CONTENTS INCLUDE:

- Flex Remoting
- Integrating Flex with Hibernate
- Integrating Pub / Sub Messaging
- Flex and Spring Security
- Hot Tips and more...

Flex 4 & Spring 3 Integration

By Jon Rose and James Ward

INTEGRATING FLEX AND SPRING

Adobe Flex has strong ties to Java, which include an Eclipse-based IDE and BlazeDS, its open source server-based Java remoting and web messaging technology. In late 2008, the Spring community began working on the Spring BlazeDS Integration project to add support for Flex development with Java and Spring.

By default BlazeDS creates instances of server-side Java objects and uses them to fulfill remote object requests. This approach doesn't work with Spring, as the framework is built around injecting the service beans through the Spring container. The Spring integration with BlazeDS allows you to configure Spring beans as BlazeDS destinations for use as remote objects in Flex.

This tutorial assumes that you are already familiar with Spring and Flex. If you need an introduction or refresher to either, check out the Very First Steps in Flex and Spring Configuration DZone Refcardz.

Hot Tip

BlazeDS provides simple two-way communication with Java back ends. Flash Player supports a serialization protocol called AMF that alleviates the bottlenecks of text-based protocols and provides a simpler way to communicate with servers. AMF is a binary protocol for exchanging data that can be used over HTTP in place of text-based protocols that transmit XML. Applications using AMF can eliminate an unnecessary data abstraction layer and communicate more efficiently with servers. To see a demonstration of the performance advantages of AMF, see the Census RIA Benchmark at: http://www.jamesward.org/census

To follow along with this tutorial you will initially need:

- An Eclipse 3.5 Distribution. You may choose either the standard distribution: Eclipse 3.5 (Galileo) for Java EE Developers (On Mac use the Eclipse Carbon version): http://www.eclipse.org/downloads/
  Or, for enhanced Spring support you might consider using the SpringSource Tool Suite:
  http://www.springsource.com/products/springsource-tool-suite-download/
- Flash Builder 4 (60 day Trial) installed as a plug-in for the Java EE Eclipse distribution: http://www.adobe.com/go/try_flashbuilder
- Tomcat 6: http://tomcat.apache.org/
- BlazeDS 4 (Binary Distribution):
  http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/
- Spring Framework 3.0.2 (vanilla release):
  http://www.springsource.org/download
- Spring BlazeDS Integration 1.0.3 (vanilla release):
  http://www.springsource.org/spring-flex
- AOP Alliance 1.0: http://sourceforge.net/projects/aopalliance/files/
- backport-util-concurrent 3.1 for the Java version you're using:
  http://sourceforge.net/projects/backport-jsr166/files/backport-jsr166/
- cglib 2.2 http://sourceforge.net/projects/cglib/files/
- asm 3.2 http://forge.ow2.org/projects/asm/

Manually downloading the dependencies is one way to get everything you need. Alternatively, you can use Maven or the SpringSource Tool Suite to automatically handle dependencies for you.

First install your chosen Eclipse distribution and then install Flash Builder 4 as a plug-in to the Eclipse distribution you just installed. Also extract the ZIP files for the other dependencies specified above. When you've completed these steps, run Eclipse. In Eclipse create a server that will run the application:

1. Choose File > New > Other
2. Select Server > Server
3. Click Next
4. Select Apache > Tomcat v6.0 Server
5. Click Next
6. Specify the location where Tomcat is installed and select the JRE (version 5 or higher) to use
7. Click Finish

There are two ways to begin setting up the Dynamic Web project in Eclipse. You can either start with a prepackaged project containing all of the dependencies or you can start from scratch and configure everything by hand. Starting from scratch will help you to better understand how everything fits together, however it will take a little bit more time to download all of the dependencies and configure Spring and BlazeDS.

To begin with the prepackaged project, download this project archive:
http://static.springsource.org/spring-flex/refcard/flexspring-refcard_justdeps.zip

Then, in Eclipse select File > Import and select the Existing Projects Into Workspace option. Using the Select Archive File option and Browse button, locate the flexspring.zip file, and then click Finish.

Alternatively to start from scratch, set up the server-side Java web project in Eclipse by creating a web application from the blazeds.war file (found inside the BlazeDS zip file).

