Professional BlazeDS

Creating Rich Internet Applications with Flex® and Java®

Shashank Tiwari
Get more out of WROX.com

**Interact**
Take an active role online by participating in our P2P forums

**Wrox Online Library**
Hundreds of our books are available online through Books24x7.com

**Wrox Blox**
Download short informational pieces and code to keep you up to date and out of trouble!

**Chapters on Demand**
Purchase individual book chapters in pdf format

**Join the Community**
Sign up for our free monthly newsletter at newsletter.wrox.com

**Browse**
Ready for more Wrox? We have books and e-books available on .NET, SQL Server, Java, XML, Visual Basic, C#/ C++, and much more!

Contact Us.
We always like to get feedback from our readers. Have a book idea?
Need community support? Let us know by e-mailing wrox-partnerwithus@wrox.com

Download from Library of Wow! eBook <www.wowebook.com>
Professional BlazeDS

Foreword ................................................................. xxii
Introduction ........................................................................ xxiii
Chapter 1: Flex for Java Developers .................................... 1
Chapter 2: Introducing BlazeDS ......................................... 41
Chapter 3: Using BlazeDS as a Server-Side Proxy ............... 65
Chapter 4: Remoting between Flex and POJOs .................... 83
Chapter 5: Accessing Spring Beans .................................... 113
Chapter 6: Communicating Real-Time via Messages ............. 137
Chapter 7: Leveraging JPA and Hibernate with Flex ............. 169
Chapter 8: Testing and Debugging Flex and BlazeDS Applications .... 215
Chapter 9: Extending BlazeDS with Custom Adapters ........... 253
Chapter 10: Making BlazeDS Applications Scalable ............... 271
Chapter 11: CRUD Applications and More ......................... 293
Appendix A: Introducing Flex 4 ........................................ 325
Index ............................................................................... 337
Professional
BlazeDS
My efforts on this book are dedicated to my wife Caren and sons Ayaan and Ezra. Thanks for always encouraging and supporting me in my endeavors, like writing this book or starting the new venture. Without your support and love I could have never completed this book.
About the Author

**Shashank Tiwari** is a technology entrepreneur, speaker, author, and innovator. He is currently a Managing Partner & CTO at Treasury of Ideas (http://www.treasuryofideas.com), a technology-driven innovation and value optimization company. As an experienced software developer and architect, he is adept in a multitude of technologies. He is an internationally recognized speaker, author, and mentor. As an expert group member on a number of JCP (Java Community Process) specifications, he has been actively participating in shaping the future of Java. He is also an Adobe Flex Champion and a common voice in the RIA community. Currently, he passionately builds rich, high-performance scalable applications and advises many on RIA and SOA adoption. His clients range from large financial service corporations to brilliant startups, for whom he helps translate cutting edge ideas into reality. He is also actively engaged in training and mentoring developers and architects in leading edge technology. He is the author of a number of books and articles, including *Advanced Flex 3* (Apress, 2008). He lives with his wife and two sons in New York. More information about him can be accessed at his website: [http://www.shanky.org](http://www.shanky.org).
Credits

Acquisitions Editor
Scott Meyers

Project Editor
Tom Dinse

Technical Editor
Yakov Fain

Production Editor
Eric Charbonneau

Copy Editor
Foxxe Editorial Services

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Proofreader
Jen Larsen, Word One

Indexer
Johnna VanHoose Dinse

Cover Photo
©Thinkstock/Punchstock
This book represents the efforts of many people and I sincerely thank them for their contribution.

Many thanks to the entire team at Wrox/Wiley. You made this book possible! Scott Meyers took this book from a proposal to a real project. Tom Dinse diligently and patiently reviewed the manuscript and managed the schedule. Many others worked seamlessly behind the scenes to make things happen.

Thanks much to Yakov Fain for meticulously reviewing and validating the manuscript for its technical correctness. He deserves special praise for his attention to detail and valuable insight.

I sincerely appreciate all of you (everyone at Wiley and Yakov) for working extra hard against tight timelines to make things happen on schedule.

Special thanks to the folks at Adobe. Thanks to Jeff Vroom for his early encouragement and support. Thanks to James Ward for the encouraging words he wrote in the forward to this book. Many thanks to Jeremy Grelle, the lead of the Spring BlazeDS project at SpringSource, for his help.

Thanks to Jerry Bezdikian for my About the Author photo.

Above all, thanks to my friends and family for their assistance and encouragement. My wife, Caren, and sons, Ayaan and Ezra, were extremely loving and accommodating while I took out time on many evenings, early mornings, and nights in the last many months to finish the book. They were very involved in the project. My five year old, Ayaan, continuously knew how I was progressing through the book, often encouraging me to finish a chapter sooner than I would otherwise have. My three year old, Ezra, was quieter than his normal naughty self when I was working hard typing away on my keyboard. My wife, Caren, planned the last many weekdays and weekends solely around my writing schedule. She fed me with delicious food as she always does and helped me stay focused all along, even when the task looked too difficult. Without their affection, I would have never been able to write this book.

Thanks also to my parents (Mandakini and Suresh), my in-laws (Cissy and Frank), and all other near and dear ones for their continuing support and encouragement.

My sincere thanks and gratitude to all of you who contributed towards this book.
# Contents

**Foreword**  xxvi  
**Introduction**  xxvii  

## Chapter 1: Flex for Java Developers

- **Introducing Flex through Its Controls and Containers**  1  
  - Tip to Remember  3  
    - Getting to Know the Controls  5  
    - Laying Out the Controls  19  
  - Leveraging Services to Access External Data  27  
    - Using Simple HTTP and REST Style Invocations  28  
    - Communicating via Web Services  30  
    - Making Remote Procedure Calls  32  
  - Making User Interfaces Engaging and Effective  33  
    - Exploring Effects  33  
    - Validating User Input  34  
    - Customizing the Look and Feel  35  
  - Extending Flex  36  
    - Creating Custom Components  36  
    - Creating and Propagating Custom Events  38  
  - Summary  39  

## Chapter 2: Introducing BlazeDS

- **An Overview of BlazeDS**  41  
  - Viewing Integration from 30,000 feet  43  
  - Viewing Integration from 10,000 feet  45  
  - Viewing It Closely from 1,000 Feet  45  
- **Getting and Installing BlazeDS**  47  
  - Downloading BlazeDS  47  
  - Installing the Compiled Distributions  49  
  - Compiling from Source  53  
- **Surveying the Configuration Options**  54  
  - First Look at services-config.xml  55  
  - Remoting, Proxy and Messaging Configuration  57  
- **Yet Another Hello to the World**  58  
- **Summary**  64

Download from Library of Wow! eBook <www.wowebook.com>
### Chapter 3: Using BlazeDS as a Server-Side Proxy

**HTTP Service Proxy**
- An HTTP Proxy Service Example
- Advanced Configuration Options of an HTTP Proxy Adapter
- Intercepting for Control and Management

**Web Service Proxy**
- Configuring a Web Service Proxy

**Summary**

### Chapter 4: Remoting between Flex and POJOs

**Essentials of Remoting**
- Remoting Communication Flow
- Remoting-Related Classes, Components, and Modules

**Configuring a Remoting Service and a Destination**
- Securing Destinations
- Intercepting and Logging Calls to a Remoting Service
- Talking AMFX
- Using HTTPS

**Diving into Serialization — AS3 to Java and Back**

**Running through a Complete Example**
- Java-Based Data Service
- Client-Side Flex Code

**Value Additions for Simple Remoting**
- Custom Security Implementation
- An Interceptor for the JavaAdapter

**Summary**

### Chapter 5: Accessing Spring Beans

**Basic Bootstrapping**
- Using BlazeDS MessageBroker with the Spring DispatcherServlet
- Serving Flex and Non-Flex Clients

**Remoting to Spring Beans**

**Message Service Integration**
- Native BlazeDS AMF Messaging
- JMS Messaging Using Spring-Specific JMS Components
- Using the Special Integration Adapter

**Leveraging Spring Security**
- Securing Remote Spring Bean Access

**Using Custom Adapters**

**Summary**
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flex and Hibernate Integration Expectations</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>BlazeDS Adapter for JPA and Hibernate</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>Aspect-Oriented Programming (AOP)</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>JPA and Hibernate Adapters for BlazeDS</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>dpHibernate</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>Gilead</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>Managed Entities for Flex Applications</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>212</td>
</tr>
<tr>
<td>Chapter 8: Testing and Debugging Flex and BlazeDS Applications</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>FlexUnit</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>Functional and Interface Testing</td>
<td>227</td>
</tr>
<tr>
<td></td>
<td>Debugging</td>
<td>243</td>
</tr>
<tr>
<td></td>
<td>Logging</td>
<td>243</td>
</tr>
<tr>
<td></td>
<td>Profiling</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>251</td>
</tr>
<tr>
<td>Chapter 9: Extending BlazeDS with Custom Adapters</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contextual Overview</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>The API</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>A Sample Custom Adapter</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>Custom Factories</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td>Extending the Messaging Adapter</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>Advanced Strategies for Custom Adapter Creation</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>270</td>
</tr>
<tr>
<td>Chapter 10: Making BlazeDS Applications Scalable</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clustering</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>Channel Failover</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>Clustering Powered by JGroups</td>
<td>274</td>
</tr>
<tr>
<td></td>
<td>Data Compression</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>Data Format Optimization</td>
<td>278</td>
</tr>
<tr>
<td></td>
<td>Robust Connection</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>Service Orientation</td>
<td>282</td>
</tr>
<tr>
<td></td>
<td>Concurrent Contracts</td>
<td>282</td>
</tr>
<tr>
<td></td>
<td>Cross-Domain Utility Layer</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td>Functional Decomposition</td>
<td>283</td>
</tr>
<tr>
<td>Contents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Rules Centralization</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>State Repository</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td><strong>Caching</strong></td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>Signed Framework RSL</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td><strong>Resource Pooling</strong></td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>JDBC Connection Pooling</td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>JMS Resource Pooling</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>Bind Messaging to the Beans Using Camel</td>
<td>290</td>
<td></td>
</tr>
<tr>
<td>Using Spring’s Support for Message Listener Containers</td>
<td>291</td>
<td></td>
</tr>
<tr>
<td><strong>Workload Distribution</strong></td>
<td>291</td>
<td></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>292</td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 11: CRUD Applications and More</strong></td>
<td>293</td>
<td></td>
</tr>
<tr>
<td><strong>CRUD with Flash Builder 4</strong></td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>CRUD with RESTful Endpoints</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>Generating Client Code for WebService Consumption</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>Generating CRUD with Remoting Service Destinations</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td><strong>Generating AS3 Model Objects</strong></td>
<td>312</td>
<td></td>
</tr>
<tr>
<td>Using Gas3</td>
<td>312</td>
<td></td>
</tr>
<tr>
<td>Auto-Populating AS3 Model Objects with XML Based on an XSD</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>323</td>
<td></td>
</tr>
<tr>
<td><strong>Appendix A: Introducing Flex 4</strong></td>
<td>325</td>
<td></td>
</tr>
<tr>
<td><strong>What Is Flex 4?</strong></td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>What’s New?</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td>Flex 4 and Before</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td><strong>Leveraging the New Framework</strong></td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>Development Workflows</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>Flex 4 Says Hello</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>337</td>
<td></td>
</tr>
</tbody>
</table>
As with many interactions between developers these days, I first met Shashank Tiwari online via his blog. He was writing thoughtfully about Java and Flex back in the days when this was mostly uncharted territory where only a handful of brave developers had ventured. Through those interactions we began making plans to lead more Java developers to the blissful world of Java and Flex.

Having been a professional software developer for more than ten years, I’ve met developers from a wide variety of backgrounds, including hackers, scientists, teachers, plumbers, and athletes. There is no better way to know what a developer is like than to write code and articles with them. Over the past few years Shashank and I have co-authored a number of articles and code samples together. From these experiences, I know that Shashank is a top notch Java developer and a Flex master. What makes Shashank’s books stand out, however, is that he is not only a superb developer but also an excellent teacher.

I was very excited when I heard that Shashank was working on a new book focused on integrating Java and Flex through BlazeDS. So many Java developers I interact with want to build better software but are reluctant to learn a new technology stack. Flex and BlazeDS provide a great path for Java developers to build better software that users love to use. The wealth of out-of-the-box and community-built Flex components enables developers to rapidly assemble great applications. But there is a learning curve for the typical Java developer who must understand and navigate the intricacies of a thick-client UI framework, data marshalling mechanisms, and a new tool chain. Shashank makes that learning curve smooth and fun.

This book provides much needed practical knowledge for assembling production applications using Java and Flex. The extensions Shashank has built to BlazeDS, his examples of real-world Spring integration, and his detailed explanations regarding Hibernate data access provide readers with end-to-end instructions for building great UIs on Java back-ends.

I’ve had the great privilege of working with and learning from Shashank. He has helped me to be a better developer and to gain a more complete understanding of new technologies. I’m sure that Professional BlazeDS will do the same for you.

James Ward

www.jamesward.com

Co-author of First Steps in Flex with Bruce Eckel

www.firststepsinflex.com
Java easily ranks among the most popular enterprise computing platforms. Adobe Flex is a leading Rich Internet Application (RIA) development framework. The combination of Java and Flex is one of the most compelling propositions for enterprises to create rich and engaging applications to enhance productivity and increase customer satisfaction.

An effective integration of Java and Flex implies that you can leverage robust scalable Java server side applications in conjunction with the superior rendering capabilities of the Flash player. It also means that your investments in Java server back-ends can be reused and leveraged in your new generation applications.

Looking at the integration from the other side, an effective combination of Java and Flex means that your clunky old web 1.0 interfaces and old-style interaction models can be replaced with their more effective new-generation counterparts.

In addition to all the benefits listed above, an effective plumbing between Java and Flex promises Flex’s reach to all enterprise resources, including databases, enterprise resource planning systems, authentication systems and organization-wide messaging systems.

Java and Flex integration is achieved with the help of a number of products and tools, including LifeCycle Data Services (LCDS) and BlazeDS from Adobe. LCDS is a commercial product and BlazeDS is an open source alternative to LCDS.

This book is all about combining Java and Flex. The book covers the integration story with BlazeDS as the protagonist. It explores all aspects of BlazeDS and explains its extension points that allow it to be stretched beyond its off-the-shelf capabilities.

This is one of the first few books that addresses the topic of Java and Flex integration using BlazeDS. From the very first draft proposal I have seen a lot of excitement around this topic and this book (at least among all involved in its creation). As I was writing this book over the last few months, BlazeDS and Flex saw a lot of rapid changes and inclusion of newer features and capabilities. It was also the time Spring BlazeDS emerged as a SpringSource and Adobe supported initiatives to integrate Spring and Flex using BlazeDS. I have made every attempt to keep the content up-to-date and relevant. However, some of the pieces of software are in beta phase and will continue to evolve as we work hard to make this book available in your hands. Even though this makes the book susceptible to being outdated on a few rare instances, I am confident the content remains useful and relevant through the next versions of the Flex framework, Java, and BlazeDS.

The book at all times attempts to balance between theory and practice, giving you enough visibility into the underlying concepts and providing you with the best practices and practical advice that you can apply at your workplace right away.
Introduction

Who This Book Is For

This book is best suited for those developers and architects who leverage Java and Flex to create rich internet applications. Such developers are likely to come from either the Java or the Flex backgrounds, or from a background that includes neither of the two technologies.

For those who are Java experts the book teaches the details of BlazeDS and its applicability in using it to plumb Java and Flex applications together. I understand that some such experts may not have sufficient knowledge of the Flex framework. Therefore I explicitly introduce Flex 3 in Chapter 1 and provide a peek into the upcoming Flex 4 version in the appendix.

For those who are Flex developers and architects the book provides explanations and examples of how they can effectively include BlazeDS in their application stack. I assume much of the conceptual content would be accessible to such developers. It’s possible though that some advanced topics on persistence, Spring integration and custom extensions could be difficult for a developer who has little or no knowledge of Java. This book does not attempt to teach Java fundamentals, which is outside its scope. Tools for learning the essentials of Java are widely available and I encourage you take advantage of them to get the most out of this book.

I advise that you read the book from start to finish for an initial understanding and then return to specific chapters to apply detailed and specific recommendations.

What This Book Covers

This book covers Java version 5 and above, Flex version 3.x, BlazeDS version 3.x and 4 beta and Flash Builder version 3.x and 4 beta. The book also introduces Flex 4 beta minimally. The content in this book should be relevant through Flex 4, BlazeDS 4 and Flash Builder 4.

Version 1.0 of Spring BlazeDS and FlexMonkey, JMS 1.1 and Hibernate 3.3.2.GA are covered.

How This Book Is Structured

This book consists of 11 chapters and an appendix. The 11 chapters gradually start from the fundamentals and progress to cover increasingly complex topics. The first chapter introduces the Flex 3 framework. The appendix minimally introduces Flex 4 beta. The rest of the chapters cover the different aspects of BlazeDS, its existing features, and its custom extensions. A majority of the chapters illustrate the proxy, remoting, and messaging capabilities of BlazeDS. For a holistic treatment, chapters on testing and scalability are also included.

As mentioned previously, I recommend that you read the book sequentially from the first to the last chapter. That way your learning curve will be smooth and gradual.

Chapter by chapter, this book covers:

- **Flex for Java Developers (Chapter 1)** — A brief primer to Flex for Java developers. The chapter teaches Flex fundamentals, providing a few examples in context.
- **Introducing BlazeDS (Chapter 2)** — This chapter gently introduces BlazeDS. It explains the BlazeDS architecture, its core features, configuration files and the deployment infrastructure. In addition, the chapter explains the installation processes and peeks into the underlying core.
Introduction

❑ Using BlazeDS as a Server-side Proxy (Chapter 3) — Flex applications can access data and web services from remote destinations if the remote destinations define an appropriate security definition, in a file called crossdomain.xml. The Flash player debars access from sources that lack such a security definition. However, BlazeDS provides server side proxy capabilities to access these sources. The focus here is on the role of BlazeDS as a server-side proxy.

❑ Remoting between Flex and POJOs (Chapter 4) — This chapter illustrates the details of remote procedure calls between Flex and Java objects and explains how objects can be streamed between the two using a binary protocol called AMF (Action Message Format).

❑ Accessing Spring Beans (Chapter 5) — SpringSource, the makers of the Spring Framework, and Adobe, makers of Flex, combined efforts to create Spring BlazeDS that helps integrate Spring and Flex applications using BlazeDS effectively. This chapter covers the features of this new integration project.

❑ Communicating Real-time via Messages (Chapter 6) — BlazeDS supports real-time messaging and data push. In this chapter I start with the essentials of messaging between Flex and Java and highlight the important aspects like AMF long polling and the JMS adapter.

❑ Leveraging JPA and Hibernate with Flex (Chapter 7) — Hibernate is the most popular Java persistence framework and JPA is the standard that defines Java EE persistence architecture. Both the product (Hibernate) and the associated specification (JPA) bring benefits of Object relational mapping to the world of Java. This chapter explains how the same features can be extended to Flex applications using BlazeDS.

❑ Testing and Debugging Flex and BlazeDS Applications (Chapter 8) — This chapter explains the nuts and bolts of logging, testing, and debugging Flex and BlazeDS applications.

❑ Extending BlazeDS with Custom Adapters (Chapter 9) — BlazeDS is a great product off-the-shelf, but the most valuable aspect of its design is that it can be extended easily to adapt to varied server side resources. This chapter will explain the available API and the approach to creating, deploying and using custom adapters.

❑ Making BlazeDS Applications Scalable (Chapter 10) — This chapter explains how you could scale BlazeDS using the concepts that involve clustering, data compression, data format optimization, robust connection, service orientation, caching, resource pooling, and workload distribution.

❑ CRUD Applications and More (Chapter 11) — This chapter illustrates CRUD application generation using Flash Builder 4 beta. It also illustrates the use of Gas3 for automatic translation from Java to AS3 and auto-population of AS3 objects based on XML content that support XML schema definitions.

❑ Introducing Flex 4 (Appendix) — The appendix introduces the fundamentals of Flex 4, the next version of the Flex framework. Currently, Flex 4 is in beta. It will be released only in early 2010. The appendix provides a sneak peek into what’s brewing.

What You Need to Use This Book

This book requires that you have the following software installed on your machine:

❑ JRE version 5 or above
❑ Flex SDK version 3
❑ BlazeDS version 3 or above
Introduction

- Flex Builder version 3 (not a necessity but convenient)
- Flash Builder 4 beta (especially to follow along the directions in Chapter 11)
- JBoss AS or Tomcat (when using Tomcat, please install an external JMS provider like ActiveMQ)

In addition, for specific chapters, you will need the following:

- Spring BlazeDS
- FlexMonkey
- JMS provider (like ActiveMQ)
- Hibernate
- JGroups
- Ruby and FunFX

Conventions

To help you get the most from the text and keep track of what’s happening, we’ve used a number of conventions throughout the book.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- We highlight new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show file names, URLs, and code within the text like so: persistence.properties.
- We present code in two different ways:
  - We use a monofont type with no highlighting for most code examples.
  - We use gray highlighting to emphasize code that is of particular importance in the present context.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book’s title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book’s detail page to obtain all the source code for the book.
Introduction

Because many books have similar titles, you may find it easiest to search by ISBN; this book’s ISBN is 978-0-470-46489-2.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book’s errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don’t spot “your” error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At http://p2p.wrox.com you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P but in order to post your own messages, you must join.
Introduction

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.
Flex for Java Developers

This book is a comprehensive resource on leveraging Flex and Java together to create robust enterprise-grade Rich Internet Applications (RIAs). The book chooses the popular open source data services product from Adobe, BlazeDS, as the critical integration layer that makes Flex and server-side Java work together. BlazeDS is a piece of Java code and works within a Java virtual machine (JVM). More precisely, it’s a web application that can be successfully deployed in a Java application server or even in a minimalist Servlet container like Apache Tomcat. The book, therefore, talks more Java than Flex and focuses on how BlazeDS can be configured, customized, and extended to integrate with all things Java. However, it’s not practical that integration be studied without looking at both sides of the puzzle. Therefore, in this chapter, we start by talking about Flex, Java’s counterpart in the context of this book. It’s not a comprehensive treatment of the subject. It can’t be in mere 40-odd pages. It’s just enough information to help you understand the basics and to present to you the interfaces and integration points. The illustration and discussion is biased toward addressing the Java developer more than anybody else, although nothing in this chapter really requires knowledge of Java itself.

Let’s begin exploring and understanding the fundamentals of Flex.

Introducing Flex through Its Controls and Containers

Flex is an RIA development framework. It provides a set of controls, services, and utilities, which can be readily used together to create rich and engaging applications. Developers programming in Flex use ActionScript 3 (AS3) and MXML to create user interfaces, consume data, provide interactivity, and maintain state.

AS3 is an object-oriented programming language that has a close resemblance to the Java and JavaScript programming languages. MXML is an XML-based declarative language, which makes programming intuitive and easy. All MXML code is translated automatically to AS3 before it’s compiled. This intermediate AS3 representation of the MXML code is often deleted unless a developer
specifies that that it be saved. Generated AS3 representations of MXML can be saved by explicitly specifying the “-keep” option to the Flex compiler. AS3 compiles to a byte code format called “.swf,” that runs within the Flash Player virtual machine.

Figure 1-1 shows the essential parts of the Flex framework in a diagram. A Flex application, on compilation, can output a .swf file that is ready to run in a Flash Player, or it can output a .swc file that can be consumed as a library and made part of a .swf file. A .swf file can be dropped into a web server and accessed over the web using the standard HTTP protocol. When accessed by a user, the .swf is downloaded locally to the user’s machine and played in the local browser that hosts the Flash Player. Figure 1-2 depicts the accessing of a .swf file in a diagram.

This book is written for the Flex 3.2 software development kit (SDK) and the Flex 4 SDK. Every attempt is made to cover the entire set of features in Flex 3.2 and Flex 4 relevant to the integration of Flex and Java. However, at the time of writing Flex 4 is still in the oven and its final form is still unknown. I will try and keep close pace with the development of Flex 4 as I write this book. Even then, it’s possible that a few things may change by the time you get hold of this book. Despite this possible small caveat, I am still certain that almost all the content of this book will be useful and relevant.
At this point, let’s write a simple example application to reinforce the basics you have learned so far. The sample application has two visual interface components: a text input and a text display area. Each of these components has a label attached to it. The labels help the user understand what to input and what sense to make of the display. The source for the visual controls alone is:

```xml
<mx:Label text="What's your name?">
<mx:TextInput id="textInput"/>
<mx:Button id="button" label="Submit" click="clickHandler(event)"/> 
<mx:Label text="{yourName}"/>
```

Adobe likes to refer to Flex user interface components as controls. In addition to user interface components, Flex contains classes that act as services, utilities, and helpers. These classes help with the following:

- Maintain data collections
- Bind data from source to destinations
- Control event propagation and handling
- Connect to external sources of data and services
- Actualize effects and transitions
- Affect the appearance of the controls
- Validate and format data
- Facilitate data visualization
- Support accessibility requirements
- Test, log, tune, and structure applications

The visual controls code suggests the presence of an event handler that is invoked on the click of the button labeled “Submit.” The code for the click handler is:

```javascript
[Bindable]
public var yourName:String;

public function clickHandler(evt:Event):void {
    yourName = "Hello, " + textInput.text + "!";
}
```

Next, the source needs to be compiled. For now, I compile this code simply by pressing the Run button in Flex Builder (which is now called Flash Builder), the official Eclipse based Flex integrated development environment (IDE) from Adobe. Unlike the Flex SDK, which is an open source and freely available tool set, Flash Builder is a purchased, but inexpensive, product. IntelliJ IDEA is an emerging alternative to Flash Builder, as it now supports Flex development. IntelliJ IDEA does not support the Flash Builder style “design” mode though. No IDE at all is also an option. The Flex SDK can be used without any IDE to compile the code successfully. The compiled output using any of the preceding methods produces the same program in a .swf file format.

**Tip to Remember**

When not using Flash Builder or IntelliJ IDEA, leverage the Flex ANT Tasks, which are bundled with the free Flex 3.2 SDK, to compile Flex applications. When using ANT Tasks, you can avoid tediously specifying all necessary parameters on the command line.
Chapter 1: Flex for Java Developers

When this compiled .swf is accessed via a web browser (which hosts a Flash Player plug-in of version 9 and above), the initial screen shows up, as in Figure 1-3.

![Figure 1-3](image)

In the screen, I put in my name and click the Submit button, and I see the application greeting me by my name, as was expected. Figure 1-4 depicts this output.

![Figure 1-4](image)

Next, we refactor this simple application a bit just to emphasize the flexible data-binding aspects of Flex. The Submit button and the click handler are actually superfluous code and can be gotten rid of. The TextInput control can directly be bound to the Label that displays the output. The refactored code for the visual control takes the following form:

```xml
<mx:Label text="What's your name?">
<mx:TextInput id="textInput"/>
<mx:Label text="Hello, {textInput.text} !"/>
```

Actually, this code does not just represent the visual control but is pretty much the entire code. The only thing missing is the enclosing Application tags, which form the essential container that houses all Flex applications. The Application class also works as the entry point for Flex applications in a manner analogous to the main method in an executable Java or C++ program.

In the refactored code, none of the click handlers or the extra variables is required. This was possible because the Flex framework under the hood is implicitly triggering and handling events when we bind the text property of the TextInput control to the text property of the display Label.

On the contrary, an intermediate variable was used earlier to pass the value from the TextInput control to the display Label. This intermediate variable was declared Bindable to make sure that its value was propagated through whenever it changed. A variable, when annotated as Bindable, implicitly creates an underlying set of events and event handlers, which facilitate the data update from the source of the data to the destination at which it is consumed. Destination variables are wrapped in curly braces that represent data binding.

Before we move forward to surveying the list of available controls, let’s see the new output of the refactored application in Figure 1-5.
I guess the simple example has whetted your appetite and gotten you curious to learn more. Next, we dive into the set of controls that come off the shelf with Flex.

### Getting to Know the Controls

Controls, or UI controls, are the user interface components in the Flex framework. On the basis of functionality they could be classified into the following bundles:

- Buttons and clickable links
- Menus
- Lists
- Grids and tabular representations
- Data entry and user input controls
- Tree
- Progress and range indicators
- Image, SWF Loader, and video display
- Alert

In the following subsections, we dig in a bit deeper into each of these controls and see them in use.

### Buttons and Clickable Links

Buttons and links are typically suited for triggering an action. In some cases, this can facilitate navigation within and without the application. Six different types of buttons and clickable links are available in Flex for ready use. More such controls can be created by combining and extending these available components. In addition, you have the choice to create some from scratch by extending the `UIComponent` class, which, by the way, lies at the root of the entire set of Flex user interface classes’ hierarchy.

The seven different prebuilt button and clickable link controls are:

- `Button`
- `LinkButton`
- `ButtonBar`
- `ToggleButtonBar`
- `LinkBar`
- `TabBar`
- `PopUpButton`
To demonstrate the capabilities of these controls and understand their usage pattern, let’s build an application showcase that puts them together in one place. It’s aptly called the ButtonShowcase and its code corresponds to an MXML file by the same name. You will find this and all other code in the downloadable bundle of code associated with this book.

The first of the seven button types is a Button. A Button is a clickable control that all of us are familiar with. Creating a Button using Flex is simple and involves only a line or two of code, as follows:

```xml
<mx:Button id="button" label="Just a Button! You can click me." click="buttonClickHandler(event)"/>
```

Clicking a Button triggers a click event. A function could be defined as an event handler for this click event. This function is invoked every time the click event is triggered. In the button showcase, the click handler function merely pops up an alert on-screen, but you are likely to do more useful things within this function in a real application. The event handler for the simple button is:

```actionscript
public function buttonClickHandler(event:Event):void {
    Alert.show("The button click handler called me!", "Button Click Alert", Alert.OK, this);
}
```

A traditional button could be replaced with a link, which could demonstrate the same set of features as that of a button. Using the LinkButton class, you can quickly create a link that behaves like a button. Here is the code for it:

```xml
<mx:LinkButton id="linkButton" label="I am a Link! You can click me too." click="linkButtonClickHandler(event)"/>
```

Once you have two or more buttons, you may desire to group them together, especially if they are related. The ButtonBar class comes handy in grouping a bunch of buttons. For our button showcase, we will create four buttons labeled Flex, AIR, Flash, and Acrobat, respectively, and add them together to a bar of buttons. The process of creating this aggregation is quite simple. In our code, we don’t really create each of these four buttons individually and add them one by one. Instead, we use thedataProvider property of the ButtonBar. An Array of strings is defined and bound to a bar of buttons by assigning it as the value of the bar’s data provider. Here is the code for your reference:

```xml
<mx:ButtonBar id="buttonBar1" itemClick="itemClickHandler(event);">
    <mx:dataProvider>
        <mx:Array>
            <mx:String>Flex</mx:String>
            <mx:String>AIR</mx:String>
            <mx:String>Flash</mx:String>
            <mx:String>Acrobat</mx:String>
        </mx:Array>
    </mx:dataProvider>
</mx:ButtonBar>
```

The dataProvider property allows for a loosely coupled architecture that facilitates the separation of data and its view in the context of a control.

Data providers further the purpose of separating sources of data from the ways they are manipulated and consumed within user interface controls.
Buttons in a ButtonBar are laid out horizontally by default. Setting the direction property of the ButtonBar control to vertical lays out the constituent buttons vertically. On clicking any of the buttons in the bar, an itemClick event is dispatched. An item click handler function can be defined and bound as the event handler for this event. Our example application itemClick handler is programmed to pop up an alert dialog box, showing the selected index and the associated label. Here is the source code for it:

```typescript
public function itemClickHandler(event:ItemClickEvent):void {
    Alert.show("The item click handler at index " + String(event.index) + " with label " + event.label + " called me!",
            "Item Click Alert",Alert.OK,this);
}
```

The ButtonBar stacks constituent buttons together and acts as a convenient navigation aid. It does not maintain state though. This means it does not remember the current selection. However, another control called ToggleButtonBar comes to your rescue in situations where you need to remember the last selection. A ToggleButtonBar allows only one button from the group to be selected at a time. Making a new selection deselects the last choice and sets the new choice as the selected one. Setting the toggleOnClick property to true deselects a selected item on clicking it.

Like buttons, link buttons can be grouped in a LinkBar. Doing it is easy. Here is some code that demonstrates how to do it:

```xml
<mx:LinkBar id="linkBar" itemClick="itemClickHandler(event)">
    <mx:dataProvider>
        {dataArray}
    </mx:dataProvider>
</mx:LinkBar>
```

Like buttons and links, tabs are a common artifact used for navigation. Typically, different independent views of an application are navigated to using tabs. A TabBar control lets you define a set of tabs, where each tab is effectively an element, like a button or a link. The TabBar control has a data provider property, which helps attach data elements to the control.

So far, you have seen buttons, links, and the bars that can contain buttons and links. In addition to these Flex also includes a unique button, which has two parts, the first of which is the standard button and the second of which can be used to pop up any UIComponent. Such a button is called a PopUpButton. The typical use case is to pop up a list or menu and have the selection in the list or menu update the first button label. For our example, though, we don’t show this use case. We pop up a date chooser instead. Figure 1-6 shows the date chooser popping up on clicking the second button.
Chapter 1: Flex for Java Developers

The **PopUpButton** can be created in a line or two of code, as follows:

```xml
<mx:PopUpButton id="popUpButton" label="First Button"
    click="firstButtonClickHandler(event)"
    creationComplete="initDateChooser()"/>
```

The click handler for the first button just brings up an alert component, while that of the second button pops the date chooser component. Here is the code for it:

```ActionScript
private function initDateChooser():void {
    var dateChooser:DateChooser = new DateChooser();
    dateChooser.yearNavigationEnabled = true;
    popUpButton.popUp = dateChooser;
}

public function firstButtonClickHandler(event:Event):void {
    Alert.show("The first button click handler called me!",
    "First Button Click Alert",Alert.OK,this);
}
```

To reinforce what has been stated about the seven button controls so far, they are put together in one place in Figure 1-7.

A pictorial representation always helps when talking about user interface components.

Next, we look at menu controls, which are another useful set of controls for navigation and triggering actions.

**Menus**

A **Menu** navigation control, in Flex, pops up in response to a user’s action. It has a set of selectable units or items within it. A menu may have any number of nested submenus inside it.
Once a user selects a menu item, the Menu, by default, becomes invisible. In addition a Menu also becomes invisible when the Menu.hide method is explicitly called or a user clicks outside of the Menu component or selects another component in the application. Figure 1-8 shows what a Menu looks like.

![Figure 1-8](image)

The menu can include other controls such as radio buttons, as shown in the example in Figure 1-8. Here is the data that defines the menu items:

```xml
<mx:XML format="e4x" id="menuData">
  <root>
    <item label="MenuItem 1">
      <item label="SubMenuItem 1-A"/><
      <item label="SubMenuItem 1-B"/>
    </item>
    <item label="MenuItem 2">
      <item label="SubMenuItem 2-A" type="radio"groupName="radioMenuItemGroup1" toggled="true"/>
      <item label="SubMenuItem 2-B" type="radio"groupName="radioMenuItemGroup1"/>
      <item label="SubMenuItem 2-C" type="radio"groupName="radioMenuItemGroup1"/>
    </item>
    <item label="MenuItem 3" type="check" toggled="false"/>
    <item label="MenuItem 4" type="check" toggled="true"/>
  </root>
</mx:XML>
```

Remember that this menu pops when you click a button. It’s standard practice not to paint the menu upfront but to bring it up on some user-initiated action. This practice is one of the main reasons why you will not find an MXML component for a menu. The only way to create a menu is to do so programmatically using AS3.

An alternative to popping up a menu dynamically is to assemble it statically within a menu bar. A MenuBar component lays out the top-level menu items in a row horizontally. Again, each of the menus can have nested submenus. Figure 1-9 shows the earlier menu example within a menu bar.

![Figure 1-9](image)

To reuse the data from the Menu control in a MenuBar control you need to collate the menu items as an XMLList, which is a list of XML objects or elements. The Menu data exists as an XML document or object.
In an XML document there is a single element at the root node and that does not accommodate multiple members at the main level. A MenuBar can have multiple members at the primary level. All top-level elements in an XMLList are added as primary members on the menu bar.

Menus can also appear within a PopUpMenuButton. A PopUpMenuButton is a PopUpButton that pops up a menu when its secondary button is clicked. When a menu item is selected, the primary button label is updated with the label of the selected item. The popped-up menu cannot contain submenus in it. Listing 1-1 shows the code for a PopUpMenuButton.

**Listing 1-1: A PopUpMenuButton**

```xml
<?xml version="1.0"?>
<!-- APopUpMenuButton.mxml -->
    <mx:Script>
        <![CDATA[
            import mx.controls.Menu
            private function init():void {
                Menu(popUpMenuButton.popUp).selectedIndex=0;
            }
        ]]>[
    </mx:Script>

    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="MenuItem 1"/>
            <menuitem label="MenuItem 2"/>
            <menuitem label="MenuItem 3" type="check" toggled="false"/>
            <menuitem label="MenuItem 4" type="check" toggled="true"/>
        </root>
    </mx:XML>

    <mx:PopUpMenuButton id="popUpMenuButton" dataProvider="{menuData}" labelField="@label" showRoot="false" creationComplete="init();"/>
</mx:Application>
```

That briefly covers the menus; now we move on to lists.

**Lists**

There are three types of lists available as a part of the Flex controls. These are List, HorizontalList, and TileList. A List is a vertical display of a list of items. A user can select one or more of these items. The allowMultipleSelection property value governs if multiple selections are allowed or not. A List could be created as follows:

```xml
<mx:Model id="listItems">
    <technologies>
        <technology label="Flex" data="1"/>
        <technology label="AIR" data="2"/>
    </technologies>
</mx:Model>
```
Chapter 1: Flex for Java Developers

These same items in the list can be arranged horizontally in a HorizontalList as follows:

```xml
<mx:HorizontalList id="horizontalList" dataProvider="{listItems.technology}" columnWidth="65"/>
```

Alternatively, these items could be laid out in a grid using the TileList as follows:

```xml
<mx:TileList id="tileList" dataProvider="{listItems.technology}" columnWidth="65" columnCount="2"/>
```

List items can use an icon along with the text description. These icons, when set for the list items, are rendered in the list. Figure 1-10 depicts all the three list types. (It shows a TileList with text only, no icons.)

![Figure 1-10](image)

After lists, it’s time to survey the grid and tabular representation controls.

**Grids and Tabular Representations**

Data grids and tabular representations are one of the most popular sets of controls for data-driven Flex applications. The simplest, yet most powerful and flexible, control in this set is the DataGrid. Besides the DataGrid, the AdvancedDataGrid and the OLAPDataGrid are members of this control family. For an introductory chapter like this, we will exclude a discussion on either AdvancedDataGrid or the OLAPDataGrid.

Before we get deeper into the data grid, let’s first see a pictorial representation of a sample data grid in Figure 1-11. Then, we’ll jump right into the code that’s making it work, in Listing 1-2.
Listing 1-2: A Sample DataGrid

```xml
<?xml version="1.0" encoding="utf-8"?>
    <mx:XMLList id="flashPlayerPenetration">
        <playerVersion>
            <playerVersionNumber>Flash Player 7</playerVersionNumber>
            <matureMarkets>98.6%</matureMarkets>
            <usAndCanada>98.7%</usAndCanada>
            <europe>99.1%</europe>
            <japan>97.5%</japan>
        </playerVersion>
        <playerVersion>
            <playerVersionNumber>Flash Player 8</playerVersionNumber>
            <matureMarkets>98.3%</matureMarkets>
            <usAndCanada>98.3%</usAndCanada>
            <europe>98.9%</europe>
            <japan>97.3%</japan>
        </playerVersion>
        <playerVersion>
            <playerVersionNumber>Flash Player 9</playerVersionNumber>
            <matureMarkets>97.7%</matureMarkets>
            <usAndCanada>98.1%</usAndCanada>
            <europe>98.0%</europe>
            <japan>96.3%</japan>
        </playerVersion>
        <playerVersion>
            <playerVersionNumber>Flash Player 9 Update 3</playerVersionNumber>
            <matureMarkets>89.4%</matureMarkets>
            <usAndCanada>90.0%</usAndCanada>
            <europe>88.6%</europe>
            <japan>88.3%</japan>
        </playerVersion>
    </mx:XMLList>

    <mx:DataGrid id="dataGrid" width="100%" height="100%" dataProvider="(flashPlayerPenetration)" editable="true">
```
Chapter 1: Flex for Java Developers

A data grid allows for a lot of customization. For example, columns can use item renderers to display image data types. An application containing a data grid can listen and respond to data changes. Columns can be made resizable, and data in the columns can be sorted using user-defined logic. Considering that we are merely touching upon the exhaustive set of controls in this chapter, discussing all the features of data grid is out of scope.

The data grid in our example is editable, so it also acts as a data entry component. This supplements well the existing set of data entry components, which we talk about next.

**Data Entry and User Input Controls**

In this section I briefly present a number of data entry controls, namely:

- Text
- TextInput
- TextArea
- RichTextEditor
- Label
- CheckBox
- ComboBox
- RadioButton
- NumericStepper
- DateChooser
- DateField

Text and Label are noneditable text-rendering components. A Label can display a single line of text, whereas Text can show multiple lines. Both of them can render HTML as well as plain text. TextInput, TextArea, and RichTextEditor are all editable text input components. TextInput accepts a single line of editable text, while TextArea allows multiple lines of editable text. RichTextEditor is the most powerful of the text entry controls. It can accept HTML syntax and has many ready-to-use utilities for the formatted entry and rendering of text. Figure 1-12 shows all of these five text-data-entry-related components stacked together.

Users make selections and choices among data sets as often as they enter and edit text. Some of the most common controls that facilitate users’ making choices are CheckBox, ComboBox, and RadioButton.
A check box is a very handy control for making a selection and can be created with a few lines of code, like this:

```xml
<mx:Script>
<![CDATA[
import mx.controls.Alert;

public function clickHandler(event:Event):void {
    if(checkBox.selected == true) {
        Alert.show("Check Box is selected",
            "Check Box Click Alert",Alert.OK,this);
    } else {
        Alert.show("You deselected the Check Box now",
            "Check Box Click Alert",Alert.OK,this);
    }
}
]]>
</mx:Script>
<mx:CheckBox id="checkBox" label="A Sample Check Box" click="clickHandler(event)"/>
```

Sometimes, selection is not about choosing an individual item but making a choice from among a list or group of items. This is where a `ComboBox` and `RadioButton` are more appropriate than a `CheckBox`. Both the `RadioButton` and the `ComboBox` allow a user to make a single exclusive selection. A `ComboBox` is like a drop-down list and in some ways is a better component to go with when the list is fairly long. Besides,
it can be made editable and can allow for the addition of new items to the list if required. Figure 1-13 shows a CheckBox, a ComboBox, and a group of RadioButton controls.

![Figure 1-13](image)

Two more types of user input artifacts are included in this subsection. The first is a NumericStepper, and it facilitates users’ incrementing and decrementing numbers. It can be created as follows:

```xml
<mx:NumericStepper id="numericStepper" minimum="1" maximum="100" stepSize="1" value="50" width="45"/>
```

The next types of controls, DateChooser and DateField, deal with date and calendar elements. A DateChooser brings up a calendar component and makes it convenient for a user to select a date from it. A date field, on the other hand, is a text component with a calendar icon on its right. Clicking inside the date field or on the calendar icon brings up the date chooser component. Creating a date field could be done with just the following single line:

```xml
<mx:DateField id="dateField" yearNavigationEnabled="true"/>
```

Next, you will learn about the Tree.

**Tree**

Hierarchical data can be bound to a Tree. A Tree can have two types of elements, a leaf and a branch. A branch can contain other branches and leafs. A leaf is a terminating element and has no further branches or leafs within it. Branches have a folder icon and a label, whereas leaves have a file icon and a label. The icons can be customized if required. Listing 1-3 depicts the core features of a tree.

**Listing 1-3: A Tree Control**

```xml
<?xml version="1.0" encoding="utf-8"?>
  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;

      [Bindable]
      public var selectedNode:Object;

      public function treeChanged(event:Event):void {
        selectedNode=Tree(event.target).selectedItem;
    ]]>  
  </mx:Script>
</mx:Application>
```

Continued
Listing 1-3: A Tree Control (continued)

    Alert.show(selectedNode.@label,"Node Alert", Alert.OK, this);
    }
  ]]></mx:Script>
  <mx:XMLList id="treeData">
    <node label="Folder A">
      <node label="Folder A-1">
        <node label="Leaf A-1-1"/>
        <node label="Leaf A-1-2"/>
        <node label="Leaf A-1-3"/>
      </node>
      <node label="Folder A-2">
        <node label="Leaf A-2-1"/>
        <node label="Leaf A-2-2"/>
      </node>
    </node>
    <node label="Folder B">
      <node label="Leaf B-1"/>
    </node>
  </mx:XMLList>

  <mx:Tree id="myTree" width="50%" height="100%" labelField="@label"
    showRoot="false" dataProvider="{treeData}" change="treeChanged(event)"/>

</mx:Application>

Opening and closing of branches dispatches itemOpen, itemOpening, and itemClose events, which can be handled to implement behavior linked to the opening and closing of branch nodes. Selection of a node triggers a change event. In Listing 1-3, we use the change event listener to read the value of the current selection and pop up an alert with that data.

A tree can be used effectively like a file explorer or a navigator for any type of hierarchical data. It can be used in combination with a data grid or charts to filter and search through data.

In this rapid review, that’s all on the Tree control. Let’s keep moving on with our list of controls. We discuss the progress and range indicators next.

Progress and Range Indicators

The Flash platform is a dynamic engaging platform and there are numerous occasions, including application download and startup, when an effect, an action, or an animation is in progress for a short duration. At such times, it’s handy to know how far you are into the process. A ProgressBar is just the right display component to use to dynamically show the status of this progress. A sample progress bar that is updated through manual button clicks can be created effortlessly with a few lines of code, like this:

    <mx:Script>
    <![CDATA[
    private var i:int = 1;
    private function incrementIt(event:Event):void {
    }}
    </mx:Script>

Chapter 1: Flex for Java Developers

```java
if(i<=100) {
    progressBar.setProgress(i,100);
    progressBar.label = "Current Progress " + i + ";
    i+=1;
} else if(i>100) {
    i = 0;
}
}
```

The progress bar can be determinate if the bounds are known ahead of time or it can be indeterminate initially and can move to a determinate state once the range is assessed. This is useful when showing updates for processes like file downloads or audio and video streaming, where the total expected runtime is not known ahead of time.

Like the progress bar, the HSlider and the VSlider work over a range of data. Unlike the progress bar, though, they are not about showing the update or status of a running process. They are, instead, useful tools to help users skim over a range. For example, a user could manipulate the size of an image by moving over a possible range of resizing combinations or modify the color luminosity by gently moving over a range of possibilities. Figure 1-14 shows a horizontal slider used to manipulate an image size.

![Figure 1-14](image.jpg)

This nifty little trick with the image size is attractive, but this isn’t the only thing you can do with images.

**Image, SWF Loader, and Video Display**

In our example on the sliders, we embedded an image and played with its size to demonstrate the slider’s capabilities. When we embedded the image, it became part of the application’s .swf and bloated it
because of its size. This may not always be a desirable situation, especially if you aim to reduce the initial download and startup footprint.

The `Image` control makes it possible to load JPEG, PNG, GIF, and SWF files at runtime. The SWF file loading may surprise you, considering that the control is called `Image`. However, when you traverse the `Image` class hierarchy, things begin to make sense immediately when you notice that its immediate parent is the `SWFLoader` class, which can be used to load other Flex applications into the current one.

The image could be local and be available on the same web server that that the application’s `.swf` is served from or it could be at a remote destination and accessed over the network using HTTP.

`SWFLoader` can also load images like the `Image` class does, although it’s better to use the image control for the purpose. The image control lends itself well to being used in item renderers and item editors.

In addition to these resource loaders, you can also use the `VideoDisplay` control to load and play videos, which are available in the FLV file format. The `VideoDisplay` control includes advanced features like progressive downloading and has the capability to play a streaming video feed from a camera or the Flash Media Server. Flash Media Server is an advanced feature-rich media-streaming server from Adobe.

Listing 1-4 shows a `VideoDisplay` component in use.

**Listing 1-4: A Simple VideoDisplay Control**

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFFFFFF">
    <mx:VideoDisplay id="videoDisplay" height="285" width="250"
        source="20051210-w50s.flv" autoPlay="false"/>

    <mx:HBox>
        <mx:Button label="Play" click="videoDisplay.play()"/>
        <mx:Button label="Pause" click="videoDisplay.pause()"/>
        <mx:Button label="Stop" click="videoDisplay.stop()"/>
    </mx:HBox>
</mx:Application>
```

Now we are ready to pick up the last of the controls as far as our current exposition goes. It’s the `Alert` control, which is not a complete stranger to you because although it was never introduced, you have seen it pop up in the very first example on controls.

**Alert**

An `Alert` is a modal display component that pops up to display some text and solicit user feedback. Being modal, it forces a user to click a button to make a choice before focus is returned to the main application area. There is plenty of code already that illustrates an alert. Figure 1-15 shows a sample alert.

By now, you are exposed to a fair number of user interface components of the Flex framework and are possibly ready to start using these components together to create interesting applications. As a first step, we need to lay two or more of these components out in a way that is user-friendly and effectively spaced out in the available screen real estate.
In order to lay the components out effectively, we need to leverage the layout containers available in the framework. Layout containers are precisely the topic for the next section.

**Laying Out the Controls**

The Application enclosing element that every Flex application lies within is itself a container. It can behave as an absolute canvas or a horizontal or a vertical box. Its layout property, which decides how it should behave, takes the following as valid values:

- **absolute**: Absolute layout implies a container’s children are positioned as per their explicit x and y coordinates. If no coordinates are specified, they all stack up at the top left corner.
- **horizontal**: Horizontal layout positions a container’s children horizontally in the sequence in which they appear.
- **vertical**: Vertical layout positions a container’s children vertically. An Application container by default lays its children out vertically.

Apart from this essential and ubiquitous container, Flex offers you a varied choice from among the following:

- Box
- HBox
- VBox
- DividedBox
- HDividedBox
- VBox
- VDividedBox
- Tile
- Grid
- Panel
- Canvas
- ViewStack
- TabNavigator
- Accordion
- ApplicationControlBar
- Repeater
- Form
- TitleWindow
Chapter 1: Flex for Java Developers

Let’s start our journey, which will explore each of them. We will follow the order in which they are listed.

All the Boxes

In this subsection, we cover all the six types of boxes: Box, HBox, VBox, DividedBox, HDividedBox, and VDividedBox.

A Box container puts its child elements in a vertical or horizontal box. The direction property decides this orientation. In addition, HBox and VBox subclass the Box container class and provide ready-to-use horizontal and vertical boxes. Figure 1-16 shows a few button components in an HBox and a VBox.

![Figure 1-16](image)

Using MXML, arranging a few components in any of the box containers implies enclosing those component tags within the specific box type tags, for example:

```xml
<mx:HBox borderStyle="solid" borderColor="black"
    width="285" height="50"
    horizontalAlign="center" verticalAlign="middle">
    <mx:Button label="Flex"/>
    <mx:Button label="AIR"/>
    <mx:Button label="Flash"/>
    <mx:Button label="Acrobat"/>
</mx:HBox>
```

There are three more box containers like the ones mentioned earlier that, apart from being a box, include a draggable divider in the gap between their immediate child elements. These three containers are the DividedBox, HDividedBox, and the VDividedBox. Figure 1-17 shows the buttons from the last HBox example within an HDividedBox.

![Figure 1-17](image)
Like the Box, the DividedBox’s direction property determines the horizontal or vertical orientation of its child components. HDividedBox and VDividedBox, as their names suggest, lay out their children horizontally and vertically, respectively. Both these divided boxes inherit from the DividedBox component.

The boxes are extremely useful containers and used often in Flex applications. Sometimes though, the requirement is to arrange the child components in a grid-and-matrix-like structure. Such grid-based structures can be thought of as boxes with constraints along two dimensions instead of one.

**Grid and Tile**

The Grid container extends the Box container. It allows for arrangement of child components in a tabular layout. The layout has one or many rows, and each row can have a different number of columns. Each row is defined and identified by a GridRow component. Each row contains a number of items, which are wrapped up in GridItem components. Each GridItem can contain any number of components. Listing 1-5 is a program that creates a grid of two rows, the first row with three columns and the second with two.

**Listing 1-5: A Grid Container Example**

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFFFFFF">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            [Bindable]
            public var selectedNode:Object;
            public function treeChanged(event:Event):void {
                selectedNode=Tree(event.target).selectedItem;
                Alert.show(selectedNode.@label,"Node Alert", Alert.OK, this);
            }
        ]]>
    </mx:Script>
    <mx:Grid>
        <mx:GridRow>
            <mx:GridItem>
                <mx:Button id="row1col1button" label="Row 1 Column 1"/>
                <mx:TextInput id="row1col1textInput" text="Type Here"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button id="row1col2button" label="Row 1 Column 2"/>
                <mx:ColorPicker id="row1col2colorPicker"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:ComboBox id="row1col3comboBox">
                    <mx:dataProvider>
                        Continued
                    </mx:dataProvider>
                </mx:ComboBox>
            </mx:GridItem>
        </mx:GridRow>
    </mx:Grid>
</mx:Application>
```

Continued
Chapter 1: Flex for Java Developers

**Listing 1-5: A Grid Container Example (continued)**

```xml
<mx:Array>
  <mx:String>Flex</mx:String>
  <mx:String>AIR</mx:String>
  <mx:String>Flash</mx:String>
  <mx:String>Acrobat</mx:String>
</mx:Array>
</mx:dataProvider>
</mx:ComboBox>
</mx:GridItem>
<mx:GridItem>
  <mx:RadioButton groupName="adobeTechnologies" id="option1" label="Flex"/>
  <mx:RadioButton groupName="adobeTechnologies" id="option2" label="AIR"/>
  <mx:RadioButton groupName="adobeTechnologies" id="option3" label="Flash"/>
  <mx:RadioButton groupName="adobeTechnologies" id="option4" label="Acrobat"/>
</mx:GridItem>
</mx:GridRow>
<mx:GridRow>
  <mx:GridItem>
    <mx:List>
      <mx:dataProvider>
        <mx:Array>
          <mx:String>Flex</mx:String>
          <mx:String>AIR</mx:String>
          <mx:String>Flash</mx:String>
          <mx:String>Acrobat</mx:String>
        </mx:Array>
      </mx:dataProvider>
    </mx:List>
  </mx:GridItem>
  <mx:GridItem>
    <mx:Tree id="aTree" labelField="#label" showRoot="false"
      dataProvider="{treeData}" change="treeChanged(event)"/>
  </mx:GridItem>
</mx:GridRow>
</mx:Grid>
<mx:XMLList id="treeData">
  <node label="Folder A">
    <node label="Folder A-1">
      <node label="Leaf A-1-1"/>
      <node label="Leaf A-1-2"/>
      <node label="Leaf A-1-3"/>
    </node>
    <node label="Folder A-2">
      <node label="Leaf A-2-1"/>
      <node label="Leaf A-2-2"/>
    </node>
  </node>
</mx:XMLList>
</mx:Application>
```
Chapter 1: Flex for Java Developers

The height of all cells in a row is equal, but different rows can have different heights. Similarly, the width of all cells in a column is the same, but different columns can have different widths.

When you need to create a grid layout and need equal-sized cells throughout, it may be advisable to use a **Tile** instead of a **Grid**. Figure 1-18 shows a set of buttons spread out in a tile.

![Figure 1-18](image)

That’s about it as far as the grid-based layout is concerned; next, we explore the **Panel** container.

**Panel**

A **Panel** is a sophisticated layout container that can serve well as a top-level application container or a container for a distinct subpart of the application. Apart from the usual area for its child components, it has additional features such as a title, a status message, and a prominent set of borders. To see it in action is to understand what it’s good for. Figure 1-19 puts the buttons from the previous example in a **Panel**.

![Figure 1-19](image)

A **Panel** can be created as effortlessly as any other container. To testify to this claim, let’s peek into the code behind the screen in Figure 1-19 here:

```xml
<mx:Panel id="title" title="A set of four buttons" status="active"
    width="50%" height="50%" paddingTop="10" paddingLeft="10" paddingRight="10"
    paddingBottom="10">
    <mx:Button label="Flex"/>
    <mx:Button label="AIR"/>
    <mx:Button label="Flash"/>
    <mx:Button label="Acrobat"/>
</mx:Panel>
```
Chapter 1: Flex for Java Developers

A panel is a very useful component in most situations. Sometimes, though, all you need is an area to spread out a few controls, and in doing so you prefer to explicitly specify the position or coordinates where these contained controls need to be. The only Flex container that gives you the complete freedom to locate constituents by specifying their raw positions, that is, the x and y coordinates, is the Canvas, which is our next container on the list.

**Canvas**

Let’s go back to our four buttons, and this time anchor them at four different locations on a canvas. Intentionally, two of the buttons have a small overlap to show that the canvas isn’t doing any constraint-based layout and is just pasting its constituents where you instruct it to. In cases like the Panel or the Box container the layout is governed by rules or constraints and such layout schemes are called constraint-based layout. Look at Figure 1-20, which has the four buttons on a canvas.

![Figure 1-20](image)

The size of a child component in a canvas is set to the preferred size, that is, the preferred height and width. During the layout phase a component’s `measure` method is called to determine the preferred dimensions and the maximum and minimum values of such dimensions. You could override this by explicitly specifying the height and width either in pixels or in percentage terms in relation to the canvas.

We have been exploring single containers so far, but this is not always sufficient, as applications sometimes need to stack multiple containers and have users access them from a single entry point. Flex has a rich set of containers, which allows users to stack multiple containers.

**Stackable Container for Multiple Containers**

`ViewStack`, `TabNavigator`, and `Accordion` are the three off-the-shelf Flex containers that can host multiple containers quite elegantly. These containers stack multiple containers on top of each other and leave only one container visible at a time. `ViewStack` is a slightly raw container compared to `TabNavigator` and `Accordion`. A `ViewStack` does not come with a built-in navigator to navigate among its views like the other two. A `LinkBar`, a `TabBar`, or a `ToggleButtonBar` can act as a `ViewStack`’s navigator. This association between a `ViewStack` and any of these three link and button bars can be established just by assigning the `ViewStack` as the data provider for these navigation button sets. A simple `ViewStack` can be created as follows:

```xml
<mx:LinkBar dataProvider="{aViewStack}"/>
<mx:ViewStack id="aViewStack">
    <mx:Canvas id="canvas1" label="Canvas 1">
        <mx:Label text="Canvas 1"/>
    </mx:Canvas>

    <mx:Canvas id="canvas2" label="Canvas 2">
        <mx:Label text="Canvas 2"/>
    </mx:Canvas>
</mx:ViewStack>
```
The same three canvases can be wrapped in a TabNavigator instead of a ViewStack. When the canvases are hosted within a tab navigator, a link bar is not required. Tabs appear automatically for each immediate container within a tab navigator. Figure 1-21 is a view of such a tab navigator.

![Figure 1-21](image)

The TabNavigator class extends the ViewStack class and, therefore, exhibits all of the properties and behavior of a view stack container. The Accordion also implements all the properties and behaviors of a ViewStack, although unlike the TabNavigator it does not inherit these from the ViewStack.

Next, we put our three canvases within an Accordion. Figure 1-22 portrays a view of such an Accordion.

![Figure 1-22](image)

In all three of these containers, the selectedIndex and selectedChild properties keep track of the current visible container.

The ViewStack containers are extremely useful containers for navigating through large applications, which usually involve multiple distinct views. Sometimes, though, it’s important to keep a few application-level elements always visible across all possible views. This is where the ApplicationControlBar fits in.

**ApplicationControlBar**

An ApplicationControlBar is an application-wide container that contains elements that help navigate or provide some information that is relevant at all points of interaction with the application. An application control bar can either be docked, in which case it’s stuck to the top of the application display area or can be freely floating, in which case it can be anywhere on the screen. A docked application control bar with a button, an input text control, and a label can be created as follows:
Next, we look at another interesting container called the Repeater.

Repeater

A Repeater, as the name suggests, repeats a control; that is, it repeatedly places a control. Therefore, if you wanted to create a set of buttons that were numbered from 1 to 11, it would be useful to use a repeater. Such an application can be created by using some elementary code, such as this:

```xml
<mx:Script>
  <![CDATA[
    [Bindable]
    public var anArray:Array=[1,2,3,4,5,6,7,8,9,10,11];
  ]]> 
</mx:Script>
<mx:Repeater id="aRepeater" dataProvider="{anArray}"
  <mx:Button id="aButton" label="Button {aRepeater.currentIndex + 1}"/>
</mx:Repeater>
```

Next, you will learn about a container that is ubiquitous on the Web: a Form.

Form

We are all familiar with the forms for data entry on the Web so forms need little or no introduction. The Form container, along with FormHeading andFormItem, allows you to control the layout and behavior of the form elements within a Flex application. Figure 1-23 shows a simple form for data input.

![Figure 1-23](image)

The code behind this form is:

```xml
<mx:Form width="100%" height="100%">
  <mx:FormHeading label="Just a Form Heading."/>
  <mx:FormItem label="First name">
    <mx:TextInput id="firstName" width="200"/>
  </mx:FormItem>
</mx:Form>
```
Chapter 1: Flex for Java Developers

```xml
<mx:FormItem label="Last name">
    <mx:TextInput id="lastName" width="200"/>
</mx:FormItem>
<mx:FormItem label="Date of birth (mm/dd/yyyy)">
    <mx:TextInput id="dob" width="200"/>
</mx:FormItem>
<mx:FormItem label="E-mail address">
    <mx:TextInput id="email" width="200"/>
</mx:FormItem>
</mx:Form>
```

It's easy to specify elements as mandatory or optional in a form and attach validators and formatters to them where required.

In this brief tour of containers, we have come close to the end of the journey. Our last container in the list is `TitleWindow`, which we look at next.

**TitleWindow**

Resembling a `Panel` to some extent, a `TitleWindow` is a container that includes a title bar, a border, a caption, and a content area. Distinct and independent parts of an application can be housed in a `TitleWindow`. Figure 1-24 shows a `TitleWindow`.

![Figure 1-24](image)

A `TitleWindow` can appear as a regular container in an application or as a pop-up for the duration of the interaction. Its usage really depends on the demands of the application.

In the last few pages, we surveyed the containers that come with Flex. Each of these containers can be extended to create new containers. In addition, these containers can be combined and used together to create many different types of layouts.

So far you have seen the controls and the containers. Together, these cover the heart of the Flex user interface story. However, the controls and containers are not enough to create a fully functional rich application. Our next topic is about the ways and means to consume external data and interface with the world outside a Flash Player.

**Leveraging Services to Access External Data**

Except for the most trivial or rare situations, application data will almost always reside outside your Flex application. It could be in a relational database, a file system, or some other persistent data store. Whichever form it is in, you would require it to be available over a network-accessible resource like a web server. In addition, you would require that the server be a safe and trusted destination that respects the security sandbox requirements of the Flash Player. Web servers at remote destinations define the required security policy in a file called `crossdomain.xml`, which lies at the root of the server.
Embedded Data

Data files can be embedded into Flex applications and distributed as a part of the application's .swf. However, doing so bloats the application size and thereby delays the application startup. In addition, the data set is static, and any modification to it requires recompilation.

Using Simple HTTP and REST Style Invocations

One of the simplest ways to consume external data in Flex applications is to leverage the established methods that make the Web work. Flex applications are capable of making HTTP calls to external web servers and fetching documents, data, and resource representations from them. A component called HTTPService facilitates the process of making requests to and handling responses from a web server.

To illustrate further, we will walk through a simple example, in which we leverage the Geocoder.us free web service. Geocoder.us provides free and commercial services to obtain the longitude and latitude of a given address. It's possible to communicate with the geocoder service using any of the multiple available and supported formats, namely XML-RPC, SOAP, REST-RDF, or REST-CSV.

To keep things simple and focus on the behavior of the HTTPService component, let's build our example around geocoding a specific address: the corporate headquarters of Adobe at 345 Park Avenue, San Jose, CA 95110-2704. In addition, let's stick to the simple and straightforward REST-CSV format for data interchange.

REST-CSV format has two facets: REST and CSV. REST is an architectural style that promotes a design and style of communication that resembles the world wide web. In other words, it leverages the standard HTTP methods and verbs and prefers idempotent stateless representations. REST is explained in a sidebar later in this section. CSV stands for “comma separated values,” which is essentially a simple text format that uses commas as a delimiter.

First, test that the geocoder service is accessible and working. This can be done by pointing the browser to the following URL: http://rpc.geocoder.us/service/csv?address=345+Park+Avenue,+San+Jose+CA.

You should see a response as follows:

37.329407,-121.895057,345 Park Ave,San Jose,CA,95110

Next, make this identical call from within a Flex application. Listing 1-6 shows one possible way to make this call.

Listing 1-6: Communication with Geocoder.us via the HTTPService component

```xml
<?xml version="1.0" encoding="utf-8"?>
  backgroundColor="#FFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.rpc.events.ResultEvent;
      ```
public function resultHandler(event:ResultEvent):void {
    aTextArea.text = anHTTPService.lastResult as String;
}

<mx:Script>

    <mx:HTTPService id="anHTTPService"
        url="http://rpc.geocoder.us/service/csv?address=345+Park%20Avenue,
        +San%20Jose+CA"
        useProxy="false"
        result="resultHandler(event)"/>

    <mx:Button label="Send Request to GeoCoder"
        click="anHTTPService.send();"/>

    <mx:TextArea id="aTextArea" width="350" height="200"
        text="This is where you will see the response from the GeoCoder service...." />

</mx:Application>

Traversing the code, you will notice that the click of a button invokes the HTTPService send method, which sends a request and the response, when returned, shows up in a text area. HTTPService, as with most service invocators in Flex, follows an asynchronous communication mechanism, where it sends a request and does not wait for a response. It returns with a mere token. When the response returns, the token reconciles responses to requests. In many simple Flex applications — like our example above — the request to token and then the token to response matching occurs seamlessly, behind the scenes. A result event is activated if the response returns successfully. Otherwise, a fault event is triggered, if the request fails and terminates in error. You define the result and fault handler methods and bind them as event handlers to the appropriate events.

The HTTPService component is quite capable of handling most types of HTTP requests and responses. It allows both GET and POST HTTP method requests. It can deserialize the response into many different formats. Figure 1-25 illustrates the different formats in which results can be returned and accessed using the HTTPService class.

URL(s) can be defined in absolute or relative terms. The content type can be set to send requests using the HTTP POST name-value pairs or XML. Custom HTTP headers can be set for requests, and request timeouts can be manipulated for calls that may take longer than usual to complete.

The HTTPService component can thus be used gainfully to make most REST-style service calls. It does not work in every REST-like situation, though, especially when HTTP verbs other than GET and POST, say PUT and DELETE, are required or when existing HTTP headers need to be manipulated. Open source third-party components such as as3httpclientlib (http://code.google.com/p/as3httpclientlib) and the URLLoader class can possibly help in some of these situations where the HTTPService fails.
What Is REST?

REST, which is an acronym for Representational State Transfer, is an architectural style for building distributed hypermedia systems like the Web. It was introduced as a formal concept by Roy Fielding in his PhD thesis, which is accessible online at www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. REST views HTTP responses as state representations of networked resources. It promotes the use of simple, idempotent, and stateless URL(s) for accessing these resources and supports the use of HTTP methods like GET, POST, PUT, and DELETE as the primary verbs for communication. The thesis cited earlier is the best and most authentic resource to learn more about this architectural style.

How is the value returned?

<table>
<thead>
<tr>
<th>HTTPService resultFormat</th>
<th>Text (Including Name-Value Pairs as text)</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>flashvars</td>
<td>object, array (Array as top-level object)</td>
</tr>
<tr>
<td>Text</td>
<td>text</td>
<td>e4x (ECMA4 for XML), xml (For legacy support of XMLNode)</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REST has become increasingly popular for web services in the last few years because of its simplicity and elegance. The HTTPService component and other related artifacts help Flex applications work with this architectural style. Sometimes though web services support the more formal and elaborate SOAP (Simple Object Access Protocol) model. Flex supports communication with SOAP web services as well. That is our next topic.

**Communicating via Web Services**

At the heart of the SOAP web services support is the WebService class. SOAP web services publish their service endpoints using an XML-based description syntax called WSDL, Web Services Description Language. The WebService component is an object-oriented abstraction for interaction with a SOAP web service, and it incorporates methods and events for loading WSDL, making remote calls, and handling results.

The load event is dispatched on the WebService component when a WSDL is loaded. This is when you are ready to initialize and make the remote web service procedure calls. Each remote call either
Chapter 1: Flex for Java Developers

succeeds or fails. As with the HTTPService, a result or a fault event is activated, depending on whether the response was a success or failure. A single web service endpoint can define multiple methods or operations. It’s possible to define different or the same result and fault handlers for these operations.

In addition to this infrastructure, Flash Builder offers a feature to automate client stub or proxy generation for web services, based on the WSDL. Time now to delve into an example to see how all this works. For the example, I use the Flash Builder client stub generation facility to speed things up and avoid some hand-coding.

First, I select the Flex application that contains the code for Chapter 1. Then I open up the wizard to import the web service WSDL and generate the client stub by clicking through the Data->Import Web Service (WSDL) menu items.

The first wizard screen prompts me to choose the source folder, which of course is the project source folder. On selection, I move forward to the screen shown in Figure 1-26. On this screen, titled Specify WSDL to Introspect, I specify the address of the WSDL and then let Flash Builder introspect the endpoint definitions.

You will see all endpoint options come up on the next screen. The wizard will only let you select and generate client classes for endpoints that adhere to SOAP 1.1. For this example, I pick up a free public web service that allows looking up weather information by zip code and place name. Figure 1-27 shows a snapshot of the SOAP endpoint definitions for this service, which are presented after introspection. At this time, we are ready to generate the AS3 classes that represent the web service client stub or proxy, simply by clicking on the Finish button.

Figure 1-28 lists the classes that get generated in the Flex project source folder. Now these classes can be used to make remote calls on the web services operation.

Being constrained by both the quest for brevity and the scope of this chapter, we conclude the possibly long discussion on SOAP web services right here. Later, in Chapter 3, we pick up the topic of web services one more time.
The next section touches on the options of tighter integration between Flex and server-side platforms for data interchange.

**Making Remote Procedure Calls**

In addition to the simple HTTP requests and the web services, Flex also supports tightly coupled inter-language remoting with most of the popular server-side environments. Since this book is about Java and JVM integration with Flex, we will restrict the discussion to only those topics. Also, it is the central topic of this book, so obviously we will not discuss anything more than a simple definition in this section.

Flex clients can call methods on remote Java objects. These remote methods can return Java objects, which are converted to AS3 objects and served up to the Flex application. The transmission and communication between Flex and the JVM is accomplished through a binary protocol called AMF (Action Message Format), which works as an application-level protocol over HTTP. In addition, to regular remote procedure calls, Flex clients can register interest in server-side topics. A server program can send a message to a topic that a Flex client has subscribed to. Such a message is then pushed up to the client.

LifeCycle Data Services (LCDS) is a piece of commercial software from Adobe that facilitates remoting- and messaging-based integration between Flex and Java. BlazeDS is an open source alternative to LCDS. BlazeDS shares much of the code base with LCDS. Adobe created this open source offering. It continues
to enhance and support this product. BlazeDS is the primary integration layer that we talk about in this book. Apart from its off-the-shelf benefits, it provides easy methods for extension and adapter development to support a wide range of enterprise Java requirements. You will learn a whole lot about BlazeDS and Flex-Java integration in the following chapters.

Before we conclude this section and move on to the topics on making interfaces engaging and effective, it is important to mention that LCDS and BlazeDS are not the only pieces of software that help Flex and Java work together. Many alternative open source solutions, including Granite Data Services (GDS) and WebORB, achieve the same goal.

---

**Where Can I Learn More about GDS and WebORB?**

Granite Data Services (GDS) — www.graniteds.org

WebORB for Java — www.themidnightcoders.com/products/weorb-for-java/overview.html

You will learn more about LCDS and BlazeDS in the following chapters.

---

**Making User Interfaces Engaging and Effective**

In this part of the chapter, we explore a few subjects that facilitate sprucing up of user interfaces to make them more productive. By no means are these either exhaustive or detailed.

**Exploring Effects**

An effect is a dynamic behavior that is applied to a target component when an effect is triggered. Therefore, a trigger and a target are the two complementary parts of an effect. An effect is activated or triggered in the same manner as an event, although it is distinct from it. For example, both a `creationCompleteEvent` and a `creationCompleteEffect` are activated when an application is created, initialized, and set up. On an effect trigger, a specific effect is applied to a target. As an example, you may want a label to glow when the `mouseDownEffect` is triggered on that label. The label in this case is the target for the effect. Listing 1-7 shows how to accomplish such an effect.

**Listing 1-7: Glow on a Mouse Down Effect**

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFF0000">
    <mx:Glow id="glowEffect" duration="1000"/>

    <mx:Label id="firstLabel" text="First Label"
        mouseDownEffect="{glowEffect}"/>
    <mx:Label id="secondLabel" text="Second Label"/>

</mx:Application>
```
A few common effects available in Flex are:

- AnimateProperty
- Blur
- Dissolve
- Fade
- Glow
- Iris
- Move
- Pause
- Resize
- Rotate
- SoundEffect
- WipeLeft
- WipeRight
- WipeUp
- WipeDown
- Zoom

In addition, to these built-in effects, one can also create custom effects if desired. Effects have a duration property that determines the time for which the effect is applied to a component.

