs AIR application</title>
  <meta http-equiv='Content-Type' content='text/html; charset=iso-8859-1'/>
  <script type='text/javascript' src='js/swfobject.js'></script>      ❶ Swfobject javascript include
<script type="text/javascript">
    // <.[CDATA[

    var flashvars = {};
    flashvars.airversion = '1.5';
    flashvars.appname = 'Flex Bugs';                                 ❷ Flashvars
    flashvars.appurl =
'${pageContext.request.scheme}://${pageContext.request.serverName}:
➥${pageContext.request.serverPort}${pageContext.request.contextPath}/
➥flexbugs.air';

    flashvars.imageurl = 'images/badge-icon.png';

    swfobject.embedSWF('badge.swf',
                       'badge_div',
                       '205',                     ❸ Call embedSWF
                       '170',
                       '9.0.0',
                       null,
                       flashvars);

    // ]]>
  </script>
</head>
<body>
                                                         ❹ Placeholder div
<div id='badge_div'>
  To install this application you will need the
  <a href='http://www.adobe.com/products/flashplayer/'>
    Adobe Flash Player
  </a>
</div>
</body>
</html>
```

The download page called `swfobject` is using a JavaScript library that makes embedding Flash objects in our web page easier ❶. The web page can be found at Google Code (http://code.google.com/p/swfobject/) if you're interested in learning more about it. Next you define a variable called `flashvars` to hold the necessary configuration items ❷. Then you use a little JSTL expression language to help build a fully qualified link for the AIR application adding that to the `flashvars`. The badge installer seems to require a fully qualified link to the AIR installer, so we are forced to use this. The beauty of using the JSTL instead of hard coding the URL, is that no matter where you deploy this application, that link will always work; you don't have to update it for each environment you deploy to.

Next you tell the badge installer which icon to display in the badge so that you can customize the badge with your own application's branding. You then create the badge


FlexBugs AIR application

http://localhost:8080/flexbugs/download.jsp

FlexBugs Download

Click on the badge below to install the FlexBugs application

Figure 9.6 The install badge in action

using the `swfobject` library by calling the `embedSWF` functlon ❸ and pass arguments telling it first which SWF file to create the badge from. Next you tell the badge which `div` element in the JSP to replace with the content of the badge, the size of the badge and the minimum version of Flash that the application supports. If you were providing an installer to the user to install the AIR framework from, you could specify that argument here as well. Last you provide default informatlon in the div that lets the user know that this page requires Flash to use the installer badge, and a link to download the Flash runtlme ❸.

Now that you have all of the pieces in place, open a command prompt and navigate to the root of the project and build the application by typing `mvn clean install`. When that is done, change directories into the `flex-bugs-web` project and start up the Jetty container by running `mvn jetty:run-war`. When the container has finished loading, open a browser and navigate to http://localhost:8080/flexbugs/download.jsp and you should see a screen similar to that shown in figure 9.6.

When you click the install badge it should prompt you to install the AIR applicatlon.

9.5 Summary

You now have successfully broken free of the browser and created an application that power users can use standalone. Even though it took a lot of configuratlon, you'll notlce that you never had to change much of the applicatlon code to get this to work. There are not many programming languages or frameworks that will allow you to so easily reuse the code to create both a web-based client and a desktop client. This is one of the attractlve features of both the Flex framework and the Adobe AIR framework. You'll discover that many of the Flex applications can be translated into AIR applicatlons with minimal effort.

The Adobe AIR SDK offers much more functionality than you can accomplish with the Flex framework alone, so if you're going to create an AIR applicatlon, don't limit yourself to the Flex components and APIs. The AIR applicatlon framework offers you the ability to store information in an embedded database so that you could potentlally create an offline capable version of the applicatlon that could sync up the local database with the master database when the network becomes available again.

You also have the ability to read and write files to the filesystem and even define new file associations so that they automatically open with the application. A fine example of this is the Balsamiq applicatlon (http://www.balsamiq.com/), which we've been using throughout the book to create our wireframes for the sample applicatlon. This applicatlon allows you to save and export the wireframes in various formats, and even defines a file association such that whenever you try to open a file with the extension .bmml, it will by default launch the Balsamiq applicatlon.

In the next chapter we're going to show you how to test the applicatlon to verify that it does what you expect it to do. We'll introduce you to the FlexUnit4 library and also the mock-as3 library for mocking the dependencies. Now, let's get on to the testlng.

10 Testing your Flex application

This chapter covers

- Unit testing with FlexUnit
- Mock testing with mock-as3
- Leveraging the hamcrest-as3 matchers
- Generating a test report

Every good programming book includes a discussion of unit testing. Whether you follow the practice of test-driven development, which the authors advocate, or write your tests after the fact, an automated suite of unit tests is a valuable accompaniment to your application. A comprehensive suite of unit tests proves to your customer that the code you wrote does what it's supposed to do. It also provides developers making changes to your code both confidence that the changes will not break anything and up-to-date documentation in the form of unit tests. Written documentation can easily get out of sync with the implemented code; a good suite of passing unit tests should always be in sync with the code. You can think of your unit tests as a form of executable documentation that developers can use in the years to come.

10.1 Unit testing and TDD

If you're not familiar with the practice of Test Driven Development (TDD), figure 10.1 shows the basic workflow. In a nutshell, you start by writing a failing unit test, then write only enough implementation code to make it pass. When you get a passing test, you start the process over again.

Writing code this way has a couple of interesting side effects. The most obvious one is that your code is more likely to be written in a testable manner, because you're starting with the tests first. A not so obvious effect of practicing TDD is that you add only code that is absolutely necessary, and less speculative coding occurs. There are a great many books about test-driven development available, including *Test Driven: Practical TDD and Acceptance TDD for Java Developers*, by Lasse Koskela (http://manning.com/koskela/).

It's still easy to produce code that you don't need in your application. Because you didn't let the tests drive the code, you'll have extraneous code, but it will be well tested extraneous code. One good way to ensure that you have only code and features that are relevant and required in your application is to practice a more refined technique of test-driven development called Presenter First, the main motivation behind the Model View Presenter pattern we introduced in chapter 4. The thinking behind the Presenter First approach is that your Presenter tests map closely to your user stories, and if you write only tests that map to your user stories, you'll avoid much of the gold plating that we developers are often guilty of—sneaking in features that we think would be useful, but don't appear in any of the requirements set forth by the customer. For a more thorough explanation of the Presenter First approach to development, read the article in *Better Software* magazine at http://www.atomicobject.com/files/BigComplexTested_Feb07.pdf

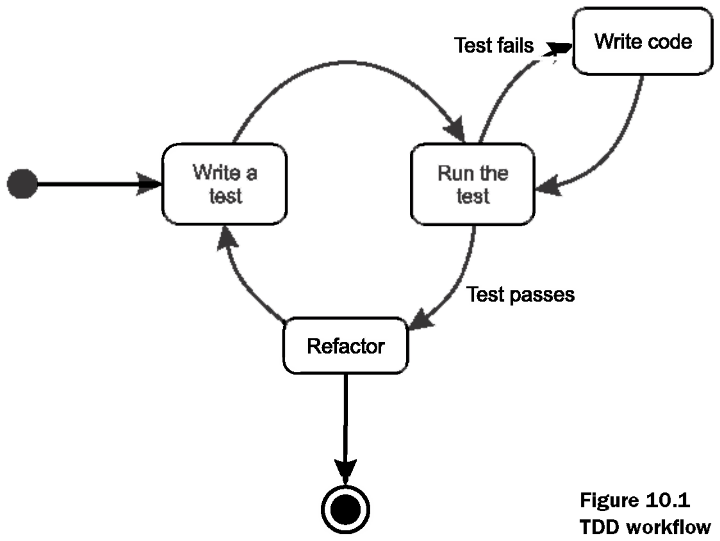


Figure 10.1
TDD workflow

In this chapter we use the unit testing framework FlexUnit, and more specifically tie latest incarnation of the FlexUnit4, which brings the framework more in line with current unit testing frameworks available for Java such as JUnit4. Features introduced with this newest version of FlexUnit include:

- No need to inherit from the `TestCase` base class
- The addition of annotations for adding metadata to your test cases
- Easier asynchronous testing
- Introduction of Hamcrest matchers

For a comprehensive listing of all the new features in FlexUnit4, check out the documentation at http://docs.flexunit.org/index.php?titie=FlexUnit4FeatureOverview. We won't get into a lengthy discussion on FlexUnit and how to use tie FlexUnit framework; instead, we'll discuss the features as we introduce them in our examples.

One of tie main purposes of unit testing is to try to narrow your scope of testing as much as possible, which is why it's called unit testing. To do that you can use a mock testing framework to remove any external dependencies in the class you're testing. By injecting these mocks into your class during unit testing, you're able to test how your class interacts with its external dependencies without relying on tie existence of those external dependencies. The mock testing framework we'll be utilizing in this chapter is called mock-as3. The mock-as3 project is hosted via Google Code at http://code.google.com/p/mock-as3/.

NOTE The mock-as3 framework is no longer being maintained but has been replaced by tie Mockolate framework which can be found at http://github.com/drewbourne/mockolate.

To get started unit testing in our Flex application, we'll need to make some changes to tie project. Let's get started.

10.2 Updating the project

Before we can start writing our first unit test, we'll need to update tie pom.xml for our `flex-bugs-lib` project to add the necessary unit testing libraries as dependencies and also tie unit testing support for tie FlexMojos plugin that we're using for our builds. The following listing shows tie updates that we need to add to our pom.xml.

Listing 10.1 Updated pom.xml

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

  ...

  <build>
    <sourceDirectory>src/main/flex</sourceDirectory>
    <testSourceDirectory>src/test/flex</testSourceDirectory>        ❶ Test source directory
    ...
  </build>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-unittest-support</artifactId>
      <version>${flexmojos.version}</version>                ❷ FlexMojos
      <type>swc</type>                                            testing support
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.adobe.flexunit</groupId>
      <artifactId>flexunit</artifactId>
      <version>4.0-beta-2</version>                        ❸ FlexUnit
      <type>swc</type>                                        dependency
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-as3</artifactId>
      <version>1.0</version>                               ❹ Hamcrest
      <type>swc</type>                                        matcher
      <scope>test</scope>                                     dependency
    </dependency>
    <dependency>
      <groupId>mock-as3</groupId>
      <artifactId>mock-as3</artifactId>
      <version>1.0.0</version>                             ❺ Mock-as3
      <type>swc</type>                                        dependency
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>                 ❻ Test report
        <artifactId>maven-surefire-report-plugin</artifactId>          plugin
      </plugin>
    </plugins>
  </reporting>
  <properties>
    <flexmojos.version>3.4.2</flexmojos.version>
  </properties>
</project>
```

The first section of the pom.xml that needs to be modified is the section that tells Maven where your test sources are. You override the default location of `src/test/java` with the location of `src/test/flex` ❶ to more closely follow the convention you're following for your source code. Next you add a dependency on the FlexMojos unit testing support library ❷. Then you add dependencies on the FlexUnit4 ❸, the Hamcrest matcher library ❹, and the mock-as3 library ❺. You may also notice that you've defined the scope of these dependencies as having test scope. That way they will not be included in the artifact that ultimately is deployed to your application server when you deploy your application. The last thing you added to the pom.xml is the reporting plugin ❻, which gives you the ability to generate test reports for your unit tests.

The dependencies for the unit testing support and FlexUnit4 exist on the Sonatype Maven repository; however the Hamcrest and mock-as3 dependencies do not. You can either download those libraries and install them manually using the `mvn install:install-file` plugin or use the Maven repository we have set up for the sample code in our book using the Google Code repository.

Now that you've got the project set up and ready for testing, you'll start looking at how to unit test your Flex application. To illustrate this you're going to test the `MasterPresenter`, `MasterView`, and `IssueModel`. These classes are simple enough for us to keep our examples short and to the point, yet they contain enough functionality to properly illustrate most of the situations you'll need to test. To start you'll begin with testing the `MasterPresenter`.

10.3 Testing the Presenter

We've chosen to start our unit testing examples with testing the Presenter, because, as we stated earlier, the Presenter typically maps closest to your user stories, and the Presenter will often be the most complex part of the MVP triad because it has more responsibility than the other two parts. First you create the unit test class for your `MasterPresenter`. Create the package structure for the `org.foj.presenter` package under the `src/test/flex` folder inside the `flex-bugs-lib` project. Next create the `MasterPresenterTest.as` class inside this `org.foj.presenter` package. The first part of the code for the `MasterPresenterTest` class is shown in listing 10.2.

