Spring BlazeDS Integration Reference Guide

Version 1.6.0.RC1

October 2014

Jeremy Grelle (Pivotal)

Chapter 1. Spring BlazeDS Integration Overview

1.1. Background

Spring has always aimed to be agnostic to the client technologies being used to access its core services, intentionally leaving options open and letting the community drive the demand for any new first-class integration solutions to be added to the Spring project portfolio. Spring BlazeDS Integration is an answer to the commmunity demand for a top-level solution for building Spring-powered Rich Internet Applications using Adobe Flex for the client-side technology.

BlazeDS is an open source project from Adobe that provides the remoting and messaging foundation for connecting a Flex-based front-end to Java back-end services. Though it has previously been possible to use BlazeDS to connect to Spring-managed services, it has not been in a way that feels "natural" to a Spring developer, requiring the extra burden of having to maintain a separate BlazeDS xml configuration. Spring BlazeDS Integration turns the tables by making the BlazeDS MessageBroker a Spring-managed object, opening up the pathways to a more extensive integration that follows "the Spring way".

1.2. Minimum major dependency versions Spring BlazeDS Integration 1.6 requires to run

Java 6

Spring 4.0

Adobe BlazeDS 4.0

1.3. Minimum major dependency versions required by optional features

Spring Security 3.0

Spring Integration 2.0

Hibernate 3.5

Chapter 2. Configuring and Using the BlazeDS MessageBroker with Spring

2.1. Introduction

The central component that must be configured to use Spring BlazeDS Integration is the `MessageBroker`. HTTP messages from the Flex client will be routed through the Spring `DispatcherServlet` to the Spring-managed `MessageBroker`. There is no need to configure the BlazeDS `MessageBrokerServlet` when using the Spring-managed `MessageBroker`.

2.2. Configuring the Spring DispatcherServlet

The `DispatcherServlet` must be configured as normal in web.xml to bootstrap a Spring WebApplicationContext. For example:

```
<!-- The front controller of this Spring Web application, responsible for handling all application
requests -->
<servlet>
    <servlet-name>Spring MVC Dispatcher Servlet</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>
```

2.3. Configuring the MessageBroker in Spring

A simplified Spring XML config namespace is provided for configuring the MessageBroker in your WebApplicationContext. To use the namespace support you must add the schema location in your Spring XML config files. A typical config will look something like the following:

```
<?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.xsd
           http://www.springframework.org/schema/flex
           http://www.springframework.org/schema/flex/spring-flex.xsd">
...
</beans>
```

This makes the Spring BlazeDS Integration configuration tags available under the `flex` namespace in your configuration files. The above setup will be assumed for the rest of the configuration examples to follow. For the full detail of every attribute and tag available in the config namespace, be sure to refer to the spring-flex-1.0.xsd as every element and attribute is fully documented there. Using an XSD-aware XML editor such as the one in Eclipse should bring up the documentation automatically as you type.

At a minimum, the `MessageBrokerFactoryBean` must be configured as a bean in your Spring WebApplicationContext in order to bootstrap the `MessageBroker`, along with a `MessageBrokerHandlerAdapter` and an appropriate `HandlerMapping` (usually a `SimpleUrlHandlerMapping`) to route incoming requests to the Spring-managed `MessageBroker`.

These beans will be registered automatically by using the provided `message-broker` tag in your bean definition file. For example, in its simplest form:

```
<flex:message-broker/>
```

This will set up the `MessageBroker` and necessary supporting infrastructure using sensible defaults. The defaults can be overriden using the provided attributes of the `message-broker` tag and its associated child elements. For example, the default location of the BlazeDS XML configuration file (/WEB-INF/flex/services-config.xml) can be overridden using the `services-config-path` attribute. The `MessageBrokerFactoryBean` uses Spring's `ResourceLoader` abstraction, so that typical Spring resource paths may be used. For example, to load the configuration from the application's classpath:

```
<flex:message-broker services-config-path="classpath*:services-config.xml"
```

The equivalent `MessageBrokerFactoryBean` definition using vanilla Spring configuration would be:

```
<!-- Bootstraps and exposes the BlazeDS MessageBroker -->
<bean id="_messageBroker" class="org.springframework.flex.core.MessageBrokerFactoryBean" >
    <property name="servicesConfigPath" value="classpath*:services-config.xml" />
</bean>
```

Note especially that with the `message-broker` tag, it is not necessary to assign a custom id to the MessageBroker, and it is in fact discouraged so that you won't have to continually reference it later. The only reason you would ever need to provide a custom id is if you were bootstrapping more than one `MessageBroker` in the same WebApplicationContext.

2.4. Mapping Requests to the MessageBroker

To properly route incoming requests to the Spring-managed `MessageBroker`, request mapping must be configured in three places:

1. `DispatcherServlet` mapping in web.xml
2. `HandlerMapping` in the Spring WebApplicationContext
3. Channel definitions in the BlazeDS services-config.xml

The simplest request mapping scenario is when the Flex front-end is the only client type for the application. In this case you can just map /messagebroker as the top-level path for requests. The mapping in web.xml would be:

```
<!-- Map all /messagbroker requests to the DispatcherServlet for handling -->
<servlet-mapping>
```

```
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

When using the `message-broker` config tag, a `SimpleUrlHandlerMapping` is installed that by default maps all incoming `DispatcherServlet` requests to the Spring-managed `MessageBroker` using a `/*`path pattern. The default mapping can be overridden by providing one or more `mapping` child elements. If you want to provide your own `HandlerMapping` bean configuration, you can disable the default using the `disable-default-mapping` attribute of the `message-broker` tag. The order of the installed `SimpleUrlHandlerMapping` can be set (for complex scenarios where multiple handler mapping types are installed in the same context) using the `mapping-order` attribute.

The `SimpleUrlHandlerMapping` in the Spring WebApplicationContext maps all requests to the Spring-managed `MessageBroker` via the `MessageBrokerHandlerAdapter`. The default setup installed by the `message-broker` config tag is equivalent to the following bean definitions:

```
<!-- Maps request paths at /* to the BlazeDS MessageBroker -->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
            /*=_messageBroker
        </value>
    </property>
</bean>

<!-- Dispatches requests mapped to a MessageBroker -->
<bean class="org.springframework.flex.servlet.MessageBrokerHandlerAdapter"/>
```

Channel definitions in the BlazeDS services-config.xml must correspond to the chosen mapping. For example, to set up a typical AMF channel in BlazeDS that matches the above mapping strategy:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

See the BlazeDS documentation for more information on configuring communication channels in services-config.xml.

2.5. Using Flex clients alongside Spring MVC Controllers

It could often be the case that your application needs to serve more than just Flex-based clients. For example, you may be constructing a RESTful architecture that is meant to serve multiple client-types. You could potentially even be consuming RESTful endpoints using the Flex HTTPService component. Spring MVC's controller model provides a simple, flexible means to create such RESTful endpoints. In these sorts of hybrid web application secenarios, you will need to consider an alternate mapping strategy.

The simplest approach is to use a hierarchical application context with multiple `DispatcherServlets`. In this approach, you configure your main application layer (services, security, supporting infrastructure, etc) in a parent context loaded via the `ContextLoaderListener`, and then configure all aspects of your Spring MVC controllers in one child `DispatcherServlet` context, and all aspects specific to your Flex client in a separate

child `DispatcherServlet` context. This approach could look as follows in web.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>flex</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>flex</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>spring-mvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>spring-mvc</servlet-name>
    <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

Here the parent application context is being assembled from a group of files ending in `-context.xml` contained in the `/WEB-INF/spring/` directory. The child context for the Flex-specific setup would be built (by convention) from `/WEB-INF/flex-servlet.xml`, and the context for the Spring MVC controllers would be built from `/WEB-INF/spring-mvc-servlet.xml`. This approach provides a nice separation of concerns and will allow Spring 2.5+ annotated controllers to work using their default configuration.

An alternate approach is to keep things consolidated under one `DispatcherServlet` context. The down-side to this approach is that it requires some additional manual configuration, and you have to modify your mapping approach accordingly, such as mapping `/spring/*` to the `DispatcherServlet`, mapping `/messagebroker/*` to the Spring-managed `MessageBroker` via the `mapping` XML namespace config tag, and modifying any BlazeDS channel definitions accordingly. You would override the default mapping strategy of the `message-broker` tag as follows:

```
<flex:message-broker>
    <flex:mapping pattern="/messagebroker/*" />
</flex:message-broker>
```

and you would have to account for the /spring/* mapping in your BlazeDS channel definitions. For example:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/spring/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

In addition to setting up the consolidated mapping strategy, you will also have to manually enable the correct `HandlerMapping` and `HandlerAdapter` for your Spring MVC controllers as described in the Spring MVC documentation, due to the fact that alternate `HandlerMapping` and `HandlerAdapter` beans are configured automatically when using the `message-broker` tag.

2.6. Using Spring-managed Destinations from the Flex Client

Explicit channel definition is a requirement when using dynamic destinations (meaning any destination that is added programmatically and not defined in the BlazeDS services-config.xml, i.e. the destinations created by the `remoting-destination` tag and the various `*-message-destination` tags). See Adobe's documentation for more detail.

The only way you don't have to explicitly define the ChannelSet on the client is if

1. you are using explicitly defined destinations in services-config.xml (i.e, not dynamic destinations) AND you compile your flex client against that file
2. your destination is using the application-wide default channel AND you compile your flex client against that file

Even if you weren't using dynamically created destinations it is debatable whether it is a good idea to ever compile your client against services-config.xml, thus coupling your client to your server configuration. It is often desirable to keep your flex client and your server side code as two distinct modules, but compiling against services-config.xml blurs the lines between those modules.

Our recommendation is that it is generally cleaner to keep the client-side configuration of ChannelSets explicitly contained within the client module. An excellent way to do this without having to hard-code the URLs in your client code is to use an ActionScript DI framework such as Spring ActionScript (a Spring Extensions project, formerly known as Prana).

If you choose to go the route of compiling your client against services-config.xml, note that you can at least keep the URL information out of the client code by using ServerConfig.getChannel as described in the referenced BlazeDS documentation.

2.7. Advanced MessageBroker Customization

The initialization of the `MessageBroker` by the `MessageBrokerFactoryBean` logically consists of two phases:

1. Parsing the BlazeDS XML configuration files and applying their settings to a newly created MessageBroker
2. Starting the MessageBroker and its services

A special `MessageBrokerConfigProcessor` callback interface is provided that allows custom processing to be done on the newly created MessageBroker after each phase, before it is made available for request processing. This interface is used internally by Spring BlazeDS Integration, but is also available for general use in advanced programmatic introspection and customization of the `MessageBroker`. A custom `MessageBrokerConfigProcessor` can be configured as a Spring bean and then registered with the `MessageBrokerFactoryBean` via the `config-processor` tag. For example, given a trivial implementation to log some additional info about the MessageBroker:

```
package com.example;

import org.springframework.flex.config.MessageBrokerConfigProcessor;

import flex.messaging.MessageBroker;
import flex.messaging.services.RemotingService;

public class MyDestinationCountingConfigProcessor implements MessageBrokerConfigProcessor {

    public MessageBroker processAfterStartup(MessageBroker broker) {
        RemotingService remotingService =
            (RemotingService) broker.getServiceByType(RemotingService.class.getName());
        if (remotingService.isStarted()) {
            System.out.println("The Remoting Service has been started with "
                    +remotingService.getDestinations().size()+" Destinations.");
        }
        return broker;
    }

    public MessageBroker processBeforeStartup(MessageBroker broker) {
        return broker;
    }
}
```

This class could be configured and registered with the `MessageBroker` as follows:

```
<flex:message-broker>
    <flex:config-processor ref="myConfigProcessor" />
</flex:message-broker>

<bean id="myConfigProcessor" class="com.example.MyDestinationCountingConfigProcessor" />
```

2.8. Using Custom Exception Translators

In order to propagate useful information back to the Flex client when an exception occurs on the server, the original exception must be translated into an instance of `flex.messaging.MessageException`. If special translation logic is not applied, a generic "Server.Processing" error will propagate to the client that doesn't give the client the chance to reason on the real cause of the error to take appropriate action. Special exception translators are configured by default for transforming Spring Security exceptions into an appropriate `MessageException`, but it could also be useful to provide custom translation for your own application-level exceptions.

Custom exception translation logic can be provided through implementations of the `org.springframework.flex.core.ExceptionTranslator` interface. These implementations must be configured as Spring beans and then registered through the XML configuration namespace as follows:

```
<!-- Custom exception translator configured as a Spring bean -->
<bean id="myExceptionTranslator" class="com.foo.app.MyBusinessExceptionTranslator"/>

<flex:message-broker>
    <flex:exception-translator ref="myExceptionTranslator"/>
</flex:message-broker>
```

2.9. Using Custom Message Interceptors

Custom message interceptors may be used to apply special processing logic to incoming and outgoing AMF

messages in their de-serialized Java form. For example, an interceptor can be used to inspect the contents of the incoming message, or to add extra information to the outgoing message.

Custom message processing logic is provided through implementations of the `org.springframework.flex.core.MessageInterceptor` interface. These implementations must be configured as Spring beans and then registered through the XML configuration namespace as follows:

```
<!-- Custom message interceptor configured as a Spring bean -->
<bean id="myMessageInterceptor" class="com.foo.app.MyMessageInterceptor"/>

<flex:message-broker>
    <flex:message-interceptor ref="myMessageInterceptor"/>
</flex:message-broker>
```

2.9.1. Resource Handling with Custom Message Interceptors

For housekeeping purposes, an additional `org.springframework.flex.core.ResourceHandlingMessageInterceptor` interface is available to use. Interceptors that implement this extended interface receive an additional guaranteed callback after message processing is completed, whether processing was successful or failed due to an exception being thrown by the Endpoint. This allows the interceptor to clean up any resources that it may have been using. This interface extends the basic `MessageInterceptor` interface, thus it is configured the same way using the `message-interceptor` tag.

2.9.2. Customizing the Message Interceptor Chain

The framework installs a number of predefined `MessageInterceptors` that are automatically configured through the use of the configuration namespace. These interceptors are configured in a specific order, and any custom interceptors are by default added to the beginning of the chain, in the order that their `message-interceptor` elements appear. It is possible to specify an explicit order via the `position`, `before`, and `after` attributes. The `position` attribute allows the user to *override* the framework-supplied interceptor at that position, or to simply specify that the interceptor should be at the beginning or end of the chain by using the `FIRST` and `LAST` values respectively. The `before` and `after` attributes allow specifying position *relative to* the framework-supplied filters.

The ordering and implementation of the framework-supplied filters is shown in the table below.

Table 2.1. Standard Interceptor Aliases and Ordering

Alias	MessageInterceptor Class	Namespace Element or Attribute
PER_CLIENT_AUTH_INTERCEPTOR	`PerClientAuthenticationInterceptor`	`message-broker / secured @ per-client-authentication`
LOGIN_MESSAGE_INTERCEPTOR	`LoginMessageInterceptor`	`message-broker / secured`
ENDPOINT_INTERCEPTOR	`EndpointInterceptor`	`message-broker / secured /`

Alias	MessageInterceptor Class	Namespace Element or Attribute
		`secured-endpoint-path` or `message-broker / secured / secured-channel`

2.10. Providing Custom Service Adapters

Using the XML config namespace automatically installs the needed implementations of `flex.messaging.services.ServiceAdapter` for use with the Remoting and Message services. Third-party adapters (such as those provided by the dpHibernate or Gilead projects) can be configured using the `org.springframework.flex.core.ManageableComponentFactoryBean`. This factory bean implementation is able to process arbitrarily complex configuration metadata supplied in JSON format (instead of arbitrarily complex XML as in the native BlazeDS configuration) and honors the lifecycle semantics (such as proper invocation of the `initialize` method) of the ManageableComponent. These custom adapters may be used by Spring-managed Remoting and Message destinations by either setting its id as the default for the Remoting or Message service, or by setting the `service-adapter` attribute for a specific destination (see the Remoting and Messaging chapters for further detail).

For example, to use the special adapter provided by dpHibernate as the default adapter with the Remoting service, the configuration would be similar to the following:

```
<bean id="hibernate-object" class="org.springframework.flex.core.ManageableComponentFactoryBean">
<constructor-arg value="net.digitalprimates.persistence.hibernate.HibernateAdapter"/>
    <property name="properties">
        <value>
            {"hibernate" :
                {"sessionFactory" :
                    { "class" : "net.digitalprimates.persistence.hibernate.utils.HibernateUtil",
                      "getCurrentSessionMethod" : "getCurrentSession"
                    }
                }
            }
        </value>
    </property>
</bean>

<flex:message-broker>
    <flex:remoting-service default-adapter-id="hibernate-object" />
</flex:message-broker>
```

Chapter 3. Exporting Spring Beans for Flex Remoting

3.1. Introduction

Using a Spring-managed `MessageBroker` enables Spring beans to be easily exported for direct remoting calls from a Flex client. This approach is quite similar to that taken with other remoting technologies in the core Spring Framework. Remoting is applied to existing Spring-managed beans as an external configuration concern. The `MessageBroker` transparently handles the process of serialization and deserialization between the Flex AMF data format and Java.

3.2. Configuring the Remoting Service

The BlazeDS `RemotingService` has traditionally been configured by the inclusion of a remoting-config.xml file in the BlazeDS XML configuration. When using only Spring-managed remoting destinations, this config file can be left out completely as the inclusion of the `message-broker` tag in your Spring configuration will cause the `RemotingService` to be configured with sensible defaults if none already exists at startup time. The end result is essentially equivalent to including the following minimal remoting-config.xml in your BlazeDS configuration:

```
<?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>

</service>
```

Note that this assumes that there is already an equivalent application-wide `default-channels` configuration. It is recommended that you set the desired service-specific channels (see example below) if not relying on an application-wide default setup. If no application-wide defaults exist, a best guess will be made by configuring the first available channel from the `MessageBroker` that uses an `AMFEndpoint` as the default for the `RemotingService`.

If you wish to have more explicit control over the defaults that will be set on the `RemotingService`, you can customize them via the `remoting-service` child element of the `message-broker` tag. For example:

```
<flex:message-broker>
    <flex:remoting-service default-adapter-id="my-default-remoting-adapter"
        default-channels="my-amf, my-secure-amf" />
</flex:message-broker>
```

If you have an existing remoting-config.xml for a legacy BlazeDS application, the

`RemotingDestinationExporter` will be able to work transparently with it, allowing you to gradually migrate to all Spring-managed remoting destinations.

3.3. Using the `remoting-destination` Tag

The `remoting-destination` configuration tag can be used to export existing Spring-managed services for direct remoting from a Flex client. Given the following Spring bean definition for a `productService` bean:

```
<bean id="productService" class="flex.samples.product.ProductServiceImpl" />
```

and assuming the existance of a Spring-managed `MessageBroker` configured via the `message-broker` tag, the following top-level `remoting-destination` tag will expose the service for remoting to the Flex client as a remote service destination named `productService`:

```
<!-- Expose the productService bean for BlazeDS remoting -->
<flex:remoting-destination ref="productService" />
```

By default, the remote service destination exposed to the Flex client will use bean name of the bean being exported as the service id of the destination, but this may be overridden using the `destination-id` attribute on the `remoting-destination` tag.

An alternate way of using the `remoting-destination` tag is as a child element of an top-level bean definition. This is even more concise and works well if you don't have a need to keep your domain-layer bean definitions separate from infrastructure concerns such as Flex remoting. (Keep in mind that keeping them separate can lead to easier testability of the core domain layer.) The following achieves the equivalent result to the previous example:

```
<bean id="productService" class="flex.samples.product.ProductServiceImpl" >
    <flex:remoting-destination />
</bean>
```

The methods that are exposed to be called by the Flex client can be more tightly controlled through use of the `include-methods` and `exclude-methods` attributes of the `remoting-destination` tag. The BlazeDS channels over which the destination is exposed can also be controlled using the `channels` attribute. (These attributes are available whether using the top-level or the nested version.) A more extensively customized example would look something like:

```
<flex:remoting-destination ref="productService"
    include-methods="read, update"
    exclude-methods="create, delete"
    channels="my-amf, my-secure-amf" />
```

The `remoting-destination` tag is transparently configuring a `RemotingDestinationExporter` bean instance for each bean being exported. The equivalent full bean syntax without the namespace support would be:

```
<!-- Expose the productService bean for BlazeDS remoting -->
<bean id="product" class="org.springframework.flex.remoting.RemotingDestinationExporter">
    <property name="messageBroker" ref="_messageBroker"/>
    <property name="service" ref="productService"/>
```

```
    <property name="destinationId" value="productService"/>
    <property name="includeMethods" value="read, update"/>
    <property name="excludeMethods" value="create, delete"/>
    <property name="channels" value="my-amf, my-secure-amf"/>
</bean>
```

3.4. Exporting Beans for Remoting with @RemotingDestination

The `@RemotingDestination` annotation may be used as an alternative to the XML remoting-destination tag when using annotation-based Spring configuration. `@RemotingDestination` is used at the type level to indicate the class being exported. `@RemotingInclude` and `@RemotingExclude` are used at the method level to mark the methods that should be included and excluded for remoting.

The following example illustrates the productService bean configured exclusively through annotations:

```
package flex.samples.product;

import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingExclude;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.stereotype.Service;

@Service("productService")
@RemotingDestination(channels={"my-amf","my-secure-amf"})
public class ProductServiceImpl implements ProductService {

    @RemotingInclude
    public Product read(String id) {
        ...
    }

    @RemotingExclude
    public Product create(Product product){
        ...
    }

    @RemotingInclude
    public Product update(Product product){
        ...
    }

    @RemotingExclude
    public void delete(Product product) {
        ...
    }
}
```

3.4.1. Supplying Externalized Channel Ids

One potential drawback of the `@RemotingDestination` approach is the potential need to hard-code AMF channel ids in multiple classes throughout your codebase. The specification of the `channels` property is optional (if not specified, the defaults for the RemotingService will be used), but in cases where they do need to be specified, one shouldn't need to edit all instances of `@RemotingDestination` any time the channel identifiers change. To support this need, the `channels` attribute is able to resolve values supplied in the `ApplicationContext` through a `PropertyPlaceholderConfigurer`. For example, the previous hardcoded example could be replaced with:

```
package flex.samples.product;
```

```
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.flex.remoting.RemotingExclude;
import org.springframework.flex.remoting.RemotingInclude;
import org.springframework.stereotype.Service;

@Service("productService")
@RemotingDestination(channels={"${channel1}","${channel2}"})
public class ProductServiceImpl implements ProductService {
    ...
}
```

assuming you have a `PropertyPlaceholderConfigurer` provided along the lines of:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
        <props>
            <prop key="channel1">my-amf</prop>
            <prop key="channel2">my-secure-amf, my-amf</prop>
        </props>
    </property>
</bean>
```

Chapter 4. Communicating with RESTful Spring MVC Endpoints using AMF

4.1. Introduction

Spring MVC 3 introduced support for building RESTful services using the @Controller programming model. This allows for serving multiple representations of the same content based on what the client has requested. For example, a single RESTful endpoint could be configured to serve HTML, XML, or JSON all from a single request handling Java method on the @Controller class. This approach allows a single service implementation to support a variety of different client types, automatically providing the representation that a particular client desires. In order to better support Flex as one of those client types, Spring BlazeDS Integration adds the necessary support to be able to support an AMF representation of a given RESTful resource.