I mentioned earlier that this section explores a few features that make a user interface more responsive. However, the one thing I did not say then was that these features are a random selection from an exhaustive lot and may not necessarily relate directly to each other. The next feature that we look at is validators, which allow for quick feedback to users during data entry.

**Validating User Input**

Validating input data is critical for maintaining data quality. Over the last few years, web applications have used both client-side and server-side validation to achieve this. Flex, being a client-centric technology, validates data right at the user end. This allows for rapid feedback when a user inputs data.

Flex comes with a few ready to use validators. These are as follows:

- CreditCardValidator
- CurrencyValidator
- DateValidator
- EmailValidator
- NumberValidator
- PhoneNumberValidator
- RegExpValidator
Chapter 1: Flex for Java Developers

❑ SocialSecurityValidator
❑ StringValidator
❑ ZipCodeValidator

Their names are descriptive enough for you to get a sense of what they validate. Their usage is fairly straightforward, too. For example, you could implement an EmailValidator as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFFFFFF">
    <mx:EmailValidator id="anEmailValidator" source="{emailInput}" property="text"/>
    <mx:TextInput id="emailInput"/>
    <mx:TextInput id="anotherTextInput"/>
</mx:Application>
```

When you navigate away from the first text input, where you enter the email address, the email validator kicks in. It verifies validity by checking for the "@" and the "." (period) symbols in the text.

You can also create custom validators and define your own custom constraint and rules to validate input data.

Next, we look at another feature for increasing richness, and this time it’s for customizing the look and feel.

**Customizing the Look and Feel**

The look and feel of a Flex application can be customized using styles and skins. Web developers familiar with CSS (Cascading Style Sheets) can style Flex applications the same way they style their HTML-technologies–based web applications. Styles are defined declaratively in a CSS file. In Flex, the style can reside in an external file or can be inline. For example the font size, color, letter spacing, and font weight of a Label can be specified in an external file called myStyle.css as follows:

```xml
Label {
    fontSize: 15;
    color: #FFFF00;
    letterSpacing: 2;
    fontWeight: bold;
}
```

This external file is imported into a Flex application using the Style tag:

```xml
<mx:Style source="../assets/myStyle.css"/>
```

Instead of an external file, the same style can be declared within a Flex application inline as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFFFFFF"/>
```
Chapter 1: Flex for Java Developers

When using either of these techniques, the style is applied to all instances of type `Label`. This is convenient because most often a uniform look and feel is what you desire. However, at times you may want a certain `Label` to override the style attribute. You can do this by using the style properties within a label component. For example, you could create a `Label` with `fontSize` 18 instead of 15 using the following code:

```xml
<mx:Label id="anotherLabel" text="This is another label" fontSize="18"/>
```

Styles can also be defined programmatically using the `StyleManager` class or using the getters and setters for styles of a component. Again, because we want to keep things simple we won’t cover these here.

In addition to styles the look and feel can also be defined in terms of aggregations of styles, skins, and associated artwork. Such aggregations are called themes. The default Halo theme that comes with the Flex framework is one such theme.

Using styles and customizing the look and feel is one way to create custom components. There are many more ways to customize components though. That’s what you’ll learn next.

### Extending Flex

The Flex framework is a feature-rich and robust framework for enterprise RIA development and is often sufficient off the shelf for many simple development tasks. However, it also derives power from the fact that it can be extended and customized with ease.

### Creating Custom Components

Creating custom components is likely to be one of the most frequent tasks you will indulge in the moment you start building complex and enterprise-grade RIAs. Sometimes, the motivation behind custom component development is to create reusable pieces of code for productivity enhancements and sometimes it is for feature enhancement and modified component behavior.

The simplest use case for custom components is the aggregation of specific components at one place or the aggregation of certain component types and data. For example, you may see benefit in creating a component that includes a `ComboBox` that has East, West, and Central as its values. Such a custom
ComboBox could be used readily in sales and marketing applications that classify data by regions within the United States. The code for this kind of a ComboBox would be as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="0xFFFFFFF">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var regions:ArrayCollection = new ArrayCollection(
                [{label:"East", data:1},
                {label:"West", data:2},
                {label:"Central", data:3} ]);
        ]]>
    </mx:Script>
    <mx:ComboBox dataProvider="{regions}" width="250"/>
</mx:Application>
```

Another use case is to create a reusable contact form that validates email addresses and makes sure that the name field is mandatorily filled in. There are many more uses like this.

Sometimes it may make sense to add new features to existing components and use AS3 to extend the control classes. One such case is to extend the Button class and add a property to keep count of the number of times a button has been clicked. The code for such a custom button is:

```as3
package as3classes {
    import mx.controls.Button;

    public class ButtonWithClickCount extends Button {
        public function ButtonWithClickCount() {
            super();
        }

        [Bindable]
        private var _clickCount:int=0;

        public function set clickCount(value:int):void {
            clickCount = value;
        }

        public function get clickCount():int {
            return clickCount;
        }
    }
}
```
If extending an existing control does not get the job done, then a control could be written from scratch by extending the `UIComponent` class. You could also use and learn from the many open source custom components available online. The `flexlib` project is such a collection (http://code.google.com/p/flexlib).

In addition to aggregating and extending components, custom events can be added and propagated.

### Creating and Propagating Custom Events

Events are the primary drivers of interactivity in a Flex application. All Flex controls dispatch events at various points of user interaction. Event handlers act on these triggers and respond to user inputs. In addition to the standard events, the framework also allows for the creation and dispatching of custom events.

Custom events can be created as follows:

```plaintext
package as3classes
{
    import flash.events.Event;

    public class CustomEvent extends Event
    {
        public function CustomEvent(type:String,
            bubbles:Boolean=false, cancelable:Boolean=false)
        {
            super(type, bubbles, cancelable);
        }

        public static const CUSTOM_EVENT:String = "customEvent";
    }
}
```

Then this custom event can be instantiated like this:

```plaintext
new Event(CustomEvent.CUSTOM_EVENT)
```

Such instantiation also ensures type checking. Events, both standard and custom, can be dispatched and propagated using the `dispatchEvent` method. This is all as far as the bare essentials of custom events are concerned. Since discussion of custom events is an advanced topic, the current coverage will suffice.

By now you are well versed in the essentials of the Flex framework. I am sure you are enthused to learn more and are maybe disappointed by the brief coverage of many topics that may have intrigued your interest. Our focus is on integration, and as we explore that thoroughly in the many chapters to come, you will continue to learn bits of Flex. However, if Flex itself is something you want to sharpen your skills on, then you are better off reading one of the books specifically on that topic and browsing through the Adobe Flex language reference, which is available at http://livedocs.adobe.com/flex/3/langref.
Summary

This is the first chapter of a book that talks about integrating Flex and Java to create an RIA. Because it is the first chapter, it did not address the central theme but worked on the essential prerequisites for the rest of the book. This chapter was an introduction to the Adobe Flex technology itself.

The fundamentals of Flex were explored. The exploration started with controls and containers, which are the core of the framework. The treatment was cursory and mostly restricted to definitions and elementary examples.

After the controls and containers, the topic of external data consumption was explained briefly. The different levels of communication and conversation between Flex and the server side was illustrated, ranging from loose HTTP-based requests to remoting. Again, the treatment was an overview.

The last parts of the chapter brought out the features in the framework that facilitate the creation of engaging and rich interfaces and that allow for extension of the framework.

That was it as far as this chapter goes. Remember, though that this is just the beginning, and the main topic is about to begin!
Introducing BlazeDS

The Adobe Flash platform with its frameworks, Flex and AIR, provides a robust set of tools, libraries, programming idioms, and runtime for developing and deploying Rich Internet Applications (RIAs). However, that’s not all you need to build an enterprise-grade RIA. You probably know well that data, more often than not, resides in external persistent stores such as relational databases, and lots of business logic resides in middleware applications that run in traditional server environments. Besides, server-centric distributed computing environments provide services from authentication, transaction management, communication, and message based integration to hooks that make interaction with web services and cloud computing farms possible. Therefore, integration with the server-side environments is pertinent and essential in most, if not all, complex RIA development scenarios. So, there is a need for a bridge to make Flex and AIR talk to various server environments, say those that run Java, PHP, Ruby, Python, or the .NET platform.

This is where BlazeDS fits in. BlazeDS is a piece of open source software that facilitates integration of Flex and AIR with Java server environments. BlazeDS enables Flex and AIR to talk with Java servers using remote procedure calls and asynchronous messages.

This chapter gently introduces BlazeDS and gets you ready for the under-the-hood details and integration wizardry that you will learn in the following chapters.

Here, and in following chapters, I will talk about Flex and Java. In all these cases, Flex implies both Flex and AIR. However, there are some cases that are relevant only for AIR, and I will be explicit when those situations show up. With that clarification, let’s jump right in to explore BlazeDS.

An Overview of BlazeDS

You know by now that BlazeDS helps integrate Flex and Java, but what exactly is BlazeDS? To answer this question, we define BlazeDS using its behavioral and structural characteristics. The behavioral aspects will establish what it’s good for, and the structural facets will explain what it’s made of.
Behavioral Definition:

BlazeDS enables and facilitates:

- Invocation of remote Java methods from a Flex application.
- Translation of Java objects returned from a server, in response to the remote method call, to corresponding AS3 objects.
- Translation of AS3 objects sent by a Flex application to corresponding Java objects for passing them in as method call arguments.
- Communication between a Flash Player instance and a Java server over a TCP/IP-based application-level binary protocol.
- Near real-time message passing between a Flex application and a Java server.
- Management of the communication channels between Flex and Java.
- Management of connection types between Flex and Java.
- Provision for adapters for communication with server-side Java artifacts like JMS queues, and persistence layers like Hibernate and JPA. (Some of these are in-built, and some can be obtained from open source projects or can be custom built.)
- Pushing data from the server to the client on the server’s initiative and not as a response to a request.

Structural Definition:

BlazeDS is a:

- Java web application that leverages the Java Servlets specification.
- Web application that runs within a Java Servlet container or a Java application server, for example Apache Tomcat, JBoss AS, IBM Websphere or BEA (now Oracle) Weblogic.
- Set of services that can be managed using JMX agents.
- Remoting and messaging program that can be extended by using its Java API.
- Program that intercepts all communication between a Flash Player and a Java server.
- Configurable web application that can be clustered and used in cases that desire a higher than normal performance. (The in-built data push mechanism has a few limitations as far as high throughput and high volume goes but there are ways to get around this shortcoming.)

This definition says a lot of things about BlazeDS. If you are a complete newbie, you are possibly overwhelmed by the number of different functions BlazeDS can perform. So, to avoid confusion and make things digestible, I delve next into the role and applicability of BlazeDS within an application.

The first journey into BlazeDS applicability exposition is from the metaphorical aerial height of 30,000 feet.
Chapter 2: Introducing BlazeDS

Viewing Integration from 30,000 feet

From this level, BlazeDS can be viewed as an enabler of two types of interactions between Flex and Java. These are:

- Requests and responses to those requests
- Exchange of messages

Traditional web applications engage in request-response–based communication, where the client (typically the browser and generically identified as the user-agent) makes a request and the server responds to this request. Communication is synchronous, which means that the response comes back in the same cycle as the request. Flex improves on this traditional style of communication and can use this improved style to communicate with a Java server.

Flex facilitates request-response–style communication over asynchronous mode. Read the side note “Think Post Office” to understand the concept of asynchronous communication in Flex through an analogy.

Flex can communicate with a Java server using the asynchronous request-response–style with and without BlazeDS. When communicating without BlazeDS, it’s a requirement that the Java server be accessible over a trusted domain. A trusted domain is a domain that defines an appropriate security policy to allow the Flash Player to communicate with the server. The security policy is defined in an XML file, called crossdomain.xml.

When using BlazeDS, the asynchronous request-response communication between a Flex client and a Java server allows for the following:

- Requests from Flex clients to server side objects leading to invocation of server side procedures and methods
- Return of Java objects marshaled and translated into their AS3 counterparts over the wire
- Passing in of AS3 objects, which are marshaled and translated into their Java counterparts, as parameters to the remote Java server procedures
- Communication over HTTP using an application-layer binary protocol, called Action Message Format (AMF), that speeds up the data exchange

The second type of interaction between Flex and Java can be over messages, which can be sent by either and received by the other. This facility allows for server initiated data push in addition to the traditional request-response communication.

Think Post Office

A good way to understand the asynchronous communication in Flex and AIR is to look at the offline communication pattern that unfolds when we post a letter.

Say you live in New York at an address that reads as follows: John Doe, 1234 A Famous Avenue, New York, NY 10001, and you wish to send a letter out to your grandmother
in San Francisco, California. Her address happens to be: Mother Doe, 4321 Another Famous Avenue, San Francisco, CA 94101. Now you want to make sure that the letter reaches her, so you avail yourself of the “return receipt” facility offered by the postal service.

How do you think the letter and the return receipt move through the system? Do you wait at the box where you drop the mail to receive the receipt? Of course not!

After you drop the letter in a post box, it gets picked up by the postal service and is put into a sorting queue. From the ZIP code and the address, the postal service figures out that your letter needs to go to a place in San Francisco, CA. So, it’s dispatched through some optimal routing mechanism in that direction. Once it reaches CA, it’s sorted again so that the correct postal delivery person can hand it over to your grandmother. Because you asked for a return receipt, the postal delivery person also asks your grandmother to sign and confirm the receipt of the letter. Then that return receipt is put back into the postal system.

The return receipt has your address, so it moves through the sorting queues and the routing logic back to your place. A different postal delivery person, the one that services your neighborhood and not your grandmother’s, puts the return receipt in your mailbox. Finally, your grandmother has the letter and you have the receipt that confirms that she did receive your letter. Wonderful, but what has this letter exchange got to do with Flex?

Crudely speaking, the Flex infrastructure works pretty much like the postal system. When you send a request up, locally or remotely, it first returns a token to you and then gets down to invoking your request. It does not wait for the response to come back and does not block any resources. Remember, you don’t stand at the drop box to receive the return receipt.

When a response returns, the Flex infrastructure reconciles the response to the initiating request, using the token and the responder that is registered against the token. If the response is a success, a “result” event is triggered. A registered responder listens to this event. On “result” event, a listening responder can invoke the appropriate handler and extract the response and apply it as desired. Similarly, if the response is a failure, a “fault” event is triggered. Again the responder listens to this event, and it can handle it in a manner similar to how it handles the “result” event.

So, after all, Flex is much like a postal system, only mutated to address the communication needs of distributed objects in the digital age. What do you think?

Therefore, both the “pull” communication style of requests and responses and the “push” communication style of messages are possible.

Figure 2-1 depicts this 30,000-foot viewpoint in a simple diagram.

Next, we zoom in closer and view BlazeDS from the 10,000-foot level.
Chapter 2: Introducing BlazeDS

Viewing Integration from 10,000 feet

As we drill into the artifacts on the Flex client side that make the request-response and message-based communication possible, we end up with list that includes the following:

- HTTPService
- WebService
- RemoteObject
- MessageService

The first two on this list, HTTPService and WebService, work with and without BlazeDS. BlazeDS and these two components come together when there is a need to proxy calls to a server destination that does not define a security policy as required by the Flash Player. You will learn more about BlazeDS proxy service in the next chapter.

RemoteObject and MessageService, on the other hand, are the primary Flex-side workhorses in the context of BlazeDS. RemoteObject facilitates request-response communication and MessageService enables publish-subscribe based message exchange. RemoteObject acts as a proxy or a stub for server-side services and distributed objects. MessageService provides a handle to a remote messaging destination and a message queue.

Figure 2-2 shows the Flex and Java integration story from the 10,000-foot level.

We continue to zoom further, till we reach the 1,000-foot level. At this level, things are close enough to get a view of some of the details.

Viewing It Closely from 1,000 Feet

You may benefit from looking at Figure 2-3 first. Figure 2-3 attempts to lay out the individual elements involved in the interchange between Flex and Java.
At a cursory level, you are familiar with the flow and the involved components from last couple of subsections. We will dig much deeper into these facets in the rest of this book. However, for now I will stop digging and move to the tasks around getting and setting up a BlazeDS instance.

I am firmly in favor of your setting up BlazeDS, understanding the essential configuration, and working out a small preliminary example, before you start diving deeper. Learning by doing can be very effective, and I am keen to help you try that method out when learning BlazeDS using this book.
Chapter 2: Introducing BlazeDS

Getting and Installing BlazeDS

So far, you know of the core parts of BlazeDS and understand the context within which it is most relevant. Now it’s time for you to roll your sleeves up and start seeing it in action. The first step, of course, is to get hold of it.

**Downloading BlazeDS**

BlazeDS is an open source project. Its source and compiled distributions are available online for anyone to download. Go to [http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/](http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/), and navigate through the links on the “Download” section of the page to download either the compiled or source versions of the code.

When you click on the “Download BlazeDS now” link, you will be taken to the download page, which is also accessible directly at: [http://opensource.adobe.com/wiki/display/blazeds/Downloads](http://opensource.adobe.com/wiki/display/blazeds/Downloads). On that page, you will find two types of downloads available for you, namely:

- **Release Builds**
- **Nightly Builds**

The release builds are the ones that are stable, tested, and ready for consumption, without fear of the code breaking in a catastrophic manner. Release builds can be of two types, milestone releases and stable builds. Stable builds are typically well tested and have worked without trouble for a considerable amount of time. Milestone releases are also well tested like stable builds but possibly are less time tested than stable builds. A milestone release typically includes a significant new feature or an important bug fix as compared to an earlier release version.

The nightly builds on the other hand are recent compilations from the source. The nightly builds are typically more unstable and flaky and can sometimes even break existing functionality. As a rule of thumb, a rookie or a developer not necessarily wanting to be at the bleeding edge of the evolution of BlazeDS should go with the latest version of the release build.

The BlazeDS project makes three types of release builds available for download. These are:

- **Turnkey** — BlazeDS deployed within an Apache Tomcat Servlet container, configured and ready to run
- **Binary or Compiled Distribution** — BlazeDS available as a compiled WAR (Web ARchive) file
- **Source Distribution** — BlazeDS source with the build scripts to compile it from source

Each release type, that is, Stable, Milestone, and Nightly, include these three types of distributions. Apart from these, which provide plenty of choices among the different types of distributions, there is an additional choice related to the software version.

The current development version of BlazeDS is 4.x, whereas the latest release version is 3.x. Therefore, you will have a choice of downloading the 3.x or 4.x version as far as the nightly builds go. Version 4.x builds may slowly also start showing up in milestone releases and will subsequently be available as stable releases.
Chapter 2: Introducing BlazeDS

As of now, all the new features are coming in the 4.x version, which resides in the repository trunk, while bug fixes and enhancements continue to evolve the stable 3.x version code in an independent repository branch.

If you are simply trying out BlazeDS or don’t have the patience to configure it, then go with the “Turnkey” download option. It’s the easiest and the fastest way to start using BlazeDS. Remember though, that you will have to move your server-side code within this new instance of Tomcat if you desire to access the server components locally within the same web components container.

All three release builds are available in .zip archive file format. On the Windows platform, WinZip can be used to unzip such files. If you don’t have WinZip on your machine, then get 7Zip from http://www.7-zip.org/. 7Zip is also available as a command-line utility called p7zip for the Mac OSX and many of the Linux/UNIX platforms, so it works well for unzipping files in most situations.

You know that BlazeDS is an open source project. Its source code repository hosted in a subversion version control system instance is accessible to public. Therefore, while builds are one way of getting BlazeDS, checking code directly out of the repository is also an option.

Subversion, popularly known as SVN, is a feature-rich open source version control system. If you know nothing or little about it, then consider reading the free book, available online at http://svnbook.red-bean.com/, to understand how it works.

Subversion’s core architecture is based on a client/server model. The code repository lies at the server and clients connect to this server to access and modify this code base. Free and open source SVN clients exist for most platforms. Clients come in various forms, including the following:

- Stand-alone clients
- Desktop integrated clients
- Plug-in to an IDE client

Client programs can have a command-line interface or a graphical user interface (GUI). An entire list of available SVN client programs is accessible at http://subversion.tigris.org/links.html#all-clients.

I prefer to use TortoiseSVN on Windows, SmartSVN on Mac OS X, Linux, and UNIX and Subclipse within the Eclipse IDE. In addition, I recommend using the command-line client, especially when the command or the operation on SVN is resource intensive. Almost always, the svn command-line client runs faster than its GUI counterparts.

The BlazeDS repository is accessible at http://opensource.adobe.com/svn/opensource/blazeds. This URL takes you to the root of the repository. Avoid checking out code at this level, as you will end up downloading a rather large bundle of code, much of which would be redundant. This is because checking out from the repository root will lead to getting all the code from the trunk, the branches and the tags. It’s highly unlikely that you need all of the code at once.

To check out the latest source from the trunk to a local folder, invoke the following SVN command:

```
```
You may recall that, as of now, the trunk code is the version 4.x code. If you want to check out the latest version of the evolving 3.x branch instead, then you invoke a command like this:

```
```

In addition, you will be able to check out the source for a previous release using a subversion client. Previous releases in the repository are tagged using their version numbers. This is a usual convention with source control management, especially when using subversion. Read more about this convention in the side note titled: Trunk, Branches, and Tags. To check out source using a tag, invoke a `svn` command as follows:

```
svn checkout http://opensource.adobe.com/svn/opensource/blazeds/tags/<tag-name>/ <path-to-the-desired-local-folder>
```

### Trunk, Branches, and Tags

Subversion, by default, has three folders in a code repository. These are: trunk, branches, and tags. The main destination of the code repository is the trunk. It's the primary code base where active development goes on. As the code evolves, the development team may decide to release a piece of it. At that stage, it makes sense to fork this code out into a separate branch of its own. By doing this, the evolution on the trunk continues per the larger plan, and progress on bug fixes and enhancements continues on the released code within the specific branch. Release and branching out of code is usually accompanied with tagging releases. Tags make it easy to access particular release builds and provide a way to bundle an aggregation of different versions of relevant constituent files in one bundle.

It's most beneficial to check out source instead of downloading the source of a release build when you have either trouble dealing with `.zip` archive files or you want to import the code straight into your IDE.

By now, you know all the ways of getting BlazeDS, both in compiled and source editions. Next, let’s delve into installing the binary versions and compiling the source.

## Installing the Compiled Distributions

Installing the compiled distributions involves expanding the archive files and deploying the web application. In the case of the turnkey distribution, things get even easier, as the step of deploying the web application is also skipped.

Installing the turnkey bundle implies merely unzipping it to a folder on the file system.

Figure 2-4 shows the top-level directory of the expanded folder structure of the turnkey distribution. There are three `.war` files listed in that directory, namely:

- `blazeds.war`
- `ds-console.war`
- `samples.war`
blazeds.war is the web application that contains BlazeDS. Technically, this is the only piece of software that’s necessary as far as BlazeDS goes. You get only this archive file when you choose to download a binary distribution that’s not labeled as “turnkey.”

The turnkey distribution contains a set of sample applications and an administrative console. The administrative console helps view and monitor BlazeDS and the sample application services, endpoints, channels, destinations, and the logging facility. Sample applications are bundled in samples.war, and the administrative console is bundled in ds-console.war.

I mentioned earlier that the turnkey distribution is the easiest and fastest way to get started with BlazeDS. That is because not only does this distribution come with an instance of the Apache Tomcat 6.x server, but the BlazeDS distribution, the samples application, and the console application are predeployed and configured in it. Figures 2-5, 2-6, and 2-7, respectively, peek into the webapps/blazeds, webapps/ds-console and webapps/samples directories of the included Apache Tomcat server. What you see in these figures are the deployed application folders.

Installing stable or nightly compiled distributions, which aren’t bundled like the turnkey option, involves an additional step of deploying blazeds.war to an application server or a Servlet container. I will quickly describe how the BlazeDS web application is deployed in Apache Tomcat 6.x, JBoss AS 5.0.0.GA, and Jetty 7 (Prerelease 5). In each of these situations, I assume that you have the respective application server installed and ready for web application deployment.

If you don’t use any of the above-mentioned Servlet containers or application servers, then you may still be able to deploy BlazeDS fairly effortlessly. After all, BlazeDS is just another web application, and it is only as complex to deploy it as it is to deploy any other web application.
Deploying BlazeDS in Apache Tomcat 6.x

Deploying an application in Apache Tomcat 6.x involves only two simple steps, namely:

- Unzipping the downloaded archive file to a folder on the file system.
- Dropping `blazeds.war`, from the download, in the `webapps` folder of Tomcat and starting Tomcat.

Figure 2-8 shows the `webapps` folder after `blazeds.war` is copied in it. Once this war file is copied over, start up Tomcat and the deployment is automatic. Figure 2-9 shows the folder snapshot after BlazeDS is deployed. You will notice that the war file has been exploded as a part of the deployment process.

This type of deployment, which is triggered on Tomcat startup, is referred to as “static” deployment. Sometimes, it may not be possible to restart Tomcat, and you may want BlazeDS to be deployed while it’s still running.
You can achieve "dynamic" deployment in a running Tomcat instance by any of the following three methods:

- Set `autoDeploy` to `true` in the "conf/server.xml" configuration file and do the same steps as in the case of static deployment
- Use the Tomcat manager utility
- Use Tomcat Client Deployer (TCD)

Setting `autoDeploy` to `true` is undoubtedly the easiest of the three options. I would just go with it. Tomcat manager, on the other hand, is a web-based utility that could be equally easy. You can read more about the Tomcat manager at [http://tomcat.apache.org/tomcat-6.0-doc/manager-howto.html](http://tomcat.apache.org/tomcat-6.0-doc/manager-howto.html).

TCD is the most complex and involved of the three choices. TCD comes as a separate download and is a feature-rich program that can do more than just runtime deployment. Among other things it can validate, compile, and compress web application builds to `.war` format. TCD needs Apache ANT to run successfully. More information on TCD can be obtained at [http://tomcat.apache.org/tomcat-6.0-doc/deployer-howto.html#Deploying using the Client Deployer Package](http://tomcat.apache.org/tomcat-6.0-doc/deployer-howto.html#Deploying using the Client Deployer Package).


Besides these, you could set up custom authentication, which will be brought up later in Chapter 4.

Next, we explore deploying BlazeDS in JBoss AS.

**Deploying BlazeDS in JBoss AS 5.0.0.GA**

Deploying the BlazeDS web application in JBoss AS is elementary. It’s a simple two-step process:

- Unzip the distribution
- Copy `blazeds.war` to the `server/default/deploy` folder in the JBoss AS 5.0.0.GA installation
JBoss AS scans the deploy directory frequently, so it will automatically deploy blazeds.war soon after it’s copied to the deploy folder. Instead of the default profile, you could also use the all or the minimal profile. JBoss AS does not need a restart for BlazeDS to successfully deploy.

Read the JBoss AS 5.0.0.GA Administration and Configuration Guide available online at https://www.jboss.org/community/docs/DOC-12927 for additional details that pertain to deployment and configuration.

Next, you will learn to install BlazeDS in Jetty.

**Deploying BlazeDS in Jetty 7**

Jetty 7, which is still in a prerelease version at the time of writing this chapter, is a small footprint, lightweight web server. It’s written entirely in Java.

Deploying BlazeDS in Jetty 7 often only involves two simple steps, which are:

- Unzip the binary distribution
- Copy blazeds.war to the webapps folder within the Jetty 7 installation

These steps are quite similar to those used to deploy web applications in Apache Tomcat or JBoss AS.

One could statically deploy BlazeDS, which means deploy and restart Jetty, or dynamically deploy it in a running Jetty 7 instance. Jetty 7 defines two classes, WebAppDeployer and ContextDeployer, to carry out static and dynamic deployments. Both these deployers are configured in Jetty’s configuration file, jetty.xml. Both these deployers are added to the Jetty server life cycle, thereby starting and stopping with the server. More information on WebAppDeployer can be obtained at http://docs.codehaus.org/display/JETTY/WebAppDeployer. More information on ContextDeployer can be obtained at http://docs.codehaus.org/display/JETTY/ContextDeployer.

With these four different short illustrations on deploying BlazeDS in Servlet containers and application servers, you must be convinced that the task is fairly straightforward and simple. Except for customizations around specific authentication schemes, some specific extensions to BlazeDS channels, and usage of custom adapters and factories, deploying BlazeDS remains a simple task. In other words, for all standard deployments things are straightforward.

**Compiling from Source**

In the last few sections, you learned to deploy the compiled BlazeDS distributions. Those sections assumed that you downloaded the compiled versions, which were available in a .zip file. You could also download the BlazeDS source instead of a binary compiled distribution. When you download the source, you need to compile it first so that it’s ready to be deployed. Once it is compiled, deploying it is the same as deploying any readily available compiled version.

In this section you, will learn how to compile BlazeDS from the source. BlazeDS source comes with Apache ANT (a Java-based build tool) build scripts to compile the source with minimal effort. Before you start calling the script to start the compilation, you need to get a hold of a few external libraries and set up a few environment variables.

First of all you need to download and install JDK 5 or higher. You probably already have that on your machine. If not, go get it from http://java.sun.com/javase/downloads/index.jsp.
Chapter 2: Introducing BlazeDS

The next necessary piece of software is Apache ANT. You can get a copy of the latest build at http://ant.apache.org/. Make sure to get a version equal to or greater than 1.7.0. Version 1.7.1 is the latest version as of this writing. Unzipping ANT to a folder is all that is required to install ANT.

In addition to Apache ANT, also download ANT-Contrib Tasks from http://ant-contrib.sourceforge.net/. ANT-Contrib Tasks are a collection of tasks for Apache ANT. ANT-Contrib Tasks extend ANT beyond simple dependencies management and make it function somewhat like a programming language. At the download site on SourceForge, you will see cpptasks downloads side by side with ant-contrib downloads. Just get the latest version of ant-contrib, which at this time is 1.0b2. ANT-Contrib can be installed in two ways. You can either copy ant-contrib-1.0b2.jar to the lib directory of the ANT folder or you can copy it anywhere else and specify the path to it in the build scripts. To keep things simple just copy the jar to the ANT lib folder.

Last of all, download JUnit from www.junit.org and copy over the jar file to the ANT/lib directory.

That’s all as far as getting external pieces of software goes. Now you need to set up a few environment variables. Set up ANT_HOME to point to the root of your ANT installation and JAVA_HOME to point to the root of your JDK installation. Also, add the bin folders of both ANT and Java to the PATH environment variable.

That’s all as far as the setup goes. Now you are ready to invoke the build scripts. Just call ant main, and the compilation is done. That was simple, wasn’t it? When running ant main the next time, precede it with an ant clean run to make sure that all cleanup is done and you are ready to build again. Last of all, you can also run all unit tests on the new build to make sure that everything built correctly. You invoke this task by calling ant checkintests.

The output is the compiled war file, which can be deployed in an application server of your choice.

Surveying the Configuration Options

Before you start using BlazeDS gainfully, you need to learn how to configure BlazeDS to work with your application.

Simply put, at the heart of BlazeDS is a Servlet that bootstraps the infrastructure that intercepts all calls between a Flex client and the BlazeDS instance. This Servlet, called MessageBrokerServlet, uses artifacts like channels, endpoints, and adapters to enable proxy, remoting, and messaging services. A default configuration file, called services-config.xml, which lies in the WEB-INF/flex folder of the BlazeDS installation, defines these artifacts and their relationships in the context of the MessageBrokerServlet. We will peek into this configuration file in this section and continue to revisit it in the later chapters in the context of BlazeDS’s various services. However, nothing in the configuration will make much sense unless you first understand the fundamentals of channels, endpoints, and adapters.

Channels and endpoints connect a Flex client to a BlazeDS server. They are the primary components that enable communication between these two entities. Endpoints reside at the BlazeDS end. Flex clients use channels to connect to these endpoints.

The BlazeDS endpoints are Servlet-based endpoints. Each endpoint defines a type and format of communication. For example, endpoints exist for simple AMF data exchange, polling AMF, and AMF data streaming.
Analogously, the Flex client defines a set of channels that vary depending on the type and format of communication. For example, the HTTPChannel facilitates communication over non-binary AMF format, AMFX (AMF in XML), and the AMFChannel enables standard binary AMF-based communication.

Matching endpoints and channels are paired, and that’s when a Flex client and BlazeDS server talk to each other. The binding of channels and endpoints to their implementation classes and their pairing is done in the services-config.xml configuration file.

In addition to the endpoints, BlazeDS includes adapters that provide the critical compatibility between the core BlazeDS Servlet that talks to Flex and a server-side resource such as a JMS resource, a persistent data store, or an Object-Relational mapping layer. Adapters are also configured in services-config.xml.

You must be curious by now to see what services-config.xml looks like, and so that is what we look into next.

**First Look at services-config.xml**

The configuration file services-config.xml, in BlazeDS, is logically split into four configuration files. The top level and the first of these four is services-config.xml itself. The other three are as follows:

- remoting-config.xml
- proxy-config.xml
- messaging-config.xml

These three files are included in services-config.xml by reference as follows:

```xml
<services>
  <service-include file-path="remoting-config.xml" />
  <service-include file-path="proxy-config.xml" />
  <service-include file-path="messaging-config.xml" />
</services>
```

The names of the configuration files suggest what they configure. remoting-config.xml is where all remoting related destination, endpoint, and adapter configuration resides. proxy-config.xml holds the proxy service settings, and messaging-config.xml defines the messaging resources, adapters, and their properties.

You will learn more about the contents of each of these files soon. However, first let’s explore the top-level configurations that reside in the main file, services-config.xml.

Standard channels and corresponding endpoints are defined as shown in Listing 2-1.

**Listing 2-1: Channel and Corresponding Endpoint Definition in services-config.xml**

```xml
<channels>
  <channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/amf"
```
Chapter 2: Introducing BlazeDS

Listing 2-1: Channel and Corresponding Endpoint Definition in services-config.xml
(continued)

```xml
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:{server.port}/{context.root}/messagebroker/amfsecure"
        class="flex.messaging.endpoints.SecureAMFEndpoint"/>
    <properties>
        <add-no-cache-headers>false</add-no-cache-headers>
    </properties>
</channel-definition>

<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>
        <polling-interval-seconds>4</polling-interval-seconds>
    </properties>
</channel-definition>

<channel-definition id="my-http" class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/http"
        class="flex.messaging.endpoints.HTTPEndpoint"/>
</channel-definition>

<channel-definition id="my-secure-http"
    class="mx.messaging.channels.SecureHTTPChannel">
    <endpoint url="https://{server.name}:{server.port}/{context.root}/messagebroker/httpssecure"
        class="flex.messaging.endpoints.SecureHTTPEndpoint"/>
    <properties>
        <add-no-cache-headers>false</add-no-cache-headers>
    </properties>
</channel-definition>
</channels>
```

The code in listing 2-1 creates five types of channels:

- AMF channel
- Secure AMF channel
Polling AMF channel
❑ HTTP channel
❑ Secure HTTP channel

If you look closer, the three AMF channels are quite similar. AMF and Secure AMF use different implementation classes and use http and https endpoints, respectively, for communication, but share a common binary format for information exchange. Polling AMF adds polling capabilities on the standard AMF channel. The polling-enabled property is set to true and polling-interval-seconds sets a polling frequency of 4 seconds for repetitive invocations.

The HTTP channel and secure HTTP channel set up channels for non-binary data transfer using XML over AMF.

Apart from the channels, you define other cross-cutting concerns such as security, logging, and configuration file deployment characteristics in the top-level configuration file. The logging settings could be as follows:

```xml
<logging>
  <target class="flex.messaging.log.ConsoleTarget" level="Debug">
    <properties>
      <prefix>[BlazeDS] </prefix>
      <includeDate>false</includeDate>
      <includeTime>false</includeTime>
      <includeLevel>false</includeLevel>
      <includeCategory>false</includeCategory>
    </properties>
    <filters>
      <pattern>Endpoint.*</pattern>
      <pattern>Service.*</pattern>
      <pattern>Configuration</pattern>
    </filters>
  </target>
</logging>
```

You will learn a lot more about logging in Chapter 4. With a cursory glance, you can easily notice that console is set as the target of the log messages, and the logging level is set to a debug level.

Next, we rapidly browse through important sections of the other three log files.

**Remoting, Proxy and Messaging Configuration**

One of the key configurations for remoting relates to the destination. A destination is a logical handle to a remote service, on which a remote procedure can be invoked. Adapters and channels are also associated with a destination. You will see a simple Java remote service in action in the next section. Both code and configuration for the simple example are explained there. In that example a Java adapter is in use. A Java adapter is included as follows:

```xml
<adapters>
  <adapter-definition id="java-object"
    class="flex.messaging.services.remoting.adapters.JavaAdapter" default="true"/>
</adapters>
```
Chapter 2: Introducing BlazeDS

The remoting service is central to BlazeDS and is explained at length at many points in this book, so there isn’t more discussion on it here.

The messaging service, like the remoting service, uses a destination and an associated set of adapters and channels. The destination configurations for messaging and remoting are similar. The adapter for messaging could include a JMS adapter, which can accept a number of settings, which are specified in the messaging-config.xml file. You have not been given any background on messaging yet, so talking about its configuration is premature. Anything that is related to messaging will only make sense when discussed in context and that will not happen until Chapter 6. If you can’t control your excitement, jump over to that chapter and find out more.

The last of the configuration trio is the proxy configuration. Proxy service allows HTTP and SOAP requests to be tunneled through the server to work around the constraints imposed by the lack of security specifications at a remote destination. BlazeDS defines adapters to work with both HTTP and SOAP protocols. Including the adapters and setting them up while bootstrapping means specifying them as follows:

```xml
<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>
```

While a lot more can be covered on configuration, I think this may be enough to get a sense of what is involved. Finally time to dig into an elementary but working example that uses BlazeDS to converse between Flex and Java.

Yet Another Hello to the World

The simple application in this section, which shows Flex and Java integration using BlazeDS, says “Hello.” The example is first sketched out so that it will be easy to comprehend the functional flow and the rudimentary, yet working, use case. This sketch is shown in Figure 2-10.

Two types of remote calls are supported in this example application. These are:

- A call to the sayHelloInString method on a Java object, which takes in a string parameter and returns a string value
Chapter 2: Introducing BlazeDS

- A call to the sayHelloInObject method on the same Java object, which takes in an object parameter and returns an object

In the case where objects are passed back and forth between Flex and Java, parameters are sent from Flex as AS3 objects, and return values are sent from the remote Java service as Java objects. BlazeDS does the important translation and marshaling between these two languages.

To create this example application, start up your Flex Builder IDE and create a new Flex project. Choose to create a project with a J2EE server and specify the parameters as shown in Figure 2-11.

![Figure 2-11](image)

By choosing the option to create a joint Java/Flex project, you ease the process of development and deployment, as the entire bundle of source can be compiled and deployed together. In addition, you can configure a target server environment to deploy the build. I have chosen a JBoss AS server instance as the target.

As you move through the wizard for project creation, you will be prompted to provide a path to a blazeds.war file and to the context root of your target. The blazeds.war file can be anywhere in the file system. I have intentionally pointed it to an instance in one of my Tomcat folders. Figure 2-12 shows this wizard screen.

The final screen before you complete the project creation allows you to change the name of your default Flex application file and lets you provide the path to the application source. I choose to go with the defaults. Look at Figure 2-13 for specifics.

The project shell is ready, and now you can begin to code.
No Flex Builder? Don’t Worry!

The “HelloBlazeDS” example is the simplest of the examples you will see in this book. I decided to go with Flex Builder for this one because I wanted to keep the focus on the essentials of remoting and on basic BlazeDS usage. In all other examples I will include illustrations, tips, or guidelines, which will help you setup, build, and deploy them without an IDE as well. In some cases, recommendations for building the examples using IntelliJ IDEA will also be included.

The task at hand requires that you do the following:

- Create a Flex client that would invoke the remote service and consume the response returned by the service. The Flex code could reside in an MXML file called HelloBlazeDS.mxml.
- Create a HelloService class on the Java server.
- Make a remoting-config.xml entry to expose HelloService so that it’s accessible from the Flex client.
Chapter 2: Introducing BlazeDS

- Create Greeting classes both in Java and AS3 to exchange objects.
- Build and run the application on the server (in my case JBoss AS. You could also choose to use Tomcat or any other server.)

The HelloBlazeDS.mxml Flex client code is printed in Listing 2-2.

**Listing 2-2: HelloBlazeDS.mxml**

```xml
<?xml version="1.0" encoding="utf-8"?>
    backgroundColor="#FFFFFF">
    <mx:Script>
        <![CDATA[
            import problazeds.ch02.Greeting;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            private var flexGreeting:Greeting = new Greeting();

            public function getGreetingFromFlex():Greeting {
                flexGreeting.sender = "Flex";
                flexGreeting.message = "Hello";
                flexGreeting.lastMessage = ";

                return flexGreeting;
            }

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the TextArea control.
                outputText.text = remoteService.sayHelloInText.lastResult;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>}
    </mx:Script>

    <!-- Connect to a service destination.-->
    <mx:RemoteObject id="remoteService"
        destination="helloService"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <mx:Panel>

        <!-- Provide input data for calling the service. -->
        <mx:TextInput id="inputText"/>

    </mx:Panel>

</mx:Application>
```

Continued
The Flex client code invokes a service called the HelloService. HelloService resides in a Java server that also hosts the BlazeDS files. The HelloService can reside in a single Java file as follows:

```java
package problazeds.ch02;

public class HelloService {
    public Greeting sayHelloInObject(Greeting incomingGreeting) {
        Greeting greeting = new Greeting();
        greeting.setSender("Java");
        greeting.setLastMessage(incomingGreeting.getLastMessage());
        greeting.setMessage("Hello");
        return greeting;
    }

    public String sayHelloInText(String textGreeting) {
        return "Hello from Java" + "Received your message:" + textGreeting;
    }

    private Greeting greeting;
}
```

Flex client calls this Java service without explicitly specifying the path to the Java class name or the fully qualified class name. This is possible because all of this information is available through a configuration entry in `remoting-services.xml`, which is:

```xml
<destination id="helloService">
    <properties>
        <source>problazeds.ch02.HelloService</source>
    </properties>
</destination>
```

When exchanging objects between Flex and Java, you keep copies of the object definition on both sides. The AS3 class is:

```as3
package problazeds.ch02 {
    [Bindable]
    [RemoteClass(alias="problazeds.ch02.Greeting")]
```
public class Greeting
{
    public var sender:String="";
    public var message:String="";
    public var lastMessage:String="";
}

The RemoteClass metadata above the class keyword specifies an alias as problazeds.ch02.Greeting. The Bindable metadata annotation specifies that all changes to any of the class attributes are propagated to all points that consume this class using data binding. This is the Java object across the wire. The Java Greeting class is:

```java
package problazeds.ch02;

/**
 * @author Shashank Tiwari
 */

public class Greeting {

    public String getSender() {
        return sender;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getLastMessage() {
        return lastMessage;
    }

    public void setLastMessage(String lastMessage) {
        this.lastMessage = lastMessage;
    }

    private String sender;
    private String message;
    private String lastMessage;
}
```
That covers all the elements that make up the example. The example is simple and further explanation may be redundant, so I will stop right here.

With the completion of this example, I am ready to wrap up this chapter. Let’s recap what you learned in the last few pages.

**Summary**

This chapter started by defining what BlazeDS is and ended with an example that showed it in action. Of course, it covered many fundamental aspects of BlazeDS, including the following:

- Setup and installation
- Compilation from source
- Essential Configuration

In the section on deployment, the steps involved were discussed in the context of four different Servlet containers and application servers. This set included Apache Tomcat, JBoss AS, and Jetty.

While discussing configuration, a few of the core BlazeDS artifacts such as channels, endpoints, and adapters were also explained. More on this will be covered throughout this book.

While there is a lot more to be learned about BlazeDS and its applicability, I hope this chapter has gotten you started. The initial definition and the viewpoints from different levels of proximity are a good way to envisage the role that BlazeDS plays in integrating Flex and Java. Knowing this early on should help you leverage BlazeDS in effective ways.

Last, but not the least, this chapter also emphasizes the need to set up your infrastructure and development environment and walks you through an example. It’s important that you get all of this straightened out right away before you go to the next chapter. You will avoid many glitches if your development environment is reliable and predictable.

With that little recommendation and the warm-up that the chapter provides, you are ready for a lot of exploration.
A Flex client often needs to access external sources of data. It can access this data using one of the following two mechanisms:

- It can request for data and get it as a response.
- It can register interest in a certain type of data and receive it if and when such data is available.

The first of these two options is “pull” based, whereas the second option is “push” based.

Off the shelf, the framework includes three methods of pull-based communication and data interchange with external data sources:

- HTTP request-response
- Web services
- Remote procedure calls involving objects

HTTP request-response is the most elementary of the three methods for external data access. It needs little preparation or prior knowledge to get started. I am assuming here that you know and understand the basics of the Web. Web services are not very complicated either, but they add a few additional protocols and semantics on top of the plain vanilla HTTP request-response. The last of the three, remote procedure calls using objects, is possibly the most complicated of the three. However, its complexity is often restricted to some initial configuration and the intricacies around the communication protocol, much of which is abstracted out for the application developer.
Chapter 3: Using BlazeDS as a Server-Side Proxy

HTTP-based communication and web services can be leveraged by an application running in a Flash Player without BlazeDS or any such remoting layer, provided that the security constraints are satisfied. In other words, if the data source domain is the same as the one that serves the Flex application or if the external data source domain defines a security policy that allows for access from the Flex client, then often there is no need for BlazeDS. Such security policy is defined in a file named crossdomain.xml. The sidebar “Crossdomain.xml” explains what this XML file contains.

Crossdomain.xml

To avoid cross-site scripting and hijacking of trusted network data from untrusted networks, Macromedia defined a security policy file for the Flash Player back in 2003. This security policy file is crossdomain.xml. The security policy when deployed at the root of a domain could allow access to its data from Flash applications served from other domains. Such access policies could be defined in the file as follows:

```xml
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="master-only"/>
  <allow-access-from domain="*"/>
  <allow-http-request-headers-from domain="*" headers="SOAPAction"/>
</cross-domain-policy>
```

In the preceding example, the use of wildcard in the allow-access-from-domain element gives data access to all external domains. Alternatively, access could be restricted to specific domains by specifying them as values of this element.

Policy restrictions apply to browser-facilitated HTTP and TCP/IP socket connections. Information on the security policy specification is available online at www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

However, security restrictions may not be the only reason for using the BlazeDS proxy service. The proxy service supports HTTP methods beyond GET and POST, allows for state management in web services, and provides an opportunity to intercept all calls and their results to pass them through a desired business logic chain or apply cross-cutting application-wide concerns. These can be reasons as well for its usage.

In this chapter, you will learn to use the proxy service in the context of HTTP service and web service. HTTP service is simpler of the two, so that’s where I will start.

HTTP Service Proxy

Rich Flash platform applications built using Flex are deployed as swf format files on a web server. When a client accesses such an application, the swf is downloaded to the client’s local Flash Player instance. These swf format files are compiled binary files that can use local components to invoke remote URL(s)
Chapter 3: Using BlazeDS as a Server-Side Proxy

using the HTTP method. If the domains that these remote URL(s) point to define appropriate security policy, then the Flex application can make these calls without any trouble. If the security policy is not defined, then these HTTP calls can be routed through a server-side proxy.

BlazeDS defines such a server-side proxy for easy and flexible routing. The Apache HttpClient is actually baked right into BlazeDS. So, when you route an HTTP call through the proxy, the call is actually invoked and completed by an HttpClient instance at the server, and the response is relayed back to the Flex application.

Apache HttpClient

Apache HttpClient is a full-featured standards-complaint Java-based HTTP client that allows invocation of URI(s) using the HTTP methods. It’s a user agent and so can be used to make calls in a manner similar to what a web browser does. Its latest release version is 3.x. It’s available under the Apache open source license. More information about the HttpClient can be obtained from the Apache HttpComponents site at http://hc.apache.org/httpcomponents-client/index.html.

When using a server-side proxy, BlazeDS needs appropriate configuration, which as you learned in the previous chapter, takes place in services-config.xml. For proper organization and clear separation, the configuration is distributed in four separate files, remoting-config.xml, proxy-services.xml, messaging-config.xml, and services-config.xml. The first three of these four configuration files are included in the fourth one, services-config.xml, by reference. Most proxy-related configurations, besides those that are common across the board such as channel and protocol definitions, reside in proxy-services.xml.

You need to define an HTTP proxy service configuration in proxy-services.xml. Before you start doing that, you may benefit from viewing Figure 3-1, which shows the various layers in our communication stack.

Sitting right on top of this stack is the Flex client itself. Next is the endpoint on which the client communicates with the BlazeDS server. The communication between a Flex client and a remoting server like BlazeDS is established with the help of a channel-endpoint pair, where the channel represents the client side of the communication pipe and the endpoint represents the server side of the communication gateway.

At the heart of BlazeDS is a message broker, which coordinates most of its activities. You were introduced to this in the last chapter and will continue to learn more about it throughout this book. The endpoints communicate with the message broker.

Next come three layers: service, destination, and adapter. This trio is the defining set of layers as far as the proxy service goes. At the end of the stack is the HTTP resource.

The HTTPProxyService is the proxy service class that accepts a message of the HTTPMessage type. This facilitates a loosely coupled association between the Flex client and the message broker and allows it to use any of the channels to communicate and still use the same proxy class. The default channel is the AMFChannel, which transmits messages as binary Action Message Format (AMF) over HTTP. You already know of AMF from Chapter 1, where it was introduced to you. If binary transmission is not preferred
and text-based communication is necessary, then an alternative channel, called the **HTTPChannel**, can be used. This might be possible when a common server-side program serves data, in a unified format, to multiple types of clients: Flex, legacy page based clients, and Ajax. **HTTPChannel** uses AMF in XML over HTTP.

An **HTTPMessage** object abstracts an HTTP or an HTTPS resource destination. Its properties hold the values for the URL, HTTP method, headers, and content type. The proxy service supports all HTTP methods, namely, **GET**, **POST**, **HEAD**, **TRACE**, **PUT**, **DELETE**, and **OPTIONS**. When calling an HTTP resource directly, without a proxy, you are restricted to the **GET** and **POST** methods only. Read the note “HTTP Methods” to review the purpose of these methods.

### HTTP Methods

- **GET** — Requests a representation of a resource in such a manner that there are no side effects; that is, no resources are modified and the response is consistent and repeatable (also technically referred to as an idempotent call).
Chapter 3: Using BlazeDS as a Server-Side Proxy

- **POST** — Submits data in the request headers and the body to the resource. Processing of data can lead to side effects like modification of the resource itself or creation of newer resources.

- **HEAD** — Same as GET, except that only the headers and no body is returned. Used to access metadata.

- **TRACE** — Echoes the request back to the invoker. Used to ascertain how the server is affecting the request.

- **PUT** — Uploads a resource representation.

- **DELETE** — Removes a resource.

- **OPTIONS** — Returns the HTTP methods a server supports.

The **HTTPProxyDestination** abstracts the remote proxy destination, and the **HTTPProxyAdapter** is the piece of code that makes the HTTP call using the **HttpClient**. The HTTP proxy service and destination classes are also relevant for web services, which use these same classes. Web services use a **SOAPProxyAdapter**, instead of the **HTTPProxyAdapter** class, to make the web service operation invocations via the server.

At this stage, working out a simple example may reinforce and illustrate the fundamentals of using an HTTP proxy service, so let’s engage in that exercise.

**An HTTP Proxy Service Example**

You can access a list of my public presentations, in an XML format, at [http://shanky.org/publish-and-present/present/list.xml](http://shanky.org/publish-and-present/present/list.xml). There is a **crossdomain.xml** file defined at the root of this domain, but let’s use the proxy service to get this list. Once you have the list, simply bind this output to a Flex data grid.

Since the focus is on how one can access this list, using the proxy service, the **HTTPService** client-side component, instead of the **DataGrid**, makes a good starting point for the example. Let’s create a simple **HTTPService** component using MXML as follows:

```xml
<mx:HTTPService
    id="presentationList"
    useProxy="true"
    destination="presentationAll"/>
```

One of the most important properties defined on the preceding **HTTPService** component is the **useProxy** property. By default, its value is **false** and that implies that HTTP calls are made directly to the resource. Setting its value to **true** acts as the vital switch that directs the calls through an associated proxy service.

The other two properties defined are **id** and **destination**. As always, **id** provides the handle for future reference to the instance. The **destination** property value corresponds to a server-side destination configuration, within the **services-config.xml** file. As mentioned earlier, the main configuration file,
that is, services-config.xml, is distributed in multiple configuration files for organization and clarity. Therefore, the proxy service destination definition lies in a file called proxy-config.xml. The proxy configuration file is included in the main configuration file by reference.

In the proxy configuration file, a configuration entry appears as follows:

```xml
<service id="proxy-service"
class="flex.messaging.services.HTTPProxyService">
<destination id="presentationAll">
<properties>
<!-- The HTTP endpoint available to the http proxy service -->
?url=http://shanky.org/publish-and-present/present/list.xml</url>
</properties>
</destination>
</service>
```

The preceding configuration creates a service instance based on the flex.messaging.services.HTTPProxyService class. This service instance has a destination property that includes a URL that points to the resource we set out to access.

Now it’s possible that you may desire to also access the list of all my publications (and not presentations as you did earlier), and that is accessible at http://shanky.org/publish-and-present/publish/list.xml. This can be done in one of two ways. Either you can add another static url property within a destination and pass the fully qualified URL as its property value or replace the existing static url element with a dynamic-url property to http://shanky.org/publish-and-present/*, like so:

```xml
<dynamic-url>http://shanky.org/publish-and-present/*</dynamic-url>
```

Now on the client-side, the HTTPService object that fetches the list of presentations is like so:

```xml
<mx:HTTPService url="http://shanky.org/publish-and-present/
present/list.xml"
useProxy="true"/>
```

Similarly, the HTTPService object that retrieves the list of publications is like so:

```xml
<mx:HTTPService url="http://shanky.org/publish-and-present/
publish/list.xml"
useProxy="true"/>
```

That’s all it takes to get a proxy in action. Next, we explore a few more nuances of the proxy service, especially those that relate to the configuration of the adapter.

**Advanced Configuration Options of an HTTP Proxy Adapter**

A fair amount of advanced configuration for a proxy service pertains to its adapter. You may recall that the HTTP calls to real destinations are made using the Apache HttpClient user agent. The HttpClient connections are established and influenced by the HTTPProxyAdapter. A number of properties can be set to define the characteristics of the Apache HTTP components connection manager (HTTPConnectionManager) that the HttpClient uses. These properties are specified in proxy-config.xml.
within the connection-manager property of an HTTP proxy service configuration. The connection-manager property includes a number of attributes, which are:

- max-total-connections
- default-max-connections-per-host
- connection-timeout
- socket-timeout
- stale-checking-enabled
- send-buffer-size
- receive-buffer-size
- tcp-no-delay
- linger
- max-per-host

The max-total-connections attribute governs the number of concurrent connections. BlazeDS uses a multi-threaded connection manager for the HttpClient instances, except when the value of max-total-connections is set to 0.

The default-max-connections-per-host attribute sets the default maximum number of connections allowed per host. This attribute is relevant in the context of clustering, which will be brought up later in Chapter 10. Definitions of connection attributes per host can be defined in the max-per-host attribute, which takes a collection of host configuration settings (defined within the BlazeDS HostConfigurationSettings class). The host configuration settings include all the parameters that relate to an HTTP connection. They are:

- host
- port
- protocol
- protocol-factory
- max-connections
- proxy
- local-address
- virtual-host

The two timeout attributes for the connection manager, connection-timeout and socket-timeout, specify the number of milliseconds to wait before timing out. The connection-timeout is the amount of time to wait for a connection to be established before it’s timed out. The default value is 0, which means that it doesn’t time out at all. socket-timeout is the amount of time to wait for data to come before the connection is timed out. Again, the default is 0, which again means that it doesn’t time out at all.

If stale-checking-enabled is set to true, which is the default case, connections are checked to ensure that they are working. Enabling this property has a slight performance overhead, but it minimizes the
Chapter 3: Using BlazeDS as a Server-Side Proxy

I/O errors emerging out of failed requests over stale connections, which falsely show as active locally but are closed at the remote server.

Data is sent and received over the socket. The two buffer-size-related parameters, send-buffer-size and receive-buffer-size, act as suggested buffer sizes for the data interchange over the socket. The underlying buffer size may not be the same as that set through these parameters.

In addition to suggesting buffer sizes, you can choose to use John Nagle’s algorithm to make the TCP/IP communication over sockets more efficient. Nagle’s algorithm combines and coalesces a few outgoing packets and reduces the number of packet transmissions. You can read more about Nagle’s algorithm at http://en.wikipedia.org/wiki/Nagle%27s_algorithm. Setting the tcp-no-delay to false (which by default is true) activates Nagle’s algorithm on your TCP/IP packets.

Last, but not least, you can also control how the sockets handle queued data when they close. When linger-on-close is set to true, a Java socket allows queued data to be sent to its peer when the close method is called. You can set the linger property on the connection manager to either 0 or -1. 0 deactivates the property, and -1 delegates the linger-on-close behavior to the JRE default.

Besides the connection manager settings, you can set the cookie-limit, allow-lax-ssl, and content-chunked properties on the adapter. The cookie-limit defaults to 200. Settings allow-lax-ssl allows self-signed certificates when using SSL. Content, by default, is not chunked. The Flash Player does not chunk content.

Sometimes, a user-agent cannot access the Internet directly and needs to route its requests through an external proxy. If such a situation arises for the HttpClient that the HTTPProxyAdapter uses, then you could configure the adapter with the external proxy settings, which include:

- server
- port
- nt-domain
- username
- password

Looking at all the configuration options listed in the last couple of pages, it’s fairly obvious that the adapter can be customized for specific cases.

Towards the beginning of this chapter, I alluded to the control and management benefits accrued from passing HTTP requests through a server proxy. Let’s clearly list those here:

- Easy applicability of cross-cutting concerns
- Central control of all outgoing requests
- Access control of the HTTP traffic
- Logging of all traffic information on the server
- Handling of HTTP errors gracefully
- Application of business logic on intercepted requests and responses
In the next subsection, let’s elaborate on a few of these points to see how BlazeDS enhances the plain and simple `HTTPService` client-side component.

**Intercepting for Control and Management**

In this section, I will illustrate three different scenarios that show the benefits of intercepting proxy calls on the server. By no means is this an exhaustive list, but it is fairly indicative of the possible benefits of interception and central control. The three use cases include:

- Access control to proxy destinations
- Logging of all proxy traffic
- Handling of HTTP errors

**Access Control to Proxy Destinations**

For illustration, an HTTP proxy destination is shielded behind an authentication scheme and an authorization scheme. Flex clients using this proxy service destination to access remote data need to pass in appropriate credentials to interact with it successfully. For simplicity, Tomcat is used as the server-side container for BlazeDS.

BlazeDS provides a way to authenticate, that is, validate, a user’s identity. It also provides a way to restrict operations based on a user’s credentials or entitlements, which is termed “authorization.”

An interface, `flex.messaging.security.LoginCommand`, defines the contract for carrying out the processes of authentication and authorization through its `doAuthentication` and `doAuthorization` methods. Readily available and prepackaged implementations of this interface exist for many of the popular application servers, namely:

- Tomcat
- JBoss (which uses Tomcat)
- BEA (now Oracle) Weblogic
- IBM Websphere
- Oracle Application Server
- Adobe JRun

Each of these implementations leverages the underlying server-specific Java EE security for authentication and authorization. In addition, you can define your own custom authentication and authorization scheme and plug it into BlazeDS by implementing the `LoginCommand` interface.

If you want to delegate the security aspects to Tomcat, then you just wire up the available implementation in `services-config.xml` as follows:

```
<login-command class="flex.messaging.security.TomcatLoginCommand"
    server="Tomcat"/>
```

This piece of configuration lies within a `security` element in `services-config.xml`. It says that authentication will use the `TomcatLoginCommand` implementation. In Tomcat, users and their roles are
defined in a file called `tomcat-users.xml`, which lies in the `conf` folder of the Tomcat installation. The entries in this file (for my instance of Tomcat) are shown in Listing 3-1.

**Listing 3-1: Contents of `conf/tomcat-users.xml`**

```xml
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <role rolename="regularusers"/>
  <user username="tomcat" password="secret" roles="manager"/>
  <user username="auser" password="apassword" roles="regularusers"/>
</tomcat-users>
```

If you choose to use the Tomcat security implementation, then you need to use the same roles in the `services-config.xml` that are defined in the `tomcat-users.xml` file.

Roles can be defined within the configuration files using the `security-constraint` definitions. Two types of authentication methods can be defined within the security constraints. These are:

- **Basic** — Uses basic HTTP authentication. The browser throws up the login screen for username and password input when connecting, and on authentication failure an HTTP 401 error is returned.
- **Custom** — Uses a custom authentication method. The credentials, username and password, need to be passed in explicitly and programmatically. The `ChannelSets.login` method can be used to pass in the credentials.

For the sake of avoiding complexity, let’s just use the Basic authentication method. After this decision, our security definition is as follows:

```xml
<services-config>
  <security>
    <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
      <per-client-authentication>false</per-client-authentication>
    </login-command>
    <security-constraint id="trusted">
      <auth-method>Basic</auth-method>
      <roles>
        <role>regularusers</role>
        <role>manager</role>
      </roles>
    </security-constraint>
  </security>
...
</services-config>
```

Setting the `per-client-authentication` to false sets the authentication on a per session basis. This means that two copies of a Flex client running in two different tabs of a browser window are authenticated using one set of credentials, as both these clients share a session.
Now you can add the defined security constraint to a definition as follows:

```xml
<destination id="presentationAll">
  ...
  <security>
    <security-constraint ref="trusted"/>
  </security>
</destination>
```

This will ensure that only authenticated principals will be able to access data using this HTTP proxy destination.

That’s all as far as access control of destinations go. The topic of access control will be revisited again in the later chapters. By the end of the book, you will learn everything about access control in BlazeDS, including how to write your own custom authentication scheme.

Like security constraints, you can easily setup server-side logging levels to record all activities in BlazeDS. This could be helpful both for application monitoring and external data sources access analytics.

**Logging All Proxy Traffic**

The BlazeDS logging framework provides for logging of messages on the server side. What is to be logged is determined by the set level, which can take the following possible values:

- **All** — Log everything.
- **Debug** — Log all Error, Info, Warn, and Debug messages. Some Flex internal messages also qualify as debug messages.
- **Error** — Log all Error messages. Errors arise when an important part of the application stops working, and such messages are logged at this level.
- **Info** — Log all Info and Error messages. Info messages usually communicate a desired piece of information.
- **None** — Log nothing.
- **Warn** — Log all Warn and Error messages. Warnings are problems that don’t bring the system down and are not the same as errors, but nonetheless need to be attended to.

Messages by default are logged out to the console using `ConsoleTarget`.

The logging framework can be customized to include date, time, a prefix, and the category. Log messages can be filtered by categories.

Setting the logging level to “All” will catch all the communication on all endpoints and interaction with all services. With the possible overhead of too much information, this level will log all HTTP proxy service–related activity on the server. You will learn more about server side logging and its extensions in many of the later chapters in this book.

The last of the three use cases is about enhanced functionality that’s missing from the client-only access option.
Handling of HTTP Errors

When using `HTTPService` to invoke a remote HTTP resource directly, that is, without a proxy service, one only receives the HTTP response bodies for success cases, where the HTTP response code is between 200 and 299. For all other status codes the Flash Player cannot access any data in the body of the response. So, for example, the common status code of 404, which represents the situation where the resource is not found, just ends up in a failure the same way that status code 400, which is caused by bad syntax, does.

When using the proxy service, you have the flexibility to access the body of these messages and process them appropriately. Appropriate handling could mean retries, alternative calls, passing up of the status messages, or any other action that’s suitable for your particular case.

A fault message is dispatched as a `FaultEvent` to the `HTTPService` component. A fault handler function, wired to the `HTTPService` component, can handle a fault event and process it as desired.

So far, you have seen most of the features of the HTTP proxy service, learned to configure it, and perceived the potential value it can add. Now we move to the second topic of this chapter: the web service proxy.

Web Service Proxy

The Flex `WebService` component facilitates the interaction of a Flex client with a SOAP-based web service. Using this component, a Flex client can invoke remote operations on a web service and access data using such a service. Like `HTTPService`, the `WebService` component can access the services directly as long as these services are served from trusted domains. The domain from which the Flex client was served and those that define a security policy in a `crossdomain.xml` file are trusted domains as far as the Flex client is concerned. Read the note on `crossdomain.xml` at the beginning of this chapter if you need to review its features.

A SOAP web service publishes its available operations, the parameters it takes, the message type it returns, and the encoding it embodies in an XML-based format file, commonly called the WSDL. WSDL, which stands for Web Services Description Language, has become an acronym for the file itself as well as representing the format and the protocol. WSDL can be stored locally in the file system or be available as a web resource over a URI.

A web service component first expects to read and parse a WSDL for a service. If it successfully parses this, it calls the exposed operations. It can bind parameters to these operations and can receive responses, which it can subsequently consume.

Web service calls can be directed via a web service proxy on the server. This helps access SOAP-complaint services from domains that may not be explicitly trusted or where server-side control may be an advantage.

Before we dig deeper into web service proxy services, let’s first view the involved communication stack. Figure 3-2 depicts the communication stack. It’s analogous to Figure 3-1, which depicts the stack for an HTTP service proxy.
You will notice that up to the destination, the communication stack for the web services proxy is the same as for the HTTPServiceProxy. Things change at the adapter level. The web services proxy leverages the SOAPProxyAdapter. The SOAPProxyAdapter extends the HTTPProxyAdapter.

Figure 3-2

Flex supports WSDL 1.1 and supports both the RPC/encoded and document/literal WSDL bindings. Figure 3-3 depicts the different possible WSDL style/use combinations, which are RPC/encoded, RPC/literal, document/encoded, and document/literal. WSDL defines a web service, whereas WSDL binding describes how the web service is bound to the SOAP message protocol. RPCs and document style often make developers believe that these styles correspond to the programming styles that each of their names suggest. Contrarily though, these styles have nothing to do with the programming styles. The styles only address the WSDL-to-SOAP message mapping.

To illustrate further, a simple operation or method in Java can be written as follows:

    public void aMethod(String x, int y);
When this method is exposed as a web service operation, it publishes an RPC/encoded WSDL for the service as follows:

```xml
<message name="aMethodRequest">
  <part name="x" type="xsd:string"/>
  <part name="y" type="xsd:int"/>
</message>
<message name="empty"/>
<portType name="aPortType">
  <operation name="aMethod">
    <input message="aMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>
...
```

Now if we were to call this method with parameters “hello” and 4, then the corresponding SOAP message would be as follows:

```xml
<soap:envelope>
  <soap:body>
    <aMethod>
      <x xsi:type="xsd:string">hello</x>
      <y xsi:type="xsd:int">4</y>
    </aMethod>
  </soap:body>
</soap:envelope>
```

If you chose the document/literal WSDL binding instead, then the WSDL would be as follows:

```xml
<types>
  <schema>
    <element name="xElement" type="xsd:string"/>
    <element name="yElement" type="xsd:int"/>
  </schema>
</types>
```
This time calling the operation with “hello” and 4 as the parameter involves a SOAP message as follows:

```
<soap:envelope>
  <soap:body>
    <xElement>hello</xElement>
    <yElement>4</yElement>
  </soap:body>
</soap:envelope>
```

In general, calling web service operations from Flex is straightforward. A WSDL is first read. If the WSDL is read and parsed successfully, web service operations are invoked by passing the required parameters. Event handlers are defined for success and failure conditions, which trigger result and fault events, respectively. On success, the response is obtained from the result event object and used within a Flex application as desired.

Alternatively, a Flex application can access a web service with the help of a client side library that publishes the endpoints. Using this option, the Flex application does not need to load and parse the WSDL.

As with the HTTP service, you set the useProxy property to true in order to leverage the proxy service. You also set the id and destination properties. The id is a reference to the WebService instance, and the destination points to a remote named destination configuration in services-config.xml. Such a WebService might look as follows:

```
<mx:WebService
  id="aWebService"
  useProxy="true"
  destination="wsdestination"/>
```

Next, we explore the server-side configuration for such a web service.

**Configuring a Web Service Proxy**

Two of the most important SOAP web service proxy configurations are:

- wsd1
- soap
A web service call is a two step-process. In the first step, an HTTP GET call is made to read and parse the WSDL. The wsdl property of WebService object specifies the URI for the WSDL. In the second step a web service operation is invoked. Web service operations are called using HTTP POST method. The soap property specifies the endpoint at which the operation can be invoked. Wildcards can be included in the URL to allow for multiple operations and their multiple related URL(s).

The soap destinations can be secured using constraints and the underlying Java EE authentication and authorization schemes as with the HTTPProxyService.

An example of destination configuration is:

```
<adapters>
  <adapter-definition
    id="soap-proxy"
    class="flex.messaging.services.http.SOAPProxyAdapter"/>
</adapters>
<destination id="wsdestination">
  <properties>
    <wsdl>http://www.shanky.org/samplews.asmx?wsdl</wsdl>
    <soap>*</soap>
  </properties>
  <adapter ref="soap-proxy"/>
</destination>
```

Two web service destination properties, wsdl and soap, are shown in the configuration above. You are already familiar with WSDL. The soap property is an optional property where you can explicitly specify the SOAP endpoint URL for HTTP POST based request-response. A wildcard, as shown above, implies all endpoints are exposed. The SOAP endpoint details are contained in the port section of a WSDL.

When you use BlazeDS, you can also leverage the state management in web services, without any extra configuration. Web service cookies are stored as Java session information in BlazeDS.

Although not exhaustive, that is as much as I have to say on web services as far as the proxy features go.

**Summary**

The focus of this chapter was on the server-side proxy for HTTP requests and web services. The first part discussed the proxy configuration and characteristics of the HTTP service. In the second part the discussion was extended to include web services.

It was established early on that both the HTTP service and web services can be accessed directly without a proxy for trusted domains. In cases where the service domains don’t define an appropriate security policy in a crossdomain.xml file or where features beyond those offered through direct access are desired, developers can choose to leverage the proxy capabilities of BlazeDS.

The HTTPProxyService uses the Apache HttpClient to make the HTTP calls. The adapter for the proxy service, therefore, allows configuration of the connection manager for the HttpClient. In addition, the service and destination configuration can include settings for aspects like cookie limits, security constraints, and the content chunking policy.
Chapter 3: Using BlazeDS as a Server-Side Proxy

The HTTP proxy service allows for server-side interception and control. This can be useful for applying cross-cutting concerns, for logging, and for more effective management.

Like the HTTP proxy service, a web services proxy service allows for tunneling web service requests and responses through a server-side proxy. The benefits of server-side control and enhanced features are applicable here, too.

Flex supports RPC/encoded and document/literal style/use WSDL bindings for Web services. These bindings were revisited in the section on the web services proxy. In the section on web services proxy configuration, wsdl and soap settings were discussed.

This chapter built on what you have learned in the first two chapters. The next chapter continues the journey of exploring BlazeDS and moves on to remote procedure calls that involve objects.
Remoting between Flex and POJOs

Among all its features, remoting between a Flash player and a Java server could qualify as the central pillar of BlazeDS. Back in Chapter 2, you were made aware of the fundamentals of remoting through a naive example. In this chapter, you will leap ahead by learning the intricate details of remoting. You will be exposed to the classes, components, modules, and formats that make up the truss that supports this feature.

While Flex and AIR are considered as one side of the pipe, a Java object on the server constitutes the other end. Java objects can be of many types, some of which only respect the Java language contract and some of which that extend specific classes or implement special interfaces. They can exist in a plain vanilla JVM (Java virtual machine) instance or they can be managed within a special container. In this chapter, I talk exclusively about the simple plain Java objects, which are often commonly termed as Plain Old Java Objects, or POJO(s) for short. Read the note on POJOs to learn more about them.

**POJOs**

POJO, which stands for Plain Old Java Object, was coined by Martin Fowler, Rebecca Parsons and Josh MacKenzie in September 2000. A POJO represents a simple Java class that does not extend a specific class or implement a special interface, or is annotated by a special piece of metadata. Sometimes, though, the definition of a POJO can be a little relaxed and may include simple objects that may be decorated by certain annotations. The term POJO also implies an alternative Java object as compared to a managed and special object like an EJB (Enterprise Java Bean).

A good start into a detailed topic is always by viewing it from a high-level perspective, and that’s exactly what is coming next.
Chapter 4: Remoting between Flex and POJOs

Essentials of Remoting

Remoting, simply put, is remote procedure calls (RPC) over the wire. Flex client applications can use RPC to communicate with remote Java objects that expose their methods as services. Such client applications can pass parameters to these remote Java objects and receive the result of these remote procedures. Java programs obviously take Java data types as inputs and return Java data types from its methods. A Flex client, on the other hand, is written in ActionScript 3 (AS3) (and its associated XML-based declarative language, MXML), so it sends and receives only AS3 objects. The remoting infrastructure needs to make the two languages talk to each other and map the data types in one language to the data types in the other. BlazeDS provides this essential infrastructure, which, apart from providing the translators, includes the communication protocol and the convenience classes that make RPC happen in an effective and efficient manner. Figure 4-1 attempts to put this definition of remoting in a diagram.

![Figure 4-1](image.png)

For successful remoting, components exist both on the client and the server. These client-side components act as stubs or handles for their server-side service counterparts. Client-side programs call the remote methods on these handles as if they were local. On receiving a remote procedure call, these client-side components pass the request down to their server counterparts. The server-side service classes are the real workhorses, which provide the desired service functionality.

The request-response process over RPC hops through multiple steps between the Flex client component and the remoting service. Walking through this transmission will elucidate the roles of the pieces involved.

Remoting Communication Flow

A picture is considered worth a 1000 words, so I have put together a sequence diagram for the flow for you. Look at Figure 4-2.

In addition, I wanted to add some words of explanation, so here it is. A RemoteObject is a client component that interacts with a remoting destination. A Flex application invokes the remote method on a remoting destination via a local RemoteObject instance. Each remoting destination can expose one or more methods. When multiple methods are exposed, each of these needs to be identified uniquely at the RemoteObject stub. On the client side, a RemoteObject stub maps an object of type mx.rpc.remoting.mxml.Operation for each of the remote methods at the remote destination. The Operation object type is associated as an attribute of the remote object instance. The name of the attribute is the same as the name of the remote method.
Chapter 4: Remoting between Flex and POJOs

Therefore, if a RemoteObject were bound to a remoting service destination called someRemotingService, which exposed two remote methods, called methodA and methodB, the RemoteObject would have two attributes of types Operation, called methodA and methodB, respectively. A RemoteObject declaration and instantiation using MXML for this case could be like so:

```xml
<mx:RemoteObject
    id="aRemoteObjectInstance"
    destination="someRemotingService"
    result="aResultHandler(event);"
    fault="afaultHandler(event);"/>
```

The remote methods, methodA and methodB, could be accessible using aRemoteObjectInstance, as follows:

```actionscript
var operation1:Operation = aRemoteObjectInstance.methodA;
var operation2:Operation = aRemoteObjectInstance.methodB;
```

A remote method, such as methodA or methodB, can be called by invoking it, using either the remote object instance or the operation’s send method. For example, each of the following results in the same method call:

- aRemoteObjectInstance.methodA()
- aRemoteObjectInstance.methodA.send()
- operation1.send()

By default, all method calls are relayed over the binary AMFChannel, that transmits AMF 3 (AMF expands to Action Message Format as mentioned in the earlier chapters) binary messages over HTTP. You know from the last chapter that communication between a Flex client and a BlazeDS instance is established using a channel-endpoint pair. Channel represents the client-side pipe and endpoint is the entry point to the pipe on the server side. If preferred, messages can also be relayed as AMF in XML (AMFX) over HTTP. This textual channel is called HTTPChannel.

Remote objects at the client side connect to BlazeDS using a channel, which is part of a channel set. The purpose of a channel set is to provide a bundle of channel options so that connection managers can fail
over to an alternative lesser preferred channel in the event of unavailability of or failure at the preferred channel.

Channels are appropriately paired with endpoints so that they can connect to each other and exchange data without a problem. So an AMFChannel is paired with an AMFEndpoint, and an HTTPChannel is paired with an HTTPEndpoint.

Once a message reaches the endpoint, it is received and directed to the specified destination by the message broker. Destinations can map to local resources that reside in the web server or remote and external programs. Toward the end of the request flow and just before the actual call on the remote service sits an adapter or an assembler. The adapter is the important translator that bridges the communication gap between BlazeDS and the server side objects. POJOs use the built-in Java adapter.

A remote call to a server-side object results in a method invocation. A call to a method on a Java object always returns a value, although it can possibly be void. In some languages, null return types like void are also called none. Being a Java method, it returns a Java data type. This returned result is transferred up the wire back to the invoking client, using a route that is the reverse of the one used by the invoking call. On its way up, the result goes through a series of transformations, the most prominent of which is serialization. Serialization with translation converts Java data types to their appropriate AS3 counterparts, which become consumable objects in a Flex application. You will learn more about serialization in a later section titled “Diving into Serialization — AS3 to Java and back.”

Next, details on the classes, components, and modules that make remoting happen will be covered. Many of these were already introduced in this section, so it only builds on what you have already learned.

### Remoting-Related Classes, Components, and Modules

Starting from the client side, the first important class is RemoteObject. A RemoteObject can be instantiated in AS3 or in MXML. Therefore, a pair of classes exists, namely mx.rpc.remoting.RemoteObject and mx.rpc.remoting.mxml.RemoteObject. So, you can create an instance of the class in AS3 as follows:

```java
import mx.rpc.remoting.RemoteObject;

public var aRemoteObject:RemoteObject;

public function init():void {
    aRemoteObject = new RemoteObject();
    aRemoteObject.destination = "someRemotingService";
    aRemoteObject.addEventListener(ResultEvent.RESULT, aResultHandler);
    aRemoteObject.addEventListener(FaultEvent.FAULT, aFaultHandler);
}
```

Alternatively, you can create a RemoteObject instance using MXML as follows:

```xml
<mx:RemoteObject
    id="aRemoteObjectInstance"
    destination="someRemotingService"
    result="aResultHandler(event);"
    fault="aFaultHandler(event);"/>
```
Chapter 4: Remoting between Flex and POJOs

The MXML object is an instance of `mx.rpc.remoting.mxml.RemoteObject`, which extends from `mx.rpc.remoting.RemoteObject`. The MXML `RemoteObject` class implements two interfaces, namely `IMXMLSupport` and `IMXMLObject`, in addition to extending from the AS3 `RemoteObject` class.

Implementing the `IMXMLSupport` interface allows `RemoteObject` to be used within an MXML document via its tags. The remote object instance is a service component and does not have a visual representation. The `IMXMLObject` interface defines the API that nonvisual components need to implement for it to work properly with the MXML compiler. The MXML version of `RemoteObject` implements `IMXMLObject`. The initialized method implementation of the `IMXMLObject` type is called once the remote object is created and all the properties specified in its MXML tags are initialized.

A remote object talks to a remote service by using a channel. Flex defines a channel as a member of a channel set. A channel set is essentially a bundle of channels. One channel is usually the preferred member of such a bundle, and the others are alternatives in some order of preference. Flex supports an open source binary protocol called AMF. The current version of AMF is AMF 3. AMF 3 provides a format and a protocol to exchange data between the client and the server in a manner that is faster and more efficient than mere textual data interchange. AMF 3 builds on top of HTTP and, therefore, does not require the use of any specific ports or proprietary handshake mechanisms for participants. The class that implements an AMF-based channel is `AMFChannel`.

The `AMFChannel` extends the `NetConnectionChannel` class, which provides the basic support for messaging. An `AMFChannel` can be polling type or nonpolling type. For RPC, the polling feature can be turned off. With polling turned on an `AMFChannel` mimics a channel for pushing data.

An alternative to sending binary data is to send text-based data. A format called `AMFX` sends AMF data in XML. This sort of text-based data can be transferred over an `HTTPChannel`.

A channel has a physical connection with an endpoint, which is its counterpart on the server. Multiple destinations share this physical connection.

An `AMFChannel` binds or connects to an `AMFEndpoint`. The channel and endpoint pairs need to be compatible. BlazeDS’s `flex.messaging.endpoint` package contains the definitions for the endpoints. The specific endpoint class that couples with an `AMFChannel` is `flex.messaging.endpoint.AMFEndpoint`. Artifacts in BlazeDS can be unmanaged or managed, with the help of the JMX technology. Therefore, `AMFEndpoint` provides two constructors, one each for unmanaged and managed scenarios. Read the note on JMX technology if you aren’t familiar with it.

What Is JMX?

JMX, which stands for Java Management Extensions, is a technology that defines a convenient and easy way to manage resources such as applications, devices, and services. Every JVM of version 5 and higher supports JMX.

The JMX specification was drafted with the help of two Java Community Process (JCP) requests, referred to as JSRs. JSR is an acronym for Java Specification Request. The two JMX JSRs are:

- JSR 3 — Java Management Extensions Instrument and Agents Specification
- JSR 160 — Java Management Extensions Remote API

Continued
Resources managed by JMX are instrumented by one or more Java objects called Managed Beans, or MBeans. These resources are managed by a management agent that can be local or remote. The management agent is also referred to as the MBean Server.

JMX provides a dynamic, lightweight, and scalable architecture for management of resources in a JVM. You can read more about JMX at http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement. The Sun Java tutorials also have a section on JMX, which can be accessed at http://java.sun.com/docs/books/tutorial/jmx.

You know that a message broker resides at the heart of a BlazeDS instance. The message broker receives messages from endpoints and sends them down to its configured services, which are identified by its destination identifier. The message broker also sends back messages to the clients via the endpoints. The message broker is implemented by the flex.messaging.MessageBroker class. The MessageBrokerServlet bootstraps the MessageBroker and sets up its endpoints and services. All HTTP-based calls to BlazeDS are intercepted by the MessageBrokerServlet and then passed down to the appropriate endpoint for handling. The endpoint itself is a Servlet-based URL and is part of the MessageBrokerServlet infrastructure. The message broker Servlet extends the Java HTTPServlet class. Therefore, it has the same lifecycle as a Servlet and can be configured like any other Servlet.

The services in BlazeDS are defined within the flex.messaging.services package. The RemotingService class enables the invocation of services that support remoting. It can process remoting messages, which are abstracted in a RemotingMessage class. Message type classes such as RemotingMessage can be found in the flex.messaging.messages package. Services usually have an adapter to invoke the method. The remoting message class tells the adapter which method to invoke. It also passes in the parameters for the adapter settings and the arguments for the method that is invoked.

The JavaAdapter class provides an implementation for a basic adapter that allows you to invoke methods on server-side POJOs.

To recap, let’s list all the classes discussed so far in somewhat the same sequence as they appear during a typical POJO method call. The classes discussed so far are:

- mx.rpc.remoting.RemoteObject
- mx.rpc.remoting.mxml.RemoteObject
- mx.messaging.channels.AMFChannel
- mx.messaging.channels.NetConnectionChannel
- mx.messaging.channels.HTTPChannel
- flex.messaging.endpoint.AMFEndpoint
- flex.messaging.MessageBrokerServlet
- flex.messaging.MessageBroker
Chapter 4: Remoting between Flex and POJOs

- flex.messaging.services.RemotingService
- flex.messaging.messages.RemotingMessage
- flex.messaging.services.remoting.adapters.JavaAdapter

All the classes in the packages whose names start with flex.messaging are from the BlazeDS Java code base. Many of these components need to be configured before they are ready to be used appropriately. The next section walks you through the details of the server configuration for supporting remoting services.

**Configuring a Remoting Service and a Destination**

You were introduced to the message broker in the last section. By now, you know that the message broker is the hub of all activity in BlazeDS. The message broker receives and processes messages from the clients to the BlazeDS endpoints and passes them down to the remoting service. It also returns the results to the client using the reverse route. Therefore, almost all communication within BlazeDS gets routed through the message broker. On startup, the message broker is bootstrapped by the message broker Servlet. This activity of bootstrapping and initializing the message broker involves reading the configurations for the associated endpoints, services, destinations, and adapters. Based on the configurations, the associated artifacts and components are instantiated and initialized and hooked on to the message broker.

Such configuration parameters for the message broker and its associated artifacts are specified in the services-config.xml configuration file. This configuration file, by default, resides in the WEB-INF/flex folder within the BlazeDS distribution. As a starting point, the important configuration elements pertaining to a remoting service and a destination are shown in Figure 4-3.

From the earlier chapters, you know that services-config.xml is broken down into multiple XML files for convenience. The common configuration remains in services-config.xml, and the additional XML files contain elements that pertain to the proxy service, the remoting service, and the messaging service. In the commercial alternative to BlazeDS, that is, LifeCycle Data Services, or LCDS, there is also a file for the data management service. Each of these additional XML files is included in services-config.xml by reference. The file that contains remoting-service-specific configuration is called remoting-config.xml. So, all the important remoting-related configuration elements shown in Figure 4-3 are included in remoting-config.xml.

To explain these configuration elements, let’s walk through an example. A possible remoting-config.xml configuration is:

```xml
<service id="aRemotingService"
  class="flex.messaging.services.RemotingService">
  <adapters>
    <adapter-definition id="java-object"
      class="flex.messaging.services.remoting.adapters.JavaAdapter"
      default="true"/>
  </adapters>
  <default-channels>
```

Download from Library of Wow! eBook <www.wowebook.com>
This configuration example wraps all configurations within a `<service>` tag. Each service in BlazeDS follows this idiom, where the `service` tag is the highest level, enclosing element for a service definition. An adapter, a channel, and a destination are associated with a remoting service. In the preceding example, an AMFChannel is used for the remoting service. The service itself binds to a POJO, which serves as the destination. A Java adapter is used to talk to the POJO and works just fine.

The destination is configured with the help of properties, two of which are shown in the preceding example. These are `source` and `scope`. The `source` property specifies the fully qualified name of the POJO. This POJO should be on the web application classpath for BlazeDS to load it; that is, it should be either within the `WEB-INF/classes` folder or be packed in a Java archive file and reside within the `WEB-INF/lib` folder. The `scope` specifies the instantiation policy, the visibility, and the lifespan for the POJO instance. The `scope` can take any of the following three possible values:
Chapter 4: Remoting between Flex and POJOs

- request
- session
- application

If you are familiar with web application development, then you probably already know what these terms imply. However, for completeness, I will define these terms here as well.

A request scope implies that the instance is relevant for a request only. This means for each request, or incoming call, a new instance of the object is created. So, a POJO with request scope is instantiated by BlazeDS every time there is a new request. A request scope is good when no state needs to be maintained on the server. A request scope is the default scope.

A session scope implies that the instance is relevant for a session. All requests originating from the same browser or user agent fall within the same session. This means requests from different tabs of a browser also get counted within the same session. So, if you open up a Flex application in two different adjoining tabs of a browser and make the same remoting service call, the same POJO will be accessed, if it is scoped to be relevant for a session.

The third alternative is to share the same instance throughout the application. This implies two or more requests from two or more separate browsers or user agents see the same instance of the object. This third type of scope is the application scope.

The configuration discussed so far is typically all that’s required as far a standard POJO-based destination goes. Sometimes though, additionally, you may want to do any or all of the following:

- Secure a destination and restrict access to it
- Intercept all calls made to a particular remoting service
- Log all calls to a particular destination
- Talk to a remoting service using a text-based protocol
- Converse with a remoting service over HTTPS
- Define custom scope for destinations
- Use managed objects side by side with POJOs

So, I will take each in turn and explain how one could include these additional features. In this section, the focus is on how to configure such a feature over and above the standard configuration. In some cases configuration alone may not be sufficient, and one may need to extend the standard classes with custom implementations. In such cases, I will restrict the discussion to configuration for now and will talk about the custom extensions later, in the section titled “Value Additions on Simple Remoting” and in the following chapters.

**Securing Destinations**

In Chapter 3, you learned the fundamentals of applying security and access control to destinations. All those concepts apply to remoting services as much as they apply to proxy services.

So, you can define a basic or custom authentication scheme and have your users authenticate to access a remoting destination. You could associate the authentication on either a per-client or a per-session basis.
When authentication is configured on a per-client basis, you need to authenticate the user separately for each browser tab and also when the browser is refreshed.

Apart from authentication and the role-based authorization security constraint, access to specific methods can be controlled at the destination level. Simple method-level access control can be applied using either of the following mechanisms:

- Include all allowed methods in the `include-methods` list
- Exclude restricted methods by including them in the `exclude-methods` list

Both these lists, that is, `include-methods` and `exclude-methods`, can be specified within the `properties` tag of a destination. The `include-methods` list also allows a security constraint to be attached to an individual method and allows fine-grained security implementation. Use of the `exclude-methods` list makes the most sense when access to only a few methods is restricted and there is no need to specify security at the method level.

Listing 4-1 illustrates a sample configuration that provides a method-level security constraint. The contextual explanation of the listed configuration follows the listing itself.

**Listing 4-1: A Sample Security Configuration for a Remoting Service Destination**

```
<service id="aRemotingService"
   class="flex.messaging.services.RemotingService">
   <adapters>
      <adapter-definition id="java-object"
         class="flex.messaging.services.remoting.adapters.JavaAdapter"
         default="true"/>
   </adapters>

   <default-channels>
      <channel ref="myAMFChannel"/>
   </default-channels>

   <destination id="myPOJODestination">
      <properties>
         <source>problazeds.ch04.MyPOJO</source>
         <scope>session</scope>
         <include-methods>
            <method name="methodA"/>
            <method name="methodB" security-constraint="admin-users"/>
         </include-methods>
      </properties>
   </security>
</destination>

<destination id="mySecondPOJODestination">
   <properties>
      <source>problazeds.ch04.AnotherPOJO</source>
   </properties>
</destination>
```
The preceding listing depicts four different aspects of securing a destination. These are:

- Security can be defined inline, within a destination configuration. Notice the presence of an inline security definition for regular users within `myPOJODestination`.

- Method-level security constraint can be defined for methods in the `include-methods` list. Only `admin-users` are allowed access to `methodB` of `myPOJODestination`. The security definition for `admin-users` is defined in `services-config.xml` and maps to either a basic or custom authentication scheme and a role. Refer to the section titled "Access Control to Proxy Destinations" from Chapter 3 for details on how to configure a security in `services-config.xml`.

- The security constraint at a destination and a method level can be different. The method-level constraint applies in precedence to the destination-level constraint for the particular method. This outcome is logical as the method-level constraint is more granular than the one at the destination level.

- Default security constraint can be defined at the service level. Such a constraint applies to all destinations that do not define a security constraint. In the listing above, `mySecondPOJODestination` uses the default security constraint, since it doesn’t define one of its own.

With these out-of-the-box facilities you can secure destinations and restrict access to specific roles and user groups without much trouble. You can also implement custom authentication and authorization schemes, which may not be available off the shelf, and use them with BlazeDS. The section titled “Value Additions on Simple Remoting” explains how to create such custom authentication and authorization schemes.

BlazeDS supports Role-Based Access Control (RBAC), which is a dominant style of access control within enterprise applications, so its security architecture works well in most situations. However, RBAC is not the only form of access control and when alternatives are required customization could involve some heavy lifting and major modifications. Read the note on access control techniques to understand the alternatives to RBAC.

---

**Access Control Techniques**

There are two important participants in any system: subject and object. A subject is an entity that can carry out tasks and an object is a resource that may need controlled access. Tasks carried out by a subject impacts the concerned objects.

Access to objects can be controlled by using one of the two popular models: capability-based or Access Control List (ACL) based. In capability-based models the access is restricted to entities who own the capability, which can be thought of as an access key. In ACL-based models, access is restricted to subjects on the list.

Continued
To implement access control, software systems typically use one of the following three techniques:

- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC)
- Role-Based Access Control (RBAC)

In a DAC-based system, the owner of an object decides which subjects get control over the objects. The owner also decides the type of control a subject gets. An object’s owner often is the creator of that object, and a typical example of a DAC-based system is the file system.

MAC is relevant where fine-grained access control is desired. In a MAC-based system, the access control is defined at a system level and the owner does not define the access control policy. Often, sensitivity-level labels are attached to subjects and objects. A subject can access only those objects that fall within the same or lower sensitivity level than the subject’s sensitivity level. MAC-based systems are also rule-based access control systems. In the simplest case, the sensitivity matching rules grant and deny access. For complicated scenarios that could involve multiple subjects and objects at different levels of sensitivity, the rules could be complex. Sometimes, Lattice-Based Access Control (LBAC) can be used in mandatory access control systems for such complex scenarios. For example, two subjects, A and B, are given access to an object only if the greatest lower bound of the sensitivity levels of these two subjects matches with the required sensitivity level for the object.

RBAC is the most popular form of access control in enterprise systems. Permissions are clubbed together in a role and users are assigned to the role. Roles can have hierarchical structures, and users can be assigned to roles based on additional rules.

In some cases, you may not only want to secure the remoting service but may also want to intercept all calls to it. The reasons for such an interception could range from auditing to enrichment, through rule-based filters.

Next, we explore the ways to intercept calls to a remoting service destination.

**Intercepting and Logging Calls to a Remoting Service**

All calls to a BlazeDS based server-side service can be intercepted using Servlet filters. BlazeDS is a Java Servlets based web application and configuring Servlet filters to work with BlazeDS is easy. For example you could set up a Servlet filter to intercept all calls to the BlazeDS message broker as follows:

```xml
<filter>
  <filter-name>AFilter</filter-name>
  <filter-class>org.shanky.filters.AUsefulServletFilter</filter-class>
</filter>

<filter-mapping>
  <url-pattern>/services/*</url-pattern>
</filter-mapping>
```
In addition, calls can be logged both on the client and on the server.

For auditing, monitoring, and debugging, the built-in logging facility is fairly robust and easy to configure. The following can be configured for the logging facility:

- **Logging target** — You can specify that the log output be written to the console or be directed to the default logger configured for the Servlet container in which BlazeDS resides.

- **Logging level** — You can set the logging level to All, Debug, Error, Info, None, or Warn. The logging level determines what types of messages are logged. Warn shows both warning and error messages, Error shows error messages only, and Debug shows debugging messages apart from warnings and errors. Info shows general information messages, in addition to errors. All and None show all and nothing, respectively.

- **Log content metadata** — You can specify to include the following:
  - **Category** — The BlazeDS subsystem or area that generated the message
  - **Date** — Log message date
  - **Time** — Log message time
  - **Level** — The chosen logging level
  - **Prefix** — a string to be prefixed to log messages

In addition, you can filter the messages and log only those that match particular category. For example, all messages generated by the remoting-related artifacts only, such as RemotingService and RemotingDestination, can be logged using the Service.Remoting filter. This can be very useful, especially if your system is complex and you only want to focus on log messages generated by a part of the system. Finer-grained logging at the level of each named destination can be a further enhancement on this feature. In Chapter 8, I show how such an enhancement can be included in BlazeDS.

Before we jump to the next topic on the list, let’s look through a sample logging configuration that captures only messages generated by the remoting service. Here it is:

```xml
<logging>
  <target class="flex.messaging.log.ConsoleTarget" level="Debug">
    <properties>
      <prefix>[Professional BlazeDS Chapter 04]</prefix>
      <includeDate>true</includeDate>
      <includeTime>true</includeTime>
      <includeLevel>true</includeLevel>
      <includeCategory>false</includeCategory>
    </properties>
    <filters>
      <pattern>Service.Remoting</pattern>
    </filters>
  </target>
</logging>
```
Talking AMFX

AMF 3 is a message format that encodes objects to a binary format and can be used in conjunction with HTTP to exchange data between a Flex application and a server. Such a server could be a Java server that uses BlazeDS. Apart from the binary option, it’s also possible to convert the AMF 3 messages to XML and then exchange them over the wire. AMF represented in XML is AMFX. AMFX is a text-based data exchange.

Use AMFX with remote object over alternatives like XML over HTTPService, especially if BlazeDS is already part of the infrastructure. Using AMFX with remote objects implies auto-translation of Java types to AS3 and vice-versa without the need of explicit hand-coded marshaling and un-marshaling as with XML over HTTPService. Besides, it allows seamless swap in of AMF in place of AMFX if need be.

To transmit data using AMFX you need to use the HTTPChannel and the corresponding HTTPEndpoint. Your services-config.xml file probably already includes a channel definition for the HTTPChannel and HTTPEndpoint pair as follows:

```xml
<channel-definition id="my-http" class="mx.messaging.channels.HTTPChannel">
  <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/http" class="flex.messaging.endpoints.HTTPEndpoint"/>
  <properties>
    <piggybacking-enabled>true</piggybacking-enabled>
    <invalidate-session-on-disconnect>true</invalidate-session-on-disconnect>
  </properties>
</channel-definition>
```

To use this channel just reference the channel by its name within the remoting service configuration.

The AMF11 channel and endpoint are supported by a set of serialization and deserialization classes that convert Java data types to AS3 and back via AMF in XML. The AmfxTypes interface provides the signature for the class that enables the XML representation of AMF11. AmfxOutput is the class that serializes Java data types to AS3 via AMFX. AmfxMessageDeserializer is a SAX based AMFX parser. AmfxInput provides the context for AmfxMessageDeserializer.

The AMFX channel is called the HTTPChannel, although AMFXChannel would have been a better name for it. The AMFChannel also uses HTTP for data exchange and explicitly calling the AMFX channel the HTTPChannel almost misleads developers to believe that the AMFChannel probably does not use HTTP. Also, the HTTPChannel is sometimes confused with a generic text-based channel for object data exchange, which is not true. It specifically exchanges data as AMFX. If you choose to encode your objects using formats like XML-RPC or JSON, or any format other than AMFX, then you need to write a custom channel and endpoint pair with its own set of classes for serialization and deserialization.

Whether you’re using AMF or AMFX, you may want to leverage the HTTPS protocol instead of plain HTTP and prevent the data exchange from being snooped into by intruders. That’s what comes next.

Using HTTPS

Using an HTTPS channel in BlazeDS merely requires a simple configuration change. Just remember to use the secure HTTPS channels for AMF and AMFX. HTTPS channels are defined in services-config.xml like so:

```xml
<channel-definition id="my-http" class="mx.messaging.channels.HTTPChannel">
  <endpoint url="https://{server.name}:{server.port}/{context.root}/messagebroker" class="flex.messaging.endpoints.HTTPEndpoint"/>
  <properties>
    <piggybacking-enabled>true</piggybacking-enabled>
    <invalidate-session-on-disconnect>true</invalidate-session-on-disconnect>
  </properties>
</channel-definition>
```
Chapter 4: Remoting between Flex and POJOs

<channel-definition id="my-secure-amf" class="mx.messaging.channels.SecureAMFChannel">
  <endpoint url="https://{server.name}:(server.port)/
    (context.root)/messagebroker/amfsecure"
    class="flex.messaging.endpoints.SecureAMFEndpoint"/>
  <properties>
    <add-no-cache-headers>false</add-no-cache-headers>
    <connect-timeout-seconds>5</connect-timeout-seconds>
  </properties>
</channel-definition>

<channel-definition id="my-secure-http" class="mx.messaging.channels.SecureHTTPChannel">
  <endpoint url="https://{server.name}:(server.port)/
    (context.root)/messagebroker/httpsecure"
    class="flex.messaging.endpoints.SecureHTTPEndpoint"/>
  <properties>
    <add-no-cache-headers>false</add-no-cache-headers>
  </properties>
</channel-definition>

These HTTPS channels exchange data over HTTP, using Secure Socket Layer (SSL). In LifeCycle Data Services, the data push channel based on RTMP uses Transport Layer Security (TLS). If you need to use TLS, you will have to handcraft those channels for your use.

If you don’t know the difference between SSL and TLS, then you may benefit from reading the “SSL vs. TLS” sidebar in this chapter.

SSL vs. TLS

SSL, or Secure Socket Layer, is a protocol for secure communication between a client and server. SSL was invented by Netscape and has gone a few revisions up to its current version 3.0. TLS, which stands for Transport Layer Security, supersedes SSL as the current-generation protocol for secure communication between a client and a server. The TLS standard was formed through an IETF initiative: RFC 2246. You can access the RFC 2246 documentation at www.ietf.org/rfc/rfc2246.txt. Although similar in intent, SSL and TLS do not interoperate. Systems using TLS, however, can be downgraded to work as SSL 3.0.

Earlier when I listed a few extension and exception cases, I also included situations that involve custom scoping and managed objects. Both these topics are discussed in the following chapters. Managed objects are the central theme of Chapter 5. Custom scopes are touched upon in the context of the Java Persistence API (JPA) and Hibernate in Chapter 7.

At this point, our coverage of remoting service configuration is fairly complete. Nonetheless, you will continue to incrementally learn more about it as you explore the customization and extension mechanisms in the following chapters. Next, we get under the hood of the serialization and deserialization story that forms the backbone of the data services’ features.
Chapter 4: Remoting between Flex and POJOs

Diving into Serialization — AS3 to Java and Back

Flex talks AS3 and BlazeDS talks Java, yet they have a fluent conversation. The secret behind this smooth flow of data between the two sides is the presence of the serialization and deserialization logic elements within BlazeDS. This logic element is the one that appropriately converts data types in one language to the types in the other. To understand how this element works and how you could help it do its job well, it is worthwhile to understand how the data types on the two sides map to each other.

The simplest case of mapping arises with the simple and primitive types, for example String, Boolean, and integer. An integer in AS3 can be int or uint implying signed and unsigned integers respectively. These simple data types convert over to exact corresponding types in Java, as follows:

- **String (AS3)** — java.lang.String (Java)
- **Boolean (AS3)** — java.lang.Boolean (Java)
- **int/uint (AS3)** — java.lang.Integer (Java)

However, BlazeDS is smart enough to support related types as well. For example, passing in an AS3 int type to a Java method that takes java.lang.Double, java.lang.Long, primitive double, or primitive int argument will not fail, but the value will be automatically converted over to the respective data type. Similarly, an AS3 Boolean type could be forced to map to a java.lang.String type, where it would pass in “true” and “false” string values to the method.

AS3 has a Number type that holds numerical values. The numerical values can be floating point type or integral. The int and uint data types convert to a java.lang.Integer type. However, a Number defaults to a floating point type and represents a number with a decimal part. Therefore a Number maps to java.lang.Double. However, other numeric types can also receive an argument that is of a Number type. These possible numeric types in Java are:

- java.lang.Long
- java.lang.Float
- java.lang.Integer
- java.lang.Short
- java.lang.Byte
- java.math.BigDecimal
- java.math.BigInteger
- double
- long
- float
- int
- short
- byte
Chapter 4: Remoting between Flex and POJOs

On reverse flow any of the above mentioned Java numeric types, except `java.lang.Integer`, and `int` are converted to an AS3 Number. The integer types convert to an AS3 `int`.

AS3 `Date` by default maps to UTC (Universal Time) formatted `java.util.Date`. However an AS3 `Date` type passed in to a server-side method is also compatible with any of the following date and timestamp types:

- `java.util.Calendar`
- `java.sql.Timestamp`
- `java.sql.Time`
- `java.sql.Date`

Sometimes peculiar behavior can occur with numeric types when the values are null. By default, the AS3 `null` and `undefined` values pass in as `null` on the Java side. With AS3 `Number`, though, null values get converted to 0 and are passed as such. Therefore a null assignment to a `Number` is received as a 0-valued Java integer or double. This is not a problem at all for Java primitive types, as they take default values and cannot be set to null anyway. However, this can get tricky and troublesome when using the object types: `Integer`, `Double`, `Long`, and `Float`. In these cases, too, the value is received as a 0-valued quantity. This may not always be correct. For example, you may have a user input form in which you may ask a user to input his or her age (rounded to the closest number of years) in a form field. If the user does not enter any value into such a form field, which holds a value of type `Number`, then the value is actually `null` (or unspecified). However, a `Number` cannot have a null value so it converts it over to 0 and that is what the Java `Integer` receives at the server side. Do you see the discrepancy? Because of the error, the user who filled in the form is technically less than 6 months of age, which is not what the user said.

To handle such problems one needs to carefully choose data types and in some cases resort to extending the `flex.messaging.io.BeanProxy` class, which provides a means to intercept the getter and setter calls on the Java side. You could possibly have chosen an AS3 `Object` type for the form field in the earlier case and the exception would not have occurred. However, then the handling of regular numerical values would need more effort. So choose a data type after considering such exceptions and border cases, while understanding the possible overheads of alternatives. The extension of the `BeanProxy` behavior is a handy way to inject custom conversion rules. Starting with Chapter 7 (Leveraging Hibernate and JPA with Flex), you will see many examples where the `BeanProxy` is extended.

Although some of the border cases, such as null value handling, are challenging, many of the serialization and deserialization rules you have seen so far are straightforward. With AS3 array types, though, the conversion rules get complex.

The most sophisticated and complicated conversion rules exist for array types. AS3 arrays can either have numerical indices only or can have string values for indices as well. When indices contain only numerical types, the arrays are called `strict` arrays. When indices include string values, the arrays are of `associative` array type. Associative array types map to a `java.util.Map` type. Strict array indices can be continuous (starting from 0) or noncontinuous. If the indices are continuous and starting from 0 then such `strict` arrays are `dense`. Dense arrays map to a `java.util.List` data type. If the indices are noncontinuous (that is, there are holes or gaps between the elements), then such `strict` arrays are `sparse`. Sparse arrays map to a `java.util.Map` type. If they are mapped to a `java.util.List` type, as the `dense` arrays are, you may see a whole lot of null values in the list that correspond to the gaps. You don’t want to waste space and computing resources, so the choice of a `Map` is prudent.
Chapter 4: Remoting between Flex and POJOs

To reinforce what you have learned about arrays and their conversion rules so far, let's illustrate with a few simple examples. A strict and dense array in AS3 is:

```javascript
var anArray:Array = new Array();
anArray[0] = "Jack";
anArray[1] = "Jill";
anArray[2] = "Hansel";
anArray[3] = "Gretel";
```

This array (anArray) when converted to Java would be by default of a `java.util.List` type. Alternatively, you could have an associative array for the same data set, defined as follows:

```javascript
var anotherArray = new Array();
anotherArray["boy1"] = "Jack";
anotherArray["girl1"] = "Jill";
anotherArray[3] = "Hansel";
anotherArray[4] = "Gretel";
```

This associative array translates to a `java.util.Map` type. The indices become the keys, and the array items become the value. In some cases, a strict array also translates to a `java.util.Map` type. This happens when the strict array is sparse and has many gaps between its indices. You know that a sparse array translated to a `java.util.List` type could lead to storage of a lot of null values. A sparse array is:

```javascript
var aSparseArray = new Array();
aSparseArray[1] = "Jack";
aSparseArray[2] = "Jill";
aSparseArray[34] = "Batman";
aSparseArray[35] = "Robin";
```

So far, only the default conversions for AS3 arrays have been elucidated. What if the method at the Java end takes in arguments of `java.util.Collection` type? Such a method could receive a dense AS3 Array. Depending on the interface type of the argument the following conversions are possible:

- List — ArrayList
- SortedSet — TreeSet
- Set — HashSet
- Collection — ArrayList

If the incoming array is a sparse array, then the Java method will still receive a `java.util.Map` type. So, it's pertinent that you closely verify the array type upfront and define your collection classes on either side of the wire. At times, one side of the puzzle may be given. In such cases, you need to reconfirm that the data structure across the wire is compatible with it.

On its way up from the server to a client a `java.util.Collection` type translates to an `mx.collections.ArrayCollection` type unless the channel is explicitly configured to convert such a collection to an AS3 Array type. BlazeDS allows specific property setting on channels for AMF serialization that support legacy, especially Flex 1.5.

While these built-in and standard types are critical for serialization, custom types or typed objects are usually the units that are passed back and forth. Typed objects, of course, have fields that can be of a
custom type, a built-in type, or a standard type. Think of a “person” type with these fields: name, ID, address, and phone number. The name can be a string, the ID and the phone number can be numerical, and the address could be a custom type. An address custom type could have fields like street, city, state, and zip code, which could be string, string, string, and number respectively. Such a typed object could be represented as a Java bean type with a set of getter and setter pairs for each attribute. In addition, for BlazeDS to instantiate it on demand it needs to have a no arguments constructor. On the AS3 end such a class could again have a structure that resembles the Java bean structure of attributes with a getter and setter pair for each. The Flex framework includes a metadata type called `RemoteClass` that can be used to annotate AS3 classes and specify their server-side counterparts. Java server-side classes are mapped by specifying their fully qualified class name within the `[RemoteClass(alias="")` metadata tag. The serializer and deserializer convert between AS3- and Java-typed objects using the value provided in this metadata element.

BlazeDS serializes all the bean properties and public fields of POJOs across the wire. No constants, transient properties, static properties, read-only properties, or private properties are serialized. The bean properties are introspected using the `java.beans.Introspector` class, which returns the property descriptors via the `BeanInfo` object types. A `BeanInfo` type specifies the methods, properties, and events of a Java bean. In addition to the bean properties that have a getter and setter pair, a POJO could contain public fields. The public fields are extracted using reflection.

On the Flex application side, the serialization and extraction is done by the Flash player. AS3 object properties, other than those that are transient, are extracted and passed across the wire.

BlazeDS translates between Java and AS3 using rules, some of which were specified and illustrated in this section. Sometimes, though, you may desire to provide custom rules for serialization.

Custom serialization is supported with the help of the `flash.utils.IExternalizable` at the Flash client side and `java.io.Externalizable` at the Java server side. On both sides the externalizable interface has `readExternal` and `writeExternal` methods for custom serialization.

On the Flash client side, two I/O interfaces define the input and output streams for reading and writing binary data. The interface for data input is `IDataInput`, and its counterpart for writing is `IDataOutput`. The `IDataInput` interface defines read methods for both primitive and object data types. The primitive read methods rely on built-in standard deserialization and the read object methods could utilize custom rules for deserialization. `IDataOutput` provides for analogous methods for writing. So, you could write primitive types and objects.

An example of custom serialization that involves `IDataInput` and `IDataOutput` within the `writeExternal` and `writeExternal` methods could be like so:

```java
public function readExternal(input:IDataInput):void {
    floatProperty1 = input.readFloat();
    object1 = input.readObject();
}

public function writeExternal(output:IDataOutput):void {
    output.writeFloat(floatProperty1);
    output.writeObject(object1);
}
```
Chapter 4: Remoting between Flex and POJOs

On the Java server side, two I/O interfaces, ObjectInput and ObjectOutput, can be used to read and write object streams. These interfaces serve the same function that IDataInput and IDataOutput serve and work with the readExternal and writeExternal methods on the Java side.

Figure 4-4 depicts custom serialization between a Flex application and a Java object on the server.

Running through a Complete Example

This section walks through an example that shows the BlazeDS remoting service in use. I use Apache Tomcat as the Servlet container and deploy a BlazeDS instance in it. From Chapter 2, you know how to deploy BlazeDS in Tomcat and many other mainstream Java application servers. For convenience, I also use Flex Builder as my IDE.

In this example, I take the energy consumption data for United States between the years 1949 and 2007 and depict that in a data grid and a time series line chart.

The data is accessible from a POJO service object called EnergyConsumptionDataService. The data itself manifests in an object-oriented format and is transferred as such. The consumption for individual data points for the time interval is encapsulated in an object called EnergyConsumption. The data for the entire time interval is a collection of such objects. The EnergyConsumption class has the following attributes:

- year
- fossilFuels
- nuclearElectricPower
- renewableEnergy
- total
As a first step, I create a Flex project in Flex Builder and choose a J2EE server. To keep all the code in one place in a uniform manner I also choose to create a combined Java/Flex project using WTP (Web Tools Platform). WTP comes as a part of the Eclipse JEE version and you will benefit from installing it. Figure 4-5 shows the first screen where this initial project information is specified.

Next, I point the target runtime to my local Tomcat installation. The assumption is that Apache Tomcat is already configured to work with the Eclipse installation. In addition, one needs to specify the following:

- Context Root
- Content Folder
- Flex WAR File

Flex Builder, by default, sets the project name as the Context Root value. The Flex WAR File variable is a legacy name, and for BlazeDS it points to the `blazeds.war` file. I chose to point to the `blazeds.war` archive file within my Tomcat installation. This isn’t a requirement though. The war file could actually be anywhere in the file system. Figure 4-6 is a view of the screen where the context root, content folder, and war file path are specified.

Now you’re ready to choose the MXML main application file name and the application URL. For now, you may just go with the defaults, which are already populated. See Figure 4-7 for details.

At this stage the project is set up in Flex Builder, and you are ready to add the code that will drive the example application. Let’s build the service first.
Java-Based Data Service

The Java-based service is exposed through a remote method of the `EnergyConsumptionDataService` class. In our rudimentary example, that method is pretty much all that class has. The source for this service class is in Listing 4-2.

**Listing 4-2: Service That Returns the U.S. Energy Consumption Record from 1949 to 2007**

```java
package problazeds.ch04;

public class EnergyConsumptionDataService {
    public List getEnergyConsumptionData() throws RuntimeException {
        List list = new ArrayList();
        Connection c = null;

        try {
            c = HSQLDBConnectionHelper.getConnection();
            Statement s = c.createStatement();
```

Figure 4-6

Figure 4-7
Chapter 4: Remoting between Flex and POJOs

ResultSet rs = s.executeQuery("SELECT * FROM energy_consumption ORDER BY year");
while (rs.next()) {
    list.add(new EnergyConsumption(rs.getInt("year"),
        rs.getDouble("fossil_fuels"),
        rs.getDouble("nuclear_electric_power"),
        rs.getDouble("renewable_energy"),
        rs.getDouble("total")));
}
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e);
} finally {
    HSQLDBConnectionHelper.close(c);
}
return list;
}

As you browse through the getEnergyConsumptionData method code, you will notice that all the persistence and data-access-related heavy lifting is done by the HSQLDBConnectionHelper class. This application cuts out the complexity induced by full-blown relational databases and instead uses the lightweight embedded HSQL database as the data store. HSQL DB’s four main advantages, which come handy here are:

- It can be embedded and can run within the same JVM as the application.
- It has a low footprint and needs little or no administration.
- It can link to CSV and text files on the file system and allow you to interact with it as you do with the relational tables, fairly effortlessly. The data for this example resides in a CSV file.
- You can use the good old SQL and the JDBC semantics to interact with HSQL DB

You can learn more about HSQL DB at http://hsqldb.org.

The HSQLDBConnectionHelper class sets up and interacts with the HSQLDB instance. It creates a singleton instance of the class, configures it, and sets up a connection. The bulk of all these activities, other than the JDBC driver class loading and the specification of the database URL, gets done in the static getConnection method, which looks like this:

```java
public static Connection getConnection() throws SQLException {
    if (singletonInstance == null) {
        singletonInstance = new HSQLDBConnectionHelper();
    }
    try {
        Connection conn = DriverManager.getConnection(singletonInstance.url);
        final StringBuilder createTable = new StringBuilder();
        createTable.append("CREATE TEXT TABLE energy_consumption (";
        createTable.append("year INT PRIMARY KEY, fossil_fuels DOUBLE,"
            createTable.append("nuclear_electric_power DOUBLE, renewable_energy DOUBLE,"
            createTable.append("total DOUBLE")");
```
final StringBuilder linkTable = new StringBuilder();
linkTable.append("SET TABLE energy_consumption SOURCE ");
linkTable.append(""us_energy_data.csv"");
linkTable.append(";ignore_first=true;all_quoted=true\"");

conn.setAutoCommit(true);
conn.createStatement().execute(createTable.toString());
conn.createStatement().execute(linkTable.toString());
return conn;
} catch (SQLException e) {
    throw e;
}

This class also defines a close method for graceful cleanup. The service returns a collection of EnergyConsumption objects. The EnergyConsumption class is depicted in a class diagram in Figure 4-8.

<table>
<thead>
<tr>
<th>EnergyConsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- year (int)</td>
</tr>
<tr>
<td>-- fossilFuels (double)</td>
</tr>
<tr>
<td>-- nuclearElectricPower (double)</td>
</tr>
<tr>
<td>-- renewableEnergy (double)</td>
</tr>
<tr>
<td>-- total (double)</td>
</tr>
</tbody>
</table>

Figure 4-8

That is all as far as our service goes. The service is configured in remoting-service.xml as follows:

<service id="remoting-service"
    class="flex.messaging.services.RemotingService">
    <adapters>
        <adapter-definition id="java-object" class="flex.messaging.
            services.remoting.adapters.JavaAdapter" default="true"/>
    </adapters>
    <default-channels>
        <channel ref="my-amf"/>
    </default-channels>
    <destination id="energyConsumptionData">
        <properties>
            <source>problazed.ch04.EnergyConsumptionDataService</source>
        </properties>
    </destination>
</service>

This service is called from a simple Flex application, which you will learn about next.
**Client-Side Flex Code**

The Flex application is aggregated together in a single MXML file. This is not ideal for a real-life example but seems functional enough for this example. In addition to this application file, there is the class that represents the client-side counterpart of the `EnergyConsumption` object. Listing 4-3 lists the code for the main MXML application.

**Listing 4-3: ProBlazeDSCh04.mxml**

```xml
<?xml version="1.0" encoding="utf-8"?>
    layout="vertical">
    <mx:RemoteObject id="srv1" destination="energyConsumptionData"/>
    <mx:Button label="Get Data"
        click="srv1.getEnergyConsumptionData()"/>

    <mx:DataGrid dataProvider="{srv1.getEnergyConsumptionData.lastResult}"
        width="100%" height="100%"/>

    <mx:Panel title="Line Chart">
        <mx:LineChart id="myChart"
            dataProvider="{srv1.getEnergyConsumptionData.lastResult}"
            showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
dataProvider="{srv1.getEnergyConsumptionData.lastResult}"
                categoryField="year"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:LineSeries
                    yField="fossilFuels"
                    displayName="Fossil Fuels"/>
                <mx:LineSeries
                    yField="nuclearElectricPower"
                    displayName="Nuclear Electric Power"/>
                <mx:LineSeries
                    yField="renewableEnergy"
                    displayName="Renewable Energy"/>
                <mx:LineSeries
                    yField="total"
                    displayName="Total"/>
            </mx:series>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

The illustration of the service, client, and the configuration covers all aspects of the example.

The code for this simple example is available for download. You may want to modify it, play with it, and experimentally explore how things work.
Now that you know all the essentials of remoting between Flex and POJOs, you may want to start exploring the ways and means of customizing and extending the code beyond the standard mechanism.

Value Additions for Simple Remoting

In this section, you will learn to extend the remoting facility in BlazeDS to include a couple of additional features that add value on top of what is already available. This is not a section that illustrates every possible extension scenario. It simply can’t, as there are too many possible cases. Many of the subsequent chapters in this book pick up some of these customization and extension topics individually.

The topics covered in this section pertain only to the following:

- Custom security implementation
- Extension of the JavaAdapter to include an interception point

Custom Security Implementation

You already know many things about the BlazeDS security from the section on configuration in this chapter and from the last chapter. Here, you will learn how to extend the standard model and implement a custom security model.

BlazeDS has built-in security implementations for most mainstream application servers, including the following:

- Tomcat
- JBoss
- Weblogic
- Websphere
- Oracle AS
- JRun

However, if you happen to use any of the other alternatives, including:

- Glassfish
- Geronimo
- ATG Dynamo

Or, if you like to delegate security to an underlying framework such as Spring or an independent identity manager, then you have a little work to do before things are ready for use.

BlazeDS security implements ways to authenticate and authorize users. Authentication identifies a user, and authorization makes sure the user engages only in permitted actions.

BlazeDS deploys within a Servlet container and a Java EE application server and so delegates its authentication and authorization as far as possible to the underlying server infrastructure. Therefore, the interface
that defines the LoginCommand has an abstract implementation in AppServerLoginCommand, and this abstract implementation forms the basis for the application-server-specific implementations. Figure 4-9 depicts this.

All classes are in the flex.messaging.security package. Essentials classes are distributed as part of the core module and application server specific implementation are part of the opt module.

So, the first extension point could possibly be the AppServerLoginCommand itself. The abstract class implements authorization by means of a set of overloaded methods. The first of these takes principal and a list of roles as parameters, and the second one takes an HttpServletRequest as an additional parameter. The authorization command relies on the isUserInRole method of the HttpServletRequest object to check if the user’s group or role has the correct permissions for the action. The isUserInRole method returns a Boolean value and acts as a guard for authorization verification.

So an implementation for Glassfish or Geronimo could be implemented by extending the AppServerLoginCommand. With Geronimo, you will also be able to use the TomcatLoginCommand out of the box, and using the TomcatLoginCommand with Glassfish would only require a few small changes. The Glassfish web server is quite similar to Tomcat and uses the concept of “Value” and the associated pipeline like Tomcat. There are a few differences between the two though.

Tomcat’s Value interface, org.apache.catalina.Value, defines an invoke method that takes a request and a response object as its parameters. The invocation in Tomcat follows a three-step process: pre-processing, invocation of the next value, and post-processing. This gets repeated for each value in the pipeline.
Chapter 4: Remoting between Flex and POJOs

Glassfish improves on this single invocation model and provides two points of invocation, one called `invoke` and another `postInvoke`. The calls to `invoke` and `postInvoke` can go on to the next value in the chain or return the call to the container and stop the chain of execution.

BlazeDS security implementation implements a `TomcatLogin` interface to support the concept of `Value`.

Spring security can also be implemented by extending the `AppServerLoginCommand`, in the same way that it is extended to support additional application server security models. You will learn more about Spring security in the next chapter.

BlazeDS provides all security concerns within a `FlexContext`. `FlexContext` exposes the current execution context. You know that BlazeDS can be made to authenticate on a per-client or per-session basis. In the case of per-client authentication, `FlexClient`, which represents a client on the server, comes, handy. On the other hand, `FlexSession`, which represents a Flex session on the server, comes, handy when authentication is set on a per-session basis. In both cases, the authentication and authorization involve the J2EE session concept. Therefore, stateless architectures in BlazeDS, which require authentication for every request, will need a lot more work. By default, an authenticated session will be allowed to go ahead, and this may disrupt an “authenticate every time” model.

When implementing a security scheme that does not use J2EE security at all, remember to plug all the pieces in around passing of credentials and handling of exceptions. When delegating the security scheme to the underlying JEE application server, the propagation of credentials from the client to the server and the mapping to the application server roles is implicit and does not require additional work on the part of the developer. Use the `ChannelSet` for such interactions with the server-side security provider.

Although you need to wait a little more before you see a full implementation of a custom security scheme, this section should have given you a sense of how to go about implementing security.

Next, we explore how to create a generic interception point for the `JavaAdapter`.

**An Interceptor for the JavaAdapter**

A Java adapter is a BlazeDS class that sits between the remoting service destination and the actual POJO on the server. Acting as the last interception layer before the Java object, it is responsible for the invocation of a Java method. The adapter itself is stateless and is instantiated every time on a request. Adapter instances can be bound to a session if explicitly configured to do so.

The `JavaAdapter` class extends `flex.messaging.services.ServiceAdapter`. Every message passing through the adapter is intercepted and handled by the `invoke` method of the adapter. The `invoke` method has the following signature:

```
public Object invoke(Message message)
```

So, adding a generic interceptor in this adapter could imply including it within the `invoke` method. Thus, every passing message would give the interceptor a chance to add the additional value if it needs to.

Let’s walk through the `invoke` method code and understand its flow. That will help you find the right place to introduce the interceptor. The `invoke` method goes through the following flow:
Chapter 4: Remoting between Flex and POJOs

- First get a reference to the destination to which the adapter is configured.
- Then cast the message as a RemotingMessage.
- Get the operations and its parameters with the help of the remoting message.
- Run the operations through the include and exclude list to ensure that the right credentials exist for method invocation.
- Invoke the method.
- Cache the invoked method to the session if the adapter is stateful.

An interception point could be introduced anywhere in this flow, but at pre-invocation seems like a good time to introduce it.

This approach assumes, though, that you would like to intercept the call close enough to the Java object itself. Alternatively, you could set up interception layers closer to the point where the data moves between Flex and BlazeDS or close to the destination invocation.

In the next chapter, you will also learn to use Aspect-Oriented Programming (AOP) in conjunction with Spring and BlazeDS, which would provide yet another way to manage a generic interception framework.

Summary

From a humble beginning expositing the essentials of remoting, this chapter graduated to a deep dive into the things under the hood.

The chapter started by explaining what remoting is between Flex and Java. It defined POJOs and laid out the context within which they interact with Flex clients. The process of remoting was then viewed from alternative perspectives. First, a flow view showed how the request originates and moves through the wire. Then a structural view brought the key classes and artifacts to light.

Once the definition and flow of remoting was explained, the discussion moved on to configuration and setup. This section explained the entire necessary configuration and peeked into some of the advanced possibilities.

After configuration, the exploration focused on data serialization. Serialization between AS3 and Java is one of the key aspects of BlazeDS, and this section delved into the serialization to a substantial level. The section talked about data types and their mapping, explained exception cases, and discussed custom serialization.

To summarize and aggregate the different bits and pieces of BlazeDS remoting into a whole, an example was presented. This example was not sophisticated but was robust enough to illustrate some of BlazeDS’s key features.

After the example and just before this summary is a small section that starts to scratch the surface of advanced BlazeDS and its customization. Many of the following chapters talk exclusively about advanced features of BlazeDS and its interesting extensions, so this section gets you started on the topic.

This chapter was all about POJOs, and the next one is about managed objects, which can be thought of as the antithesis of POJOs. In some ways, the next one starts where this one leaves off.
Although custom BlazeDS factories have helped remote to Spring beans for a while, the latest Spring BlazeDS integration project is the most comprehensive and officially supported solution to accessing the Spring framework facility from a Flex application. This chapter focuses exclusively on the Spring BlazeDS integration project. The current version of the Spring BlazeDS integration project is the 1.0.0.release.

The chapter assumes that you have some knowledge of the Spring framework. If you are a newbie who has no exposure to the Spring framework, then consider reading the following before moving ahead with the rest of this chapter:

- The first few chapters of the Spring framework reference documentation, available online at www.springsource.org/documentation.

Let’s start with basic bootstrapping and making BlazeDS work with Spring.

**Basic Bootstrapping**

BlazeDS is a Java Servlet–based web application, so it will integrate and work with the Spring framework facility that addresses the web layer. Spring framework includes an MVC framework for creating web applications. The MVC framework in Spring is often simply referred to as the Spring MVC. Spring MVC, like every other part of the Spring framework, leverages dependency injection and Aspect-Oriented Programming.

Spring MVC, like most MVC frameworks, provides a separation of concerns along the three important sets of entities, namely models, views, and controllers. Models represent data and state, views provide the interface for user interaction and display, and controllers act as the intermediary
between the models and the views. Controllers facilitate model updates based on user input. Spring MVC implements a number of well-known design patterns, including the famous Front Controller pattern.

The Front Controller pattern is based on a central controller intercepting all requests and delegating these requests to appropriate handlers. The DispatcherServlet class serves the purpose of a front controller in Spring MVC. It delegates requests to other controllers. Views are created from responses, and they are served back to the client.

DispatcherServlet extends the HttpServlet. As with any HttpServlet, you can map all requests that match a particular URL pattern to be handled by the DispatcherServlet. Using standard Java EE semantics, you can declare the Servlet and the URL mapping in web.xml as follows:

```xml
<web-app>
  <servlet>
    <servlet-name>sample</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>sample</servlet-name>
    <url-pattern>*/update/*</url-pattern>
  </servlet-mapping>
</web-app>
```

In this case, all URLs containing the word “update” are handled by the DispatcherServlet. Each DispatcherServlet has an associated WebApplicationContext. Each of these WebApplicationContext instances inherits all the beans defined in the root WebApplicationContext. These global bean definitions can be overridden, and newer beans can be defined for the particular Servlet scope.

To define beans specific to a WebApplicationContext, specify the beans in a configuration file, named <servlet-name>-servlet.xml. This configuration file, by default, would reside in the WEB-INF folder. In the earlier code snippet a Servlet was called sample, so the associated configuration for the Spring MVC components would be named sample-servlet.xml.

Next, let’s explore how the BlazeDS MessageBroker can be used with the Spring DispatcherServlet.

**Using BlazeDS MessageBroker with the Spring DispatcherServlet**

A MessageBroker component sits at the heart of BlazeDS. All messages in BlazeDS are routed through the MessageBroker. In standard BlazeDS deployments, the MessageBrokerServlet is declared in WEB-INF/web.xml as follows:

```xml
<servlet>
  <servlet-name>MessageBrokerServlet</servlet-name>
  <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
</servlet>
```
<init-param>
    <param-name>services.configuration.file</param-name>
    <param-value>/WEB-INF/flex/services-config.xml</param-value>
</init-param>

<init-param>
    <param-name>flex.write.path</param-name>
    <param-value>/WEB-INF/flex</param-value>
</init-param>

<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>MessageBrokerServlet</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>

This configuration implies the following:

- A MessageBrokerServlet instance is declared, with WEB-INF/flex/services-config.xml passed in as the configuration file to this Servlet.
- The flex.messaging.MessageBrokerServlet instance is identified as MessageBrokerServlet.
- The MessageBrokerServlet loads when the application starts up.
- All requests to resources whose URL includes the /messagebroker/* pattern are handled by the Servlet identified as MessageBrokerServlet.

If you are familiar with the Java Servlet technology, all of this would be very familiar to you.

In Spring MVC, all incoming requests are handled by the DispatcherServlet. This means you must have a way that all requests coming from the Flex client and intended to be handled by the BlazeDS instance are routed by the DispatcherServlet to the MessageBroker. The Spring BlazeDS project, which I will call “Spring BlazeDS” from now on, addresses this issue and lets you route messages to the MessageBroker from the DispatcherServlet. Therefore, you do not need to configure the MessageBrokerServlet when using Spring BlazeDS.

With Spring BlazeDS, you first need to declare the DispatcherServlet in WEB-INF/web.xml as follows:

<servlet>
    <servlet-name>springblazedstest</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/config/web-application-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

This declares an instance of the DispatcherServlet and defines WEB-INF/config/web-application-config.xml as its configuration file, which is passed as an initialization parameter.
Chapter 5: Accessing Spring Beans

Next, I will configure BlazeDS using the Spring bean configuration semantics. In order to use Spring style configuration, I need to first include the namespace that defines the Spring BlazeDS XML schema. To define beans specific to the DispatcherServlet, which I called springblazedstest in the preceding code snippet, I first create a file called springblazedstest-servlet.xml in the WEB-INF folder. You may recall a discussion on this configuration file-naming convention from an earlier part of this section.

In its simplest form, the MessageBroker can be configured using Spring bean semantics in springblazedstest-servlet.xml as follows:

```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/>
</beans>
```

The element `<flex:message-broker/>` uses many default settings, including a default path to the services configuration file. The default path to the BlazeDS configuration file is WEB-INF/flex/services-config.xml, which is normally the case in standard BlazeDS deployments. You can explicitly define an alternative path using the services-config-path attribute of the message-broker element. The configuration would then become:

```xml
<flex:message-broker services-config-path="/WEB-INF/flex/services-config.xml"/>
```

The message-broker element is defined in spring-flex-1.0.xsd. This XML schema is available online at www.springframework.org/schema/flex/spring-flex-1.0.xsd. It is also bundled locally in the distribution jar, that is, org.springframework.flex-1.0.0.RELEASE, which is available in the dist folder of the spring-flex-1.0.0.RELEASE download. The spring-flex download includes the source for the spring BlazeDS project, so you could easily access the XSD directly from the projects\org.springframework.flex\src\main\java\org\springframework\flex\config\xml folder of the download as well. The XSD has all the element and attribute definitions for the Spring BlazeDS project.

Behind the scenes, a Spring MessageBrokerFactoryBean is configured, which allows the MessageBroker to be instantiated and initialized using the Spring bean configuration. This implies that you could alternatively configure the MessageBroker as follows:

```xml
<bean id="_messageBroker"
class="org.springframework.flex.core.MessageBrokerFactoryBean">
  <property name="servicesConfigPath" value="classpath*:services-config.xml"/>
</bean>
```

instead of using the terse `<flex:message-broker/>` element in your configuration file. You may have noticed that an expression classpath*:services-config.xml was provided as the path to the
BlazeDS configuration file. This expression states that the services-config.xml should be picked up from the Spring application classpath. MessageBrokerFactoryBean uses Spring’s ResourceLoader and that is what makes it possible to specify expressions of the types shown in the configuration file path.

You would like all requests coming from the Flex client to the BlazeDS resources to be handled by the MessageBroker. In order for this to happen, you should create a mapping in the springblazedstest-servlet.xml as follows:

```xml
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <value>/*=_messageBroker</value>
  </property>
</bean>
```

This installs a SimpleUrlHandlerMapping that directs all incoming requests that come to the DispatcherServlet to go to the MessageBroker. The assumption is that the Flex client is the only one communicating with the Spring DispatcherServlet.

Besides the URL handler mapping, you should define a MessageBrokerHandlerAdapter, which the SimpleUrlHandlerMapping uses to map of all incoming requests to the MessageBroker. A MessageBrokerHandlerAdapter can be configured as follows:

```xml
<bean class="org.springframework.flex.servlet.MessageBrokerHandlerAdapter"/>
```

While the handler mapping and the handler adapter need to be configured to get the MessageBroker to handle all the requests from the Flex client to the BlazeDS resources, your work is extremely simplified because you actually don’t need to manually configure these artifacts. The inclusion of a single line <flex:message-broker/> with the appropriate namespace declarations does all this for you.

### Serving Flex and Non-Flex Clients

It’s possible that on occasion your Spring server side may be serving clients beyond a single Flex application. For example, it could be serving a Flex client and a JavaScript-based client. In such situations, you may want to configure multiple instances of the DispatcherServlet, one for the Flex client and one for the JavaScript client. Further, you may want the core application infrastructure and the cross-cutting elements to be configured in your main application layer. A possible configuration for such architecture is:

```xml
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/*-context.xml</param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<servlet>
  <servlet-name>flexclient</servlet-name>
```

117
In this case, bean definitions specific to the Flex client and the JavaScript client are defined in the `flexclient-servlet.xml` and `javascriptclient-servlet.xml` configuration files, respectively.

Sometimes there is a need to create multiple client bean artifacts within a single WebApplicationContext and, therefore, have a single DispatcherServlet. In such cases, you may need to create a hierarchy within the URL patterns. So, URL patterns conforming to `/spring/messagebroker/*` would be directed to the BlazeDS MessageBroker, and URL patterns conforming to `/spring/jsbroker/*` would be directed to a JavaScript request broker, which is outside of scope of BlazeDS.

A few additional changes will be required to make such a hierarchical URL pattern work with BlazeDS. First, you will need to configure the mapping attribute in the `message-broker` element as follows:

```xml
<flex:message-broker>
    <flex:mapping pattern="/messagebroker/**" />
</flex:message-broker>
```

Second, you would need to modify the endpoint URL from `http://{server.name}:{server.port}/(context.root)/messagebroker/amf` to `http://{server.name}:{server.port}/(context.root)/spring/messagebroker/amf`. This is needed because you want everything with the URL pattern `/spring/messagebroker/*` to be handled by the BlazeDS MessageBroker in the hierarchical URL architecture.

A MessageBroker can be customized further, but I will skip the advanced configuration for now. In the next section, I use Spring beans as Flex remoting destinations.
Remoting to Spring Beans

Existing and new Spring beans can be exposed as Flex remoting service destinations. Prior to the availability of the Spring BlazeDS project, it was common to write a custom factory to get access to the Spring beans. Spring beans are managed objects and in standard BlazeDS instantiating the beans and maintaining their lifecycle had to be delegated to an external Spring framework container. With Spring BlazeDS, using Spring beans as remoting destinations is considerably simplified.

Let’s take a simple example to explain things a bit further. I create a simple Spring bean, which I call `BookService`. The `BookService` allows me to perform the create, read, update, and delete (CRUD) operations on a catalog of books. Each book is identified by four simple attributes:

- **Id** (a catalog specific identifier)
- **Title**
- **Author** (I make an assumption that each book in the simple example has only one author)
- **Price**

In a real-life book catalog, you are likely to find other attributes like the name of the publisher, the year of publication, and the two types of ISBN numbers for a book. To keep things simple and to keep the focus on the core aspects of remoting Spring beans to Flex, all these additional attributes are not included in the simple example.

In the simple example, a `Book` could be modeled as a POJO, which could be as follows:

```java
package problazeds.ch05;
import java.io.Serializable;

public class Book implements Serializable {
    static final long serialVersionUID = 103844514947365244L;

    private int bookId;
    private String title;
    private String author;
    private double price;

    public Book() {}

    public int getBookId() {
        return bookId;
    }

    public void setBookId(int bookId)
```
Next, a BookService could be created to perform CRUD operations on the collection of Book objects in the books catalog. Using a clean design, it may be prudent to create an IBookService interface to define the behavior of the BookService. That way, there could be alternative implementations to support the same contract. The IBookService interface is shown in Listing 5-1.

