

**GraniteDS Documentation**

# **Java/JavaFX Reference Guide**

**3.0.0.M2**

by Franck Wolff and William Drai

---

---

---

|                                                                          |           |
|--------------------------------------------------------------------------|-----------|
| Project overview .....                                                   | vii       |
| <b>1. Getting Started .....</b>                                          | <b>1</b>  |
| 1.1. Requirements (Free Tools) .....                                     | 1         |
| 1.2. Hello World, POJO .....                                             | 1         |
| <b>2. Usage Scenarios .....</b>                                          | <b>9</b>  |
| 2.1. Client options .....                                                | 9         |
| 2.2. Server options .....                                                | 9         |
| 2.3. Common server stacks .....                                          | 10        |
| <b>3. Project Setup .....</b>                                            | <b>11</b> |
| 3.1. Server libraries .....                                              | 11        |
| 3.2. Configuring web.xml .....                                           | 11        |
| 3.3. Framework configuration .....                                       | 12        |
| 3.4. Application configuration .....                                     | 13        |
| 3.5. Client libraries .....                                              | 14        |
| 3.6. Developing with Maven .....                                         | 15        |
| <b>4. Remoting and serialization .....</b>                               | <b>17</b> |
| 4.1. Using the Tide API .....                                            | 17        |
| 4.1.1. Basic remoting .....                                              | 18        |
| 4.1.2. Basic remoting with dependency injection .....                    | 19        |
| 4.1.3. Using the TideResponder Interface .....                           | 21        |
| 4.1.4. Simplifying asynchronous interactions .....                       | 22        |
| 4.1.5. Global exception handling .....                                   | 22        |
| 4.2. Mapping between client and server Java objects .....                | 24        |
| 4.3. Externalizers and Java code generation .....                        | 25        |
| 4.3.1. Example of a JPA entity and its corresponding JavaFX bean .....   | 25        |
| 4.3.2. Standard configuration .....                                      | 29        |
| 4.3.3. Autoscans configuration .....                                     | 30        |
| 4.3.4. Built-in externalizers .....                                      | 31        |
| 4.3.5. Built-in client externalizers .....                               | 32        |
| 4.3.6. Custom externalizers .....                                        | 32        |
| 4.3.7. @ExternalizedBean and @ExternalizedProperty .....                 | 33        |
| 4.3.8. Custom class getters .....                                        | 34        |
| 4.3.9. Instantiators .....                                               | 34        |
| 4.4. JPA and lazy initialization .....                                   | 38        |
| 4.4.1. Single-valued associations (proxied or weaved associations) ..... | 38        |
| 4.4.2. Collections (List, Set, Bag, Map) .....                           | 39        |
| 4.5. Securing remote destinations .....                                  | 41        |
| 4.5.1. Configuration .....                                               | 42        |
| 4.5.2. Fine-grained per-destination security .....                       | 44        |
| 4.5.3. Deserialization protection .....                                  | 45        |
| <b>5. JavaFX Code Generator .....</b>                                    | <b>47</b> |
| 5.1. Overview .....                                                      | 47        |
| 5.2. Generated JavaFX Classes .....                                      | 47        |
| 5.3. Java Classes and Corresponding Templates .....                      | 50        |

|                                                                   |            |
|-------------------------------------------------------------------|------------|
| 5.4. Eclipse Plugin .....                                         | 50         |
| 5.5. Ant Task .....                                               | 51         |
| 5.6. Maven Plugin (Flexmojos) .....                               | 55         |
| 5.7. Template Language .....                                      | 58         |
| <b>6. Messaging (Gravity) .....</b>                               | <b>59</b>  |
| 6.1. Example usage with Consumer/Producer .....                   | 59         |
| 6.2. Topics and Selectors .....                                   | 61         |
| 6.3. Common configuration .....                                   | 63         |
| 6.3.1. Supported application servers for Comet/long polling ..... | 65         |
| 6.3.2. Supported application servers for WebSocket .....          | 65         |
| 6.3.3. Advanced configuration .....                               | 66         |
| 6.3.4. Tomcat and JBoss/Tomcat specific configuration tips .....  | 67         |
| 6.4. Integration with JMS .....                                   | 68         |
| 6.5. Using an Embedded ActiveMQ .....                             | 70         |
| 6.6. Server to client publishing .....                            | 72         |
| 6.7. Securing Messaging Destinations .....                        | 75         |
| <b>7. Integration with EJB3 .....</b>                             | <b>79</b>  |
| 7.1. Using the RemoteService API .....                            | 79         |
| 7.1.1. Basic Remoting Example .....                               | 79         |
| 7.1.2. Common configuration .....                                 | 80         |
| 7.1.3. Configuration for Remote EJBs .....                        | 83         |
| 7.1.4. Automatic Configuration of EJB Destinations .....          | 84         |
| 7.1.5. Configuration for Stateful EJBs .....                      | 87         |
| 7.1.6. Security .....                                             | 89         |
| 7.2. Using the Tide API .....                                     | 90         |
| 7.2.1. Configuration .....                                        | 90         |
| 7.2.2. Basic remoting with dependency injection .....             | 94         |
| 7.2.3. Typesafe remoting with dependency injection .....          | 96         |
| <b>8. Integration with Spring .....</b>                           | <b>99</b>  |
| 8.1. Spring MVC setup .....                                       | 99         |
| 8.2. Using the RemoteService API .....                            | 100        |
| 8.2.1. Basic remoting example .....                               | 100        |
| 8.2.2. Configuration with a MVC setup .....                       | 102        |
| 8.2.3. Default configuration .....                                | 104        |
| 8.2.4. Automatic configuration of destinations .....              | 105        |
| 8.2.5. Integration with Spring Security .....                     | 106        |
| 8.3. Using the Tide API .....                                     | 108        |
| 8.3.1. Configuration with a MVC setup .....                       | 108        |
| 8.3.2. Default configuration .....                                | 109        |
| 8.3.3. Basic remoting with dependency injection .....             | 113        |
| 8.3.4. Typesafe remoting with dependency injection .....          | 114        |
| 8.3.5. Integration with Spring Security .....                     | 115        |
| 8.4. Messaging with Spring (Gravity) .....                        | 118        |
| <b>9. Integration with CDI .....</b>                              | <b>121</b> |

