Spring BlazeDS Integration Reference Guide

Jeremy Grelle

Version 1.0.0.M1

Published December 2008

Table of Contents

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. What Spring BlazeDS Integration requires to run

Java 5 or higher

Spring 2.5 or higher

Adobe BlazeDS 3.2 or higher

1.3. Where to get support

Professional from-the-source support on Spring BlazeDS Integration is available from SpringSource, the company behind Spring.

2. Configuring 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

The `MessageBrokerFactoryBean` must be configured as a bean in your Spring WebApplicationContext in order to bootstrap the `MessageBroker`. For example, the `MessageBrokerFactoryBean` configured in its simplest form:

```
<!-- Bootstraps and exposes the BlazeDS MessageBroker -->
<bean id="mySpringManagedMessageBroker" class="org.springframework.flex.messaging.MessageBrokerFactoryBean" />
```

The `MessageBrokerFactoryBean` will look for a BlazeDS xml configuration file in its default location: /WEB-INF/flex/services-config.xml. The location may be overridden using the `servicesConfigPath` property. 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:

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

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>
```

Then the `HandlerMapping` in the Spring WebApplicationContext maps all requests to the Spring-managed `MessageBroker` via the `MessageBrokerHandlerAdapter`:

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

<!-- Dispatches requests mapped to a MessageBroker -->
<bean class="org.springframework.flex.messaging.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.e
    <properties>
```

```
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

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

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. In this case, you will want to choose a more flexible mapping strategy, such as mapping `/spring/*` to the `DispatcherServlet`, mapping `/messagebroker/*` to the Spring-managed `MessageBroker`, and modifying any BlazeDS channel definitions accordingly.

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 remoting-config.xml

Though the remoting support in Spring BlazeDS Integration removes the need to define individual remoting destinations in the BlazeDS remoting-config.xml, it currently still requires a minimal setup for the BlazeDS `RemotingService`. The minimal setup for remoting-config.xml is as follows:

```
<?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" defa
    </adapters>

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

</service>
```

The requirement to have this minimal remoting-config.xml will be removed in a future milestone release of Spring BlazeDS Integration.

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

3.3. Using the FlexRemotingServiceExporter

The `FlexRemotingServiceExporter` is 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` with a bean name of `mySpringManagedMessageBroker`, the following `FlexRemotingServiceExporter` definition will expose the service for remoting to the Flex client as a remote service destination named `product`:

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

By default, the remote service destination exposed to the Flex client will use bean name of the `FlexRemotingServiceExporter` as the service id of the destination, but this may be overridden using the `serviceId` property.

The methods that are exposed to be called by the Flex client can be more tightly controlled through used of the `includeMethods` and `excludeMethods` properties of the `FlexRemotingServiceExporter`.

4. Securing BlazeDS Destinations with Spring Security

4.1. Introduction

Coming in Future 1.0 Milestones

5. Integration with the BlazeDS Message Service

5.1. Introduction

Coming in Future 1.0 Milestones

6. Using AMFView in a Spring 3.0 RESTful Architecture

6.1. Introduction

Coming in Future 1.0 Milestones