### Listing 5-1: IBookService Interface

```java
cpyackage problazeds.ch05;

import java.util.List;

public interface IBookService {
    public List<Book> findAll();
    public Book findById(int id);
    public String getTitle()
    { return title; }
    public void setTitle(String title)
    { this.title = title; }
    public String getAuthor()
    { return author; }
    public void setAuthor(String author)
    { this.author = author; }
    public double getPrice()
    { return price; }
    public void setPrice(double price)
    { this.price = price; }
}
```
public Book create(Book item);
public boolean update(Book item);
public boolean remove(Book item);
}

In traditional BlazeDS implementations that do not involve Spring BlazeDS, you would configure the BookService as a remoting service destination in the remoting-config.xml file. The remoting-config.xml file contains all definitions related to the remoting service in BlazeDS. This XML file is included in services-config.xml by reference. The configuration for the BookService remoting destination in remoting-config.xml is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service id="remoting-service" class="flex.messaging.services.RemotingService">
    <adapters>
        <adapter-definition id="java-object" class="flex.messaging.services.remoting.adapters.JavaAdapter" default="true"/>
    </adapters>
    <default-channels>
        <channel ref="my-amf"/>
    </default-channels>
    <destination id="bookService">
        <properties>
            <source>problazeds.ch05.BookService</source>
            <scope>session</scope>
        </properties>
        <adapter ref="java-object"/>
    </destination>
</service>
```

Using Spring BlazeDS, the BookService Spring bean is first configured either in the DispatcherServlet configuration or the services context configuration as follows:

```xml
<bean id="bookService" class="problazeds.ch05.BookService"/>
```

Following this, the BookService can be exposed as a remoting service destination by configuring it as follows:

```xml
<flex:remoting-destination ref="bookService"/>
```

Once again convention is preferred over configuration. Following a standard convention and going with defaults is more effective and efficient than configuring every single configurable element. A lot of configuration is usually standard boilerplate and one easily avoids extra work if convention is followed. The Java adapter class and the AMF channel are configured by default.
You can also specify the adapter and channel defaults, like so:

```xml
<flex:message-broker>
  <flex:remoting-service
    default-adapter-id="my-alternative-default-remoting-adapter"
    default-channels="my-amf, my-secure-amf" />
</flex:message-broker>
```

In the remoting-destination configuration, you can also specify the list of methods to include and exclude. If you desire to allow read-only access to the books catalog, you can configure the remoting destination as follows:

```xml
<flex:remoting-destination
  ref="bookService"
  include-methods="findAll, findById"
  exclude-methods="create, update, remove" />
```

In addition to the advanced configuration, you also have the choice of specifying the remoting destination characteristics as an inline property. Therefore, the last remoting-destination configuration can be combined with the bean configuration, like so:

```xml
<bean id="bookService" class="problazeds.ch05.BookService">
  <flex:remoting-destination
    ref="bookService"
    include-methods="findAll, findById"
    exclude-methods="create, update, remove" />
</bean>
```

At this stage, you may have realized that with Spring BlazeDS you do not need to configure remoting destinations in `remoting-config.xml`.

The Spring style remoting-destination configuration is easier, intuitive, and terse. The remoting-destination configuration is supported under the hood by the `org.springframework.flex.remoting.RemotingDestinationExporter` factory class that creates the Spring managed remoting destinations for a Flex client.

Your last configuration using `RemotingDestinationExporter` explicitly is:

```xml
<bean id="book" class="org.springframework.flex.remoting.RemotingDestinationExporter">
  <property name="messageBroker" ref="_messageBroker" />
  <property name="service" ref="bookService" />
  <property name="destinationId" value="bookService" />
  <property name="includeMethods" value="findAll, findById" />
  <property name="excludeMethods" value="create, update, remove" />
</bean>
```

So far, all configurations have been done using XML. You could also use Java annotations to achieve the same result. The `BookService` class with annotations for exposing it as a BlazeDS remoting destination is shown in Listing 5-2.
BlazeDS provides two important services, namely remoting and messaging. This section explained how you could use a Spring bean as a remoting destination. The next section will explain how BlazeDS messaging could work with the Spring messaging services.
Chapter 5: Accessing Spring Beans

Message Service Integration

A Flex client can leverage two message service components to send and receive messages. These are the Producer and Consumer, respectively. These components can be used with Spring BlazeDS in the same manner as they can be used in standard BlazeDS installations. On the server side, though, Spring BlazeDS offers a few value-added extensions.

In a Spring BlazeDS server, three types of message service components can interact with the Flex message service. The messaging service in Flex itself is agnostic to the messaging protocol used on the server side. Therefore, multiple server-side messaging alternatives easily work with Flex messaging. The three alternative server-side message services in Spring BlazeDS are:

- Native built-in BlazeDS AMF messaging
- JMS messaging using the Spring-specific JMS components
- Spring BlazeDS messaging integration

Let’s explore each in turn.

Native BlazeDS AMF Messaging

BlazeDS supports an ActionScriptAdapter for simple message routing between two or more Flex clients. The communication between the two or more Flex clients happens via a server-side destination. This messaging adapter is limited in its functionality and does not support transactions. The core of the messaging adapter is the two methods that help route messages. These two methods are:

- pushMessageToClients — Delivers message to all clients connected to a server
- sendPushMessageFromPeer — Delivers messages to peer servers in the cluster, which in turn deliver the message to connected clients

To use this adapter, you will typically configure a server-side destination and assign the ActionScriptAdapter as the adapter to use. In addition, you would also configure a messaging channel, probably a polling AMF channel. In standard BlazeDS such configuration resides in the messaging-config.xml configuration file. The messaging-config.xml file is included in the services-config.xml file by reference. The configuration would be something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service id="messaging-service" class="flex.messaging.services.MessageService">
    <adapters>
        <adapter-definition id="actionscript" class="flex.messaging.services.messaging.adapters.ActionScriptAdapter" default="true"/>
    </adapters>
    <default-channels>
        <channel ref="my-polling-amf"/>
    </default-channels>

    <destination id="myTopic">
        <properties>
The configuration does not explicitly assign the ActionScriptAdapter as the adapter for the myTopic messaging destination. However, the ActionScriptAdapter is the default messaging adapter, so it is used. In Spring BlazeDS much of this configuration is reduced to only a few lines of configuration as follows:

```xml
<flex:message-destination
  id="myTopic"
  message-broker="messageServiceBroker"
  channels="my-polling-amf, my-secure-amf"
  subscription-timeout-minutes="0"
  allow-subtopics="true"
  message-time-to-live="0"
  subtopic-separator="." />
```

In fact, just `<flex:message-destination id="myTopic"/>` is enough if you are using the default adapter and the default channel. By default Spring configures `flex.messaging.services.messaging.adapters.ActionScriptAdapter` as the default messaging adapter and `my-polling-amf`, which I assume is already configured, as the default channel. You can also define alternative defaults by explicitly specifying values as follows:

```xml
<flex:message-broker>
  <flex:message-service
    default-adapter-id="my-alternative-default-messaging-adapter"
    default-channels="my-polling-amf" />
</flex:message-broker>
```

As with remoting and as you may have guessed by now, you can skip configuring the `messaging-config.xml` when using Spring BlazeDS. Instead, you can leverage the `spring-flex-1.0.0.xsd` elements and attributes to configure messaging destinations. Such configuration could be handled completely in the Spring-related configuration files.
Although the ActionScriptAdapter is very handy, you may more likely use a more robust adapter, such as the JMSAdapter, when integrating with Java server-side messaging systems.

Next, the Spring BlazeDS JMSAdapter is discussed.

### JMS Messaging Using Spring-Specific JMS Components

JMS is a standard messaging API in Java EE. To send and receive messages using JMS, you first need to have a JMS provider available. The JMS provider makes it possible to create connection factories and destinations, queues, and topics. Once you have the provider and its managed objects, connection factory, and destinations set up and configured, you need to create a connection to the connection factory and open a session on the connection.

Within a JMS session, you send and receive messages with a message producer and a message consumer. Once the messages are sent and received, you close the session and the connection. During a JMS interaction a JMSException can be thrown, which needs to be handled.

The Spring framework defines a couple of components to interact with JMS message domains: queues and topics. It includes a template-based solution, JmsTemplate, to send and receive JMS messages. The JmsTemplate eliminates much of the boilerplate code associated with opening and closing connections and sessions and sending and receiving messages. It also transforms the JMSException into part of Spring’s runtime exception hierarchy. JMSException is converted to org.springframework.jms.JmsException. Spring simplifies access to and programming with many resources such as JDBC, JNDI, and JMS by defining templates that help eliminate much of the boilerplate code and let you focus on core tasks.

To show the JmsTemplate in use, I assume that you have a JMS topic called quotes on which Flex and Java JMS consumers and producers interact. Let’s also assume that the messages sent to the topic are about a product, which is defined by its ID, name, description, and price. So, a Java JMS client, leveraging the Spring JmsTemplate, could send messages to quotes using JavaJmsClient, which would be done as follows:

```java
package problazeds.ch05

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class JavaJmsClient {
    private JmsTemplate jmsTemplate;
    private Destination destination;
    
    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
```
public void setDestination(Destination destination) {
    this.destination = destination;
}

public void sendMail(final Product product) {
    jmsTemplate.send(destination, new MessageCreator() {
        public Message createMessage(Session session) throws JMSException {
            MapMessage message = session.createMapMessage();
            message.setString("productId", product.getProductId());
            message.setString("name", product.getName());
            message.setString("description", product.getDescription());
            message.setDouble("price", product.getPrice());
            return message;
        }
    });
}

The message itself can be sent using a MessagingAgent, which would be as follows:

```java
package problazed.ch05;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MessagingAgent {
    public static void main(String[] args) {
        ApplicationContext context =
        new ClassPathXmlApplicationContext("beans-front.xml");

        JavaJmsClient javaJmsClient =
        (JavaJmsClient) context.getBean("javaJmsClient");
        javaJmsClient.sendMail(new Product("1", "Professional BlazeDS", "A book on Java
        and Flex integration via BlazeDS", "39.99"));
    }
}
```

The assumption is that you have wired up these classes and a JMS provider in the Spring configuration files. You could use ActiveMQ, and then your configuration would be as follows:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
      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">
  <bean id="connectionFactory"
       class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>

  <bean id="quotes"
       class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="product.topic" />
  </bean>

  <bean id="jmsTemplate"
```
Chapter 5: Accessing Spring Beans

```xml
<bean class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="javaJmsClient" class="problazeds.ch05.JavaJmsClient">
  <property name="destination" ref="quotes" />
  <property name="jmsTemplate" ref="jmsTemplate" />
</bean>
</beans>
```

If you wanted the JavaJmsClient to act as a message listener as well, then you would need to add a method as follows:

```java
public Mail receiveMail() {
    MapMessage message = (MapMessage) jmsTemplate.receive(destination);
    try {
        if (message == null) {
            return null;
        }
        Product product = new Product();
        product.setProductId(message.getProductId());
        product.setName(message.getName());
        product.setDescription(message.getDescription());
        product.setPrice(message.getPrice());
        return product;
    } catch (JMSException e) {
        throw JmsUtils.convertJmsAccessException(e);
    }
}
```

The code above is just a small snippet that shows how using the Spring JmsTemplate reduces a whole lot of boilerplate code and simplifies sending and receiving messages using JMS. The assumption I make here is that you are familiar with standard JMS programming idioms. You can learn more about JMS at http://java.sun.com/javaee/5/docs/tutorial/doc/bncdq.html.

This chapter is focused on the Spring BlazeDS integration and so does not provide the level of detail on JmsTemplate that a newbie may desire. You will benefit from reading the Spring documentation on JmsTemplate at http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/ch23.html.

While a JmsTemplate simplifies sending and receiving messages, receiving messages in the way just described is synchronous. Synchronous message listeners are not efficient, as they involve blocking operations. Resources like threads wait until a message arrives. In asynchronous operations such threads could gainfully carry out other operations in this waiting time.

To address this situation (where synchronous messaging listeners blocked resources), MessageDriveBeans (MDBs) were introduced into Java EE. EJB containers could then listen to JMS destinations and trigger an MDB when a message arrived. The Spring framework extended this concept to POJOs and, therefore, makes it easy to have asynchronous message listeners without the need for EJB containers. Spring’s counterparts of a MDB are a Message Driven POJO (or MDP). You can also write an asynchronous JMS
message listener without Spring, although MDPs are a robust set of asynchronous message listeners that you could readily use.

As a further value addition, Spring defines built-in message listener containers for MDP that cache JMS resources and provide for XA transactional support. There are three such containers, namely:

- SimpleMessageListenerContainer
- DefaultMessageListenerContainer
- ServerSessionMessageListenerContainer

Spring BlazeDS brings all the Spring JMS components together. It defines a new JMSAdapter, org.springframework.flex.messaging.jms.JmsAdapter, which behind the scenes makes use of the JmsTemplate and the DefaultMessageListenerContainer.

A JMS destination, such as quotes, can be exposed as a BlazeDS messaging destination using Spring BlazeDS with the following configuration:

```
<flex:jms-message-destination id="aJMSMessageDestination"
    jms-destination="myJMSTopic" />
```

The destination configuration shown above gets included in the Spring message broker servlet configuration file, usually called <servlet-name>-servlet.xml.

The configuration for the ActiveMQTopic and connectionFactory already exist:

```
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="quotes"
    class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="product.topic" />
</bean>
```

The jms-message-destination element is defined by Spring-flex-1.0.0.xsd. Strings passed into the jms-destination attribute are mapped to the actual JMS destination with the help of Spring’s DestinationResolver class.

Next, I will explore the special IntegrationAdapter defined by the Spring BlazeDS project.

**Using the Special Integration Adapter**

Spring BlazeDS defines org.springframework.flex.messaging.integration.IntegrationAdapter, which enables sending and receiving of messages via Spring BlazeDS’s message channels. Complicated message routing could be established using this adapter. You will also be able to connect to FTP and email endpoints using the integration adapter.
Chapter 5: Accessing Spring Beans

As an example, a Spring extension project, called SESIA, provides an FTP adapter that works with the integration adapter. You can get more information on the SESIA project online at http://opensource.atlassian.com/confluence/spring/display/INTEGRATE/SESIA.

To read files from a remote FTP destination, you could use the FtpFileSource class of the FTP adapter. FtpFileSource uses FtpInboundSynchronizer to access files from a remote FTP destination and make them available in a local directory. The FtpInboundSynchronizer class needs an FtpClientPool to communicate.

An FtpClientPool can be configured as follows:

```xml
<beans:bean id="ftpClientPool"
    class="org.springframework.integration.ftp.QueuedFTPClientPool">
    <beans:constructor-arg>
        <beans:bean
            class="org.springframework.integration.ftp.DefaultFTPClientFactory"/>
    </beans:constructor-arg>
</beans:bean>
```

Once an FtpClientPool is configured it can be associated, along with other needed configurations, with an FtpFileSource as follows:

```xml
<beans:bean id="ftpFileSource"
    class="org.springframework.integration.ftp.FtpFileSource">
    <beans:property name="clientPool" ref="ftpClientPool" />  
    <beans:property name="localWorkingDirectory" value="file:${java.io.tmpdir}/FtpConfigIntegrationTests" />
    <beans:property name="taskScheduler" ref="taskScheduler" />
    <beans:property name="trigger">
        <beans:bean
            class="org.springframework.integration.scheduling.IntervalTrigger">
            <beans:constructor-arg value="1500" />
        </beans:bean>
    </beans:property>
</beans:bean>
```

Once the FtpFileSource is configured and ready you just need to configure an IntegrationAdapter instance to make use of this FTP adapter. Such an integration adapter can be configured as follows:

```xml
<integration:inbound-channel-adapter
    channel="file-channel" ref="ftpFileSource" />
```

With this example of an FTP adapter, all the three types of adapters supported by the Spring BlazeDS project — namely BlazeDS native AMF adapter, Spring JMS specific adapter, and Spring BlazeDS integration adapter — are covered.

Before we move beyond messaging though, it may be worthwhile to mention a few things about the MessageTemplate that provides a convenient way to send messages to BlazeDS destinations. You could configure a MessageTemplate instance as follows:

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

130
Chapter 5: Accessing Spring Beans

Once configured you could leverage this message template to send AMF messages to a JMS destination. Messages to *quotes* can be sent using the message template’s *send* method, passing the destination name as the first parameter and the message as the second parameter.

With a discussion on remoting and messaging, most of the core features are already covered. In addition to these core features, Spring BlazeDS provides a way to leverage all the other features of the Spring framework. One such robust feature is Spring security, which we discuss next.

**Leveraging Spring Security**

Spring has a very flexible and robust set of framework components that help implement sophisticated security profiles and constraints, including the most commonly used authentication and authorization patterns. In standard BlazeDS, security is typically delegated to the underlying container in which BlazeDS is deployed.

Spring BlazeDS provides an alternative mechanism for implementing security other than that provided by standard BlazeDS. If Spring is your primary server-side framework, leveraging Spring security with your Flex and Spring application is a good idea.

In the simplest case, you can define a role-based login using in-memory definitions. A configuration for such a simple in-memory definition is:

```xml
<authentication-provider>
  <user-service>
    <user name="admin" password="secret" authorities="ROLE_ADMIN,ROLE_USER" />
    <user name="aUser" password="somePassword" authorities="ROLE_USER" />
    <user name="anotherUser" password="someOtherPassword" disabled="true" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
```

This configuration could be better organized by externalizing the user-service content to a file, say `users.properties`, that could be located in the *WEB-INF/config* folder. The configuration would then be as follows:

```xml
<authentication-provider>
  <user-service properties="/WEB-INF/config/users.properties" />
</authentication-provider>
```

Alternatively and more likely in most enterprise applications, the security will be more complicated than a few entries inline or in a file. Also, you may want to restrict access and thereby avoid having any credentials defined in files that are deployed with the application. A possible alternative could be keeping security profiles in the database.

Spring has built-in support for database-based authentication. Spring relies on the following default SQL queries to access the security definition:

```sql
SELECT username, password, enabled
FROM users
WHERE username = ?
```
If you choose to use database-based authentication, with the default queries as specified previously, do not forget to create the users and authorities tables using the following Data Definition Language (DDL) statements:

```
CREATE TABLE USERS (
    USERNAME VARCHAR(10) NOT NULL,
    PASSWORD VARCHAR(32) NOT NULL,
    ENABLED SMALLINT,
    PRIMARY KEY (USERNAME)
);

CREATE TABLE AUTHORITIES (
    USERNAME VARCHAR(10) NOT NULL,
    AUTHORITY VARCHAR(10) NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

To use database-driven authentication in Spring BlazeDS, first configure a data source to connect to your database and then reference that data source as the value of the `data-source` attribute of the `jdbc-user-service` element. The `jdbc-user-service` is the `authentication-provider` in this case. A configuration for the data source and then the authentication provider are as follows:

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
        value="<Specify the JDBC driver class name here>" />
    <property name="url"
        value="<Specify the database access URL here>" />
    <property name="username" value="<Specify the user name here>" />
    <property name="password" value="<Specify the password here>" />
</bean>

<authentication-provider>
    <jdbc-user-service data-source-ref="dataSource" />
</authentication-provider>
```

To configure an authentication provider with Spring BlazeDS, also configure an HTTP entry point reference as follows:

```
<http entry-point-ref="preAuthenticatedEntryPoint" />
<beans:bean id="preAuthenticatedEntryPoint"
    class="org.springframework.security.ui.preauth.PreAuthenticatedProcessingFilterEntryPoint" />
```

After this preliminary initial configuration, you are ready to configure security for the BlazeDS MessageBroker. The simplest such definition is:

```
<flex:message-broker>
    <flex:secured />
</flex:message-broker>
```
Like other configurations you have seen so far, a single line, `<flex:secured>`, achieves quite a bit. Again, convention is preferred over configuration and defaults are relied upon. BlazeDS implements authentication using the LoginCommand. Spring BlazeDS with the minimal configuration includes a custom LoginCommand implementation that combines ChannelSet.login and ChannelSet.logout with the Spring security definitions. In addition, it also makes the security definitions available within the Spring BlazeDS configuration elements defined by Spring-flex-1.0.0.xsd.

Standard security constraints can be defined within BlazeDS destinations as usual. When they are configured with Spring security, the underlying Spring BlazeDS LoginCommand is used for authentication.

While this style of authentication is easy to use, it provides less flexibility in terms of access to the response data from the authentication requests. Sometimes, though, programmatic access to authentication data can be used to drive role-based access control within the Flex application. In such cases, you may want to explicitly call the ChannelSet.login method and pass in the login credentials. Such a method is asynchronous, like most methods in Flex. You can set a responder for this call. On response, you can match the user credentials from the authorities and take appropriate action.

Next, I will cover a few examples that illustrate various ways of securing the Spring BlazeDS remoting destination.

**Securing Remote Spring Bean Access**

There are multiple ways to secure access to Spring BlazeDS destinations. Let’s revisit the BookService example from the section on remoting to Spring beans. In that example, the BookService had five methods to support CRUD operations on a books catalog. The five methods were:

- findAll
- findById
- create
- update
- remove

If you had to restrict access to these methods based on the logged-in user’s role, you could set up a security constraint as follows:

```xml
<bean id="bookService" class="problazeds.ch05.BookService">
  <flex:remoting-destination/>
  <security:intercept-methods>
    <security:protect method="find*" access="ROLE_USER" />
    <security:protect method="create" access="ROLE_ADMIN,ROLE_USER" />
    <security:protect method="update" access="ROLE_ADMIN" />
    <security:protect method="remove" access="ROLE_ADMIN" />
  </security:intercept-methods>
</bean>
```

It’s also possible to define such security constraints using annotations instead of XML. As an example, you could secure the remove method and only allow access to ROLE_ADMIN as follows:

```java
@Secured("ROLE_ADMIN")
public boolean remove(Book item);
```
However, if you wanted to instead provide access to all methods of all classes, defined within the `problazeds.ch05` package, then you could rely on a configuration like this:

```xml
<global-method-security>
  <protect-pointcut
    expression="execution(* problazeds.ch05.*(..))"
    access="ROLE_USER"/>
</global-method-security>
```

Spring security is very flexible and is handy for multiple types of security definition implementations. As a case in point, instead of defining security for classes or their methods, you may have a situation where you want to restrict access by channel type. For example, all your remoting destinations may use the `my-amf` channel, and all your messaging destinations may use the `my-polling-amf` channel. Now you may allow users who are logged in as `ROLE_USER` to access all destinations on `my-amf` but may restrict access to all `my-polling-amf` to only those users who are logged in as `ROLE_ADMIN`. One such case may be that the near real-time updates are sending monitoring data that only users with `ROLE_ADMIN` need to use.

Such a security configuration is:

```xml
<flex:message-broker>
  <flex:secured>
    <flex:secured-channel channel="my-amf" access="ROLE_USER" />
    <flex:secured-channel channel="my-polling-amf" access="ROLE_ADMIN" />
  </flex:secured>
</flex:message-broker>
```

You could also set up Spring security for all channels whose URL matches a particular pattern. This could be configured like so:

```xml
<flex:message-broker>
  <flex:secured>
    <flex:secured-endpoint-path
      pattern="**/messagebroker/amf**"
      access="ROLE_USER" />
  </flex:secured>
</flex:message-broker>
```

So far I have covered a few examples where Spring security provides an extensible and flexible way of defining security. Spring security itself has many more features, but they are not discussed in this chapter. You may benefit from reading more on the Spring framework to better understand its features.

**Using Custom Adapters**

In Chapters 7 and 9, you will learn about custom adapters. In this brief section, I will illustrate an example that shows how you can configure a custom adapter.
Gilead, a custom Hibernate adapter that you will learn about in Chapter 7, can be configured in remoting-config.xml like so:

```xml
<adapters>
  <adapter-definition id="persistent-adapter"
    class="net.sf.gilead.blazeds.adapter.PersistentAdapter">
    <properties>
      <persistence-factory>
        <class>net.sf.gilead.sample.server.ApplicationContext</class>
        <singleton>true</singleton>
        <method>getSessionFactory</method>
      </persistence-factory>
    </properties>
  </adapter-definition>
</adapters>
```

Using the Spring BlazeDS adapter JSON format is used to configure parameters on a custom adapter. The preceding configuration under Spring BlazeDS would be:

```xml
<bean id="persistent-adapter"
  class="org.springframework.flex.core.ManageableComponentFactoryBean">
  <constructor-arg value="net.sf.gilead.blazeds.adapter.PersistentAdapter"/>
  <property name="properties">
    <value>
      { "persistenceFactory" :
        { "class" :
          "net.sf.gilead.sample.server.ApplicationContext",
          "singleton" : "true",
          "method": "getSessionFactory" }
        }
    </value>
  </property>
</bean>
```

The Spring BlazeDS configuration for custom adapters configures these adapters using the org.springframework.flex.core.ManageableComponentFactoryBean.

**Summary**

This chapter should ideally be titled “Integration between Spring and Flex” because that is what the chapter covers. The chapter introduces the latest release 1.0.0 of the Spring BlazeDS project. Through examples, it explains how usage patterns and configurations apply to projects that leverage Spring BlazeDS. For further detailed examples please explore the BlazeDS examples ported over to use Spring BlazeDS. The ported over examples are bundled with the Spring BlazeDS distribution.

The chapter starts with the basics of setting up Spring BlazeDS for an application. It discusses essential configuration and shows how defaults and conventions are preferred over configuration. The first section
Chapter 5: Accessing Spring Beans

also sets the tone for the chapter, emphasizing better integration and productivity enhancements with minimal configuration.

After the basics, the chapter covers remoting and messaging under Spring BlazeDS. Remoting explores exporting of Spring beans as remoting destinations. The section on messaging explains the different use cases to which the message service applies. It explains JMS messaging with Spring BlazeDS with some level of detail. The section also speaks about the special IntegrationAdapter that Spring BlazeDS introduces. The IntegrationAdapter is used to connect to a remote FTP service from a Flex application.

The following part of the chapter focuses on Spring security and its applicability on projects that leverage Spring BlazeDS. Toward the end, custom adapter configuration is discussed.
Communicating in Real-Time via Messages

Over the history of web applications, we have become comfortable with viewing application interactions as a series of requests and responses. Users and their agents, which are often called clients, pull data from servers or ask the server to carry out a set of actions; the server, in turn, manipulates and modifies data. Web applications, until recently, hardly ever involved data pushing, initiated by a server. Such interactions remained in the realm of the server-centric applications and the desktop.

Part of the reason behind the request-response model is the stateless nature of HTTP. In its plain vanilla form, HTTP works fine for static resource access. However, the statelessness gets tricky with applications. Many applications require that the system remember who the client is so that requests subsequent to the first one can be associated with the original request. Keeping this continuity avoids repeated authentication, helps maintain transactional integrity, and implements a conversation between a user and a system. From the very first few such requirements, developers figured out hacks for maintaining such state across multiple requests and created the concept of a session, which has become part of the standard HTTP repertoire.

Once stateful web applications were possible, the next challenge was to break the synchronous nature of the request-response model. The XMLHttpRequest object and Ajax were the frontrunners in this area. They set the tone for asynchronous processing on the web. Full-blown RIA frameworks, such as Flex, took this to the next step, where every important interaction is asynchronous.

However, server-initiated data pushing was still fairly alien until a couple of years back. Things are different now. Data pushing from a server to a browser-based application is not only possible but a viable option for creating serious applications. The initial attempts to push data based on continuous polling have given way to more sophisticated and robust alternatives. In this chapter, you will learn to push data up to Flex clients and understand the intricacies involved in making that happen.
Data Push Essentials

Data pushing in a web application context implies the server sending data to a browser-based client without the client necessarily requesting it. There are ways to achieve this partially and fully, but before we explore those, let’s gain an understanding of where data pushing is important. Understanding the usage scenarios will help you choose the correct alternative and highlight the potential upsides and downsides of these alternatives.

Data Push Usage Scenarios

The financial market data continuously changes as buyers and sellers present their bid and offer prices and engage in trading. If you are building an investment management application or simply an application for displaying market data, then you want to receive the latest data snapshot as soon as it’s available. In such cases, server-initiated data updates are suitable.

Online collaboration and chat applications benefit from real-time instant updates. Message passing between the interacting clients is appropriate in this case. Monitoring applications could be more effective if the object being monitored sent a status message to the monitor at periodic intervals to report its current state. Online gaming applications need to instantly relay a player’s choices to all other participants for the game to be consistent and relevant.

In all these cases and more data pushing provides the basis for the application’s usability and pertinence. Next, you’ll see the ways to push data.

The Alternatives

The simplest improvement to a regular request-response method, which mimics data pushing, is polling. Polling uses multiple request-response interactions at a regular predefined interval. Such periodic and recurring calls are usually driven by a timer and run over long durations. Polling is resource-intensive and can often be ineffective because multiple calls could end with no useful results. In order to minimize this overhead, the frequency of these calls can be reduced. However, decreasing the frequency could lead to larger time intervals between requests and increased latency. Striking a balance between reducing overhead and reducing latency can be difficult and requires an iterative methodology. Figure 6-1 depicts the process of polling in a diagram.

A typical application can create, read, update, or delete data. These four actions have been nicely put together in an acronym: CRUD. When you poll you are repeating your read requests. However, you make independent round trips to the server for create, update, and delete operations. When you go for one of these CUD operations, you could also read data and check for any updates. This simple enhancement on plain vanilla polling is called piggybacking. In piggybacking, you benefit from an additional read while you are on a trip to the server anyway, so it makes sense to reinitialize the polling counter every time such a piggyback trip occurs. That way, you truly avoid some extra trips. Figure 6-2 depicts piggybacking in a diagram.

A further enhancement on polling would be to wait at the server until an update arrives. That is, make a request and do not return instantly but instead wait till an update arrives. This process immediately hints that you would be blocking resources. It also says that scaling up could be troublesome as more concurrent waiting operations could mean more resource requirements, possibly to the point that one
Chapter 6: Communicating in Real-Time via Messages

runs out of such resources. Nonetheless, we enlist this as a possible option. Not all applications need scalability as a necessary requirement and sometimes waiting is better than going back and forth. This process of waiting until an update arrives is termed long polling. Long polling can improve latency requirements. Figure 6-3 depicts long polling in a diagram.

Beyond long polling, opening a direct connection could be an alternative. Direct socket-like connections can provide the pipe for servers or clients to freely interact via messages. However, opening up sockets in a Web world can be challenging. Custom options, such as the usage of XML Sockets, binary sockets, RTMP (Real Time Messaging Protocol), or HTML 5 WebSocket can be promising. Figure 6-4 shows the direct connection option in a diagram.

Finally, a hybrid option between long polling and direct connection exists. In this case, the direct connection is open, as with sockets, but resources are not necessarily blocked. Multiple client connections are shared over a set of resources. Connections waiting for updating are kept in a pending state and resources are released back to the pool. There has been much talk suggesting that “Comet” could be such an option. Figure 6-5 attempts to show a Comet-style interaction in a diagram.
Chapter 6: Communicating in Real-Time via Messages

What is Comet?

Comet (or Reverse Ajax) is a style of communication between a web based client and a server where the server initiates data push to the client. Unlike the traditional request-response communication model that a client originates, Comet-based communication involves a server-initiated data transmission. Comet, therefore, improves on the traditional style of polling for updates and scales better than long polling.

Comet is still an emerging framework and so while implementations exist from multiple vendors including Apache, Webtide, and DWR, the standards have yet to be established. A standard protocol, called Bayeux, and a common abstraction API, called Atmosphere, seem to be gaining traction and making progress in the direction of standardization.

Comet emerged in the world of Ajax but has seen adoption in other technologies as well. Comet can be used with Flash platform based clients.

Figure 6-2

Comet (or Reverse Ajax) is a style of communication between a web based client and a server where the server initiates data push to the client. Unlike the traditional request-response communication model that a client originates, Comet-based communication involves a server-initiated data transmission. Comet, therefore, improves on the traditional style of polling for updates and scales better than long polling.

Comet is still an emerging framework and so while implementations exist from multiple vendors including Apache, Webtide, and DWR, the standards have yet to be established. A standard protocol, called Bayeux, and a common abstraction API, called Atmosphere, seem to be gaining traction and making progress in the direction of standardization.

Comet emerged in the world of Ajax but has seen adoption in other technologies as well. Comet can be used with Flash platform based clients.
You can read more about Comet in Alex Russell’s seminal article on the topic titled “Comet: Low Latency Data for the Browser,” which is accessible online at http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/.

BlazeDS off-the-shelf supports polling and long polling and can be extended to work with Comet. LCDS, the commercial alternative to BlazeDS, supports RTMP.
XML and binary sockets can be opened to a BlazeDS server, but often it is difficult to open the socket in a reverse direction because of the absence of unique network identifiers for the clients. You will learn about the details of the available data push options and the extensions to standard BlazeDS data push functionality later in this chapter.

Next, you will learn about messaging domains, which are an important concept, necessary for effective usage of real-time messaging in BlazeDS.

**Messaging Domains**

The data push infrastructure in BlazeDS uses a message-based interaction model. Such an interaction model involves entities called *producers*, which send messages, and entities called *consumers*, which receive messages.
Producers and consumers can send and receive messages using either of two styles of communication: point-to-point or publish/subscribe. These two messaging styles are also referred to as messaging domains. In both these cases, the message-based communication between producers and consumers is predominantly asynchronous.

**Point-to-Point**

Point-to-point is a peer-to-peer messaging domain model. In this case, a logical entity called a queue is associated with each peer, and posting a message to a peer is equivalent to sending it to the queue that represents the peer. The communication is one-to-one.

Multiple producers can send messages to a single queue, and multiple consumers can fetch messages from a shared queue. When multiple consumers receive messages at the same queue, only one consumer can process an individual message. Let's illustrate this with an example. Say there are three consumers, c1, c2, and c3, and there are four messages, m1, m2, m3, and m4. Now if c1 consumes m1, then m1 cannot be consumed by c2 or c3. This behavior has two important implications:

- If you need a message to be consumed by multiple consumers, then a point-to-point messaging domain may not be optimal. In a point-to-point model, sending a message, say m1, to all consumers, say c1, c2, and c3, will mean sending three distinct messages, one each to c1, c2, and c3. This means that as the number of senders and receivers increases, you could end up with a large number of message transmissions. The number of message transmissions will equal the cross-product of the number of senders and receivers involved.

- If the order of message consumption is important, then you will have to externally manage the sequence in which the consumers, c1, c2, and c3, pick up the messages. In general, messages are placed sequentially in a queue, but their processing priority could be affected by factors such as the message expiration date, the message selectors, and message priority.

Consumers can be attached and detached to and from queues on demand, and that allows for a certain level of decoupling. Figure 6-6 summarizes a point-to-point messaging model.

If multiple producers and consumers need to interact, and each of the multiple consumers needs to process the same message, then using the publish/subscribe messaging model is more appropriate than using a point-to-point messaging model. You will learn about publish/subscribe messaging style next.

**Publish-Subscribe**

In the publish/subscribe messaging model, producers or senders of a message do not send messages directly to consumers or receivers of messages. Messages published by a producer can be classified into categories. Receivers or consumers register interest in particular categories and receive messages that are classified into those categories. Therefore, consumers receive only a subset of all the messages published.
The category or classification is sometimes also referred to as a filter or selector as it forms the criteria for selection of relevant messages from the entire bundle. Such category or filter definitions can be based on either of the following:

- **Topic** — In such a scheme logical channels called topics define the categories. Publishers publish to a topic and receivers register interest in a topic. Receivers receive all messages published to their topic of interest.

- **Content** — In such a scheme message metadata or content attributes define the classifier or the constraints. Receivers subscribe to messages based on these classifiers or constraints and receive all messages that match their subscription.

Publish/subscribe is most appropriate for cases where multiple consumers want to receive the same message. The model provides a system where the producers and receivers are totally decoupled. Therefore, such systems are scalable. Figure 6-7 summarizes a publish/subscribe messaging model.
Both the point-to-point and publish/subscribe messaging models involve a broker or a messaging infrastructure to send and receive messages. In the case of publish/subscribe, the broker could relay messages via a broadcast or a multicast. A broadcast involves sending messages to all connected consumers, whereas a multicast involves sending messages only to interested consumers and also optimizing the delivery by sending messages only once to a networked entity. This means that, in multicast, two logical entities sharing a network entity will receive only one message that will be shared between them.

Now that we have covered the fundamentals of messaging and messaging domains, I would like to jump to the topic of Java Message Service (JMS), which is used often with BlazeDS messaging.

**JMS**

Java Message Service (JMS) is a standard API for messaging within a Java EE infrastructure. JMS is part of the Java EE specification. In this section, you will learn the fundamentals of JMS. For most readers, this section will be enough to get you ready to start using JMS with BlazeDS.

If you are already familiar with the subject, then you can rapidly browse through this section, just to review the underlying concepts, or even skip the section. On the other hand, if JMS is completely new to you, then you’ll want to supplement the content in this section with the JMS tutorial that forms part of the Java EE tutorial, accessible at http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ (for J2EE 1.4) and http://java.sun.com/javaee/5/docs/tutorial/doc/ (for Java EE 5). In addition, there are a lot of good books on JMS.

**JMS Architecture**

You learned about the two alternative messaging models, point-to-point and publish/subscribe, in the last section. JMS supports both those models. You also learned there that the messaging model is mostly asynchronous. With JMS, you can implement both synchronous and asynchronous behavior. For BlazeDS, though, we will continue to stick with asynchronous interactions.

The JMS API and infrastructure can be architecturally classified into three important parts:

- **JMS provider** — JMS provider implements the JMS API and provides a necessary infrastructure for creating, sending, receiving, and reading JMS messages.
- **JMS clients** — Programs that interact with JMS to send and receive messages.
- **Administered objects** — JMS provides a few built-in objects, such as destinations and connection factories. You will learn more about them later in this section.

Figure 6-8 depicts the JMS architecture in a diagram.

To use JMS you need to explore the JMS programming model and its API.
JMS Programming Model and the API

The starting point for JMS is the administered objects. Destinations and connection factories are the two administered objects that help you get connected to the JMS provider. Connection factories can be of the following types:

- ConnectionFactory
- QueueConnectionFactory
- TopicConnectionFactory

Destinations can be:

- Queue
- Topic

To start a JMS interaction, a JMS client connects to the JMS provider and starts a session. A session forms a logical boundary within which all activities of sending and receiving messages occur. A session can also work as a transactional unit of work.

The simplest ways to establish a new connection and start a session are:

```java
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

The `createSession` method that creates a new session takes two parameters. The first of these specifies if the session is transactional or not and second one specifies the message’s acknowledgement characteristics. In the preceding code snippet, a transactional session with auto-acknowledgement is created.
Now that a session is ready for use, let’s create a message producer instance and send a message to a destination. From the section on messaging domains, you know that destinations can be queues or topics. For now, let’s choose a queue.

A queue-administered object can be injected into a variable as follows:

```java
@Resource(mappedName="jms/Queue")
private static Queue myQueue;
```

The preceding code works with Java 5 and greater, which supports annotations for resource injection. For Java 1.4, you will need to replace the preceding code with the following:

```java
Context ctx = new InitialContext();
private static Queue myQueue;
myQueue = (Queue) ctx.lookup("jms/Queue");
```

In the code above for Java 1.4, you create an instance of the `InitialContext` object. An `InitialContext` instance provides a starting point into the namespace from where the naming and directory service can be accessed. You first select a service provider for the Java Naming and Directory Interface (JNDI) services for an `InitialContext` instance. Then you provide any additional configuration that the `InitialContext` may need. Finally you create an instance of the `InitialContext` and use that instance to look up resources that are bound to the JNDI.

In either case, the administered object is looked up from the JNDI (Java Naming and Directory Interface) repository. It’s a general practice, when using Java EE, to register administered objects with the JNDI for easy access.

Next, you create a message producer and send a message via the producer to the queue you just initialized. This requires just a couple of lines of code:

```java
MessageProducer producer = session.createProducer(myQueue);
Producer.send(message);
```

The assumption is that you have already created a message object and set its headers, properties, and body.

Finally, you create a message consumer to consume the message you sent from the producer. A message consumer can be created as follows:

```java
MessageConsumer consumer = session.createConsumer(myQueue);
```

This consumer can receive the message synchronously using: `Message m = consumer.receive();` or asynchronously by registering a message listener. When the consumer receives a message synchronously, it blocks the queue until the message arrives. For asynchronous receiving, register a listener like this:

```java
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```
Chapter 6: Communicating in Real-Time via Messages

In either case, remember to start the connection to receive the messages. No messages will be received until the connection is explicitly started. Similarly, you could stop the connection to temporarily stop receiving messages.

Once the JMS messages are sent and the required operations are done, you should close the connection. Closing the connection makes sure that the administered objects and the other resources are put back in the pool for reuse or garbage collection, as the case may be.

The interactions for a topic-based message exchange follow a similar programming model, with a few changes to adjust for the messaging domain variance.

JMS is a very robust API that defines the contract and the model for message-based interaction. It is not a transport layer protocol or a specific implementation. Multiple vendors implement the JMS specification. Implementations of the JMS specification are referred to as JMS providers. JMS providers include a lot of features for message durability, transactional integrity, reliability, clustering, and exception handling. I will not discuss any of those issues here and leave them for you to explore on your own. It’s out of the scope of this book to explain JMS in detail. However, with the essential knowledge you’ve gained so far, you will be able to use JMS with BlazeDS.

Now that you are aware of the messaging and JMS prerequisites, it’s time to start learning BlazeDS messaging.

A BlazeDS-Powered Messaging Application

This section covers most things about BlazeDS messaging. The explanations and discussion are in the context of a sample application.

The Application Use Case

Before I start with the application itself, it may be prudent to explain what the application does. This example application is a tool for effectively watching changing currency exchange rates. It does the following:

- Allows you to set up currency pairs to watch for exchange rate changes
- Updates the rates in real time on the basis of the incoming rate change messages
- Plots the changing rates on a line chart in real time
- Provides a chat window to connect with your investment advisor for questions, clarifications, and advisory services related to investment opportunities created by these rate changes
- Allows you to buy or sell either side of a currency pair

Such an application in real life would connect to a currency OTC (over the counter) exchange, an interbank system, or market data provider for real-time price quotes. It would then connect to trading venues or broker-dealer systems for trade execution. In our sample application, I don’t do any of this, the reasons being:

- Access to trading venues, market data, and broker networks is neither trivial nor inexpensive.
Chapter 6: Communicating in Real-Time via Messages

- The complexity of connectivity to these external systems can overwhelm our sample application and overshadow the BlazeDS messaging characteristics that I want to highlight.

The dilemma though is that, without the updating exchange rates and the interactions thereafter, the application is pretty useless from a messaging standpoint. So, I have created a simple simulation program that generates some random changes to the rates, which BlazeDS pushes up to the Flex client. In order to further reduce the complexity and stay focused on the core issues, I have preselected a set of currency pairs for the watch list and excluded the feature to add and remove elements to/from the list. With that information let's start looking at the example.

I will build the application iteratively and walk you through all the interesting steps as I progress from the clean slate to the running application.

The First Few Steps

The example's exposition starts with setting up a Flex project, drawing out the initial view, and initializing the application with test data. As with earlier examples in this book, the Flex Builder is utilized for convenience, although the application will compile and work fine without it as well.

To get things started, create a Flex project and choose the J2EE server option as you create it. I use the JBoss AS (application server) for my example, but you have the choice to go with any other application server that supports JMS. If you decide to use Apache Tomcat, then remember to wire up an external JMS provider like ActiveMQ with it so that you can use the JMS infrastructure and the API. JMS is essential for this application. It is used with BlazeDS to push the exchange rate updates from the Java server side to the Flex client.

In Flex Builder, where I chose to create a joint Flex and Java project, using WTP, the structure and the folders appear as shown in Figure 6-9.

![Figure 6-9](image.png)

The Flex source resides in `flex_src`; the web application code is in the `WebContent` folder. Within this folder are the usual `META-INF` and `WEB-INF` folders. Java Servlet- and JSP-based web applications are commonly bundled as war files. "war" is an acronym for Web Archive file and is a special type of Java jar archive file. It's standard practice to bundle web components into a war file for deployment. The war file format is supported by all Java application servers.
BlazeDS is a Java Servlet–based web application, so the content being bundled as a war file is appropriate for deployment. A war file can be created using the `jar cvf \path\to\war\file\WarFileName.war -C \path\of\files\to\archive` command. In JBoss a war file can be deployed by simply copying the war file over to the `server/default/deploy` folder of a JBoss AS installation. In some situations, BlazeDS may form part of a larger enterprise application that uses middle tier components like Enterprise Java Beans (EJBs). Java enterprise applications are bundled as “ear” files. “ear” is an acronym for Enterprise ARchive, and it is yet another special form of the Java jar file.

The META-INF folder within WebContent is where the manifest file for the war file resides. Manifest files define the content of a Java jar file and work as its metadata. You will almost never need to directly play with the manifest file in the context of BlazeDS.

The WEB-INF folder, which resides side by side with the META-INF folder, is the most important folder and holds the following:

- The Java classes that implement the remote service or work in association with it.
- The `services-config.xml` file and its associated configuration files.
- `web.xml`, the necessary deployment descriptor that defines how a Java Servlet–based web application is deployed and determines how requests get routed to it.
- All the external library archive files required for the web application to work. These files are in the `lib` folder. All compiled classes bundled as a library are in the `lib` folder, and all compiled classes that are standalone are in the `classes` folder.

The compiled Flex code resides in the bin-debug folder and finally gets bundled at the root of the war file. There are a few additional files that hold assets and the metadata information for Flex Builder and WTP, but we will not delve into those here.

Once the project is all set up, go to the application’s MXML file and start coding the initial version of the view. I named my project `XchangeRate`, and my application’s MXML file is named `XchangeRate.mxml`. I only have a plain and simple DataGrid component at the beginning, which is in line with my idea to first get the static initial data and plumb that using the remoting service facility that BlazeDS provides. In many enterprise applications, you will use the remoting and messaging features in association with each other, as I do in this example.

The code for my initial screen is only a few lines:

```xml
<?xml version="1.0" encoding="utf-8"?>
    creationComplete="xRateDS.getxRateData()">
    <mx:RemoteObject id="xRateDS" destination="xrate"/>
    <mx:DataGrid id="xRateDG" dataProvider="{xRateDS.getxRateData.lastResult}"/>
</mx:Application>
```

If you look at the code carefully, you will notice that the data for the data grid is being fetched through a remoting service. Essentially, the result of a remote procedure call is set as the `dataProvider` of the DataGrid. The remote service itself is accessible via its destination handle, which in this case is `xrate`. 
Chapter 6: Communicating in Real-Time via Messages

You know from Chapter 4 how the remoting service works, and if you are feeling rusty, you may want to peek back into that chapter.

The destination name and its attributes are defined in `remoting-config.xml`, which by reference is part of the `services-config.xml`. The `services-config.xml` file is the configuration file that BlazeDS uses to set up and initialize its core artifacts, including channels, endpoints, services, destinations, and adapters.

I added the following to `remoting-config.xml`:

```xml
<destination id="xrate">
  <properties>
    <source>problazeds.ch06.XchangeRateDataService</source>
  </properties>
</destination>
```

The `remoting-config.xml` configuration binds the `XchangeRateDataService` Java class as the class behind the remote service that is accessible via the `xrate` destination. In this case, the class returns the initial data, which is, essentially, the following data for seven currency pairs:

- **symbol** — The currency pair identifier
- **bid** — Buy price
- **offer** — Sell price
- **lastUpdated** — Time when the data was available

The source for this service class is shown in Listing 6-1. The currency pair data points are represented in an object-oriented format. Listing 6-2 is the source for this object-oriented representation.

### Listing 6-1: The Remote Service Class That Returns the Initial Data

```java
package problazeds.ch06;

import java.util.List;
import java.util.ArrayList;
import java.util.Date;

public class XchangeRateDataService {

  public List getXRateData() {
    List list = new ArrayList();
    list.add(new XchangeRatePair("EUR.USD", 1.3238, 1.3241, new Date()));
    list.add(new XchangeRatePair("USD.JPY", 98.25, 98.27, new Date()));
    list.add(new XchangeRatePair("EUR.JPY", 130.08, 130.13, new Date()));
    list.add(new XchangeRatePair("GBP.USD", 1.4727, 1.4731, new Date()));
    list.add(new XchangeRatePair("USD.CHF", 1.1396, 1.1400, new Date()));
    list.add(new XchangeRatePair("EUR.CHF", 1.5091, 1.5094, new Date()));
    list.add(new XchangeRatePair("EUR.GBP", 0.89844, 0.89890, new Date()));

    return list;
  }
}
```
Listing 6-2: A DTO representing the Exchange Rate Data

package problazeds.ch06;

import java.util.Date;

public class XchangeRatePair
{
    XchangeRatePair()
    {
    }

    XchangeRatePair(String symbol, double bid,
                     double offer, Date lastUpdated)
    {
        this.symbol = symbol;
        this.bid = bid;
        this.offer = offer;
        this.lastUpdated = lastUpdated;
    }

    private String symbol;
    private double bid;
    private double offer;
    private Date lastUpdated;

    public String getSymbol()
    {
        return symbol;
    }

    public void setSymbol(String symbol)
    {
        this.symbol = symbol;
    }

    public double getBid()
    {
        return bid;
    }

    public void setBid(double bid)
    {
        this.bid = bid;
    }

    public double getOffer()
    {
        return offer;
    }

    public void setOffer(double offer)
The only other piece of code so far is the AS3 counterpart of the exchange rate object, which is as follows:

```as3
package problazeds.ch06 {
  [RemoteClass(alias="problazeds.ch06.XchangeRatePair")]
  public class XchangeRatePair {
    public var symbol:String;
    public var bid:Number;
    public var offer:Number;
    public var lastUpdated:Date;
  }
}
```

I don’t use this class yet, but it will come handy later. Also, so far there is no usage of the messaging service, but that’s what is going to come next.

**Sprucing Up the Client Code**

To get closer to our example application, I recraft the view component and spruce it up with a lot of additional plumbing to get it ready to listen for updates and process them once it receives them.

As a first move, I introduce a LineChart component. A LineChart draws out a line joining its data points. Such a chart can accommodate more than one series on the same chart and that’s something I am going to leverage. I will chart the bid and the offer quotes for a currency pair on a single component, right next to each other. That way as the updates come in, the spread between the two gets charted out in real-time. The code for the line chart is:

```xml
<mx:LineChart id="xRateLC" width="100%" height="50">
  <mx:series>
    <mx:LineSeries yField="bid" displayName="Bid" />
    <mx:LineSeries yField="offer" />
  </mx:series>
</mx:LineChart>
```
This chart is redrawn on every update. The chart is drawn using a custom function called `drawChart`, which is:

```ActionScript
private function drawChart():void {
    xRateLC.dataProvider = xRates[xRateDG.selectedItem.symbol];
}
```

If you look at this rather small but useful function, you will notice that the chart's data is based on the selected item in the data grid. Every currency pair in the data grid has an evolving pair of data points for its bid and offer quotes. Charting these data points reasonably within the available visual real estate implies depicting them one at a time. Therefore, chart displays are linked to the chosen item within the data grid.

To keep the data grid and the chart data synchronized and to accommodate the initial data load and subsequent updating, the data grid code is now:

```ActionScript
<mx:DataGrid id="xRateDG" dataProvider="(xRateLatestData)" change="drawChart()"
    width="100%" height="50%" />
```

Here, you will notice four important changes from the original version at the beginning of this section:

- The result is not bound directly as the data provider anymore. You will notice that `XchangeRatePair` is not annotated with the `[Bindable]` metadata.
- An event handler handles the returned results.
- On success, a separate currency rates array collection is created for each currency pair. This array collection holds all the data as it changes. This time series data is the underlying data for the charts.
- The charts are drawn after a successful data fetch.

The result of the remote procedure call is not the `dataProvider` of the data grid anymore. Instead a variable of an `ArrayCollection` type called `xRateLatestData` is bound as the `dataProvider`.

As soon as the application is ready, the call to the remote procedure is invoked. This hasn’t changed from the previous version of the code. How the results are handled has changed though. On success, the result is bound as the data provider of the data grid. The success cases invoke the `resultHandler`, which is the event listener for the `result` event. The `resultHandler` code is:

```ActionScript
private function resultHandler(event:ResultEvent):void {
    xRateLatestData = event.result as ArrayCollection;
    var xRatePair:XchangeRatePair;
    for(var i:int; i<xRateLatestData.length; i++) {
        xRatePair = xRateLatestData.getItemAt(i) as XchangeRatePair;
```
Besides all these changes, a new component called the **Consumer** is also used. **Consumer** and **Producer** are components that are used to receive and send data, respectively. These are the two client-side Flex components that support messaging and play an important role in data push applications. The consumer is this specific case is backed by a destination referenced as **xrateupdate**. This destination is configured in **messaging-config.xml**, which is included in **services-config.xml** by reference. The configuration details are shown in Listing 6-3. Most of the configuration in this example is about the JMS-specific parameters, which will become understandable after you read through the next subsection, which explains the simulated feed client.

Once a message arrives, the **messageHandler** event listener function is invoked. The **messageHandler** function does a few things, including the following:

- Finds out which pair has the update and updates the latest collection for the data grid
- Adds the updates to the currency rates array collection for the specific pair
- Sorts the currency rates on the basis of the **lastUpdate** value
- Draws the chart to include the update

The **messageHandler** code is:

```actionscript
private function messageHandler(event:MessageEvent):void {
    var xRatePair:XchangeRatePair = event.message.body as XchangeRatePair;
    //var initialXRatePair:XchangeRatePair;
    for(var i:int; i<xRateLatestData.length; i++) {
        var initialXRatePair:XchangeRatePair = xRateLatestData[i];
        if(initialXRatePair.symbol == xRatePair.symbol) {
            initialXRatePair.symbol = xRatePair.symbol;
            initialXRatePair.bid = xRatePair.bid;
            initialXRatePair.offer = xRatePair.offer;
            initialXRatePair.lastUpdated = xRatePair.lastUpdated;
        }
    }
    xRates[xRatePair.symbol].addItem(xRatePair);
    var sort:Sort = new Sort();
    sort.fields = [new SortField("lastUpdated", true)];
    xRates[xRatePair.symbol].sort = sort;
    xRates[xRatePair.symbol].refresh();
    drawChart();
}
```
Chapter 6: Communicating in Real-Time via Messages

Listing 6-3: Configuration for the Messaging Destination

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service id="message-service"
class="flex.messaging.services.MessageService">

    <adapters>
        <adapter-definition id="actionscript" class="flex.messaging.services.
messaging.adapters.ActionScriptAdapter" default="true" />
        <adapter-definition id="jms" class="flex.messaging.services.messaging.
adapters.JMSAdapter"/>
    </adapters>

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

    <destination id="xrateupdate">

        <properties>
            <jms>
                <destination-type>Topic</destination-type>
                <message-type>javax.jms.ObjectMessage</message-type>
                <connection-factory>ConnectionFactory</connection-factory>
                <destination-jndi-name>topic/XchangeRateUpdate</destination-jndi-name>
                <destination-name>XchangeRateUpdate</destination-name>
                <delivery-mode>NON_PERSISTENT</delivery-mode>
                <message-priority>DEFAULT_PRIORITY</message-priority>
                <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
                <transacted-sessions>false</transacted-sessions>
            </jms>
        </properties>

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

        <adapter ref="jms"/>

    </destination>

</service>
```

As mentioned before the best way to understand this configuration is to first understand the simulation feed.

The Simulated Feed Program

I mentioned previously that connecting our application to real-world data may not be feasible. So, a simulated feed is used to mimic such a situation and provides a way to demonstrate how updates can be handled. First, look at the code in Listing 6-4, which contains everything for the simulation feed.
package problazeds.ch06;

import java.util.*;
import javax.jms.*;
import javax.naming.*;

public class XchangeRateSimulatedFeed {

    private static Random random;

    /**
     * @param args
     */
    public XchangeRateSimulatedFeed() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        TopicSession pubSession;
        TopicPublisher publisher;
        TopicConnection connection;

        String _providerurl = "jnp://localhost:1099";
        String _ctxtFactory = "org.jnp.interfaces.NamingContextFactory";

        try {
            // Obtain JNDI Context
            Properties p = new Properties();
            p.put(Context.PROVIDER_URL, _providerurl);
            p.put(Context.INITIAL_CONTEXT_FACTORY, _ctxtFactory);
            p.put(Context.SECURITY_PRINCIPAL, "admin");
            p.put(Context.SECURITY_CREDENTIALS, "admin");
            Context context = new InitialContext(p);

            // Lookup a JMS connection factory
            Object tmp = context.lookup("ConnectionFactory");
            TopicConnectionFactory factory = (TopicConnectionFactory) tmp;

            // Create a JMS connection
            connection = factory.createTopicConnection();

            // Create publisher session
            pubSession = connection.createTopicSession(false,
                                                      Session.AUTO_ACKNOWLEDGE);

            Continued
// Lookup a JMS topic (If you use JRun, topics are configured in SERVER-INF\jrun-resources.xml)
Topic topic = (Topic) context.lookup("topic/XchangeRateUpdate");

// Create a publisher and a subscriber
publisher = pubSession.createPublisher(topic);

List xRateInitialData = new XchangeRateDataService().getXchangeRateData();
random = new Random();
while (true) {
    Thread.sleep(100);
    for(int i=0;i<xRateInitialData.size();i++) {
        ObjectMessage message = pubSession.createObjectMessage();
        XchangeRatePair xcrp = updatedXRate((XchangeRatePair)xRateInitialData.get(i));
        message.setObject(xcrp);
        publisher.publish(message, Message.DEFAULT_DELIVERY_MODE, Message.DEFAULT_PRIORITY, 5 * 60 * 1000);
        System.out.println(message.toString());
    }
}
}

private static XexchangeRatePair updatedXRate(XchangeRatePair xchangeRatePair) {  
    boolean bidOrOffer = random.nextBoolean();  
    System.out.println(" bidOrOffer " + bidOrOffer);  
    boolean positiveOrNegative = random.nextBoolean();  
    System.out.println(" positiveOrNegative " + positiveOrNegative);  
    double bid = xchangeRatePair.getBid();  
    double offer = xchangeRatePair.getOffer();  
    if(bidOrOffer == true) {
        double bidChange = bid * 0.0009;  
        if(positiveOrNegative == true) {
            bid = bid + bidChange;  
        } else 
        
        offer = offer - bidChange;
    }
    xchangeRatePair.setBid(bid);
else
{
    double offerChange = offer * 0.0009;

    if(positiveOrNegative == true){
        offer = offer + offerChange;
    } else
    {
        offer = offer - offerChange;
    }

    xchangeRatePair.setOffer(offer);
}

xchangeRatePair.setLastUpdated(new Date());

return xchangeRatePair;
}

I will dissect the code and point to the most important parts without necessarily walking through every single aspect of the semantics. The first part of the code focuses on the JMS client-specific features of the program. The administered objects are accessed from the JNDI instance accessible at jnp://localhost:1099, in the case of JBoss. A JMS connection and then a session are created.

Within a JMS session a logical connection is established with a JMS topic identified as topic/XchangeRateUpdate. You will notice that, in the configuration in the last section, the topic has the same identifier. This is how a JMS client, like this program, ends up sending the message to a topic that is visible to BlazeDS and its JMS adapter.

The simulation Java program sends an object message. An object message is created by using ObjectMessage message = pubSession.createObjectMessage();. The output of a simple simulation program that creates a small random change to the original value is set as the body of the message object. The simulation itself is encapsulated in a small function: updatedXRate. The details of the implementation in this function are not as important as the fact that it just creates a small random change to the bid and offer values, based on some random choices. You are, of course, free to investigate further and find the logic, if that interests you. In addition, you are free to change it as desired.

That completes bulk of the application. What is still left are the chat window and the order entry screen. I will not explain the order entry screen, although it’s there for you in the code download. Let me briefly discuss the chat application.

**The Chat Window**

Although the chat window in the example application is part of the entire system, it’s actually logically isolated from the market data updating functionality for the most part. You can chat with your investment advisor about the moving prices and that is where the unity exists from a user interaction...
Chapter 6: Communicating in Real-Time via Messages

perspective, but apart from that, the common elements are limited. You can develop the chat application independently of the part that relies on JMS messaging. Let’s see how.

The messaging infrastructure in BlazeDS can be utilized for message-based communication between two Flex clients. Messages are routed between the two participating clients via the server. That is, a Flex client sends a message down to a server-side messaging destination via a channel and an endpoint pair. Another Flex client listens for messages at this destination. When a destination receives a message, all subscribers to that destination receive the message.

JMS and the JMSAdapter are not required in this entire interaction. Instead, an ActionScriptAdapter facilitates the communication between the Flex clients. In the next section, you will delve more deeply into these adapters and also learn how to create a custom messaging adapter.

The messaging infrastructure with ActionScriptAdapter supports a publish/subscribe messaging domain. That is, message producers and consumers are coupled together on the basis of an intermediary, which in this case is a topic. This implies that messages sent to a topic will be received by all subscribers. However, sometimes such coarse-grained broadcasting may not be desirable. The messaging infrastructure allows the use of selectors and subtopics to filter messages. Message filtering ensures that only those subscribers who satisfy the filter criteria receive the messages. Although the filtering mechanisms help narrow down the message receivers, it’s still a publish/subscribe messaging domain. For point-to-point messaging, you need to either use the JMS queues with the JMS adapter or create a custom adapter to implement such behavior.

Next, the code for the simple chat application is discussed.

**The Simple Chat Application**

The code for the simple chat application is shown in Listing 6-5.

**Listing 6-5: The Simple Application for the Chat Window**

```xml
<?xml version="1.0" encoding="utf-8"?>
creationComplete="initApplication();">  
<mx:Script>
<!--[CDATA[
    import mx.controls.Alert;
    import mx.messaging.events.MessageEvent;
    import mx.messaging.messages.AsyncMessage;
    import mx.messaging.events.MessageFaultEvent;

    [Bindable]
    private var usersData:Array = new Array("John", "Jane", "Jack", "Jill");

    private function initApplication():void
    {
        messageConsumer.subscribe();
    }

    private function showMessages(message:String):void
```
private function messageHandler(event:MessageEvent):void
{
  var message:String = null;
  message = event.message.headers["sender"] + ":" + "(to "
  + event.message.headers["receiver"] + ") " + event.message.
  body;
  showMessages(message);
}

private function sendMessage():void
{
  var message:AsyncMessage = new AsyncMessage();
  message.headers = new Array();
  message.headers["sender"] = userName.text;
  message.headers["receiver"] =
      users.selectedItem.toString();
  message.body = chatText.text;
  messageProducer.send(message);
  chatText.text = "";
}

private function faultHandler(event:MessageFaultEvent):void
{
  Alert.show(event.message.body.toString());
}
This simple application has a message producer and consumer. Both the entities communicate on a single destination, `xrchat`. From the user interface perspective, there are four important components, which are:

- A text box in which to enter your name; the value entered in this text box becomes the sender's name
- A list of names of message receivers
- A chat message display area
- A text box and an associated button for sending messages

When you type a message and click the button to send the message out, the `sendMessage` function is invoked. The `sendMessage` function creates a new instance of `AsyncMessage` and then sets the headers and body for that message. Once a message is ready an instance of a message producer's `send` method is used to send the message. Headers are stored as an `Array` and are bound to the message through its `headers` property. A message has BlazeDS-specific headers, which start with the keyword `DS` and can have custom headers, as shown in the preceding example.

In our simple example, a message consumer subscribes to the same topic to which a producer sends the messages. The consumer defines a message event. On receiving a message, a message listener is triggered. The message listener extracts information from the received message and adds it to the chat message display area.

In order to get this application up and running, you also need to add a messaging destination configuration to `messaging-config.xml`. The configuration information is:

```xml
<destination id="xrchat">
  <properties>
    <network>
      <session-timeout>0</session-timeout>
      <throttle-inbound policy="ERROR" max-frequency="50"/>
      <throttle-outbound policy="REPLACE" max-frequency="500"/>
    </network>
    <server>
      <max-cache-size>1000</max-cache-size>
      <message-time-to-live>0</message-time-to-live>
    </server>
  </properties>
  <channels>
    <channel ref="my-polling-amf"/>
  </channels>
</destination>
```
That completes the application. By now, you have seen both the JMSAdapter and the ActionScriptAdapter. Next, you will learn a few things about these two and also explore how a custom adapter can be written and configured.

The Adapters at Work

You have seen the JMSAdapter and the ActionScriptAdapter in the preceding examples. This section explains both these adapters and lays out the configuration options for them. It also includes a brief discussion on custom adapters.

The ActionScriptAdapter

The ActionScriptAdapter is a lightweight adapter that enables publish/subscribe messaging between Flex clients. The ActionScriptAdapter extends the MessagingAdapter class, which is the base abstract class that supports messaging. Adapters sit at the end of the service call pipe and facilitate interaction with server-side objects or special formats and standards.

The ServiceAdapter interface defines the core contract of an adapter in BlazeDS. The HTTPProxyAdapter that you learned about in Chapter 3 and the JavaAdapter that you learned about in Chapter 4 extend the ServiceAdapter interface. The MessagingAdapter abstract class also extends the ServiceAdapter.

The core features of the ActionScriptAdapter are contained in the invoke method of the adapter. Messages are routed to subscribed Flex clients from inside the invoke method. Following is the code that sends the messages out:

```java
msgService.pushMessageToClients(message, true);
msgService.sendPushMessageFromPeer(message, true);
```

The pushMessageToClients method of the MessageService class pushes the message to all subscribed local Flex clients. By local, I mean those that reside on the same node within a cluster. If a message needs to be transmitted to clients that are members of the other cluster nodes, then the method sendPushMessageFromPeer comes in handy. You can configure one of the following message routing modes for a cluster:

- broadcast
- server-to-server

In the broadcast mode, the message is broadcast through the cluster. In the server-to-server mode the message is sent only to servers with matching subscription requests.

That covers the core aspects of the ActionScriptAdapter. Next, you will learn about the JMSAdapter, which you are already familiar with from the previous sections of this chapter.
Chapter 6: Communicating in Real-Time via Messages

The JMSAdapter

Like the ActionScriptAdapter, the JMSAdapter extends the MessagingAdapter. The JMSAdapter bridges the gap between BlazeDS and the JMS queues and topics. The essential mechanism involves message routing and passing as with the ActionScriptAdapter. However, things are more complicated than what you saw in the ActionScriptAdapter because the program needs to connect and send messages using the JMS idioms and syntax. The JMSAdapter class is assisted by a set of classes that specifically cater to interfacing with the JMS infrastructure. The JMS specific classes are:

- **JMSConfigConstants** — Constants for the JMS adapter
- **JMSConsumer** — JMS Proxy subclass for the JMS message consumer
- **JMSEExceptionEvent** — Event triggered on a JMS exception
- **JMSEExceptionListener** — Listener for the JMS exception
- **JMSMessageEvent** — Event triggered on JMS message
- **JMSMessageListener** — Listener for the JMS message
- **JMSProducer** — JMS Proxy subclass for the JMS message producer
- **JMSProxy** — Superclass of the JMS consumers and producers
- **JMSQueueConsumer** — JMS queue receiver
- **JMSQueueProducer** — JMS queue sender
- **JMSSettings** — JMS adapter settings and configuration
- **JMSTopicConsumer** — JMS topic receiver
- **JMSTopicProducer** — JMS topic sender

You have already seen the JMS adapter in action and configured an instance with the example application so repeating the same here would be redundant.

The **invoke** method still remains the primary entry point of the adapter. The invoke method of the JMSAdapter first checks to find out the destination type from its settings. If the destination type is a topic, then it uses a **TopicProducer** to send a message and a **TopicConsumer** to consume a message. On the other hand, if the destination type is a queue, then it uses a **QueueProducer** to send a message and a **QueueConsumer** to consume a message.

To create a custom adapter, first create a class that extends the **MessagingAdapter** abstract class. Then start coding the logic for message sending and receiving in the invoke method. It’s as straightforward as that. Thus, the logic itself could be complicated, but the adapter API ensures that it does not add further complications and keeps the adapter contract simple and elegant.

Next, we touch upon the topic of extending BlazeDS for enterprise-grade data pushing. That is the last section before it’s time to wrap up this chapter.

Advanced Extensions to BlazeDS Data Push

Although BlazeDS supports data pushing through its streaming and polling channels, its readiness as an enterprise-grade data pushing solution is suspect. The commercial alternative to BlazeDS, LifeCycle
Chapter 6: Communicating in Real-Time via Messages

Data Services (LCDS) does not suffer from this problem though. LCDS has a direct scalable connectivity option in RTMP, which is an acronym for Real Time Messaging Protocol. RTMP is a proprietary protocol that is capable of high-performance data pushing and streaming. Earlier this year, Adobe announced that it would open the RTMP specification to the community. Until that happens and subsequently an implementation for RTMP becomes available for BlazeDS, from a scalability and reliability perspective, data push options in BlazeDS look weak.

So, your choices are to either go with LCDS or bring the robust Comet and NIO options into BlazeDS’s fold. Going with LCDS is a commercial decision, since the product is licensed and the license fee is not inexpensive. Going with Comet requires the creation of new endpoints and plumbing changes in the core BlazeDS message broker and its artifacts. In this section, I will explain how one can incorporate Comet and NIO into BlazeDS. First, though, let me define these terms for you.

Comet, as a term, was proposed by Alex Russell of the Dojo project (www.dojotoolkit.org) in a blog post titled “Comet: Low Latency Data for the Browser.” You can access Alex’s blog post on Comet at http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/. He used the term to connote event-driven server-initiated data pushing of streams to web applications from the server over long-lived HTTP connections. This was a distinct deviation from the traditional request-response–based interactions. It was also clearly separated from the polling-based mechanism to get frequent updates. Comet today serves as an umbrella term for all server-initiated data push options over persistent HTTP connections. A few standards like the Bayeux protocol (http://svn.cometd.org/trunk/bayeux/bayeux.html) are attempts at standardizing Comet, but we are still a long way from universal acceptance of this standard.

NIO, or Java NIO, is the new Java I/O specification that was introduced in Java 1.4. NIO introduces new functionality for enhanced performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching. Its nonblocking I/O support is extremely useful in creating persistent connections that form the truss for a successful Comet implementation. Therefore, NIO servers as an important ingredient in Comet implementations with Java-based web and application servers.

In addition, the new Servlet 3.0 specification directly adds support for persistent connections, better thread management, and asynchronous communication, to allow for Comet-like interactions in Java Servlet containers and application servers. A few open source application servers, such as Webtide Jetty and Apache Tomcat, have already started integrating these features into their new releases.

For this section, I will choose Jetty 7 as the web and application server of choice and show how Comet could be leveraged with BlazeDS in that server.

**Comet Inclusion with BlazeDS in Jetty 7**

One of the compelling reasons to choose Jetty 7 as the server is the support for “continuations” in that server. The word “continuations” in the context of Jetty implies the ability to suspend an ongoing request processing and resume it at a later time. The process could resume either after a timeout ends or on a notification.

In Jetty 7, calling ContinuationSupport.getContinuation(HttpServletRequest request, Object lock) returns a reference to a Continuation instance. You can call the suspend method of this continuation instance to stop the current request from processing. The suspend method takes a timeout parameter of type long. Once the timeout period elapses or on some notification the request processing can be
Chapter 6: Communicating in Real-Time via Messages

resumed. An invocation of the `resume` method resumes the request processing. The continuation object’s status verification methods, such as `isPending` and `isResumed`, are helpful in checking the current state of the continuation. In nonblocking continuations `isPending` is set to true for the entire period between the first call to suspend and a subsequent second call to the `suspend` method. The `isResumed` method is true once the processing is resumed. If the call needs to maintain state, it’s advisable to save it by calling the `setObject(Object o)` method before a `suspend` method is called. Once the call resumes, the saved state can be retrieved by calling the `getObject` method of the continuation instance. Any arbitrary objects can be saved away and retrieved in the context of a continuation in this manner.

The request processing in BlazeDS starts with the `MessageBrokerServlet`. This is because of the configuration to route all requests to URL(s) with the `/messagebroker/*` pattern in their name to `MessageBrokerServlet`. The relevant configuration in `web.xml` is:

```xml
<servlet-name>MessageBrokerServlet</servlet-name>
<servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>

....
<servlet-mapping>
  <servlet-name>MessageBrokerServlet</servlet-name>
  <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

Now an easy hack to include Comet support and yet not disturb the existing BlazeDS infrastructure very much is to create an `HttpServlet` that supports continuations and have all calls to a new endpoint, which you may want to call `JettyStreamingAMFEndpoint`, divert to this new Servlet. The continuations support Servlet could be named `ContinuationMessageBrokerServlet`.

Within `ContinuationMessageBrokerServlet`’s `service` method, you want to get hold of a continuation instance and call the `JettyStreamingAMFEndpoint`’s `service` method by passing in the continuation parameter to it:

```java
Continuation c = ContinuationSupport.getContinuation(req, null);
...
endpoint.service(c, request, response);
```

Within the `JettyStreamingAMFEndpoint`, you may want to keep stock of all pending requests and put them in a collection called `pendingContinuations`. Then you may want to include the following to include the functionality of suspending a processing request in a nonblocking manner:

```java
if (!c.isPending()) {
    pendingContinuations.add(c);
}

c.suspend(timeout);
```

When the response arrives, that is, a server side data push event occurs, you want to resume the processing and send the response out. The code is:

```java
c.resume();
pendingContinuations.clear();
```
That completes a high-level view of how comet-based data pushing can be included in BlazeDS. Such an option is restrictive to the extent that it will only work with Jetty. In future, as the Comet standards are stabilized and more application servers support it, I expect BlazeDS to start supporting it, too.

I am working on creating an open source robust Comet-supportive data push option for BlazeDS. For now, only Jetty and Tomcat are included in that effort. This open source initiative is christened “dsadapters.” The project is hosted on Google Code: http://code.google.com/p/dsadapters/.

That brings us to the end of the chapter on messaging, but before we move to the next topic, let’s recap what was covered in this chapter.

**Summary**

This chapter starts with a general explanation of data pushing and moves on to cover topics that relate to messaging, messaging domains, and JMS. In the context of data pushing, polling, piggybacking, long polling, persistent connections, and Comet were discussed. The messaging domains of point-to-point communication and publish/subscribe interactions were described and compared. JMS is introduced and the essentials of the infrastructure, the administered objects, and the API were illustrated.

Then the chapter builds on an example that includes the usage of both JMSAdapter and the ActionScriptAdapter. Messaging within BlazeDS, the classes involved, the configurations, and custom classes were all explained with the help of this example. The example is the biggest part of the chapter. Although not necessarily complex, it brings most important features in BlazeDS to life.

The sample application can also be looked at as an aggregation of a few related but independents concepts, namely:

- Message-based integration between a Flex client and a Java server
- Use of JMS in conjunction with a Flex application
- Messaging between two Flex clients
- Usage of built-in messaging adapters in BlazeDS

After the entire example was presented, there was a small section on the two robust built-in messaging adapters, namely ActionScriptAdapter and JMSAdapter.

Finally there was a last, small section that focused on enhancements to BlazeDS for enterprise-grade data pushing. Continuations in Jetty were explored as an option for Comet implementation.

Hopefully, you are fairly familiar with the messaging features in BlazeDS after reading this chapter. I assume that you are inspired and excited to leverage this feature in building real-time Rich Internet Applications.
7

Leveraging JPA and Hibernate with Flex

Java is an object-oriented programming language and as such, applications built using it model domain entities as objects. Objects encapsulate data and behavior together and provide the familiar real-world semantics to interact with it. A majority of modern-day applications store persistent data in popular data storage engines like databases, most of which are relational in nature.

The world of objects and the world of relational entities view the same context from different perspectives, so mismatches exist between them. I will present a small example and illustrate some of the most common problems related to this mismatch. However, indulging in a thorough discussion on this topic is beyond the scope of this chapter.

Before I start digging into the example, it is worthwhile to state up front that the best known solution to the problem comes from the breed of tools classified under the Object Relational Mapping (ORM) category. Such tools reduce the impedance to the flow and data interchange between the objects and the relational formats. Hibernate is one such open source Java-based ORM solution. Hibernate is both extremely popular and well liked by Java developers.

Java Persistence API (JPA) is a Java Community Process (JCP)–created standard specification that defines the contracts and interfaces for ORM between Java and the relational world. Hibernate is a JPA implementation and so are TopLink and OpenJPA. Hibernate has features that go beyond the JPA and so can be thought of as a superset of JPA.

Flex applications are programmed in ActionScript 3 (AS3). AS3, like Java, is an object-oriented language. Therefore, the object/relational impedance mismatch becomes relevant in Flex applications as well. In the set of Flex applications that connect to data stores through a Java middle tier, most of the bridging between the two worlds of objects and relational data is the responsibility of the Java layer. Therefore, Flex applications, with a Java middle tier or service layer, can leverage JPA and Hibernate as an ORM solution. Custom extensions to BlazeDS can make loading, storing, updating, and the propagation of domain model objects through the chain, from the rich client to the database connector, smooth and seamless. This chapter concentrates on such extensions and explains how they work with JPA and Hibernate.
Next, we go back to the topic of object relational mismatch and walk through an example to understand the problems.

Object Relational Mismatch

This section of the chapter walks through a simple example and brings to light a few of the common differences between the object world and the relational world.