---

|                                                             |            |
|-------------------------------------------------------------|------------|
| 9.1. Configuration with Servlet 3 .....                     | 121        |
| 9.2. Default Configuration .....                            | 123        |
| 9.3. Using the Tide API .....                               | 124        |
| 9.3.1. Basic remoting with dependency injection .....       | 124        |
| 9.3.2. Typesafe remoting with dependency injection .....    | 125        |
| 9.3.3. Integration with Events .....                        | 126        |
| 9.4. Messaging with CDI (Gravity) .....                     | 127        |
| <b>10. Client-Side Validation API (JSR 303) .....</b>       | <b>131</b> |
| 10.1. Integration with code generation tools (Gfx) .....    | 131        |
| 10.2. Using the FormValidator class .....                   | 133        |
| <b>11. Data Management .....</b>                            | <b>137</b> |
| 11.1. JPA and Managed Entities .....                        | 137        |
| 11.2. Transparent lazy loading .....                        | 140        |
| 11.3. Dirty checking and conflict handling .....            | 141        |
| 11.4. Data validation .....                                 | 145        |
| 11.5. Data paging .....                                     | 145        |
| 11.6. Data push .....                                       | 150        |
| <b>12. Extensibility .....</b>                              | <b>161</b> |
| 12.1. Writing a security service .....                      | 161        |
| 12.2. Custom exception handlers .....                       | 162        |
| 12.3. Server message interceptors .....                     | 163        |
| 12.4. Custom AMF3 (De)Serializers (Advanced use only) ..... | 164        |
| 12.5. ServiceInvocationListener (Advanced use only) .....   | 166        |
| <b>13. Configuration Reference .....</b>                    | <b>167</b> |
| 13.1. Framework Configuration .....                         | 167        |
| 13.2. Application Configuration .....                       | 169        |
| 13.2.1. Factories .....                                     | 169        |
| 13.2.2. Remoting destinations .....                         | 170        |
| 13.2.3. Messaging destinations .....                        | 170        |
| <b>14. Appendix .....</b>                                   | <b>173</b> |
| 14.1. GraniteDS config DTD .....                            | 173        |
| 14.2. GraniteDS Spring/Seam Configuration XSD .....         | 173        |
| 14.3. Release notes .....                                   | 174        |



---

## Project overview

*Granite Data Services* (GraniteDS) is a comprehensive development and integration platform for building RIA applications with a Java EE backend and a Java/JavaFX frontend. The framework is completely open source and released under the LGPL v2 license.

GDS has been designed to be lightweight, robust, fast, and highly extensible.

The main features of GraniteDS are :

- An implementation of the [Adobe AMF remoting](#) protocol and of the AMF3 data format, with out-of-the-box adapters for all usual Java frameworks.
- An implementation of a [messaging](#) framework based supporting Comet and Websocket transports which can connect to JMS servers.
- A [data management framework](#) which simplifies the handling and synchronization of persistent data through client and server applications.

**Who we are.** The core development team is Franck Wolff and William Draï, two engineers from Granite Data Services. Many people have contributed to GraniteDS by giving ideas, patches or new features. If you feel you should be listed below, please email [me](mailto:me) [<http://www.graniteds.org/confluence/display/~fwolff>].

### Spring integration.

- Igor SAZHNEV: Initial Spring service factory implementation and Java Enum externalizer.
- Francisco PEREDO: Acegi security support and Spring/Acegi/EJB 3 sample application.
- Sebastien DELEUZE (aka Bouiaw): Spring 2 security service.

### Seam 2 Integration.

- Cameron INGRAM, Venkat DANDA and Stuart ROBERTSON: Seam integration implementation and Tide framework.

### Guice/Warp integration.

- Matt GIACOMI: Initial Guice/Warp integration implementation and sample application.

### Grails plugin.

- Ford GUO: major improvements of the GDS/Grails plugin.

### OSGi integration.

- Zhang BIN: GDS/OSGi integration.

### **DataNucleus Integration.**

- Stephen MORE: initial DataNucleus engine support.

### **Web MXML/ActionScript3 compiler.**

- Sebastien DELEUZE (aka Bouiaw) and Marvin FROEDER (aka VELO): A servlet-based compiler that compiles your MXML and ActionScript3 sources on the fly.

### **Maven integration.**

- Rafique ANWAR: Initial Maven POM files and deploy script (java.net).
- Edward YAKOP: Improved Maven POM files and deploy script (Sonatype).

### **ActionScript3 code generation.**

- Francesco FARAONE and Saverio TRIONE: Gas3 extension with typed as3 client proxies generation.

### **Documentation.**

- Michael SLINN: Oversight.
- Elizabeth Claire MYERS: Proofreading/editing.

### **Other contributions.**

- Francesco FARAONE: HibernateExternalizer Map support.
- Marcelo SCHROEDER: Service exception handler.
- Sebastien GUIMONT: Initial Java Enum support in Gas3.
- Pedro GONCALVES: Improved service method finder for generics.

# Getting Started

This section introduces:

- GraniteDS software requirements.
- The complete setup of a basic "Hello, world" GDS project using Java, Eclipse, and Tomcat.

## 1.1. Requirements (Free Tools)

You need at least the following four free development tools:

- Java 6+ (5+ supported): [Download Sun JDK](http://java.sun.com/javase/downloads/index.jsp) [http://java.sun.com/javase/downloads/index.jsp] and install it.
- Eclipse 3.2+: [Download Eclipse 3.2+](http://www.eclipse.org/downloads/) [http://www.eclipse.org/downloads/] and unzip it somewhere (NOTE: It is always a good idea to avoid paths with whitespace). You may, of course, use another IDE, but you will not be able to use Eclipse specific plugins and example projects.

## 1.2. Hello World, POJO

This section will guide you through the setting up of a very basic GraniteDS project deployed in Tomcat and a Java command line client. Expected result is a typical "Hello, world" application.

The client program will pass its argument to the remote service and display the result which should be a string:

```
"Hello " + <argument> + "!"
```

In order to create, build, and deploy this sample application you need these free tools:

- Java 6+ (6+ working): Download [Sun JDK](http://java.sun.com/javase/downloads/index.jsp) [http://java.sun.com/javase/downloads/index.jsp] and install it.
- Eclipse 3.5+: Download [Eclipse](http://www.eclipse.org/downloads/) [http://www.eclipse.org/downloads/] and unzip it somewhere.
- Tomcat 7+: Download [Tomcat](http://tomcat.apache.org/download-70.cgi/) [http://tomcat.apache.org/download-70.cgi/] and unzip it somewhere. For example, /apache-tomcat-7.0.29 (for Windows users: C:\apache-tomcat-7.0.29).
- *granite.jar*: You may take it from any of the GraniteDS sample applications or from GraniteDS source distribution in the build folder. Download it [here](http://www.graniteds.org/confluence/display/DOWNLOAD) [http://www.graniteds.org/confluence/display/DOWNLOAD].