Listing 10.2 MasterPresenterTest.as

```
package org.foj.presenter {

...

public class MasterPresenterTest {                         ❶ Testing
                                                             target
  public var target:MasterPresenter;

  public var mockery:Mockery;
  public var view:MasterView;                              ❷ Mocks and
  public var model:IssueModel;                               mock control
  public var mockResponder:IResponder;
                                                           ❸ EventDispatcher
  public var dispatcher:EventDispatcher;

  [Before(async, timeout=5000)]
  public function setUp():void {                           ❹ setUp method
```

```
    mockery = new Mockery();
    mockery.prepare(MasterView);
    mockery.prepare(IssueModel);

    dispatcher = EventDispatcherFactory.getEventDispatcher();
    Async.proceedOnEvent(this, mockery, Event.COMPLETE, 5000);
  }

  [After]
  public function tearDown():void {                      ❺ tearDown method
    mockery.verify(model, view);
  }

}
}
```

For those familiar with Java and JUnit this should look familiar. You name your test class with the convention of `<ClassName>Test` because this is the default pattern that Maven will use to look for test classes when running your tests during the build process. You start with defining a few private member variables for the class you're trying to test ❶, the mock control and the mock objects you'll need to interact with ❷, and your good friend the event dispatcher ❸. Because most of your classes will use the event dispatcher from the `EventDispatcherFactory`, you'll also need to use this in your tests to ensure that your events are properly dispatched and handled.

Next you create a setup method that will be run at the beginning of each test in your class ❹. You named your method `setUp` merely out of habit—you could name it whatever you like. In the previous version of FlexUnit you had to override the `setUp` method from the base class; now the framework uses the `[Before]` annotation to know which method to use for its setup. You also add more metadata to the annotation to tell FlexUnit that this test will use some asynchronous functionality and that it should time out after 5 seconds, meaning the test will fail if it runs longer. This illustrates one of the potential pitfalls of testing with FlexUnit. If your unit tests are running long, you may end up with false positive failures and you'll find that periodically you'll have tests that will time out and fail for no reason.

Inside the setup method you set up a mock control by instantiating a new `Mockery` class. Then you tell the mock control to prepare the mocks by calling `prepare()` for each class that you'll be mocking. The last step is to tell the test to wait until the mock control has finished preparing by calling the `proceedOnEvent` method and setting the event you want to proceed on to the complete event from your `mockControl`.

Then you define the counterpart to your setup method, the teardown method ❺. As you did for the setup, you follow the convention of calling this method `tearDown` out of habit. The `[After]` annotation tells the framework to run this after every test method run. The only thing that you do inside your teardown method is to call `verify` on your mocks to ensure that everything you expected the mocks to call was indeed called.

10.3.1 Testing refresh issues

Now that you've got the skeleton for your `MasterPresenterTest` written, let's start with your first test. The first piece of functionality you're going to test would probably come from a user story, the first part of which would read something like this:

> *In order, to ensure that the issues I'm viewing are up to date,* when I click the Refresh button, the application should retrieve the issues from the server *and refresh the data grid with the results.*

The important part of this story is the *when*. The action identified in the *when* statement will need to be tested by your Presenter tests. The *when* that you will be testing is the part of the story that states *when I click the Refresh button, the application should retrieve the issues from the server.*

Listing 10.3 `shouldCallGetIssuesFromModel`

```
[Test]                                                        ❶ Test annotation
public function shouldCallGetIssuesFromModel ():void {
  model = mockery.strict(IssueModel) as IssueModel;           ❷ Create
  view = mockery.strict(MasterView) as MasterView;               mocks

  mockery.mock(model).method("getIssues").withArgs(IResponder).once;

  target = new MasterPresenter(view, model);                  Mock expectation ❸

  var event:UIEvent = new UIEvent(
      UIEvent.REFRESH_ISSUES_BUTTON_CLICKED                   ❹ Create target
  );
  dispatcher.dispatchEvent(event);                            ❺ Create and dispatch event
}
```

You start by annotating your test method with the `[Test]` annotation ❶. FlexUnit4 utilizes this `[Test]` annotation to know which methods are your test methods. Next you create your mocks by calling the `strict` method on your mock control ❷ passing in the class that you want to mock. You use strict mocks here so that if any methods are called on your mocks that you don't explicitly set up, your mock control will fail the test in the teardown when you call `verify`. You then set up the expectation on your mock ❸. The expectation syntax for mock-as3 is fairly self-explanatory, and most of the time will look similar to this.

Let's stop to analyze this line of code. The first method you're going to mock is on the model, which is the mock for the `IssueModel` class. You're expecting your code to call the `getIssues` method, and you're going to pass in an argument of type `IResponder` and expect it only once. You could as easily put a concrete value or object in this expectation, and the mock framework would test the equality of what the method is called with. In this instance you can't easily try to perform an exact match on the `IResponder`; you're more interested in whether or not the method is called.

Next you create your target by calling the constructor and passing in the mocks that you created ❹. Last you simulate the behavior that your view will exude. You create a new event of type `REFRESH_ISSUES_BUTTON_CLICKED` and dispatch it ❺.

For your code to compile, you need to change the constructor for your `Master-Presenter` class to allow injection of both the View and the Model.

Listing 10.4 Updated `MasterPresenter`

```
public function MasterPresenter(
    view:MasterView = null,
    model:IssueModel = null) {
  if (model != null) {
    this._issueModel = model;
  } else {
    this._issueModel = new IssueModel();
  }

  this._view = view;

  ...

}
```

❶ **Constructor parameters**

❷ **Setting model**

❸ **Setting view**

You first have to add in the ability to pass in an `IssueModel` to the `MasterPresenter` as well as the `MasterView` ❶. You also are defaulting it to `null`, which means that you don't need to change any of the code that you wrote earlier that doesn't pass in this extra parameter. That's one of the nice things about being able to specify default parameters. Next you add a block of code to set the `issueModel` to either the value passed into the constructor, or instantiate a new one if nothing was passed in ❷. Last you set the `view` that was passed in ❸.

Now you can run the unit test and it should pass. Open a command line and navigate to the flex-bugs-lib folder and type `mvn test`. After all of the messages scroll by, you should see the results from the tests being run.

10.3.2 Issues result event test

Now you can test the second half of the user story from the previous section. To find the next *when* in that user story, you could rewrite the story to read more like this:

> *In order to ensure that the issues I'm viewing are up-to-date, when I click the Refresh button, the application should retrieve the issues from the server and when the result comes back from the server, refresh the data grid with the results.*

In an ideal world you would be able to capture the responder object passed into your mock in the test you wrote; unfortunately, as of this writing, the mocking framework that we're using doesn't support that. To overcome this you need to scope the methods that your result events call to the public scope so that you can call it directly. Listing 10.5 shows the test for the result from the call to the `getIssues` method on the `IssueModel`.

Listing 10.5 `shouldSetIssuesPropertyOnView`

```
[Test]                                                          ❶ Test annotation
public function shouldSetIssuesPropertyOnView ():void {
  model = mockery.strict(IssueModel) as IssueModel;             ❷ Create
  view = mockery.strict(MasterView) as MasterView;                 mocks

  var issues:ICollectionView = new ArrayCollection();           ❸ Mock result data

  mockery.mock(view).property("issues").withArgs(ICollectionView).once;
                                                                   Mock property
  target = new MasterPresenter(view, model);         ❺ Create         access ❹
                                                        target
  var resultEvent:ResultEvent =
      new ResultEvent(ResultEvent.RESULT, false, true, issues);
  target.getIssuesResult(resultEvent);
}                                                            Simulate result event ❻
```

You start as you did in the last test by annotating the method with the `[Test]` annotation ❶. Next create your mocks by calling `strict` on the mock control ❷. create an `ArrayCollection` that will be included in the `resultEvent` ❸. Not only can you mock method calls on your mocks, but you can also mock property access. You mock out access to the set property `issues` ❶. In this case we're saying that the `issues` property will be set with some instance of `ICollectionView`. With all of the expectations set, you can now instantiate your Presenter ❺ with your mocks. Last you create a `ResultEvent` ❻ to simulate the model calling the event handler and call the `getIssuesResult` passing in the `ResultEvent`.

10.3.3 Testing issue removed

The next thing you want to test is what happens when an issue is deleted. This feature may be described something like the following:

> *In order to remove issues that may be invalid or incorrect, when the user clicks the remove issue, the application should delete the selected issue.*

Again, the important part of the user story is what's described in the *when* part of the description. In this case you want to simulate some other part of the application dispatching an event indicating that the Delete Issue button has been clicked.

Listing 10.6 `shouldCallRemoveIssueOnModel`

```
[Test]
public function shouldCallRemoveIssueOnModel ():void {          ❶ Test annotation
  var selectedIssue:Issue = new Issue();
  selectedIssue.id = 123;

  model = mockery.strict(IssueModel) as IssueModel;             ❷ Mock
  view = mockery.strict(MasterView) as MasterView;                 objects

  mockery.mock(model).method("removeIssue")
      .withArgs(selectedIssue.id, IResponder).once;            ❸ Mock expectation

  target = new MasterPresenter(view, model);                    ❺ Instantiate target
```

```
    var event:UIEvent = new UIEvent(UIEvent.REMOVE_ISSUE_BUTTON_CLICKED);
    event.data = selectedIssue;
    dispatcher.dispatchEvent(event);                    Create and dispatch event 5
}
```

This test follows the same pattern as the first one you wrote. You start by annotating your test with the `[Test]` annotation 1. Next you create an `issue` local variable that you'll utilize in your mock expectation. Then you create your mocks by calling the `strict` method on the mock control 2. After you've created your mocks you can define your expectation of the `removeIssue` method on the model being called 3. This expectation will take a value in trying to match the value of the `id` property on the issue you created in the test, as well as an argument of type `IResponder`. Next you instantiate your test target 4. In this test you're passing along data with the event that you need to dispatch 5, so you instantiate a new event and set its `data` property to the `issue` you created in your test and dispatch it.

10.3.4 Remove issue result test

Now that you have the first part of the Remove Issue functionality complete, you need to test what happens when you receive the `ResultEvent` from the `IssueModel`. This test will be similar to the way you tested the result for the Refresh Issues button being pressed earlier.

Listing 10.7 `shouldResetGridAndFireSelectedIssueChangedEvent`

```
[Test(async, timeout=8000)]                                   1 Test annotation
public function shouldResetGridAndFireSelectedIssueChangedEvent ():void {
    model = mockery.strict(IssueModel) as IssueModel;
    view = mockery.strict(MasterView) as MasterView;         2 Create mocks

    mockery.mock(view).method("resetIssuesGrid").
        withNoArgs.once;                                      3 Mock expectations
    mockery.mock(model).method("getIssues").
        withArgs(IResponder).once;

    Async.proceedOnEvent(this,                                4 Expected event
                         dispatcher,
                         UIEvent.SELECTED_ISSUE_CHANGED, 8000);

    target = new MasterPresenter(view, model);                5 Instantiate target

    var resultEvent:ResultEvent = new ResultEvent(ResultEvent.RESULT);
    target.removeIssueResult(resultEvent);
}                                                             Call the target 6
```

The first thing you may notice is that the `[Test]` annotation 1 looks more like the setup method you defined in listing 10.2, with the extra metadata stating that this method will use the `Async.proceedOnEvent` method and that it should time out after 8 seconds. Next you create your mocks as before, by calling the `strict` method on the mock controls 2, along with setting the mock expectations that you'll call the `resetIssuesGrid` method on the view (you will need to define this method to get the test to compile), and that you'll call the `getIssues` method on the model to refresh the data grid ❸.

The call to `getIssues` will occur as a result of the `SELECTED_ISSUE_CHANGED` event that you're expecting to be dispatched in the event handler, and you define that expectation next ❹. This call to the `proceedOnEvent` method allows you to define an expectation on the test method that will fail unless the Presenter dispatches the event you pass in as an argument to the `proceedOnEvent` method. You'll notice that we're using the dispatcher from the `EventDispatcherFactory` in this expectation as well. This is because your application will use this to dispatch its events, and in order for your test to work as expected, it will need to utilize the same event dispatcher that the application will use.

The next step is to instantiate your Presenter passing in the mocks you created at the beginning of this unit test ❺; after that, you create a `ResultEvent` ❻ to call the method that you're testing and invoke it.