Let’s assume that your application keeps records for the employees of a company. Each employee is a person who has a few attributes such as name, ID, role, manager’s name, and contact information. Each person’s contact information could include his or her email, phone, fax, and instant messaging ID.

For starters, let’s create the database schema and the object model for the “employee” entity and its associated information. The first problem that you will immediately encounter is that of granularity.

Coarse-Grained or Fine-Grained

Following object-oriented principles, you may end up creating three objects, namely Employee, ContactInfo, and Role, with definitions like this:

```java
public class Employee
{
    private int employeeId;
    private String name;
    private Set roles;
    private String managerName;
    private ContactInfo contactInfo;

    //accessor methods
}

public class ContactInfo
{
    private String email;
    private String phone;
    private String fax;
    private String im;

    //accessor methods
}

public class Role
{
    private int roleId;
    private String description;
    private String businessUnit;

    //accessor methods
}
```
On the other hand, you will create two tables to keep the relational schema optimized. The Data Description Language (DDL) statements for the two table creation operations are like so:

```sql
create table employees (
    id INT NOT NULL PRIMARY KEY,
    name VARCHAR(100),
    manager_name VARCHAR(100),
    email VARCHAR(75),
    phone VARCHAR(75),
    fax VARCHAR(75),
    im VARCHAR(75)
);

create table roles (
    role_id INT NOT NULL PRIMARY KEY,
    description VARCHAR(100),
    business_unit VARCHAR(100),
    employee_id INT NOT NULL,
    FOREIGN KEY (employee_id) REFERENCES employees(id)
);
```

These DDL statements are for MySQL. Other databases may have a slightly different syntax.

Immediately, you will notice that the contact information is stored differently under the two schemes. In the object-oriented world, the contact information is encapsulated in a separate class, which can be considered finer grained than the Employee class. In the relational world, attributes of the contact information are stored as columns within the employees table. Why is this so?

Data is selected and manipulated in the relational world using SQL. SQL selects are more efficient when data resides in a single table as opposed to when it resides in multiple tables. Concurrently fetching data from multiple tables leads to join operations that are expensive and less efficient.

This is one point of difference between object representations and relational tables. Objects can represent data at varying levels of granularity. For example, the Employee and the ContactInfo class contain data at different levels of granularity. Relational tables, on the other hand, have no notion of granularity. They are all made up of the constituent columns. The only deviation from this rule comes in the form of custom data types or user-defined types, which, by the way, are not supported by all relational databases.

Using user-defined types, one could define a complex data type for the contact information. Then, instead of four separate columns, you could keep all the information on email, phone, fax, and instant messaging within a single column. However, “custom types” compromise on portability and performance, so sticking to multiple columns appears as the best choice.

In addition to differences in terms of granularity, there are some major points of divergence when it comes to object-oriented concepts like inheritance and polymorphism.

**The Cases of Inheritance and Polymorphism**

So far our model does not talk about inheritance and polymorphism, which are central concepts in the object-oriented paradigm. Now, I will introduce these ideas into the existing model for the employee records management application.
Say that there are two subtypes for the Role type, namely: LineResource and SupportStaff. LineResource represents all those employees who are directly involved in the production and delivery of the products and services of the company. Every other supporting member is classified as a SupportStaff.

Using objects, it’s elementary to depict this relationship, as both LineResource and SupportStaff extend from the Role class. These extensions can contain additional attributes and override standard behavior. The employee record management program methods can work with the Role super class and a specific subclass can be wired in depending on the context or business rules, thereby incorporating polymorphic behavior.

These ideas of inheritance and polymorphism get fuzzy and complex when applied to the relational databases. First, relational systems have no notion of hierarchy among tables. There are no super- and subtables in a relational database management system. Views or virtual tables can pretend to create such a hierarchy but they aren’t the same, because they don’t represent a type and its subtype. In general, the notion of type has no relevance in the world of the relational tables.

Second, polymorphic behavior implementation implies the existence of a foreign key relationship with multiple subtypes in the same place. That is, one foreign key will need to define a constraint that holds for the Role type and its subtypes LineResource and SupportStaff. Such foreign key semantics are not supported by relational systems. At best, one could use a database procedure to impose such a foreign key constraint.

So, there are some major disconnects when it comes to inheritance and polymorphism. This problem is pervasive, even in the context of defining associations between the entities and navigating from one to the other.

**Entity Relationships**

Entities are related to each other through defined associations and foreign key constraints. When you access data, you traverse these interrelationships to get the entire relevant data set.

Taking the case of employees and their roles, one can safely assume that a many-to-many relationship could exist between the two. An employee could wear multiple hats and so be associated with multiple roles, and many employees could be classified under the same role. (I am not restricting to only two subtypes of a Role, as defined in the last section. That was just an example. Role types could be structured in many different ways, including classifications based on business functions and hierarchy.)

The many-to-many cardinality in the relationship poses no complications in the world of objects. All one needs to do is to use collections instead of individual entities when referring to the association. For example, the Employee class could access its roles using the following methods:

```java
public Set getRoles() {
    return roles;
}

public void setRoles(Set roles) {
    this.roles = roles;
}
```
For Java 5 and above, which support generics, these methods would work with a `Set<Role>` type.

Tables representing one-to-one and one-to-many (or many-to-one when viewed from the other side) relationships simply involve appropriate foreign key definitions. However, things get tricky when many-to-many associations exist. The foreign key constraint method isn’t appropriate. In relational databases, a many-to-many cardinality demands the creation of a new table, called the *association* or the *link table*, which maps the two entities. So, a many-to-many employees-to-roles association will imply creation of a table on the following lines:

```sql
create table employees_roles (
    id INT NOT NULL,
    role_id INT NOT NULL,
    FOREIGN KEY (id) REFERENCES employees(id),
    FOREIGN KEY (role_id) REFERENCES roles(role_id)
);
```

Such tables have no counterparts in the object world because objects don’t treat many-to-many relationships any differently from one-to-many relationships.

So far, I have always started with the object viewpoint and then shown how the particular idea translates to the world of relational tables. This may have given you the false impression that the object viewpoint is more flexible and robust than its relational counterpart. However, this is not true at all and that is why I wanted to state it explicitly here. Relational databases serve well in more than one way and are a time-tested preferred way of storing and accessing data.

One place where they distinctly outsmart the object approach is accessing associated or networked entities. Let’s consider the employees and roles association once again. Using SQL, one could retrieve all the associated roles of an employee whose ID is 1, with a single query, like so:

```sql
select er.id, er.role_id from employees_roles er, employees e
where er.id = e.id
and e.id = 1;
```

In the object-orientated approach, to get the same data set you would first invoke a method as follows: `anEmployee.getRoles()`. This would give you a set of all roles associated with an `Employee`. Then you would iterate over this set and call `aRole.getRoleId()` for each member of this set. This is when you would get the list of all role IDs associated with an employee. So, if there were `n` roles associated with an employee, you would invoke `n + 1` selects to get to the required data. Compare this with the single select statement using SQL and you can immediately see how traversing the object network may not be efficient when it comes to retrieving associations that have one-to-many or many-to-many cardinality.

This inefficiency is amplified because of the additional overhead that object orientation imposes at the time the association is defined. In the earlier example, the relationship between employees and roles could be bidirectional, meaning that you will not only query all roles for an employee but may also query all employees that map to a role. In terms of implementation, this means accessor methods will be defined in an `Employee` to fetch roles, and similar counterparts would be defined in a `Role` to fetch all employees. You saw the getter and setter for roles in an `Employee` class earlier in this section. The getter and setter used in a `Role` class to fetch the set of employees might look like this:

```java
public Set getEmployees() {
    return employees;
}
```
In SQL, you do not need to define the same relationship twice. Apart from these differences, the two sides have another major point of difference and that relates to the way they uniquely identify a record.

**What Defines Identity?**

In relational databases, a row in a table is uniquely identified by its primary key. A simple primary key maps to a single column, whereas a composite primary key involves a combination of multiple columns.

Rows in a database table map to object instances. Object instances have two notions of identity, which are based on either of the following:

- **Reference**
- **Value**

When evaluated in terms of “reference,” two objects are same when they both point to the same instance. Such objects return “true” when compared using the `==` operator.

When evaluated in terms of “value,” two objects are same when the values of their attributes are identical. Usually, the `equals` and the `hashCode` methods implement the notion of equality by value.

The pertinent question, then, is which object identity accurately represents the primary key based identity that the relational model adheres to. The straight answer is that neither reference- nor value-based object identity maps to primary key–based identity.

However, ORM tools like Hibernate provide a way to accommodate the differences and facilitate the interaction between the two models. ORM tools also reduce the problems arising out of the discrepancies stated earlier in this chapter.

While it’s very tempting to indulge further in the discussion on object relational mismatch, you should have enough information to understand the differences between the two paradigms.

It’s time now to explore the Java Persistence API and Hibernate and see how these options help objects interact successfully with relational tables.

**Introducing JPA**

Java Persistence API (JPA) defines an easy and standard way to manage persistence in the Java EE and Java SE environments. The lightweight API defines a central coordinating persistence manager, called the `EntityManager`. Each persistence context, which essentially consists of a set of managed persistent entities that exist in a particular data store, has an `EntityManager`. The `EntityManager` API allows interaction with the persistence context and provides for the creation, persistence, and removal of entities.

Entities that are persistent have their defining classes annotated with the `javax.persistence.Entity` annotation. Classes can define their attributes as fields or properties. Sometimes both fields and
properties can coexist in a single class. When a class adheres to the Java bean–style accessor pairs for its attributes, they are referred to as properties. In general, all but transient fields and properties are persisted to the data store.

**Primary Key**

Each persistent object is identified by a unique identifier, referred to as the primary key. If a single field or property is the primary key, then such a field or property is annotated with the `javax.persistence.Id` annotation. An example is:

```java
@Entity
public class Person {
    ....

    String personId;

    @Id
    public String getPersonId{
        return personId;
    }
    ....
}
```

A `String` type property is the primary key in the preceding example. Other data types, namely the following, are also possible types for a primary key field or property:

- Java primitive types (example: `int`)
- Java primitive wrapper types (example: `Integer`)
- `java.util.Date`
- `java.sql.Date`

Avoid using a floating point type for primary key type and, as far as possible, stick to integers for generated IDs. Only generated integer IDs are portable.

Sometimes, the primary key is complex and can involve multiple fields or properties. In such cases, an external class defines the primary key. Such a class can be referred with the help of either of the following annotations:

- `javax.persistence.IdClass`
- `javax.persistence.EmbeddedId`

For example, you could have the `Person` class from the last example define a primary key basis as a person’s first and last name. Then the example, using the `@EmbeddedId` annotation, would look like this:

```java
@Embeddable
public class PersonId implements Serializable {
    private String firstName;
    private String lastName;
    ....
}
Chapter 7: Leveraging JPA and Hibernate with Flex

```java
@Entity
public class Person implements Serializable {
    private PersonId id;

    @EmbeddedId public PersonId getId() {
        return id;
    }
    ....
}
```

Alternatively, using the `@IdClass` annotation, the example would be as follows:

```java
public class PersonId implements Serializable {
    private String firstName;
    private String lastName;
    ....
}
```

```java
@IdClass(PersonId.class)
@Entity
public class Person implements Serializable {
    private PersonId id;

    @EmbeddedId public PersonId getId() {
        return id;
    }
    ....
}
```

The unique identifier, the primary key, is used to query for a specific entity instance.

Entities hardly ever exist in isolation and often have relationships with other entities in a persistence context. Therefore, in the context of entity persistence, managing relationships becomes almost as important as making use of an appropriate identifier or a primary key.

**Entity Relationships**

The relationship between entities can be defined using two characteristics:

- Multiplicity
- Directionality

Multiplicity in relationships can be of four types:

- One-to-one
- One-to-many
Annotations with names corresponding to the type of relationship multiplicity decorate the appropriate fields, properties, or methods that either specify or utilize a relationship to access the persistent objects. You will learn more about relationships in the context of queries, which you will come across later in this section.

Apart from multiplicity, the relationship between objects also has a sense of directionality. The directionality value can be:

- Unidirectional
- Bidirectional

In the earlier discussion on the object relational mismatch, I had pointed to relationship directionality in the subsection that focused on entity relationships. There, I mentioned that, unlike relational tables, which always set up a bidirectional relationship, objects need to specify a relational pointer on both entities if you desire the relationship to be bidirectional.

For bidirectional relationships, an owning side and an inverse side of the relationship are identified. This helps manage the data integrity during any data modification. The inverse side specifies the owning side with the `mappedBy` element. The `mappedBy` element is used with the `@ManyToOne`, `@OneToMany`, and `@ManyToMany` relational annotations. `@ManyToOne` annotation does not define the `mappedBy` element, as the many side always owns the relationship.

The relationship annotations define a number of elements that affect their behavior. `mappedBy` is one such element. Another important element is `cascade`. For those who are conversant with SQL, this word may immediately remind them of the “cascade delete” operations, which deletes all dependent entities when a primary entity is deleted. JPA relationship annotations define five valid `CascadeType` enumerations as values, which are:

- **ALL** — Cascade all operations.
- **MERGE** — Cascade merge operations. Merge operations occur when regular objects are merged back as managed entities.
- **PERSIST** — Cascade persist or “save to data store” operations.
- **REFRESH** — Cascade synchronization and refresh operations.
- **REMOVE** — Cascade remove or delete operations.

For example, a `@OneToMany` annotation could specify a “cascade persist” preference so that all related entities are persisted as the primary or owning entity is persisted to the data store. To illustrate this, consider a customer entity that has multiple addresses. If this customer entity defines a “cascade persist” relationship to the set of addresses it owns, then on persisting a customer entity, its related addresses are also persisted. The definition of such a cascade relationship might look like this:

```java
public class Customer {
    private int customerID;
    private String name;
    private String email;
```
private Set<Address> addresses;
//additional fields and properties

//accessor methods

@OneToMany(cascade=PERSIST, mappedBy="customer")
public Set<Address> getAddresses() {
    return addresses;
}

}

public class Address {

    private String street;
    private String state;
    private String zip;
    private String country;

    //accessor methods

}

With entities defined and their relationships established, you are ready to start interacting with a persistence context.

**Interacting with a Persistence Context**

An EntityManager defines the API to interact with a persistence context. How you get hold of an EntityManager depends on the way it’s configured. If you choose to go with container-managed entity manager, then the container manages the lifecycle of the entity manager. All you have to do then is declare a variable of the EntityManager type in your application component and annotate it with the @PersistenceContext annotation. The container injects an EntityManager instance into this variable.

In contrast, going with application-managed entity manager implies that you have to instantiate the entity manager yourself. You first inject an entity manager factory, using the @PersistenceUnit annotation, like this:

```
@PersistenceUnit EntityManagerFactory emf;
```

Then use the createEntityManager method of the EntityManagerFactory to create an instance of an EntityManager.

In either case, once you get hold of an entity manager instance, you are ready to interact with the persistent entities in that context.

Persistent objects can be in any of the following four states:

- **New** — Has a persistence identity but is not (yet) attached to a persistence context
- **Managed** — Has a persistence identity and is attached to a persistence context
Chapter 7: Leveraging JPA and Hibernate with Flex

- **Detached** — Has a persistence identity but is not attached to a persistence context (anymore)
- **Removed** — Has a persistence identity and is attached to a persistence context but is marked for removal

You use the entity manager to interact with entities to carry out any of the following operations:
- Create
- Read
- Update
- Delete

In addition, you would likely query a particular data store and synchronize the entities between your persistence context and the data store.

Persistence operations are best described with the help of an example. To keep things simple and focus on the operations themselves, I chose an elementary application that catalogs open source software products and keeps a record of its contributors.

I created the example application as a standalone JPA application, which gets bundled as a JAR. JPA works in the context of both a simple web application and an enterprise Java EE application. It also works in Java SE environments. In any of these three target environments, you could use the created JPA example application JAR and leverage the managed entities within that setup.

For this JPA example, the following pieces of software are used:

- **MySQL 5.2.16 Community Edition** — The data store
- **Hibernate EntityManager 3.4.0.GA and associated libraries** — The JPA implementation (see the note on Hibernate EntityManager distribution to learn more about the associated libraries)
- **Eclipse Ganymede 3.4.1 with support for JPA projects** — The IDE (Support for JPA projects is built-in and bundled as a part of the Java EE support in Eclipse Ganymede 3.4.1)
- **MySQL Connector/J 5.1.7** — The JDBC driver
- **Apache-Tomcat 6.0.18** — The target Servlet container (application server)
- **JRE 6** — The Java runtime environment

You will be able to replace many of these listed software products and libraries with their appropriate substitutes. For example, you could use IntelliJ IDEA instead of Eclipse as the IDE and deploy the web application in a JBoss application server as opposed to Tomcat. Also, if you do not want to take the overhead of administering a database, then you could go with the embedded HSQLDB database.

---

**Hibernate EntityManager 3.4.0.GA Distribution**

Hibernate EntityManager is a Hibernate-based JPA implementation for Java SE and Java EE environments. The current general availability release version of Hibernate EntityManager is 3.4.0.GA. You can download a copy of the software from www.hibernate.org/6.html.
The software is packaged and distributed in `tar.gz` and `.zip` archive file formats. When you unpack the distribution to a folder on the file system, you will find `hibernate-entitymanager.jar` at the root of a newly created subfolder called `hibernate-entitymanager-3.4.0.GA`. This jar file is the primary JPA implementation library.

The main JPA implementation jar file requires a few additional archive files to function properly. The archive files are all available in the `lib` directory of the expanded subfolder. The jar files in the `lib` directory are as follows:

- `dom4j.jar`
- `ejb3-persistence.jar`
- `hibernate-annotations.jar`
- `hibernate-commons-annotations.jar`
- `hibernate-core.jar`
- `javassist.jar`
- `jta.jar`
- `slf4j-api.jar`

Add all these jar files to your classpath. In Eclipse, add them as referenced library files.

### Set Up the Database

As a first step install MySQL. When configuring the instance, remember the administrator password that you set. You will use it shortly to log in and create a database. Once installed, start and log in to the instance, using the command line prompt. Then create a new database. I name my database `oss_catalog_db` as the application maintains a catalog of Open Source Software (OSS) products and its contributors. The command to create a database, named, `oss_catalog_db`, in MySQL is:

```sql
create database oss_catalog_db;
```

Don’t create the tables yet. The persistence infrastructure will generate them on the basis of entity definitions and the object/relational mapping metadata. Next, create an Eclipse project so that you can start programming your application.

### Create an Eclipse JPA project

Start up the Eclipse IDE, which I assume is installed and set up. Create a new JPA application project. Figure 7-1 to 7-9 show screenshots of the Eclipse JPA Application Project Creation Wizard.

Once you choose to create a JPA project, as illustrated in Figure 7-1, you will be prompted, as shown in Figure 7-2, to specify a name for the project, a target runtime, and some basic configuration. All configured server runtime environments are listed in the target runtime drop-down list. I selected Apache
Tomcat as the target runtime. As a result of this selection, the option to be a member of an enterprise Java archive file, an EAR file, is grayed out. Apache Tomcat does not support EAR file formats.

Figure 7-1

As you move to the next screen in the wizard, you come to the screen that defines the JPA facets. Figure 7-3 shows the screen when no information has been filled yet, and Figure 7-8 shows the same screen after all the necessary information is provided.

Database connection and JPA implementation are the two most important JPA facets. You can add a new database connection or use an existing one. I added a new MySQL named connection. Figure 7-4 depicts the screen where the database credentials are provided.
Then a JPA Implementation is configured. Look at Figure 7-5 to Figure 7-7 to view the steps involved in creating a Hibernate EntityManager-based JPA Implementation configuration.

Finally, you are ready with a configured set of JPA facets, as shown in Figure 7-8. The next and last screen in the wizard completes the process and creates an Eclipse Ganymede JPA project.

After creating a new project, add the following JAR files to your project:

- dom4j.jar
- ejb3-persistence.jar
- hibernate-annotations.jar
Once the external JAR files are added, your Java Build Path should be as illustrated in Figure 7-9.

Next, the entities and the persistence configuration need to be created.
Figure 7-7

Figure 7-8
Program the Entities and specify metadata

The sample JPA application involves only two entities: OSSProduct and Contributor. The source code for the OSSProduct class, stripped of all the comments and import statements, is available in Listing 7-1. The source for the Contributor class is available in code download available for the book.

**Listing 7-1: OSSProduct.java**

```java
package problazeds.ch07;
//import statements
@SuppressWarnings("unused")
@Entity
@Table(name = "ossproducts")
@NamedQueries({
    @NamedQuery(name = "ossproducts.findAll", query = "from OSSProduct"),
    @NamedQuery(name = "ossproducts.findById", query = "Select ossp from OSSProduct ossp
" +
    "where ossp.oSSProductId = :oSSProductId") })
public class OSSProduct {
    public OSSProduct() {
    }

    public int getOSSProductId() {
        return oSSProductId;
    }

    public void setOSSProductId(int productID) {
    }
}
```

Continued
The `OSSProduct` class listed here is decorated with a large number of annotations. These annotations act as the necessary and valuable metadata for JPA to appropriately persist and manage the entity.
Some of the most important set of annotations look like this:

- @Entity: Marks the class as an JPA entity.
- @Table: Maps the class to the underlying table.
- @NamedQuery: Defines a named query. There can be two types of queries:
  - Dynamic — Those that are generated on the fly, which are analogous to ad hoc SQL statements.
  - Static — Named queries, which are analogous to Java prepared statements or database stored procedures.
- @ManyToMany — Defines a many-to-many relationship between OSSProduct and Contributor in the example. In the listing above, the many-to-many annotation has two attributes:
  - fetch — The fetch attribute defines the fetch strategy, which can be eager or lazy.
  - cascade — The cascade attribute defines how the operations should cascade from the owning entity to the related entities.
- @Id — Marks a field or a property as a primary key.
- @GeneratedValue — Defines the primary key generation logic.
- @Column — Maps a field or property to the table column name.

JPA allows you to query entities and their collections using the JPA query language, which resembles SQL in its style and structure. For example, from OSSProduct, gives you a list of all OSSProduct entities in the data store. This query is already defined as aNamedQuery, so getting a list of all OSSProduct(s) simply involves invoking this query as follows:

```java
Query findAllQuery = em.createNamedQuery("ossproducts.findAll");
List<OSSProduct> ossproducts = findAllQuery.getResultList();
```

Each set of managed JPA entities forms a persistence unit. If the application is bundled as a JAR, a configuration file called persistence.xml, which resides in the META-INF directory, specifies the core configuration parameters for a JPA implementation. The configuration parameters are specific to a persistence unit. The contents of the persistence.xml file for the example application are shown in Listing 7-2.

### Listing 7-2: persistence.xml for the Sample Application

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <persistence-unit name="oss_catalog_db">
    <properties>
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
    </properties>
  </persistence-unit>
</persistence>
```

Continued
This JPA application that serves as the persistence layer for the OSSProduct and Contributor entities can be made part of a Java web application. Entities can be queried using named queries or ad hoc JPA QL queries. A small snippet for calling a named query that fetches all the OSSProduct(s) was exhibited a couple of paragraphs back.

You could access, modify, and save an OSSProduct property, say licenseType, for a particular instance that represents a row in the database table, using a code like this:

```java
@PersistenceContext
EntityManager em;
...
public OSSProduct modifyOSSProductLicense(OSSProduct ossproduct, String licenseType) {
    ossproduct.setLicenseType(licenseType);
    em.persist(ossproduct);
    return ossproduct;
}
```

Although a discussion on JPA could go a lot further than our coverage so far, I will stop right here. The objective of this section was to introduce JPA, and that goal has been accomplished.

Next, I will introduce the essentials of Hibernate. The core Hibernate object/relational mapping and persistence framework has been around longer than the JPA specification. In addition, it is a major source of inspiration for JPA.

**Bare Essentials of Hibernate**

In this section, I will rapidly and only tersely introduce the features of the Hibernate object/relational mapping and persistence framework. I will not talk about Hibernate-EntityManager, which as a JPA implementation serves to bridge the minor gap in syntax and semantics between JPA and Hibernate.

Hibernate is feature-rich compared to the lightweight JPA, but they share the same core concepts. This is no surprise, as the lead developers of Hibernate created and influenced the JPA specification. In some
sense, Hibernate’s simplicity and elegance were brought into the Java persistence architecture as a much needed relief to the beleaguered world of the legacy entity beans.

A Hibernate entity is a Plain Old Java Object (POJO) that declares a no-argument constructor. The no-argument constructor allows instantiation of the POJO using reflection. In addition, it’s recommended that entities use Java bean–style getter and setter methods for its properties and declare its properties private.

Entities can define metadata using annotations or external XML configuration files. For the most part, Hibernate entity annotations are the same as JPA annotations. They both utilize Java 5 annotations. Hibernate Annotations, which is available as a separate download, maps the Java 5 and JPA annotations to the Hibernate metadata. XML configuration files are more traditional and have existed from the first few versions of Hibernate. By convention, entity metadata XML files follow a .hbm.xml file extension naming scheme.

Following convention, the configuration file for OSSProduct.java would be OSSProduct.hbm.xml. This XML file would reside in the same directory where OSSProduct.java resides. If you were to create a Hibernate entity that was the same as the OSSProduct JPA entity that I listed in the last section and use an XML configuration file to wire up its metadata, then you would likely see a configuration file (OSSProduct.hbm.xml) that contains the code shown in Listing 7-3.

**Listing 7-3: OSSProduct.hbm.xml**

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="problazeds.ch07.OSSProduct" table="OSSPRODUCTS">
    <id name="oSSProductId" column="OSSPRODUCT_ID">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <property name="licenseType" column="LICENSE_TYPE"/>
    <set name="contributors" table="OSSPRODUCT_CONTRIBUTOR">
      <key column="OSSPRODUCT_ID"/>
      <many-to-many column="CONTRIBUTOR_ID" class="problazeds.ch07.Contributor"/>
    </set>
  </class>
</hibernate-mapping>
```

You could create the Contributor entity along the same lines as the OSSProduct entity. Once the entities and their configuration files are ready, you can configure Hibernate itself. Hibernate can be configured using any of the following three methods:

- Using hibernate.properties
- Using hibernate.cfg.xml
- Programatically
Chapter 7: Leveraging JPA and Hibernate with Flex

Using hibernate.cfg.xml, the configuration for an application that maintains a record of open source software products and its contributors will turn out as shown in Listing 7-4. I am intentionally reusing the same example I used to illustrate JPA. This will allow you to compare and contrast the two.

**Listing 7-4: hibernate.cfg.xml**

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost/oss_catalog_db</property>
    <property name="connection.username">root</property>
    <property name="connection.password">specify the password here</property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto">create</property>

    <mapping resource="probkazed/ch07/OSSProduct.hbm.xml"/>
    <mapping resource="probkazed/ch07/Contributor.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Now that the entities are created and configured, you could compile it; after which, you will be able to load and store objects from and to the data store.

In JPA, the **EntityManager** is the coordinator that interacts with loads, and stores entities, from a transactional perspective. In Hibernate, where this concept originated, the coordinator and context manager that loads and stores data is a **Session**. You can create a **Session** using a **SessionFactory**. A session
Chapter 7: Leveraging JPA and Hibernate with Flex

factory is typically an application-wide singleton. A session factory is either created by a utility class as a static singleton or registered with and accessed from a JNDI repository.

If you want to change the licenseType of an OSSProduct instance, you can use a session, within which you can carry out this operation as follows:

```java
session.beginTransaction();
OSSProduct ossproduct = new OSSProduct();
ossproduct.setLicenseType(newLicenseType);
session.save(ossproduct);
session.getTransaction().commit();
```

As mentioned earlier, the session itself would be built out of the session factory.

At this stage, the rudimentary example is wired up and ready to run using Hibernate. However, all of what I have discussed so far is just the tip of a giant iceberg. There is a lot more in Hibernate, including support for complex transactions, different types of collections, sophisticated mapping strategies, a query language, interceptors, and whole lot of other features and utilities.


The focus of this chapter is on integrating Flex and JPA/Hibernate using BlazeDS. Now, with some information on JPA and Hibernate itself, you should be ready to understand the nuances of this integration.

A Rudimentary Approach to Integration with JPA and Hibernate

Let me start with an effortless but a limited way to integrate Flex and JPA/Hibernate. This time, the sample application that maintains a list of open source software products and its contributors will have a Flex interface.

Setting Up

To create this project, I leverage the facility to create a joint Flex and Java WTP project in Eclipse. Here are the steps that you need to follow:

- Using the Flash Builder plug-in for Eclipse-with WTP, first create a new Flex Project. You will be prompted with a screen as in Figure 7-10.

Give your project a name. I call it FlexOSSCatalog. Choose a J2EE server type and opt to create a joint Java/Flex project using WTP. Also select a LifeCycle Data Services (LCDS) remote object service type. Although it reads LCDS, the settings work just fine for BlazeDS as well.

- On clicking Next, a new screen is presented to you. This new screen allows inputs for J2EE settings, compilation options, and the output directory. Figure 7-11 shows the details. I selected JBoss AS 4.2 as the deployment environment and point to blazeds.war version 3.2.0.3978 archive file as the Flex WAR file.
Chapter 7: Leveraging JPA and Hibernate with Flex

Figure 7-10

Figure 7-11
Finally edit and confirm the source folder name, main application filename, and output folder URL, and click the Finish button to create the project. Figure 7-12 depicts what this last Project Creation Wizard screen looks like.

On creation, your project structure should have the following files and folders at its root:

- `.settings` — Eclipse project-specific configuration.
- `bin-debug` — Compiled Flex application resides here.
- `flex_libs` — Folder for referenced `.swc` files.
- `flex_src` — Flex source folder.
- `html-template` — Folder that contains wrappers for the Flex application.
- `src` — Java source folder.
- `WebContent` — Folder that contains the content that finally gets packaged as a WAR file and gets deployed in an application server or a Servlet container.
- `.actionScriptProperties` — ActionScript 3 (AS3) properties for the Flex SWF and SWC compilers, namely `mxmlc` and `compc`.
- `.classpath` — The classpath that specifies the Eclipse modules on the Java classpath.
- `.flexProperties` — Flex project-specific properties. Open this file and change the `serverContextRoot="/WebContent"` to `serverContextRoot="/<Your project name>"`, which in our case becomes `serverContextRoot="/FlexOSSCatalog"`.
- `.project` — Eclipse project properties file.
Chapter 7: Leveraging JPA and Hibernate with Flex

A MySQL instance and a database called oss_catalog_db were created earlier in the section titled "Introducing JPA." You can reuse the same database. Remember to start the instance before you start interacting with it.

In addition, it's assumed here that:

- The required JPA and Hibernate JAR files are downloaded and available.
- A JBoss AS 4.2 server environment is preconfigured in Eclipse.

All of these were discussed in the section titled "Introducing JPA." You may want to look back at that section if you are unsure about the steps involved.

At this stage, add a few additional JAR files to your project classpath. This can be done by adding them to the WebContent/WEB-INF/lib folder. These JAR files must include the following:

- Hibernate JPA implementation
- A log4j-based logging facility
- The MySQL JDBC driver

The list of additional JAR files, beyond the ones added by BlazeDS, is:

- hibernate-entitymanager.jar (part of Hibernate-EntityManager GA 3.4.0.GA, abbreviated as HEM 3.4.0.GA for the rest of this list. It can be downloaded from www.hibernate.org/6.html).
- dom4j.jar (lib folder of HEM 3.4.0.GA).
- ejb3-persistence.jar (lib folder of HEM 3.4.0.GA).
- hibernate-annotations.jar (lib folder of HEM 3.4.0.GA).
- hibernate-commons-annotations.jar (lib folder of HEM 3.4.0.GA).
- hibernate-core.jar (lib folder of HEM 3.4.0.GA).
- javassist.jar (lib folder of HEM 3.4.0.GA).
- jta.jar (lib folder of HEM 3.4.0.GA).
- slf4j-api.jar (lib folder of HEM 3.4.0.GA).
- hibernate-validator.jar (part of Hibernate-Validator 3.1.0.GA. It can be downloaded from www.hibernate.org/6.html).
- log4j.jar (The latest version of Apache log4j is 1.2.15. Can be downloaded from http://logging.apache.org/log4j/1.2/download.html).
- slf4j-log4j12.jar (a simple logging facade for Java; more details are available at www.slf4j.org).

At this stage most of the initial setup is complete. Let's start programming the Flex and JPA/Hibernate application.
Chapter 7: Leveraging JPA and Hibernate with Flex

Programming the Flex and JPA/Hibernate Application

As a first step, create the persistence.xml file and copy it over to a folder called META-INF within the Java src folder. For the first time, you will need to create the META-INF folder. The contents of persistence.xml can be the same as shown earlier in Listing 7-2.

In the Java source folder, create the entities: OSSProduct and Contributor. The classes can be the same as defined in the section titled “Introducing JPA.” The OSSProduct class is also illustrated in Listing 7-1.

At this stage, your project should look like the screenshot shown in Figure 7-13.

![Figure 7-13](image)

Next, create a service class that helps connect Flex to the persistence layer. Listing 7-5 includes the contents of this service class. Then configure the service class as a remote object destination. Configuring the service class as a remote destination is straightforward. Hopefully, it is something that you are familiar with by now. Just add the following lines to flex/remoting-config.xml:

```xml
<destination id="OSSProductService">
  <properties>
    <source>problazeds.ch07.OSSProductService</source>
  </properties>
</destination>
```

The remoting-config.xml, which is part of services-config.xml, has an AMF-based channel and a Java adapter defined by default. The configured OSSProductService destination uses the default AMF channel and the Java adapter.

Listing 7-5: OSSProductService.java

```java
package problazeds.ch07;

import java.util.List;
```

Continued
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class OSSProductService
{
    public OSSProductService() {
    }

    public List<OSSProduct> getOSSProducts() {
        EntityManagerFactory entityManagerFactory =
                Persistence.createEntityManagerFactory(FlexOSSCatalogConstants.
PERSISTENCE_UNIT);

        EntityManager em = entityManagerFactory.createEntityManager();

        Query findAllQuery = em.createNamedQuery("ossproducts.findAll");
        List<OSSProduct> ossproducts = findAllQuery.getResultList();

        return ossproducts;
    }

    public void createOrUpdateOSSProduct(OSSProduct oSSProduct) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory(FlexOSSCatalogConstants.
PERSISTENCE_UNIT);

        EntityManager em = emf.createEntityManager();

        if (oSSProduct.getOSSProductId() == 0) {
            oSSProduct.setOSSProductId(0);
        }

        EntityTransaction etx = em.getTransaction();
        etx.begin();

        try {
            em.merge(oSSProduct);
            etx.commit();
        } catch (Exception e) {
            etx.rollback();
        } finally {
            em.close();
        }
    }

    public void deleteOSSProduct(int oSSProductId){

Chapter 7: Leveraging JPA and Hibernate with Flex

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory(FlexOSSCatalogConstants.PERSISTENCE_UNIT);
EntityManager em = emf.createEntityManager();
Query findByIdQuery = em.createNamedQuery("ossproducts.findById");
findByIdQuery.setParameter("oSSProductId", oSSProductId);
OSSProduct oSSProduct = (OSSProduct)findByIdQuery.getSingleResult();

if (oSSProduct != null) {
    EntityTransaction etx = em.getTransaction();
    etx.begin();
    try {
        em.remove(oSSProduct);
        etx.commit();
    } catch (Exception e) {
        etx.rollback();
    } finally {
        em.close();
    }
}
```

The JPA/Hibernate code is ready and is wired up for consumption by a Flex client. Last of all, let’s create a simple Flex client to interact with it.

In this Flex client, the main view contains a pair of data grids. The main or parent data grid contains the collection of OSS products. An associated child data grid contains the collection of contributors, each time making visible only those that are related to the selected OSS product in the parent data grid.

In addition to the two data grids, there is a view that brings up a form to enter details of an OSS product. This provides a way to edit OSS products and add them to the data store.

The sources for the two main data grids are:

```xml
<mx:DataGrid id="parentDataGrid"
   dataProvider="{oSSProducts}"
    rowCount="8"
    editable="true"
    resizableColumns="true"
    itemClick="itemClickEventHandler(event);"/>
<mx:columns>
    <mx:DataGridColumn headerText="Name" dataField="name" />
    <mx:DataGridColumn headerText="License Type" dataField="licenseType" />
</mx:columns>
</mx:DataGrid>

<mx:DataGrid id="childDataGrid"
    dataProvider="{}"
    rowCount="8"
```
The source for the form is:

```xml
<mx:Form width="100%" height="80%" id="ossProductsForm">
  <mx:FormItem label="Name:" id="name_form">
    <mx:TextInput id="name_TextInput" text="" />
  </mx:FormItem>
  <mx:FormItem label="License Type:" id="license_type_form">
    <mx:TextInput id="licenseType_TextInput" text="" />
  </mx:FormItem>
</mx:Form>
```

All of the four CRUD operations (create, read, update, and delete) can be accomplished with the help of functions that invoke the corresponding server-side service methods via a remote object handle. The AS3 functions in `scripts/flexOSSCatalogScript.as` are as follows:

```as3
private function loadAllOSSProducts():void {
    remoteOSSPService.getOSSProducts();
}

private function deleteOSSProduct():void {
    if(parentDataGrid.selectedItem != null) {
        var selectedItem:OSSProduct = parentDataGrid.selectedItem as OSSProduct;
        remoteOSSPService.deleteOSSProduct(selectedItem.oSSProductId);
    }
}

private function addOSSProduct():void {
    applicationScreens.selectedChild = update;
}
private function updateOSSProduct():void {
    applicationScreens.selectedChild = update;
    if(parentDataGrid.selectedItem != null) {
        var selectedItem:OSSProduct = parentDataGrid.selectedItem as OSSProduct;
        name_TextInput.text = selectedItem.name;
        licenseType_TextInput.text = selectedItem.licenseType;
    }
}
```

The main view with the two data grids and the second view with a form are put together in a `ViewStack`. Navigation is accomplished between the two views as a result of appropriate button clicks, whose click handlers select the appropriate `ViewStack` child component.

All the communication between the Flex client and the persistence layer is accomplished through the remoting service. The remote object that acts as the proxy for the service on the client side defines result handlers for each of the service methods to handle the results appropriately. The source is:
When the results come back from the server-side service, the output is cast into appropriate strongly typed objects, which are counterparts of the server-side Java POJOs. This simple application has two such objects: OSSProduct and Contributor.

Build and deploy the application and you are up and ready with a Flex and JPA/Hibernate-based CRUD application. This was simple, and even if it involved a few steps, it didn’t seem extremely complex. Why then is the much talked about lack of off-the-shelf support for JPA and Hibernate with BlazeDS such a problem? This solution did not involve any special adapters or such!

The answer to this question is not a tricky puzzle. However, instead of stating it right up front, let me ask you a question: if you were to load a large number of related entities and hardly navigate through the entire interconnected network, which is probably often the case, would you still want to load everything eagerly? Probably not; after all, you desire good performance and resource optimization.

With this naive approach, you do not leverage the lazy loading and effective relationship management features of JPA and Hibernate. Also, you set yourself up for possible exceptions, which are thrown when there are attempts to access referenced objects that are still not initialized. In the next section, I enlist some of these and put together an expected set of features for Flex and JPA or Hibernate integration.

**Flex and Hibernate Integration Expectations**

This section focuses on what should be expected when integrating Flex and JPA or Hibernate and not what is necessarily present in the adapters today. It attempts to put together a set of requirements that could be used as the starting point to evaluate adapters and solutions that make Flex and JPA/Hibernate work together smoothly.

For every solution that integrates Flex and JPA or Hibernate, you should consider supporting the following:

- Lazy loading
- Incremental updating
- Adherence to standard JPA and Hibernate semantics
- Managed CRUD operations
- Minimal dependency
- Non-invasive value addition
Lazy loading is by far the most important of the list of expectations. JPA and Hibernate allow you to lazily fetch and load referenced entities, providing performance gains and optimal resource utilization. Say that a person has many friends, where each friend is a person. Logically, each friend, who is a person, will have friends and that way the connected list could go on to an infinite depth. There is popular belief that any two people are only six such links apart. So, let’s assume that six links is the maximum depth it can descend to. Now if an initial list of the 100 most connected people is fetched and you desire to have access to their extended network, you have two simple choices in hand:

- Either you eagerly fetch the 100 people and their entire networks up to six levels of depth, or
- You fetch the 100 people only and do not load anything from their networks. You only send along handles or references that can be used to walk the interconnected network and fetch the required data later when needed.

It doesn’t take more than a couple of minutes of thinking to go with the second choice to load connected objects lazily. You know that fetching immense amounts of data up front creates a performance drag and eats up more memory than should be used up. Also, it’s almost certain that many interconnected network branches will not be walked at all.

Once a collection is loaded, leveraging the lazy optimization, a user manipulates this data on the client, which in the case of Flex is outside the JVM. Say that a user plays with the list of 100 person objects from the earlier illustration and modifies 5 of them. Again, you have the choice to either send the entire collection back or just send the modified set. Going with the modified set seems prudent, but this imposes its own complexity. If you send only modified sets, you need to a have reference to the original set. At least copies of the original elements that were modified will be required at the time you reconcile the changed entities at the persistence layer.

When you send the changes back to the persistence layer, you need to make sure that the entities are in a form and style that JPA and Hibernate can consume and work with. That’s theoretically not asking for much, as JPA and Hibernate impose few restrictions on what entities should be like. Often, associated metadata or a few annotations are enough for the persistence layer to manipulate these entities. Java bean–style getter and setter methods and private properties are recommended, but even that’s not a requirement. The only requirement is to have a no-argument constructor so that reflection could be used to instantiate these objects. Therefore, it’s desirable that the adapter not impose restrictions beyond the simple standard ones either. You will learn later in this chapter that this problem, which appears benign at first, occurs quite commonly in adapters, creating oddities and restrictions on leveraging existing persistence layers.

Many Flex applications that interact with persistence layers indulge in CRUD operations. Developers tend to reinvent the wheel every time they have a new application, even when their interactions are restricted to CRUD. Why then should adapters and assemblers not support this natively and provide ready to use convenience methods? Although some believe that adding CRUD isn’t an adapter’s responsibility, it is still a valuable feature to have and could be supported through an independent but related plug-in. In a section titled “Managed Entities for Flex Applications,” a few aspects of CRUD and managed entities are brought up.

Adding managed entity features and CRUD could ask for a client side API to provide a standard interface at the Flex end. While this may become a necessity for consistent behavior, adapters should impose as minimal as possible dependency on the client-side code. If a plain vanilla adapter that only sends up collections is asking you to use a specific client-side library and API, then I would recommend looking it
over very carefully. Remember that every bit of extra API-based restriction makes your code brittle as far as refactoring, maintenance, and extension is concerned.

In all the cases listed above and others, the general expectation should be that, as far as possible, an adapter is non-invasive and does not break the standard contract either of the server-side persistence layer or of the client-side application. The task of the adapter is to extend the scope of the persistent entity beyond the JVM, and all it should focus on is the transition in form as entities move back and forth between the two layers.

In the next section, some of the ideas presented here are analyzed in the context of implementation.

**BlazeDS Adapter for JPA and Hibernate**

This section explains facets of a JPA and Hibernate adapter for BlazeDS and walks through some of the core concepts involved in writing such an adapter.

Writing a JPA or Hibernate adapter involves two distinct tasks:

- Writing to the BlazeDS API so that the adapter can hook onto the existing infrastructure
- Incorporating the JPA and Hibernate semantics so that the behavior is consistent and the benefits of JPA and Hibernate are leveraged

Chapter 9 explains the nuances of writing an adapter in detail, so I will not repeat the same content here. The only point worth mentioning is that a Hibernate adapter could best be written by extending the built-in JavaAdapter. If you want to learn more, then jump over to Chapter 9 and come back after reading it.

In this section, the focus is on understanding the complexities involved in incorporating the JPA and Hibernate semantics. JPA and Hibernate return persistent entities and their collections with built-in proxy, which as its name suggests sits in place of the real data. The proxy helps lazily load data sets and provides for efficient management of entities and their collections. These entities and their collections lie within a JVM. When you integrate these persistence layers into a Flex application, you expose these entities and their collections to a language and environment beyond what they are scoped for.

BlazeDS is capable of serializing and transforming Java-based objects to their AS3 counterparts so that you may wonder why taking these entities and their collections needs any special handling. This is where you will benefit from understanding how BlazeDS carries out the serialization and the translation between the two environments.

BlazeDS has a set of endpoints where a Flex application channel sends requests up to BlazeDS that resides within a Servlet container or an application server. Responses from BlazeDS follow the route back up from the endpoint to the channel. On endpoints that support translation and serialization between AS3 and Java (or even web services), a serialization filter is defined to intercept calls to the endpoint. When an incoming or outgoing message hits the filter, serialization and deserialization occur with the help of a MessageSerializer and a MessageDeserializer.

During serialization, the serializer eagerly fetches all the JPA and Hibernate persistent objects and sends them across the wire. The JPA and Hibernate proxies are replaced with the data that they stand in place...
of. This breaks the lazy loading semantics, rendering the idea of proxies useless. Therefore, any Hibernate
adapter needs to preserve the proxy characteristics while keeping the standard behavior of the essential
serialization and de-serialization mechanism between Flex and Java intact.

A common design pattern used to solve this problem is what I like to call the “Clone and Merge Transfer
Pattern,” CMTP for short. The fundamental tenets of CMTP are shown in Figure 7-14.

![Clone and Merge Transfer Pattern (CMTP)](image)

The cornerstones of this pattern are the clone and merge operations. On the way from the Java server
side to the Flex client, a persistent object with proxy-based references is cloned, and the clone is sent
to the serializer-deserializer, which in turn converts it into an AS3 object. This object reaches the Flex
client. In the reverse direction, an AS3 object, after being converted to Java, is merged back to the original
persistent object. One important requirement for the merge operation to work successfully is that you
keep a copy of the original object when it’s cloned. This copy can be maintained on the server or within
the class locally.
When the copy of the object is stored on the server, the HTTP session seems like the best place for it. Keeping an application scoped and shared areas may make the object visible beyond its rightful owners.

When the copy is maintained locally, you need to have a structure within the class for it. This is where you require entities to extend specific classes that accommodate for this structure. One could leverage Aspect-Oriented Programming (AOP) to induce such behavior at runtime and keep persistent objects from extending a specific class or have them adhere to a specific API. The sidebar provides a few links and references to AOP.

**Aspect-Oriented Programming (AOP)**

Aspect-oriented programming (AOP) is a programming paradigm that enhances modularity by enabling improved separation of concerns. Object-oriented programs are partitioned along distinct functions or concerns. Some of these concerns are abstracted as independent modules and objects, while a few others such as logging or security are cross-cutting in nature.

AOP encapsulates each concern independently into aspects and allows interception points within classes and methods, which it operates on. An aspect can modify the behavior of a class by applying an advice or additional behavior at various join points specified in an expression or query called a pointcut. An aspect can also make byte code compatible structural changes to classes. It can add members or parents to a class.

AspectJ is a leading AOP implementation for Java. Details can be accessed online at [www.eclipse.org/aspectj](http://www.eclipse.org/aspectj). If AOP is something you know little or nothing about then start by reading the 3 part JavaWorld article series titled: “I want my AOP!”

The links to the three-part article series are:


AOP is gaining popularity ever since it first appeared. The popular Spring Framework also supports AOP.

You will learn later that the Gilead Flex BlazeDS Hibernate adapter follows many of these enlisted principles. The “dsadapters” Hibernate adapter also embodies all these principles. In addition, as far as I know, the BlazeDS adapter from dsadapters is the only adapter that leverages AOP for runtime object manipulation.

As you will learn in Chapter 9, the `invoke` method of an adapter class is the place where all the adapter logic resides or is called from. A message sent to an adapter triggers a call to the `invoke` method.

In addition to appropriately handling lazy collections, an adapter may also provide services for data management, which is discussed in the section before the summary, later in this chapter.

Now you have a sense of how to write your own adapter, but you don’t necessarily need to jump into writing one of your own. There are a few open source options that you could use.
JPA and Hibernate Adapters for BlazeDS

Unlike LCDS, BlazeDS does not come with a built-in adapter or assembler for JPA or Hibernate. However, a few open source projects provide such an adapter. Two relatively old adapters that have at least been around for a few months are:

- **dpHibernate** — [http://code.google.com/p/dphibernate](http://code.google.com/p/dphibernate)
- **Gilead** — [http://gilead.sourceforge.net](http://gilead.sourceforge.net)

In addition to these the dsadapters project (accessible online at [http://code.google.com/p/dsadapters](http://code.google.com/p/dsadapters)), which I have championed, is adding a JPA and Hibernate adapter to its suite of BlazeDS extensions. Currently, it’s an early stage product, but a beta release should be available by the time you have this book in your hands.

A few other open source Flex and JPA or Hibernate integration options exist, but these products do not leverage BlazeDS and it’s not easy to structure these as BlazeDS adapters or assemblers.

The most prominent member of this alternative set is GraniteDS. GraniteDS is a full-suite replacement for LCDS and is a very robust and enterprise-ready tool. It uses the concept of custom externalizers to hook managed persistent objects like JPA and Hibernate entities. You can read more about GraniteDS at [www.graniteds.org](http://www.graniteds.org). The document that explains externalizers is accessible at [www.graniteds.org/confluence/display/DOC/2.+Externalizers](http://www.graniteds.org/confluence/display/DOC/2.+Externalizers).


In this section, I will cover dpHibernate and Gilead briefly, pointing mainly to their salient features, benefits, and shortcomings. I don’t discuss dsadapters here, although much of what is explained in the last section pertains to them.

Let’s start with dpHibernate.

**dpHibernate**

dpHibernate is an open source BlazeDS adapter for Hibernate. Over a year old now, this adapter supports lazy loading. The adapter library comes in two parts:

- An adapter, which is configured in BlazeDS
- A client-side library that one incorporates in one’s Flex application

The use of proxies is pervasive in this adapter. All lazily loaded Hibernate proxies are transformed to their AS3 proxy counterparts. On demand and at the time of access, the proxy is replaced with real data that is fetched from the server.

You can access more information on dpHibernate at [http://code.google.com/p/dphibernate](http://code.google.com/p/dphibernate). For now, I will show a simple example that leverages the library. This example should give you a flavor of what dpHibernate is like.

First, create a new Flex project. Opt for the joint Flex/Java project option. Download the dpHibernate server-side JAR and the client-side .swc files. You can download the files from
Add the JAR to the WEB-INF/lib folder and the .swc file to the flex_libs folder.

Configure the adapter in remoting-config.xml as follows:

```xml
<adapters>
  <adapter-definition id="hibernate-object" class="net.digitalprimates.persistence.hibernate.HibernateAdapter" default="true">
    <properties>
      <hibernate>
        <sessionFactory>
          <class>net.digitalprimates.persistence.hibernate.utils.HibernateUtil</class>
          <getCurrentSessionMethod>getCurrentSession</getCurrentSessionMethod>
        </sessionFactory>
      </hibernate>
    </properties>
  </adapter-definition>
</adapters>
```

Notice that the configuration for the adapter takes the name and method of a class that provides a handle to the current session. `dpHibernate` includes a `HibernateUtil` class that instantiates a Hibernate session for you. This session instance is bound to a particular thread using the `ThreadLocal` semantics. Thread-local variables differ from their normal counterparts in that each thread that accesses a thread-local variable (via its accessor methods, i.e. `get` or `set` method) has its own, independently initialized copy of the variable. `ThreadLocal` instances are typically private static fields in classes. Such classes typically associate state with a thread.

Next, you need to incorporate an Open Session in View solution to avoid the problems caused by typical web application scenarios where the business method completes and transactions are committed before the view’s rendering is complete. What happens then is that possibly unloaded collections or uninitiated proxies are invoked by the view-rendering program and that throws a `LazyInitializationException`.

To correct this problem, you can do one of the following:

- Include a Servlet filter that intercepts the calls and keeps the session open till the view is rendered. The database transaction is committed explicitly after the view-rendering phase.
- Break the single transactional session into two parts: complete the read and write activities to process the request and execute the business logic in the first unit of work. Then just read the data in the second unit of work for rendering the view. AOP can help accomplish such an interception scheme. The fundamentals of AOP are described in the “Aspect-Oriented Programming (AOP)” sidebar earlier in this chapter. The Spring framework has an AOP implementation that could be used in almost all application servers. Alternatively you could also leverage JBoss AOP or AspectJ.

You can read more about “Open Session in View” at the Hibernate wiki; the specific page is accessible online at [www.hibernate.org/43.html](http://www.hibernate.org/43.html).

dpHibernate supports the Servlet filter mechanism to keep the session open till the view rendering activities are done. It provides a built-in Servlet filter, called `HibernateSessionServletFilter`, for that purpose. You can use this built-in filter or write one of your own. The built-in Servlet filter that comes with dpHibernate is almost identical to the one proposed as a sample filter in the “Open Session In View” discussion on the Hibernate wiki.
Before you start programming, add the Servlet filter and set up a mapping in your `web.xml` instance as follows:

```xml
<filter>
  <filter-name>hibernateSessionFilter</filter-name>
  <filter-class>net.digitalprimates.persistence.hibernate.utils.filters.HibernateSessionServletFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>hibernateSessionFilter</filter-name>
  <url-pattern>/messagebroker/*</url-pattern>
</filter-mapping>
```

Finally, you are ready to create a sample application and see how `dpHibernate` works.

The Hibernate Reference documentation introduces Hibernate by using a sample application that manages events and persons who participate in these events. This same example also appears in the `project/tutorials` folder of the core Hibernate distribution. I re-create parts of this very example to see how all our available Hibernate adapters function.

The source for the `dpHibernate` specific example is available with the code download for the book, but let’s briefly walk through some of the key aspects of the example here.

First, you need to create your Hibernate entities and specify the metadata and mapping for these. This seems like a normal and a usual activity when it comes to Hibernate. However, now when you create your entities you have to extend them from the `dpHibernate`’s `HibernateProxy` class (fully qualified name: `net.digitalprimates.persistence.hibernate.proxy.HibernateProxy`).

Therefore, with `dpHibernate` an `Event` entity looks like this:

```java
package problazeds.ch07.model;

import java.util.Date;
import net.digitalprimates.persistence.hibernate.proxy.HibernateProxy

public class Event extends HibernateProxy {
    private Long id;
    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }
}
```
public Date getDate() {
    return date;
}

public void setDate(Date date) {
    this.date = date;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

Similarly, when you consume them in a Flex application across the wire, the counterpart bean type classes extend the dpHibernate’s HibernateBean AS3 class. Event.as looks like this:

```as
package problazeds.ch07.model
{
    import mx.collections.ArrayCollection;
    import net.digitalprimates.persistence.hibernate.HibernateBean;

    [RemoteClass(alias="problazeds.ch07.model.Event")]
    [Managed]
    public class User extends HibernateBean
    {
        public Long id;
        public String title;
        public Date date;
    }
}
```

Now, to load and store events you create a service class. The Hibernate tutorial has an EventManager class that implements this functionality. You could reuse that class. In addition, you will need a utility class to create a Hibernate session so that you can interact with it. dpHibernate recommends extending the service class from a dpHibernate class called HibernateService, which wraps the basic Hibernate functions. However, you are not limited to using the HibernateService class.

That is pretty much all that dpHibernate needs to function. You, of course, need to incorporate the additional business logic per your requirements.

At this stage, I may be ready to comment on dpHibernate’s approach. While, the adapter supports lazy loading with the help of proxies and seems to function properly for the cases I have tested, it pollutes the domain model by making Java server-side entities extend the HibernateProxy class and the AS3-side entities extend the HibernateBean class. Not only does this appear cumbersome, but in my opinion it also has some drawbacks, namely:

- Entities or domain models are bound to the dpHibernate-adapter-specific implementation and are rendered useless from a generic standpoint. This spoils the case for a common set
of entity and domain models that could be consumed across multiple clients and interfacing programs.

- Works as an anti-pattern as far as object orientation goes. Classes should only extend from their type and not helper wrappers, as in dpHibernate’s case.
- Makes it difficult to work with already existing Hibernate-based persistence layers because, obviously, those do not extend their entities with dpHibernate-specific classes.

In addition, the server and client side are very tightly coupled, using the dpHibernate approach. The dpHibernate adapter will not work for Flex applications that choose not to use the client-side dpHibernate library.

Still, it’s one of the early Hibernate adapters for BlazeDS and seems to have developed some positive momentum despite its serious shortcomings. In any case, the creators don’t claim the product to be robust and problem free and do deserve kudos for their effort.

Next, let’s explore Gilead and see how that adapter fares.

**Gilead**

Gilead, as a project, existed as “hibernate4gwt” for a while. From late 2008, it has included adapters for both GWT and BlazeDS (for Flex).

Our interest in this project is exclusively the BlazeDS Hibernate adapter, so that’s what I will talk about in this subsection. Gilead uses an approach similar to that illustrated in the last section, “BlazeDS Adapter for JPA and Hibernate.”

To explore Gilead-specific features, let’s download and configure the adapter and create a simple example application. To download the latest Gilead build, go to the project’s site on SourceForge: http://sourceforge.net/projects/gilead. You can also navigate to this website from the Gilead home page at http://gilead.sourceforge.net, which points to http://noon.gilead.free.fr/gilead.

Three download bundles are available from the Gilead SourceForge site, which are:

- **adapter4appengine**: Adapter for Google AppEngine
- **gilead**: Most of the Gilead source and binary distribution is bundled in this package. You need this download.
- **gilead_samples**: Sample applications. Includes a Flex BlazeDS stateless example. Consider downloading this, especially if you want to explore and understand Gilead further.

You will find the compiled JAR files in the `dist` folder within the Gilead package. Only two JAR files from the distribution (in the `dist` folder) are required to work as a Flex BlazeDS Hibernate adapter. These are:

- **adapter4blazeds-1.2.3.823**
- **adapter-core-1.2.3.823**
Chapter 7: Leveraging JPA and Hibernate with Flex

Now fire up Flash Builder IDE and create a new Flex project. Opt for the joint Java/Flex project and point to the blazeds.war file as the Flex WAR file. You are already familiar with these steps, so I will not walk through the details again.

Copy the Gilead Flex BlazeDS Hibernate adapter JAR files: adapter4blazeds-1.2.3.823.jar and adapter-core-1.2.3.823.jar to the WEB-INF/lib folder. Then configure the adapter in remoting-config.xml as follows:

```xml
<adapters>
  <adapter-definition id="persistent-adapter"
    class="net.sf.gilead.blazeds.adapter.PersistentAdapter" >
    <properties>
      <persistence-factory>
        <class>net.sf.gilead.sample.server.ApplicationContext</class>
        <singleton>true</singleton>
        <method>getSessionFactory</method>
      </persistence-factory>
    </properties>
  </adapter-definition>
</adapters>
```

Gilead also has a PersistentMessagingAdapter that extends the ActionScriptAdapter, a messaging adapter. In this example, only the PersistenceAdapter is used. The PersistentMessagingAdapter is not explored at all.

The persistence adapter needs to get hold of a Hibernate session or a JPA entity manager during the course of its operations. Hibernate sessions are created using a session factory. Entity managers are created in a similar fashion, using an entity manager factory.

The preceding adapter configuration shows one way to specify the session factory. The same sort of configuration also applies to an entity manager factory. Within the persistence-factory property the following elements are specified:

- **class** — The session factory or the entity manager factory class
- **singleton** — A Boolean flag, which when true, appends a call to getInstance() before a call to the factory method is made
- **method** — The factory method that instantiates or returns an existing session or entity manager instance

Alternatively, within Java EE environments you could leverage the persistence.xml configuration or the JNDI. Gilead allows for the additional elements, namely: entityManagerFactory and jndi, to be specified within the persistence-factory property. The entityManagerFactory elements allows you to point to the factory defined in persistence.xml.

As mentioned in the last subsection on dPHibernate, I’ve re-created parts of the basic Hibernate Reference documentation and tutorial example so that you can see how all the available Hibernate adapters function. This example manages events and persons, who participate in these events. In this subsection, the example is re-created using the Gilead Flex BlazeDS adapter.
Chapter 7: Leveraging JPA and Hibernate with Flex

The full source is available with the chapter’s code download. Snippets from the code appear with annotations and comments in the next few paragraphs.

First, you create the entities on the server side. This time you have a choice to either extend a specific class or just create a Java bean–style POJO. Choosing between the two has some implications, which was discussed in an earlier section titled: “BlazeDS Adapter for JPA and Hibernate.” At this juncture, you may benefit from rereading that section.

When you want to restrict persistent objects to Java bean–style POJOs only, you need to either save a copy of the class on the server before it’s cloned or use the dynamic proxy, available in Gilead. Gilead uses the HTTP session to save the object copy when you choose to go with a server-centric stateful way of saving this information.

However, for simplicity and to directly compare and contrast this adapter with dpHibernate, let’s go with the option to save the copy locally in the class. In that case, the Event Java class is:

```java
package problazeds.ch07.model;
import java.util.Date;
import net.sf.gilead.pojo.java5.LightEntity;
public class Event extends LightEntity implements Serializable {
    //private static final long serialVersionUID = <generate one>;
    private Long id;
    private String title;
    private Date date;
    public Event() {}
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
}
```
public void setTitle(String title) {
    this.title = title;
}
}

Again on the AS3 end, the bean class extends a specific class, which in this case is called LightEntity. The source for Event.as is:

```java
package problazeds.ch07.model {
    import mx.collections.ArrayCollection;
    import net.sf.gilead.pojo.actionscript.LightEntity;

    [RemoteClass(alias="problazeds.ch07.model.Event")]
    [Managed]
    public class User extends LightEntity {
        public Long id;
        public String title;
        public Date date;
    }
}
```

There are service classes on the server side to interface with the Event and the Person entities. I will cover none of that here. The source is available as a part of the code download, so you could surely grab that and explore the details.

Gilead supports lazy loading, and you have seen how it allows multiple ways of saving object copies locally and globally. When entities extend LightEntity, Gilead introduces the same drawbacks as dpHibernate does. Again it pollutes native Hibernate objects with special API restrictions and also makes it difficult to leverage existing Hibernate domain objects. Saving object copies in the HTTP session avoids such restrictions but introduces scalability issues if a large number of users simultaneously access many domain entities.

That brings us to the end of the brief survey of the dpHibernate and Gilead.

Next, we venture into the topic of data management, after which it will be time to wrap up as far as this chapter goes.

**Managed Entities for Flex Applications**

So far, the integration discussion has hovered around persistent objects transmission across the layers. Effective proxy management and lazy loading have been touted as the clear winning features. However, that may not be all that a complex business application desires. More often than not, what is required are managed data objects and entities.

Managed data sets are not new to Flex. LCDS supports them. A few open source Flex and Java projects also incorporate this idea in their approach to business problem solutions. However, there is still open debate on its applicability and its merits. Some in the community swear by it; others consider it an unnecessary overhead.
Chapter 7: Leveraging JPA and Hibernate with Flex

So, what exactly are managed entities and how do they differ from regular data access patterns in Flex? How can you include them in BlazeDS and where does JPA or Hibernate come in within this context? The answers to these questions is the content of this section.

Many a Flex application fetches a data collection from the server and allows CRUD operations on such a collection. In traditional hand-coded applications, each of the CRUD operations is passed down to the server and managed through custom service methods, which in turn call the persistence layer functions to make the modifications permanent. Sometimes, these service methods resort to invoking SQL statements on data stores and manipulating the result sets, which is certainly something that should be avoided in most situations.

With JPA and Hibernate, managing CRUD operations on the server side is easy. What you need is a service layer that provides a common API and a point of interception for the CRUD operation calls. LCDS implements data management using the concept of a ChangeObject, where it keeps track of the current or changed values and the previous value. Depending on the type of CRUD operation, appropriate action is taken and conflicts, if any, are managed. The ChangeObject interface can be looked up in the LCDS documents at http://livedocs.adobe.com/livecycle/es/sdkHelp/programmer/lcdsjavadoc/flex/data/ChangeObject.html. From the Flex client perspective, collections have a simple contract with only two main methods, commonly named fill and sync. The fill method reads or gets a collection and the sync method implements any of the CUD methods of the CRUD set. As of now, there is no fully integrated Java, BlazeDS, and Hibernate–based open source product that provides such a solution. Efforts have begun to create such a solution. The project is called Usable, and you can keep track of it at the initiative’s project site: http://code.google.com/p/usable.

The open source Clear Toolkit (http://sourceforge.net/projects/cleartoolkit/) is a possible alternative. It has a library of Flex components that includes a DataCollection class that supports Managed objects. It works with Flex, Java, and BlazeDS applications. Support for Hibernate on the server side can be manually hooked up if needed.

Another alternative for a managed CRUD solution using Flex and Hibernate with BlazeDS exists in the Grails Flex Scaffold (GFS) project. The project web site is www.grails.org/plugin/flex-scaffold. GFS is a grails plug-in that generates Flex scaffolding code and includes support for presentation and service layers by providing embedded data in your domain classes. It leverages BlazeDS.

Chapter 11 will revisit this topic and deal with it in detail. However, for now it’s time to wrap up and then move on to testing and debugging BlazeDS-based Flex applications.

Summary

Starting with the object/relational mismatch, this chapter covers a whole gamut of topics that relate to Flex and Java persistence. In a brief exposition, differences between the relational and object-oriented world were considered in the context of granularity, inheritance, polymorphism, relationships, and identity.

Next, the chapter covered the fundamentals of JPA, and Hibernate. JPA was introduced via a complete example. The example itself was rudimentary, but it brought many aspects of configuration, entity definition, annotations, and entity manager to light. If you were a newbie and knew little or nothing about JPA, then this section hopefully brought you up to speed. If you already knew a lot about JPA, then it might have served the purpose of a quick review. After JPA, Hibernate essentials were covered. The coverage
of Hibernate was minimal, and almost all of the content was restricted to the core Hibernate features. There was absolutely no discussion of the Hibernate validator, annotations, search, or shards.

Once the fundamentals of Java persistence were covered, the topic of Flex and JPA/Hibernate integration started. The topic started with the demonstration of an anti-pattern, which for some reason has seen some adoption among developers. An example illustrated that eagerly fetching all Hibernate records is not a wise approach. The example was complete in its own right and hopefully acted as a pointer for what might go wrong if you don’t use a specialized adapter.

After you were made aware of the problems and the anti-pattern, you came to a section that stated the Flex and Hibernate adapter expectations. This section served the dual purpose of acting as a wish list and a benchmark so that you could see how good the existing adapters are.

After elucidating the expectations, there was an illustration of the key aspects of creating a Flex and Hibernate adapter. The illustration was at a conceptual level and didn’t delve into actually writing a complete adapter. That might end up taking a few chapters, if not an entire book on its own.

Once you understood how to write an adapter on your own, you were presented with a couple of existing open source adapter options and made aware of an upcoming alternative. This is where you see fully working examples that support lazy loading and avoid any initialization exceptions.

Finally, there was short but important coverage of data management. The discussion wasn’t detailed enough for you to start using data management with Flex, Java, BlazeDS, and Hibernate. In fact, there isn’t an option yet from the open source stack that addresses this area. LCDS has robust support for data management and open source alternatives in GraniteDS Tide exist, but there isn’t anything there that leverages BlazeDS and Hibernate. However, it’s not all gloomy; work on such a piece of open source software has begun!
Developing an application is often synonymous with programming it. Despite all the emphasis on testing from the process experts, in most cases unit testing and documenting the application program is a retrospective, half-hearted activity. This is an unfortunate fact, because as soon as an application enters the alpha and beta launch phases, the absence of thorough prior proactive testing implies the beginning of a large phase of effort, often overshadowing the original programming effort.

Despite the fact that inappropriate proactive testing delays projects and increases work, convincing developers to adopt testing as a part of their repertoire is not easy. However, for those who see the merit and importance of testing, the tools and framework need to be available. This chapter talks about the testing, logging, and debugging tools available to build bug-free Flex and BlazeDS applications.

**Testing**

Testing Flex and BlazeDS applications involves testing applications both sides of the wire and also testing the connector in the middle. Flex and Java support unit testing frameworks that can automate and streamline the testing activity.

JUnit ([http://junit.sourceforge.net/](http://junit.sourceforge.net/)), which belongs to the xUnit testing framework family, provides an easy way to implement repeatable unit tests in Java. JUnit is not the only unit testing framework for Java. Alternatives like TestNG ([http://testng.org/](http://testng.org/)) exist. This chapter uses JUnit 4.7 to illustrate Java code testing strategies and methodology.

Like JUnit, FlexUnit ([http://opensource.adobe.com/wiki/display/flexunit/FlexUnit](http://opensource.adobe.com/wiki/display/flexunit/FlexUnit)) provides a way to implement repeatable tests for Flex and AS3 applications. The latest version, FlexUnit 4, is a major improvement over its original FlexUnit 1.0 release. The testing idioms used in FlexUnit and JUnit are similar.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

To explain testing strategies and the necessary tools, I first fork the discussion along two independent lines, following JUnit and FlexUnit, respectively. Once I explain each of these individually, I merge the discussion to consider testing client- and server-side technologies together. I will start with FlexUnit.

**FlexUnit**

FlexUnit 1.0 is the current stable release. FlexUnit 4 beta was launched in May 2009 and will be the next release candidate. FlexUnit has evolved and changed substantially from version 1.0 to version 4, so it is prudent to understand both the versions.

I'll first demonstrate the use of FlexUnit 1.0 through an example.

**A Simple FlexUnit 1.0 Example**

First, using Flash Builder or another application, create a new Flex project and add FlexUnit1Lib.swc as a referenced archive file. Once you have the project set up, create a simple AS3 class, called `NumericalValue`, that allows you to add and subtract a number to and from an initial value of 0. Then, create a class that tests the methods of the `NumericalValue` class. Finally, add the test class to a test suite and run it using a test harness.

`NumericalValue.as` is written like this:

```asciidoc
package test
{

public class NumericalValue
{

private var value:Number=0;

public function add(addedValue:Number){

    value = value + addedValue;

}

public function subtract(subtractedValue:Number){

    value = value - subtractedValue;

}

public function getValue():Number{

    return value;

}
}
}
```

Using FlexUnit version 1, define a test class by extending the framework `TestCase` class. Then, you define a number of methods in this class whose names begin with the word “test”. This is a requirement. Only methods starting with the words “test” are included by the runner. The test class for `NumericalValue` is called `NumericalValueTest`. A test class name is not required to follow any particular nomenclature system or pattern. By convention, such classes are named by appending “Test” to the name of the class they are trying to test. The source for `NumericalValueTest` is shown in Listing 8-1.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

Listing 8-1: NumericalValueTest Source

```java
package test {
    import flexunit.framework.TestCase;

    public class NumericalValueTest extends TestCase {
        public function testAdd():void {
            var numericalValue:NumericalValue = new NumericalValue();
            numericalValue.add(50);
            assertTrue("Value after adding 50 to the initial 0 value is 50",
                        numericalValue.getValue() == 50);
            numericalValue.add(50);
            assertEquals("Value after adding 50 to 50 is 100",
                         100, numericalValue.getValue());
        }

        public function testSubtract():void {
            var numericalValue:NumericalValue = new NumericalValue();
            numericalValue.subtract(100);
            assertTrue("Value after subtracting 100 from the initial 0 value is -100",
                        numericalValue.getValue() == -100);
        }
    }
}
```

Now to run the tests included in the test class, you instantiate a test suite and add the test class to a test suite, before running the test suite using a test harness. Following is a simple illustration of how you can set the test suite and the runner:

```xml
<?xml version="1.0" encoding="utf-8"?>
    xmlns:flexunit="flexunit.flexui.*"
    creationComplete="onCreationComplete()">

    <mx:Script>
        <![CDATA[
            import flexunit.framework.TestSuite;
            import test.NumericalValueTest;

            /**
             * Function called on creationComplete.
             * Set the test suite as the test property of the TestRunnerBase and then run it.
             */
            private function onCreationComplete():void {
```
Once you run this test runner Flex application, you will see a user interface with the test results, as shown in Figure 8-1.

Figure 8-1

That completes an elementary example, which uses FlexUnit 1. As mentioned earlier, a substantially modified and improved version of FlexUnit (version 4) is now available as a beta release. The next section covers FlexUnit 4.