In Eclipse, import the blazeds.war file to create the project:

1. Choose File > Import
2. Select the WAR file option. Specify the location of the blazeds.war file. For the name of the web project, type **flexspring**
3. Click Finish

First remove the xalan.jar file from the WebContent/WEB-INF/lib folder. Next, go into the project properties. One way to get there is by right-clicking on the project and selecting Properties. In the project

properties, select Java Build Path and then click the Source tab. Set the Default Output Folder to be the following (you will need to create the classes folder):

```
flexspring/WebContent/WEB-INF/classes
```

This causes all Java classes that are created in the project to be deployed in the web application.

In the `WebContent/WEB-INF/flex` folder update the services-config.xml file with the following contents:

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

Listing 1: services-config.xml - Updated to contain only the channels defined.

Now select the Servers tab in Eclipse. If the tab is not visible you can make it visible by choosing Windows > Show View > Servers. Right-click the Tomcat Server and select Add and Remove. To add the flexspring web application to the server, select it in the Available list and then click Add. Finally, click Finish.

Next, you need to add the required dependencies to the flexspring web application. Copy all of the Spring Framework libraries / JAR files (located in the dist folder) to the `WebContent/WEB-INF/lib` folder. Also copy the Spring BlazeDS Integration library (located in the dist folder) to the lib folder. Do the same for aopalliance.jar, backport-util-concurrent.jar, cglib-2.2.jar, asm-3.2.jar.

Simple Flex Remoting

To configure the server for simple Flex Remoting first edit the web.xml file in the WebContent/WEB-INF folder. Replace its contents with:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>flex.messaging.HttpFlexSession</listener-class>
    </listener>
    <servlet>
       <servlet-name>flexspring</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>flexspring</servlet-name>
        <url-pattern>/messagebroker/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Listing 2: web.xml - Setup for simple Flex Remoting with Spring.

Eclipse may try to change the web-app version to 2.4 instead of 2.5. If this happens just manually change it back to 2.5.

Spring will now handle requests to the `/messagebroker/` url.

Now configure Spring by creating an applicationContext.xml file in the `WebContent/WEB-INF` folder with the following contents:

```
<?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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <flex:message-broker>
        <flex:remoting-service default-channels="my-amf"/>
    </flex:message-broker>
    <context:component-scan base-package="flex" />
</beans>
```

Listing 3: The basic Spring configuration for Flex Remoting.

Listing 3 sets up the Flex `message-broker`, which will enable the Flex remoting-service using the my-amf channel. The component-scan will find classes in the "flex" package that have been annotated for Remoting. Typically with Spring, the configuration files are split out into more logical pieces. The approach used in Listing 3 is a simple way to get started but does not represent the best practice for Spring config file organization. For more details on how to better organize Spring config files see: http://refcardz.dzone.com/refcardz/spring-configuration

Now create a simple Java class that will be exposed through the AMF channel to a Flex application. In the src folder create a new Java Class in the flex package with the name `"HelloWorldService"`. Set the contents of the HelloWorldService.java file to:

```
package flex;
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.stereotype.Service;
@Service
@RemotingDestination
public class HelloWorldService {
    @RemotingInclude
    public String sayHello(String name) {
        return "howdy, " + name;
    }
}
```

Listing 4: HelloWorldService.java - A simple Java object exposed for remoting through Spring annotations.

On the `HelloWorldService` class you will notice two annotations. The `@Service` annotation tells Spring that the class is a Service and the `@RemotingDestination` annotation exposes the class as a remoting endpoint for Flex. This class also contains a single method named sayHello, which simply takes a string and returns a slightly modified version of the string. By default all public methods on a class are available for remoting. You can keep a public method from being exposed as a remoting endpoint by using the `@RemotingExclude` annotation.

Create the Flex Application

Now you can create a Flex application that will call the `sayHello` method on HelloWorldService. Remember that Flex applications run on the client side so all interactions with a server must happen over some remote call. Usually (and in the case here) these calls happen over HTTP. They can use a variety of serialization options including SOAP, JSON, and RESTful XML. In this case, Flex Remoting will use the binary AMF serialization protocol inside of HTTP request / response bodies.

To begin building the Flex application, simply create a new Flex Project in Eclipse. In the New Flex Project wizard, type **sayHello** as the name, select Web as the Application type, and set the Flex SDK Version to Flex 4.0 (usually the default). Also select J2EE as the Application Server Type, enable Use Remote Object Access Service, and select BlazeDS. Ensure that the Create Combined Java/Flex Project Using WTP option is not checked and then click Next. Now enter the information for your flexspring project. The Root folder is the `WebContent` folder in your flexspring project. The Root URL should be `http://localhost:8080/flexspring/`. The Context root should be `/flexspring`.