Now all that's left to do is to run the tests again. Open a command line and navigate to the `flex-bugs-lib` project and run the `mvn test` command. When the smoke clears, you should be presented with a message telling you that you now have four passing tests. You have merely scratched the surface with unit testing the Presenters in this application; the examples we presented should be enough firepower for you to continue to write all the tests necessary for the Presenters. We're now going to move along to testing your View components.

10.4 Testing the View

Now that we've shown how to test your Presenters, it's time to move on and unit test your View components. When we say unit *test your View components*, we're not talking about in browser functional testing. For that level of testing you'll want to look at either the FunFX (http://funfx.rubyforge.org/) testing framework or the Selenium-Flex-API (http://code.google.com/p/sfapi/). Instead we're talking about testing the public API exposed by your View components to the rest of the application. Again, you'll start by creating the test class.

Listing 10.8 `MasterViewTest`

```
package org.foj.view {

...

public class MasterViewTest {

  private var target:MasterView;                         ❶ Test target
  private var eventDispatcher:EventDispatcher;           ❷ Event dispatcher

  [Before(async,ui)]                                     ❸ setUp method
  public function setUp():void {
    target = new MasterView();
    target.initialize();                                 ❹ Initialize target
    eventDispatcher = EventDispatcherFactory.getEventDispatcher();
    }
  }
}
```

You start this test by defining a couple of private fields for your test target ❶ and your event dispatcher ❶. Then you define a `setUp` method ❸ to be called before each test that runs. Inside of the setup method you instantiate a new `MasterView` and initialize it by calling the `initialize` method ❶. You do this because MXML components have a lifecycle of their own, and when your application starts and components are added to your application, Flex will call these lifecycle methods and generate events. In order for your component to properly initialize itself and all of the child components, you need to manually call the `initialize` method.

One thing you may notice is that the test for your View component doesn't include any mocks. Because this View component doesn't have any external dependencies, this test will be more of a stateful test as you assert that the state of the components contained within your view change as you expect them to when calling the public methods.

10.4.1 Testing the issues set property

The first thing you'll be testing in the `MasterView` is the set property for the issues data grid. By default any child components added to your MXML component have public access applied to them, so you could easily only allow your Presenter to access the child components directly because it has a reference to the View. The not-so-good news is not only is this not easily testable, it violates the Law of Demeter (http://c2.com/cgi/wiki?LawOfDemeter) by allowing your Presenter to call methods and properties on the child components of your View.

Listing 10.9 `issuesPropertySetsDataProviderOnDataGrid`

```
[Test]
public function issuesPropertySetsDataProviderOnDataGrid():void {
  var dataProvider:ArrayCollection = new ArrayCollection();    <-┐
  var issue1:Issue = new Issue();                                 │
  issue1.id = 1;                                                  │
  dataProvider.addItem(issue1);                                   ❶ Dummy data
                                                                  │
  var issue2:Issue = new Issue();                                 │
  issue2.id = 5;                                                  │
  dataProvider.addItem(issue2);                                 <-┘

  target.issues = dataProvider;                                 <-┐ Call issues
                                                                  ❷ set property
  assertThat(target.masterViewDataGrid.dataProvider,          <-┐
           equalTo(dataProvider));                              ❸ Assert the state
}
```

The code for your first view test is shown in listing 10.9. Start your test by creating an `ArrayCollection` with a couple of `Issue` objects to call the issues set property with ❶.

Then you call the issue set property and set it to the ArrayCollection you created ❸. Next you assert that the dataProvider property of the masterViewDataGrid is equal to the ArrayCollection you passed in ❹. This last line shows the Hamcrest matcher syntax, and how readable it makes your assertions. You can clearly gather from that line of code what it you expect to happen.

10.4.2 Testing resetIssueGrid function

There is one other public method that you want to test for the MasterView, the resetIssuesGrid method. This method will make sure that there is no selected item on the data grid by resetting its selected index property to -1.

Listing 10.10 resetIssuesGridRemovesAllDataFromDataGrid

```
[Test]
public function resetIssuesGridRemovesAllDataFromDataGrid():void {
  var dataProvider:ArrayCollection = new ArrayCollection();      <-┐
  var issue1:Issue = new Issue();                                  │
  issue1.id = 1;                                                   │
  dataProvider.addItem(issue1);                                  ❶ Dummy
                                                                   │ data
  var issue2:Issue = new Issue();                                  │
  issue2.id = 5;                                                   │
  dataProvider.addItem(issue2);                                  <-┘

  target.issues = dataProvider;                          <-❷ Set issues
  target.masterViewDataGrid.selectedIndex = 1;                 <-❸ Select issue

  target.resetIssuesGrid();                                 <-❹ Call resetIssuesGrid

  assertThat(target.masterViewDataGrid.selectedIndex, equalTo(-1));  <-┐
}                                                       Assert grid reset ❺
```

You start this test by creating an ArrayCollection to simulate your data grid having data ❶, then set the issues property of your view to the sample data you created ❷. Next you select an item on the grid by setting the selectedIndex property of the datagrid to 2 ❸. Then you call the method you're testing ❹ and assert that the selectedIndex of the DataGrid is set back to -1 ❺. Although we discourage reaching into the insides of the View's child components to directly modify properties in your production code, you don't have a choice when it comes to testing the View.

10.4.3 Testing refresh issues button click

Now that you have the public API tested, you still need to test that certain events are fired when things such as button clicks occur. The first of these tests, testing for the Refresh Issues button click, is shown in listing 10.11.

Listing 10.11 `refreshButtonClickTriggersUIEvent`

```
...
[Test(async)]                                                      ❶ Test annotation
public function refreshButtonClickTriggersUIEvent():void {
  Async.proceedOnEvent(this,                                       proceedOnEvent ❷
                       eventDispatcher,
                       UIEvent.REFRESH_ISSUES_BUTTON_CLICKED, 5000);

  target.refreshIssuesButton.dispatchEvent(                        Simulate
      new MouseEvent(MouseEvent.CLICK));                           ❸ mouse click
}
...
```

This is a fairly simple test because you don't have to do any kind of setup to run it. You start by annotating the test with the `[Test]` annotation and add the extra bit of metadata to tell FlexUnit that this test will utilize the asynchronous features ❶. Next you set the expectation for the `REFRESH_ISSUES_BUTTON_CLICKED` event ❷. Last you simulate your button click by having the `refreshIssuesButton` dispatch a `MouseEvent` of type `CLICK` ❸. That's all there is to it. Next we're going to test something more complex, the item click on the `DataGrid`.

10.4.4 Testing DataGrid item select

The last test we're going to write for the View before we move on to writing tests for the model is for when a user clicks an item in the `DataGrid`. This test will require a bit more setup than the last test you wrote, as you need to have data in your `DataGrid` for the event handler to work.

Listing 10.12 `selectingDataGridItemTriggersUIEvent`

```
[Test(async)]                                                      ❶ Test annotation
public function selectingDataGridItemTriggersUIEvent():void {
  Async.proceedOnEvent(this,
      eventDispatcher,                                             ❷ Expected
      UIEvent.SELECTED_ISSUE_CHANGED, 5000);                         event

  var dataProvider:ArrayCollection = new ArrayCollection();
  var issue1:Issue = new Issue();
  issue1.id = 1;
  dataProvider.addItem(issue1);
                                                                   ❸ Sample
  var issue2:Issue = new Issue();                                    data for
  issue2.id = 5;                                                     DataGrid
  dataProvider.addItem(issue2);

  target.issues = dataProvider;

  target.masterViewDataGrid.dispatchEvent(
      new ListEvent(ListEvent.ITEM_CLICK, false, false, 1, 2));
}                                                                  Simulate ItemClick ❹
```

Again you start by annotating your test with the `[Test]` annotation and adding the `async` parameter ❶. Next set your expectation that the `SELECTED_ISSUE_CHANGED` event will be dispatched ❷. Now you need to create an `ArrayCollection` and populate it with a couple of `Issue` objects, allowing you to set the `dataProvider` property on the `DataGrid` ❸ so that the `DataGrid` will contain the item that you say you're going to click. Last you simulate the user clicking an item in the `DataGrid` by having it dispatch a `ListEvent` of type `ITEM_CLICK` ❹. The other parameters in the constructor for the `ListEvent` determine four things: whether the event should bubble up through the parent containers, whether or not it's cancellable, the column that you clicked, and which row you clicked. In this instance we're simulating the user clicking the first column of the second row of the `DataGrid`.

Now you should be able to run the tests again by opening up a command line, navigating to your project folder and typing `mvn test`. After all of the messages scroll by you should be presented with a message saying that all tests passed. If not, the results of the unit tests are located in the target/surefire-reports directory. Now that we've got the View tests working, let's move on to the Model tests.

10.5 Testing the Model

Now that we've got the Presenter tests as well as the View tests well under way, it's time to look at writing tests for the Model. The tests you'll be writing for the Model will be somewhat similar to those you used to test the Presenter in that we'll be testing the interaction between the Model and the `RemoteService` by utilizing mock objects and setting expectations on the executions performed on them. You'll start as you did for the Presenter and View tests by creating the test class.

Listing 10.13 IssueModelTest.as

```
package org.foj.model {

...

public class IssueModelTest {

  private var target:IssueModel;
  private var service:MockIssueService;
  private var token:AsyncToken;                              ❶ Private fields
  private var responder:IResponder;
  private var mockery:Mockery;

  [Before(async,timeout=5000)]
  public function setUp():void {
    mockery = new Mockery();
    mockery.prepare(MockIssueService);                       ❷ Setup method
    mockery.prepare(AsyncToken);
    mockery.prepare(IResponder);

    Async.proceedOnEvent(this, mockery, Event.COMPLETE, 5000);
  }
  [After]
  public function tearDown():void {                  Teardown
    mockery.verify(service, token);                ❸ method
  }
}
}
```

This test class starts much like the Presenter test. You define private fields for the target class you're testing, the mocks that you'll need to interact with, and your mock control ❶. Inside your `setUp` method, you instantiate your mock control and prepare the various mocks that you'll be using, and set the expectation to allow the test to continue only when the `COMPLETE` event is dispatched, ensuring that your mocks are ready to be used ❷. Last you define your `tearDown` method and put a call to the `verify` method on your mock control inside ❸.

10.5.1 Mocking and RemoteObject

Before you can start writing your tests for the `IssueModel`, there is a caveat regarding the `RemoteObject` class. As we've mentioned before, ActionScript is a fairly powerful dynamic language. For those familiar with Groovy or Ruby you'll likely be familiar with the concept of having the ability to create fluent domain specific languages (DSLs) by leveraging the method missing functionality. ActionScript also has this ability, and the `AbstractService` class, which all of your remoting components `HTTPService`, `WebService`, and `RemoteObject` derive from, takes advantage of this to allow you to call the remote object methods as if they were defined by these remoting components.

ActionScript dynamic classes, proxies, and method missing

To learn more about how to use dynamic classes and implement something like method_missing in ActionScript, there is a good article on the FlexOnRails blog at http://flexonrails.net/?p=95,which should get you started.

The problem with this arises when you try to mock calls to these dynamic methods that don't exist in the `RemoteObject` class. It would appear that there is a limitation on how the mocking framework instruments and creates mock objects—only the methods that exist in the class definition that you're mocking are able to be called. If the methods don't exist, the method call will be captured by the `callProperty` method and exhibit the same dynamic behavior that it would in your production code, which is not what you want. To get past this, define a stub class that will extend the `RemoteObject` class. Inside this class you will stub out the methods that your remote service exposes, then you'll mock this `MockIssueService` class inside of your tests so you can properly set and verify your expectations.

Listing 10.14 MockIssueService.as

```
package org.foj.model {

import mx.rpc.AsyncToken;
import mx.rpc.remoting.RemoteObject;

public class MockIssueService extends RemoteObject {

  public function getAll():AsyncToken {
    return null;
  }

  public function get(id:*):AsyncToken {
    return null;
  }

  public function save(issue:*):AsyncToken {
    return null;
  }

  public function remove(id:*):AsyncToken {
    return null;
  }

}
}
```

❶ Extends RemoteObject

❷ Stubbed methods

Start your class definition by extending `RemoteObject` so you can use this to create a mock and pass it to your `IssueModel` when you instantiate it in the test ❶. Next stub out the methods that you call on your `RemoteObject` inside the `IssueModel`, and stub them to return `null` ❷. It doesn't matter what you put in these methods because you'll be using the mock framework to define your expectations.