Exploring FlexUnit 4

FlexUnit 4 is a thoroughly revised and improved version of FlexUnit 1. It includes the following features:

- Annotations or metadata decorations mark test cases and test suites. Requirements to extend specific classes are avoided.
- Multiple functions can be defined to help set up and tear down the environment or fixtures for testing for each individual test run.
- Static classes that are executed before any test run starts and after all of them are run can be included.
- Test cases that should not run need only to be annotated with the “Ignore” metadata term.
- Test cases can expect exceptions and assert success when such exceptions are thrown.
Hamcrest assertions, which are based on the idea of matchers, offer newer assertion types.

Support is available for multiple asynchronous events and asynchronous setup and teardown.

Multiple test runners can exist. The `RunWith` metadata term can associate test suites with specific runners.

Theories or insights can be created and assumptions can be checked against these theories.

Custom parameters can be passed into the testing framework and displayed later with an indication of their success or failure.

Sequences of user interface interactions can be run using the automated FlexUnit testing infrastructure.

It includes UIComponent testing façade to add and remove components to and from the display list. This serves as an extension point for integration and functional testing.

To show FlexUnit 4 features in action, I will re-create the example from the earlier section on FlexUnit 1, using the new framework.

As a first step, create a Flex project, using Flash Builder. Then copy the following library files to the `libs` folder of the project:

- FlexUnit1Lib.swc
- FlexUnit4.swc
- FlexUnit4UIRunner.swc
- Hamcrest-as3.swc

Next, create the `NumericalValue` class with its add and subtract methods, exactly as in the last section.

In the last section, the `NumericalValueTest` class was created immediately after this stage. Listing 8-1 includes the source for that class. In this second take on the example, you create the `NumericalValueTest` class again, but this time things are done a bit differently.

Before I highlight the differences, you may want to glance at the modified source of the `NumericalValueTest` class, which is shown in Listing 8-2.

### Listing 8-2: NumericalValueTest FlexUnit4 Source

```java
package test {
    import flexunit.framework.Assert;
    public class NumericalValueTest {
        [Test(order=2)]
        public function checkAdd():void {
            var numericalValue:NumericalValue = new NumericalValue();
            numericalValue.add(50);
            Assert.assertTrue("Value after adding 50 to the initial 0 value is 50",
            Continued
```
There are a few differences in the two versions of NumericalValueTest. Since both the classes implement identical functionality, comparing these two versions highlights the change in style and semantics between the old and the upcoming versions of FlexUnit. The things that are different are:

- With FlexUnit 4, the test class does not extend the framework TestCase class.
- In FlexUnit 1, test method names start with the word “test”. In FlexUnit 4, test method names do not need to have “test” prefixed to their names. Instead such test methods are annotated with the [Test] metadata.
- If you want the tests to run in a particular order, you can do so by assigning a value to the order attribute of the [Test] metadata.
- In FlexUnit 4, the assert statements are accessed as methods of the static Assert class. In the earlier version of FlexUnit, these methods were accessible as local methods because the test class inherited them by virtue of extending the TestCase class.

Now, to run the tests defined in the test class, NumericalValueTest, create a test suite and add the test class to it, and finally run the test suite using a test runner. There a few changes in the way how this is done in FlexUnit 4 as opposed to how it’s done in Flex Unit 1. Again, to understand the changes let’s first scan the test suite and test runner code. A test suite, FlexUnit4ExampleTestSuite, that I use to run the simple tests is shown here:

```java
package test
{
    [Suite]
    [RunWith("org.flexunit.runners.Suite")]
    public class FlexUnit4ExampleTestSuite
    {
        public var numericalValueTest:NumericalValueTest;
    }
}
```
Instead of extending `TestSuite` or creating an instance of a built-in framework `TestSuite`, as was done in FlexUnit 1, in FlexUnit 4 you merely annotate the test suite class with the `[Suite]` metadata. In addition, you specify a runner using the `[RunWith]` metadata.

FlexUnit 4 allows you to configure and use multiple runners to run your tests. A runner finds a test, executes it, and reports the results of such a test. By default, the FlexUnit 4 framework attempts to assign the most appropriate runner for a test class. However, you can override the default behavior and explicitly define a runner by using the `[RunWith]` metadata. A number of built-in runners come with the framework for the following:

- FlexUnit 1
- FlexUnit 4
- Fluint ([http://code.google.com/p/fluint/]()), an open source Flex testing framework that has inspired some of the new features in FlexUnit 4
- Structured Log Testing (SLT) ([http://structuredlogs.com/]()), a testing framework that uses structured log statements and visually advanced tools

In addition, the framework allows you to plug in a custom runner of your own. The API provides a way for you to write custom runners that leverage the framework and return the results using the same interface that built-in runners use.

The `org.flexunit.runners.Suite` runner used in the example instructs the framework to use an appropriate runner for each of the test classes.

Once a runner is defined, add a GUI as a test result listener and run the test suite like so:

```xml
<?xml version="1.0" encoding="utf-8"?>
creationComplete="creationCompleteHandler(event)">

<mx:Script>
<![CDATA[
import test.NumericalValueTest;
import test.FlexUnit4ExampleTestSuite;
import mx.events.FlexEvent;

import org.flexunit.listeners.UIManager;
import org.flexunit.runner.FlexUnitCore;

private var flexUnitCore:FlexUnitCore;

protected function creationCompleteHandler(event:FlexEvent):void {
    flexUnitCore = new FlexUnitCore();
    flexUnitCore.addListener( new UIManager( testRunner )
    flexUnitCore.run( FlexUnit4ExampleTestSuite );
}
]]>
<mx:Script>
```
When compared with FlexUnit 1, there are a few big changes in the way a test is run. Instead of directly starting the test run on a GUI test runner, the responsibilities of running tests and displaying the results are decoupled. A FlexUnitCore instance now acts as the test harness, which allows a GUI to register itself as a UIListener. This FlexUnitCore instance runs the test suite.

Our example under the new framework is now fully ready. Next, you edit the test class a bit and add a couple of extra test cases to demonstrate a few of the newer features.

The first set of additions are two methods, one runs before every test and the other after every test. These methods are:

```[Before] public function runBeforeEveryTest():void {
    numericalValue.setValue(50);
}

[AFTER] public function runAfterEveryTest():void {
    numericalValue.setValue(0);
}
```

After adding these two methods, you will see your tests fail. The tests were written with an initial value of zero in mind. The new method runBeforeEveryTest sets the initial value to 50, which obviously makes the tests fail.

Apart from these methods, you could add additional tests that utilize hamcrest assertions and verify theories. A hamcrest assertion involves the concept of a matcher, which allows for flexible expressions as test conditions. Verifying a theory, on the other hand, is based on the concept of testing an insight or a set of assumptions. You consider a few data points, which you pass in as parameters to the test and then verify your theory. You can learn more about the concept of “theories” from a white paper titled “The Practice of Theories: Adding ‘For-all’ Statements to ‘There-Exists’ Tests.” This white paper on “theories” is authored by David Saff and Marat Boshernitsan and is accessible online at http://shareandenjoy.saff.net/tdd-specifications.pdf.

A simple hamcrest assertion could be added as shown in Listing 8-3. This new test class can be added to a test suite, which could possibly be named FlexUnit4TestSuiteHamcrest, and then passed in as a parameter to the run method of a FlexUnitCore instance.

**Listing 8-3: NumericalValueTestHamcrest Source**

```java
package test
{
    import org.hamcrest.number.between;
    import flexunit.framework.Assert;
```
import org.hamcrest.Description;
import org.hamcrest.StringDescription;
public class NumericalValueTestHamcrest {
    var numericalValue:NumericalValue = new NumericalValue();
    var numbers:Array;
    [Before]
    public function runBeforeEveryTest():void {
        numericalValue.setValue(50);
        numbers = new Array( -250, -200, -150, -100, -50, 0, 50, 100, 150, 200, 250, 300 );
    }
    [After]
    public function runAfterEveryTest():void {
        numericalValue.setValue(0);
        numbers = null;
    }
    [Test(order=2)]
    public function checkBetweenRangeExclusive():void {
        numericalValue.add(50);
        Assert.assertTrue("Value after adding 50 to the initial 50 value is 100",
            between(numbers[5],numbers[11],true)
                .matches(numericalValue));
        numericalValue.add(50);
        Assert.assertFalse("Value after adding 50 to 100 is 150",
            between(numbers[8],numbers[11],true)
                .matches(numericalValue));
    }
    [Test(order=1)]
    public function checkBetweenRangeInclusive():void {
        numericalValue.subtract(100);
        Assert.assertTrue("Value after subtracting 100 from the initial 50 values is -50",
            between(numbers[3],numbers[5],true)
                .matches(numericalValue));
        var description:Description = new StringDescription();
        description.appendDescriptionOf(
            between(numbers[0], numbers[11]));
        Assert.assertEquals("Expected mismatch description",
            "a Number between <-250> and <300>",
                description.toString());
    }
}

Next, an elementary example of testing theories is explained. Say that you assume a particular numerical value is greater than a given number, for example 150. You could test this insight by checking if your
data point passes a "greater than 150" test or not. The "greater than 150" test then becomes a theory, and you run tests to establish if your assumption is correct or not. Part of code that defines [DataPoint] and [Theory] for this elementary example is:

```
[DataPoint]
public static var number:Number = 200;

[Theory]
public function testGreaterThanTheory( number:Number ):void
{
    assumeThat( number, greaterThan( 150 ) );
}
```

Theories are RunWith a special theory runner, org.flexunit.experimental.theories.Theories.

Apart from all the newer assertion types, FlexUnit 4 supports concurrent processing of multiple asynchronous events. Test methods are annotated with the [Test] metadata. This metadata has optional async and timeout properties. Methods that need asynchronous processing need to simply specify the async property in their metadata. In addition, such methods could have a timeout property that defines the maximum time after which the method does not wait for the asynchronous processing to complete.

So far, you have learned a lot about FlexUnit and the ways of unit testing in Flex applications. Next, attention is turned to JUnit and the unit testing facility for the Java code. The server-side code in BlazeDS runs in a JVM and is Java.

**JUnit**

FlexUnit draws inspiration and ideas from JUnit, both being xUnit testing framework types. You have learned a fair bit about FlexUnit. So now as you discover JUnit, things will, not surprisingly, look very familiar.

Let’s first see JUnit in use. Once again consider a class that holds a numerical value and allows for values to be added to and subtracted from it. Let the class and method names be the same as in the AS3 class of the Flex application. That way, you will get a chance to compare and contrast the two.

Start by downloading the latest release of JUnit from http://sourceforge.net/projects/junit/. The latest stable release version is 4.6, but the exact version number could vary depending on when you finally get down to downloading JUnit. Just make sure that you have version 4 or higher. Versions of JUnit 4.x leverage Java annotations. Annotation support was introduced with Java 5. So, also make sure to have a version of JRE equal to or greater than 5.

JUnit is distributed as a JAR archive file. Once the download is complete, add the JAR file to the Java CLASSPATH environment variable. All classes specified in the CLASSPATH can be loaded by the JVM and are, therefore, available for an application to use.

After downloading JUnit and setting it up on the classpath, create the NumericalValue class. The NumericalValue class is a simple class that has only one property, named value. The class defines a getter and a setter method for this property. In addition, it has two methods, called add and subtract,
that facilitate the addition and subtraction operations between the value and a passed in numerical parameter. The source for NumericalValue.java is:

```java
package test;

public class NumericalValue {
    private int value = 0;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public void add(int addedValue) {
        value = value + addedValue;
    }

    public void subtract(int subtractedValue) {
        value = value - subtractedValue;
    }
}
```

Next, create a test class to verify the results of the addition and subtraction operations. The source of this test class, named NumericalValueTest, is shown in Listing 8-4.

**Listing 8-4: NumericalValueTest Source**

```java
package test;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import junit.framework.Assert;

public class NumericalValueTest {
    @Before
    public void setUp() throws Exception {
        // Setup code...
    }

    // Test methods...
}
```

Continued
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

Listing 8-4: NumericalValueTest Source (continued)

```
{
}

@After
public void tearDown() throws Exception{
{

@Test
public void testAdd(){
{
    NumericalValue numericalValue = new NumericalValue();
    numericalValue.add(50);
    Assert.assertTrue("Value after adding 50 to the
    Initial 0 value is 50", numericalValue.getValue() == 50);
    numericalValue.add(50);
    Assert.assertEquals("Value after adding 50 to 50 is 100",
     100, numericalValue.getValue());
}

@Test
public void testSubtract(){
{
    NumericalValue numericalValue = new NumericalValue();
    numericalValue.subtract(100);
    Assert.assertTrue("Value after subtracting 100 from the
    Initial 0 value is -100", numericalValue.getValue() == -100);
}
}

Compile both NumericalValue and NumericalValueTest and then run the JUnit tests using the following command:

    java org.junit.runner.JUnitCore test.NumericalValueTest

You can, alternatively, make the test part of a test suite and run it with a test runner of your choice, in a
manner similar to that shown in the example for FlexUnit 4.

There are plenty of books and online resources on JUnit, and I leave it for you to explore those resources.
However, before I move on to the next topic, let me quickly summarize the key features, which are:

- Test classes do not need to extend a specific class but only need to be annotated with the @Test
  metadata term.
- Methods can be run before and after every test. The methods that run before every test are
  annotated with the @Before metadata term, while the methods that are run after every test are
  annotated with the @After metadata term.
- The setting up and tearing down of fixtures can be done not only before and after
  every test method run but before any test runs and after all tests are run. A class that
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

contains code to run before any test is run is annotated with the @BeforeClass metadata term, while a class that runs after all tests are run is annotated with the @AfterClass metadata term.

- Tests to be ignored during a test run need to be annotated with the @Ignore metadata term.
- Tests that are expected to throw exceptions should include the expected property in their @Test annotation. For example: @Test(expected = ArithmeticException.class).
- Multiple test suites can be included in a @Suite.SuiteClasses annotation.
- A runner could be defined for the suite by using the @RunWith annotation.
- A timeout property can be defined for the @Test annotation. The test fails if it doesn’t complete before the timeout.
- Hamcrest assertions, assumptions, and theories can be checked.
- Junit4Adapter can be used to run JUnit 4 tests with JUnit 3

So far, you have been exposed to the fundamentals of unit testing using FlexUnit and JUnit. You can use the two frameworks in tandem to write and run tests for both the client- and server-side code. Such test runs can be automated using Apache ANT and can be integrated into continuous integration tools like CruiseControl. You will learn about both ANT-based automation and CruiseControl integration later in this chapter.

While unit tests give a developer power to test and fortify one’s code, it’s often imperative that user interfaces and systems as a whole be subject to testing. Often the individual pieces could work well, but they have trouble working together.

This is where functional testing and integration testing come into play, which is what you learn about next.

Functional and Interface Testing

Functional testing or integration testing tests a system as a whole against its requirements. It’s a form of black box testing. In applications that involve a graphical user interface, user interface interaction testing also plays a critical role in system testing.

In this section, I will illustrate the following:

- FlexMonkey — An open source testing framework for Flex applications that provides capture, replay, and verification functionality for UI testing.
- FunFx — An open source Ruby-based tool for functional testing of Flex applications.

Although xUnit testing frameworks and its extensions can be used for system testing, I will not indulge in that discussion here.

Further, although I discuss system testing, especially UI testing, with Flex in perspective, I do try to bring in the BlazeDS aspects when relevant. After all, we are trying to test the entire application and that includes both Flex and BlazeDS.

Let’s start with FlexMonkey.
Flex Monkey

FlexMonkey is a testing framework that automates the process of capturing, replaying, and verifying Flex user interfaces. It leverages the Flex automation API and builds on top of the sample automation adapter, AutoQuick.

The Flex automation framework consists of four distinct parts, namely:

- **Core Flex API** — The `SystemManager` class parents all display components in Flex. It calls the `init` method of all instantiated mixins. The `SystemManager` class is the parent of the automation framework components and calls the `init` method of the `AutomationManager` mixin.

- **Automation API** — At the heart of the automation framework is a singleton called the `AutomationManager`, which facilitates the recording of interactions. Each UI component has a delegate that facilitates manipulation of these components by the automation framework. Components that don’t have a delegate of their own use their parent’s delegate to work with the automation framework. A third important part of the automation framework is a static class, called `Automation`, that stores and manages the map between components and their delegates.

- **Agent or Adapter** — This is the intermediary that facilitates communication between a Flex application and the automation tools.

- **Tool** — The automation tool itself that implements the recording and playback features.

To understand FlexMonkey better, it would be useful to walk through a simple usage scenario. As a first step download and install FlexMonkey. The FlexMonkey project is hosted online at http://code.google.com/p/flexmonkey/. Download the latest version of the distribution. Currently, the latest release version is 0.8a. The testing framework has not reached the version 1.0 milestone, but it is fairly ready for serious use. The distribution is available in a `.zip` archive file format. Expand the `.zip` archive file to a folder on your file system. When you expand the archive file, you should see the following files and subfolder(s):

- **lib** — External `.swc` files that FlexMonkey uses.
- **src** — The source code for the `MonkeyContacts` sample application and the associated tests.
- **FlexMonkey.swc** — A library that contains the FlexMonkey recording and playback API, the FlexMonkey user interface, and a FlexUnit test runner.
- **FlexMonkeyLauncher.swf** — Launches Flex applications to be used with FlexMonkey. Recorded tests are also loaded using this launcher.
- **MonkeyContacts.swf** — A sample application that uses FlexMonkey.
- **FlexMonkeyTests.swf** — A set of unit tests for the `MonkeyContacts` application. The `FlexMonkeyTests` application shows how unit test scaffolding and test methods can be generated. It also illustrates how custom tests can be added to the script generated by the FlexMonkey recorder.
- **MonkeyContactsTest.html** — An HTML wrapper that includes a `FlexMonkeyLauncher` instance that launches the `MonkeyContacts` sample application.

If you are impatient to see how FlexMonkey works, then simply open `MonkeyContactsTest.html` in a web browser, and you will see a screen come up, as shown in Figure 8-2. In this figure, you will notice...
input controls labeled: App to Test and Tests to Run (Optional). App to Test is where you specify the .swf application file you would like FlexMonkey to load. Once you have loaded a .swf and recorded and played back a few interactions with it, you have the choice to extend and customize the generated test scripts and package it off as a separate Flex application. This test application .swf is specified in the Tests to Run field.

![Figure 8-2](image)

FlexMonkey provides a unobtrusive way to record and play back target Flex applications. To prove this point and to explain how to include FlexMonkey with your existing Flex application, I will indulge in a small exercise.

The turnkey version of BlazeDS comes with a set of sample applications, which are bundled together in a web application, called Samples. This web application is distributed in a WAR archive file format. You can deploy this archive file, samples.war, to an application server of your choice. In many application servers, such as Apache Tomcat and JBoss AS, dropping this WAR file in the deployment folder and possibly restarting the application server is all that’s required. In Apache Tomcat, deploy the application in its webapps folder. The turnkey distribution comes with an instance of Apache Tomcat server, so all you need to run and access the samples is to simply start the server instance. In JBoss AS, samples.war can be deployed in the server/default/deploy folder.

In order to enable FlexMonkeyLauncher to load and launch a Flex application .swf, you need to copy FlexMonkeyLauncher.swf file in the same directory as the compiled Flex application. My intention is to run FlexMonkey against an example that uses BlazeDS remoting, so I add FlexMonkeyLauncher.swf to the folder that contains the compiled .swf for such an application.

Remoting example applications are available in the BlazeDS samples.war. The simplest of the remoting examples in this bundle is testdrive-remoteobject. The compiled .swf for
testdrive-remoteobject, called main.swf, resides in samples/testdrive-remoteobject. In order to add FlexMonkeyLauncher.swf to samples.war, you follow a three-step process like so:

- Unpack samples.war to any folder in the file system. You could use jar xvf samples.war to unpack samples.war.
- Copy FlexMonkeyLauncher.swf to the testdrive-remoteobject within the unpacked samples application folder.
- Re-create a samples.war file out of the folder content. FlexMonkeyLauncher.swf will now be included. You could use jar cvf samples.war *, when in the folder where the contents of the expanded samples.war reside, to create samples.war.

Next, redeploy the modified samples.war file to the application server of your choice. I deploy it to JBoss AS. Finally, access the FlexMonkey Launcher.swf file using the appropriate URL. In my case, it’s accessible at http://localhost:8080/samples/testdrive-remoteobject/FlexMonkeyLauncher.swf. You will see a screen like that depicted in Figure 8-3.

![Figure 8-3](image)

Load the testdrive-remoteobject application, main.swf, using this Flex Monkey launcher. A screen like that depicted in Figure 8-4 will appear.

Click the Get Data button. That will populate the application data grid. Remember to start the sampledb by going to the sampledb folder in the BlazeDS turnkey distribution and running the startdb script file, before you click the Get Data button, or you will end up with an error. The sampledb is a small-footprint database to populate sample data in the sample BlazeDS applications. This database is distributed as a part of the BlazeDS sample applications bundle.

You know that FlexMonkey lets you record your interactions and user interface gestures with a Flex application. Once the interactions are recorded, this allows you to play back these recordings. It also generates test case code for these interactions. To start recording interactions, click the Record button on
the FlexMonkey user interface. The Record button is a toggle button, so alternate clicks cause recording to start and stop. Figure 8-5 shows a recording of a series of interactions with the application.

Next, click on the TestCase Code tab to view the generated test case code. In my case the code is:

```actionscript
// Test test method
public function testSomething():void {
    var cmdRunner:CommandRunner = new CommandRunner();
    cmdRunner.addEventListener(MonkeyEvent.READY_FOR_VALIDATION,
```
addAsync(verifySomething, 10000));

// Called after commands have been run
private function verifySomething(event:MonkeyEvent):void {
    var comp:UIComponent = MonkeyUtils.findComponentWith("someAutomationName")
    Assert.assertEquals("value", comp.prop);
}

The generated source will be different for you and will depend on what interface interactions you indulge in. The generated code, like the preceding sample, is the starting point for incorporating test cases. testSomething uses a command runner to rerun recorded user interactions for testing purposes. Once the command runner runs the command, a READY_FOR_VALIDATION event is dispatched. To handle this event and run assertions, add the verifySomething method as the event listener for this event. Make sure to add this using FlexUnit’s addAsync method. verifySomething is the method in which you can include the test assertions to verify the logic. FlexMonkey generates scaffolding to retrieve a generic UIComponent using the automation name, which provides a reference to the UIComponent in the context.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

of the Flex automation framework. If you assign a specific ID to a UIComponent it is assigned as its automation name. If no explicit ID is provided by the developer, the system generates an automation name. Once you get a reference to a UIComponent, you can run assertions on the component’s property values. Generic UIComponents can be cast to strongly typed components and then run through assertions. So the generated statement:

```javascript
var comp:UIComponent = MonkeyUtils.findComponentWith("someAutomationName");
```

can be replaced with:

```javascript
var comp:DataGrid = MonkeyUtils.findComponentWith("_main_DataGrid1") as DataGrid;
```

in this example, since we know the component referenced with _main_DataGrid1 automation name is the DataGrid of our application. In this particular situation, you can make an assertion that the first item in the collection belongs to the category identified by the value of 9000. Such an assertion can be included as follows:

```javascript
Assert.assertEquals("9000", ArrayCollection(comp.dataProvider).getItemAt(0).category);
```

After you have created the test cases, include and compile them as a separate Flex application. Doing this allows unobtrusive test running. Test cases can easily be added to the FlexMonkey test suite using FlexMonkey.addTestSuite method. The test Flex application can be launched using FlexMonkeyLauncher.swf.

You learned earlier that the recorded user interaction sequence is run with the help of a series of corresponding FlexCommand instances. These instances are run with a default gap of 5 seconds between them. You can explicitly control this delay by adding the PauseCommand and specifying the required delay in milliseconds as an argument to the method. You can also call external AS3 methods and modify the recording run by adding a CallCommand method at the appropriate point in the sequential run. External methods can be called by passing them as arguments to the CallCommand method.

FlexMonkey can be extended and customized using AS3 alone, and that makes it more attractive than alternatives that require the use of custom scripting languages. Sometimes the use of an external language can be advantageous, especially if the external language makes the testing experience more robust or provides easier idioms and syntax for productivity enhancement. The next open source tool I will consider for functional testing of Flex and BlazeDS applications is FunFX, which leverages the popular Ruby programming language.

**FunFX**

FunFX is an open source, functional testing tool for Flex applications. It is built on Ruby, a popular dynamic language. The tool leverages the Flex automation API to implement a user interface, but exposes its own API in Ruby to enable developers to write user interface interactions and user gestures. The tool works only with a Flex application working in an Internet Explorer (IE) browser instance, which may be a showstopper for IT shops that develop applications on Linux or MAC OS computers.
You can download FunFX from its RubyForge site at http://funfx.rubyforge.org/. The current release version is 0.0.4, indicating that the tool is very new and still in its alpha release phase. Although the tool is in its alpha phase, it appears promising and that is why I talk about it in this book. Besides making it possible for a developer or tester to walk through a user interaction scenario, the tool allows the inclusion of both unit testing and behavior-driven testing frameworks.

FunFX is available in a .zip archive file. After you download and expand the archive file to a folder on your file system, you will find the following files in it:

- AutomationGenericEnv.xml
- FunFXAdapter.swc
- FunFX-0.0.4.gem

I mentioned earlier that FunFX leverages the automation API. Therefore, to get FunFX to work you need to have access to the .swc files that contain the automation framework classes. For Flex 3, the automation framework is part of Flex Builder 3. Flex Builder (which is now renamed Flash Builder) is a licensed tool, although it’s inexpensive and 30-day trial versions of this IDE IS available.

To run FunFX, you need the following automation-related .swc library files:

- automation_agent.swc (in the Flex Builder 3 Plug-in\sdk\3.2.0\frameworks\libs folder)
- automation_agent_rb.swc (in the Flex Builder 3 Plug-in\sdk\3.2.0\frameworks\locale\en_US folder)
- automation.swc (in the Flex Builder 3 Plug-in\sdk\3.2.0\frameworks\libs folder)

Before you start using FunFX with your Flex application, download and install Ruby and its gems infrastructure. Steps on getting and installing Ruby or its libraries are beyond the scope of this book. However, it’s easy to find plenty of information on this subject by simply googling for it. Also, the following online resources could help you download, install, and set up Ruby and RubyGems:

- http://rubygems.org/read/chapter/1
- http://rubyonrails.org/download
- http://rubygems.org/read/chapter/15

Once Ruby and RubyGems are installed and ready for use, install the FunFX gem. FunFX gem is available in the .zip file that you download from the FunFX RubyForge site. You can install the FunFX gem with the help of a simple command, like so:

```bash
gem install FunFX-0.0.4.gem
```

FunFX provides a way to walk through a set of interactions in a Flex application. These interactions are specified in Ruby. The Ruby program interacts with the Flex application with the help of two key connectors:

- Win32OLE — This helps drive the IE browser interactions.
- ExternalInterface — This helps the Ruby scripts connect with a Flex application.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

On the Flex side, a library that serves the purpose of an adapter interfaces with the automation API. This library is FunFXAdapter.swc. The Ruby API and framework for FunFX itself is dynamically built from an XML file, named AutomationGenericEnv.xml. This XML file declares and defines all the display objects and their properties.

In order to leverage FunFX to write automated interaction use cases and associate them with tests, first create a new Flex project. Then pass in the following command line parameters to the Flex compiler:

```
--include-libraries
"C:\Program Files\Adobe\Flex Builder 3 Plug-in\sdks\3.2.0\frameworks\libs\automation.swc"
"C:\Program Files\Adobe\Flex Builder 3 Plug-in\sdks\3.2.0\frameworks\libs\automation_agent.swc"
"C:\Program Files\Adobe\Flex Builder 3 Plug-in\sdks\3.2.0\frameworks\locale\en_US\automation_agent_rb.swc"
```

Now you are ready to start writing your Ruby scripts to drive user interaction scenarios.

For example, you can click the Get Data button of the BlazeDS distribution testdrive-remoteobject sample application using FunFX ruby scripts, like so:

```ruby
@ie = Funfx.instance
@ie.start(true)
@ie.speed = 1
@ie.goto("http://localhost:8080/samples2/testdrive-remoteobject/index.html", "main")
@ie.button("Get Data").click
```

FunFX does not have a means of testing the application. For that, you could leverage Test::Unit or RSpec. Together with the testing frameworks, FunFX provides a complete automated functional testing facility.

Next, I will illustrate a rudimentary example of using RSpec with FunFX. The purpose of this illustration is only to provide you with a sample usage pattern of a functional testing scenario. It is advisable that you learn the xUnit framework, Test::Unit, and the behavior-driven testing framework, RSpec, in detail if you are interested in leveraging Ruby for Flex and BlazeDS application functional testing.

To start using RSpec, first install it using Ruby Gems simply by typing `gem install rspec` on your command line. RSpec will be installed, and you will be prompted with a nice thank you note on your command line.

Next, create three important files to use RSpec (with FunFX). These three files will contain the following:

- **Story** — This allows the definition of test scenarios using simple text. It provides a way for a business user to understand the test case as well.
- **Steps** — This provides the steps that need to run so that the story translates to the underlying code.
- **Test runner** — This is a file that brings the story and the steps together and uses the RSpec’s rBehave to run the behavior-driven tests.
Your sample story file, which only covers the trivial usage scenario of clicking a Get Data button to retrieve a list of products is:

```
Story: Get Product List

As a user
I want to be able to get a list of all products,
which in this case is a list of Nokia phones
So that I can see what is available

Scenario: Get product
Given the user is logged into the application
When the user clicks the Get Data button
Then the product list view is visible in the data grid

Then close the application
```

A corresponding steps file for this story can be like so:

```
require 'funfx'

steps_for :product do
  Given "the user is logged into the application" do
    @ie = Funfx.instance
    @ie.start(true)
    @ie.speed = 1
    @ie.goto("http://localhost:8080/samples2/testdrive-remoteobject/index.html", "main")
  end

  When "the user click the $button_name button" do |button_name|
    @ie.button(button_name).click
  end

  Then "the product list view is visible in the data grid" do
    @ie.data_grid().visible == "true"
  end

  Then "close the application" do
    @ie.unload
  end
end
```

Finally, create a file that runs this RSpec test scenario. The runner file can be written like this:

```
require 'rubygems'
require 'spec/story'
$:unshift(File.dirname(__FILE__) + '/lib')
require 'steps/product'

with_steps_for :product do
  run File.dirname(__FILE__) + '/stories/get_products'
end
```
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

For this trivial example, that is all you need to do to run a RSpec test case. You will obviously need to do a lot more work for real-life, complicated Flex and BlazeDS applications. However, the general approach would be the same as that taken in this trivial case.

Having seen FlexMonkey and FunFX, you now know the automated functional testing possibilities for Flex and BlazeDS applications. Beyond these tools, you do have commercial offerings like QTP and SilkTest that perform such tasks and provide a bunch of bells and whistles over and above the basic testing. If you don’t like any of these, then you also have the choice to use the automation API and write a tool of your own.

Next, we look at automating the task of running the unit tests and functional tests using Apache ANT. An important aspect of thorough testing is the automation of repeatable test runs. This allows for continuous checks after every new function implementation and refactoring.

Automating with ANT

In this section, you will learn to use Apache ANT to automate the tasks of running your tests. Apache ANT is a Java-based build tool that aims to provide a flexible, robust, and extensible build system across all platforms that support Java. You can download the software and learn more about it at http://ant.apache.org/.

To run FlexUnit tests using ANT, it is prudent to leverage the FlexAntTasks, a freely available utility for this purpose. FlexAntTasks is now bundled with the Flex SDK and available to you if you have a Flex SDK greater than version 3 installed on your machine. FlexAntTasks extends the ideas of the JUnit ANT task. JUnit, as you already know from an earlier section in this chapter, is a xUnit testing framework for Java.

FlexAntTasks runs a test Flex application using an ANT script. In addition, it starts up a socket server, which is used by the test runner to connect back to relay the test results. After the application is launched, the flex unit test runner opens a connection back to the socket server that is part of the flex unit ANT task. As the test runs complete, results are sent back over the socket connection. The result is sent back in an XML format. Toward the end of the test suite run, the different XML pieces, the test run results, are combined into a useful report. The aggregated and structured test results reports follow the same style and format as the JUnitReport. You can read more about the off-the-shelf report format, named JUnitReport, which aggregates the individual XML files into one, at http://ant.apache.org/manual/OptionalTasks/junitreport.html.

I mentioned that FlexUnit ANT tasks are derived from the corresponding JUnit ANT tasks. Therefore, it’s obvious that JUnit ANT tasks follow a similar pattern of communication and results aggregation. You can use the FlexUnit and JUnit ANT tasks together to automate your test runs for a Flex application and its BlazeDS remote service objects.

FlexAntTasks can be downloaded from http://weblogs.macromedia.com/pmartin/archives/2007/01/flexunit_for_an_1.html. The binary compiled versions of the FlexAntTasks are distributed as a JAR file. Make sure to add this JAR file to Apache ANT’s classpath. At this stage, the assumption is that you have downloaded, installed, and set up Apache ANT.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

If you haven’t installed ANT yet or are having trouble getting it up and running, then you will benefit from reading the ANT documentation on the steps involved in downloading, installing, and setting up ANT. Usually, the installation and setup only involves downloading and unpacking the compiled distribution, setting the `ANT_HOME` environment variable to point to the folder where ANT is unpacked, and adding the bin directory of the unpacked ANT folder to the `PATH` environment variable.

As mentioned earlier, the FlexUnit ANT task is assisted by a test runner that formats the results in a desired XML format and sends it up to the socket server, started by the FlexUnit ANT task. The test runner can be downloaded from http://weblogs.macromedia.com/pmartin/archives/2007/01/flexunit_for_an_1.html. It is distributed as a `.swc` archive file, named FlexUnitOptional.swc.

Next, to explain the usage of FlexAntTasks I will walk you through a simple example. FlexAntTasks and the test runner come with a sample application that illustrates a usage scenario. You certainly have the option to look that up. However, to keep things consistent with what you have learned in this chapter, I use FlexAntTasks to run the NumericalValueTest class from the section titled “FlexUnit.”

Using Flash Builder, first create a new Flex application. Add the following `.swc` files to the `libs` folder:

- `flexUnit.swc` — The FlexUnit core features are located in this archive file.
- `FlexUnitOptional.swc` — The JUnitTestRunner that formats and sends XML result messages to the Flex unit ANT tasks–initiated socket server is contained in this archive file.

The tests are already defined. So, the only real change is the use of `flexunit.junit.JUnitTestRunner` to run the tests. This change can be incorporated by replacing the earlier test runner method with the following:

```actionscript
[Bindable]
private var runner : JUnitTestRunner;
private function onCreationComplete(): void {
    runner = new JUnitTestRunner();
    runner.run( createSuite(),
        function(): void {
            fscommand("quit");
        } );
}

private function createSuite(): TestSuite {
    var testSuite : TestSuite = new TestSuite();
    testSuite.addTestSuite( NumericalValueTest );

    return testSuite;
}
```

Besides the new test runner method, you can define additional classes to receive and parse the resulting XML files. The preceding illustration describes test runs of individual hand-coded unit tests. However, the same approach can be used to run FlexMonkey-generated tests.
Next, the XML result files will be used to integrate the FlexUnit test runs with “CruiseControl,” which is a popular continuous integration software.

**Continuous Integration**

Continuous integration is a software development practice in which each developer in a team frequently integrates his or her work into the overall application. Usually, developers integrate their work on a daily basis and utilize automated scripts to build and run tests after integration. Many agile developers follow the practice of continuous integration. Continuous integration helps in significantly reducing integration problems.

CruiseControl and Hudson are two popular continuous integration management and monitoring tools. In addition, Maven and ANT build systems could be leveraged for continuous integration. You can read more about CruiseControl at [http://cruisecontrol.sourceforge.net/](http://cruisecontrol.sourceforge.net/). More about Hudson can be explored from its website accessible at [https://hudson.dev.java.net/](https://hudson.dev.java.net/).

**Using CruiseControl**

CruiseControl comes in three forms:

- A binary distribution in a .zip archive file, which you can simply expand and start using
- A source distribution that needs to be compiled before it is ready for use
- A Windows installer that allows you to install CruiseControl with the help of a wizard on the windows platform

CruiseControl is built on Java but has a version for Ruby as well. If you prefer to use the Ruby version, then download the required software, named CruiseControl.rb, from [http://cruisecontrolrb.thoughtworks.com/](http://cruisecontrolrb.thoughtworks.com/).

FlexUnit ANT tasks and CruiseControl can be integrated using any of the following two strategies:

- Consume the XML output from the FlexAntTasks
- Run an AIR application side by side with your Flex application for running your tests and generating XML-format-based results

After you have created the ANT build scripts to build your application from source and run the tests, getting them to work under CruiseControl does not require any special effort. The only complications come up in the following scenarios:

- Running tests in a headless mode, where the user interface is not brought up
- Running network calls from within the local Flash Player security sandbox

When you launch Flex applications to test, an instance of the Flash Player is started and the application is run within that Player’s virtual machine. However, there may be situations in which you do not desire or have the flexibility to launch a Flash Player instance. One such case could be a Linux server that does
not have a local display. In such cases, it is advisable to launch the application using a virtual display within a VNC server instance. CruiseControl and Hudson, the alternative continuous integration tools, have hooks to launch VNC displays.

When running within a local Flash player instance, remember to specify the SWF and assets as trusted sources.

I don’t cover all the procedural details of configuring and running the continuous integration software tools here, but none of the steps, apart from the ones mentioned previously, are any different for Flex applications than for other standard cases, such as Java server-side applications.

Next, I briefly survey an open source code coverage tool, Flexcover.

**Code Coverage**

An open source tool, called Flexcover, is a code coverage tool for Flex, AIR, and AS3 applications. Code coverage is a measure used in software testing. It helps find out how much of the code has been tested. There are multiple ways to find out the amount of code that has been tested, namely:

- **Statement coverage** — Tests and measures if each line of code has been executed
- **Decision coverage or branch coverage** — Tests and measures if each control structure has been evaluated for true and false conditions
- **Function coverage** — Tests and measures if each function has been called
- **Condition coverage** — Tests and measures if each subexpression has been evaluated for true and false conditions
- **Path coverage** — Tests and measures if every possible path of execution has been executed

Flexcover provides for statement and branch coverage. In order to provide the code coverage instrumentation and help collect the coverage data, the tool defines a custom version of the AS3 compiler. This modified compiler inserts extra function calls in the code within the SWF and SWC outputs. These functions help assess the code coverage at runtime and report this information to a separate part of the tool that prints the output as a structured report.

To use Flexcover, all you need to do is use the Flexcover SDK to compile your code. The function calls will be implicitly injected. Download a copy of Flexcover from [http://code.google.com/p/flexcover/](http://code.google.com/p/flexcover/).

The next and last topic on testing is load testing.

**Load Testing**

When you build a Flex application, you often want to know how your application would scale and work when multiple users access the application concurrently. When it’s a Flex and BlazeDS application, these scalability constraints also go to how the server handles the service requests under load. In this section, I focus exclusively on BlazeDS and show you how to test load conditions to find out how your application scales.

The first requirement is to intercept the Flex application’s call to the remote service and explore if such calls can be made programmatically or using a tool that helps generate load. A good way to intercept
the request from a Flex application to a server is to set up a proxy in the middle. There are many proxies available in the market, but there are three that I would like to recommend:

- **Charles** — A very well-known web proxy among Flex and Flash developers can be leveraged to intercept requests. Charles is a commercial tool and details about this tool can be found online at [www.charlesproxy.com](http://www.charlesproxy.com).
- **Fiddler** — A free web proxy for Windows platform users, it works well and is completely free. You can download a copy from [www.fiddler2.com/fiddler2](http://www.fiddler2.com/fiddler2).
- **Wireshark** — A network protocol analyzer that works well to intercept requests and provide details on the request URL and parameters. Wireshark is open source and available for Windows, Linux, and OS X. You can download a copy from [www.wireshark.org](http://www.wireshark.org).

Once you have a web proxy or a network protocol analyzer in place you can determine the server-side URL, request parameters, and mime type for the request a Flex application makes to its BlazeDS-based remote services.

Once again, let’s take the BlazeDS sample `testdrive-remoteobject` application to find out the request details. As soon as I click on the Get Data button, I can see that this button invokes a server-side service that is accessible at [http://localhost:8080/samples/messagebroker/amf](http://localhost:8080/samples/messagebroker/amf). The request made is a POST HTTP method request, and the `content-type` is set to `application/x-amf`.

Now I need a tool that can make this request multiple times and tell me how much time and resources each of these of requests consumed. It would tell me if successive requests were slower in responding and if there is a point beyond which the server cannot handle a request at all.

Load- or stress-testing tools usually are most appropriate for making multiple requests and assessing how the requests performed. I recommend the use of Apache JMeter, available online at [http://jakarta.apache.org/jmeter/](http://jakarta.apache.org/jmeter/), for load testing. JMeter is an open source Java desktop tool that is very flexible and provides for load-testing configuration in a variety of situations.

To generate multiple requests to the BlazeDS server using JMeter, first create a new `ThreadGroup` in JMeter. You will be prompted with a screen like that shown in Figure 8-6.

![Figure 8-6](http://www.wowebook.com/image.png)
I named my thread group “BlazeDS Load Thread Group” and specified the following configuration parameters:

- **Number of Threads** — I choose initially to go with 5 to imply 5 users
- **Ramp-Up Period** — I choose 0, which means all threads start up at once. You could set up a delay between thread starts using this parameter.
- **Loop Count** — I chose to go with the Forever option to imply that there will be continuous requests till I choose to stop.

Next, I add an HTTP Request sampler to the thread group. Then, I specify the URL, HTTP method, and content type to the HTTP sampler. Look at Figure 8-7 for details.

![HTTP Request](image)

**Figure 8-7**

Then, I add a Graph Results listener and specify how I would like to aggregate and view data. See Figure 8-8 for details.

Finally, I run the test and see the results in real time. That’s it as far as load testing goes. This also brings us to the end of the discussion on testing. Next, it’s the topic of debugging.
Debugging

Unlike testing, debugging is an activity that starts after the code is ready and compiled to run. Debugging starts with monitoring and logging at runtime to troubleshooting as the problems emerge in a program’s behavior.

The simplest and the most important starting point for debugging is the logging of operations. You can start to understand and troubleshoot problematic behavior only after you know what is going on.

Logging

Flex and BlazeDS applications have the advantage and flexibility of logging the operations both on the client and on the server side. In this section, I will cover client side logging first and then explain server-side logging.

Client-Side Logging

You can log all client-side interactions to a file locally on your computer. The Flash debug player has two primary ways of logging:

![Figure 8-8](image)
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

- **Trace method** — The global trace method writes log messages to a file. Log messages are written as strings.

- **Logging API** — This provides an abstraction on top of the global trace method, allowing greater control and definition of custom log targets for client-side log messages.

By default, the global trace method’s output is written to a file called `flashlog.txt`, which resides in the Macromedia/Flash Player/Logs folder. This Macromedia folder, along with its subfolders, resides in a location that depends on your operating system. Platform-wise the location will be like so:

- **Windows** — `C:\Documents and Settings\<username>`
- **OS X** — `/Users/<username>/Library/Preferences`
- **Linux** — `/home/<username>` (The folder itself is named .macromedia)

The output of the global trace method depends on your custom configuration, which is defined in a file called `mm.cfg`, which lies at the same location as the Macromedia preferences folder. You can configure error and trace output logging in the configuration file, as follows:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Apart from logging your messages with explicit `trace` method calls, you can also direct all Flex-generated debug messages to `flashlog.txt` by including `<mx:TraceTarget loglevel="2"/>` in your code.

The standard client-side logger writes to a file on the local computer, but you may want to write the log file to a different destination, say the server. In such a case, you can leverage the Logging API and create a custom log target for the purpose.

The logging API defines three entities that work together to facilitate logging. The first of these three is a Logger, which implements the `ILogger` interface. The Logger captures log messages on the basis of two parameters:

- **Category** — The identifier for a class or a subsystem.
- **Level** — The attribute that specifies the message type. Values like Error, Debug, Warning, and Info make up the different levels.

The second entity in the logging framework is a LogTarget. The LogTarget gets the messages from a Logger and processes them. The default target, `TraceTarget`, writes all trace messages to the debugger console or the designated local file. The third entity is the destination itself where the log messages are written to.

In order to create a custom logging facility so that messages are written to the server, create a new LogTarget that captures the log messages and sends them down to the server. You can send the messages to the server using any of the remote service facilities such as web services, remote objects, or the HTTP service. Here, I will only illustrate the custom LogTarget, whose code is:

```
use namespace mx_internal;

public class ServerLogTarget extends LineFormattedTarget { 
```
private var server : ClientLogService;

public function ServerLogTarget() {
    super();

    super.fieldSeparator = "##";
    super.includeTime = true;
    super.includeDate = true;
    super.includeCategory = true;
    super.includeLevel = true;
    super.level = LogEventLevel.ALL;

    //server = get an instance of the service that would send messages to the server }

public override function logEvent(event : LogEvent) : void {
    var category : String = ILogger(event.target).category;
    if (category.indexOf("mx.") != 0) {
        super.logEvent(event);
    }
}

mx_internal override function internalLog(message:String) :void {
    if (server != null) {
        //Call service method and pass the message to it so that it can write it to the server }
}

I choose to extend LineFormattedTarget, but you could also write one from the ground up. The internalLog method is where the messages are sent to the server by calling a service method that takes on the responsibility of writing messages to a server. The default internalLog method is overridden, and in order to do that successfully, you need to use the mx_internal namespace.

Next, let’s look at the server-side logging facility that BlazeDS offers.

**Server-Side Logging**

The BlazeDS logging facility resembles a standard Java logging infrastructure such as log4j. The logging configuration is specified in services-config.xml and could look like this:

```xml
<logging>
    <target class="flex.messaging.log.ServerLogTarget" level="All">
        <properties>
            <prefix>[BlazeDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.*</pattern>
            <pattern>Service.*</pattern>
            <pattern>Configuration</pattern>
        </filters>
    </target>
</logging>
```
You will notice that the following can be specified:

- **Logging level** — This acts as the filter level at which the messages are logged.
- **String prefix** — In the preceding example, it’s BlazeDS, but you could specify any other prefix. It will be appended to every message that BlazeDS writes.
- **Date and time** — The date and time of the log messages can be recorded.
- **Level** — The level of information can be specified. If, for example, you chose to record “ALL” messages, you might want to know which messages are of what level type.
- **Category** — This is the category that the message pertains to. This is a way of logically partitioning the log messages based on the class or subsystem that generates each one.

In the sample logging configuration, `ServerLogTarget` is the logging target. This means all that log messages are directed to the same destination as the Servlet logging facility. You could also choose to log messages to the console by specifying `ConsoleTarget` as the target.

You could write a custom log target to write the log messages to the Firebug console, a database target, or any other that you prefer.

Log files are the place you go to understand problems in your system. However, after preliminary investigation, you often need to analyze the application’s behavior at runtime. Also, you may want to monitor application runtime details for reasons for problems beyond errors. You may want to monitor the amount of memory usage or the amount of CPU cycles used up by the application. This is when the profiler is a great tool to seek help from.

**Profiling**

Flash Builder comes with a built-in profiler, called Flex Profiler, which provides detailed views into a program runtime behavior. Alternatively, an open source debugger, called DeMonster Debugger, can be used. Sometimes it makes sense to use both in conjunction with each other.

In this section, you will learn about the Flex profiler first. The Flex profiler is an agent that connects with the Flex application it’s profiling using a local socket connection. It uses the concept of sampling to collect data from your application at a certain frequency or sampling rate. This has two important side effects:

- Not every operation is recorded, especially not to an accuracy level that exceeds the sampling rate.
- The agent’s connection and collection introduce new operations and impact performance, even if minimally.

The profiler collects and aggregates method runtime statistics, including the time it took to run a particular method, and on object creation and garbage collection.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

The recorded and aggregated metrics are available for you to analyze using multiple views, which can be classified under two categories:

- Performance Profile Views
- Allocation Trace Views

**Live Objects**

Live Objects view displays information on the classes in the current application, showing which objects are instantiated and how much memory they are taking up.

A Live view shows the following attributes in a tabular format:

- **Class** — The name of the class
- **Package** — The package name in which the class resides
- **Cumulative Instances** — The number of object instances of a class that were created since application started
- **Instances** — The number of object instances of a class that are currently in memory
- **Cumulative memory** — The total amount of memory used up by all instances since the application started
- **Memory** — The total amount of memory currently used up by all object instances

The Live Objects view is updated in real time during the interactions with the application. Look at Figure 8-9 to see what the Live Objects view looks like.

![Figure 8-9](image.png)

**Memory Snapshot**

The memory snapshot view is a static point in time view. This means that, unlike the Live Objects view, the Memory Snapshot view is not a view of updating data. You can take a memory snapshot at any point during interaction with an application. The Memory Snapshot view shows the following attributes in a tabular format:

- **Class** — The name of the class
- **Package** — The name of the package in which the class resides
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

- **Instances** — The number of object instances at the time the memory snapshot was recorded
- **Memory** — The amount of memory used at the time the memory snapshot was recorded

Look at Figure 8-10 to see what the memory snapshot view looks like.

![Figure 8-10](image)

**Loitering Objects**

A few objects may remain in memory between two memory snapshots. These objects are visible in the loitering objects view. The loitering objects view shows the following attributes in a tabular view:

- **Class** — The name of the class
- **Package** — The name of the package in which the class resides
- **Instances** — The number of instances created between two memory snapshots
- **Memory** — The amount of memory allocated during the snapshot interval

Look at Figure 8-11 to see what the loitering objects view looks like.

![Figure 8-11](image)

**Allocation Trace**

The allocation trace view provides statistics for all objects created and memory used by method calls between two snapshot views. The allocation trace view shows the following attributes in a tabular view:

- **Method** — The name of the method.
- **Package** — The package in which the class resides.
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

- **Cumulative Instances** — The total number of objects instances by a method and all the methods it calls in turn.
- **Self Instances** — The total number of objects instances creates by the method itself. This does not count the objects created by the methods that this method calls.
- **Cumulative memory** — The total memory used by all object instances created by a method and all methods called in turn.
- **Self memory** — The total memory used by all object instances created by the method alone. The objects created by the methods called from this method are not counted.

Look at Figure 8-12 to see what the allocation trace view looks like.

![Figure 8-12](image)

**Object References**

Object references, which appear when you click on the class name in the Live Object or Memory Snapshot view, shows the referenced objects. The following attributes are shown:

- **Instance** — The class of the object that holds a reference to the particular object
- **Property** — The property of the object that holds a reference to the particular object
- **ID** — The reference ID of the object that holds a reference to the particular object

Look at Figure 8-13 to see what the object references view looks like.

![Figure 8-13](image)
Chapter 8: Testing and Debugging Flex and BlazeDS Applications

Object Statistics

The object statistics show the caller and callee of a group of objects. This view is accessed by double-clicking on the class name in the allocation trace view. There are three parts of the Object Statistics view:

- **Summary view** — Summary statistics contains the same data as the corresponding row in the allocation trace view.

- **Self Instances** — This contains information only on the objects instantiated by the particular method call and not the subsequent chain of method calls. The following pieces of information are shown:
  - **Class** — The name of the class
  - **Package** — The name of the package in which the class resides
  - **Self Instances** — The number of object instances directly instantiated by the method itself
  - **Self Memory** — The amount of memory consumed directly by the object instances created by the method itself

- **Callee Instances** — This provides information on object instances created by method calls from within a method. The following pieces of information are shown:
  - **Class** — The name of the class
  - **Package** — The name of the package in which the class resides
  - **Cumulative instances** — The total number of instances created in the method and in the subsequent calls
  - **Cumulative memory** — The total amount of memory consumed in the method and in the subsequent calls

Performance Profile

This view provides information on how the methods performed during a particular time interval. This view is accessed by double-clicking on the performance snapshot in the profile view. The following information is shown:

- **Method** — The name of the method
- **Package** — The name of the package in which the class that contains the method resides
- **Calls** — The number of the times the method was called
- **Cumulative time** — The total time taken during the method call and all its subsequent calls from within the method
- **Self time** — The total time taken during the method call alone
- **Average cumulative time** — The average time taken during the calls to the method and its subsequent calls from inside the method
- **Average self time** — The average time taken during the calls to the method
You can also click on the method name to see the specific statistics. In addition, the memory graph also shows the memory usage pattern.

All the discussion so far on profiling has only taken Flex Profiler into consideration. You could also use DeMonster Debugger. Besides this, for the BlazeDS part of the code, you could use the Eclipse Java Profiler and leverage Find Bugs, http://findbugs.sourceforge.net/, to debug the Java application at runtime.

This chapter was an aggregation of numerous strategies and approaches to testing and debugging and, while it cannot claim complete coverage, it certainly is a place where you have learned a lot about getting your Flex applications ready for release.

### Summary

This chapter is description of an eclectic mix of tools and libraries that help you test and debug Flex and BlazeDS applications. Although hints on the Java aspects of testing and debugging are included in various sections in the chapter, the focus is primarily on Flex applications.

The chapter started with a discussion on unit testing frameworks and methods and explained FlexUnit and JUnit. FlexUnit was explained in a fair amount of detail. FlexUnit 1 and FlexUnit 4 were covered. The section on FlexUnit 4 illustrated the use of new types of assertions, such as hamcrest assertions and theories, in addition to the standard assertions.

After unit testing, the chapter picked up the topic of functional testing. Two tools, FlexMonkey and FunFX, were explored. After the functional testing exposition, the topic of automation using ANT scripts and integration with continuous integration tools, such as CruiseControl, was considered.

After unit and functional testing, two more important testing topics, code coverage and load testing, were covered. The topic on load testing focused on load generation in Flex and BlazeDS applications. Apache JMeter was used as a tool for load testing.

The second part of the chapter focused on debugging. The part on debugging started with logging. In that subsection, both client- and server-side logging were explained.

The last part of the chapter illustrated the Flex Profiler and showed the different views that the Profiler offers for debugging.
Extending BlazeDS with Custom Adapters

In the last few chapters you learned to use BlazeDS to communicate between Flex clients and Java servers, using remote procedure calls and messages. In those chapters, you have relied on the built-in adapters and factories for communication and server-side method invocation. This is no surprise, since these built-in adapters and factories suffice for a large number of communication patterns and interaction with Java server-side objects.

However, there are use cases where you need to extend and customize existing adapters and create newer ones. Adobe recognized this need to create custom adapters from the time it created the very first version of Flex Data Services, which is a precursor to BlazeDS. Therefore, a fairly robust set of APIs is available to extend and create custom adapters for BlazeDS. In this chapter, you will learn all about creating custom adapters. In addition, you will also learn to create custom factories.

Contextual Overview

To understand the extension point architecture, it is best to review the BlazeDS server-side architecture first. That way, the overall context and environment of the adapters is well understood. You are familiar with the BlazeDS server-side architecture from the last few chapters. Here, I only rapidly recap the key aspects of that architecture.

BlazeDS is a Java web application. It receives requests and messages from a Flex application via its Java Servlets–based endpoint. An endpoint is accessible via a URL. Each endpoint supports communication with a particular channel type, which represents the client side of Flex application. A particular channel is distinguished by the protocol and data exchange format it supports. For example, a channel-endpoint pair could support binary transmission over AMF or text-based exchange using AMFX. Also, a channel-endpoint pair could support pure request-response communication, exchange of messages, or request polling. Once a request or message arrives via an endpoint, it is intercepted and handled by the message broker, which is the central coordinator of all communications in BlazeDS. The message broker routes requests and messages to services and
destinations. A service and destination pair acts as the server-side target for a message from a Flex client. For example, a RemotingService is the target of a message from a Flex RemoteObject. Associated with this RemotingService is a configured RemotingDestination. The service and destination pair abstracts a server-side entity, say a POJO, and provides a handle to such a POJO so that a Flex client can invoke methods on it. The last element in the communication chain between the destination and the server-side entity is an adapter. The adapter is the final infrastructure element within BlazeDS that serves a request and acts as the translator between the service and the server-side entity. Figure 9-1 illustrates this.

![Diagram of BlazeDS architecture]

From a configuration and usage standpoint, an adapter, built-in or custom, is configured in services-config.xml and is used with one of the services and destinations. The adapter is wired to a destination in one of the configuration files that pertains to proxy service, remoting, or message service. The configuration files for proxy service, remoting, and message service are proxy-config.xml, remoting-config.xml, and messaging-config.xml, respectively. In BlazeDS, these three configuration files are included in services-config.xml.

A configured adapter’s lifecycle is managed by BlazeDS. BlazeDS instantiates and initializes an adapter instance and allows you to expose it as a JMX resource. Built-in adapters are, by default, wired as JMX managed resources. Adapters are instantiated and configured during the MessageBrokerServlet bootstrapping process.
Chapter 9: Extending BlazeDS with Custom Adapters

Figure 9-2 depicts how an adapter is used and wired up in a BlazeDS instance.

Now that you have a sense of where and how an adapter is used in a BlazeDS instance, you can progress to learning more about the BlazeDS API. You will use the API to create your own custom adapter.

The API

BlazeDS exposes a public Java API. You can leverage that API to customize the behavior of BlazeDS resources.

Before you start using the API it’s a must that you explore it thoroughly and understand the role of some of its main classes. A good starting point of exploration is a sketch of its key classes and their interrelationships. Figure 9-3 provides such a sketch.

The `flex.messaging.services.ServiceAdapter` abstract class sits at the root of the hierarchy depicted in Figure 9-3. All built-in adapter classes inherit from the `ServiceAdapter` abstract class. Abstract classes not only define a contract like interfaces do but also define behavior through partial method implementations. The `ServiceAdapter` class states the base behavioral characteristics of a BlazeDS adapter.

The `ServiceAdapter` class extends the `ManageableComponent` class. The `ManageableComponent` abstract class implements the `Manageable` and the `FlexComponent` interfaces. The `Manageable` interface allows a server-side BlazeDS component to be managed and controlled by a peer MBean. The `FlexComponent` interface defines the lifecycle methods that allow you to instantiate, initialize, start, and stop a component.
Apart from the lifecycle’s “start” and “stop” methods, which start and stop an adapter, the ServiceAdapter class defines the following methods:

- A getter and setter pair to get and set a destination.
- A getter and setter pair to get and set the adapter state.
- The manage method to perform an internal action based on a command passed to it by the adapter’s service.
- The invoke method to process the data message for the adapter. This is the most important method of all. All core adapter functions that translate data format or apply logic on the message are referenced from this method.
- The handlesSubscriptions method; this method returns a Boolean value, which when “true” states that the adapter manages subscriptions.

Three main built-in adapters, flex.messaging.services.http.HTTPProxyAdapter, flex.messaging.services.remoting.adapters.JavaAdapter, and flex.messaging.services.messaging.adapters.
Chapter 9: Extending BlazeDS with Custom Adapters

MessagingAdapter, directly extend from the ServiceAdapter. MessagingAdapter is an abstract class and is not used directly. However, it defines the essential behavioral characteristics of a messaging adapter. The ActionScriptAdapter and the JMSAdapter extend from the MessagingAdapter. The SOAPProxyAdapter extends from the HTTPProxyAdapter. Therefore, all built-in adapters in BlazeDS extend directly or indirectly from ServiceAdapter.

The hierarchical structure of adapter classes creates a clear separation of features. It provides a structure where the core features are part of the base abstract classes and the communication style and server-side entity-specific features are part of the extensions to the base classes. Therefore, depending on the set of features you want, you can extend ServiceAdapter or one of its subclasses to create a custom adapter.

The HTTPProxyAdapter defines a few additional methods besides the core methods available in the ServiceAdapter class, to manage connections, set cookie limits, manage external proxies, allow content chunking, and permit self-signed certificates in SSL-based communication.

From Chapter 3, you know that an HTTPProxyAdapter can be initialized with a set of properties that can be configured with an HTTPProxyService instance in proxy-config.xml as follows:

```xml
<connection-manager>
  <max-total-connections>150</max-total-connections>
  <default-max-connections-per-host>2</default-max-connections-per-host>
  <connection-timeout>0</connection-timeout>
  </socket-timeout>
  <stale-checking-enabled>false</stale-checking-enabled>
  <send-buffer-size></send-buffer-size>
  <receive-buffer-size></receive-buffer-size>
  <tcp-no-delay>true</tcp-no-delay>
  <linger>-1</linger>
  <max-per-host>
    <host>hostname</host>
    <port>80</port>
    <protocol>http</protocol>
    <protocol-factory class="flex.messaging.services.http.ProtocolFactory">
      <properties/>
    </protocol-factory>
    <max-connections>2</max-connections>
    <proxy>
      <host>hostname</host>
      <port>80</port>
    </proxy>
    <local-address></local-address>
    <virtual-host></virtual-host>
  </max-per-host>
</connection-manager>
```

```xml
<cookie-limit>200</cookie-limit>
<allow-lax-ssl>false</allow-lax-ssl>
<content-chunked>false</content-chunked>
<external-proxy>
  <server>/server/>
  <port>80</port>
  <nt-domain></nt-domain>
```
These properties define the core behavior of the adapter and help it connect with external HTTP resources, which is what its primary role is. Refer to Chapter 3 to read more about these properties and other settings on the HTTPProxyService.

Information for a proxy request is stored in a ProxyContext. A ProxyFilter performs the tasks of pre- and post-processing on the ProxyContext when a message passes through an instance of the HTTPProxyAdapter. The SOAPProxyAdapter extends the HTTPProxyAdapter and acts as a placeholder for supporting future web services features.

The HTTPProxyAdapter or the SOAPProxyAdapter is a good extension point if you desire to include specific properties and behavior with HTTP requests or web services. Such services could include a specific authentication mechanism, encryption, and support for specific formats or protocols.

Besides HTTP request-response and web services–based communication, BlazeDS allows remote procedure call–based communication with Java server-side objects, using the JavaAdapter. You are already familiar with the JavaAdapter from Chapter 4.

JavaAdapter extends ServiceAdapter. It allows a Flex client to invoke the methods of a server-side Java class. By default, all methods of a target Java class can be invoked through the JavaAdapter. However, if required, the JavaAdapter allows you to include only a select set of a class’s methods or exclude a few selected methods. Therefore, a JavaAdapter class has methods to include and exclude target Java class methods and manage the list of such included or excluded methods. The available methods for managing the inclusion and exclusion of target class methods are:

- addExcludeMethod(RemotingMethod value) — Add a method to the list of excluded methods
- addIncludeMethod(RemotingMethod value) — Add a method to the list of included methods
- getExcludeMethodIterator() — Get an Iterator over the currently registered exclude methods
- getIncludeMethodIterator() — Get an Iterator over the currently registered include methods
- removeExcludeMethod(RemotingMethod value) — Remove a method from the list of excluded methods
- removeIncludeMethod(RemotingMethod value) — Remove a method from the list of included methods

Flex and Java applications can interact with messages and remote procedure calls. Message-based interactions between Flex and Java leverage JMS on the server side. To talk with JMS, BlazeDS includes a JMSAdapter. JMSAdapter extends MessagingAdapter and plays well with the JavaAdapter to pass Java objects through as messages. In Chapter 6, you learned a whole lot about message-based integration between Flex and Java. Refer to that chapter for details on this topic.

The MessagingAdapter abstract class supports publish/subscribe messaging. It’s the base class for all publish/subscribe messaging adapters in BlazeDS. It has two implementations in ActionScriptAdapter and JMSAdapter. ActionScriptAdapter facilitates publish/subscribe messaging between two Flex clients via a BlazeDS server. A MessageService instance controls the message delivery for an
An ActionScriptAdapter relies on a simple routing logic, where it sends a message using a MessageService’s following two methods:

- `pushMessageToClients` — delivers message to all clients connected to a server
- `sendPushMessageFromPeer` — delivers messages to peer servers in the cluster, which in turn deliver the message to connected clients

Unlike the ActionScriptAdapter, the JMSAdapter is a lot more complicated in terms of the style and idioms it follows for message delivery. The JMSAdapter delivers messages using the JMS communication model. The JMSAdapter manages the queue and topic producers and consumers and sets JMS specific properties. Refer to Chapter 6 for details on how a JMSAdapter can be configured. You can extend a JMSAdapter to include additional features and properties if a particular JMS provider included provider specific extensions that you are required to use with your Flex application.

In this section, all the built-in adapters and their API have been surveyed so far. This should help you get started on creating a custom adapter.

**A Sample Custom Adapter**

In this section, I create a simple BlazeDS adapter to illustrate how you could create one, too. The custom adapter extends `JavaAdapter` and allows you to call multiple POJOs (Plain Old Java Objects, in case you forgot!) class methods, using a single destination. Under normal conditions, one destination maps to a single Java class. Although the custom adapter itself provides a convenient implementation that could be very handy, especially if your application involves remote procedure calls to a large number of server-side Java classes, the purpose here is to illustrate how you could create a custom adapter.

First, look at the `invoke` method of the multiple POJO adapter, shown in Listing 9-1. The `MultiplePOJOAdapter` is available as a part of the open source dsadapters project, which aims to provide a number of useful extensions to BlazeDS. Details on the dsadapters project are available at http://code.google.com/p/dsadapters/.

**Listing 9-1: invoke Method of the MultiplePOJOAdapter**

```java
@Override
public Object invoke(Message message)
{
    Object result = null;
    try
    {
        RemotingDestination remotingDestination = (RemotingDestination)getDestination();
        RemotingMessage remotingMessage = (RemotingMessage)message;
        FactoryInstance factoryInstance = remotingDestination.getFactoryInstance();
        String remoteObjectSourceProperty = remotingMessage.getSource();
        boolean sourceMatchesRegEx = false;
        String classToInvoke = null;
        Continued
```
Listing 9-1: invoke Method of the MultiplePOJOAdapter (continued)

```java
if(remoteObjectSourceProperty != null && remoteObjectSourceProperty.length() > 0)
{
    if(sourceRegEx != null && sourceRegEx.length() > 0)
    {
        try
        {
            sourceMatchesRegEx = remoteObjectSourceProperty.matches
                (sourceRegEx);
        }
        catch(PatternSyntaxException patternEx)
        {
            throw new MessageException("Invalid regular expression specified " +
                "in the source attribute");
        }
    }
    else
    {
        //since we allow all classes by default
        sourceMatchesRegEx = true;
    }
    if(sourceMatchesRegEx)
    {
        classToInvoke = remoteObjectSourceProperty;
    }
    else
    {
        //user is not allowed to invoke this class
        throw new IllegalAccessException("This class does not match the specified regular expression");
        Log.getLogger(LogCategories.MESSAGE_REMOTING).error("source specified in the RemoteObject does not match the configured source regular expression");
    }
}
else
{
    //we are not getting the source value from Flex
    //use the default source
    if(defaultSource != null && defaultSource.length() > 0)
    {
        classToInvoke = defaultSource;
    }
    else
    {
        throw new MessageException("No source was specified in the RemoteObject and " +
            "default source was specified either.");
    }
}
```
//check if the class to invoke is in the excluded classes list
if(excludeClasses.contains(classToInvoke))
{
    Log.getLogger(LogCategories.MESSAGE_REMOTING).error("The class is
    in the exclude classes list");
}

String methodName = remotingMessage.getOperation();
List parameters = remotingMessage.getParameters();

Object instance = createInstance(classToInvoke);
if (instance == null)
{
    throw new MessageException("Null instance returned from: " +
            factoryInstance);
}
Class c = instance.getClass();

MethodMatcher methodMatcher = remotingDestination.getMethodMatcher();
Method method = methodMatcher.getMethod(c, methodName, parameters);
result = method.invoke(instance, parameters.toArray());

saveInstance(instance);
}
catch (InvocationTargetException ex)
{
    Throwable cause = ex.getCause();
    if ((cause != null) && (cause instanceof MessageException))
    {
        logMessage(null, cause);
        throw (MessageException) cause;
    }
    else if (cause != null)
    {
        throw new MessageException(cause.getClass().getName() + " : " +
            cause.getMessage());
    }
    else
    {
        throw new MessageException(ex.getMessage());
    }
}
catch (IllegalAccessException ex)
{
    throw new MessageException(ex.getMessage());
}
return result;

The MultiplePOJOAdapter class allows you to specify a regular expression pattern as the value of
the source property in your remoting destination. The name of the class whose methods are invoked
remotely is specified as the source property of a RemoteObject instance in Flex. A class specified as a value of a RemoteObject’s source property needs to satisfy the restrictions imposed by the regular expressions specified as a value to the source property of a remoting destination on the server. Besides, you can explicitly configure classes you don’t want to be exposed as remoting destinations.

You know that the invoke method of an adapter is called every time a message is processed by an adapter. So, this is where it’s verified that a class name matches the specified regular expression or matches the exclude-classes list. The invoke method of the MultiplePOJOAdapter goes through a series of conditional branches to find out if the destination class name specified in a RemoteObject matches the regular expressions and if it is specified in the exclude-classes list or not. A default class is also specified and calls fall back to this class if the class name specified via the Flex client side is null or undefined.

Each class instance is created, by default, in the request scope. However, you can bind a specific class to a session or the application scope as well. The adapter configuration allows specification of scopes for classes within the standard configuration files, i.e., remoting-config.xml (which you know is included in services-config.xml). A custom property called classes-scopes holds the scope mapping for all those classes whose scopes differ from the default request scope.

Once a class successfully passes through all the guard conditions specified by the destination source regular expression and the exclude-classes list, an instance of the class is created, which is then used to invoke the remote procedure calls. A call to a local method called createInstance from within the invoke method creates an instance of the class. The createInstance method source is:

```java
protected Object createInstance(String classToInvoke) {
    RemotingDestination remotingDestination = (RemotingDestination) getDestination();
    FactoryInstance factoryInstance = remotingDestination.getFactoryInstance();
    JavaFactory factory = (JavaFactory)remotingDestination.getFactory();

    //get scope for this class
    //we are allowing to set scope for classes in configuration file
    String scope = getClassScope(classToInvoke);

    ConfigMap localProperties = new ConfigMap(factoryInstance.getProperties());
    localProperties.put(FlexFactory.SOURCE, classToInvoke);
    localProperties.put(FlexFactory.SCOPE, scope);

    FactoryInstance localFactoryInstance = factory.createFactoryInstance(
        factoryInstance.getId(),
        localProperties);

    Object instance = localFactoryInstance.lookup();

    if (isStarted() && instanceof FlexComponent
        && !((FlexComponent)instance).isStarted())
    {
        ((FlexComponent)instance).start();
    }
    return instance;
}
```
Chapter 9: Extending BlazeDS with Custom Adapters

The `createInstance` method relies on Java reflection to create an instance of the class, which is identified by its name. An instance is created within the specified scope. Once an instance is created, its `start` method is called to initialize the component and get it ready.

Before an adapter itself is ready for use, its instantiated and initialized. Two methods called `initialize` and `start` run the initial setup tasks. The `initialize` and `start` methods for the multiple POJO adapter are:

```java
@override
class MultiplePOJOAdapter {

    @Override
    public void initialize(String id, ConfigMap properties) {
        super.initialize(id, properties);

        defaultSource = properties.getPropertyAsString(PROPERTY_DEFAULT_SOURCE, null);
        defaultScope = properties.getPropertyAsString(PROPERTY_DEFAULT_SCOPE, null);
        sourceRegEx = properties.getPropertyAsString(PROPERTY_SOURCE, null);

        ConfigMap classesToExclude = properties.getPropertyAsMap(PROPERTY_EXCLUDE_CLASSES, null);
        if (classesToExclude != null) {
            List<String> classes = classesToExclude.getPropertyAsList(CLASS_ELEMENT, null);
            if ((classes != null) && !classes.isEmpty()) {
                int n = classes.size();
                for (int i = 0; i < n; i++) {
                    ConfigMap classSettings = (ConfigMap)classes.get(i);
                    String name = classSettings.getPropertyAsString(NAME_ATTRIBUTE, null);
                    addExcludeClass(name);
                }
            }
        }