- *granite-client.jar* and *granite-java-client.jar*: You may take it from any of the GraniteDS sample applications or from GraniteDS source distribution in the `build` folder. Download it [here](http://www.graniteds.org/confluence/display/DOWNLOAD) [http://www.graniteds.org/confluence/display/DOWNLOAD].

Creation of the project in Eclipse:

Start Eclipse and create a new Java project named `helloworld`. You may just type in `helloworld` for `Project name` and accept all other default settings.

We are now going to create a new POJO service named `HelloWorldService`. Right-click on the `java` source folder and select *New / Class*, enter `org.test` for `Package` and `HelloWorldService` for `Name` in the following dialog, and then click on the *Finish* button. In the Java source file editor, modify the code so it is just as follows:

```
package org.test;

public class HelloWorldService {

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```

Next we have to create the GraniteDS configuration file `services-config.xml` and the web application `web.xml` at the root of the project.

Copy and paste the following code into these files:

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

    <services>
        <service
            id="granite-service"
            class="flex.messaging.services.RemotingService"
            messageTypes="flex.messaging.messages.RemotingMessage">
            <destination id="helloWorldService">
                <properties>
                    <scope>application</scope>
                    <source>org.test.HelloWorldService</source>
                </properties>
            </destination>
        </service>
    </services>
</services-config>
```

```

    </destination>
  </service>
</services>
</services-config>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">

  <!-- general information about this web application -->
  <display-name>Hello World</display-name>
  <description>Hello World Sample Application</description>

  <!-- read services-config.xml file at web application startup -->
  <listener>
    <listener-class>org.granite.config.GraniteConfigListener</listener-class>
  </listener>

  <!-- handle AMF requests ([de]serialization) -->
  <filter>
    <filter-name>AMFMessageFilter</filter-name>
    <filter-class>org.granite.messaging.webapp.AMFMessageFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>AMFMessageFilter</filter-name>
    <url-pattern>/graniteamf/*</url-pattern>
  </filter-mapping>

  <!-- handle AMF requests (execution) -->
  <servlet>
    <servlet-name>AMFMessageServlet</servlet-name>
    <servlet-class>org.granite.messaging.webapp.AMFMessageServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>AMFMessageServlet</servlet-name>
    <url-pattern>/graniteamf/*</url-pattern>
  </servlet-mapping>

```

```
</web-app>
```

Next we have to build and deploy the server application:

Create a folder named `lib` at the root of the project and put `granite.jar` in this folder. Create a new file named `build.xml` at the root of the project and copy/paste the following content into it; you may have to modify `TOMCAT_HOME` to reflect your environment:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="hello-world" default="deploy">

  <!-- Modify TOMCAT_HOME properties to reflect your environment -->
  <property name="TOMCAT_HOME" value="/apache-tomcat-7.0.29"/>

  <!-- Build a war suitable for Tomcat (and other) -->
  <target name="war">
    <mkdir dir="build"/>
    <war destfile="build/helloworld.war" webxml="web.xml">
      <zipfileset file="services-config.xml" prefix="WEB-INF/flex" />
      <lib dir="lib"/>
      <classes dir="bin"/>
    </war>
  </target>

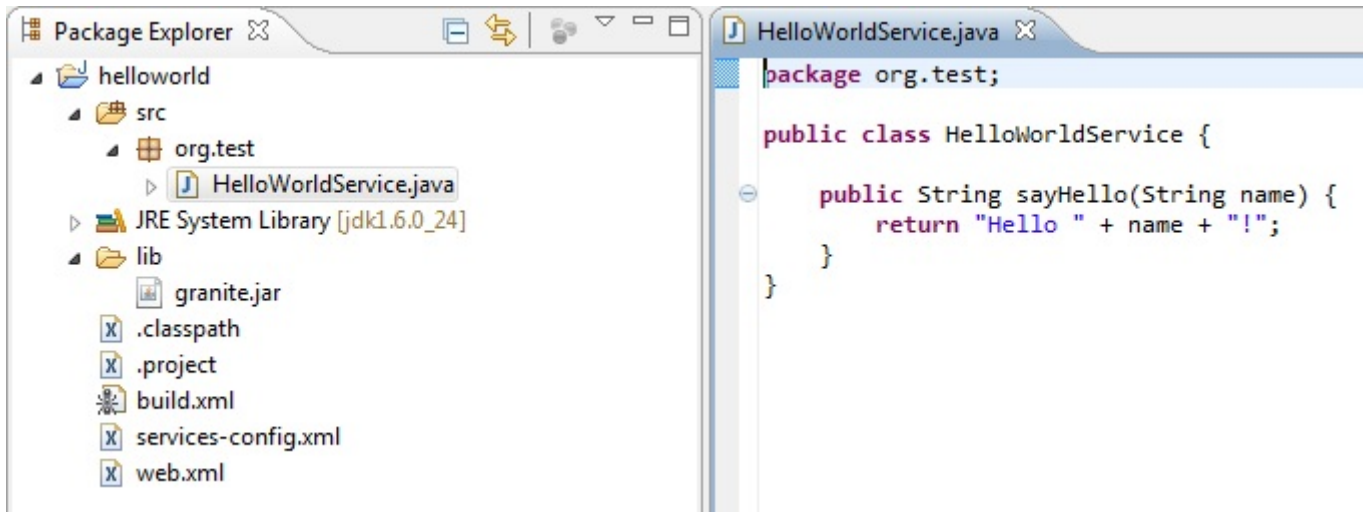
  <!-- Deploy the war in Tomcat -->
  <target name="deploy" depends="war">
    <copy todir="${TOMCAT_HOME}/webapps" file="build/helloworld.war"/>
  </target>