`AmfView` and `AmfHttpMessageConverter` provide the foundation necessary to use AMF with Spring MVC @Controllers. The implementations are quite similar to `MappingJacksonJsonView` and `MappingJacksonHttpMessageConverter` which provide Spring's out-of-the-box JSON support. The basics of configuring these classes are shown throughout the rest of this chapter.

4.2. Using AmfView

`AmfView` is a Spring MVC View implementation that is intended to be used in conjunction with Spring MVC's ContentNegotiatingViewResolver. In particular, it is expected that it will be configured as a "default" singleton view implementation that can render any MVC model as AMF, rather than needing to have a unique instance per logical view name.

The following example shows a simple configuration of `ContentNegotiatingViewResolver` to support both JSON and AMF representations:

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="json" value="application/json"/>
      <entry key="amf" value="application/x-amf"/>
    </map>
  </property>
  <property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
      <bean class="org.springframework.flex.http.AmfView" />
    </list>
  </property>
</bean>
```

The above example includes setting the `mediaTypes` property to map file extensions to media types. This is helpful for clients (such as Flex) that aren't always able to set an appropriate `Accept` header. So given the above configuration, and a simple controller such as the following:

```
@Controller
@RequestMapping("/contacts")
public class ContactsController {
```

```
    @Autowired
    private IContactDAO contactDAO;

    @RequestMapping(method=RequestMethod.GET)
    public List<Contact> find() {
        return contactDAO.findAll();
    }
}
```

an HTTP `GET` request issued to either `http://localhost/myapp/contacts.amf` *without* an appropriate `Accept` header or to`http://localhost/myapp/contacts` *with* an accept header of `application/x-amf` will return the list of contacts as an AMF message that may then be deserialized to an `ArrayCollection` of ActionScript objects on the Flex client.

4.3. Using AmfHttpMessageConverter

`AmfHttpMessageConverter` is a HttpMessageConverter implementation that can convert to and from AMF. In order to be able to convert HTTP `POST` and `PUT` request message bodies from AMF to Java objects to be passed as a @RequestBody arguments to `@Controller` methods, it must be added to the converters used by Spring MVC's `AnnotationMethodHandlerAdapter`. In order to add AMF support while also leaving the framework's default `HttpMessageConverters` in place, a simple BeanPostProcessor such as the following example can be used:

```
package org.springframework.flex.samples.rest;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.flex.http.AmfHttpMessageConverter;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.util.ObjectUtils;
import org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter;

public class HandlerAdapterPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException {
        if (bean instanceof AnnotationMethodHandlerAdapter) {
            AnnotationMethodHandlerAdapter adapter = (AnnotationMethodHandlerAdapter) bean;
            HttpMessageConverter<?>[] converters = adapter.getMessageConverters();
            adapter.setMessageConverters(
                (HttpMessageConverter<?>[]) ObjectUtils.
                addObjectToArray(converters, new AmfHttpMessageConverter()));
        }
        return bean;
    }

}
```

With this configuration in place, and a controller such as the following:

```
@Controller
@RequestMapping("/contacts")
public class ContactsController {
```

```
    @Autowired
    private IContactDAO contactDAO;

    @RequestMapping(method=RequestMethod.POST)
    public Contact create(@RequestBody Contact contact) {
        return contactDAO.create(contact);
    }
}
```

the Flex client can send an HTTP `POST` request with an ActionScript `Contact` serialized to AMF in the message body to `http://localhost/myapp/contacts` in order to persist that contact to the database.

4.4. Interacting with RESTful AMF Endpoints from the Flex Client

Effectively interacting with a RESTful service from the Flex client using AMF is generally a matter of dropping down to some lower-level Flash APIs such as `flash.net.URLRequest` and `flash.net.URLStream`. There are some limitations to how a Flash/Flex application can interact with a REST service from within the browser. These limitations are generally imposed by the host browser, and are similar to the limitations imposed on HTML pages. Fortunately, Spring MVC already accounts for such browser limitations and provides useable workarounds.

The first major limitation is that the Flash player is typically not able to set the `Accept` header for the request to `application/x-amf` when using `URLRequest`. Spring supports mapping file extensions to media types as an alternative (see the example Section 4.2, “Using AmfView” configuration for how this is achieved).

The second major limitation is the inability to properly send HTTP `PUT` and `DELETE` requests. Spring provides the HiddenHttpMethodFilter to help cope with this. With the filter configured, a `DELETE` request (for example) could be simulated by sending a `POST` request to a URL such as `http://localhost/myapp/contacts/1.amf?_method=DELETE`.

A complete `insync-rest` example of interacting with a Spring MVC @Controller is now provided in the Test Drive. This sample duplicates the full functionality of the complete inSync application that was originally built using BlazeDS Remoting. It shows in detail how to work around Flash player's limitations to use AMF to interact with a RESTful service that supports it.

Chapter 5. Enhanced AMF Support

5.1. Customizing AMF Conversion

While BlazeDS provides excellent support for serialization/deserialization between Java beans and AMF, there are certain limitations in the types of objects that can be handled out-of-the-box. For example, objects to be converted to/from AMF must have getter and setter methods and must have a public no-arg constructor. In order to allow for a more flexible approach, we have provided an extension mechanism for easily customizing the AMF type conversion process through Spring configuration.

5.1.1. Configuring AMF Type Conversion

A special Spring-aware `PropertyProxy` (a BlazeDS-specific interface for customizing serialization) is provided that uses Spring's `PropertyAccessor` interface for type introspection and delegates to the Spring 3+ `ConversionService` to allow for additional property conversion logic to be plugged in during the serialization/deserialization process. This enhanced `PropertyProxy` optionally supports:

- Direct field access for properties (instead of requiring getters and setters)
- Use of an alternate constructor annotated with `@AmfCreator` for deserialization (instead of a public no-arg constructor)
- Ignoring certain properties during serialization and/or deserialization when marked with `@AmfIgnore` or `@AmfIgnoreField`

To take advantage of the enhanced AMF support, an instance of `SpringPropertyProxy` must be registered for each individual type that could potentially be converted to/from AMF. A specialized set of `MessageBrokerConfigProcessors` are provided that take varied approaches to locating the types to be registered:

Table 5.1. AMF Handling MessageBrokerConfigProcessors

Class	Implementation
org.springframework.flex.core.io. HibernateConfigProcessor	Uses the Hibernate metadata API to locate mapped Hibernate types to register for AMF conversion and configures specialized Hibernate `Converters`. See Section 5.2, “Working with Hibernate”
org.springframework.flex.core.io. JpaHibernateConfigProcessor	Uses the Hibernate metadata API to locate mapped Hibernate JPA types to register for AMF conversion and configures specialized Hibernate `Converters`. See Section 5.2, “Working with Hibernate”
org.springframework.flex.core.io. ClassPathScanningAmfConversionService ConfigProcessor	Uses classpath scanning to locate types to register for AMF conversion. See Section 5.3, “Using Classpath Scanning for AMF Configuration”

Explicit configuration of one of these classes is done just as with any other `MessageBrokerConfigProcessor`,

by defining the desired implementation as a bean and wiring it into the `MessageBroker` using the `config-processor` namespace tag.

5.1.1.1. Using Direct Field Mapping

By default, the `SpringPropertyProxy` will read and write property values during AMF conversion using JavaBean standard get and set methods, just as the default BlazeDS `BeanProxy` does. Alternatively, you many configure it to access fields directly. This is useful in such cases as when you have a class that does not expose public getter or setter methods, such as the following:

```
@Entity
public class PackagePrivatePerson {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    Integer id;

    @Version
    @Column(name = "version")
    Integer version;

    String name;
}
```

In order to be able to convert such a class to/from AMF, you must configure the `useDirectFieldAccess` property of one of the supplied AMF configuration processors. For example:

```
<flex:message-broker>
    <flex:config-processor ref="myJpaConfigProcessor" />
</flex:message-broker>

<bean id="myJpaConfigProcessor" class="org.springframework.flex.core.io.JpaConfigProcessor">
    <property name="useDirectFieldAccess" value="true" />
</bean>
```

5.1.1.2. Deserializing Immutable Objects with @AmfCreator

By default, the `SpringPropertyProxy` will try to instantiate types using a public no-arg constructor during the AMF deserialization process. Types that do not have a no-arg constructor must provide an alternate constructor annotated with `org.springframework.flex.core.io.AmfCreator`, whose properties are in turn annotated with `org.springframework.flex.core.io.AmfProperty`, in order to be eligible for AMF conversion. For example:

```
public class ImmutableValueObject {

    final String foo;

    final Integer zoo;

    @AmfCreator
    public ImmutableValueObject(@AmfProperty("foo") String foo, @AmfProperty("zoo") Integer zoo) {
        this.foo = foo;
        this.zoo = zoo;
    }

    public String getFoo() {
        return this.foo;
    }
```

```
    public Integer getZoo() {
        return this.zoo;
    }
}
```

5.1.1.3. Ignoring Properties During AMF Conversion

Sometimes it is desirable to ignore certain properties during serialization and/or deserialization. For example, there may be a property in your Java object that you do not wish to ever send to the Flex client. `SpringPropertyProxy` provides a means of achieving this by adding some additional annotations to your Java objects.

The annotation `org.springframework.flex.core.io.AmfIgnore` may be used on getter and setter methods. Annotating a getter method with `@AmfIgnore` will cause the property to be ignored during serialization to AMF, and annotating a setter method with `@AmfIgnore` will cause the property to be ignored during deserialization from AMF. For example:

```
public class IgnorablePropsObject {

    //Will be ignored on both serialization and deserialization
    private String foo;

    //Will be ignored only on serialization
    private String bar;

    //Will be ignored only on deserialization
    private String baz;

    @AmfIgnore
    public String getFoo() {
        return foo;
    }

    @AmfIgnore
    public void setFoo(String foo) {
        this.foo = foo;
    }

    @AmfIgnore
    public String getBar() {
        return bar;
    }

    public void setBar(String bar) {
        this.bar = bar;
    }

    public String getBaz() {
        return baz;
    }

    @AmfIgnore
    public void setBaz(String baz) {
        this.baz = baz;
    }
}
```

When using direct field mapping, the annotation `org.springframework.flex.core.io.AmfIgnoreField` may be used on fields to achieve equivalent functionality. For example:

```

```

```
public class IgnorableFieldsObject {

    //Will be ignored on both serialization and deserialization
    @AmfIgnoreField
    private String foo;

    //Will be ignored only on serialization
    @AmfIgnoreField(onDeserialization=false)
    private String bar;

    //Will be ignored only on deserialization
    @AmfIgnoreField(onSerialization=false)
    private String baz;

}
```

5.1.1.4. Providing Custom Converters

Customizing the way Java objects are converted to/from AMF is a matter of providing Spring 3 type `Converters` and registering them with the `ConversionService`. The provided AMF config processors all extend from `org.springframework.flex.core.io.AbstractAmfConversionServiceConfigProcessor`, which provides a template method for registering additional converters. One such use for this would be if you wanted to use classpath scanning to locate AMF-eligible types, but still wanted to register the specialized Hibernate converters. The following example illustrates this:

First, we configure the `ClassPathScanningAmfConversionServiceConfigProcessor` using the Spring 3 @Configuration style:

```
package com.foo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.convert.converter.ConverterRegistry;
import org.springframework.flex.core.io.ClassPathScanningAmfConversionServiceConfigProcessor;
import org.springframework.flex.hibernate3.io.HibernateProxyConverter;
import org.springframework.flex.hibernate3.io.PersistentCollectionConverterFactory;

@Configuration
public class CustomConverterConfiguration {

    @Bean
    public ClassPathScanningAmfConversionServiceConfigProcessor myConfigProcessor() {

        return new ClassPathScanningAmfConversionServiceConfigProcessor("com.foo.domain") {

            @Override
            protected void configureConverters(ConverterRegistry registry) {
                registry.addConverter(new HibernateProxyConverter());
                registry.addConverterFactory(new PersistentCollectionConverterFactory());
            }

        };
    }
}
```

Then we simply reference that config processor as usual in our `MessageBroker` configuration:

```
<flex:message-broker>
    <flex:config-processor ref="myConfigProcessor" />
</flex:message-broker>
```

5.2. Working with Hibernate

Using Hibernate to handle persistence concerns is quite common in Spring applications. A common problem arises, though, when attempting to directly use Hibernate-loaded entities in AMF messages with BlazeDS. In essence, the default BlazeDS serialization mechanism tries to walk the entire object graph, forcing initialization of any lazily loaded associations that might be encountered. This generally leads either to LazyInitialization errors, or (if using a solution such as Spring's `OpenSessionInViewFilter`) the undesirable n+1 selects scenario. The other problem that arises is when associations are eagerly loaded, but still wrapped in a Hibernate-specific collection or proxy, which can cause unexpected results in the way the association gets serialized.

Spring BlazeDS Integration provides a simple solution that will allow the direct use of Hibernate entities in AMF messages. A few special Hibernate-aware `Converters` are automatically installed that understand how to properly convert Hibernate-specific types when they are encountered.

5.2.1. Configuring Hibernate Entity Serialization/Deserialization Support

The Hibernate serialization/deserialization support will be automatically configured with a default `ConversionService` upon application startup if Hibernate is detected on the classpath. This support is provided in the form of two different `MessageBrokerConfigProcessors` - one for "native" Hibernate and one for using Hibernate as a JPA provider. The proper one will be configured based on the presence of the JPA api on the classpath.

The default behavior of the `SpringPropertyProxy` (in the case of both "native" Hibernate and JPA), when this support is configured is as follows:

1. Check each bean property to see whether it is contains a Hibernate-specific Proxy or Collection
2. If the Proxy/Collection is initialized, unwrap the underlying value for serialization
3. If the Proxy/Collection is uninitialized, return `null` for the value to be serialized

At startup time, Hibernate's Metadata API is used to determine which types to register with BlazeDS to use the specialized `SpringPropetyProxy`.

The default behavior may be extended/overridden by manually deploying either the `HibernateConfigProcessor`, `JpaHibernateConfigProcessor`, or some other custom extension of `AbstractAmfConversionServiceConfigProcessor` as a Spring bean and wiring it into the `MessageBroker` via the `config-processor` XML namespace tag as shown in Section 5.1.1, “Configuring AMF Type Conversion”.

5.3. Using Classpath Scanning for AMF Configuration

As an alternative to using the Hibernate Metadata API to find types to which the `SpringPropertyProxy` should be applied, a more general config processor implementation, `org.springframework.flex.core.io.ClassPathScanningAmfConversionServiceConfigProcessor`, is supplied as an alternative that will instead find types using Spring's internal classpath scanning support. The implementation scans recursively starting from a given base package, and all classes found in the scan will be registered to have the `SpringPropertyProxy` applied to them for AMF conversion. The scanning process may be customized further by configuring `org.springframework.core.type.filter.TypeFilter` implementations

to either include or exclude matching types. For example, Spring provides out-of-the-box RegEx-based and Annotation-based `TypeFilter` implementations. The easiest way to configure this implementation is by using the Java config support provided in Spring 3+, as in the following example:

```
package com.foo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.convert.converter.Converter;
import org.springframework.core.convert.converter.ConverterRegistry;
import org.springframework.core.type.filter.AnnotationTypeFilter;
import org.springframework.flex.core.io.ClassPathScanningAmfConversionServiceConfigProcessor;
import org.springframework.flex.hibernate3.HibernateProxyConverter;
import org.springframework.flex.hibernate3.PersistentCollectionConverterFactory;

@Configuration
public class ApplicationConfig {

    @Bean
    public ClassPathScanningAmfConversionServiceConfigProcessor amfConfigProcessor() {
        ClassPathScanningAmfConversionServiceConfigProcessor configProcessor =
            new ClassPathScanningAmfConversionServiceConfigProcessor("com.foo.domain") {

                @Override
                protected void configureConverters(ConverterRegistry registry) {
                    registry.addConverter(new HibernateProxyConverter());
                    registry.addConverterFactory(new PersistentCollectionConverterFactory());
                    registry.addConverter(new MyCustomConverter());
                }

            };

        configProcessor.addIncludeFilter(new AnnotationTypeFilter(MyCustomAmfAnnotation.class));

        return configProcessor;
    }

}
```

```
<flex:message-broker>
    <flex:config-processor ref="amfConfigProcessor" />
</flex:message-broker>
```

Chapter 6. Securing BlazeDS Destinations with Spring Security

6.1. Introduction

Spring Security provides an extremely flexible alternative to the container-based security support provided out-of-the-box with BlazeDS. Spring BlazeDS Integration provides explicit integration support for incorporating Spring Security smoothly into your Flex/BlazeDS application. Spring Security provides a wealth of different configuration options, but rather than go into the many different combinations here, we'll leave most of that to the Spring Security documentation.

6.1.1. A simple Spring Security 3 configuration

Here is a simple Spring Security starting configuration for use in conjunction with the explicit integration features provided by Spring BlazeDS Integration that should be a solid starting point for securing a typical Flex application:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="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.xsd
                        http://www.springframework.org/schema/security
                        http://www.springframework.org/schema/security/spring-security-3.2.xsd">

    <http entry-point-ref="entryPoint">
        <anonymous enabled="false"/>
    </http>

    <beans:bean id="entryPoint"
        class="org.springframework.flex.security3.FlexAuthenticationEntryPoint"/>

    <authentication-manager>
       <authentication-provider>
           <user-service>
            <user name="john" password="john" authorities="ROLE_USER" />
            <user name="admin" password="admin" authorities="ROLE_USER, ROLE_ADMIN" />
            <user name="guest" password="guest" authorities="ROLE_GUEST" />
           </user-service>
       </authentication-provider>
    </authentication-manager>

</beans:beans>
```

With a typical Flex application, this approach is preferred to using Spring Security's auto-config setup. Auto-config sets up a number of features that typically are not needed with a Flex application. For instance, auto-config sets up a default `intercept-url` entry that requires authentication for all URL paths within the application. The `FlexAuthenticationEntryPoint` will return a proper AMF error response when such URL's are hit via normal Flex communication processing, but it's invocation should generally be considered a last resort. Most of the time, authentication and authorization processing should happen deeper within the BlazeDS request handling process, where there are opportunities to provide more useful information to the client. (See Securing BlazeDS Channels by Endpoint URL Path for an alternative to `intercept-url` that generates proper AMF responses for the Flex client.) It is recommended to start simple as in this example, and add the additional features as needed.

6.1.2. Enabling the Spring Security filter chain in web.xml

For a typical setup with Spring Security, it is critical to remember to enable the Spring Security filter chain by adding the appropriate entry to web.xml:

```
<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>
```

We will assume the above configuration is in place for the remainder of the examples in this chapter. For additional details on the many options available in configuring and using Spring Security, please refer to that project's documentation.

6.2. Configuring the Spring Security Integration

Spring Security integration is enabled through the `secured` child element of the `message-broker` tag. The simplest possible configuration would be:

```
<flex:message-broker>
    <flex:secured />
</flex:message-broker>
```

This enables the basic security features. A special BlazeDS `LoginCommand` implementation is automatically installed that enables ChannelSet.login and ChannelSet.logout requests to integrate with Spring Security's Authorization mechanisms. Additionally, the special `LoginCommand` enables Spring Security granted authorities to be referenced in BlazeDS XML security constraints. For example, if we were using a traditional BlazeDS remoting destination defined in remoting-config.xml, we could have something like the following:

```
<destination id="productService">
    ...
    <security>
        <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
                <role>ROLE_USER</role>
            </roles>
        </security-constraint>
    </security>
</destination>
```

As you can see, we are referencing the "ROLE_USER" authority from our simple Spring Security setup. The invocation of this remote destination would cause the provided `LoginCommand` to be invoked to both verify that the user is logged in and to check that they have the appropriate role. Violation of either will result in an exception being thrown by Spring Security.

The provided `LoginCommand` mimics as much as possible the functionality of Spring Security's

`AbstractAuthenticationProcessingFilter` including taking the necessary actions to coordinate Spring Security's additional features such as, Remember Me, Session Fixation Protection, and Concurrent Session Management upon commencement of the authentication process. Additionally, it will invoke any configured `LogoutHandlers` as part of the logout process, including both those auto-configured by Spring Security, and any that might be supplied by the application developer.

6.2.1. Using a Custom LoginCommand

While the supplied `LoginCommand` covers many of the most common security configuration scenarios, sometimes it can be useful to provide your own implementation that either extends `SpringSecurityLoginCommand`, while using the Spring container to wire in any necessary dependencies. A custom `LoginCommand` can be supplied via the `login-command` attribute of the `secured` tag as follows:

```
<flex:message-broker>
    <flex:secured login-command="myLoginCommand"/>
</flex:message-broker>

<bean id="myLoginCommand" class="com.foo.app.security.CustomLoginCommand"/>
```

If your custom `LoginCommand` extends `SpringSecurityLoginCommand`, the following properties will be autowired by the container if possible:

- rememberMeServices
- sessionAuthenticationStrategy
- logoutHandlers

6.2.2. Accessing User Details

When using the ChannelSet.login API call from the Flex client with Spring Security integration enabled, the resulting ResponseEvent fired client-side upon successful completion will contain additional information that can be inspected about the current user. The name and authorities will be extracted from the Authentication object and added to the body of the response message. This information, for example, can then be used to conditionally display different portions of the UI based on the user's identity and granted roles:

```
var token:AsyncToken = myChannelSet.login("jeremy","atlanta");
token.addResponder(
    new AsyncResponder(
        function(event:ResultEvent, token:Object = null):void {
            if (event.result.authorities.indexOf("ROLE_ADMIN") >= 0) {
                displayAdminPanel(event.result.name);
            } else {
                displayUserPanel(event.result.name);
            }
        },
        function(event:FaultEvent, token:Object = null):void {
            displayErrorMessage("Login Failed: "+event.fault.faultString);
        }
    )
);
```

6.2.3. Security Exception Translation

Another feature that is automatically installed when the `secured` tag is used is automatic exception translation from any thrown SpringSecurityException to the proper BlazeDS SecurityException. The exceptions are caught and translated at the proper point in the execution chain such that it will result in the proper AMF error message being serialized and sent back to the client.

This is alternative to the normal Spring Security behavior where a filter in the chain catches the exception and sends back a corresponding HTTP status code. The problem with sending back HTTP status codes other than 200 is that this causes the Flex client to throw a generic and rather unhelpful exception, and often the status code can't be determined from the Flex client. Sending back specific AMF error messages instead causes a `FaultEvent` to be thrown client-side that contains the proper security fault code that can then be reasoned on and appropriate action can be taken. This behavior is equivalent to that of the out-of-the-box container-based security mechanisms provided with BlazeDS, so the programming model client-side remains the same.