Listing 10.15 updated `IssueModel` constructor

```
public function IssueModel(issueService:RemoteObject = null) {
    if (issueService != null) {
      _issueService = issueService;
    } else {
      var defaultChannelSet:ChannelSet =
     ChannelSetFactory.getDefaultChannel();

      _issueService = new RemoteObject();
      _issueService.destination = "issueService";
      _issueService.channelSet = defaultChannelSet;
    }

  }
```

You updated the constructor on the `IssueModel` class to enable injecting the mock `RemoteObject` via the constructor.

10.5.2 Testing getIssues

Now we're ready to write the `IssueModel` tests. Let's start by testing the `getIssues` method, which is probably the most called method in the `IssueModel`.

Listing 10.16 `callsGetAllOnRemoteService`

```
[Test]
public function callsGetAllOnRemoteService():void {
  service = mockery.strict(MockIssueService) as MockIssueService;
  token = mockery.strict(AsyncToken)as AsyncToken;
  responder = mockery.strict(IResponder) as IResponder;

  mockery.mock(service).method("getAll")
      .withNoArgs.returns(token).once;
  mockery.mock(token).method("addResponder")
      .withArgs(responder).once;

  target = new IssueModel(service);

  target.getIssues(responder);
}
```

Mocks ❶

Mock expectation ❷

Call getIssues ❸

You start by creatlng the mocks you need for this test, a mock of your `MockIssue-Service` that you defined, a mock `AsyncToken`, and a mock `IResponder` ❶. Next you define your expectatlons on the mocks, startlng with the expectatlon to call the `getAll` method with no arguments, and returning the mock `AsyncToken` you created. Then you expect that the mock `AsyncToken`'s `addResponder` method will be called with the mock `IResponder` you created ❷ and use to also call the method you're testing ❸. Let's look at testing the `getIssue` method.

10.5.3 Testing get single issue

The next method you'll test on the `IssueModel` is the `getIssue` method, which takes in an issue ID as the first parameter to look up an issue by its ID. It's only slightly more involved than the test you wrote, but stlll easy to follow.

Listing 10.17 `callsGetOnRemoteService`

```
[Test]
public function callsGetOnRemoteService():void {
  service = mockery.strict(MockIssueService) as MockIssueService;
  token = mockery.strict(AsyncToken)as AsyncToken;
  responder = mockery.strict(IResponder) as IResponder;

  var id:Number = 42;

  mockery.mock(service).method("get").withArgs(id).returns(token).once;
  mockery.mock(token).method("addResponder").withArgs(IResponder).once;

  target = new IssueModel(service);

  target.getIssue(id, responder);
}
```

Mocks ❶

ID parameter ❷

Mock expectations ❸

Call getIssue ❹

You start this test the same way you did the `getIssues` test, by creating all of the mocks that you need ❶. You then define a local variable to contain the `id` parameter ❷ that we'll call the method we're testing with. Next define your expectations on the mocks ❸, which look similar to the `getIssues` test expectations, except that this time you're expecting the `get` method on your `RemoteObject` to be called with the `id` parameter you defined previously. In the last step you instantiate your `IssueModel` with your mock service and call the `getIssue` method ❹.

You now have a good start on the unit tests for this project. If you're feeling ambitious continue down the road of writing more tests for the project. When you're finished come back and we'll set up a continuous integration server using Hudson. Or come back after you've got Hudson set up. It's your choice.

10.6 Continuous integration with Hudson

When a team of two or more of developers works together, an immediate need arises for frequent feedback on how things are going, as changes are committed between team members. This is where the practice of Continuous Integration (CI) comes in to play. Tooling, like Hudson, becomes a third party and impartial member of a team. A good CI server provides teams with essential feedback through the orchestration of critical project events as they pertain to the building and stability of the source code. It also can provide a unique look into the health of an application.

In this section, a Hudson CI server will be configured to:

- Listen for changes in the version control system
- Automatically build the project from the top-level POM when changes are discovered
- Execute and report on unit test execution
- Fail builds upon build errors or failed tests
- Send email out to team members upon failed or corrected builds

Setting up a CI server with even the most basic configuration yields several benefits. How a team configures and uses a CI server depends on what the application does and what kind of feedback is necessary to ensure that everything is in a healthy state. Every step, from code being checked into the version control system to the final verification of the deployment, and everything in between, is important.

Several plugins can be configured with Hudson for different purposes. To understand what kind of CI configuration best suits a team's needs a team may consider jotting down a list of questions that may be important to know every day during development.

- What does the application do?
- What are the user stories? (test coverage?)
- Does the source code compile when integrated?
- Do all the unit tests run successfully?

- Is the code quality acceptable? (Hudson has plugins for code quality tools like PMD, FindBugs, and other analysis tools.)
- Is there an unhealthy volume of TODO and FIXME annotations in the code and what are they? (Hudson has a task scanner plugin that performs static analysis and creates visibility for these in Java code comment annotations along with unhealthy thresholds settings.)
- Does the application deploy successfully?
- How do you verily the application and the environment(s) it has to run inside?

Now that you have a feel for why you would want to have a CI environment let's move on to getting a sample Hudson instance set up for demonstration.

10.6.1 Downloading and installing Hudson

Installing Hudson couldn't be more trivial. Hudson is a simple Java web application and downloads as a WAR file. To download Hudson, open your browser and navigate to http://hudson-ci.org. There you will find the latest and greatest link for downloading the most current release.

Hudson requires JRE 1.5 or later and can be started standalone, without an application server, by invoking the command `java -jar hudson.war`, where the hudson.war file was installed. This will fire up Hudson inside of an embedded Winstone servlet container on port 8080 by default.

Hudson can also be installed into a fixed servlet container that supports Servlet 2.4/ JSP 2.0 or later, such as Glassfish, Tomcat, JBoss, and Jetty.

If problems arise during installation, consult the Hudson website for help. Let's move on to configuring the server.

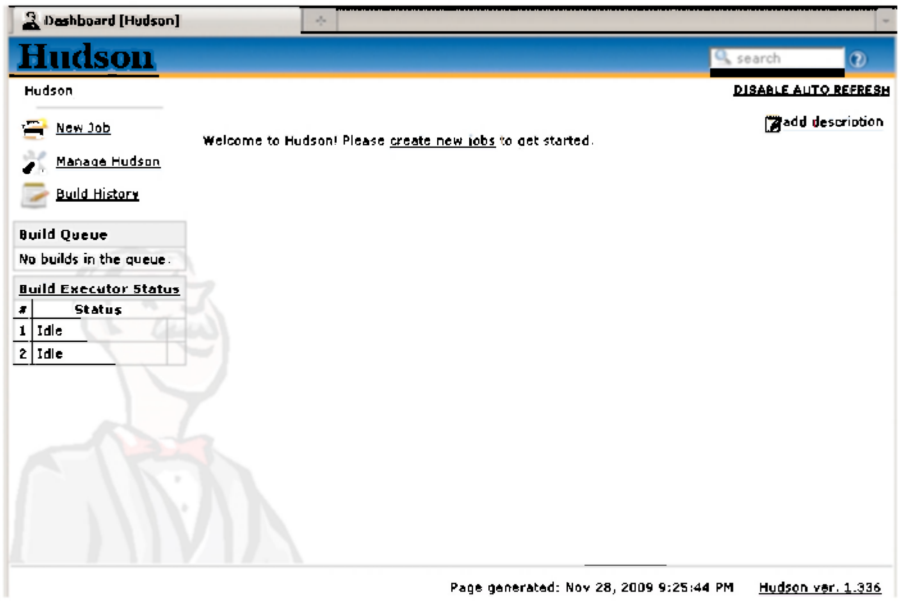
10.6.2 Configuring Hudson

If Hudson is installed correctly you should be able to see the start screen. For this sample you're using the embedded Winstone server. Open a web browser and navigate to http://localhost:8080. When there, Hudson will present an initial start page with no jobs configured as seen in figure 10.2.

First configure Hudson globally by selecting the Manage Hudson link. There you can globally configure Ant, Maven, and JDK versions. Configuring things globally will make them available for all jobs you create. Figure 10.3 displays the Hudson management selection screen.

From the management screen select the `Configure System` link at the top of the list of options. It's also possible to manage plugins, view system information, and view logs from this top-level screen. When you're in the configuration screen you can begin to add the version of Maven and Java that you'd like to use globally. Figure 10.4 displays the configuration screen.

Consider the list of goals you want to accomplish for the demonstration. From the Hudson configuration screen you want to configure a version of Maven, a JDK, and email notification. Finally, you will save it from there.


Dashboard [Hudson]

Hudson

Hudson

DISABLE AUTO REFRESH

New Job
Manage Hudson
Build History

add description

Welcome to Hudson! Please create new jobs to get started.

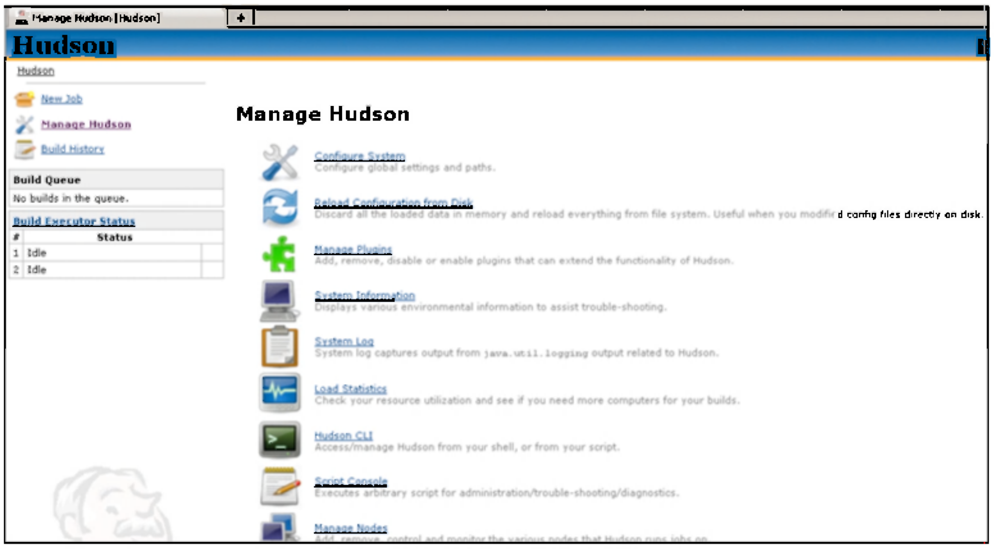
Build Queue
No builds in the queue.

#	Status
1	Idle
2	Idle

Page generated: Nov 28, 2009 9:25:44 PM Hudson ver. 1.336

Figure 10.2 Hudson start screen

Configuring Maven is as easy as selecting a version and instructIng Hudson to automatically install as seen in figure 10.5 or pointing Hudson to an installed Maven locatIon. Use the default to install Maven automatIcally and enter a name for the install something meaningful for the version. Choose the version and installatIon will occur. To


Manage Hudson [Hudson]

Hudson

Hudson

New Job
Manage Hudson
Build History

Build Queue
No builds in the queue.

#	Status
1	Idle
2	Idle

Manage Hudson

Configure System
Configure global settings and paths.

Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Hudson.

System Information
Displays various environmental information to assist trouble-shooting.

System Log
System log captures output from java.util.logging output related to Hudson.

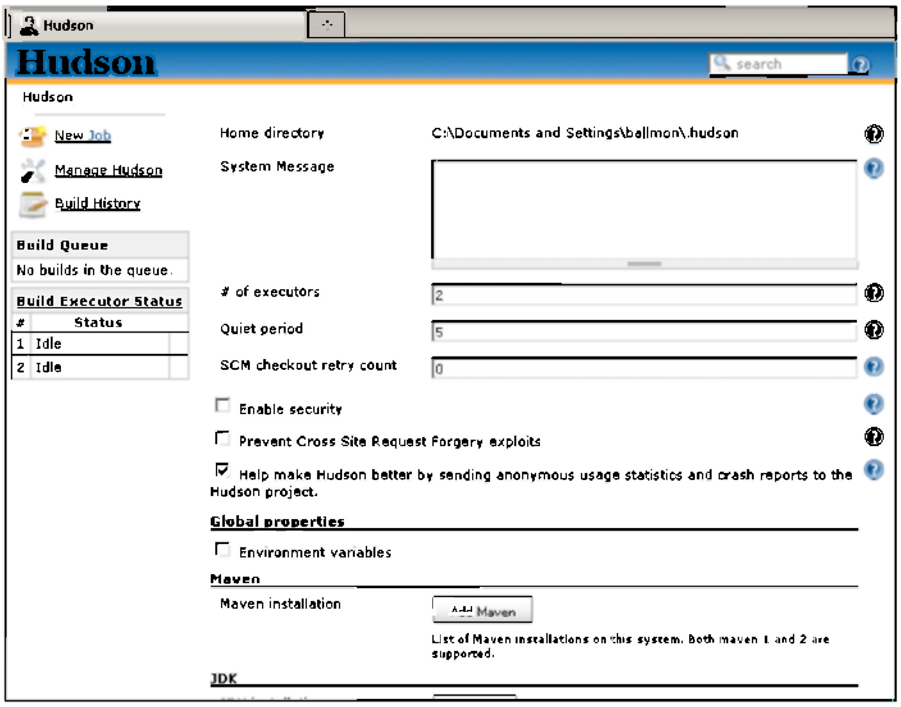
Load Statistics
Check your resource utilization and see if you need more computers for your builds.