Click Finish to create the project. You should now see the template code for the application. Replace the code with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Declarations>
```

```
        <s:RemoteObject id="ro" destination="helloWorldService"
endpoint="http://localhost:8080/flexspring/messagebroker/amf"/>
    </fx:Declarations>
    <s:layout><s:VerticalLayout/></s:layout>
    <s:TextInput id="t"/>
    <s:Button label="sayHello" click="ro.sayHello(t.text)"/>
    <s:Label text="{ro.sayHello.lastResult}"/>
</s:Application>
```

Listing 5: sayHello.mxml - Basic Flex application using Remoting with Spring.

Hot Tip

Externalizing Configuration

These examples hardcode URLs into the applications. This is not recommended for real-world applications. Configuration should be externalized with Flex using one of the many methods available. Flex frameworks like Swiz, Parsley, or Spring ActionScript provide straightforward ways to externalize configuration. Using these methods is highly recommended for real-world applications.

When you save the file it should automatically compile. When it is compiled it should automatically be deployed in your web application. You are now ready to start the server and test the application. Go to the Servers View in Eclipse and start the Tomcat server. Watch the console and make sure that there are no startup errors.

Now run the sayHello application (one way is to right-click the sayHello.mxml file, select Run As, and then select Web Application). Your browser should open the sayHello.html file that then downloads and runs the Flex application. Type your name in the TextInput box and click the sayHello button. This will initiate a Flex Remoting request to the Tomcat server calling the Spring DispatcherServlet, which then will look up the service based on the destination helloWorldService. This destination is automatically mapped to the HelloWorldService Spring Bean. Then the sayHello method will be invoked on the bean, passing in the String that was specified in the RemoteObject call on the client. The method returns a new String, which is then serialized into AMF, inserted into the HTTP Response body, and sent back to the client. On the client side the Flex application will parse the AMF and then set the ro.sayHello.lastResult property to what was returned from the server. Data binding in the Label will note the property change and refresh its view of the data.

You now have completed a basic web application with a Flex application communicating with Spring through BlazeDS using the Spring BlazeDS Integration! Next you will add Hibernate into the mix and see how the Flash Builder 4 data-centric development features can help you quickly build data-driven Flex applications.

Integrating Flex with Hibernate through Spring

Using the same flexspring server as the Remoting example you can now add Hibernate to do data persistence. To get started you first need some additional Java libraries:

- Hibernate 3.5.2: http://hibernate.org/downloads.html
- H2 Database: http://h2database.com/html/main.html
- slf4j 1.5.8: http://www.slf4j.org/dist/

Extract the libraries and copy their JAR files into the `WEB-INF/lib` folder. For Hibernate copy the hibernate3.jar file, the JAR files in the lib/required folder, and the JAR file in the lib/jpa folder. For slf4j copy the slf4j-simple-1.5.8.jar file and for H2 copy the h2-1.2.134.jar file.

Now you will need to update the Spring config file so that your code can connect to an H2 database. Update the applicationContext.xml file with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:flex="http://www.springframework.org/schema/flex"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    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-3.0.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd">
    <flex:message-broker>
        <flex:remoting-service default-channels="my-amf" />
    </flex:message-broker>
    <context:component-scan base-package="flex" />
    <tx:annotation-driven />
    <jdbc:embedded-database id="dataSource" type="H2"/>
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan" value="flex" />
        <property name="hibernateProperties">
            <props>
               <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
               <prop key="hibernate.hbm2ddl.auto">create</prop>
            </props>
        </property>
    </bean>
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

Listing 6: applicationContext.xml - Hibernate and H2 Spring configuration.

The changes to the Spring config file introduce the ability to apply transactions through annotations (which you will see shortly), a DataSource to create a database connection, a SessionFactory used to manage the Hibernate Session, and a TransactionManager that handles the database transactions through the SessionFactory. The SessionFactory scans for Hibernate entities in the flex package.

In order to use the new data-centric development features in Flash Builder 4 you need to add the following servlet and servlet mapping to the web.xml file:

```
    <servlet>
        <servlet-name>RDSDispatchServlet</servlet-name>
        <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
        <init-param>
            <param-name>messageBrokerId</param-name>
            <param-value>_messageBroker</param-value>
        </init-param>
        <init-param>
            <param-name>useAppserverSecurity</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>10</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>RDSDispatchServlet</servlet-name>
        <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
    </servlet-mapping>
```

Listing 7: web.xml snippet - Enable the data-centric development in Flash Builder 4.

This new servlet from BlazeDS enables Flash Builder 4 to introspect the services that are configured on a server. If you add this servlet in a real application make sure that you either configure the security for the RDSDispatchServlet or remove the servlet when the application goes to production. Leaving a wide open RDSDispatchServlet in a production system can have adverse security implications.

Now create a Hibernate bean by creating a new Java class in the flex package called Friend with the following contents:

```
package flex;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
public class Friend {
    @Id
    @GeneratedValue
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    private String id;
    private String name;
}
```

Listing 8: Friend.java - The Hibernate Entity used to persist data.

You will notice that the Friend class has an `@Entity` annotation on it. This annotation enables this class to be used by Hibernate for data persistence. The `getId()` method of Friend has two annotations. The `@Id annotation` uses the id property as the primary key for the entity while the `@GeneratedValue` annotation will tell Hibernate to automatically assign a value when a new Entity is created.

In order to communicate with the Friend entity from Flex you will need a Remoting service to connect to. Similar to the HelloWorldService example, you can create a Spring bean that will be exposed to Flex. Create a new Java class in the flex package called FriendService with the following contents:

```
package flex;
import java.util.List;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
@Repository
@RemotingDestination
public class FriendService {
    private SessionFactory sessionFactory;
    @Autowired
    public void setSessionFactory(SessionFactory factory) {
        sessionFactory = factory;
    }
    @SuppressWarnings("unchecked")
    @RemotingInclude
    @Transactional
    public List<Friend> getFriends() {
       return sessionFactory.getCurrentSession().createQuery("from Friend").list();
    }
    @RemotingInclude
    @Transactional
    public void createFriend(String name) {
        Friend f = new Friend();
        f.setName(name);
        sessionFactory.getCurrentSession().save(f);
    }

}
```

Listing 9: FriendService.java - A Spring bean to manage Friend entities.

The FriendService class is annotated with `@Repository` to allow it to connect to a DataSource and `@RemotingDestination` to expose it as a Flex Remoting destination. The SessionFactory is *Autowired* into instances of the FriendService using Spring's dependency injection. In this case the `SessionFactory` that is injected is defined in the applicationContext.xml file. The `getFriends` method is included for Remoting access but also uses the `@Transactional` annotation to wrap the database query into a transaction. The `createFriend` method, which simply creates a new `Friend` entity and saves, is also Transactional and included for Remoting.

After making these changes, restart your Tomcat server. Verify that there were no errors on startup.

Create the Flex Application

Create a new Flex project called myFriends. Specify the same parameters as in the first example for the Application Type, Flex SDK version, and Server technology. Click Next and specify the same parameters as the sayHello project for Server Location. Click Finish.

Instead of writing code this time, switch to Design view for the myFriends.mxml file. Now in the Data/Services View click Connect to Data/Service, select BlazeDS, and then click Next. When you are asked for a username and password, check No Password Required and click OK. You should now see a list of services that are configured on your server. Among them should be friendService; select the checkbox next to that service and then click Finish. This automatically generates the client-side service stubs and value objects used to connect to friendService. In the Data/Services view you should now see the friendService methods and data types used.

To create the UI for the application, locate the DataGrid component in the Components view. Drag the DataGrid onto the Design canvas of the myFriends.mxml application. Now drag the getFriends method from the Data/Services View onto the DataGrid of the myFriends.mxml application. When you are prompted to confirm how you want to bind the service call to the DataGrid, click OK. Your DataGrid should now have two columns: id and name.

Next, you need some way to create new Friend objects. Right-click the `createFriend` method in the Data/Services View and select Generate Form. When you are presented with a dialog box asking for details about how to create the form, simply click Finish. You should now have a form in the Design canvas for myFriends.mxml. You can reposition the form so that it doesn't overlap the DataGrid. Now double-click the arg0 label and rename it by typing **Name.**

While you were manipulating elements in the Design canvas for the myFriends.mxml application you were actually writing Flex code. Switch back to Source view so you can see the code and make a few minor changes to the generated code.

First, you need to tell the generated FriendService instance how to connect to the server. Find the `<services:FriendService` tag and add a property called endpoint with a value of `http://localhost:8080/flexspring/messagebroker/amf`, for example:

```
<services:FriendService id="friendService"
    endpoint="http://localhost:8080/flexspring/messagebroker/amf"/>
```

Listing 10: myFriends.mxml snippet - FriendService with an endpoint.

Another small change you will want to make is to refresh the data in the DataGrid after a new Friend has successfully been created. To do that, first make the event parameter in the `dataGrid_creationCompleteHandler` function optional. For example:

```
protected function dataGrid_creationCompleteHandler(event:FlexEvent=null):void {
```

Listing 11: myFriends.mxml snippet - Optional FlexEvent parameter.

Finally update the `createFriendResult CallResponder` so that on a result event the `dataGrid_creationCompleteHandler` function is called.

```
<s:CallResponder id="createFriendResult"
result="dataGrid_creationCompleteHandler()"/>
```

Listing 12: myFriends.mxml snippet - Refresh the data on result event

Now run the myFriends application and create a new Friend by typing a name in the TextInput and clicking CreateFriend. Verify that the new Friend shows up in the DataGrid. You now have a Flex application that calls a Spring service to interact with Hibernate!

Hot Tip

Hibernate Lazy Loading and Flex
When BlazeDS serializes data to send to Flex, that data must be serializable. This can cause problems with lazy loading of associations in Hibernate. There are several approaches for dealing with this problem. One is to potentially not serialize the same object graph that is defined in Hibernate Entities but instead utilize Data Transfer Objects for serialization to AMF.

Integrating Pub / Sub Messaging with Flex and Spring

Flex Remoting is always done in a request / response manner. Sometimes applications also need to communicate in a publish / subscribe manner. BlazeDS provides the ability for Flex clients to connect to a message service and listen for messages or send messages. The message service can optionally be connected to other messaging systems like JMS, ActiveMQ, or any messaging system supported in Spring Integration.

Sending messages over HTTP is usually technically simple to enable because it can easily be built on HTTP's request / response nature. However receiving messages is more difficult since the client must initiate connections. To overcome this hurdle BlazeDS has various channels that are configured to do a pseudo push of messages from the server to the client. The primary channels available in BlazeDS are HTTP Streaming, HTTP Long-Polling, and HTTP Polling. Each of these channels has different advantages and disadvantages. For instance HTTP Streaming provides very low latency for sending messages but consumes one of the available HTTP connections the browser allows to a given site. HTTP Streaming in BlazeDS also uses the default HTTP connector in the Java app server, which usually has a very limited number of concurrent connections available. Java NIO provides a way to work around those connection limits but today NIO-backed connections are only available in Adobe's commercial superset of BlazeDS, called LiveCycle Data Services. HTTP Long-Polling and HTTP Polling can help alleviate some of the connection load but they also increase latency for sending messages from the server to the client. Ultimately which channel configuration you select will depend on your requirements. It's important to also note that channels can be configured to do automatic failover. So if an HTTP Streaming

connection can't be made then Flex can try Long-Polling or Polling. Channel configuration is done in the services-config.xml file.

The Spring BlazeDS integration exposes the Messaging features of BlazeDS so that they can be configured in Spring config and easily connected to Spring services and messaging systems. To configure a Messaging destination for Flex simply add the `message-service` to the message broker in the applicationContext.xml file. Then create a `message-destination` in that same file. For this example set the id of the `message-destination` to chat.

```
    <flex:message-broker>
        <flex:remoting-service default-channels="my-amf" />
        <flex:message-service
            default-channels="my-streaming-amf,my-polling-amf" />
    </flex:message-broker>
    <flex:message-destination id="chat" />
```

Listing 13: applicationContext.xml - Messaging enabled configuration.

Now create a new Flex Project just as before but call it "chat". Update the chat.mxml file with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
    <fx:Script>
        import mx.messaging.messages.AsyncMessage;
    </fx:Script>
    <fx:Declarations>
        <s:ChannelSet id="channelSet">
            <s:StreamingAMFChannel
                uri="http://localhost:8080/flexspring/messagebroker/streamingamf"/>
            <s:AMFChannel
                uri="http://localhost:8080/flexspring/messagebroker/amfpolling"
                pollingEnabled="true" pollingInterval="5"/>
        </s:ChannelSet>
        <s:Consumer id="c" destination="chat" channelSet="{channelSet}"
                    message="ta.text += event.message.body + '\n'"/>
        <s:Producer id="p" destination="chat" channelSet="{channelSet}"/>
    </fx:Declarations>
    <s:applicationComplete> c.subscribe(); </s:applicationComplete>
    <s:layout><s:VerticalLayout/></s:layout>
    <s:TextArea id="ta" width="300" height="150"/>
    <s:TextInput id="ti"/>
    <s:Button label="send" click="p.send(new AsyncMessage(ti.text))"/>
</s:Application>
```