6.2.4. `secured` Configuration Attributes

The `secured` tag has several additional attributes that allow further customization.

If you are not using Spring Security's default bean ids for the `AuthenticationManager` or `AccessDecisionManager`, you can specify your custom bean references using the corresponding `authentication-manager` and `access-decision-manager` attributes respectively on the `secured` tag.

The configuration of the provided `LoginCommand` can be further controlled via the `secured` tag. The `invalidate-flex-session` attribute controls whether the current Flex session is invalidated when the `logout()` method is called on the `LoginCommand`, and defaults to "true" if not specified. The `per-client-authentication` attribute turns BlazeDS's per-client authentication mode on when true, and defaults to "false" if not specified. Enabling per-client authentication will cause the Security context to no longer be stored in the session between requests and thus will prevent the use of any Spring Security filters that rely on the Security Context being available in the session, but the authentication and authorization integration will otherwise work as expected. (See the BlazeDS docs for further information on the difference between per-session and per-client authentication.)

6.3. Configuring Endpoint and Destination Security

The Spring Security integration allows flexible control over how you secure your application. You can secure BlazeDS endpoints in a manner similar to Spring Security's traditional URL security, and you can secure your Spring services using the many existing object security mechanisms of Spring Security just as if you were writing a traditional web application.

6.3.1. Securing Specific BlazeDS Channels

You can set security constraints on specific BlazeDS channels using the `secured-channel` child element of the `secured` tag. For example:

```
<flex:message-broker>
    <flex:secured>
        <flex:secured-channel channel="my-amf" access="ROLE_USER" />
    </flex:secured>
</flex:message-broker>
```

This results in any request being routed to the "my-amf" channel to require the user to be logged in and to have

the "ROLE_USER" authority. If either of those is violated, a `FaultEvent` will be signaled on the client.

6.3.2. Securing BlazeDS Channels by Endpoint URL Path

You can set security constraints on multiple BlazeDS channels at once using the `secured-endpoint-path` child element of the `secured` tag. In this case you specify a URL pattern to be secured instead of a specific channel id. For example:

```
<flex:message-broker>
    <flex:secured>
        <flex:secured-endpoint-path pattern="**/messagebroker/**" access="ROLE_USER" />
    </flex:secured>
</flex:message-broker>
```

This results in any request being routed to any channel whose endpoint URL contains "/messagebroker/" in the path to require the user to be logged in and to have the "ROLE_USER" authority. If either of those is violated, a `FaultEvent` will be signaled on the client.

6.3.3. Securing Exported Spring Services

Earlier in this chapter you saw an example of using the BlazeDS XML configuration to secure a BlazeDS-managed destination. Since most of the time you will instead be defining destinations by exporting Spring beans using the `remoting-destination`tag, an alternate approach to securing destinations is needed. This is where Spring Security comes in, as all of its existing authorization mechanisms should "just work" when security integration is enabled using the `secured` tag.

> **Note**
>
> When securing destinations as in the examples shown below, it is important to keep in mind that this feature of Spring Security is using Spring AOP. By default, Spring AOP uses JDK dynamic proxies. As such, if you have need to secure methods on a class that are not part of an interface that the class implements, then you will need either factor those methods out into an interface, or configure Spring to use CGLIB-based proxies instead. See the following Spring documentation references for futher guidance:
>
> 1. AOP Proxies
> 2. Proxying Mechanisms

One of the major strengths of Spring Security is the multiple levels of granularity it provides you when securing your Spring services. You can go from securing your entire service layer in one concise statement:

```
<global-method-security>
    <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))" access="ROLE_USER"/>
</global-method-security>
```

to controlling access in a more fine-grained manner at the method layer using XML:

```
<bean id="myService" class="com.mycompany.myapp.MyService">
    <flex:remoting-destination/>
```

```
    <security:intercept-methods>
        <security:protect method="set*" access="ROLE_ADMIN" />
        <security:protect method="get*" access="ROLE_ADMIN,ROLE_USER" />
        <security:protect method="doSomething" access="ROLE_USER" />
    </security:intercept-methods>
</bean>
```

to using a combination of XML and annotations:

```
<security:global-method-security secured-annotations="enabled" jsr250-annotations="enabled"/>
...
<flex:remoting-destination ref="myBankServiceImpl" />
```

```
public interface BankService {

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account readAccount(Long id);

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account[] findAccounts();

    @Secured("ROLE_TELLER")
    public Account post(Account account, double amount);
}
```

to even more fine-grained ACL-based domain object permissions. For more details on the options available, see the Spring Security documentation.

Chapter 7. Integration with the BlazeDS Message Service

7.1. Introduction

The BlazeDS `MessageService` provides a common abstraction for asynchronous messaging style communication that is ultimately agnostic to the messaging protocol being used on the server side. Messages can be passed exclusively between Flex clients, from Java POJOs to subscribed Flex clients, from Flex clients to POJO message handlers, or between just about any combination thereof. Using the Spring-managed `MessageBroker` enables support for using BlazeDS-native AMF messaging, JMS messaging based on Spring's proven and simple JMS abstractions, or messaging using Spring Integration's `MessageChannel` abstraction, all from a common programming model.

The same `Consumer` and `Producer` APIs are used to interact with message destinations from the Flex client, regardless of which underlying messaging protocol is being used on the server. As such, this chapter will focus mainly on setting up and using the various message destination types on the server side. For more details on how to use the `Consumer` and `Producer` APIs in the client, see the BlazeDS documentation.

7.2. Configuring the Message Service

The BlazeDS `MessageService` has traditionally been configured by the inclusion of a messaging-config.xml file in the BlazeDS XML configuration. When using only Spring-managed message destinations, this config file can be left out completely as the inclusion of the `message-broker` tag in your Spring configuration will cause the `MessageService` to be configured with sensible defaults if none already exists at startup time. The end result is essentially equivalent to including the following minimal messaging-config.xml in your BlazeDS configuration:

```
<?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"/>
    </adapters>

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

</service>
```

Note that this assumes that there is already an equivalent application-wide `default-channels` configuration. It is recommended that you set the desired service-specific channels (see example below) if not relying on an application-wide default setup. If no application-wide defaults exist, a best guess will be made by configuring the first available channel from the `MessageBroker` that uses an `AMFEndpoint` with polling enabled as the default for the `MessageService`.

If you wish to have more explicit control over the defaults that will be set on the `MessageService`, you can customize them via the `message-service` child element of the `message-broker` tag. For example:

```
<flex:message-broker>
    <flex:message-service default-adapter-id="my-default-messaging-adapter"
        default-channels="my-polling-amf" />
</flex:message-broker>
```

If you have an existing messaging-config.xml for a legacy BlazeDS application, the `MessageDestinationFactory` will be able to work transparently with it, allowing you to gradually migrate to all Spring-managed messaging destinations.