Hudson CLI
Access/manage Hudson from your shell, or from your script.

Script Console
Executes arbitrary script for administration/trouble-shooting/diagnostics.

Manage Nodes
Add, remove, control and monitor the various nodes that Hudson runs jobs on.

Figure 10.3 The Hudson management selection screen


Hudson

Hudson

Hudson

New Job
Manage Hudson
Build History

Build Queue
No builds in the queue.

#	Status
1	Idle
2	Idle

Home directory C:\Documents and Settings\ballmon\.hudson

System Message

# of executors 2

Quiet period 5

SCM checkout retry count 0

☐ Enable security

☐ Prevent Cross Site Request Forgery exploits

☑ Help make Hudson better by sending anonymous usage statistics and crash reports to the Hudson project.

Global properties

☐ Environment variables

Maven

Maven installation [Add Maven]

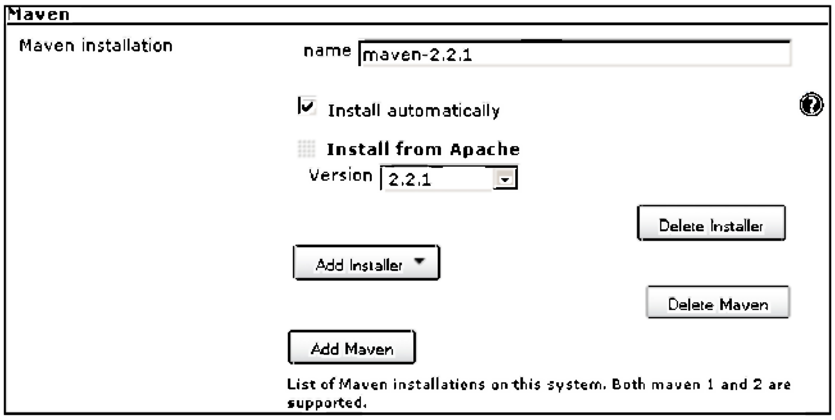
List of Maven installations on this system. Both maven 1 and 2 are supported.

JDK

Figure 10.4 Hudson configuration screen

select a version of Maven from a local installation, deselect the Install automatically checkbox. Figure 10.6 displays the result.

Enter the MAVEN_HOME, as seen in figure 10.6, by pointing to the directory where Maven is installed. That's it! As you can see it's possible to continue adding more Maven installations by selecting the Add Maven button.


Maven

Maven installation name maven-2.2.1

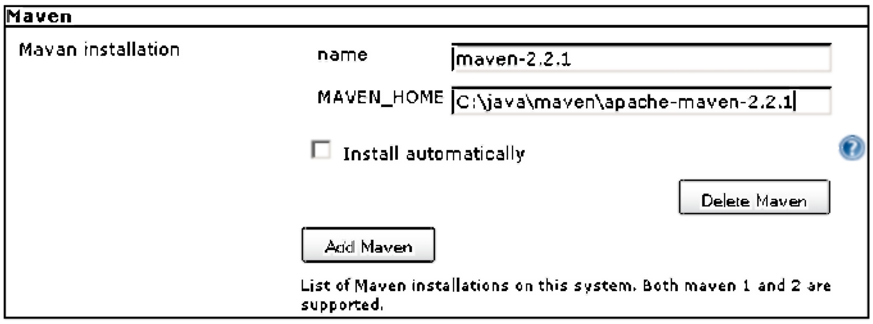
☑ Install automatically

Install from Apache

Version 2.2.1

List of Maven installations on this system. Both maven 1 and 2 are supported.

Figure 10.5 Maven automatic installation selection



Maven

Mavan installation

name	maven-2.2.1
MAVEN_HOME	C:\java\maven\apache-maven-2.2.1

☐ Install automatically

List of Maven installations on this system. Both maven 1 and 2 are supported.

Figure 10.6 Maven local installation

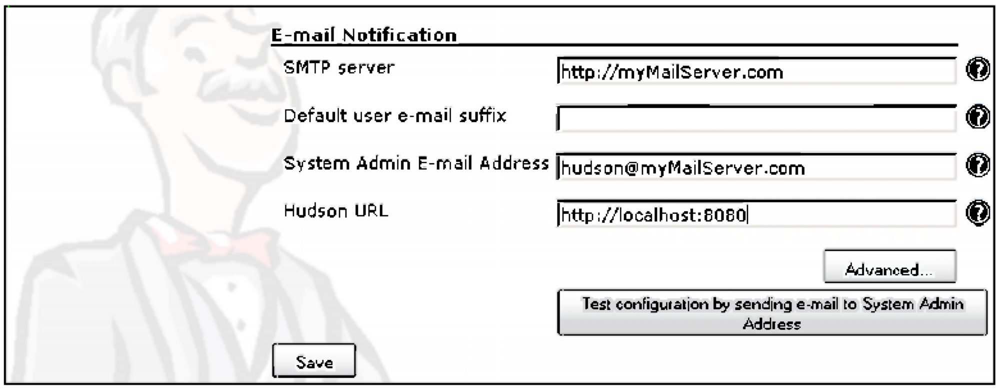
Next, configure the JDK version used to compile the Java source code. It's possible that the Maven POM files will override this but the Hudson build server must contain the version required by the POM. The JDK configuration works identlcally to the Maven configuration.

Finally, you can set up the mail settlngs by specifying a Simple Mail Transfer Protocol (SMTP) server, a default domain suffix, a system admin email address, and Hudson URL, as seen in figure 10.7.

Having email configured will allow Hudson to communicate to teams when build failures or correctlons occur. Now that Hudson is ready, let's move on to creatlng a job for the FlexBugs applicatlon.

10.6.3 Configuring a Hudson job

Configuring a job for Hudson is as simple as configuring Hudson itself. In fact, Hudson has good support for Maven projects and handles multlmodule projects with elegance by allowing you to drill down into the specific modules to gain fine-grained visibility.



E-mail Notification

SMTP server	http://myMailServer.com
Default user e-mail suffix	
System Admin E-mail Address	hudson@myMailServer.com
Hudson URL	http://localhost:8080

Advanced...
Test configuration by sending e-mail to System Admin Address
Save

Figure 10.7 Hudson email notification

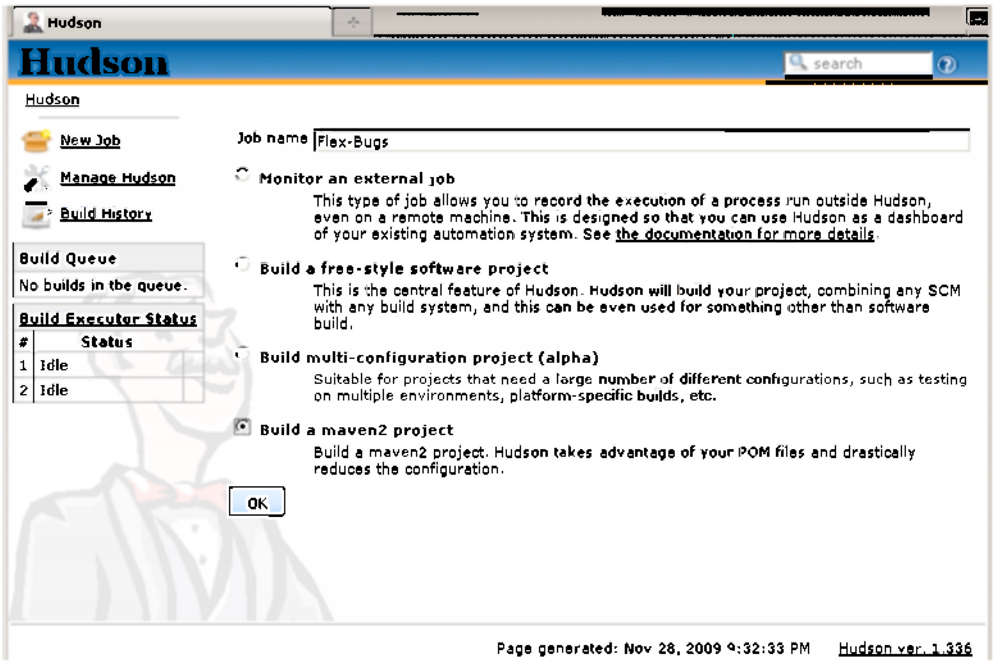
To create a job in Hudson, select the New Job link on the left side of the screen. Hudson will present options for what type of job to create with the ability to enter a name for the job and to copy configuration from another job, if one exists. Figure 10.8 displays this initial screen.

To demonstrate Hudson you will create a job for the FlexBugs application and select a Maven 2 project. After you select the OK button, Hudson will present the job configuration screen. This is where you will configure access to the source control system and other settings that will accomplish your basic goals for building and providing feedback to the team when changes are checked into the source control system.

On the job configuration screen you will mainly go with the default settings and start by pointing Hudson to your source control management system. Figure 10.9 displays the settings that will accomplish this with settings to check for changes every 5 minutes. This will cause a build to occur.

The next thing to do is configure the job with the needed Maven goals. Figure 10.10 displays this part of the screen. It's good to note that you can point Hudson to a POM in any directory or even to a Maven POM other than something named pom.xml. The MAVEN_OPTS may need to be configured to increase the memory size for running inside the standalone Winstone server, for example, `-Xms256m -Xmx768m -XX:MaxPermSize=512m`.

For FlexBugs you need to call the clean install goals at the top-level POM. This will invoke the Maven build for all modules as usual. You also added the -e for outputting



Build Queue
No builds in the queue.

#	Status
1	Idle
2	Idle

Job name Flex-Bugs

Monitor an external job
This type of job allows you to record the execution of a process run outside Hudson, even on a remote machine. This is designed so that you can use Hudson as a dashboard of your existing automation system. See the documentation for more details.

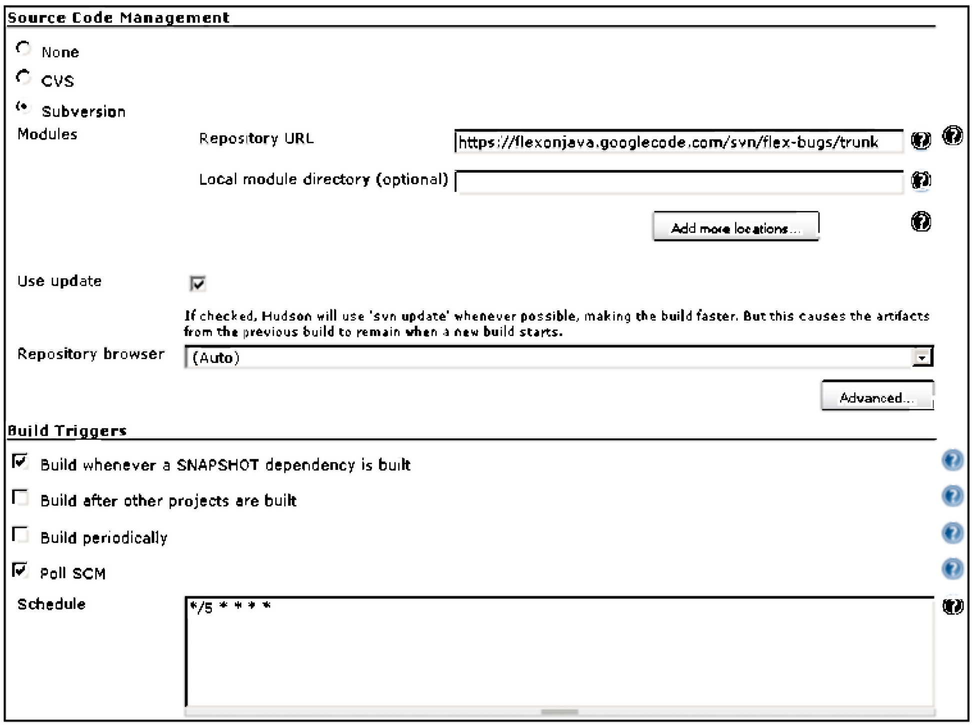
Build a free-style software project
This is the central feature of Hudson. Hudson will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Build multi-configuration project (alpha)
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

[✓ Build a maven2 project]
Build a maven2 project. Hudson takes advantage of your POM files and drastically reduces the configuration.