</project>
```

You may now right-click on the `build.xml` file and select *Run As / Ant Build*. This will launch the build process, create a WAR (Web Archive), and copy it into your Tomcat webapps directory.

Then start Tomcat. Go to the directory `bin` just under your Tomcat installation directory, `/apache-tomcat-7.0.29/bin` for example, and double-click on `startup.bat`, or `startup.sh` for Unix/Mac users. After a short while, you should see in the console that Tomcat has started.

You should now see something like the following picture under Eclipse:



Now let create the Java client code, for this example we are simply going to create a command line application but we could use any Java view technology, such as Swing, JavaFX or SWT.

Create a new Java project named helloworld-client. Create a new class directly in this new folder and name it HelloWorldClient in the package org.test.client by right-clicking on the src folder and selecting *New / Class*. In the editor, type in the following code:

```
package org.test.client;

import java.net.URI;
import java.util.concurrent.TimeUnit;

import org.granite.client.messaging.RemoteService;
import org.granite.client.messaging.ResultFaultIssuesResponseListener;
import org.granite.client.messaging.channel.amf.AMFRemotingChannel;
import org.granite.client.messaging.events.FaultEvent;
import org.granite.client.messaging.events.IssueEvent;
import org.granite.client.messaging.events.ResultEvent;
import org.granite.client.messaging.transport.apache.ApacheAsyncTransport;

public class HelloWorldClient {

    public static void main(String[] args) throws Exception {
        ApacheAsyncTransport transport = new ApacheAsyncTransport();
        transport.start();
        AMFRemotingChannel channel = new AMFRemotingChannel(transport,
            "graniteamf", new URI("http://localhost:8080/helloworld/graniteamf/amf.txt"));
        RemoteService service = new RemoteService(channel, "helloWorldService");
```

```
service.newInvocation("sayHello", args[0]).setTimeToLive(5, TimeUnit.SECONDS)
    .addListener(new ResultFaultIssuesResponseListener() {

        @Override
        public void onResult(ResultEvent event) {
            System.out.println("Result: " + event.getResult());
        }

        @Override
        public void onFault(FaultEvent event) {
            System.err.println("Fault: " + event.toString());
        }

        @Override
        public void onIssue(IssueEvent event) {
            System.err.println("Issue: " + event.toString());
        }
    }).invoke();
}
```

You will also need to add a few libraries in a `lib` folder and add them to the build path of the project with *Right Click/Build Path/Add to Builder Path*:

- `httpclient-4.2.1.jar`
- `httpcore-4.2.1.jar`
- `httpcore-nio-4.2.1.jar`
- `httpasyncclient-4.0-beta2-SNAPSHOT.jar`
- `httpclient-4.2.1.jar`
- `granite-client.jar`
- `granite-java-client.jar`

You may now run the Java application in Eclipse by right-clicking the class `HelloWorldClient` and *Run As.../Java Application*. The result should appear in the Eclipse console. You can test different results by changing the run arguments in the Eclipse Run configuration for the application.

Here are some highlights on some parts of the code and configuration:

```
public String HelloWorldService.sayHello(String name)
```

The HelloWorldService is a simple Java service which declares a method sayHello() that takes a String argument and returns another String.

```
<destination id="helloWorldService">
  <channel ref="my-graniteamf"/>
  <scope>application</scope>
  <source>org.test.HelloWorldService</source>
</destination>
```

This part of the services-config.xml defines a mapping between a destination name and the service class and its scope. This is a basic declaration for an application scoped bean that will be created by GraniteDS itself but there are other kinds of configurations that give access to beans managed by an existing container such as Spring, or that use annotations to declare the remoting-enabled classes.

```
<url-pattern>/graniteamf/*</url-pattern>
```

This part of web.xml defines the mapping between the target url and the GraniteDS servlet. Other kinds of configuration are also possible which use a Spring MVC dispatcher servlet or use Servlet 3 features to automatically initialize the GraniteDS servlet. /graniteamf/\* is the default and recommended url mapping for GraniteDS, but any other can work.

```
ApacheAsyncTransport transport = new ApacheAsyncTransport();
transport.start();
AMFRemotingChannel channel = new AMFRemotingChannel(transport, "my-graniteamf",
    new URI("http://localhost:8080/helloworld/graniteamf/amf.txt"));
RemoteService srv = new RemoteService(channel, "helloWorldService");
```

This is the initialization part of the GraniteDS Java client. It requires creating a transport (here the default transport based on the Apache asynchronous HTTP client), a remoting channel and a RemoteService whose target destination matches the destination we declared earlier in the server configuration.

```
srv.newInvocation("sayHello", args[0]).setTimeToLive(5, TimeUnit.SECONDS)
    .addListener(new ResultFaultIssuesResponseListener() {

    @Override
    public void onResult(ResultEvent event) {
        System.out.println("Result: " + event.getResult());
    }

    @Override
    public void onFault(FaultEvent event) {
        System.err.println("Fault: " + event.toString());
    }