        ConfigMap classesWithDefinedScopes = properties.getPropertyAsMap(PROPERTY_CLASSES_SCOPES, null);
        if (classesWithDefinedScopes != null) {
            List<String> classes = classesWithDefinedScopes.getPropertyAsList(CLASS_ELEMENT, null);
            if ((classes != null) && !classes.isEmpty()) {
                int n = classes.size();
                for (int i = 0; i < n; i++) {
                    ConfigMap classSettings = (ConfigMap)classes.get(i);
                    String name = classSettings.getPropertyAsString(NAME_ATTRIBUTE, null);
                    String scope = classSettings.getPropertyAsString(SCOPE_ATTRIBUTE, null);
                    addToClassesScopesMap(name, scope);
                }
            }
        }
    }
}
```
public void start()
{
    if (isStarted())
    {
        return;
    }
    super.start();
    validateInstanceSettings();
}

You will notice that the initialize method is where the configured properties are read, parsed and saved up in local variable. The initialize method, like the invoke method, overrides the corresponding implementations in the JavaAdapter class. The start method on the other hand calls the JavaAdapter class's start method via a call to its super class method, in addition to validating the settings for the instance.

By now you may have realized that writing a custom adapter is fairly straightforward and most of the complexity lies in the business logic your adapter needs to support. An adapter can be created by extending a built-in adapter or one of its abstract super classes.

Custom Factories

BlazeDS destinations by default hook up to simple Java classes or POJOs. These destination classes are instantiated, and their lifecycle is managed, by BlazeDS.

The creation and lifecycle management processes for objects like EJBs and Spring beans (i.e., components used by the Spring Framework) are handled by their specialized containers. Further, these objects exist in their own independent namespace. In such cases, BlazeDS needs to leverage the object-specific creation and lifecycle management policy to use it within its domain.

BlazeDS includes a factory mechanism that you can leverage to instantiate and manage EJBs, Spring beans, and other such managed objects. All you need to do is create a special factory class for your managed component. Such special factory classes need to implement the flex.messaging.FlexFactory interface. Once available, the special factory class is used to create a FactoryInstance, which is associated with a destination and is used to get a reference to the managed object.

Here, I create a simple specialized factory to use EJB3 objects as destinations. You will benefit from looking at the source code first. The source for the EJB3Factory class is shown in Listing 9-2.

Listing 9-2: A Custom Factory to Use EJB3 Objects as BlazeDS Destinations

    package dsadapters.remoting_factories;
    import flex.messaging.FactoryInstance;
    import flex.messaging.FlexFactory;
    import flex.messaging.config.ConfigMap;
    import flex.messaging.services.ServiceException;
    import java.text.MessageFormat;
    import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class EJB3Factory implements FlexFactory
{

    /* (non-Javadoc)
     * @see flex.messaging.FlexFactory#createFactoryInstance(java.lang.String,
     * flex.messaging.config.ConfigMap)
     */
    @Override
    public FactoryInstance createFactoryInstance(String id, ConfigMap configMap)
    {
        FactoryInstance factoryInstance = new FactoryInstance(this, id, configMap);
        factoryInstance.setSource(configMap.getPropertyAsString(SOURCE_CLASS,
            factoryInstance.getId()));

        return factoryInstance;
    }

    /* (non-Javadoc)
     * @see flex.messaging.FlexFactory#lookup(flex.messaging.FactoryInstance)
     */
    @Override
    public Object lookup(FactoryInstance factoryInstance)
    {
        Object ejb3Object = null;
        try
        {
            ejb3Object = locate(factoryInstance.getSource());
        }
        catch(Exception e)
        {
            throw createServiceException(MessageFormat.format("error referencing
                EJB {0}, {1}",
                new Object[] { factoryInstance.getSource(), e.getMessage() } ),
                e);
        }

        return ejb3Object;
    }

    /* (non-Javadoc)
     * @see flex.messaging.FlexConfigurable#initialize(java.lang.String, flex.
     * messaging.config.ConfigMap)
     */
    @Override
    public void initialize(String id, ConfigMap configMap)
    {
        /**
         * No initialization required for this EJB3Factory
         */
    }
}

Continued
protected ServiceException createServiceException(final String message, final Throwable cause )
{
    ServiceException serviceException = new ServiceException();
    serviceException.setMessage(message);
    serviceException.setRootCause(cause);
    serviceException.setDetails(message);
    serviceException.setCode(ERROR_CODE);

    return serviceException;
}

protected Object locate(String name) throws Exception
{
    try
    {
        final Context ctx = new InitialContext();
        final Object reference = ctx.lookup(name);
        return reference;
    }
    catch (NamingException e)
    {
        e.printStackTrace();
        throw new Exception( MessageFormat.format(  
            "error locating local resource {0}", new Object[]  
            {name} ), e );
    }
}

private static final String SOURCE_CLASS = "source";
private static final String ERROR_CODE = "EJB.Invocation";
}

The FlexFactory interface defines three methods as a part of the factory class contract. These three methods are:

- createFactoryInstance — A method that returns a FactoryInstance object for the factory
- lookup — A method that finds and retrieves the managed object
- initialize — A method where any properties and initialization parameters can be set for the factory

In the preceding EJB3 factory, the lookup method merely looks up an EJB object from a JNDI (Java Naming and Directory Interface) repository and returns a reference to it. The locate method implements the specifics of how the EJB instance is looked up.

In the locate method, you create an instance of the InitialContext class and use that instance to look up an EJB object by its JNDI name. An InitialContext instance provides a starting point into the namespace from where the naming and directory service can be accessed.
Chapter 9: Extending BlazeDS with Custom Adapters

In the preceding code listing, a local JNDI resource is accessed. You may be aware that JNDI is supported by Java EE application servers, for example the JBoss AS. If your JNDI repository is remote, be sure to set the URL to the JNDI repository as the PROVIDER_URL in a Hashtable instance and then pass that Hashtable to the constructor of the InitialContext instance. To illustrate this further, let jnp://someHostName:1099 be the URL to your JNDI repository; you can then bind this repository to your InitialContext as follows:

```
Hashtable env = new Hashtable();
....
env.put(Context.PROVIDER_URL, providerURL);
....
InitialContext initialContext = new InitialContext(env);
```

Now that the custom factory is ready, let's see how you could use it. First, add the custom factory within the factories property of the service-config.xml and then use the factory with a destination.

A factory could be added to the set of factories in services-config.xml as follows:

```
<factories>
  <factory id="myCustomFactory" class="dsadapters.remoting.factories.EJB3Factory">
    <properties>
      ....
    </properties>
  </factory>
</factories>
```

The custom EJB3 factory does not specify any attributes that need to be passed in to the factory instance, so you do not add any properties to the configuration. If you created a factory that accepted an attribute (say myAttribute) and you wanted to configure a value, myAttributeValue, for that attribute, then you would need to configure the factory as follows:

```
<factories>
  <factory id="myCustomFactory" class="dsadapters.remoting.factories.EJB3Factory">
    <properties>
      <myAttribute>
        myAttributeValue
      </myAttribute>
    </properties>
  </factory>
</factories>
```

Factory-specific attribute values are set to a factory using its initialize method. In the initialize method, you can access the passed-in attribute, using the getPropertyAsString method of the passed in ConfigMap object, which holds all the configured properties specified in the configuration file.

Once the factory is configured it can be used with a destination as follows:

```
<destination id="myDestination">
  <properties>
    <source>packageName.RemoteEJBClassName</source>
    <factory>myCustomFactory</factory>
  </properties>
</destination>
```
That’s about it! You have learned how to extend a remoting adapter and how to create a custom factory. Next, you learn to create a custom messaging adapter.

## Extending the Messaging Adapter

The ActionScriptAdapter is the simplest of the built-in messaging adapters. It extends the MessagingAdapter abstract class.

Almost all of the functionality in the ActionScriptAdapter class resides in its invoke method. In its invoke method, an ActionScriptAdapter uses a MessageService instance that takes control of the message delivery to all connected clients and server peers. In the invoke method, a call is made to the MessageService’s pushMessageToClients method, which sends a message to all connected clients within a local node of a cluster. In addition, the MessageService instance’s sendPushMessageFromPeer method is invoked to send messages to the peer servers in the cluster. These servers, in turn, deliver the message to their connected clients.

In this section, let’s create another simple messaging adapter like the ActionScriptAdapter. In the custom adapter, I continue to send messages the way the ActionScriptAdapter does. The only modifications I make are:

- Modify the message body and add the text “Before message body” and “After message body” to the existing message body
- Add a header named “interceptor” with “CustomActionScriptAdapter” as the value to the message

The source for CustomActionScriptAdapter is shown in Listing 9-3.

### Listing 9-3: CustomActionScriptAdapter Source

```java
package dsadapters.messaging.adapters.as3;

import flex.messaging.messages.AsyncMessage;
import flex.messaging.messages.Message;
import flex.messaging.services.MessageService;
import flex.messaging.services.messaging.adapters.MessagingAdapter;

public class CustomActionScriptAdapter extends MessagingAdapter {

    /**
     * @param enableManagement
     */
    public CustomActionScriptAdapter()
    {
        
    }

    /**
     * @param enableManagement
     */
```
public CustomActionScripAdapter(boolean enableManagement)
{
    super(enableManagement);
}

/* (non-Javadoc)
 * @see flex.messaging.services.ServiceAdapter#invoke(flex.messaging.messages.Message)
 */
@Override
public Object invoke(Message message)
{
    AsyncMessage modifiedMessage = (AsyncMessage)message;
    modifiedMessage.setBody("Before message body " + message.getBody() + " After message body");
    modifiedMessage.setHeader("interceptor", "CustomActionScriptAdapter");
    MessageService msgService = (MessageService)getDestination().getService();
    msgService.pushMessageToClients(modifiedMessage, true);
    msgService.sendPushMessageFromPeer(modifiedMessage, true);
    return null;
}

If you browse through the invoke method in the preceding listing, you will notice that a new AsyncMessage instance is created from an existing incoming message. Then the body of the new message is modified to add “before message body” before the message body and “after message body” after the message body. Also, a custom header is added to the modified message. After these modifications, the message is sent using a MessageService instance.

Next, let’s explore a few advanced strategies for custom adapter creation.

Advanced Strategies for Custom Adapter Creation

From Chapter 7, which covered Flex and Hibernate integration, you are familiar with a few advanced adapters, which in that particular case worked within the semantics of Hibernate, the object-relational mapping tool.

One of the adapters illustrated there is the Gilead Hibernate PersistentAdapter. The Gilead PersistentAdapter extends the built-in JavaAdapter. In its invoke method, the PersistentAdapter stores the Hibernate POJO in the HTTPSession (if the adapter needs to maintain state) and then merges the input parameters from the remoting message before it calls the invoke method of the JavaAdapter. This adapter class relies on the JavaAdapter for the standard instantiation and invocation functions and just adds the value-added functions on top of what is already present in a JavaAdapter.

Going with this use case, a ready piece of advice is to leverage existing adapters for standard features, especially if your adapter also needs them.
An adapter class is a Java class, and you can call any Java code from within it. Sometimes, an adapter needs to work exhaustively with the API of the destination, as is the case with a JMSAdapter. In such cases, the complexity is in the target destination interaction and not in the adapter itself.

An adapter is instantiated and managed within a Servlet container, so it’s advisable not to include any direct threading within that code. Also, for the sake of clean design and clear separation of duties, remember to keep all external resource access code in classes separate from the adapter class.

Summary

This chapter was all about creating custom adapters and factories. The chapter started with a contextual overview and provided a starting point for you to understand where adapters fit into the overall BlazeDS architecture. After the contextual overview, the BlazeDS API was discussed. BlazeDS is a Java web application. It exposes a Java API to extend and customize its behavior and add newer adapters and factories to it.

Following the initial overview sections, the focus turned to creating a custom adapter. That section included an implementation of a MultiplePOJOAdapter, which extends the JavaAdapter to support multiple classes via a single destination. That section lists substantial parts of the adapter code and provides an in-context explanation of some of the key aspects of the source class.

After an illustration of a custom adapter, the focus shifted to custom factories. Custom factories are useful when you need to access objects that exist in a namespace external to BlazeDS. Managed objects like EJBs and Spring beans are components that need a custom factory to make them accessible via remoting destinations. The section includes details on a custom EJB3 factory.

BlazeDS supports remote procedure calls and message-based interactions. While a custom adapter example in an earlier section talked about the remote procedure call–based interactions, the section on extending message adapters explained the custom messaging adapter. Two messaging adapters come off the shelf with BlazeDS. These are the JMSAdapter and ActionScriptAdapter. In the section on custom messaging adapters, a custom adapter, similar to the ActionScriptAdapter, was created.

Finally a small section just prior to this summary mentioned a few thoughts on creation of advanced custom adapters.

By the end of this chapter, I hope you are convinced that creating custom adapters and factories is not only possible but also easy. The complexity, if any, is imposed by what you want to achieve and not the extension points and the API.
For many enterprise Flex applications, it’s important that the applications scale as the number of its users and volume of data interchange increase. This chapter explains the following factors that contribute positively to better scalability:

- Clustering
- Data compression
- Data format optimization
- Robust connection
- Service orientation
- Caching
- Resource Pooling
- Workload distribution

Let’s start with clustering.

**Clustering**

Replacing a single server with a cluster of servers is often a good technique for scaling applications to support greater number of clients. Clustering multiple servers implies that multiple instances act in unison, serving external clients as a single logical entity. Functionality is typically replicated across the clustered servers, and members of a cluster often share state information.

Clustering efforts could involve either vertical clustering or horizontal clustering. In vertical clustering, multiple instances of a software server are installed on the same piece of hardware. In contrast to this, in horizontal clustering multiple instances of a software server are installed on different pieces of hardware.
Vertical clustering helps optimize hardware’s CPU and memory usage. It does not provide fallback options on a hardware failure though. Horizontal clustering, on the other hand, provides a failover option among servers, in addition to providing scaling capabilities.

Besides software clustering, one also has the choice of hardware clustering, in which case a load balancer acts as the front end for calls to the clustered hardware infrastructure. Typically, a load balancer optimizes the way requests from clients are distributed among the clustered members. Once a load balancer associates a client to a cluster member, it sends all subsequent requests from that client to the same cluster member.

Software and hardware clustering can be combined. In this section, though, I will focus on software clustering.

You can cluster a set of BlazeDS instances to enhance the scalability and availability of your application. Clustered instances of BlazeDS share data and messages across instances. Therefore, message producers connected to a clustered server instance can send messages to message consumers connected to a different server within the same cluster. Besides sharing state information and routing information through the clustered servers, clustering provides for channel failover support.

Flex applications connect to a BlazeDS destination via a Channel. For example, a Flex application could connect to a RemoteObject, using an instance of mx.messaging.channels.AMFChannel. In clustered scenarios, channel failover is possible. That is, if a Flex application fails to connect to a destination via a channel to a server, connection via alternative channels and connection to alternative servers, in the cluster, is tried. Channel failover is supported for remote procedure call– and message–based services.

**Channel Failover**

When a Flex client tries to connect for the first time to a server in a cluster, it receives a set of endpoint URI(s) for possible channels that it can connect to. This returned set of channels is saved locally in the Channel.failoverURIs property in the client. The client refers to this property when it relies on channel failover to connect to a cluster of servers. The order followed for failover is illustrated in a flow diagram in Figure 10-1.

In order to support failover, all servers in a cluster should define the same set of channels for a destination and specify the same security policy.

Failover is also impacted by retry configurations and the timeout property. Flex-side RPC and message service components allow developers to configure a few such properties. Following is a list of failover-related client-side properties:

- **reconnectAttempts** — A Producer component property that defines the number of times the producer should attempt to reconnect on connection failure. The default value is 0, which implies no retries. For indefinite retries, set the value to -1.

- **reconnectInterval** — A Producer property that defines the interval between reconnect attempts. Setting the property value to 0 disables the retries. The default value is 5000 milliseconds, or 5 seconds.

- **resubscribeAttempts** — A Consumer property that defines the number of times the consumer should attempt to resubscribe on connection failure. The default value is 5. For indefinite retries, set the value to -1.
- **resubscribeInterval** — A **Consumer** property that defines the interval between resubscribe attempts. Setting the property value to 0 disables the retries. The default value is 5000 milliseconds, or 5 seconds.

- **requestTimeout** — A property of a **RemoteObject**, a **WebService**, an **HTTPService**, a **Producer**, and a **Consumer** that defines the time interval beyond which a request for which no response is received is timed out. An acknowledgment, a successful result, or a fault could be a response to a request.

- **connectTimeout** — A property of a **Channel** that defines when a hung connection should be timed out.

![Flowchart](Figure 10-1)
Each of these properties impacts the way that a fault event is finally dispatched after connection attempts are made across all servers and all channels in a cluster.

While I have explicitly mentioned that BlazeDS servers can be clustered, I have explained little about how they can be clustered and what makes the data and messages replicate successfully across the cluster. Next, I will attempt to explain the infrastructure that supports clustering BlazeDS servers.

**Clustering Powered by JGroups**

JGroups ([www.jgroups.org](http://www.jgroups.org)) is an open source toolkit for reliable multicast communication. JGroups allows the creation of groups spread across the network, where members of a group can send messages to each other using TCP, UDP, and JMS. Messages can be sent point-to-point and point-to-multipoint. The toolkit supports detection and notification as members join, leave, or crash.

BlazeDS leverages JGroups to propagate data and message across a cluster. You need to configure JGroups to enable BlazeDS clustering.

When you download the turnkey distribution of BlazeDS, you will find a `resources` folder in that distribution. Within the `resources` folder you will find a folder named `clustering`. All the JGroups-related packages and configuration files are in the `clustering` folder.

To use JGroups, first copy `jgroups.jar` to the WEB-INF/lib folder of your BlazeDS-powered Flex application. Then, copy `jgroups-tcp.xml` and `jgroups-udp.xml` to the WEB-INF/flex folder. Finally configure JGroups to use it with BlazeDS.

Use TCP multicast over UDP multicast with JGroups for reliability. You can configure JGroups for BlazeDS to use TCP multicast by making entries in `jgroups-tcp.xml` as follows:

```xml
<TCP bind_addr="<IP Address or Host Name>" start_port="7800" loopback="false"/>
<TCP/IPING timeout="3000"
  initial_hosts="cluster_server1.com[7800],
  cluster_server2.com[7800],cluster_server3.com[7800]"
  port_range="1"
  num_initial_members="3"/>
<FD.SOCK/>
<F D.timeout="6000" max_tries="4"/>
<VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
<MERGE2 max_interval="10000" min_interval="2000"/>
<picast.NAKACK gc_lag="100" retransmit_timeout="600,1200,2400,4800"/>
<picast.STABLE stability_
  delay="1000" desired_avg_gossip="20000" down_thread="false"
  max_bytes="0" up_thread="false"/>
<picast.GMS
  print_local_addr="true" join_timeout="5000" join_retry_timeout="2000" shun="true"/>
```

The `TCP bind_addr` holds the IP address or the hostname of the server on which this configuration file is deployed. The `start_port` is the port on which one can make a connection to the JGroups member. The `TCP/IPING` property points to all the other members in the group. Specifically, the `initial_hosts` property holds the IP address or the domain name of the other cluster members. Much of the rest of it is present in the default `jgroups-tcp.xml` file available with the distribution.
To use this JGroups configuration with a BlazeDS cluster, make the following entry in services-config.xml:

```xml
<?xml version="1.0"?>
<services-config>
  ...
  <clusters>
    <cluster id="blazeds-cluster" properties="jgroups-tcp.xml" default="true"/>
  </clusters>
  ...
</services-config>
```

Then refer to this cluster within a destination configuration as follows:

```xml
<destination id="aDestination">
  ...
  <properties>
    <network>
      <cluster ref="blazeds-cluster"/>
    </network>
  </properties>
  ...
</destination>
```

The cluster “blazeds-cluster” has a property named “default” whose value equals “true” and this cluster is the default cluster. Therefore, even if you only specified `<cluster/>` within the network property of a destination in the example illustrated, the “blazeds-cluster” would still be used.

If the destination that uses a cluster is a messaging destination, then you also have the option to configure server-to-server messaging, which sends subscribe and unsubscribe messages to all servers in the cluster, but sends data messages only to the active cluster members. To configure server-to-server messaging in a messaging destination, add the following:

```xml
<destination id="aTopic">
  <properties>
    ...
    <server>
      <cluster-message-routing>server-to-server</cluster-message-routing>
    </server>
    ...
  </properties>
</destination>
```

That completes most of the clustering configuration. By now you are, hopefully, convinced that BlazeDS clustering is beneficial and easy to implement.

While clustering provides connection scalability advantages, sometimes the performance bottlenecks are the result of long-running data interchanges over the connections. Such cases occur when data packets exchanged are very large and bulky or the connection bandwidth is too small, or both. In the next section, you will learn to compress the data and send it across the wire. Compressed data increases resource utilization efficiency and enhances scalability.
Chapter 10: Making BlazeDS Applications Scalable

Data Compression

Data compression is a technique that packs the data from its raw format into compact bundles, thereby reducing its size. There are many algorithms available for compressing data. One of the most popular of these is the “deflate” algorithm, which is a combination of Lempel-Ziv (LZ) 77 and Huffman encoding. The GNU zip, better known as gzip, is based on this algorithm. If you desire to read more about gzip and the algorithms it’s based on, then the Wikipedia page on the topic, accessible online at http://en.wikipedia.org/wiki/Gzip, would be a good starting point.

BlazeDS, as you know well, is a Java Servlets–based web application. As data from a BlazeDS server leaves for the Flex client, it can be intercepted using the familiar Java Servlet filter mechanism that you saw in action earlier in Chapter 7. Servlet filters can intercept requests to and responses from a Java Servlet.

In order to gzip the data transferred from the server to the client, I create a filter to manipulate and compress the outgoing response. Then I configure it for my BlazeDS instance. I don’t need any changes in the BlazeDS code. Destinations called by the Flex client that return the data are totally unaware of this filter.

To get started and gain an understanding of how the filter works, first browse through the filter code in Listing 10-1.

Listing 10-1: GZipFilter Java Source

```java
package dsadapters.filters.gzip;

public class GzipFilter implements Filter
{

    @Override
    public void destroy()
    {
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException
    {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        HttpSession session = request.getSession(true);
        String gzip = request.getParameter("gzip");
        String ae = request.getHeader("accept-encoding");
        if ((ae != null) && (ae.indexOf("gzip") != -1) && (gzip != null))
        {
```

Download from Library of Wow! eBook <www.wowebook.com>
Chapter 10: Making BlazeDS Applications Scalable

GzipResponseWrapper wrappedResponse = new GzipResponseWrapper(response);
filterChain.doFilter(servletRequest, wrappedResponse);
wrappedResponse.finishResponse();

else
{
    filterChain.doFilter(servletRequest, servletResponse);
}

@Override
public void init(FilterConfig filterConfig) throws ServletException
{
    servletContext = filterConfig.getServletContext();
}

private ServletContext servletContext;

The main method of a filter class is the doFilter method. The doFilter method calls the filter with the request and response pair and calls the next filter in the chain till it reaches the end of the chain. In the GzipFilter class, the response is wrapped up and sent to the doFilter method. The response wrapper is the class that facilitates gzip compression of the outgoing response.

To dig deeper into the response wrapper and see how the gzip compression takes place, you may want to peek into the GzipResponseWrapper source, which is available in the code download bundle. Here, though, I will illustrate only the salient features of that class.

The Java Servlet 2.3 specification provides for two classes, javax.servlet.ServletResponseWrapper and javax.servlet.http.HttpServletResponseWrapper, that provide a mechanism to override the standard response object behavior. A similar set of wrapper classes exists for overriding the standard request object as well. The GzipResponseWrapper class extends HttpServletResponseWrapper. It creates a stand in stream and writes compressed data to it. The stream class itself is called GzipResponseStream. GzipResponseStream extends ServletResponseStream and uses the java.util.zip.GZIPOutputStream utility class to compress the byte array output stream and then write it to the output that the browser receives.

To use the GzipFilter with BlazeDS, add the compiled classes to WEB-INF/classes (if packaged as independent .class files) or WEB-INF/lib (if packaged as a jar archive file) folder the following configuration to web.xml:

    <filter>
        <filter-name>GzipFilter</filter-name>
        <filter-class>dsadapters.filters.GzipFilter</filter-class>
        <init-param>
            <!-- debug prints size and other information to the console -->
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

    <!-- The minimum size, in bytes, for when to start using compression. -->
    <!-- Responses below this threshold are sent without compression. -->
    <!-- Download from Library of Wow! eBook <www.wowebook.com> -->

277
Chapter 10: Making BlazeDS Applications Scalable

At this point, I am certain that you are convinced that compressing data is fairly easy and straightforward. While compression can be applied to any data format, text or binary, and size, the effect is quite different, depending on the data format and size.

Next, a few data formats and their impact on application performance and scalability are analyzed.

Data Format Optimization

Data can be exchanged between a BlazeDS server and a Flex client in a number of formats, which include both binary and text-based forms. AMF3 facilitates a very efficient way of binary transmission of data between the server and the client. In addition, you can choose to go with a text-based format that could be well structured like XML or delimited like comma-separated or tab-delimited text. JSON (JavaScript Object Notation) and AMFX, where AMF is transmitted in XML, are also options.

In each of these cases, the following performance-related measures need to be considered:

- **Data size** — The volume or size of the data in a particular format
- **Server execution time** — The time the server takes to serve the data
- **Transmission time** — The time the data takes to traverse the wire from the server to the client
- **Parsing time** — The time the client-side program takes to parse the data so that it’s available for consumption
- **Rendering time** — The time it takes to render the data in a UI component on the client side

Although accurately measuring the size and duration intervals in the preceding list is fraught with inefficiencies and challenges, these measures do provide an indication of the scale of differences among the various options. In this section, I will present a way to measure the data size and the execution, transmission, parsing, and rendering time intervals.

For starters, let’s measure the execution time on the server. The execution time on the server is not only impacted by the format but also by the time it takes to generate the data and collect it.
Chapter 10: Making BlazeDS Applications Scalable

One of the best unobtrusive ways of measuring the time it takes to generate a response is to set up a Servlet filter to do so. Such a Servlet filter could also measure the size of data that is generated. A Servlet filter that measures data traffic is available in the dsadapters project, which is accessible online at http://code.google.com/p/dsadapters/.

The core snippet of the doFilter method where the Servlet filter measures the data traffic size and execution time is as follows:

```java
Calendar startTime = Calendar.getInstance();

int contentLength;
long execTime;

try {
    ByteArrayResponseWrapper wrapper = new ByteArrayResponseWrapper(
        (HttpServletRequest) response);
    chain.doFilter(request, wrapper);
    contentLength = wrapper.getOutputStreamAsByteArray().length;
    execTime = Calendar.getInstance().getTimeInMillis() - startTime.getTimeInMillis();
    response.getOutputStream().write(wrapper.getOutputStreamAsByteArray());
} catch (Exception e) {
    throw new ServletException(e);
} finally {

}
```

The two most important lines of code as far as the traffic measures go are sandwiched between the line where the chain.doFilter is called and the line where response.getOutputStream is called. Both these lines are in the try block of the preceding code snippet.

The filter uses a ByteArrayResponseWrapper class to wrap the response up as a byte array. The size of the data is measured simply by measuring the length of the byte array. The server execution time is simply the time elapsed between the point when the doFilter method starts executing to the point where it writes the response out to the output stream.

The transmission, parsing, and rendering time need to be captured at the client end. Transmission time can best be captured by adding an event listener for the acknowledge event of the mx.messaging.MessageAgent of the mx.messaging.ChannelSet in use. The time elapsed between the application’s initialization and the invocation of the acknowledge event listener provides a measure of the transmission time. Parsing time can be assessed by measuring the time from transmission to the point when the result event handler of the RPC component is called. Rendering time can be measured from the time the result is available at the RPC component to when a UI component such as a DataGrid dispatches its updateComplete method.

You can choose to alter these measures and come up with your own set of metrics, but as you run these tests you will notice that a few results are consistently unambiguous, which are the following:
Chapter 10: Making BlazeDS Applications Scalable

- Data transferred as binary AMF is consistently the smallest and the fastest among all the alternatives.
- The parsing time for AMF data is negligible, as the data is already available in AS3 form.
- XML parsing using E4X is often faster than hand-coded parsing for text formats like CSV or TSV.
- As the amount of data increases, the gap in size and time taken between the binary AMF and the text-based formats widens sharply, sometimes a magnitude of over 10 times in favor of AMF.
- Text formats are very sluggish in performance and often don’t scale beyond 10,000 rows or so. Although one could debate why you would need to retrieve 10,000 rows at once, the metrics stand as mentioned.

Therefore, choose the data format appropriately and almost always favor the more efficient binary AMF unless there is a good reason to support an alternative format.

In the last few sections, you have learned about clustering, data compression, and data formats, and each of these enhances performance and increases scalability. In the section on clustering, the technique of connection failover and client load distribution was explored. Next, the benefits of asynchronous connections for scalable server-side pushing are explained.

Robust Connection

In Chapter 6, the concept of Comet and nonblocking IO was briefly introduced. The context there was scalable messaging and real-time data pushing. In this section, I extend the discussion to a more generic discussion on nonblocking IO in Java. The concepts here can be leveraged to build nonblocking channels that will work well with most existing channel types and could also serve well to create nonblocking socket connections between a Flex application and a BlazeDS server. Non-blocking channels allow for a greater number of connections to be served provided they are not all active at the same time always. LCDS uses NIO channels for connection scalability. You can include the same robustness in BlazeDS as well. Only Java NIO fundamentals are covered in this section.

As a result of JSR 51 efforts and starting with Java 1.4, a new IO package is available in Java. It’s appropriately called the New IO, or NIO for short. The NIO package includes a number of new improvements to the traditional IO in Java. The one that we are interested in the context of this section is the ability to create nonblocking sockets, which allows input/output operations on a channel without blocking the underlying processes. This is a great new feature. Traditional multi-threaded servers for processing multiple simultaneous client requests can now be replaced in many situations with nonblocking sockets.

NIO nonblocking channels utilize a `java.nio.Buffer` to read and write data. A `Buffer` supports all Java primitive types. You can think of a `Buffer` as a stream. You can write to it and read from it. It has a current position, sort of a cursor, at all times. After reading a particular value, the cursor moves to the next item in the `Buffer`. A buffer defines a capacity and you can set a limit to the capacity while reading and writing data. Data is written to a buffer using the `put` method and read from it using the `get` method.

In a NIO nonblocking socket server, an important intermediary called the selector sits between a socket channel and the socket server. The selector takes incoming requests, assigns them a key and passes them on to the server to process. This is what creates the asynchronous scalable model.
The key, apart from a reference to the request, also holds the request type, which could be:

- Client attempting to connect
- Server attempting to connect
- Reading operation
- Writing operation

The server itself listens for new requests in an infinite loop. The server infinite loop could be as follows:

```java
for(;;) {
    selector.select();

    Set keys = selector.selectedKeys();
    Iterator i = keys.iterator();

    while(i.hasNext()) {
        SelectionKey key = (SelectionKey) i.next();

        i.remove();

        if (key.isAcceptable()) {
            SocketChannel client = server.accept();
            client.configureBlocking(false);
            client.register(selector, SelectionKey.OP_READ);
            continue;
        }

        if (key.isReadable()) {
            SocketChannel client = (SocketChannel) key.channel();

            int BUFFER_SIZE = 32;
            ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
            try {
                client.read(buffer);
            } catch (Exception e) {
                e.printStackTrace();
                continue;
            }

            buffer.flip();
            Charset charset=Charset.forName("ISO-8859-1");
            Charbuffer decoder = charset.newDecoder();
            Charbuffer charBuffer = decoder.decode(buffer);
            System.out.print(charBuffer.toString());
            continue;
        }
    }
}
```
Chapter 10: Making BlazeDS Applications Scalable

In the code snippet above you will notice that the server waits in an infinite loop for requests to come. As requests arrive a selector intercepts them and attaches a key before passing it on to the server. In the server, the waiting request listener iterates through the set of keys. For each key, it checks for the `isAcceptable` and `isReadable` states. If `isAcceptable` is true then a client requires a connection, which is established. If `isReadable` is true then a server is ready to read. In the simple example above once the connection is established and a server is ready to read, bytes are read and the output is sent to console. You can modify existing BlazeDS channels to include Java NIO style socket servers for connection handling while keeping the functionality as before.

After a sneak peek into how asynchronous nonblocking IO could make connections more robust and scalable, let’s move on to the architectural issue of service orientation.

Service Orientation

Service-Oriented Architecture (SOA) could very well be the most popular buzzword of the last couple of years. While universal definitions of SOA are still being debated in forums small and large, I will see how it could apply to RIA. In this section, I choose five patterns that belong to the realm of service orientation and illustrate them. These patterns are worthy of consideration as they can positively impact RIA architectures and make them more maintainable and better in terms of desired output.

The five patterns are:

- Concurrent contracts
- Cross-domain utility layer
- Functional decomposition
- Rules centralization
- State repository

Let’s take each in turn and discuss it briefly.

Concurrent Contracts

Multiple contracts should be created for a single service, each targeted at a particular type of consumer. It is highly likely the enterprise data and systems will be shared by consumption points besides the RIA and so it makes sense to create multiple contracts.

Sometimes it makes sense to create a service façade to act as the front end for the requests to the service. The façade can then choose the correct contract to service the client.

Further to different use cases, multiple contracts and endpoints can also simultaneously support consumption using multiple protocols and data formats. So, one could define an `HTTPService`, a `WebService`, a `RemoteObject`, or a JSON-based endpoint for a single service. That way the same service can allow multiple types of consumers, even if they were all Flex applications.
Cross-Domain Utility Layer

From a consumption standpoint, it’s best if an RIA connects to a service layer that dis-intermediates two or more domains, thereby providing a common service layer and shielding service consumption points from dealing with multiple domain inventories at the same time.

A cross-domain utility layer on the service end simplifies access for disparate enterprise systems. This becomes increasingly relevant when an RIA spans divisions or functional verticals. A good example is a corporate performance dashboard.

Functional Decomposition

A large enterprise-grade RIA both on the client and server end needs to be decomposed into smaller functional bundles. Using modular server-side architectures and dynamically loaded modules on the client is advisable.

Functional decomposition has a positive side effect of making your application modular and easier to extend and maintain.

Rules Centralization

Sometimes it makes sense to keep business rules in a central repository from which the different functional modules can access and use them. The central rules repository applies well to server-centric logic that does the heavy lifting of implementing business rules, process constraints, and application-wide validations.

Using a rules engine in association with a centralized rules repository makes an enterprise-grade Flex and BlazeDS application easily extensible. It also makes them more maintainable.

State Repository

Although keeping all state in one place would classify as an anti-pattern in a RIA, storing state in a central repository for collaborative applications may actually be an advantage.

In applications that share data state across multiple clients, keeping the state at a server-side repository makes it possible for all or multiple clients to listen to changes.

This section only lists a few service orientation–related recommendations quickly. The next section is on caching.

Caching

If the accessed data is not changing during the course of its reuse it always makes sense to cache it. Caching is a time-tested way of increasing performance by avoiding data fetches across the network and using pre-fetched local data instead.
Chapter 10: Making BlazeDS Applications Scalable

Caching emerged in the world of databases and operating system file access and became a popular technique in web applications. Most browsers today cache content that promises to be idempotent and rarely changing. Therefore, a web site’s static content and images are often fetched from cache after the first time, when they are loaded. Browsers also define a cache expiration policy to make sure content does not reside for periods beyond which its usefulness is questionable, becoming stale.

A Flex application runs in a browser and delegates all its web user agent responsibilities to the browser. Therefore, all HTTP requests that involve the GET method are, by default, cached on the user’s disk. Only in situations where the browser cache is turned off or the HTML wrapper page explicitly defines <META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-CACHE"> head tag is the caching not available.

Sometimes, the caching facility also has a notorious side effect, when calls to remote destinations for data involve GET method calls. Subsequent calls to the data fetch remote URI using the GET method are served from the cache and a trip to the remote destination is skipped. This is sometimes a problem as the application data may be stale. A small workaround like appending a random variable to the URL string can make the browser believe that the request is new and different from the earlier one. Alternatively, using the POST method avoids the problem altogether as POST methods are not idempotent and are not cached by the browsers.

Apart from the browser cache, you also have the Flash Player cache that could be leveraged to enhance the performance of Flex applications. Flex applications download an application .swf before they play it. If the Flex application .swf is large, there could be a substantial wait time during download. The upfront high download time is a deterrent to superior user interactivity and responsiveness.

To alleviate the pains of big upfront download, Adobe introduced the concept of Runtime Shared Libraries (RSL) for Flex applications. RSLs use the concept of dynamically linking referenced libraries at runtime.

To explain this further, let’s take a hypothetical example. Say that you have two applications, A.swf and B.swf, both of which are served from the same domain. You realize that both these .swf files share a fair bit of code. So, you abstract out the common elements into a self-contained library, say common.swf. This prompts you to refactor A.swf and B.swf into much smaller compiled binaries which you now perhaps call A1.swf and B1.swf.

When you access A1.swf, both A1.swf and common.swf are downloaded, as both these pieces are required to run the application, which was earlier bundled as A.swf. However, if you subsequently access B1.swf only B1.swf is downloaded. The shared library file, common.swf, is not downloaded, as it’s already available in the browser’s memory cache.

By using a shared library that is downloaded at runtime, you can save on download size and time when multiple subsequent applications from the same domain use the same shared library. This enhances performance and interactivity.

Next, consider another application, which I now call C1.swf. Say that C1.swf is served from a domain different from the one from where A1.swf and B1.swf are served. Also assume that C1.swf can leverage the same shared library common.swf. The question then to ask is: If A1.swf and common.swf are already downloaded and then C1.swf is downloaded, would it be able to reuse the already locally available common.swf? The answer to this important question is “no”. A1.swf and C1.swf will not be able to share common.swf, as they are served from different domains. They can share an RSL only if it’s served from the same domain.
However, starting with Flash Player 9,0,0,115 (Flash Player 9 Update 3), there is a way $A1.swf$ and $C1.swf$, even though served from different domains, could leverage a runtime shared library.

The Flex Framework RSLs now can come in a special flavor called signed RSLs. RSLs signed by Adobe are cached in the Flash Player, which can be shared by Flex applications served from different domains and accessed by different browsers that share the same Flash Player cache. Such RSLs signed by Adobe end with a .swz extension.

**Signed Framework RSL**

At this stage, I will create a sample application and link it to the signed Flex "framework.swz" file to illustrate how one could include and leverage Adobe signed RSLs.

To use RSLs, signed or unsigned, you need three pieces of information:

- Path to the library .swc file that will be delivered later as RSL. Flex needs the .swc at compile time. The RSL “digests” are contained in the .swc file, which are used for validation when the RSL is available at runtime.
- URL of the RSL.
- URL of the security policy file (crossdomain.xml) for the domain from where the RSL will be served.

In addition, you may need a pair of failover URLs for the RSL and a URL to access the cross-domain policy file for the domain that will serve the failover RSL. Failover RSLs are useful when there are problems connecting to the primary server and when Flash Players do not support the version of RSLs available at the primary URL.

Next let’s open up Flash Builder and create a new Flex project that will leverage RSLs, in particular signed RSLs. You are familiar with the process of a Flex project creation in Flash Builder so I will skip all those details here.

The Flex application, called SignedRSLSampleApplication, is elementary and all the code is as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
  <mx:Label text="Signed RSL Sample Application"/>
</mx:Application>
```

Now, go to the project Flex Builder Path view accessible through the project Properties menu option. Once the screen comes up, choose the Library Path tab. When you are at the Library Path screen, you will see a screen like the one shown in Figure 10-2.

To use the framework RSL all you have to do is switch the Framework Linkage combo box selection from Merged Into Code to Runtime Shared Library (RSL). See Figure 10-3 to view this selection.

Now once you recompile, you will see a drop of over a 100k in file size for this low footprint .swf. This is because the framework .swc, which contains the Flex framework essential components, is linked to the application as an RSL. This was possible because flex-config.xml, which resides in the \sdk\3.2.0\frameworks directory contains the following entries by default:
<runtime-shared-library-path>
  <path-element>libs/framework.swc</path-element>
  <rsl-url>framework_3.2.0.3958.swz</rsl-url>
  <policy-file-url></policy-file-url>
  <rsl-url>framework_3.2.0.3958.swf</rsl-url>
  <policy-file-url></policy-file-url>
</runtime-shared-library-path>

Figure 10-2

Figure 10-3
Chapter 10: Making BlazeDS Applications Scalable

To conclude, RSLs reduce the application’s size and thereby work more efficiently with signed RSLs, which once downloaded from any domain remain in the local Flash Player cache and can be reused. At this time signed RSLs are only available for the Flex framework — related RSLs. User RSLs are unsigned.

I used the Flash Builder to create the signed RSL but you could also use the command line mxmlc compiler and pass in these options when using it. Alternatively, you could also leverage the Flex ANT tasks for creating applications that leverage signed RSLs.

Next, resource pooling is illustrated. Resource pooling provides further performance gains on top of caching.

Resource Pooling

BlazeDS interacts with and interfaces with message queues and topics, databases and Java enterprise infrastructure. Many of these external systems and libraries, such as messaging infrastructure, database connections and stateless business services, lend themselves to pooling. When resources are pooled, they are shared over multiple clients.

Usually the creation and initialization, as well as the cleanup and garbage collection, of a resource is expensive. If such a resource is reused often and each time it is used the interaction time is not substantially long, then it is a good candidate for pooling.

In typical enterprise-grade Flex and BlazeDS applications, interaction with a database is an important and necessary part of the application. If such applications utilize near real-time updates, then JMS and messaging infrastructure also become very relevant.

In this section, I discuss JDBC connection and JMS resource pooling to illustrate the use of resource pooling to enhance Flex and BlazeDS application performance. Much of the discussion may not even refer to BlazeDS directly. However, such is typically the case with resource pooling, where much of the activity happens quietly under the hood, without impacting the application code at all.

I start with JDBC connection pooling.

JDBC Connection Pooling

It is always expensive to establish database connections. The process of object creation, transfer of messages across the network for authentication and login, and then initialization on both the database and the application ends is typically intensive. Therefore, it makes sense to create a JDBC connection pool on startup and then allocate connections from this pool. When work using a connection allocated from the pool is done, the connection is returned to the pool.

BlazeDS is a Java Servlets web application and so is deployed in an application server that includes a Servlet container. Popular open source choices for deploying BlazeDS are Apache Tomcat and JBoss. Both these application servers, and almost every other available in the market today, support JDBC connection pooling by default.

I will explore JDBC connection pooling in both Apache Tomcat and JBoss. Let’s start with Tomcat. In Tomcat, Apache Commons DBCP is used by default for JDBC connection pooling. You can read more about the Apache Commons DBCP connection pool at http://commons.apache.org/dbcp/. To use
Chapter 10: Making BlazeDS Applications Scalable

DBCP, you need to configure it with a JNDI DataSource. In Tomcat, you configure this in the server.xml configuration file that resides in the conf directory of your Tomcat installation. Add the following lines to server.xml:

```xml
<Context path="/dbcp" docBase="dbcp" debug="5"
replacable="true" crossContext="true">

<Resource name="jdbc/YetAnotherDB" auth="Container"
  type="javax.sql.DataSource" removeAbandoned="true"
  removeAbandonedTimeout="30" maxActive="100"
  maxIdle="30" maxWait="10000" username="<provide the user name>"
  password="<provide the password>"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost/yadb"/>
</Context>
```

In addition, add the following lines to web.xml so that the JNDI can resolve this reference:

```xml
<resource-ref>
  <description> DB Connection Pooling</description>
  <res-ref-name>jdbc/YetAnotherDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

The preceding configuration for the DataSource in server.xml assumes the underlying database is MySQL. If that’s not the case, then please replace the driverClassName and URI with the appropriate values for your target database. Also, don’t forget to add the particular database’s JDBC driver-related classes in Tomcat’s classpath. Once the DataSource is configured in server.xml, you can create a connection using this DataSource as the connection factory. The code for getting hold of a JDBC connection then could involve referencing the DataSource as follows:

```java
Context envCtx = (Context) new InitialContext().lookup("java:comp/env");
dataSource = (DataSource) envCtx.lookup("jdbc/YetAnotherDB");
```

Once you have obtained a reference to the DataSource, you can call the getConnection method of the DataSource to get a connection to the database. A connection obtained in this manner is a pooled connection. Therefore, the first time such a connection is created, but the next time it’s obtained from the pool.

You probably want to optimize this further by instantiating the DataSource handle only once on startup for a web application. In a Java web application the ServletContext has application wide visibility, and instantiating something within a ServletContextListener could make it to run only once on application initialization. Therefore, you could create a class as follows:

```java
Package somePackage.someSubPackage;
public class JDBCPoolingListener implements ServletContextListener{
public void contextInitialized (ServletContextEvent sce){
    try {
```

288
// Obtain our environment naming context
Context envCtx = (Context) new InitialContext().lookup("java:comp/env");

// Look up our data source
DataSource ds = (DataSource) envCtx.lookup("jdbc/YetAnotherDB");

cce.getServletContext().setAttribute("JDBCPool", ds);
}
}

To use this listener, you can configure it in web.xml as follows:

<listener>
  <listener-class>somePackage.someSubPackage.JDBCPoolingListener</listener-class>
</listener>

That is all it takes to leverage JDBC connection pooling in Tomcat. Next, let's briefly explore how a pooled connection is configured and setup in JBoss.

A DataSource can be created in JBoss by creating a configuration file with a "-ds.xml" naming convention. For MySQL you create a file called: mysql-ds.xml. Deploy mysql-ds.xml in JBoss by copying it over to the /server/default/deploy folder of the JBoss installation. The contents of mysql-ds.xml could be as follows:

<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/YetAnotherDB</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/yadb</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>provide the user name</user-name>
    <password><provide the password></password>
  </local-tx-datasource>
</datasources>

I will skip the details on how this pooled connection can be used in an application that is deployed in JBoss. By now, though, you must be convinced that database connection pooling is effortless and could help make your application more scalable.

The next topic briefly explores pooling of JMS resources.

**JMS Resource Pooling**

JMS promotes the principle of instantiation of a number of components on startup and reuse of these components subsequently. Such JMS components are:
Chapter 10: Making BlazeDS Applications Scalable

- Connection
- Session
- MessageProducer
- MessageConsumer

As with database connections, creating and destroying JMS resources have overhead. Therefore, pooling JMS resources is advisable for optimal resource utilization and performance boost.

Next, you will briefly learn two techniques that pool and cache JMS resources and increase message processing performance. These involve:

- The use of Spring framework’s ([http://springsource.com/](http://springsource.com/)) AbstractMessageListener-Container and its subclasses that provide an asynchronous message consumer that caches JMS consumers, connections, and sessions

Although the following discussion may apply to many JMS providers, they are especially useful in the context of the open source JMS provider ActiveMQ. ActiveMQ can be used with almost any Java application server, including Tomcat, JBoss, and Jetty.

Before we go to the two recommendations I listed, let me start by suggesting the use of Java Connector Architecture (or JCA) on the server to wire up JMS resources. JCA is a Java EE standard for integrating enterprise information systems with Java application servers. You can read more about it online at [http://java.sun.com/j2ee/connector/overview.html](http://java.sun.com/j2ee/connector/overview.html).

JCA supports the pooling of JMS connections, session, and message listeners. It introduces efficiencies by supporting thread pooling and parallel message processing. JCA works with any JMS provider that has a JCA Resource Adapter. ActiveMQ JMS infrastructure is typically configured in JBoss as a JCA Resource.

Next, let’s explore the two recommendations that increase JMS message processing performance.

**Bind Messaging to the Beans Using Camel**

This technique recommends binding the messages to your beans instead of directly using the JMS API.

Apache Camel is an open source integration framework that lets you implement routing and mediation rules in either of the following:

- A Java-based domain-specific language (DSL)
- Scala DSL
- Spring framework XML configuration

The Camel project started in 2007 with the goal of implementing all the integration patterns described in the book *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf (Addison Wesley, 2003). The Camel project is part of the Apache ActiveMQ project.
Camel defines a bean binding mechanism in which it specifies which methods are invoked on a particular message and how the message is translated into the parameters for the method when it is invoked. You can avoid the JMS API and directly leverage bean binding to consume your messages.

Explaining Camel in detail is beyond the scope of this book, but I encourage you to explore it further to understand how you could utilize the enterprise integration patterns in your applications to drive efficiency in your messaging systems.

**Using Spring’s Support for Message Listener Containers**

If you are totally unfamiliar with the open source Spring framework, this section may sound alien. So possibly reading Chapter 5, which covers BlazeDS and Spring integration, may be a better place to start. Although Chapter 5 is not an introduction to the Spring framework itself, fundamentals of that technology are covered there.

In the world of EJBs, Message Driven Beans (MDBs) serve the purpose of message listeners. So when JMS and EJBs are used together, it makes sense to leverage the MDBs as message consumers. Spring extended this message listener concept to POJOs, introducing what they called a Message Driven POJO (or MDP for short).

A message listener container is used to receive messages from a JMS message queue. Incoming messages drive the message listener as they arrive. The message listener container serves as an intermediary between a message producer and an MDP. The container also provides additional services such as registration, transaction, resource acquisition, and release and exception handling. The Spring framework comes with three message listener containers. These three are:

- SimpleMessageListenerContainer
- DefaultMessageListenerContainer
- ServerSessionMessageListenerContainer

The DefaultMessageListenerContainer is the most popular of the three. It allows for the creation of a few JMS sessions on startup and then adapts dynamically at runtime. It can work with external transaction managers. JMS resources like sessions and message consumers are cached and pooled.

If your technology stack includes the Spring framework and you need messaging in your application as well, then you may be better off leveraging the Spring message listener containers to make your applications superior in performance to its counterparts that deal with the raw JMS API.

This section emphasized resource pooling. The next one focuses on optimal utilization of resources on both sides of the wire: the client and the server.

**Workload Distribution**

Distributing work optimally between a client and its server is an important challenge when architecting RIA. In traditional web applications, this decision was simple, as servers did most of the heavy lifting and clients were thin. With RIA things are no more as straightforward. Rich clients allow you to do a lot more than thin clients by maintaining local state and providing a rich interactive infrastructure. Greater interactivity often implies fewer trips back and forth between the client and the server.
Besides, you need to consider the fact that client machines today are more powerful than their counterparts a few years back. Current generation client machines have substantially larger CPU and memory resources at their disposal.

What then constitutes the right workload balance between the client and the server? While there is no universally correct answer to this question, this section attempts to lay out the criteria that govern and affect the decision of distributing work optimally between a rich client and its server.

Simply put, creating thin clients should not be preferred. Flex applications are capable of maintaining state locally and carrying out client-side interaction logic and responding to user gestures, especially by dispatching appropriate events.

Servers for Flex applications should serve the purpose of connecting to enterprise server-side resources, databases and other middle-tier workflow and rules engines. RESTful interfaces are preferred for loose coupling and scalability.

Summary

This chapter on making Flex and BlazeDS applications scalable is a collection of varied recommendations and viewpoints. The chapter started with the important topic of clustering and explained how multiple logical BlazeDS instances aggregated logically can help service a larger number of Flex clients concurrently.

The chapter then delved into efficient tricks for managing data. Gzip-based data compression and binary object data formats were recommended over uncompressed textual data.

The discussion then moved to the topic of Java NIO-based asynchronous nonblocking sockets. Java NIO, with its selector-based intermediary, makes socket connections scalable without the need for extensive multi-threaded program manipulation.

After robust connection and nonblocking IO, the tenets of service orientation that apply to RIA were elaborated and recommendations were discussed at a conceptual level.

Caching and resource pooling are important methods of increasing scalability and availability. These methods exist in every area of system development from operating systems to application software. The section on caching briefly demonstrated how signed RSLs can be effective. The section on resource pooling explained database connection and JMS resource pooling.

The final section ended with a quick discussion of workload distribution, which reiterated that creating thicker clients should be preferred over creating thin clients as in traditional web applications.
In a number of applications, the primary objective of the application is to allow a user to perform create, read, update, and delete (CRUD) operations on the underlying data. The underlying data itself could be stored in some persistent store such as a relational database.

The process of creating two different CRUD applications is quite similar and involves similar types of complexity. Therefore it has been a quest among developers to automate the task of creating such applications.

CRUD applications often involve a list view of the collection of items and a form view of the details of each individual item in this collection. In current times, value additions on top of these simple CRUD views and operations has led to the emergence of the following:

- Management of the collections, where modifications are synchronized, in the list view as multiple users access and manipulate data
- Auto-population of model objects
- Introspection of available services and wiring up of service operations as CRUD methods
- Analytical processing of data, including structuring of data in hierarchical structures
- Charting and graphing of data for useful visual renderings

In this chapter, I will start with basic CRUD and demonstrate how such an application can be generated. Then I will illustrate a few of the advanced features that go beyond simple CRUD.
CRUD with Flash Builder 4

Flash Builder (earlier called Flex Builder) is the Adobe-supported Integrated development environment (IDE) for development of Flex and AS3 applications.

Flash Builder, from version 3 on, includes a CRUD code generation utility that generates the list and item detail views as a database table. This generated application works for Java, PHP, and ColdFusion server-side environments. That is, three different server-side components can be generated to support the CRUD operations.

The latest version of Flash Builder is Flash Builder 4, which as I write this book is in public beta. This new version of the IDE enhances the CRUD generation capabilities substantially from its earlier version. In this section, you will get a chance to see the various available features of this new and improved IDE. You can download the latest copy of Flash Builder from http://labs.adobe.com/technologies/flashbuilder4/. Flash Builder is a commercial product. You can apply your existing Flex Builder 3 licenses to the beta release or use the trial version, which is valid for 60 days.

In the first example, I will demonstrate CRUD generation for simple HTTP service endpoints.

CRUD with RESTful Endpoints

In this section, I will create a simple Flex application and wire it up to consume data from HTTPService endpoints. Such endpoints could typically adhere to the REST architectural style, which has gained in popularity over the past few years.

REST and associated concepts are explained in Chapter 1, in a section titled: “Leveraging Services to Access External Data.” In case you are unsure of REST concepts, you could benefit from rereading parts of Chapter 1 that pertain to it.

I used a book catalog example earlier in this book to illustrate Spring and BlazeDS integration. I am going to reuse that example, with minor modifications, to illustrate the CRUD generation in this chapter.

For starters, let me describe what the book catalog application does. A book catalog maintains a list of books. For each book, it saves the following pieces of information:

- **ID** — A unique identifier for the book. One could have used ISBN numbers to uniquely identify books. However, ISBNs are of two types: one has 10 digits, while the other has 13. From January 1, 2007 every book is supposed to have a 13-digit ISBN and so it could be considered as the unique ID. However, for simplicity sake I consider an internal numeric identifier.
- **Title**
- **Author** — I assume that there is only one author per book, in favor of reducing the domain complexity.
- **Price** — All in a single currency.
To manipulate the book catalog records, that is, books, I create a service abstraction which has the following methods:

- `findAll` — Returns a list of books
- `findById(int id)` — Returns a particular book that matches the unique ID
- `create(Book item)` — Creates a new book
- `update(Book item)` — Updates a particular book record
- `remove(Book item)` — Deletes a particular book record

Once I have the service class ready, I create a URL mapping to service method invocation. Mapping from URLs to service methods involves inclusion of a front controller that intercepts all requests that match a particular URL pattern. Once the request is intercepted at the front controller (fine-grained patterns, for example, reference specific method names or references), this leads to the invocation of specific service methods.

I create a URL mapping to service method calls using REST-style abstractions. So, the CRUD operations listed previously are mapped to URLs as follows:

- `findAll` — `http://<rootURL>/books`
- `findById` — `http://<rootURL>/books/<id>`
- `create` — `http://<rootURL>/books/<id>?_method=CREATE`
- `update` — `http://<rootURL>/books/<id>?_method=UPDATE`
- `remove` — `http://<rootURL>/books/<id>?_method=DELETE`

The first two find calls are **HTTP GET** method calls and the three create, update, and delete calls are **HTTP POST** method calls. Purists may argue that adhering to REST, **HTTP PUT** and **DELETE** should be leveraged for create and delete methods. However, often because of resource security concerns the **HTTP PUT** and **DELETE** methods are not supported in production environments. In any case, these URLs are one possible representation and certainly alternatives are possible.

Once the URLs are identified, I am ready to generate a CRUD application using Flash Builder.

First, create a new Flex project. I will create a new project called **HTTPServiceCRUD**. When I create the Flex project I choose the Server Technology as None/Other, as I will rely on **HTTPService** to access data and save the modified data back into the store. You can route **HTTPService** calls through a BlazeDS server. This comes in handy when the destination does not define a `crossdomain.xml` security policy. In any case, I rely on an external directly accessed **HTTPService** in this example.

Once a project is created I choose Connect To HTTP from the Data menu. Figure 11-1 shows the screen where I make this selection.

This choice brings me to the Configure HTTP Service screen, which before providing any data is as shown in Figure 11-2. Once I provide all the information about the operation names and the corresponding
Chapter 11: CRUD Applications and More

URLs, the screen is as shown in Figure 11-3. For each POST HTTP method–based operation you can also define parameters and their data types.

![Figure 11-1](image1)

Once you click the Finish button, a screen with suggestions to specify the return type for the service operation and to bind the service to UI components appears. The screen is shown in Figure 11-4.

![Figure 11-2](image2)

At this stage, you will benefit from analyzing the generated code. If you browse to the services.BookService class, you will notice that the class extends _Super_BookService. The BookService class is meant to be the extension point where you can define additional behavior. The superclass of this service class is where all the generated code resides.
In the constructor of `_Super_BookService`, operations for each of the five mapped methods are created. The parameters and HTTP method type is set for each operation. All defined operations are added to an Array called “operations”. For illustration, the operation creation and initialization code for the five methods in the `_Super_BookService` constructor is as follows:

```javascript
_serviceControl = new HTTPMultiService();
var operations:Array = new Array();
var operation:Operation;
var argsArray:Array;

operation = new Operation(null, "findAll");
operation.url = "http://www.shashanktiwari.com/books";
operation.method = "GET";
operation.resultType = Object;
operations.push(operation);

operation = new Operation(null, "findById");
operation.url = "http://www.shashanktiwari.com/books/1234";
```

Figure 11-3

Figure 11-4
operation.method = "GET";
operation.resultType = Object;
operations.push(operation);

operation = new Operation(null, "create");
operation.url = "http://www.shashanktiwari.com/books/1234";
operation.method = "POST";
operation.contentType = "application/xml";
operation.resultType = Object;
operations.push(operation);

operation = new Operation(null, "update");
operation.url = "http://www.shashanktiwari.com/books/1234";
operation.method = "POST";
operation.contentType = "application/xml";
operation.resultType = Object;
operations.push(operation);

operation = new Operation(null, "remove");
operation.url = "http://www.shashanktiwari.com/books/1234";
operation.method = "POST";
argsArray = new Array("id");
operation.argumentNames = argsArray;
operation.contentType = "application/xml";
operation.resultType = Object;
operations.push(operation);

_serviceControl.operationList = operations;

Next, let’s peek into one of the service method implementations. The findAll method implementation is as follows:

public function findAll() : AsyncToken
{
    var _internal_operation:AbstractOperation =
        _serviceControl.getOperation("findAll");
    var _internal_token:AsyncToken =
        _internal_operation.send() ;

    return _internal_token;
}

Now that the service is created you could choose any of the following Data menu items to generate the views:

- **Bind to data** — You could bind a DataGrid, List, TextField, or any other valid component to bind it to the data that the service retrieves.
- **Generate Form** — Bind the details editable view to a form.
- **Generate Details Form** — Bind the details editable view to a form. You could first bind the list view and then generate the details form for the items in the list.
I generate a simple form to create a new item in the book catalog. In order to generate a form, I choose Generate Form from the Data menu. The first screen is as shown in Figure 11-5. Since no return type has been configured yet, you are given a chance to configure the method return type. On clicking the Configure Return Type button, you will see a screen like the one in Figure 11-6. You can either use an existing AS3 type or create a new custom type to configure the return type for the method. It’s the create method, and I choose to go with Boolean return type. I would like my service to tell me if the create method was able to successfully create an item or not. I make my choice as shown in Figure 11-6.

As I move forward after specifying the return type, I am presented with a screen to define the property control mapping. See Figure 11-7 to understand what it looks like.

At this stage, you may want to view the details screen, which is shown in Figure 11-8. You will notice a form for creating a new book item. You will also notice a check box that binds to the returned Boolean value of the create method call. If a new book item is created successfully a “true” value is returned, which shows up as a selection on the check box. So far the service data and all the views are not wired enough to make things work seamlessly, but service methods for the CRUD calls have been identified.
and a detail screen has been generated. The moment the server-side RESTful service is available and the list views are generated, the complete application will be ready as far as the essential CRUD features go.

![Figure 11-7](image)

Figure 11-7

![Figure 11-8](image)

Figure 11-8

The illustration so far has presented the code generation strategy with CRUD applications. The server side has relied exclusively on `HTTPService`, which is agnostic to the way that a Flex application that consumes the service is actually built. Now we move to code generation for a `SOAP WebService` to reinforce the concept that the code generation feature is both pervasive and comprehensive. Code generation for Web Services based on WSDL was available in Flex Builder 3 as well.

**Generating Client Code for WebService Consumption**


You can choose to go with an alternative web service. Just make sure to point to the appropriate WSDL. Code generation works off the WSDL.

Figure 11-9 shows the screen where the WSDL URI and service name are specified. This screen is brought up when you choose the Connect to WebService option on the Data menu.
On WSDL introspection, the code generator discovers three methods for the CDYNE weather service, namely:

- GetWeatherInformation
- GetCityForecastByZIP
- GetCityWeatherByZIP

Figure 11-10 shows the screen where the methods after introspection are shown. I select all three methods and click the Finish button. Again, client-side service methods are generated and you have the opportunity to bind list view and item details screens to the service methods. In this example, the supported operations were not CRUD operations. However, if your web service supports CRUD operations, then the code generation process will be similar.

I will just give you a peek into the generated constructor of _Super_CDYNE_Weather and leave the rest for you to discover. Here is the code:

```java
_serviceControl = new WebService();
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "ForecastReturn"), ForecastReturn);
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "temp"), Temp);
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "Forecast"), Forecast);
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "WeatherReturn"), WeatherReturn);
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "WeatherDescription"), WeatherDescription);
SchemaTypeRegistry.getInstance().registerClass(new QName("http://ws.cdyne.com/WeatherWS/", "POP"), POP);
```
var operations:Object = new Object();
var operation:Operation;
operation = new Operation(null, "GetWeatherInformation");
operation.resultElementType = WeatherDescription;
operations["GetWeatherInformation"] = operation;

operation = new Operation(null, "GetCityForecastByZIP");
operation.resultType = ForecastReturn;
operations["GetCityForecastByZIP"] = operation;

operation = new Operation(null, "GetCityWeatherByZIP");
operation.resultType = WeatherReturn;
operations["GetCityWeatherByZIP"] = operation;

_serviceControl.operations = operations;
_serviceControl.service = "Weather";
_serviceControl.port = "WeatherSoap";
_serviceControl.loadWSDL();

As you walk through the code you will notice that XML schema for data types is registered as specific AS3 classes. Then web service operations are created and initialized for each of the three functions. Finally, the service, port and WSDL properties of the _serviceControl are set. By the end of the constructor, the service class is instantiated and initialized for use.

After looking at HTTPService andWebService, I turn my attention to remoting service destinations that allow more effective RPC over AMF, as opposed to the HTTPService andWebService options.
Chapter 11: CRUDD Applications and More

**Generating CRUD with Remoting Service Destinations**

Flash Builder 4 brings automated CRUD generation and client-side data management to Flex and Java applications that use BlazeDS. This increases the developer productivity and brings some of the built-in features that only LCDS users have enjoyed until now. You have already seen some of the code generation capabilities, in the context of `HTTPService` and `WebService`, in the last couple of sections.

In order to understand the code generation and the client-side data management features, follow along as I create a simple application that leverages the new features.

First, download the latest nightly build of BlazeDS. The latest code in the BlazeDS subversion trunk is version 4.0, which will continue to evolve till the next release. You can download the nightly builds of SVN trunk from [http://opensource.adobe.com/wiki/display/blazeds/download+blazeds+trunk](http://opensource.adobe.com/wiki/display/blazeds/download+blazeds+trunk).

The download is available as a zip file archive. I expanded the zip archive to get the `blazeds.war` file in the distribution. I deployed this `blazeds.war` file as `blazeds4` in the tomcat installation I have from the BlazeDS turnkey download. Deploying the downloaded `blazeds.war` as `blazeds4` involved the following two steps:

- Creating a folder called `blazeds4` in the Tomcat `webapps` folder
- Extracting the contents of `blazeds.war` into the `blazeds4` folder

You could also deploy the latest version of BlazeDS is any other application server of your choice.

Now that the latest version of BlazeDS is deployed, make a few small configuration changes before starting on coding up the server-side services.

Next, open `blazeds4\WEB-INF\web.xml` in your favorite code editor. You will notice a section on the `RDSDispatcherServlet` that by default is commented out. Go ahead and uncomment the portion that pertains to the `RDSDispatcherServlet`. This Java Servlet enables the remoting service class introspection and code generation that I will illustrate soon. The `RDSDispatcherServlet` configuration is as follows:

```xml
<servlet>
  <servlet-name>RDSDispatchServlet</servlet-name>
  <display-name>RDSDispatchServlet</display-name>
  <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
  <init-param>
    <param-name>useAppserverSecurity</param-name>
    <param-value>false</param-value>
  </init-param>
  <load-on-startup>10</load-on-startup>
</servlet>

<servlet-mapping id="RDS_DISPATCH_MAPPING">
  <servlet-name>RDSDispatchServlet</servlet-name>
  <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
</servlet-mapping>
```
The `useAppserverSecurity` default value is `true`. You may set that to `false` to avoid the application server authentication as a requirement for using the CRUD generation and the client-side data management with BlazeDS.

Next, let's write up the server-side service classes. I create the following classes:

- **Book.java** — A bean type model class that contains the attributes and accessor methods for a book.
- **IBookService.java** — An interface that defines the five CRUD methods, namely `findAll`, `findById`, `create`, `update`, and `remove`.
- **BookService.java** — A service class that implements the `IBookService` interface and provides an implementation of a service that interacts with a book catalog.
- **HsqldbDao.java** — I also create a data access object class to access an HSQLDB database. You can create your own data access object to access an alternative data store.

I will not show the code for all the implementation classes here but only show the `IBookService` interface to provide you a view into the available methods, their parameters, and return types. The `IBookService.java` source is as follows:

```java
package problazeds.ch11;

import java.util.List;

public interface IBookService {
    public List<Book> findAll();
    public Book findById(int id);
    public boolean create(Book item);
    public boolean update(Book item);
    public boolean remove(Book item);
}
```

Once all the Java classes are coded, I compile them and copy over the compiled code to the `blazeds4\WEB-INF\classes` folder within a folder structure that matches the packages hierarchy. In this particular example, all classes are copied over to the `problazeds\ch11` folder within the `WEB-INF\classes` folder.

Next I configure the service class as a remoting service destination. I create the following entry in `remoting-config.xml`, which you know is included in the `services-config.xml` file that the BlazeDS MessageBrokerServlet reads on startup. The remoting service configuration for the `BookService` class is:

```xml
<service id="remoting-service" class="flex.messaging.services.RemotingService">
```

Download from Library of Wow! eBook <www.wowebook.com>
<adapters>
  <adapter-definition id="java-object" class="flex.messaging.services.
remoting.adapters.JavaAdapter" default="true"/>
</adapters>
<default-channels>
  <channel ref="my-amf"/>
</default-channels>
<destination id="bookService">
  <properties>
    <source>problazeds.ch11.BookService</source>
    <scope>application</scope>
  </properties>
  <adapter ref="java-object"/>
</destination>
</service>

This gets the server side ready. Next, I create a Flex application to consume these server-side services.

Using Flash Builder 4, I create a new Flex project. I call it RemotingServiceCRUD. You can choose a name that you like or go with my choice. I choose BlazeDS as the Server Technology. Look at Figure 11-11 to see what the initial screen looks like.

![Figure 11-11](image)

On clicking the Next button, a screen to specify the server location comes up. You need to specify the following information in this screen:

- **Root folder** — The root of the web application where the BlazeDS server side is deployed. In my case, the value is C:\applications\blazeds_b1_020108\tomcat\webapps\blazeds4. You
can browse to the folder where your web application is deployed, which provides a convenient way to choose the location.

- **Root URL** — The URL at which the application is available. In my case the value is `http://localhost:8400/blazeds4`. I am using the BlazeDS turnkey distribution that listens for HTTP requests by default on port 8400. Make sure to specify the URL correctly.

- **Context root** — The context root for the web application, which is the same as the web application name; in my case, this is blazeds4.

After you have specified the information, you can validate the configuration settings. Start up your server before you validate the configuration. Also, turn off any antivirus programs that might not allow connections from an Eclipse instance. On this screen, you also have the option to specify the output folder, which you are most likely to leave as the default value. The screen where you provide the server location information is shown in Figure 11-12.

![Figure 11-12](image)

Clicking Next takes you to the screen that confirms the Flex application name. After confirming the value for the application name and optionally providing an output folder, the Flex project is created.

Once a Flex project is created, select the Connect to BlazeDS option from the Data menu. Figure 11-13 shows how you make this selection.

![Figure 11-13](image)

You will be prompted with an authentication screen asking you to provide the credentials to authenticate with RDS. Select the No password required check box and click OK. You will be taken to the screen that will let you import the BlazeDS services. The authentication screen before and after the No password required selection is shown in Figures 11-14 and 11-15.
The Import BlazeDS/LCDS Service screen, shown in Figure 11-16, is where you choose the services and generate the client-side data management classes.

All the remoting service destinations configured in \texttt{remoting-config.xml} are available to be imported into your Flex application. In my simple example, I have only one service, called \texttt{BookService}, and that is available to be wired in to the Flex user interface. You also have to give your service a name,
which I have chosen as “bookService”. You could choose a different name. Figure 11-17 shows the choices I made.

![Figure 11-17](image)

Once I click the Finish button, a screen comes up suggesting that I configure return types for services and bind service operations to UI components.

The Data/Services view lists all the five CRUD methods. The Data/Services view is shown in Figure 11-18.

![Figure 11-18](image)

At this stage, you may also want to peek into the generated code. Browsing to the `services.bookService` folder in the project source, you will notice the following AS3 classes:

- `BookEntityMetadata`
- `Super_Book`
- `Super_BookService`
- `Book`
- `BookService`
As you saw earlier in the discussion of the `HTTPService` and `WebService`, `Book` and `BookService` extend `_Super_Book` and `_Super_BookService`, respectively. The constructor for `_Super_BookService` adds the five CRUD operations as follows:

```javascript
_serviceControl = new RemoteObject();

var operations:Object = new Object();
var operation:Operation;

operation = new Operation(null, "findAll");
operation.resultElementType = Object;
operations["findAll"] = operation;
operation = new Operation(null, "findById");
operation.resultType = Book;
operations["findById"] = operation;
operation = new Operation(null, "remove");
operation.resultType = Boolean;
operations["remove"] = operation;
operation = new Operation(null, "update");
operation.resultType = Boolean;
operations["update"] = operation;
operation = new Operation(null, "create");
operation.resultType = Boolean;
operations["create"] = operation;
_serviceControl.operations = operations;
_serviceControl.convertResultHandler = TypeUtility.convertResultHandler;
_serviceControl.destination = "bookService";
```

The `Book` attributes are managed through the generated `_BookEntityMetadata` class. Property maps in this class are initialized as follows:

```javascript
if (model_internal::dependentsOnMap == null)
{
    // dependents map
    model_internal::dependentsOnMap = new Object();
    model_internal::dependentsOnMap["id"] = new Array();
    model_internal::dependentsOnMap["author"] = new Array();
    model_internal::dependentsOnMap["title"] = new Array();
    model_internal::dependentsOnMap["price"] = new Array();

    // collection base map
    model_internal::collectionBaseMap = new Object()
}

model_internal::_instance = book;
```

Now, let’s put this generated code to work. I will create a `DataGrid` next and bind the `findAll` method that returns the list of books as the data provider for the grid. I will be able to do all of this with a few clicks and with no hand-coding.
Chapter 11: CRUD Applications and More

Change to the design view and drag a DataGrid from the Data Controls to the main application MXML file. The screen will be as shown in Figure 11-19.

![Image 1](image1.png)

Figure 11-19

Then right-click on the DataGrid and select Bind To Data from the context menu choices. See Figure 11-20.

![Image 2](image2.png)

Figure 11-20

You will be presented with a view to choose the service operations. I choose `findAll` as the operation to get the list of books. See Figure 11-21.

I also have the option to specify a return type. If you choose to specify a return type by clicking the Configure Return Type button, you will be presented with a screen where you can choose an existing AS3 data type or define a custom AS3 data type. I choose an existing `Books[]` as the return type. Once the return type is specified, you will see a screen like the one in Figure 11-22.

Once I make all the selections and confirm, the DataGrid automatically updates with the correct fields and is wired up with the RemoteObject. The automatically generated grid as viewed in the design view is shown in Figure 11-23.
That's it! You have the list view fully generated and ready for use. Similarly, you could generate the detail views as well.

These new code generation and client-side data management features enhance developer productivity and provide a means to create Flex application rapidly. The code generation, although static, allows for versioning and maintenance to keep pace with changing requirements.

Prior to Flash Builder 4, Flex Builder 3 supported the generation of simple CRUD applications by pointing to a database. In addition, Clear Data Builder, a code generator from Clear Toolkit, an open source framework and component library, had a way to generate CRUD applications basis xDoclet annotations of SQL on server side methods for the CRUD operations. Clear Toolkit is available at http://sourceforge.net/projects/cleartoolkit/ . If you are interested in finding out the details on how the Clear Toolkit generates SQL-based CRUD applications, read a document titled “Cooking RIA CRUD with Flex and BlazeDS,” which is available online at http://www.myflex.org/demos/Cooking_CRUD.pdf.

Code generation that does not require SQL, but is rather based on Java data transfer objects (DTOs), is described in http://sourceforge.net/projects/cleartoolkit/files/Clear Toolkit Docs/General Docs/CDB31_release_notes.pdf.
Next, we view a couple of options of auto-generation of AS3 model objects, basis their server side counterparts, and basis an XML Schema.

**Generating AS3 Model Objects**

In an enterprise Flex and Java application, the model objects can rapidly increase from a few in numbers to a substantial amount. When that happens, keeping tab of object counterparts on both sides of the wire gets cumbersome. On top of that, if you are tasked with hand-coding them, then you are in for some serious tedious effort.

In order to make things easy and enhance developer productivity, a few tools have emerged that make it possible to generate AS3 model objects based on the Java-side counterparts and populate the value objects at runtime based on the XML schema that defines their structure.

In this section, I will explain both these scenarios and show how to leverage some of the open source tools and framework libraries to auto-generate and auto-populate AS3 model objects, which are also referred to as value objects (VOs) or Data Transfer Objects (DTOs).

First, I consider auto-generation from Java to AS3 and use the GraniteDS Gas3 utility to achieve this goal.

**Using Gas3**

It’s possible the mention of GraniteDS with BlazeDS is making you suspicious, wondering if it was a covert way of introducing the non-Adobe open source data service into the bundle. Well, if you do have that fear, then worry not, because there is no such intention! GraniteDS includes a number of utilities that can be used and leveraged even if you don’t use GraniteDS as a whole, and that is exactly what I am doing here.

GraniteDS comes with a very useful utility called Gas3 that helps generate AS3 classes on the basis of its Java counterparts. Gas3 can be used as an ANT task or as an Eclipse plug-in, so it is a very handy tool for flexible code generation.

To understand Gas3, let’s first set it up and use it to generate a sample AS3 class. I will use the Eclipse plug-in version first. Subsequently I will explore the Gas3 ANT tasks.

The Gas3 Eclipse plug-in is bundled as a part of the GraniteDS Eclipse plug-in distribution. Therefore, to get Gas3 you need to download the GraniteDS Eclipse plug-in. You can get the GraniteDS Eclipse plug-in from the GraniteDS project site on SourceForge, at http://sourceforge.net/projects/granite/files/. Download the org.granite.builder distribution, whose latest general availability release is org.granite.builder_2.0.0.GA.jar.

Installing the plug-in is simple. Just copy the downloaded plug-in to the plugins directory of your Eclipse installation.

Once installed, restart Eclipse. The GraniteDS plug-in is now available. Create a new Java project. I create a simple class Book.java in it. Next apply the GraniteDS nature to the project simply by right-clicking on
project name and choosing Add GraniteDS Nature from the context menu. See Figure 11-24 to view the context menu choices.

![Figure 11-24](image)

On adding GraniteDS nature, you will be presented with a configuration wizard. The first screen of this wizard, as shown in Figure 11-25, asks you to select the Java source folder.

By clicking the Add Folder button, I get a dialog box that lets me choose the source folder that contains `Book.java`. Figures 11-26 and 11-27 show the dialog box and the screen after the selection is made.

In addition to specifying the source folder you can configure dependent projects and the classpath. See Figures 11-28 and 11-29.
Figure 11-27

Figure 11-28

Figure 11-29
Before the files are actually generated, you are also given a choice to configure the Groovy templates and the file generation options. The template configuration screen is in Figure 11-30. The file generation options screen is in Figure 11-31.
On clicking the Finish button, AS3 code is generated for the chosen Java files. In this case, the Java object is Book.java. The two generated AS3 classes are:

- BookBase.as
- Book.as

Book.as extends BookBase.as. BookBase is updated every time there are modifications to the underlying Java class or if the current generated class is too outdated. All custom behavior and extensions should be written in Book.as, not BookBase.as. Book.as is generated once and then not touched by Gas3.

The output messages displayed in the Eclipse console are shown in Figure 11-32.

![Figure 11-32](image)

Listing 11-1 shows Book.java and Listing 11-2 and 11-3 show BookBase.as and Book.as, respectively.

**Listing 11-1: Book.java**

```java
package problazeds.ch11;

public class Book {
    private int id;
    private String title;
    private String author;
    private double price;

    public int getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }
}
```

Continued
**Listing 11-1: Book.java (continued)**

```java
public void setAuthor(String author) {
    this.author = author;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
    this.price = price;
}
```

**Listing 11-2: BookBase.as**

```javascript
/**
 * Generated by Gas3 v2.0.0 (Granite Data Services).
 *
 * WARNING: DO NOT CHANGE THIS FILE. IT MAY BE OVERWRITTEN EACH TIME YOU USE
 * THE GENERATOR. INSTEAD, EDIT THE INHERITED CLASS (Book.as).
 */

package problazeds.ch11 {

    import flash.utils.IDataInput;
    import flash.utils.IDataOutput;
    import flash.utils.IExternalizable;

    [Bindable]
    public class BookBase implements IExternalizable {

        private var _author:String;
        private var _id:int;
        private var _price:Number;
        private var _title:String;

        public function set author(value:String):void {
            _author = value;
        }
        public function get author():String {
            return _author;
        }

        public function set id(value:int):void {
            _id = value;
        }

    }

}
public function get id():int {
    return _id;
}

public function set price(value:Number):void {
    _price = value;
}
public function get price():Number {
    return _price;
}

public function set title(value:String):void {
    _title = value;
}
public function get title():String {
    return _title;
}

public function readExternal(input:IDataInput):void {
    _author = input.readObject() as String;
    _id = input.readObject() as int;
    _price = function(o:*):Number { return (o is Number ? o as Number : Number.NaN) } (input.readObject());
    _title = input.readObject() as String;
}

public function writeExternal(output:IDataOutput):void {
    output.writeObject(_author);
    output.writeObject(_id);
    output.writeObject(_price);
    output.writeObject(_title);
}
}

Listing 11-3: Book.as

/**
 * Generated by Gas3 v2.0.0 (Granite Data Services).
 *
 * NOTE: this file is only generated if it does not exist. You may safely put
 * your custom code here.
 */

package problazeds.ch11 {

    [Bindable]
    [RemoteClass(alias="problazeds.ch11.Book")]
    public class Book extends BookBase {
    }
}
Gas3 can also be invoked from an ANT build script. The ANT task is available in the main GraniteDS distribution. A simple ANT task that could generate a set of AS3 classes based on Java beans defined in a particular folder is as follows:

```xml
<target name="generate.as3">
  <gas3 outputdir="as3">
    <classpath>
      <pathelement location="classes"/>
    </classpath>
    <fileset dir="classes">
      <include name="path/to/your/Java/class/files/**/*.class"/>
    </fileset>
  </gas3>
</target>
```

`outputdir` is where the generated AS3 classes reside. For each Java class, two AS3 classes are created.