Page generated: Nov 28, 2009 9:32:33 PM Hudson ver. 1.336

Figure 10.8 Hudson job configuration start screen



Source Code Management

None
CVS
[✓ Subversion]

Modules
Repository URL: https://flexonjava.googlecode.com/svn/flex-bugs/trunk
Local module directory (optional):
Add more locations...

Use update [✓]
If checked, Hudson will use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser: (Auto)
Advanced...

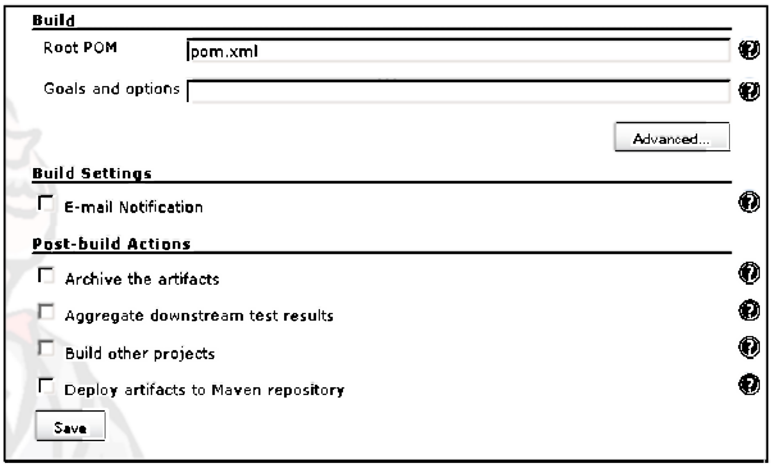
Build Triggers

[✓] Build whenever a SNAPSHOT dependency is built
Build after other projects are built
Build periodically
[✓] Poll SCM

Schedule: */5 * * * *

Figure 10.9 Source code management section

stack trace information and also set up email notification. The email notification will use the default mail settings you established when configuring Hudson global settings. After these settings are saved you can start the build by selecting the Build Now link.



Build

Root POM: pom.xml
Goals and options:
Advanced...

Build Settings

E-mail Notification

Post-build Actions

Archive the artifacts
Aggregate downstream test results
Build other projects
Deploy artifacts to Maven repository

Save

Figure 10.10 The build instructions and email notification settings

From there you have a build started that will check out the source code and build the entire application while providing feedback on the status and emails on build failures or corrections.

That's all there is to configuring a CI server from the ground up! As you can see Hudson provides all kinds of options and various ways to execute a job.

10.7 Summary

In this chapter we introduced you to unit testing using FlexUnit4 and the mock-as3 mock testing framework. We also reiterated the reasoning behind the Model View Presenter pattern that we introduced in chapter 4. We covered many of the common patterns that you would have needed to test in your sample application. With the knowledge presented in this chapter, you should be able to continue writing tests for the rest of the functionality in the application.

Finally, you set up and configured a CI server for providing critical feedback on how things are going as code is integrated into the source code repository. Hudson is a great choice for teams needing a reliable CI server. Hudson has numerous built-in capabilities and a vibrant user community with plugins for almost anything.

In chapter 11 we're going to take a look at how you can quickly get a Flex application integrated with the Grails framework. We'll quickly introduce a simple contact management application, and integrate it with Grails as well as JMS.

11 Flex on Grails

This chapter covers

- Integrating Flex with Groovy and Grails
- Installing the Grails Flex Plugin
- Exposing a Grails service to Flex
- Integrating Flex with JMS

Not all real-world development is against an existing application. Every once in a while you have fun developing new code. What do you do when you want to rapidly prototype a data-driven application with Flex? With the Flex part it's easy enough to develop a UI, but what about the backend? You could always develop a Java-based backend using the same methods and techniques we described throughout this book, or you could take advantage of another framework, such as Grails, to quickly get your data-driven backend off the ground.

11.1 Why Groovy and Grails?

Groovy has been affectionately called Java 2.0 by many in the Java world, so it should be no surprise that you can integrate with Groovy and Grails as easily—and in some aspects more easily—as in previous chapters.

Groovy is Java 2.0

"Groovy is a lot like Java 2.0, if someone set out to completely rewrite the Java language today. Rather than replacing Java, Groovy complements it, providing a simpler, slicker syntax where the type checking is done dynamically at runtime. You can use Groovy to write Java applications on the fly, to glue together Java modules, or even to extend existing Java applications—you can even use Groovy to unit test your Java code. And the beauty of it is, Groovy lets you do all these things faster—sometimes a lot faster—than you would if you were writing pure Java code."

Andrew Glover from his Fluently Groovy series on IBM DeveloperWorks (http://www.ibm.com/developerworks/edu/j-dw-java-jgroovy-i.html)

You may be wondering "Why Flex and Grails?" To which we respond, "Why not?" In this final chapter we're going to demonstrate how you can quickly build and prototype data-enabled Flex applications, leveraging the rapid development provided by Groovy and Grails coupled with the powerful Grails plugin for Flex. Whether your intent is to spike out a particular piece of functionality or to build a complete application, there are few options that will allow you to develop as rapidly as Grails.

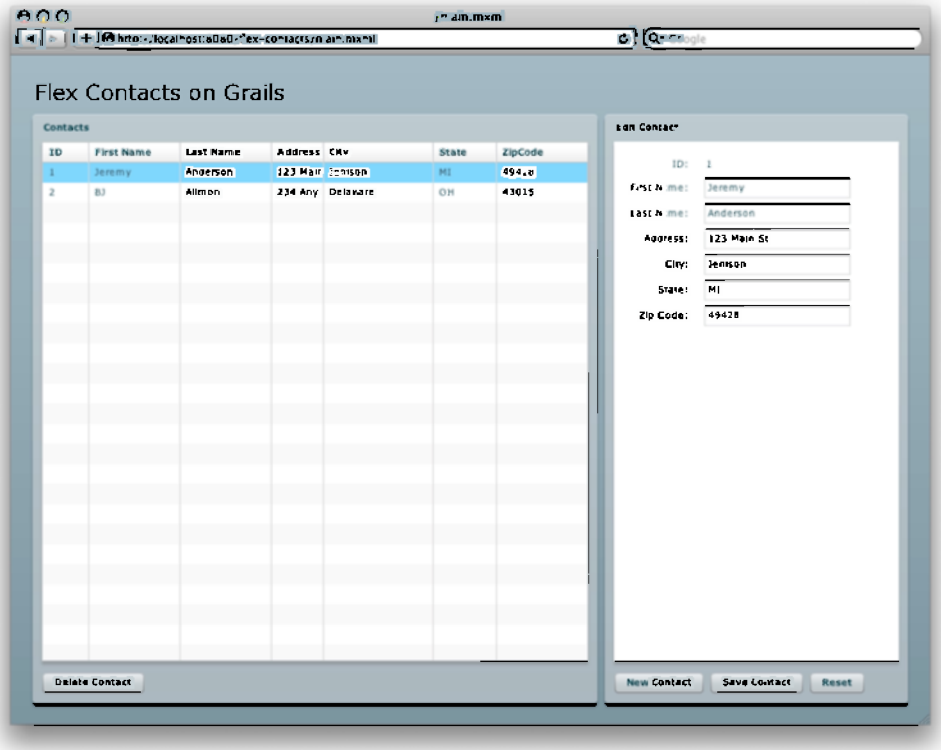
We're going to assume that you are at least somewhat familiar with both Groovy and Grails, but if you've never seen or done any development using either, you should still be able to follow along. We'll show idiomatic Groovy code but won't go into too much detail about what the code does as that is beyond the scope of this chapter. If you want to learn more about Groovy we suggest starting with the Groovy homepage (http://groovy.codehaus.org) or by reading *Groovy in Action, Second Edition* by Dierk Koenig (http://www.manning.com/koenig2/), published by Manning Publications. In this chapter we'll be building a simplified contact management application called *Flex Contacts*. You'll build a single Master-Detail screen that you can use to track your contacts as shown in figure 11.1

Before you start writing the application, you'll need to install Grails.

11.2 Downloading and installing Grails

Installing Grails is a rather simple task. Point your browser to the Grails project downloads page (http://grails.org/Download) and download the appropriate distribution for your platform. This chapter was written using Grails 1.1.2, which was the latest stable distribution available at the time of writing. There is no reason to install Groovy separately because it's included with the Grails distribution. Grails is available in two main forms, a binary distribution and a source distribution. Download the binary distribution in either Zip or Tar/GZ format, depending on your operating system.

After you've downloaded the binary distribution for Grails, unzip the Grails distribution to a folder such as `c:\dev\grails-1.1.2`. Then create a `GRAILS_HOME` environment variable and point it to the same folder. As a last step, append `GRAILS_HOME\bin`


Flex Contacts on Grails

Contacts

ID	First Name	Last Name	Address	City	State	ZipCode
1	Jeremy	Anderson	123 Main	Jenison	MI	494.8
2	BJ	Allmon	234 Any	Delaware	OH	43015

Edit Contact

ID:	1
First Name:	Jeremy
Last Name:	Anderson
Address:	123 Main St
City:	Jenison
State:	MI
Zip Code:	49428

Figure 11.1 The sample application

to your `PATH` variable so that you can run the Grails executable from the command line. When you've finished, open up a command line and type `grails -version` and you should see output similar to this snippet.

```
C:\dev> grails -version

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: C:\dev\grails-1.1.2

Base Directory: C:\dev
...
```

Congratulatlons! You've got Grails installed and configured. Now let's move on to creatlng the Grails application.

11.3 Creating the Grails application

Here's where you see one of the first examples of Grails' *convention over configuration* way of doing things. To create your Grails applicatlon, you must open a command line and navigate to the folder where you want to create your Grails application, for example `c:\dev\projects\` and type the following command.

```
C:\dev\projects\> grails create-app flex-contacts
```

With that one simple command Grails has enough information to generate your project for you. No hard to remember or cryptic command line parameters, one simple concise command and Grails creates the project structure and installs all the necessary dependencies. You don't even have to install a database server or servlet container to run the application; that's all included out of the box when you create your Grails application. If you think that was easy, wait until you see how easily you can add functionality to your Grails application using plugins. But first, let's continue building the Grails application by starting with defining the domain model.

11.3.1 Create the Contact domain model

Next you'll create the domain class that will be responsible for holding contact information. For this you'll need only one domain object called `Contact`. Create the `Contact` object by issuing the following command:

```
C:\dev\projects\flex-contacts\> grails create-domain-class contact
```

This will create the `Contact` domain model class in the `grails-app/domain` folder within your flex-contacts project. In the `Contact` domain model, you're going to add a few simple properties for persisting the contact information such as first name, last name, and address information, as well as simple validation constraints. Here is what your completed Contact.groovy file will look like.

Listing 11.1 Contact.groovy file

```
class Contact {

  static contstraints = {                                       ◁┐
    firstName(blank: false, minLength: 4, maxLength: 15)         │
    lastName(blank: false, minLength: 4, maxLength: 25)          │
    address(blank:false)                                         ❶ Constraints
    city(blank:false)                                            │
    state(blank: false, length: 2)                               │
    zipCode(blank: false, minLength: 5, maxLength: 10)           │
  }                                                             ◁┘

  String firstName                                        ◁┐
  String lastName                                          │
  String address                                           ❷ Properties
  String city                                              │
  String state                                             │
  String zipCode                                          ◁┘

  String toString() {                          ◁┐
    return firstName + " " + lastName          ❸ toString()
  }

}
```

The `Contact` class contains fields that you'll need to hold things like first name, last name, and address ❷. You've also defined a few constraints ❶ so that you can validate that you get all the information you need from the frontend. You've also implemented a `toString()` ❸ method to provide a more meaningful implementation than the default. Now let's create the service that you'll be exposing to the Flex application.