    @Override
    public void onIssue(IssueEvent event) {
        System.err.println("Issue: " + event.toString());
    }
}).invoke();
```

This is the main client part where the `RemoteService` triggers a server request that will call the `sayHello()` method with the first argument of the main method: `srv.sayHello(args[0])`.

The result of this call will be displayed, when available, in the console output in the asynchronous result handler of the remote call.

## Usage Scenarios

The main value of GraniteDS is to provide integration with other frameworks, both client and server side, so there really are lots of different possible combinations of deployment types and usage scenarios. This chapter will describe various options, and common combinations of technologies.

### 2.1. Client options

There are two main use cases for the GraniteDS Java client. The first is to help testing of GraniteDS-enabled services outside of a Flex environment, the other is for building rich client applications with any Java view technology (Swing, SWT, JavaFX).

Note that the GraniteDS client library provides extensive support for most advanced features of JavaFX 2.1+.

There are also two main choices for the client/server API:

- Use the low-level `RemoteService` API. This is the recommended API if you want to test existing GraniteDS services from a Java client, or do not need advanced features.
- Use the *Tide* remoting API with the GraniteDS/Tide server framework integration (supporting Spring, EJB3 and CDI). It provides the most advanced features and greatly simplifies asynchronous handling and client data management. It should be preferred for new projects, JavaFX clients or more generally when you want to benefit from all of GraniteDS functionalities.

The Tide remoting API and data management framework include a very minimalistic client application framework (event bus, component container...) and may be integrated with more robust frameworks such as Spring. There is a built-in Spring client integration, and a SPI can be implemented to integrate with other frameworks (Java client support for CDI may be added later).

### 2.2. Server options

On the server there are mostly two options :

- If you use the `RemoteService` API, just choose the GraniteDS service factory depending on your server framework. This will additionally bring you the GraniteDS support for externalization of lazily loaded JPA entities/collections, and support for scalable messaging through Gravity.
- If you use the Tide API, choose the GraniteDS/Tide service factory for your server framework. This will bring the full feature set of Tide data management and further integration with data push through Gravity. The Tide server integration also provides more specific features depending on the server framework, for example complete support for Spring security or integration with CDI events.

### 2.3. Common server stacks

This section describes some classic technology stacks used with Java applications and GraniteDS.

**Spring/Hibernate on Tomcat 6+ or Jetty 6+.** This is one of the most common use cases and allows for easy development and deployment. You can furthermore benefit from the extensive support for serialization of Java objects and JPA detached objects, and of NIO/APR asynchronous support of Tomcat 6.0.18+ or Jetty 6 continuations.

**EJB3/Hibernate on JBoss 4/5.** This is another common use case and it provides roughly the same features than Spring/JPA. The main difference is that it requires a full EE container supporting EJB 3.

**Tide/Spring/Hibernate on Tomcat 6+ or Jetty 6+.** This is an extension of the first case, with the additional use of the Tide remoting and data management API on the client. This will enable the most advanced features such as data paging, transparent lazy-loading of collections, real-time data synchronization... Tide also provides advanced client-side support for Spring Security authorization that for example allow to easily hide/disable buttons for unauthorized actions. This is currently the most popular technology stack.

**Tide/EJB3/Hibernate on JBoss 4/5 or Tide/EJB3/EclipseLink on GlassFish v3.** It's also similar to the previous case, but using EJB 3 instead of Spring.

**Tide/CDI/JPA2/Java EE 6 on JBoss 6/7 or GlassFish 3.** Well this is not really a "common" stack but at least it is a fully Java EE standard6. If you are on a Java EE 6 compliant application server and can live without Spring, it is definitely the best option.

## Project Setup

GraniteDS consists in a set of client libraries and a set of server libraries. It is designed to be deployed in a Java application server and packaged in a standard Java Web application, either as a WAR file or as an EAR file. The configuration of a GraniteDS project will generally involve the following steps :

1. Add the GraniteDS jars to the `WEB-INF/lib` folder of the WAR file or the `lib` folder of the EAR file
2. Add the GraniteDS listener, servlets and filters in the standard `WEB-INF/web.xml` configuration file
3. Define the internal configuration of GraniteDS in the `WEB-INF/granite/granite-config.xml` file
4. Define the application configuration of GraniteDS (remoting destinations, messaging topics...) in the `WEB-INF/flex/services-config.xml`
5. Build you Java client project with the GraniteDS libraries

Depending on which framework and application server you use on the server (Spring, Seam...) and on the client, some of these steps may be completely omitted, or implemented differently. For example, when using the Spring framework on the server, almost all the configuration can be defined in the standard Spring context instead of the `granite-config.xml` and `services-config.xml` files. GraniteDS tries to be as transparent and integrated as possible with the application environment, however it can be useful to know how things work at the lower level if you have specific requirements.

### 3.1. Server libraries

The GraniteDS jars are available from the `build` folder of the distribution. You will always need `granite.jar`. Additionally you will have to include the jar corresponding to your server framework (`granite-spring.jar` for Spring for example), the jar for your JPA provider (`granite-hibernate.jar` for Hibernate) and the `granite-beanvalidation.jar` if you want to benefit from the integration with the Bean Validation API on the server.

### 3.2. Configuring web.xml

At the most basic level, GraniteDS is implemented as a servlet (in fact a servlet and a filter) and thus has to be configured in `web.xml`. Here is a typical code snippet that maps the GraniteDS AMF servlet to `/graniteamf/*`. Of course it's possible to define a different URL mapping if required. It is also highly recommended to also add the configuration listener that will release resources on application undeployment.

```
<listener>
  <listener-class>org.granite.config.GraniteConfigListener</listener-class>
</listener>

<filter>
  <filter-name>AMFMessageFilter</filter-name>
  <filter-class>org.granite.messaging.webapp.AMFMessageFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>AMFMessageFilter</filter-name>
  <url-pattern>/graniteamf/*</url-pattern>
</filter-mapping>

<servlet>
  <servlet-name>AMFMessageServlet</servlet-name>
  <servlet-class>org.granite.messaging.webapp.AMFMessageServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>AMFMessageServlet</servlet-name>
  <url-pattern>/graniteamf/*</url-pattern>
</servlet-mapping>
```

### 3.3. Framework configuration

The configuration of the various GraniteDS parts is done in the file `WEB-INF/granite/granite-config.xml`. There are many options that can be defined here, you can refer to the [configuration reference](#).

As a starting point, you can create an empty file :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config/>
```

Or much easier a configuration that will use class scanning to determine the default setup.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config scan="true"/>
```

### 3.4. Application configuration

The last thing to define on the server is the application configuration in `WEB-INF/flex/services-config.xml`. This is for example the place where you will define which elements of your application you will expose to GraniteDS remoting, or the topic for messaging. You can refer to the [configuration reference](#) for more details.

For example a simple configuration for an EJB 3 service would look like :

```
<services-config>
  <services>
    <service id="granite-service"
      class="flex.messaging.services.RemotingService"
      messageTypes="flex.messaging.messages.RemotingMessage">

      <destination id="example">
        <channels>
          <channel ref="graniteamf"/>
        </channels>
        <properties>
          <factory>ejbFactory</factory>
        </properties>
      </destination>
    </service>
  </services>

  <factories>
    <factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
      <properties>
        <lookup>myapp/{capitalized.destination.id}ServiceBean/local</lookup>
      </properties>
    </factory>
  </factories>
```