Listing 14: chat.mxml - A simple Chat application using Flex Messaging

Restart the Tomcat server and run the application. Open two browser windows to verify that the messaging is happening correctly.

In the Chat application there is a ChannelSet declaration that defines how messages will be sent and received from the server. The first child inside the ChannelSet is the StreamingAMFChannel. Flex will try to make that connection first. Next is the AMFChannel with polling enabled on a 5 second poll interval. Also declared in the Chat application is a Consumer, which can listen for messages from the server using the ChannelSet. A message event handler on the Consumer tells the application to append the body of the message to the TextArea. A Producer declaration is used to send messages to the message service. Notice that both the Consumer and Producer have their destination set to chat, which is the id of the message-destination that was defined in the Spring configuration.

An applicationComplete event handler calls the subscribe method on the Consumer to begin listening for messages from the server.

The UI for the Chat application is very simple. The TextArea displays the chat messages. A TextInput allows the user to enter the text for a new chat message, and a Button allows the user to send the message. In the click handler for the Button a new AsyncMessage is created with its body set to the text the user entered in the TextInput. That message is then sent to the Producer, which connects to the server and sends the message. BlazeDS then tries to send that message out to all clients that are connected to the destination. When a connection is available the Consumer will receive any queued messages on the server and fire message events for each message.

Flex Messaging uses AMF internally so complex objects can be sent and received through the message service, just like with Remoting.

On top of the Producer and Consumer APIs, very powerful real-time and near real-time applications can be built. These can range from collaborative applications to trader desktops. Sometimes the message service is used simply for application management to do things like notify the client that their version of the data is stale.

Flex and Spring Security

Most real-world Flex applications require a user to login before they can perform certain actions in the application. Enabling user authentication and application security with Flex and Spring is simple to set up and implement. To get started you will need to download Spring Security 3.0.2 from http://www.springsource.org/download and copy the following JAR files into the `WEB-INF/lib` for the flexspring web app project:

- spring-security-acl-3.0.2.RELEASE.jar
- spring-security-config-3.0.2.RELEASE.jar
- spring-security-core-3.0.2.RELEASE.jar
- spring-security-web-3.0.2.RELEASE.jar

Now modify the applicationContext.xml file to add in some Spring Security information:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:flex="http://www.springframework.org/schema/flex"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:security="http://www.springframework.org/schema/security"
    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-3.0.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.0.xsd">
    <security:global-method-security secured-annotations="enabled"
        jsr250-annotations="enabled"/>
    <security:http entry-point-ref="preAuthenticatedEntryPoint">
        <security:anonymous enabled="false"/>
    </security:http>
    <bean id="preAuthenticatedEntryPoint"
class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"
/>
    <security:authentication-manager>
       <security:authentication-provider>
           <security:user-service>
            <security:user name="john" password="john" authorities="ROLE_USER" />
           </security:user-service>
       </security:authentication-provider>
    </security:authentication-manager>
    <flex:message-broker>
        <flex:remoting-service default-channels="my-amf" />
        <flex:message-service
            default-channels="my-streaming-amf,my-polling-amf" />
        <flex:secured/>
    </flex:message-broker>
... (file truncated)
```

Listing 15: applicationContext.xml snippet - Enable Security annotations.

The configuration in Listing 15 simply tells Spring about a basic user-service authentication-provider. Real-world applications use a database or LDAP server for user credentials and authentication.

Add the following filter, and filter-mapping to the web.xml file to enable Spring Security for the web application:

```
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

Listing 16: web.xml snippet - Enables Spring Security for the web application.