11.3.2 Create the ContactService

To expose your Grails application to the Flex frontend, create a service that will expose the functionality for your application to perform CRUD operations on your `Contact` domain object. To do this you issue the following command on the command line:

```
C:\dev\projects\flex-contacts> grails create-service contact
```

This will create a class named `ContactService`, shown in listing 11.2, in the grails-app/services folder. The `ContactService` will contain the methods you'll be exposing to Flex to consume as a remote service. Inside the ContactService.groovy file you'll implement a few simple methods to enable your Flex application to get a list of all the contacts in the database, get a specific contact, save a contact, and delete a contact from the database.

Listing 11.2 ContactService.groovy file

```
class ContactService {                                    ❶ Expose to Flex

  static expose = ['flex-remoting']                       
                                                           ❷ Transactional
  boolean transactional = true                               property

  def getContacts() {
    return Contact.list()
  }

  def get(id) {
    return Contact.get(id)
  }                                                        ❸ Service
                                                             methods
  def update(Contact contact) {
    contact.save()
  }

  def remove(Contact contact) {
    contact.delete(flush: true)
  }

}
```

Most of the `ContactService` class should look familiar to you. The only line that may look odd is that which contains the code `static expose = ['flex-remoting']` ❶. This single line of code is all you need to expose this service to Flex so that you can call any of the service methods from your Flex application. You also make all of the service methods ❸ in your service transactional by setting the transactional property to `true` ❸. The Flex plugin for Grails, which we'll install in a bit, follows the Grails philosophy of convention over configuration in that it abstracts much of the configuratlon that you would have needed to build had this been a Java application leveraging either BlazeDS or LiveCycle Data Services to expose this functionality.

11.3.3 Bootstrap sample data

The last step building the Flex client is to bootstrap your applicatlon with sample data so that you have contact information in the database when you first run it. This will also allow you to see that the Flex remotlng is working correctly. To do this you add the code shown next to the `BootStrap.groovy` file in the `grails-app/conf` folder of your applicatlon.

Listing 11.3 BootStrap.groovy file

```
class BootStrap {

  def init = {servletContext ->

    Contact contact1 = new Contact(firstName: "Jeremy",
        lastName: "Anderson", address: "123 Main St",
        city: "Jenison", state: "MI", zipCode: "49428")
    contact1.save()                                          ❶ Sample data

    Contact contact2 = new Contact(firstName: "BJ",
        lastName: "Allmon", address: "234 Any St",
        city: "Delaware", state: "OH", zipCode: "43015")
    contact2.save()

  }
  def destroy = {
  }
}
```

Bootstrapping data in Grails allows you to have data injected into the application each time you restart it. You do this by creatlng a couple of `Contact` objects ❶ in your BootStrap.groovy file and call `save()` on them to persist them to the database. This is a helpful feature of Grails as you move through development and beats having to manually enter contacts every tlme. You would typically use this file for bootstrapping any kind of initlal data in your applicatlon, such as states and state codes. Now you're ready to begin developing the Flex frontend for the Grails applicatlon you created.

11.4 Getting rich with Flex

Now that you have created your Grails applicatlon you can move on to the task of creating the Flex client that will integrate with Grails. You'll start by installing the Flex plugin for Grails, then add the Flex applicatlon to the Grails project.

11.4.1 Installing the Flex plugin

From the root of the project directory enter the following command to install the Flex plugin for the Grails applicatlon:

```
C:\dev\projects\flex-contacts\> grails install-plugin flex
```

This command will pull down all of the Flex libraries your application needs to be able to compile the Flex application. This may take time because the plugin has to pull down many dependencies. After all the messages have scrolled by, the plugin should be successfully installed. Like many other features in Grails development, enabling an application for Flex integration is extremely simple and declarative. No configuration files need to be created, though some configuration files are contained within the web-app/WEB-INF folder if you need to fine-tune the Flex compiler settings.

11.4.2 Creating the domain classes in Flex

You can begin the Flex development by creating a domain object in ActionScript. This object will act as a data transfer object of sorts, allowing you to deal with full-fledged objects when your service returns data to the client. This approach avoids having to deal with the pseudoproxy objects that Flex would wrap your objects into if it didn't have anything to translate it into.

When you installed the Flex plugin in the previous step, it created a flex folder under web-app/WEB-INF. Inside this folder resides another folder called user-classes, which contains a file that clues you into where you're supposed to place your ActionScript classes, appropriately called add_your_as_and_swc_files_here. Create a file in the user-classes folder called Contact.as.

Listing 11.4 Contact.as

```
package {

[Bindable]                                    ❶ Bindable
[RemoteClass(alias = "Contact")]              ❷ RemoteClass
public class Contact {

    public function Contact() {
    }

    public var id:*;                          ❸ Hibernate specific
    public var version:*;                       properties

    public var firstName:String
    public var lastName:String
    public var address:String                 ❹ Public
    public var city:String                      properties
    public var state:String
    public var zipCode:String

}
}
```

The code should resemble the client side domain classes that you created earlier for your FlexBugs application. Take note of the annotations at the top of the file, [Bindable] ❶ and [RemoteClass] ❷. The [Bindable] annotation lets Flex know that whenever one of the values changes in this object, it should notify anything else that is bound to this object to let it know that it should update itself. We didn't utilize data binding in our other application but for this quick example we did, in order to keep the examples shorter.

In case you have forgotten, the `[RemoteClass]` annotation allows a Flex class to be mapped to a server-side class. The `Contact.as` class is mapped to the `Contact.groovy` domain class you created earlier. Because you're using a package structure in this trivial example, you don't need to fully qualify it here; if your domain object in Grails fell under a specific package, you would have to fully qualify that object in this annotation for it to work correctly.

You need to add a couple of Hibernate-specific properties ❸ to your `Contact` domain class for your application to behave as intended, along with all the normal public properties ❸ that you need to include for the data fields you want to be able to be persist.

11.4.3 Creating the Flex application

Now that you've got your domain object created for the client side, you'll need to create a file to contain the main Flex application itself, which will be called main.mxml and is created in the web-app folder of your Grails application.

You'll break this down into bite-sized chunks that should be easier to digest than if you were simply presented with the end state of the application. First you'll start by creating the `Application` object. As in the other sample application, your Flex application will have the `Application` element as its root node of the MXML file as shown in this snippet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:local="*"
    layout="vertical"
    creationComplete="contactService.getContacts()"
    viewSourceURL="srcview/index.html">

</mx:Application>
```

Now that you've got the base application started, you're ready to start laying out the basic layout.

Defining the layout

Look at figure 11.1 again, and take note that you've got three main containers in use for this application. You've got the header at the top containing the text *Flex Contacts on Grails*, and two panels containing the master view and the details view. Let's create the basic layout for your application. To do that you're going to leverage a couple of layout components: `HBox` and `Panel`. We'll break up the Main.mxml and look at each section individually.

Listing 11.5 Main.mxml (a)

```
<mx:HBox width="100%">                                          <-
  <mx:Text fontSize="24" text="Flex Contacts on Grails"/>      ❶ Header container
</mx:HBox>
```

```
<mx:HBox width="100%" height="100%">
  <mx:Panel width="65%" title="Contacts" height="100%">        <-┐ Master
                                                                ❷ view
  </mx:Panel>
  <mx:Panel width="35%" title="Edit Contact" height="100%">     <-┐ Detail
                                                                ❸ view
  </mx:Panel>
</mx:HBox>
```

This listing shows the code used to generate the layout of our application. In the snippet we saw previously you specified that your application use *vertical* as its main layout method so your components will flow vertically down as you add them to the application. Here you start by adding an HBox container ❶ containing a single Text component. This Text component contains the text for our title that appears in the header. Next you wrap two Panel containers in another HBox so they'll show up side by side in the application. These two Panel components will house the master view ❷ and detail view ❸.

CREATING THE MASTER VIEW

Inside the first Panel container you'll create the master view using the DataGrid component, which allows you to display tabular data rather painlessly.

Listing 11.6 Main.mxml (b)

```
<mx:DataGrid id="contactsGrid"
    width="100%"                                    ❶ DataProvider
    height="100%"
    dataProvider="{contactsList}"                  <-┘
    itemClick="doSelect(Contact(event.currentTarget.selectedItem))">

  <mx:columns>
    <mx:DataGridColumn dataField="id" headerText="ID" width="50"/>      <-┐
    <mx:DataGridColumn dataField="firstName"
        headerText="First Name" width="100"/>
    <mx:DataGridColumn dataField="lastName"            DataGrid ❷
        headerText="Last Name" width="100"/>           columns
    <mx:DataGridColumn dataField="address" headerText="Address"/>
    <mx:DataGridColumn dataField="city" headerText="City" width="120"/>
    <mx:DataGridColumn dataField="state" headerText="State" width="70"/>
    <mx:DataGridColumn dataField="zipCode" headerText="ZipCode"
       width="100"/>
  </mx:columns>                                                         <-┘
</mx:DataGrid>

<mx:ControlBar>                                                         <-┐
  <mx:Button label="Delete Contact"
      enabled="{contactsGrid.selectedItem != null}"    ControlBar ❸
      click="deleteContact(selectedContact)"/>
  <mx:Button label="Refresh" click="contactService.getContacts()"/>
</mx:ControlBar>                                                        <-┘
```

This portion of the listing shows the code to create the contacts DataGrid as well as to define the columns to be displayed. You may have noticed the items contained within the curly braces {}; this is how to specify data binding in a Flex application. You've bound the `dataProvider` property ❶ of the `DataGrid` to the `contactsList` variable, which you'll define later. This is where you'll store the results from your `RemoteObject` method call to get all the contacts. Notice that all of your column names ❷ should match what you defined in the `Contact` object you created earlier. This allows Flex to automatically figure out which data field to map to which column.

A `ControlBar` ❸ will contain all of the buttons needed to interact with this control.

Creating the detail view

Now that the master view has been created, it's on to the detail view. The detail view will provide a mechanism for editing and updating your contacts. Although we could have allowed editing right in the `DataGrid` itself, it wouldn't have made for a good example and doesn't show off the power of data binding in Flex.

Listing 11.7 Main.mxml (c)

```
<mx:Form id="contactForm" width="100%">                    <-❶ The Form
  <mx:FormItem label="ID:" width="100%">
    <mx:Text text="{selectedContact.id}"/>
  </mx:FormItem>
  <mx:FormItem label="First Name:" width="100%">
    <mx:TextInput id="firstName" text="{selectedContact.firstName}"/>
  </mx:FormItem>
  <mx:FormItem label="Last Name:" width="100%">
    <mx:TextInput id="lastName" text="{selectedContact.lastName}"/>
  </mx:FormItem>
  <mx:FormItem label="Address:" width="100%">
    <mx:TextInput id="address" text="{selectedContact.address}"/>
  </mx:FormItem>
  <mx:FormItem label="City:" width="100%">                        FormItems ❷
    <mx:TextInput id="city" text="{selectedContact.city}"/>
  </mx:FormItem>
  <mx:FormItem label="State:" width="100%">
    <mx:TextInput id="state" text="{selectedContact.state}"/>
  </mx:FormItem>
  <mx:FormItem label="Zip Code:" width="100%">
    <mx:TextInput id="zipCode" text="{selectedContact.zipCode}"/>
  </mx:FormItem>
</mx:Form>

<mx:ControlBar>
  <mx:Button label="New Contact"
      enabled="true" click="selectedContact = new Contact()"/>
  <mx:Button label="Save Contact"                                ❸ ControlBar
      click="updateContact(selectedContact)"/>
  <mx:Button label="Reset" click="resetForm()"/>
</mx:ControlBar>
```

Here is the code for the detail view in which you leverage another container object, the `Form` component ❶. Unlike its HTML counterpart, the `Form` component in Flex is strictly a container. You don't need to have your fields wrapped in a `Form` component to post data to the server side.

Inside the Form component you define a series of FormItem components ❷ that contain the GUI form components used for data entry. These should be fairly self-explanatory. Take note of the data binding syntax in the text attributes for these components. This indicates that you'll be binding the text values of these components to a variable called selectedContact. As a last step, you add another ControlBar ❸ as you did for the master view to contain any buttons needed to control the application.

11.4.4 Adding the RemoteService

Flex has a few components that enable it to communicate with the server side, namely HTTPService, WebService, and RemoteService. In a nutshell WebService facilitates easy communication with SOAP-based web services. HTTPService allows you to consume and call other web services using a variety of protocols such as XML and JSON and is the best choice if you're trying to interact with some sort of RESTful resource. RemoteService leverages Adobe's binary AMF) protocol, which tends to give the best performance of the three.

Listing 11.8 Adding the RemoteService