```
<channels>
  <channel-definition id="graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>
```

This configuration file declares 3 different things, let's list them in the reverse order :

- Channel endpoint : this defines the uri on which the remote service can be accessed through GraniteDS remoting. This should match the servlet url mapping defined previously in `web.xml`.
- Service factories : here the configuration defines an EJB 3 factory, meaning that destinations using this factory will route incoming remote calls to EJB 3. GraniteDS provides factories for all popular server frameworks. Most factories require specific properties, here for example the JNDI format for EJB lookup.
- Service/destinations : this section defines a remoting service (described by its class and message type) and one destination interpreted as an EJB 3 as indicated by the factory property.

Depending on the kind of framework integration that is used, the `services-config.xml` file may not be necessary and can be omitted. With Spring and Seam for example, everything can be defined in the respective framework configuration files instead of `services-config.xml`.

### 3.5. Client libraries

GraniteDS comes with 3 client jar libraries. First `granite-client.jar`, a stripped down version of the core `granite.jar` that includes the minimal core of GraniteDS. Then `granite-java-client.jar` that includes the Java client library, and finally `granite-javafx-client.jar` that contains the specific integration for JavaFX.

The GraniteDS client also depends on the small class scanning library `extcos`. For remoting and Comet, the GraniteDS client requires the Apache Asynchronous HTTP client, and for WebSocket, the Jetty WebSocket client. All these jars can be found in the `libs` folder of the Java client distribution.

You simply have to add the necessary GraniteDS jars and dependencies to your application classpath.

## 3.6. Developing with Maven

Maven is a popular build tool. Though GraniteDS is not itself built with Maven, its artifacts are available in the Maven central repository and can thus be easily added as dependencies to any Maven project.

The Java dependencies for the server application are in the group `org.graniteds`.

```
<dependency>
  <groupId>org.graniteds</groupId>
  <artifactId>granite-core</artifactId>
  <version>${graniteds.version}</version>
  <type>jar</type>
</dependency>

<dependency>
  <groupId>org.graniteds</groupId>
  <artifactId>granite-hibernate</artifactId>
  <version>${graniteds.version}</version>
  <type>jar</type>
</dependency>

...
```

The dependencies for the Java client application are as follows:

```
<dependency>
  <groupId>org.graniteds</groupId>
  <artifactId>granite-client</artifactId>
  <version>${graniteds.version}</version>
  <type>jar</type>
</dependency>

<dependency>
  <groupId>org.graniteds</groupId>
  <artifactId>granite-java-client</artifactId>
  <version>${graniteds.version}</version>
  <type>jar</type>
</dependency>
```

```
<!-- Only for JavaFX integration -->
<dependency>
  <groupId>org.graniteds</groupId>
  <artifactId>granite-javafx-client</artifactId>
  <version>${graniteds.version}</version>
  <type>jar</type>
</dependency>

<!-- Default dependencies -->
<dependency>
  <groupId>net.sf.extcos</groupId>
  <artifactId>extcos</artifactId>
  <version>0.3b</version>
  <type>jar</type>
</dependency>

<!-- Apache HTTP client dependencies (remoting, Comet) -->
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpasyncclient</artifactId>
  <version>4.0-beta1</version>
  <type>jar</type>
</dependency>

<!-- Jetty WebSocket client dependencies (WebSocket) -->
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-client</artifactId>
  <version>8.1.5.v20120716</version>
  <type>jar</type>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-websocket</artifactId>
  <version>8.1.5.v20120716</version>
  <type>jar</type>
</dependency>
```

## Remoting and serialization

Data serialization between a client application and a Java EE server may use different kinds of transfer encodings, including XML, JSON, Java serialization, or various other serialization frameworks. GraniteDS provides an implementation of the Adobe AMF3 (ActionScript Message Format) binary encoding which is very compact, fast and efficient. Other formats may be added later but AMF3 is a really easy-to-use and performant format.

The AMF3 format allows for serialization of strongly typed objects. GraniteDS adds the concept of externalization to transform the serialized objects before and after they are serialized. This allows for example to serialize JPA entities without triggering initialization of all lazy properties.

When building a JavaFX client, you can then easily deserialize these entities to a properly JavaFX-bindable bean having the same properties. This way the client and server parts of the application are cleanly separated, the JavaFX bean does not have any dependency (even internal runtime) on the JPA provider and the JPA entity having no dependency on the JavaFX binding API.

**The AMF3 format.** AMF3 is a very compact binary format for data serialization/deserialization and remote method invocation. A key feature of this format is that it preserves the entire graph of your data without duplicating identical objects (contrary to JSON for example). For example, if A1 and A2 contain a reference to the same B1, the serialization of A1 and A2 does not duplicate B1. The target client VM will contain exactly the same data graph with only one B1 referenced by one A1 and one A2. Furthermore, there is no risk of infinite recursion if the data graph contains circular references. For example, if B1 contains the set of A# that references B1. AMF3 messages are sent as a part of a AMF0 envelope and body. GraniteDS implements an AMF3 serializer/deserializer and relies on some code borrowed from the [OpenAMF](http://sourceforge.net/projects/openamf/) [http://sourceforge.net/projects/openamf/] project for AMF0 serialization/deserialization. The AMF0 and AMF3 specifications are now public. You may download them [here](http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf) [http://download.macromedia.com/pub/labs/amf/amf3\_spec\_121207.pdf]. You will need a Macromedia or Adobe account.

### 4.1. Using the Tide API

The Tide remoting API is an alternative to the low-level RemoteService API that simplifies the handling of asynchronicity and brings much more features that will be described in the next chapters.



#### Note

This section describes the usage of the Tide API with a standard AMF provider. When the Tide API is used in conjunction with GraniteDS and Tide-enabled server framework adapters, there are some specificities that are described in the chapters concerning each framework integration ([EJB3](#), [Spring](#), [CDI](#)).

### 4.1.1. Basic remoting

Let's see the same hello example with Tide. Note the usage of the Tide context object which represents the client application container.