Java classes in a particular package structure can be translated to corresponding AS3 classes that reside in a completely different package structure. Such translators can be defined as follows:

```xml
<translator
  java="/path/to/my.java.class"
  as3="/path/to/my.as3.class"/>
```

Gas3 uses Groovy templates for code generation from Java to AS3. Although the built-in templates are fairly comprehensive, you can write newer templates if you desire. Newly created templates can be attached to a Gas3 instance by defining the fully qualified name of the template as the value of the `entitytemplate` Gas3 node attribute.

The built-in templates in Gas3 are as follows:

- `entitytemplate` and `entitybasetemplate` — Used for EJBs and JPA objects
- `interfacetemplate` and `interfacebbaseTemplate` — Used for Java interfaces
- `generictemplate` and `genericbasetemplate` — For generic Java classes
- `enumtemplate` — Template for `java.lang.Enum` type

To provide custom translation and support for custom types, you can provide your own implementation for `org.granite.generator.as3.Java2As3`. `DefaultJava2As3` is the default implementation.

### Auto-Populating AS3 Model Objects with XML Based on an XSD

In the last section, AS3 value objects were generated on the basis of their Java counterparts. In this section, AS3 model objects are populated on the basis of the underlying XML Schema (XSD). XSD-to-AS3 translation comes as a handy tool when using web services that publish their data types as XSD.

To get started, first create a sample XSD. A sample XSD is shown in Listing 11-4.
**Listing 11-4: book.xsd**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema id="schema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.shanky.org/problazeds"
    targetNamespace="http://www.shanky.org"
    elementFormDefault="qualified">

    <xs:complexType name="book">
        <xs:sequence>
            <xs:element name="id" type="xs:int"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="author" type="xs:string"/>
            <xs:element name="price" type="xs:double"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

An XML document in adherence to `book.xsd` is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.shanky.org/problazeds"
    xsi:schemaLocation="http://www.shanky.org book.xsd"
    xsi:noNamespaceSchemaLocation="http://www.shanky.org book.xsd">

    <id>1</id>
    <title>pro blazeds</title>
    <author>shashank</author>
    <price>54.99</price>
</book>
```

A value object for the Book entity is:

```typescript
package problazeds.ch11
{
    [Bindable]
    public class Book
    {
        public var id:int;
        public var title:String;
        public var author:String;
        public var price:Number;

        public function Book()
        {
        }
    }
}
```
Chapter 11: CRUD Applications and More

Now that we have the XSD, a sample XML adhering to the XSD and an AS3 model object, I will illustrate how the AS3 model object can be populated with the data available as XML. The XSD and the value object are used to map the data between the XML and the object structure.

The `mx.rpc.xml.SchemaLoader` class can be used to load an XSD and then map it to an AS3 object type. First, you need to instantiate a `SchemaLoader` and then you need to set up an event listener for the `SchemaLoadEvent.LOAD` event, which is dispatched when a schema is loaded using the `SchemaLoader` instance. Once the listener is added, you actually load an XML schema. When the schema is loaded the `SchemaLoadEvent.LOAD` event is dispatched and the event listener handles this event.

The code for setting up the event listener and calling the `load` method is:

```java
schemaLoader = new SchemaLoader();
schemaLoader.addEventListener(SchemaLoadEvent.LOAD, schemaLoader_loadHandler);
schemaLoader.load("book.xsd");
```

In the event handler, the schema is first accessed via the event object; then it is added to a `SchemaManager` instance. After that, the XSD is mapped to the AS3 model object. The mapping is done with the help of an instance of the `mx.rpc.xml.SchemaTypeRegistry` object, which registers an AS3 class with an XSD, identified by its target namespace URI.

The event handler itself could be a function as follows:

```java
schemaManager.addSchema(event.schema);
var schemaTypeRegistry:SchemaTypeRegistry = SchemaTypeRegistry.getInstance();
schemaTypeRegistry.registerClass(new QName(schema.targetNamespace.uri, "book"), Book);
```

The example so far has suggested event handlers for the `SchemaLoadEvent` event, which is dispatched when a `SchemaLoader` has completed the task of loading the schema. As the schema is loaded, the `SchemaLoader` also dispatches the `XMLLoadEvent.LOAD` event. This means that if an XSD references other XSDs, the `XMLLoadEvent` is dispatched each time a schema is loaded. As opposed to this, the `SchemaLoadEvent` is dispatched only once per load.

Like any other loading functions, a schema load could end up in error, and you will benefit from defining a fault handler to take care of such situations.

Once the schema and the model object are mapped, auto-population of model objects with incoming XML data that complies with the XSD becomes possible. To populate a value object at runtime, you do the following:

- Create an instance of `mx.rpc.xml.XMLDecoder`
- Register the `XMLDecoder` instance with the `SchemaManager` instance that was created for the application
- Instantiate a `QName` with two parameters: the schema target namespace URI and the root of the XML document
- Invoke the `decode` function of the `XMLDecoder` instance passing in the XML and the `QName` instance as parameters
On calling the `decode` method of the `XMLDecoder`, the AS3 model object is returned, which can be accessed by type casting to the particular data type, in our example the Book AS3 class.

You could also carry out the reverse translation of object to XML using an `XMLEncoder` and calling its `encode` method with the object and a `QName` instance as parameters. The `QName` would again map the target namespace URI to the root of the XML document.

The `XMLDecoder` code could be as follows:

```java
QName qName = new QName(schema.targetNamespace.uri, "book");
XMLDecoder xmlDecoder = new XMLDecoder();
xmlDecoder.schemaManager = schemaManager;

decodedObject = xmlDecoder.decode(xml, qName);
book = decodedObject.book as Book;
```

The `mx.rpc.xml` package has many of these useful utilities, many of which are not even listed in the public live docs. It’s worthwhile to explore these and use them for auto-population, which saves time and enhances developer productivity.

**Summary**

The last chapter of the book provides a walkthrough of some of the ways of generating CRUD applications. It also explores a couple of utilities to convert Java objects to their AS3 counterparts and to auto-populate objects from the incoming XML at runtime.

The chapter starts with a discussion on the client-side data management features for `HTTPService` and `WebService`. It illustrates how URL endpoints could be bound to CRUD methods for an `HTTPService`. It also explains how a WSDL could be used to introspect and generate client-side stubs for a web service.

CRUD generation to a limited extent was supported in the earlier versions of Flex Builder. However, the new support is far more flexible, extensible, and easy to use.

After a discussion on `HTTPService` and `WebService` the focus is on remoting service, which is represented by its client-side component: `RemoteObject`. A complete walkthrough of a CRUD application creation provides a comprehensive view of the CRUD and client-side data management features of Flash Builder 4.

After the section on CRUD application generation, two important topics are presented. The first of these explains how the GraniteDS Gas3 utility can be used to generate AS3 counterparts of Java objects. The second explains how an XML document can be auto-populated as a model object. An XSD is used to map an XML and an object structure.

With that, the chapter comes to an end and so does the book. Hope you have enjoyed reading it and find it useful as you embark on your own Flex and Java applications.
Introducing Flex 4

In a little more than a couple of years, through its evolution from its initial versions under Macro-
media, Flex has emerged as a comprehensive framework and component library in its current form,
version 3.3, under Adobe. As the framework has matured, so have the already superior Flash run-
time, in which every Flex application runs, and the integration hooks, which make Flex work with
popular server-side technologies such as Java, PHP, .NET, and Ruby.

Now this popular and successful framework, Flex, is going through a prominent transformation
again, as it’s getting ready for version 4. This concise write-up aims to illustrate these changes and
explain them in a manner that is accessible not only to those who know Flex 3.x but even to those
who are just scratching the surface as far as this topic goes. As I write this write-up on Flex 4, it still
being baked in the oven. So, while I make every attempt to keep this write-up current and complete,
I don’t promise that it will be exhaustive. For those who thrive at the cutting edge, the last statement
may not come as a surprise.

To get started on Flex 4, I begin with an overview of its proposed architecture and the motivations
underlying this new proposal.

What Is Flex 4?

Flex 4 (codenamed ‘‘Gumbo’’), like its earlier versions, is an ActionScript 3 (AS3) framework for
application development. It includes a set of ready-to-use components and layout containers and
provides service components to access external data. For rapid and intuitive development, the
framework includes an XML-based declarative markup language called MXML, which can be used
to write a lot, if not all, of a Flex application. The framework code bundled with a compiler and a
set of utilities is distributed as a software development kit (SDK). Besides, Adobe offers an Eclipse-
based IDE for Flex application development. The IDE is branded as Flex Builder. In this book, I
focus on the SDK and leave the IDE out of discussion.
Appendix A: Introducing Flex 4

Flex applications, on compilation, become byte code that can be understood by the ActionScript Virtual Machine (AVM) of a Flash Player. The byte code files are commonly referred to as SWF (pronounced “swiff”) files. These files derive this name from “Shockwave Flash” and include “.swf” as extension to their filenames. Flex 4 exclusively targets Flash Player 10 and the AVM 2 virtual machine.

Let’s start by exploring what’s new in Flex 4.

What’s New?

In order to understand the new features in Flex 4, you need to know what exists in the current versions of Flex. Chapter 1 introduces Flex 3, and you will benefit from reading it again to refresh your memory about the concepts.

The modifications and additions to Flex are inspired by an underlying set of goals, which as stated by Adobe are:

- Design in mind
- Developer productivity
- Framework evolution

“Design in mind” refers to the goal of making designers and developers work together seamlessly. Often, designers and developers work in disconnected workflows. Designers use their favorite design tools to craft beautiful but static interfaces. Developers use these graphic designs as wireframes and create a usable interface that attempts to look as close as possible to these original designs. Developers have difficulty recreating the artwork, so at times designers are avoided altogether and developers create the interface themselves. The creators of Flex 4 attempt to solve this problem by making it simple for designers and developers to work together effectively.

As a start, the creators have attempted to create a common language and format for the design elements and the program components. That way, designs created by designers can be consumed as is by the developers’ tools. This common language and format needs to be simple and easy to understand and manipulate for it to be widely adopted by designers, developers, and their tool makers. The choice of a declarative syntax seems like a viable idea for this common language.

Flex 4 establishes an XML-based syntax, named FXG, to define graphic elements. Using this declarative syntax, you can draw elementary shapes, such as rectangles and ellipses, and limn out any path using a set of coordinates. You can also define text and bitmap elements within it.

The problem of ineffective collaboration between designers and developers isn’t solved with the creation of FXG alone. At the very least, a few more changes need to be incorporated, which are:

- Allow design tools to export the design elements as FXG
- Provide a tool to convert and manipulate existing graphic files to FXG formats
- Allow graphic elements to coexist with Flex user interface components
- Refactor Flex user interface components to separate presentation and behavior into individual pieces to allow graphics to be applied to the presentation aspects cleanly
Adobe realizes the importance of each of these and is actively working on incorporating the preceding list of requirements into its existing and new tools, frameworks, and libraries. The company has refactored its popular design tools like PhotoShop and Illustrator to export the artwork as FXG. It has created a new product, branded as “Flash Catalyst,” to convert and manipulate graphic files so that they can be used as FXG. In addition, it has modified Flex 4 significantly to work with FXG and the graphic elements. A completely new user interface component model, containment collection, and layout strategy have been introduced to make Flex components, FXG graphic elements, and Flash display objects work together.

The “design in mind” goal has the most significant impact on Flex 4 and many of its new features. The second goal is “developer productivity.”

The current Flex compiler is improved to work faster both for complete and incremental compilations. The Flex team has stated an objective of a fivefold increase in compiler performance. You will also notice improved garbage collection performance, when compiling your 3.x code with the Flex 4 compiler. Other productivity improvements include support for two-way data binding and automation support in AIR. The automation framework allows for recording and playback of Flex and AIR framework-based applications.

The third and last goal is “framework evolution.” Framework evolution simply implies catching up with change and user expectations. The Flash Player is evolving rapidly. For example, Flash Player 10 has a new text layout framework and pixel bender effects. Flex 4 is keeping pace with these changes and incorporating them into the framework, where applicable. Flex 4 is built for Flash Player 10. You may want to explore the high-fidelity text layout framework and the pixel bender effects.

In addition, in its path of evolution, Flex 4 attempts to meet the growing expectations of its users.

The goals and their impact say a lot about the big changes in Flex 4 and, hopefully, set your expectations for the new framework. The trickle down of these changes is not covered exhaustively in this write-up.

Reading about change can be unsettling and promising at the same time. While change here means better features, the Flex developer has obvious worries about the fate of his current applications and skill sets. Next, you will see how Flex 4 and its predecessors can exist together.

**Flex 4 and Before**

Flex 3.x and Flex 2 UI components and the related artifacts, such as effects and transitions, are now referred to as “Halo.” Flex 4, as mentioned before, is codenamed “Gumbo.”

For reasons of backward compatibility and incremental adoption, it’s necessary that Halo and Gumbo components work together. Therefore, the framework creators aim to make the two models interoperable. Halo components should work in Gumbo containers. You want to use your cool custom components and existing application code in the future. Gumbo components should work in Halo containers. That way you can start using Gumbo components as they come fresh out of the oven, without waiting for the entire set to bake or be packaged.

There are two potential problems in using the two sets of components together, namely:

- Halo and Gumbo have classes with the same name and share a number of components, leading to namespace clashes.
The existence of classes from different versions of Flex in a single application leads to class-loading issues within an application domain.

Both these problems were anticipated from the beginning, so provisions have been made to mitigate and avoid them. For avoiding namespace collisions, multiple namespaces are available that keep a clear separation between the different language versions. For version clashes within an application domain, a “Marshall Plan” has been drafted.

The advent of Gumbo has inspired the creation of four sets of MXML namespaces, two for language tags and two for the component tags. The two language tags are:

- **MXML 2006** — With http://www.adobe.com/2006/mxml as the URI (Uniform Resource Identifier) and mx as the default prefix
- **MXML 2009** — With http://ns.adobe.com/mxml/2009 as the URI and fx as the default prefix

The two component tags are:

- **Spark** — With library://ns.adobe.com/flex/spark as the URI and s as the default prefix
- **Halo** — With library://ns.adobe.com/flex/halo as the URI and mx as the default prefix

MXML 2006 is the namespace currently used in Flex 3. It contains the following:

- MXML language tags
- AS3 built-in types
- Flex 3 (Halo) component tags
- Remote procedure call (RPC) tags
- Data services
- Charts

The MXML 2006 language tags are:

- Binding
- Component
- Metadata
- Model
- Repeater
- Script
- Style

In Flex 3, the MXML 2006 namespace URI and default prefix are assigned using xmlns:mx="http://www.adobe.com/2006/mxml" as an attribute of the application root tag. A language tag like Binding is then accessed as <mx:Binding>.
Appendix A: Introducing Flex 4

The AS3 built-in data types are included in this name space. Following are the list of AS3 built-in types that have corresponding tags:

- Array
- Boolean
- Class
- Date
- Function
- int
- Number
- Object
- String
- uint
- XML
- XMLList

The UI components in Flex 3, like the Button and ComboBox controls and the Box and Panel containers, also form part of the legacy MXML 2006 namespace. In addition, RPC components like HTTPService and WebService, components related to the LifeCycle Data Service on the client side, and the components from the charting package, which is not part of the Flex SDK, are also included. Therefore, the current namespace is one monolithic structure with little scope for using specific parts separately.

Flex 4 defines a new version of MXML: MXML 2009. MXML 2009 includes the language tags that support Gumbo. Gumbo language tags reuse most of Flex 3 language tags and introduce only a small set of new tags to work with the newer features. The new language tags in Flex 4 (Halo) are:

- Declarations
- Definition
- DesignLayer
- Library
- Private
- Reparent

MXML 2009 contains only the language tags that determine how the MXML tags are parsed. However, it does not contain any component or service tags.

XML namespace for components, both old (Halo) and new (Spark), are defined to bring in the old and new components in MXML 2009. It also allows the use of Spark components with MXML 2006. Just remember that two language tags cannot be used simultaneously.

At this time, it is possible you are completely baffled by the sudden emergence of a new term: “Spark.” So, clarification may be of help. Gumbo is the codename for the Flex 4 open source SDK project. It
Appendix A: Introducing Flex 4

includes a few free Adobe tools that may not necessarily be open source. It does not include the Flex Builder IDE, although Flex Builder Gumbo is used at times to connote Flex Builder 4. The new set of components introduced by Gumbo, often referred to as Gumbo components, or even gumbonents, has been christened as “Spark.” You will see “Spark” appearing in the namespace and the package names.

Figure A-1

Figure A-1 summarizes how Halo and Spark can be used with MXML 2006 and MXML 2009. From the figure, it should be clear that mixing and matching is possible. So two buttons, one each of type Flex 3 and Flex 4, can be used together in a Flex 4 application as follows:

```xml
<Application xmlns="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <mx:Button />
    <s:Button />

</Application>
```

Namespace collision is nicely managed by the presence of multiple namespaces; the two classes of the same name are just put within different package structures.

It’s popular to build large Flex applications in a modular fashion. Modular applications promote reuse and allow for aggregation of applications, as in a portal. As large applications evolve, it’s likely that you will have modules and subapplications written in different versions of Flex. Using the Flex module system to load these subapplications will not work, as it tends to load conflicting classes only once and that might end up producing an error. Different versions of Flex will have classes with the same fully qualified name but with varying functionality, and it’s important to load them into distinct partitions.
Appendix A: Introducing Flex 4

As mentioned earlier, Adobe recognized this problem and came up with a solution in what is called a “Marshall Plan” to allow subapplications written in different versions of Flex to be used together effortlessly. The only constraint they imposed was to limit this compatibility to as far back as version 3.2. This means that modules written in Flex 2 or Flex 3 prior to version 3.2, won’t work under the Marshall Plan.

The Marshall Plan puts each subapplication, a SWF, in a separate ApplicationDomain and makes them interact with each other through event passing. Grouping subapplications into ApplicationDomains and SecurityDomains allows appropriate sharing and interaction between both trusted and untrusted subapplications. Under this new plan, a few things like the history manager and the shared services don’t seem to work, but it’s expected that the Marshall Plan will be revisited and revised as developers start leveraging it to build applications that involve multiple versions of Flex.

Having understood the initial definition and backward-compatibility requirements, it is time now to see the new framework in action.

Leveraging the New Framework

In this section, let’s attempt to use Flex 4. In the first subpart I will talk about the new development process in Flex 4, and in the second subpart you will write your first simple Flex 4 application.

Development Workflows

One of the biggest changes in Flex 4 is the refactoring of the component model and the inclusion of design elements as first class citizens. A detailed discussion of the new component model is not provided here.

Flex 4 cleanly and clearly separates the presentation and behavior elements of a user interface component. A component’s core features, the events it triggers, the data it holds, and the gestures or interactions it supports are part of the behavior element. Its look and feel, and layout, are part of its presentation element. Flex user interfaces are dynamic and rich and support changes in view states as a user interacts with Flex. These view states represent an aggregation of changes to a view. The behavior element defines the triggers for these state changes, and the presentation elements manage the view state transitions. The behavior element name corresponds to the component name as we know it in version 3.x, say Button. The presentation element corresponds to a corresponding skin class, say ButtonSkin.

Like the component class, the skin class can be written in MXML. It can include new graphics elements like geometrical shapes, line drawings, text, and bitmaps. The graphical elements can be expressed in an XML language called FXG, which can be used in conjunction with MXML.

Figure A-2 shows a Flex 3 Button class split into a behavior and presentation class in Flex 4. Decoupling of behavior and presentation has also lead to reduction of redundant components. In Flex 3, as a case in point, we have two components, List and HorizontalList, to display a list of data vertically and horizontally. Once the List component is split into its behavior and presentation parts, you are left with one behavior class and one or more skin classes. In the new model, the skin class takes care of the vertical and horizontal layout of the list data.
So, developing a Flex 4 application involves the two separate but related tasks of writing a component’s behavior and creating its skin. For large projects, you may consider using a developer for the task of writing the logic and a designer for drawing out the presentation element. In smaller projects, you could combine the roles but still carry out the two separate tasks. Work on the two parts can be done in parallel, as the interfaces between the two are clearly defined.

In large projects, where designers take on the task of creating the look and feel, it’s possible to use the design assets from the familiar Adobe designer’s tools like Photoshop and Illustrator in Flex projects. Adobe has created a new tool, called Flash Catalyst, which was earlier codenamed Thermo, to convert and manipulate graphic elements to forms that can be used in a Flex application. Flash Catalyst allows one to include view state transitions on existing design elements. It has the option to export such artifacts in FXP format that become consumable within a Flex project in Flex Builder 4.

For our simple example, I will stick to using FXG to draw out a skin.

**Flex 4 Says Hello**

A simple Flex 4 application that prints out “Hello” as its label when you hover over a button is built in this subsection. On hovering over it, not only does the button say hello, but it also changes its background color and shape.

You can download Flash Builder 4 Beta, the latest version of the Flash Builder, from [http://labs.adobe.com/technologies/flashbuilder4/](http://labs.adobe.com/technologies/flashbuilder4/). Alternatively, set up the latest stable Gumbo SDK in your Flex Builder 3 first. Downloading and using Flash Builder 4 beta will give you access to the latest version of the SDK and the Flash Builder, whereas setting up the new SDK within the old Flex Builder will only provide a view into the new SDK.
Setting up the Flex 4 SDK in Flex Builder 3 is fairly straightforward. In general, it involves the following tasks to be carried out in sequence:

- Go to opensource.adobe.com and download the latest stable Gumbo SDK (with Adobe Add-ons) to your local drive.
- The downloaded file is a .zip file. Unzip the file to a local folder.
- Open up Flex Builder, and select the Preferences menu item from the Window menu.
- Within the Preferences panel, select the Flex-related options.
- Within the Flex tree node, select Installed Flex SDKs. You can add, remove, and edit SDK definitions there.
- Click the Add button on the right, and you will be prompted to provide a path and name for the new SDK you want to add.
- Specify the name and path to the SDK you want to add, and accept the changes by clicking the OK button.
  - Choose the root of the folder where you zipped the Gumbo SDK as the path. You can browse to the location using the file browser option, which is presented on clicking the Browse button.
  - Specify an appropriate name for the SDK. Possibly name it so that you can identify its version from the name.

Now you are ready to create your first elementary Flex 4 application. Create a new Flex project using Flex Builder. Call it HelloFlex4. Now create a Button in the main application as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
<FxApplication name="HelloFlex4"
    xmlns="http://ns.adobe.com/mxml/2009"
    backgroundColor="white">
    <layout>
        <BasicLayout />
    </layout>

    <FxButton horizontalCenter="0"
        verticalCenter="0"
        skinClass="MyButtonSkin"/>

</FxApplication>
```

The Button is the only user interface component in this application. Apart from defining its relative vertical and horizontal alignment, it also associates a Skin class called MyButtonSkin with the Button in the example. MyButtonSkin is the Skin class that defines the look and feel of the button. It associates a different look and feel with the up and the down button states. The source for this skin class is:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Skin xmlns="http://ns.adobe.com/mxml/2009">
    <Metadata>
        [HostComponent("mx.components.FxButton")]
    </Metadata>
</Skin>
```
When run, you will notice that the button is rendered as a cornflower blue square initially. The square button is shown in Figure A-3. When hovered over, it is transformed into a yellow circle, which is depicted in Figure A-4.

If you peek at the code carefully, you will notice a rectangle and an ellipse (with equal width and height) drawn using FXG. You will also notice a TextBox component containing a text definition, which is displayed on the button. Here, the text values are hard-coded, but they could be bound to model values like the button’s label.

A deeper scrutiny will also reveal that the colors of the displayed components are defined as properties of the graphic elements. The color corresponding to each state is stated using a dot notation state-specific extension of the color attribute. A few other attributes like text and font size also have similar state-specific values bound with them.
This was an elementary example, but hopefully it has given you a sense of how the behavior and presentation are separated out in Flex 4. In addition, you saw a first bit of FXG in action and were exposed to the new state-related enhancements.

For this write-up, we are done and ready to wrap up.

**Summary**

This was a terse and introductory write-up on Flex 4, the new Flex framework. Being a brief write-up, it concentrated on getting you started, setting the context for the discussions and exposing you to the first bits of the new framework.

The write-up has two parts. The first of these two parts described the new features and the backward-compatibility capabilities of Flex 4. The second one illustrated a typical development workflow and walked through an elementary application.

This write-up is the icebreaker and the context setter. Now that you are ready to learn Flex 4, you may want to explore more about it online at [http://opensource.adobe.com/wiki/display/flexsdk/Gumbo](http://opensource.adobe.com/wiki/display/flexsdk/Gumbo). Much of what you learned about BlazeDS is relevant to Flex 4.
-keep option, 2

A
access control techniques, 93–94
Accordion container, 24
ACL (Access Control List), 93–94
ActionScriptAdapter, 160, 163, 259
    CustomActionScriptAdapter, 268–269
default message adapter, 125
extending MessagingAdapter, 268
invoke method, 268
messaging, 124
ActiveMQ, 127
ActiveMQTopic, 129
adapter classes, hierarchical structure, 257
adapters, 67
custom, 134–135
dpHibernate, 204
dsadapters, 203
Gilead, 204
AdvancedDataGrid, 11
AIR, 41
    Java servers and, 41
Alerts, 18–19
allocation trace, 248–249
AMF (Action Message Format), 32–33, 43, 67, 253
    channels, 56
    native messaging, 124–126
AMFChannel, 86
    method calls, 85
AMFEndpoint, 86
AMFX, 96, 253, 278
    channel, 96
AmfxTypes interface, 96
anArray, 100

ANT, 53–54, 237
    running, 238
ANT Tasks, 3
    Gas3, 312
AOP (Aspect-Oriented Programming), 203, 205
Apache Camel, 290–291
Apache Commons DBCP, 287
    JNDI DataSource and, 288
Apache HttpClient, 67
Apache Tomcat, 109, 179
    BlazeDS, deploying, 51–52
    EAR files, 181
    TCD (Tomcat Client Deployer), 52
APIs (application programming interfaces)
    automation, 228, 234
    Core Flex, 228
    Java, 255
    Logging, 244
Application class, 4
Application tags, 4
ApplicationControlBar container, 25–26
Applications, controls, 19–27
applications, deploying, 51–52
AppServerLoginCommand, 109
arrays, 99
    AS3, 100
    associative, 100
    byte, length, 279
    operations, 297
AS3 (ActionScript 3), 1, 169
    arrays, 100
    classes, Java classes and, 320
    Flex clients, 84
    FlexMonkey, 233
    Gas3, 312–320
    model objects, populating, 320–323
    Number type, 98
    XSD and, 320–323

Download from Library of Wow! eBook <www.wowebook.com>
Aspect-Oriented Programming

Aspect-Oriented Programming, 113
associative arrays, 100
authentication
  Basic, 74
  Custom, 74
destination, 92
screen, 306
author's presentations, 69
automation API, 228, 234
AutoQuick, 228
AVM (ActionScript Virtual Machine), 326

B
Basic authentication, 74
bean properties, serialization, 101
BeanInfo type, 101
bidirectional relationships, 177
Binary build, 47
binary data, text-based data as alternative, 87
Bind to data, 298
Bindable metadata, 63
BlazeDS
  adapters
    Hibernate, 201–203
    JPA, 201–203
  behavioral definition, 42
  deploying
    in Apache Tomcat, 51–52
    in JBoss AS, 52–53
    in Jetty, 53
downloading, 47–49
messaging, 148–159
as Servlet, 54
structural definition, 42
blazeds.war file, 303
book catalog, 294
Book.as, 319–320
BookBase.as, 318–319
_BookEntityMetadata class, 309
Book.java, 317–318
BookService
  annotations, 122–123
  CRUD operations, 120
  remoting destination class, 123
  as remoting service destination, 121

BookService Spring bean, configuration, 121
book.xsd, 321
Boolean type, 98
bootstraping, Spring, 113–118
boxes, 20–21
branch coverage, 240
buffer, sizes, suggested, 72
builds
  nightly, 47
  release, 47
  stable, 47
Button, 6
ButtonBar
b-buttons, 7
direction property, 7
ButtonBar class, 6
buttons, 5
grouping, 6
byte arrays, length, 279
ByteArrayResponseWrapper class, 279

C
caching, 283–287
  Flash Player cache, 284
  origins, 284
  RSL (Runtime Shared Libraries), 284
  side effects, 284
Camel, 290–291
cascade delete operations, 177
cascade persist preference, 177
CascadeType enumeration, 177
CDYNE weather service, 300–302
centralizing rules, 283
channel failover, 272–274
channels, 54–55
  AMF, 56
  AMFX, 96
  endpoints, 86
  HTTP channel, 57
  HTTPS, 96–97
  Polling AMF channel, 57
  remote objects, 87
  Secure AMF, 56
  Secure HTTP channel, 57
  services-config.xml, 55–56
Index

Charles (proxy), 241
chat window, 159–163
  application, 160–163
check boxes, 14
class keyword, 63
classes
  Application, 4
  _BookEntityMetadata, 309
  ButtonBar, 6
  ByteArrayResponseWrapper, 279
  flex.messaging.services.ServiceAdapter, 255
  FtpClientPool, 130
  FtpFileSource, 130
  FtpInboundSynchronizer, 130
  HibernateProxy, 206
  HibernateUtil, 205
  HSQLDBConnectionHelper, 105
  JMS (Java Message Service), 164
  JMSConfigConstants, 164
  JMSConsumer, 164
  JMSExceptionEvent, 164
  JMSExceptionListener, 164
  JMSMessageEvent, 164
  JMSMessageListener, 164
  JMSProducer, 164
  JMSProxy, 164
  JMSQueueConsumer, 164
  JMSQueueProducer, 164
  JMSSettings, 164
  JMSTopicConsumer, 164
  JMSTopicProducer, 164
  LinkButton, 6
  ManageableComponent, 255
  MessageService, 163
  MessagingAdapter, 257
  mx.rpc.xml.SchemaLoader, 322
  NetConnectionChannel, 87
  NumericalValue, 216
  NumericalValueTest, 217, 219–220
  remoting and, 86–89
  RemotingDestinationExporter, 122
  ServiceAdapter, 255
  TestCase, 216
Clear Data Builder, CRUD applications, 311
Clear Toolkit, 311
click events, 6
clickable links, 5
client code, generating for WebService
  consumption, 300–302
client-side Flex code, 107–108
client-side logging, 243–245
clients
  command-line interface, 48
  desktop integrated, 48
  Flex, 117–118
  plug-in to an IDE, 48
  stand-alone, 48
Clone and Merge Transfer Pattern (CMTP), 202
clustering, 271–275
  channel failover, 272–274
  JGroups, 274–275
  vertical, 271–272
CMTP (Clone and Merge Transfer Pattern), 202
code coverage, 240
  Flexcover, 240
collaboration, online, 138
ComboBox, 14
Comet, 140–141, 165
  Jetty 7, 165–167
collection
  command-line interface, 48
  communication flow, remoting, 84–86
  compiled distribution, 47
  installation, 49–53
compiling, 3
  from source, 53–54
components
  custom, 36–38
  remoting and, 86–89
compressing data, 276–278
concurrent contracts, 282
condition coverage, 240
configuration
  files, names, 55
  messaging service, 58
  options, 54–58
  proxy, 58
  remoting, 57–58
  services-config.xml, 55–57
connection-manager property, 71
connectionFactory, 129
connectTimeout, 273
Consumer component, 155
consumers, 142
containers
  Accordion, 24
  ApplicationControlBar, 25–26
  Form, 26–27
  Grid, 21–23
  multiple, 24
  Panel, 23–24
  Repeater, 26
  stackable, 24
  TabNavigator, 24
  TitleWindow, 27
  ViewStack, 24
context root, 306
Continuation instance, 165
continuous integration, 239
  CruiseControl, 239–240
  Hudson, 239
Contributor entity, 185
controls
  bundles, 5
  data entry, 13–15
  Image, 18
  reusing data, 9–10
  Tree, 15–16
  user input, 13–15
  VideoDisplay, 18
Core Flex API, 228
createEntityManager method, 178
createlInstance method, 262
creationCompleteEffect, 33
creationCompleteEvent, 33
cross-domain utility layer, 283
cross-site scripting, 66
crossdomain.xml, 66
CRUD (create, read, update and delete)
  applications, 138, 200, 293
  Clear Data Builder, 311
  Flash Builder 4 and, 294–312
  managing operations, 199
  methods, Data/Services view, 308
  remoting service destinations, 303–312
  RemotingServiceCRUD, 305
REST, 294–300
_Super_BookService, 309
CruiseControl, 239–240
CSS (Cascading Style Sheets), 35
custom adapters, 134–135
Custom authentication, 74
custom components, 36–38
custom serialization, 101, 102
CustomActionScriptAdapter, 268–269
D
DAC (Discretionary Access Control), 94
data compression, 276–278
data entry, 13–15
data files, embedded, 28
data format, 278–280
data pushing, 138
data service, Java-based, 104–106
database
  coarse-grained, 170–171
  fine-grained, 170–171
  setup, 180
DataSource
  configuration, 288
  getConnection method, 288
  JBoss, 289
DDL (Data Description Language), 171
debugging
  allocation trace, 248–249
  DeMonster Debugger, 246
  Live Objects view, 247
  logging
    client-side, 243–245
    server-side, 245–246
  loitering objects, 248
  memory snapshot view, 247–248
  object references, 249
  object statistics, 250
  performance profile, 250–251
  profiling, 246–251
decision coverage, 240
decode method, 323
decomposition, 283
DELETE method, 69
DeMonster Debugger, 246
dependency, 199
deploying
BlazeDS
  in Apache Tomcat, 51–52
  in JBoss AS, 52–53
  in Jetty, 53
dynamic deployment, 52
static deployment, 51
design in mind, 326, 327
destinations, 67
  authentication, 92
  BookService, 123
  characteristics, specifying, 122
  configuration, 89–97, 156
  EJB3 objects as BlazeDS destinations, 264–266
  messaging service, 58
  methods, 84
  properties, 90
  remote services, 57
  securing, 91–94
  Spring beans, 122
devlopment workflows, 331–332
direction property, 7
DispatcherServlet
  declaring, 115
  MessageBrokerServlet, 114
DispatcherServlet class, 114
distribution work, 291–292
DividedBox, 20
document/encoded WSDL bindings, 77
document/literal WSDL bindings, 77
doFilter method, 277
Dojo project, 165
downloading
BlazeDS, 47–49
  Binary, 47
  Compiled Distribution, 47
  Source Distribution, 47
  Turnkey, 47
  FlexAntTasks, 237
  FunFX, 234
JUnit, 224
  nightly build, 303
dpHibernate adapter, 204–208
  Event entity, 206–207
  Servlet filter, 205
drawChart, 154
dsadapters, 203, 204
DTOs (data transfer objects), 152–153, 311
duration property, 34
dynamic deployment, 52
E
ear files, 150
EAR files, Tomcat, 181
Eclipse Ganymede, 179
  Gas3, 312
  JPA object, creating, 180–185
effects, 33–34
EJB3 objects as BlazeDS destinations, 264–266
EJBs (Enterprise Java Beans), 150
embedded data files, 28
endpoints, 54–55, 201
accessibility, 253
channels, 86
connecting to, 129
services-config.xml, 55–56
entities
  managed, 211–212
  metadata, 189
  programming, 185–188
  relationships, 172–174, 176–178
EntityManager, 174
  persistence contexts, 178–180
error handling, 76
event handlers, buttons, 6
Event Java class, 210–211
events
  click, 6
  creating, 38
  load, 30
  propagating, 38
  SchemaLoadEvent.LOAD event, 322
exclude-classes list, 262
exclude-methods, 92
extension points

extension points, 109
architecture, 253
  HTTPProxyAdapter, 258
  SOAPProxyAdapter, 258
externalizable interfaces, 101

F

Fiddler (proxy), 241
Fielding, Roy, 30
files
  ear, 150
  war, creating, 150
filter classes, methods, 277
findAll method, 298
Flash Builder, 3, 191
  CRUD, 294–312
Flash Builder 4 Beta, 332
Flash Player cache, 284
Flex. See also AIR
  clients, 117–118
  channels, 54–55
  endpoints, 54–55
  Hibernate integration, 199–201
  Java servers and, 41
  managed entities, 211–212
Flex 4
  compiler, 327
  framework evolution, 327
  new features, 326–327
  overview, 325–326
  previous versions, 327–331
Flex Builder, 3, 234, 332–333
  new project, 58
Flex Profiler, 246
FlexAntTasks, 237
  downloading, 237
flexclient-servlet.xml, 118
FlexComponent interface, 255
FlexContext, 110
Flexcover, 240
FlexFactory interface, methods, 266
flex.messaging.MessageBrokerServlet, 115
flex.messaging.services package, 88
flex.messaging.services.messaging.adapters, 125
flex.messaging.services.ServiceAdapter
  abstract class, 255
FlexMonkey, 227, 228–233
  Adapter, 228
  Agent, 228
  AS3, 233
  Automation API, 228
  Core Flex API, 228
  Flex applications, 229
  FlexMonkeyLauncher, 229
  FlexMonkeyTests application, 228
  MonkeyContactsTest.html, 228
  Tool, 228
FlexOSSCatalog, 191
FlexUnit, 215, 216–224
  addAsync method, 232
  ANT and, 237
  verifySomething method, 232
FlexUnit 4, 218
FlexUnitCore, 222
flexunit.junit.JUnitTestRunner, 238
flexUnit.junit.swc, 238
flexUnit.swc, 238
floating point types, primary keys, 175
foreign key constraint method, 173
Form container, 26–27
formatting data, 278–280
FTP (File Transfer Protocol), connecting to, 129
FtpClientPool class, 130
FtpFileSource class, 130
FtpInboundSynchronizer class, 130
function coverage, 240
functional decomposition, 283
functional testing, 227–243
FunFX
  automation API, 234
  downloading, 234
  RSpec, 235
  Ruby, 233
  running, 234
FXG, 326–327
G
Gas3, 312
   ANT tasks, 312
   Eclipse, 312
   Groovy templates, 320
   templates, 320
GDS (Granite Data Services), 33
Generate Details Form, 298
Generate Form, 298
generated IDs, 175
Geocoder.us, 28–29
Geronimo, 109
GET method, 68
getConnection method, 288
GFS (Grails Flex Scaffold), 212
Gilead, 135
downloading, 208
Gilead adapter, 204, 208–211
Gilead Flex BlazeDS Hibernate adapter, 203
Gilead Hibernate PersistentAdapter, 269
Glassfish, 109, 110
GraniteDS, 312
   Eclipse, 312
   LCDS and, 204
   org.granite.builder, 312
Graph Results listener, 242
Grid container, 21–23
GridItem component, 21
GridRow component, 21
grids, 11–13
Groovy templates, 316
   Gas3, 320
GUI (graphical user interface), 48
   registering, as UIListener, 222
Gumbo, 325, 327
GzipFilter Java Source, 276–277

H
Halo, 327
hamcrest assertions, 219, 222
handling errors. See error handling
HDividedBox, 20
HEAD method, 69
HelloBlazeDS.mxml, 61–62
HelloService, 62
Hibernate, 169
   BlazeDS adapter, 201–203
   EntityManager, 179
   Flex integration, 199–201
   JPA and, 169
   object/relational mapping, 188
   persistence framework, 188
   POJO, 189
   semantics, 199
Hibernate Reference documentation, 206
HibernateBean AS3 class, 207
hibernate.cfg.xml, 190
HibernateProxy class, 206
HibernateUtil class, 205
hierarchy of adapter classes, 257
HorizontalList, 10, 11
HSlider, 17
HSQLDB, 105, 179
HSQLDBConnectionHelper class, 105
HTTP
   error handling, 76
   methods
      DELETE, 69
      GET, 68
      HEAD, 69
      OPTIONS, 69
      POST, 69
      PUT, 69
      TRACE, 69
   proxy service, example, 69–70
HTTP-based communication, 66
HTTP channel, 57
HTTP GET method calls, 295
HTTP POST method calls, 295
HTTP request-response, 65
HTTP service proxy, 66–76
   configuration, 67
   proxy-services.xml, 67
HTTPChannel, 86
   AMFX, 96
HttpClient, 67, 70–71
HTTPEndpoint, 86
   AMFX, 96
HTTPMessage, 67
HTTPProxyAdapter, 69, 163
configuration options, 70–73
as extension point, 258
methods, 257
HTTPProxyDestination, 69
HTTPS, channels, 96–97
HTTPService component, 28–30
HTTPServiceCRUD, 295
HttpServlet, 114
Hudson, 239
I
I/O, 165
IBookService interface, 120–121, 304
IDataInput interface, 101
IDataOutput interface, 101
identity, 174
Image control, 18
images, embedded, 17–18
IMXMLSupport interface, 87
include-methods, 92
incremental updating, 199
incrementing/decrementing numbers, 15
infinite loop, 281
inheritance, 171–172
InitialContext instance, 147, 266
initialize method, 263
instances
Continuation, 165
setup, 46
integers, 98
generated IDs, 175
IntelliJ IDEA, 60, 179
interactions, 43
interface testing, 227–243
interfaces
AmfxTypes interface, 96
effects, 33–34
externalizable, 101
FlexComponent, 255
FlexFactory, 266
IBookService, 120–121, 304
IDataInput, 101
IDataOutput, 101
IMXMLSupport interface, 87
look and feel, 35–36
Manageable, 255
ServiceAdapter interface, 163
TomcatLogin, 110
user input, validating, 34–35
invoke method, 164, 203
isAcceptable state, 282
isReadable state, 282
J
JAR files
Hibernate, 194
JUnit, 224
Java
classes, AS3 classes and, 320
DTOs (data transfer objects), 311
WebORB for Java, 33
Java-based data service, 104–106
Java Build Path, 183
Java NIO, 165
JavaAdapter class, 110–111, 163, 258
java.persistence.Entity, 174
javascriptclient-servlet.xml, 118
JBoss AS
BlazeDS, deploying, 52–53
DataSource, 289
JCP (Java Community Process), 87–88
JPA, 169
JDBC connection pooling, 287–289
jdbc-user-service element, 132
J2EEE, server option, 149
Jetty, BlazeDS, deploying, 53
Jetty 7, Comet, 165–167
JGroups, clustering, 274–275
JMeter, 241–242
jms-destination, 129
JMS (Java Message Service), 145
administered objects, 145
API, 146–148
architecture, 145–146
chat window, 160
classes, 164
clients, 145
createSession method, 146
message consumer, 126
Download from Library of Wow! eBook <www.wowebook.com>
message producer, 126
MessagingAgent, 127
programming model, 146–148
provider, 145
quotes, 126
resource pooling, 289–290
Spring-specific JMS components, 126–129
topic/XchangeRateUpdate, 159
jms-message-destination element, 129
JMSAdapter, 164
chat window, 160
MessagingAdapter and, 258
JMSCfgConstans class, 164
JMConsumer class, 164
JMEexception, 126
JMEExceptionEvent class, 164
JMSMessageListener class, 164
JMSProducer class, 164
JMSQueueConsumer class, 164
JMSQueueProducer class, 164
JMSSettings class, 164
JmsTemplate, 126
boilerplate code and, 128
JMSTopicConsumer class, 164
JMSTopicProducer class, 164
JMX (Java Management Extensions), 87–88
JNDI (Java Naming and Directory Interface), 147, 266–267
JPA (Java Persistence API), 97
BlazeDS adapter, 201–203
Eclipse, project creation, 180–185
EntityManager, 174
Hibernate and, 169
integration, 191–199
JCP, 169
OpenJPA, 169
semantics, 199
Sessions, 190–191
JRE 6, 179
JSON (JavaScript Object Notation), 278
JSRs (Java Specification Requests), 87–88
JUnit, 215, 224–227
ANT tasks, 237
downloading, 224
ignoring tests, 227
timeout property, 227
JVM (Java virtual machine), 1
K
keywords, class, 63
L
Label component, 13
labels, 3
lazy loading, 199, 200
LazyInitializationException, 205
LBAC (Lattice-Based Access Control), 94
LCDS (LifeCycle Data Services), 32, 164–165, 191
GraniteDS, 204
scalable connectivity, 165
LineChart component, 153
LinkButton class, 6
links, clickable links, 5
List, 10
lists, 10–11
live objects view, 247
load event, 30
load testing, 240–243
logging
calls, remoting services, 94–95
client-side, 243–245
content metadata, 95
level, 95
server-side, 245–246
target, 95
trace method, 244
Logging API, 244
LoginCommand, 133
LogTarget, 244
loitering objects, 248
loops, infinite, 281
MAC (Mandatory Access Control)

M
MAC (Mandatory Access Control), 94
Manageable interface, 255
ManageableComponent class, 255
many-to-many relationships, 172, 177
many-to-one relationships, 177
mappedBy element, 177
matchers, 222
MDBs (Message Driven Beans), 291
MDP (Message Driven POJO), 128
memory snapshot view, 247–248
MenuBar component, 9
menus, 8–10
radio buttons, 9
submenus, 8
message-based communication, 45
message broker, 67, 88
message-broker element, 116
message service integration, 124–131
MessageBrokerServlet, 166
DispatcherServlet, 114
MessageDeserializer, 201
MessageDriveBeans (MDBs), 128
messageHandler event listener, 155
messages, 43
LCDS, 32
pull communication style, 44
push communication style, 44
MessageSerializer, 201
MessageService class, 163
MessageTemplate, 130
messaging
BlazeDS, 148–159
consumers, 142
producers, 142
publish-subscribe model, 143–145
messaging adapter, extending, 268–269
messaging-config.xml, 67, 124
configuration, skipping, 125
messaging domains, 142–145
point-to-point, 143
messaging service, 58. See also JMS (Java Message Service)
destination, 58
MessagingAdapter, 164
publish/subscribe messaging, 258
MessagingAdapter abstract class, 257
MessagingAgent (JMS), 127
META-INF folder, 150
creating, 195
persistence.xml, 187
metadata, entities, 189
method-level security constraint, 93
methods
doFilter, 277
findAll, 298
getConnection, 288
HTTP
DELETE, 69
GET, 68
HEAD, 69
OPTIONS, 69
POST, 69
PUT, 69
TRACE, 69
HTTPProxyAdapter, 257
initialize, 263
invoke, 164, 203
JUnit, 226
pushMessageToClients, 124, 163
sayHelloInString, 58
sendPushMessageFromPeer, 124, 163
ServiceAdapter class, 256
start, 263
suspend, 165
trace, 244
milestone releases, 47
modules, remoting and, 86–89
MultiplePOJOAdapter, 259–261
MXML, 1
language tags, 328
namespaces, 328
mx.rpc.xml.SchemaLoader class, 322
my-polling-amf, 125
MySQL, 179
DDL statements, 171
N
namespace collision, 330
cache messaging (AMF), 124–126
NetConnectionChannel class, 87
nightly builds, 47
downloading, 303
installation, 50
NIO channels, 280
NIO (New IO), 165
nonblocking IO, 280
null values, number types, 99
numbers, incrementing/decrementing, 15
numeric types, null values, 99
NumericalValue class, 216
NumericalValue.java, 225
NumericalValueTest class, 217, 219–220
NumericStepper, 15
NumericalValueTestHamcrest, 222–223

O
object references (debugging), 249
object representations, relational tables, 171
object statistics (debugging), 250
object viewpoint, 173
OLAPDataGrid, 11
one-to-many relationships, 173, 176
one-to-one relationships, 173, 176
online collaboration, 138
OpenJPA, 169
operations array, 297
OPTIONS method, 69
org.granite.builder, 312
org.springframework.flex.messaging
.integration.IntegrationAdapter, 129
org.springframework.flex.remoting, 122
org.springframework.jms.JmsException, 126
ORM (Object Relational Mapping), 169
OSS (Open Source Software), 180
oss_catalog_db, 194
OSSProduct entity, 185
OSSProduct.hbm.xml, 189
OSSProduct.java, 185–186
annotations, 187
OSSProduct.Service.java, 195–197

P
packages, flex.messaging.services, 88
Panel container, 23–24
path coverage, 240
performance profile, 250–251
PersistantAdapter, 269
PersistantMessagingAdapter, 209
persistence contexts, 178–180
persistence objects, states, 178–179
persistence operations, 179
persistence.xml, 187–188, 209
Pimento, 204
point-to-point messaging domain, 143
POJOs (Plain Old Java Objects), 83, 119, 254
Hibernate, 189
MDP (Message Driven POJO), 128
polling, 138
Polling AMF channel, 57
polymorphism, 171–172
PopUpButton, 8
PopUpMenuButton, 10
POST method, 69
post office analogy, 43–44
primary keys, 175–176
Producer component, 155
producers, 142
profiling, 246–251
progress indicators, 16–17
ProgressBar, 16
properties
connection-manager, 71
dataProvider, 6
direction, 7
duration, 34
remoting destinations, 90
timeout, 227
proxies
Charles, 241
canfiguration, 58
destinations, access control, 73–75
Fiddler, 241
traffic, logging, 75
web service proxy, 76–80
Wireshark, 241
proxy-config.xml, 70
proxy requests, information for, 258
proxy-services.xml, 67
ProxyContext, 258
ProxyFilter, 258
publish/subscribe messaging model, 143–145
  MessagingAdapter, 258
pull-based communication, 65
pull communication style, 44
push communication style, 44
pushMessageToClients method, 124, 163
PUT method, 69

R
radio buttons, 9
RadioButtons, 14
range indicators, 16–17
RBAC (Role-Based Access Control), 93, 94
RDSDispatcherServlet, 303
reconnectAttempts, 272
reconnectInterval, 272
reference, 174
relational tables
  associations, 173
  link tables, 173
  object representations, 171
relationships
  bidirectional, 177
  directionality, 176, 177
  many-to-many, 172
  multiplicity, 176
  one-to-many, 173
  one-to-one, 173
release builds, 47
releases
  Binary build, 47
  Compiled Distribution, 47
  Source Distribution, 47
  Turnkey build, 47
remote calls, types, 58–59
remote methods, invoking, 84, 85
remote objects, remote services and, 87
remote procedure calls, 32–33
remote service classes, initial data, 151
remote services, remote objects and, 87
RemoteClass metadata, 63
RemoteObject, 84, 254
  attributes, 85
remoting, 57–28
  classes and, 86–89
  communication flow, 84–86
  components and, 86–89
  destination, 57
  modules and, 86–89
  overview, 84–89
  Spring beans, 119–123
remoting-config.xml, 67
  XchangeRateDataService class, 151
remoting destinations. See destination
remoting services
  calls
    intercepting, 94–95
    logging, 94–95
  configuration, 89–97
remoting-service.xml, 106
RemotingDestination, 254
RemotingDestinationExporter factory class, 122
RemotingService, 254
RemotingServiceCRUD, 305
Repeater, 26
repository
  accessing, 48
  rules, 283
  state, 283
request-response-style communication, 43
requests, 43
requestTimeout, 273
resources, pooling, 287–291
  JMS, 289–290
responses to requests, 43
REST-CV, format, 28
REST (Representational State Transfer), 30
  CRUD, 294–300
RESTful interfaces, 292
resubscribeAttempts, 272
resubscribeInterval, 273
return types, specifying, 310
reusing data, 9–10
RIAs (Rich Internet Applications), 1, 41
  workload distribution, 291–292
RichTextEditor, 13
Index

root folder, 305–306
root URL, 306
RPC/encoded WSDL bindings, 77
RPC/literal WSDL bindings, 77
RPCs (remote procedure calls), 84
RSL (Runtime Shared Libraries), 284, 285
signed, 285–287
RSpec, 235
RTMP (Real Time Messaging Protocol), 139, 165
Ruby, 233, 234
RubyGems, 234
rules centralization, 283
runners, 221

S
sayHelloInString method, 58
SchemaLoader, instantiation, 322
SchemaLoadEvent event, 322
SchemaLoadEvent.LOAD event, 322
SchemaManager, 322
SDK (software development kit), 325
Secure AMF channels, 56
Secure HTTP channel, 57
security
crossdomain.xml, 66
custom, implementation, 108–110
destinations, 91–94
FlexContext, 110
method-level security constraint, 93
Spring, 131–134
sendMessage function, 162
sendPushMessageFromPeer method, 124, 163
serialization, 201
bean properties, 101
custom, 101, 102
server-side logging, 245–246
server-side proxies, configuration and, 67
servers, clustering, 271–275
service, 67
service abstraction, 295
<service> tag, 90
ServiceAdapter class, 255
extending, 258
methods, 256
ServiceAdapter interface, 163
services, HelloService, 62
services-config.xml, 55–57, 67, 254
factories, 267
Servlets, BlazeDS as, 54
SESIA, FTP, 130
signed framework RSL, 285–287
SignedRSLSampleApplication, 285
SimpleUrlHandlerMapping, 117
simulation feed, 156–159
simulation Java program, 159
sliders, 17
SmartSVN, 48
SOA (Service-Oriented Architecture), 282
SOAP (Simple Object Access Protocol), 30
web services, 30–32
SOAPProxyAdapter, as extension point, 258
source, compiling from, 53–54
Source Distribution, 47
Spark, 329–330
Spice Factory, Pimento, 204
Spring
bootstrapping, 113–118
database, authentication, 131–132
JMSTemplate documentation, 128
message listener containers, 291
security, 131–134
Spring beans, 264
remote access, securing, 133–134
remoting-desination configuration, 122
Spring MVC, 113
Front Controller pattern, 114
springblazedtest-servlet.xml, 116
SpringMessageBrokerFactoryBean, 116
SSL (Secure Socket Layer), 97
TLS and, 97
stable builds, 47
installation, 50
stackable container, 24
start method, 263
state repository, 283
statement coverage, 240
static deployment, 51
stress testing tools, 241
String type, 98

Download from Library of Wow! eBook <www.wowebook.com>
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclipse</td>
<td>48</td>
</tr>
<tr>
<td>submenus</td>
<td>8</td>
</tr>
<tr>
<td>Subversion (SVN)</td>
<td>48</td>
</tr>
<tr>
<td>suspend method</td>
<td>165</td>
</tr>
<tr>
<td>SVN (Subversion)</td>
<td>48</td>
</tr>
<tr>
<td>branches folder</td>
<td>49</td>
</tr>
<tr>
<td>SmartSVN</td>
<td>48</td>
</tr>
<tr>
<td>tags folder</td>
<td>49</td>
</tr>
<tr>
<td>TortoiseSVN</td>
<td>48</td>
</tr>
<tr>
<td>trunk folder</td>
<td>49</td>
</tr>
<tr>
<td>swf</td>
<td>2</td>
</tr>
<tr>
<td>swf format files</td>
<td>66</td>
</tr>
<tr>
<td>SWFloader</td>
<td>18</td>
</tr>
<tr>
<td>TabNavigator container</td>
<td>24</td>
</tr>
<tr>
<td>tabular representations</td>
<td>11–13</td>
</tr>
<tr>
<td>TCD (Tomcat Client Deployer)</td>
<td>52</td>
</tr>
<tr>
<td>TCP multicast</td>
<td>274</td>
</tr>
<tr>
<td>templates, Groovy</td>
<td>316</td>
</tr>
<tr>
<td>TestCase class</td>
<td>216</td>
</tr>
<tr>
<td>testing</td>
<td>215–216</td>
</tr>
<tr>
<td>FlexUnit</td>
<td>215</td>
</tr>
<tr>
<td>FlexUnitCore</td>
<td>222</td>
</tr>
<tr>
<td>functional</td>
<td>227–243</td>
</tr>
<tr>
<td>FunFX</td>
<td>233–237</td>
</tr>
<tr>
<td>interface</td>
<td>227–243</td>
</tr>
<tr>
<td>JUnit</td>
<td>215, 224–227</td>
</tr>
<tr>
<td>load testing</td>
<td>240–243</td>
</tr>
<tr>
<td>runners</td>
<td>221</td>
</tr>
<tr>
<td>TestNG</td>
<td>215</td>
</tr>
<tr>
<td>TestNG, 215</td>
<td></td>
</tr>
<tr>
<td>text-based data, as alternative to binary data</td>
<td>87</td>
</tr>
<tr>
<td>Text component</td>
<td>13</td>
</tr>
<tr>
<td>TextArea</td>
<td>13</td>
</tr>
<tr>
<td>TextInput</td>
<td>13</td>
</tr>
<tr>
<td>theories</td>
<td>222</td>
</tr>
<tr>
<td>ThreadLocal</td>
<td>205</td>
</tr>
<tr>
<td>TileList</td>
<td>10, 11</td>
</tr>
<tr>
<td>timeout property</td>
<td>227</td>
</tr>
<tr>
<td>TitleWindow container</td>
<td>27</td>
</tr>
<tr>
<td>TLS (Transport Layer Security)</td>
<td>97</td>
</tr>
<tr>
<td>SSL and</td>
<td>97</td>
</tr>
<tr>
<td>Tomcat. See Apache Tomcat</td>
<td></td>
</tr>
<tr>
<td>tomcat-users.xml</td>
<td>74</td>
</tr>
<tr>
<td>TomcatLogin interface</td>
<td>110</td>
</tr>
<tr>
<td>topic/XchangeRateUpdate</td>
<td>159</td>
</tr>
<tr>
<td>TopLink</td>
<td>169</td>
</tr>
<tr>
<td>TortoiseSVN</td>
<td>48</td>
</tr>
<tr>
<td>trace method</td>
<td>244</td>
</tr>
<tr>
<td>TraceTarget</td>
<td>244</td>
</tr>
<tr>
<td>Tree control</td>
<td>15–16</td>
</tr>
<tr>
<td>trees</td>
<td>15–16</td>
</tr>
<tr>
<td>branches, opening/closing</td>
<td>16</td>
</tr>
<tr>
<td>turnkey build</td>
<td>47</td>
</tr>
<tr>
<td>administrative console</td>
<td>50</td>
</tr>
<tr>
<td>applications</td>
<td>50</td>
</tr>
<tr>
<td>installation</td>
<td>49</td>
</tr>
<tr>
<td>UDP multicast</td>
<td>274</td>
</tr>
<tr>
<td>UIListener</td>
<td>222</td>
</tr>
<tr>
<td>URLs (Uniform Resource Locators), root URL</td>
<td>306</td>
</tr>
<tr>
<td>useAppserverSecurity</td>
<td>304</td>
</tr>
<tr>
<td>user-defined types</td>
<td>171</td>
</tr>
<tr>
<td>user input, validating</td>
<td>34–35</td>
</tr>
<tr>
<td>user input controls</td>
<td>13–15</td>
</tr>
<tr>
<td>UTC (Universal Time)</td>
<td>99</td>
</tr>
<tr>
<td>value</td>
<td>174</td>
</tr>
<tr>
<td>non-invasive value addition</td>
<td>199</td>
</tr>
<tr>
<td>value objects, populating</td>
<td>322</td>
</tr>
<tr>
<td>VDividedBox</td>
<td>20</td>
</tr>
<tr>
<td>vertical clustering</td>
<td>271–272</td>
</tr>
<tr>
<td>VideoDisplay control</td>
<td>18</td>
</tr>
<tr>
<td>ViewStack container</td>
<td>24</td>
</tr>
<tr>
<td>VSlider</td>
<td>17</td>
</tr>
<tr>
<td>war files, creating</td>
<td>150</td>
</tr>
<tr>
<td>web applications, data pushing</td>
<td>138</td>
</tr>
<tr>
<td>WEB-INF folder</td>
<td>150</td>
</tr>
<tr>
<td>WEB-INF/web.xml</td>
<td>114–115</td>
</tr>
<tr>
<td>web service proxy</td>
<td>76–80</td>
</tr>
<tr>
<td>configuration</td>
<td>79–80</td>
</tr>
</tbody>
</table>

350
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>web services, client code</td>
<td>300–302</td>
</tr>
<tr>
<td>WebApplicationContext</td>
<td>114</td>
</tr>
<tr>
<td>WebORB for Java</td>
<td>33</td>
</tr>
<tr>
<td>WebService component</td>
<td>30, 76</td>
</tr>
<tr>
<td>Wireshark (proxy)</td>
<td>241</td>
</tr>
<tr>
<td>workload distribution</td>
<td>291–292</td>
</tr>
<tr>
<td>WSDL (Web Services Description Language)</td>
<td>30, 76, 300–302</td>
</tr>
<tr>
<td>WTP (Web Tools Platform)</td>
<td>103</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td></td>
</tr>
<tr>
<td>XML (eXtensible Markup Language)</td>
<td></td>
</tr>
<tr>
<td>AMFX, 96</td>
<td></td>
</tr>
<tr>
<td>namespaces, 329</td>
<td></td>
</tr>
<tr>
<td>XMLDecoder</td>
<td></td>
</tr>
<tr>
<td>code, 323</td>
<td></td>
</tr>
<tr>
<td>decode method, 323</td>
<td></td>
</tr>
<tr>
<td>XMLEncoder, encode method</td>
<td>323</td>
</tr>
<tr>
<td>XSD (XML Schema), AS3 and</td>
<td>320–323</td>
</tr>
</tbody>
</table>
Take your library wherever you go.

Now you can access more than 200 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the Wrox Reference Library. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!
Related Wrox Books

Adobe AIR: Create - Modify - Reuse
This book walks you through eleven fully implemented AIR applications with source code that you can use as they currently exist or customize. Each project begins with a discussion of architecture and design, followed by code implementation. You’ll get hands-on knowledge of AIR application design and development that you can then use to build dynamic RIAs.

Beginning Adobe AIR: Building Applications for the Adobe Integrated Runtime
ISBN: 978-0-470-22904-0
This guide is a perfect mix of tutorials and hands-on coding, and provides the resources you need to build AIR applications quickly. Adobe expert Rich Tretola discusses the different programming languages and tools you can use for development and presents multiple methods for storing data, including within the file system and embedded database as well as storage on remote servers.

Professional Adobe Flex 3
Understand the potential of the Flex 3 platform through practical examples and hands-on advice on topics like desktop deployment, developing applications in MXML, creating custom flex components, charting, targeting AIR, and data interconnectivity.

Professional Cairngorm
This book provides a much-needed reference on Cairngorm, the oldest and most widely used Flex framework, by showing you how Cairngorm facilitates team development, encourages best practices for building Flex-based RIAs, and works well with medium- to large-scale projects in which code needs to be separated and organized.
Use BlazeDS to create exceptional rich applications with Flex and Java

BlazeDS is a popular, open source web application that can successfully be deployed into a Java application server, making it an ideal option for Flex and Java integration. With this comprehensive resource, you’ll walk through the process of creating Rich Internet Applications (RIA) using BlazeDS. After a look at BlazeDS basics, the book then moves on to explain how to configure, customize, and extend BlazeDS in order to integrate with new and existing Java data sources and applications with Flex-based RIA.

- Explains how to install BlazeDS from source as well as its compiled distributions
- Demonstrates ways to make user interfaces engaging and effective
- Introduces techniques for intercepting for control and management
- Features tips for configuring a remoting service and a destination
- Reviews remoting communication flow and remoting related classes, components, and modules
- Includes coverage of testing and debugging Flex and BlazeDS applications
- Shows how to extend BlazeDS with custom adapters and factories to connect to enterprise Java resources.
- Discusses methods for leveraging JPA and Hibernate with Flex
- Delves into the topic of communicating real-time via messages

Shashank Tiwari is a technology entrepreneur, speaker, author, and innovator. He is currently a managing partner & CTO at Treasury of Ideas (http://www.treasuryofideas.com), a technology-driven innovation and value optimization company.

Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.