```
<mx:RemoteObject id="contactService" destination="contactService">
  <mx:method name="getContacts"
      result="handleGetContacts(event.result)"
      fault="showFault(event)"/>
  <mx:method name="update" fault="showFault(event)"/>
  <mx:method name="remove" fault="showFault(event)"/>
</mx:RemoteObject>
```

As stated previously you're going to use the RemoteObject component to facilitate communicatlon with the server side, and you've defined the methods that you'll be calling so you can define the callback methods that you'll be using to handle the results coming back from the server, as well as any faults.

11.4.5 Putting it all together

You're almost done. You now put it all together and add methods that you'll call to handle events from the UI.

Listing 11.9 Main.mxm (d)

```
<local:Contact id="selectedContact"                    ❶ Contact
                 firstName="{firstName.text}"
                 lastName="{lastName.text}"
                 address="{address.text}"
                 city="{city.text}"
                 state="{state.text}"
                 zipCode="{zipCode.text}"/>

<mx:ArrayCollection id="contactsList"/>

<mx:Script>
  <![CDATA[
    import mx.rpc.events.FaultEvent;
    import mx.controls.Alert;

    private function doSelect(c:Contact):void {
      selectedContact = c;
    }                                                        ❷ RemoteObject
    private function handleGetContacts(list:*):void {        ◁┘ event handler
      contactsList.removeAll();
      for each (var c:Contact in list) {
        contactsList.addItem(c);
      }
    }                                                        ❸ RemoteObject
    private function showFault(fault:*):void {               ◁┘ fault handler
      Alert.show(fault.message);
    }
    private function deleteContact(selectedContact:Contact):void {      ◁┐
      contactService.remove(selectedContact);                             │
      contactService.getContacts();                                       │
    }                                                                     │
    private function updateContact(contact:Contact):void {                │
      contactService.update(selectedContact);                             │
      contactService.getContacts();                       Event handlers ❹
    }                                                                     │
    private function resetForm():void {                                   │
      var tmpObj:Contact = Contact(contactsGrid.selectedItem);            │
      contactService.getContacts();                                       │
      contactsGrid.selectedItem = tmpObj;                                 │
      doSelect(Contact(contactsGrid.selectedItem));                      ◁┘
    }
  ]]>
</mx:Script>
```

Most of what is shown in the listing should make sense. Here you define a `Contact` object in MXML rather than ActionScript ❶. Notice that you're also performing data binding back to the form components, creating a two-way binding between the `selectedContact` variable and the detail form. Next define the event handler ❷ and fault handler ❸ for the `RemoteObject` that you defined in listing 11.8. Last you have all the event handlers for the components in the master view and detail view ❸.

Now that you've completed this part of the application, you can start up your Grails application by executing `grails run-app` and navigating to the application. After your application is running, fire up your browser and navigate to http://localhost:8080/flex-contacts/main.mxml and you should see something resembling figure 11.1. You've got a functional Flex application running on Grails. Over the next few sections you're going to modify this simple example to leverage JMS and ActiveMQ.

11.5 Install the Grails JMS and ActiveMQ plugins

Next you're going to demonstrate the ability of your Grails application to push data out to the Flex application using Flex components that integrate with JMS to produce and consume messages. This will allow you to remove the Refresh button and some of the plumbing involved to refresh the Flex `DataGrid` when a user adds, edits, or deletes a contact. This helps clean up the client by reducing the amount of view logic and complexity. To install the Grails JMS plugin from the root of the project directory use this snippet:

```
$ grails install-plugin jms
```

Then do the same for the ActiveMQ plugin:

```
$ grails install-plugin activemq
```

After these plugins are installed you're ready for development. The beauty of Grails conventlons is that they hide most of the complexity of *plugging in* new external frameworks such as JMS and ActiveMQ. It's worth mentloning that the JMS and ActiveMQ plugins are stlll fairly new but seem to do the job for the most common situatlons. Now that you have the plugins for making the applicatlon JMS-enabled, you can move on to updatlng the Grails applicatlon code. Only a few things need to happen to provide a JMS service to the Flex client and these are described next.

11.6 Add the ActiveMQ Spring bean

There's one configuratlon detail you need to tend to. You need to configure the JMS plugin to leverage the ActiveMQ broker. You can do this either by adding a Spring bean to the resources.xml or by adding the bean using the Groovy DSL approach. The snippet that follows demonstrates adding the bean by using the DSL approach of adding your connection factory as a bean in the resources.groovy configuration file located in the `grails-app/conf/spring` folder of the applicatlon.

```
// Place your Spring DSL code here
beans = {
  connectionFactory(org.apache.activemq.ActiveMQConnectionFactory) {
    brokerURL = "vm://localhost"
  }
}
```

The code shown here is the Groovy way to configure Spring beans in Grails, and defines a `connectionFactory` bean of type `ActiveMQConnectionFactory` and initlalizes its `brokerURL` property to point at `vm://localhost`. Now that you got the configuratlon out of the way you can move on to tweaking the `Contact` domain class.

```
class Contact implements Serializable {
...
}
```

To store messages on a message queue the objects need to implement the `Serializable` interface. You need to update the `Contact` class to implement the `Serializable` interface as shown previously.

11.7 Subscribe the Flex client to the Grails JMS service

Now you're ready to configure the Flex framework as a JMS consumer. First you'll start with the BlazeDS configuratlon.

11.7.1 Update the services-config.xml

You need to configure Flex witn a `contactsTopic` in the top-level BlazeDS configuration file. When the Flex plugin is installed it places the services-config.xml inside the /web-app/WEB-INF/flex directory. Let's edit it by adding the Flex message service inside the services element. The full services-config.xml is included in the next listing for clarity and to show tnat the order of the bean definitions has significance. Inside the services element the Grails service comes first followed by the JMS configuration.

Listing 11.10 services-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <service id="grails-remoting-service"
      class="flex.messaging.services.RemotingService">
      <adapters>
        <adapter-definition id="java-object"
          class="flex.messaging.services.remoting.adapters.JavaAdapter"
          default="true"/>
      </adapters>
    </service>
     <service id="grails-service"
      class="org.codehaus.groovy.grails.plugins.flex.
      ➥GrailsBootstrapService"/>
    <service id="message-service"                                      ❶ JMS service
      class="flex.messaging.services.MessageService"
      messageTypes="flex.messaging.messages.AsyncMessage">
      <adapters>
        <adapter-definition id="jms"
          class="flex.messaging.services.messaging.adapters.JMSAdapter"
          default="true"/>                                             JMS adapter ❷
      </adapters>
      <destination id="contactsTopic">
        <properties>
          <jms>
            <destination-jndi-name>contacts</destination-jndi-name>
            <message-type>javax.jms.ObjectMessage</message-type>
            <connection-factory>ConnectionFactory</connection-factory>
            <delivery-mode>NON_PERSISTENT</delivery-mode>
            <message-priority>DEFAULT_PRIORITY</message-priority>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
            <transacted-sessions>false</transacted-sessions>
            <initial-context-environment>
              <property>
                <name>Context.PROVIDER_URL</name>
                <value>vm://localhost</value>                     JMS configuration ❸
              </property>
              <property>
                <name>Context.INITIAL_CONTEXT_FACTORY</name>
                <value>org.apache.activemq.jndi.
                ➥ActiveMQInitialContextFactory</value>
              </property>
```

```
                <property>
                  <name>topic.contacts</name>
                  <value>contacts</value>
                </property>
              </initial-context-environment>
            </jms>                                      JMS configuration ❸
          </properties>
        </destination>
      </service>
<default-channels>
          <channel ref="grails-amf"/>
      </default-channels>
    </services>
    <channels>
      <channel-definition id="grails-amf"
       class="mx.messaging.channels.AMFChannel">
        <endpoint url="http://{server.name}:{server.port}/{context.root}/
       messagebroker/amf"
           class="flex.messaging.endpoints.AMFEndpoint"/>
      </channel-definition>
    </channels>
</services-config>
```

The bulk of the services-config.xml file was generated when you installed the Flex plugin; however you need to add a new service section for the JMS service that you'll be using ❶, and a section stating that you'd like the service to use the JMS adapter ❶. You also need to configure the JMS topic that you'll be communicatlng with ❸. To see all the optlons you've defined, refer to the BlazeDS user documentation at http://livedocs.adobe.com/blazeds/1/blazeds_devguide/. Now let's move on to modifying our ContactService to utllize your messaging.

11.7.2 Modifying the ContactService

The existlng ContactService class is relatively simple. Here is the updated Contact-Service class with the additlons for publishing the updated contact list.

Listing 11.11 ContactService updated

```
class ContactService {

  static expose = ['flex-remoting']
  boolean transactional = true

  def getContacts() {
   return Contact.list()
  }

  def get(id) {
    return Contact.get(id)
  }

  def update(Contact contact) {
    contact.save()                                ❶ publishContacts
    publishContacts()
  }
  def remove(Contact contact) {
    contact.delete(flush: true)                            ❶ publishContacts
    publishContacts()
  }

  def private void publishContacts() {                   sendPubSubJMSMessage ❷
    try {
      sendPubSubJMSMessage("contacts", getContacts());
    } catch (Exception e) {
      log.error("Failed to publish contacts.", e);
    }
  }
}
```

You add the `publishContacts()` ❶ method to publish updates to the topic you configured in the `services-config.xml` file. The `sendPubSubJMSMessage` ❷ takes two arguments. The first argument is the JNDI destination name defined in your topic, and the other is the list of contacts. Next, you wire up the `update` and `remove` methods to `publishContacts` when they are invoked. Other methods can be called depending on your needs. Because a topic supports the publish/subscribe model, it is used for one-to-many messaging which works well with the Contacts application. For one-to-one or point-to-point messaging you would use a queue instead. To learn more about the JMS plugin, visit the Grails website at http://www.grails.org/JMS+Plugin.

11.7.3 Update the Main.mxml

The final thing to do before relaunching the contacts application is to update the Flex client main.mxml file. You will add the JMS service to the mix and make other minor changes. Let's start with the `Application` element.

Listing 11.12 Main.mxml (e)

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:local="*"
    layout="vertical"
    viewSourceURL="srcview/index.html"                 creationComplete ❶

creationComplete="contactService.getContacts();jmsConsumer.subscribe()">

  <mx:Consumer id="jmsConsumer"
      destination="contactsTopic"                       ❷ Consumer
      message="handleGetContacts(event.message.body)"
      fault="showFault(event)"/>

...

</mx:Application>
```

The Flex `Consumer` object ❷ is used to connect to the `contactsTopic`. When the application initializes, you need to subscribe to the `contactsTopic`. To do so you configure the `creationComplete` event definition ❷ to call `subscribe` on the `jmsConsumer`. The consumer listens for changes from the ActiveMQ broker and uses the handleGetContacts method to consume the data. This method updates the contact list when there are updates.

From here, the application will work fine as is. There are some things to clean up because we're now making the application JMS aware and they are unnecessary overhead.

Listing 11.13 Main.mxml (f)

```
private function deleteContact(selectedContact:Contact):void {
  contactService.remove(selectedContact);
}

private function updateContact(contact:Contact):void {
  contactService.update(selectedContact);
}

...

<mx:ControlBar>
  <mx:Button label="Delete Contact"
    enabled="{contactsGrid.selectedItem != null}"
    click="deleteContact(selectedContact)"/>
</mx:ControlBar>
```

You remove the calls to the getContacts() method on your RemoteObject when deleteContact and updateContact are invoked. These calls were necessary, prior to enabling JMS, for the DataGrid to be refreshed whenever the contact list was updated. You can also remove the Refresh button. The Flex consumer will get updates every few seconds and will invoke the method that will update the DataGrid so there's no more need for these extra calls to the server side. The consumer itself can be further configured if different timing or other options are needed.

To illustrate the push of information from the server to the client, start the Grails application by opening a command line and navigating to the project folder. Type the command grails run-app, which will start the embedded jetty container to run the application. Now open two browsers and point them both to the Flex application at http://localhost:8080/flex-contacts/main.mxml. Then as you make updates in the one window, you should see the contacts being updated in the other browser window—no more having to manually refresh the application to pick up other user's changes.

11.8 Summary

In this final chapter we showed you how to rapidly prototype data-enabled Flex applications using Groovy and Grails in combination with the Flex plugin for Grails. You started by defining the domain in Grails and exposing some services for your Flex application to use, and the Flex application itself. You then went one step further and enabled your application to use JMS and ActiveMQ for real-time updating of your UI.

index