7.3. Using AMF Message Destinations

For simple messaging needs where there are no requirements for message durability, transaction support, or advanced routing logic, the BlazeDS-native AMF-based message destination is the ideal choice. These destinations can be fully configured in a Spring application context using the `message-destination` XML namespace tag. For example, assuming a Spring-managed `MessageBroker` has been configured, all that is needed to set up a basic destination named "event-bus" with default settings is the following:

```
<flex:message-destination id="event-bus" />
```

This sets up a destination to use the BlazeDS `ActionScriptAdapter` to handle incoming messages. The settings of the destination can be further customized through the various attributes of the `message-destination` tag. Here is an example of the "event-bus" destination configured with most of the available attributes:

```
<flex:message-destination id="event-bus"
    message-broker="messageServiceBroker"
    channels="my-polling-amf, my-secure-amf"
    allow-subtopics="true"
    cluster-message-routing="broadcast"
    message-time-to-live="1"
    send-security-constraint="fooConstraint"
    subscribe-security-constraint="barConstraint"
    subscription-timeout-minutes="1"
    subtopic-separator="/"
    throttle-inbound-max-frequency="500"
    throttle-inbound-policy="ERROR"
    throttle-outbound-max-frequency="500"
    throttle-outbound-policy="IGNORE" />
```

The `message-broker` attribute is a reference to the id of a Spring-managed `MessageBroker`. The `channels` attribute allows you to specify a comma-delimited list of the BlazeDS channels to be used (in order of preference) for this destination. The remaining attributes correspond to the options available via the `network` and `server` settings when configuring a message destination in the BlazeDS-specific XML. Each of these additional attributes is documented in the XSD to provide live code-completion assistance. For additional details on their usage, see the BlazeDS documentation. The `message-destination` tag serves as a base for the `jms-message-destination` and `integration-message-destination` tags so that the same configuration options are available no matter the type of the underlying `MessagingAdapter`.

The only attribute available on the `message-destination` tag that is not available in the JMS and Spring Integration implementations is the `service-adapter` attribute, which can be used to provide a custom `ServiceAdapter` via a reference to a `ManageableComponentFactoryBean`. This can be used to provide integration with additional messaging protocols not directly supported by Spring BlazeDS Integration. See Providing Custom Service Adapters for additional information on using the

`ManageableComponentFactoryBean`.

7.4. Using JMS Message Destinations

For integration with JMS, a special `JmsAdapter` is provided that internally makes use of Spring's `JmsTemplate`, `DestinationResolver`, `DefaultMessageListenerContainer` and other such JMS abstractions for simplified interaction with JMS resources. The `jms-message-destination` XML namespace tag is used to expose JMS destinations as BlazeDS message destinations. The minimal attributes that must be specified are the destination `id` and exactly one of `jms-destination`, `queue-name`, or `topic-name`. A JMS `ConnectionFactory` reference is also required, but does not have to be explicitly specified if there is already one configured in the current application context with an id of "connectionFactory". For example, to configure a BlazeDS message destination named "chatIn" that uses a Spring-managed ActiveMQ JMS queue with a local ActiveMQ installation:

```
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<bean id="chatInQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="queue.flex.chat.in"/>
</bean>

<flex:jms-message-destination id="chatIn" jms-destination="chatInQueue" />
```

Using `queue-name` or `topic-name` will cause the destination to be resolved using a Spring `DestinationResolver`. The `destination-resolver`, `message-converter`, and `transaction-manager` attributes may be used to set custom references to a Spring-managed `DestinationResolver`, `MessageConverter`, or `TransactionManager` respectively.

7.5. Using Spring Integration Message Destinations

For routing messages with Spring Integration, a special `IntegrationAdapter` is provided that is able to send/receive messages via a `MessageChannel`. This is especially useful when you have more complex routing needs for your messages, such as connecting to email or FTP endpoints. The `integration-message-destination` XML namespace tag is used to expose a Spring Integration `MessageChannel` as a BlazeDS message destination. For example, to configure a BlazeDS message destination named "chatOut" that uses a Spring Integration `PublishSubscribeChannel`:

```
<integration:publish-subscribe-channel id="chatOutPubSubChannel" />

<flex:integration-message-destination id="chatOut" message-channel="chatOutPubSubChannel" />
```

7.6. Sending AMF Messages with the MessageTemplate

A convenient `MessageTemplate` helper class is provided that allows you to push messages to any BlazeDS `MessageDestination` from a simple POJO. This provides a nice abstraction over push style messaging that hides away the details of the underlying messaging protocol. Whether using a simple AMF based destination or full-blown JMS, etc., the use of the `MessageTemplate` stays the same. The only thing the `MessageTemplate` requires is a reference to a Spring-managed `MessageBroker`. If the `MessageTemplate` is configured as a Spring

bean, it will try and auto-detect the `MessageBroker` from its application context.

As an example of how the `MessageTemplate` could be used, suppose we have a RESTful travel application that has a Flex-based admin console but also exposes an API over HTTP. To give the admin console a "live" view of the data, we want to push updates to it anytime a new hotel booking is created. Given the following setup in our application context:

```
<flex:message-broker />

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

<flex:message-destination id="bookingUpdates" />
```

and assuming the Flex client is subscribed to the "bookingUpdates" destination, this could be achieved with the following controller code:

```
@Controller
public class BookingController {

    private MessageTemplate template;

    private BookingService bookingService;

    @RequestMapping(value="/bookings", method=RequestMethod.POST)
    public String createBooking(Booking booking){
        booking = bookingService.saveBooking(booking);
        template.send("bookingUpdates", booking);
        return "redirect:/bookings/"+booking.getId();
    }

    @Autowired
    public void setTemplate(MessageTemplate template) {
        this.template = template;
    }

    @Autowired
    public void setBookingService(BookingService bookingService) {
        this.bookingService = bookingService;
    }
}
```

Chapter 8. Building and Running the Spring BlazeDS Integration Samples

8.1. Introduction

Included in the project distribution is a collection of samples called the Spring BlazeDS Integration Test Drive. This samples project is set up to be built with Maven and either run via an embedded Tomcat instance using the Maven Tomcat plugin, or else imported into Eclipse for running via WTP.

8.1.1. Maven Setup

The sample build requires Maven 2.2.0 or greater. Because the build compiles several seperate Flex and AIR projects, it can require setting the MAVEN_OPTS variable for your environment to allocate more memory than the default. The setting we find works well is:

```
MAVEN_OPTS="-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256m"
```

8.1.2. Building and Running the Test Drive

Once your Maven environment is set up correctly, cd to {project distribution root}/samples/spring-flex-testdrive and execute:

```
mvn clean install
```

This will first build all of the individual Flex projects and then finally assemble the 'testdrive' WAR project. After building successfully, if you'd like to run the application from the command line with an embedded Tomcat instance, execute:

```
mvn tomcat:run
```

Once the application has started successfully, you can access the samples walk-through at http://localhost:8080/testdrive/index.html

8.1.3. Using BlazeDS 4 with Maven

Spring BlazeDS Integration 1.6 requires BlazeDS 4, but as of this writing the BlazeDS 4 artifacts have not yet been published to Maven central. The artifacts have been published to the Spring External maven repository (as required for the Spring BlazeDS Integration automated builds). Declaring a dependency on the spring-flex-core jar *should* cause the BlazeDS 4 artifacts to be pulled down transitively from the Spring External repository, but if you encounter any problems, add the following repository entry (either to settings.xml or to your project's pom.xml):

```
<repository>
    <id>spring-external</id>
    <name>Spring repository for third party libraries</name>
    <url>http://repo.spring.io/ext-release-local</url>
</repository>
```

8.1.4. Importing and Running the Test Drive in Eclipse

The individual Test Drive projects are pre-configured to be imported in Eclipse and run with WTP. (There are a number of individual projects, so you may want to consider creating a fresh workspace or at least create a new working set to manage the projects.) We recommend using the free SpringSource Tool Suite to work with the samples so that you can take full advantage of its extensive Spring support, but any version of Eclipse 3.5+ with WTP should work.

To import the samples, select File->Import...->General->Existing Projects into Workspace and navigate to the {project distribution root}/spring-flex-samples/spring-flex-testdrive directory and import all of the projects found.

There is an individual project for each Flex sample, and one WTP project for the 'testdrive' WAR. Once the projects have been imported, you can start the web application by selecting the 'testdrive' project, right-clicking and selecting Run As->Run on Server. The samples have been most thoroughly tested in Tomcat 6.0, but should run in any Servlet 2.4 container that WTP supports. Once the application has started successfully, you can access the samples walk-through at http://localhost:8080/testdrive/index.html (If running on a server other than Tomcat, change the port number as needed.)