```
public class HelloExample {

    public static void main(String[] args) {

        ContextManager contextManager = new SimpleContextManager(new DefaultPlatform());
        Context context = contextManager.getContext();

        ServerSession serverSession = new ServerSession("spring", "/myapp", "localhost", 8080);
        context.set(serverSession);
        serverSession.start();

        Component helloService = new ComponentImpl(serverSession);
        context.set("helloService", helloService);

        // Asynchronous call using handlers
        helloService.call("sayHello", "Barack", new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }

            @Override
            public void fault(TideFaultEvent fault) {
                System.err.println("Fault: " + fault.getFault());
            }
        });

        // Synchronous wait of Future result
        Future<String> futureResult = helloService.call("sayHello", "Barack");
        String result = futureResult.get();
        System.out.println("Sync result: " + result);
    }
}
```

This is a bit different than the RemoteService API. It looks like a mostly cosmetic changes, but there are many internal things that differ.

The core of the Tide framework is the context which contains the various elements of the application. Here we create a simple `ContextManager` which implements a very minimalistic built-in application container. For more demanding environment, we recommend using the `SpringContextManager` which will use a Spring application container.

The `Platform` SPI is a simple interface that allows to integrate the Tide context with the client UI framework. For example, JavaFX requires that all graphic operations are executed in the main UI thread. The JavaFX platform implementation will ensure that the asynchronous result handlers of remote calls will be executed in the UI thread so you can do whatever UI operation you need using the received data.

The `ServerSession` encapsulates all communication between the client application and the remote services for a particular server endpoint. Note that here it has to be "attached" manually to the Tide context with `context.set()`. In a Spring environment, it would just have to be declared as a Spring bean.

Finally the `Component` instance represents a client proxy to the actual remote service. The method `call` executes the remote call and returns a `Future` object which can be used to get the result. It is also possible to provide a last argument to the method `call` which can implement `TideResponder` and `result, fault`. Here we use an untyped `ComponentImpl` implementation but it's also possible to generate typesafe client proxies from the service interfaces.

#### 4.1.2. Basic remoting with dependency injection

The previous example was a bit simplistic, and in more realistic applications you might want to use the client proxies from some controller class instead of the main application (!). For a more 'enterprisy' usage, we might configure a Spring container on the client application.

```
package com.myapp.client;

@Configuration
public class Config {

    @Bean
    public SpringEventBus eventBus() {
        return new SpringEventBus();
    }

    @Bean
    public SpringContextManager contextManager(SpringEventBus eventBus) {
        return new SpringContextManager(new JavaFXPlatform(eventBus));
    }

    @Bean(initMethod="start", destroyMethod="stop")
    public ServerSession serverSession() throws Exception {
```

```
        return new ServerSession("spring", "/test", "localhost", 8080);
    }

    @Bean
    public Component helloService(ServerSession serverSession) {
        return new ComponentImpl(serverSession);
    }

    @Bean(initMethod="start")
    public App app() {
        return new App();
    }
}
```

```
package com.myapp.client;

public class App {

    public static void main(String[] args) {
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext();
        applicationContext.scan("com.myapp.client");
        applicationContext.refresh();
        applicationContext.registerShutdownHook();
        applicationContext.start();
    }

    @Inject @Qualifier("helloService")
    private Component helloService;

    public void start() {
        helloService.call("sayHello", "Barack", new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }

            @Override
            public void fault(TideFaultEvent fault) {
                System.err.println("Fault: " + fault.getFault());
            }
        });
    }
};
```

```

    }
}

```

Here we use the Spring 3.1 Java configuration mechanism, but you could also do all this in XML or any other Spring configuration style. The important things here are that we declared two components of types `EventBus` and `ContextManager`, the `ServerSession` and a Component as Spring beans. Once everything is properly wired together, you can simply inject the client proxies in whatever bean you want to execute the remote calls.

### 4.1.3. Using the TideResponder Interface

In some cases, you may need to pass some value to the result/fault handler to be able to distinguish different calls on the same method. You can then override the default `TideResponder` implementation and store a token object:

```

public static class HelloResponder implements TideResponder {

    private final String token;

    public HelloResponder(String token) {
        this.token = token;
    }

    @Override
    public void result(TideResultEvent event) {
        System.out.println("Result for " + token + ": " + event.getResult());
    }

    @Override
    public void fault(TideFaultEvent event) {
        System.err.println("Fault for " + token + ": " + event.getFault());
    }
}

public void call() {
    helloService.call("sayHello", "Barack", new HelloResponder("firstCall"));
    helloService.call("sayHello", "Barack", new HelloResponder("secondCall"));
}

```

In this case, there will be two outputs for each token. Note that as everything is asynchronous, the order of the results is undefined.

### 4.1.4. Simplifying asynchronous interactions

The `TideMergeResponder` interface is an extension of `TideResponder` that makes possible to provide a return object that will be merged with the server result. It helps working with the asynchronous nature of remoting by limiting the need for result handlers.

```
private List<Product> products = new ArrayList<Product>();

public function call():void {
    productService.findAllProducts(new TideMergeResponder<List<Product>>() {
        @Override
        public void result(TideResultEvent<List<Product>> event) {
            System.out.println("Result was merged: " + (event.getResult() == products));
        }

        @Override
        public void fault(TideFaultEvent event) {
            System.err.println("Fault for " + token + ": " + event.getFault());
        }

        @Override
        public List<Product> getMergeResultWith() {
            return products;
        }
    });
}
```

This may not seem very useful in this case, but when combined with a data binding mechanism such as the one in JavaFX, that means that you don't have to handle the actual result. By using a JavaFX `ObservableList` The binding would transparently propagate all incoming remote data to the UI. Note that this kind of merge will work correctly only with mutable objects (so no `String`, `Number`, ...). It is generally the most useful with collections.

### 4.1.5. Global exception handling

The server exceptions can be handled on the client-side by defining a fault callback on each remote call. It works fine on a case by case basis but it is very tedious and you can always forget a case, in which case the error will be either ignored or result in a global error popup that is not very elegant.

To help dealing with server exceptions, it is possible to define common handlers for particular fault codes on the client-side, and exception converters on the server-side, to convert server exceptions to common fault codes.

On the server, you have to define an `ExceptionHandler` class. For example we could write a converter to handle the JPA `EntityNotFoundException` (in fact there is already a built-in converter for all JPA exceptions):

```
public class EntityNotFoundExceptionConverter implements ExceptionConverter {

    public static final String ENTITY_NOT_FOUND = "Persistence.EntityNotFound";

    public boolean accepts(Throwable t, Throwable finalException) {
        return t.getClass().equals(javax.persistence.EntityNotFoundException.class);
    }

    public ServiceException convert(
        Throwable t, String detail, Map<String, Object> extendedData) {