Modify the HelloWorldService.java file to protect calls to the sayHello method by adding the following annotation above the method:

```
    @Secured("ROLE_USER")
```

Listing 17: HellowWorldService.java snippet - Only users with ROLE_USER can call the sayHello method

Make sure you add the import statement for org.springframework.security.access.annotation.Secured then save the class and restart Tomcat. Now run the sayHello Flex application. Remote calls to the helloWorldService sayHello method should now fail. Update the sayHello.mxml file with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx">
        <fx:Script>
        import mx.rpc.AsyncResponder;
        import mx.rpc.events.FaultEvent;
        import mx.rpc.events.ResultEvent;
        private function handleFault(event:FaultEvent, o:Object=null):void {
            l.text = event.fault.faultString;
        }
    </fx:Script>
    <fx:Declarations>
        <fx:Boolean id="loggedIn">false</fx:Boolean>
        <s:RemoteObject id="ro"
                destination="helloWorldService"
                fault="handleFault(event)">
            <s:channelSet>
                <s:ChannelSet>
                    <s:AMFChannel uri="/flexspring/messagebroker/amf"/>
                </s:ChannelSet>
            </s:channelSet>
        </s:RemoteObject>
    </fx:Declarations>
    <s:layout><s:VerticalLayout/></s:layout>
    <s:TextInput id="t"/>
    <s:Button label="sayHello" click="ro.sayHello(t.text)"/>
    <s:Label id="l" text="{ro.sayHello.lastResult}"/>
    <s:Button label="login" enabled="{!loggedIn}">
        <s:click>
            ro.channelSet.login('john', 'john').addResponder(new AsyncResponder(
                function(result:ResultEvent, o:Object):void {
                    loggedIn = true;
                    l.text = "logged in";
                }, handleFault));
        </s:click>
    </s:Button>
    <s:Button label="logout" enabled="{loggedIn}">
        <s:click>
            ro.channelSet.logout().addResponder(new AsyncResponder(
                function(result:ResultEvent, o:Object):void {
                    loggedIn = false;
                    l.text = "logged out";
                }, handleFault));
        </s:click>
    </s:Button>
</s:Application>
```

Listing 18: sayHello.mxml - Enables the user to login and logout.

The updated sayHello application now has buttons that allow the user to login to the server. In this example the credentials for login are hardcoded. For a real-world application the credentials would likely come from TextInput controls – allowing the user to enter their username and password. Also the Boolean variable `loggedIn` tracks whether the user is authenticated or not. This Boolean is also used to enable and disable the login and logout buttons.

Rerun the sayHello application and try to call the `sayHello` method when not logged in. Now login and try it again. That is how simple it is to set up Spring Security with Flex!

CONCLUSION

In this Refcard, you learned how to use Flex Remoting and Messaging to connect to Spring and Hibernate. You also learned how to use Spring Security with Flex. As you can see, the Spring BlazeDS Integration project makes integrating Flex and Spring easy and straightforward. The combination of the two technologies creates a powerful platform for building robust RIAs. You can learn more about integrating Flex and Spring on the Spring BlazeDS Integration project site: http://www.springsource.org/spring-flex

To receive help with Spring BlazeDS integration ask questions in the Spring Forums: http://forum.springsource.org/forumdisplay.php?f=61

Download the full source code, configuration, and dependencies for all of the projects from: http://static.springsource.org/spring-flex/refcard/flexspring-refcard_completed.zip

Find screencasts which explain these code examples at: http://www.jamesward.com/flex-and-java-resources/

ABOUT THE AUTHOR

Jon Rose is the Flex Practice Director for Gorilla Logic, an enterprise software consulting company located in Boulder, Colorado. He is an editor and contributor to InfoQ.com, an enterprise software community. Visit his website at: www.ectropic.com

Gorilla Logic, Inc. provides enterprise Flex and Java consulting services tailored to businesses in all industries. www.gorillalogic.com

James Ward is a Technical Evangelist for Flex at Adobe. He travels the globe speaking at conferences and teaching developers how to build better software with Flex. Visit his website at: www.jamesward.com

First Steps in Flex is an introductory Flex book by James Ward and Bruce Eckel. It will give you just enough information, and just the right information, to get you started learning Flex. For more information visit: http://www.firststepsinflex.com

RECOMMENDED BOOK

First Steps in Flex will give you just enough information, and just the right information, to get you started learning Flex. Enough so that you feel confident in taking your own steps once you finish the book. For more information visit http://www.firststepsinflex.com.

BUY NOW
books.dzone.com/books/adobeflex

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-71-4
ISBN-10: 1-934238-71-6
50795
9 781934 238714
$7.95

Version 1.0