        ServiceException se = new ServiceException(
            ENTITY_NOT_FOUND, t.getMessage(), detail, t
        );
        se.getExtendedData().putAll(extendedData);
        return se;
    }
}
```

This class will intercept all `EntityNotFound` exceptions on the server-side, and convert it to a proper `ENTITY_NOT_FOUND` fault event.

The argument `finalException` contains the deepest throwable in the error and can be used to check if some higher level exception converter should be used to handle the exception. For example, the `HibernateExceptionHandler` checks if the exception is wrapped in a `PersistenceException`, in which case it lets the JPA `PersistenceExceptionHandler` accept the exception.

This exception converter has to be declared on the GDS server config :

- When using `scan="true"` in `granite-config.xml`, ensure that there is a `META-INF/granite-config.properties` file (even empty) in the jar containing the exception converter class (same principle than the `seam.properties` file to specify which jars need to be scanned in JBoss Seam 2.x).

- When not using automatic scan, you can add this in `granite-config.xml` :

```
<exception-converters>
  <exception-converter type="com.myapp.custom.MyExceptionHandler"/>
</exception-converters>
```

On the client side, you then have to define an exception handler class:

```
public class EntityNotFoundExceptionHandler implements ExceptionHandler {

    public boolean accepts(FaultMessage emsg) {
        return "Persistence.EntityNotFound".equals(emsg.getCode());
    }

    public void handle(Context context, FaultMessage emsg, TideFaultEvent faultEvent) {
        System.err.println("Entity not found: " + emsg.getMessage());
    }
}
```

... and register it as an exception handler in the Tide context. That is simply declare it as a managed bean with `context.set(new EntityNotFoundExceptionHandler())` or as a Spring bean when using Spring.

## 4.2. Mapping between client and server Java objects

The server data objects are usually defined as JPA entities. Using them directly on the client is possible but requires having a runtime dependency on the JPA provider on the client, which may not be practical or suitable at all. This is for example what would happen by using standard Java serialization. Additionally, using a JPA entity on a JavaFX client (for example) means that your data beans will not benefit from all the data binding machinery of JavaFX which requires the use of special properties implementations (`javafx.beans.property.Property`). You could probably build a 'dual' Java class which is both a JPA entity and a bindable JavaFX bean but that would imply a very tight coupling between the client and the server (and a dependency of the server application on JavaFX !!) and might at last not work at all (in particular for collection properties).

Having two different classes for the same data object on the client and the server is thus a cleaner approach and simply requires some tooling to automatically generate one from the other. GraniteDS provides a JPA/JavaBean to JavaFX class generator which handles exactly this task.

## 4.3. Externalizers and Java code generation

### 4.3.1. Example of a JPA entity and its corresponding JavaFX bean

Let's say we have a basic entity bean that represents a person. The following code shows its implementation using JPA annotations:

```
package com.myapp.entity;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Version;

@Entity
public class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    private Integer id;

    @Version
    private Integer version;

    @Basic
    private String firstName;

    @Basic
    private String lastName;

    public Integer getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }
}
```

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

With GraniteDS automated externalization and without any modification made to our bean, we may serialize all properties of the Person JPA entity, and convert them to a Person JavaFX bean. Furthermore, thanks to the Gfx code generator, we do not even have to write the JavaFX bean by ourselves. Here is a sample generated bean implementation:

```
@JavaFXObject
public class PersonBase implements Identifiable, Lazyable, DataNotifier {

    private boolean __initialized = true;
    @SuppressWarnings("unused")
    private String __detachedState = null;

    private final BooleanProperty __dirty = new SimpleBooleanProperty(this, "dirty", false);

    private EventHandlerManager __handlerManager = new EventHandlerManager(this);

    @Override
    public EventDispatchChain buildEventDispatchChain(EventDispatchChain tail) {
        return tail.prepend(__handlerManager);
    }

    public <T extends Event> void addEventHandler(EventType<T> type, EventHandler<? super T> handler) {
        __handlerManager.addEventHandler(type, handler);
    }

    public <T extends Event> void removeEventHandler(EventType<T> type, EventHandler<? super T> handler) {
        __handlerManager.removeEventHandler(type, handler);
    }
}
```

```
public boolean isInitialized() {
    return __initialized;
}

@IgnoredMethod
public BooleanProperty dirtyProperty() {
    return __dirty;
}

public boolean isDirty() {
    return __dirty.get();
}

private ObjectProperty<Long> id = new SimpleObjectProperty<Long>(this, "id");
private StringProperty uid = new SimpleStringProperty(this, "uid");
private ObjectProperty<Integer> version = new SimpleObjectProperty<Integer>(this, "version");
private StringProperty firstName = new SimpleStringProperty(this, "firstName");
private StringProperty lastName = new SimpleStringProperty(this, "lastName");

public ObjectProperty<Long> idProperty() {
    return id;
}

@Id
public Long getId() {
    return id.get();
}

public StringProperty uidProperty() {
    return uid;
}

public void setUid(String value) {
    uid.set(value);
}

public String getUid() {
    return uid.get();
}

public ObjectProperty<Integer> versionProperty() {
    return version;
}

@Version
public Integer getVersion() {
    return version.get();
}
```

```
}

    public StringProperty firstNameProperty() {
        return firstName;
    }
    public void setFirstName(String value) {
        firstName.set(value);
    }
    public String getFirstName() {
        return firstName();
    }
}

    public StringProperty lastNameProperty() {
        return lastName;
    }
    public void setLastName(String value) {
        lastName.set(value);
    }
    public String getLastName() {
        return lastName.get();
    }
}
```

This JavaFX bean reproduces all properties found in the JPA entity, public and private and even includes some extra properties and features, (`__initialized` and `__detachedState`), that correspond to the JPA internal state for lazy loading. Note that these two fields are present because the Gfx generator has detected that our class is a JPA entity annotated with `@Entity`. For simple Java beans, these two fields would not be present, but this shows that the pluggable externalizer mechanism in GraniteDS allows to do a lot more than simply serializing public data and value objects.

You may also notice a few more additions in the generated bean that are useful with more advanced features of the framework. `DataNotifier` is an interface for bean that can dispatch events related to their internal state, that is used by the form validation framework. `dirtyProperty` is a bindable property updated by the data management framework that indicates whether the bean has been modified since its last server update.

With the externalizer mechanism in GraniteDS, serializing data between a client and a server is almost as powerful as pure Java serialization and additionally allows to maintain a clean decoupling between the client and server applications, whatever framework is used on both sides.

### 4.3.2. Standard configuration

In order to externalize the `Person.java` entity bean, we must tell GraniteDS which classes we want to externalize with a special rule in the `granite-config.xml` file:

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

<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <class-getter type="org.granite.hibernate.HibernateClassGetter"/>

  <externalizers>
    <externalizer type="org.granite.hibernate.HibernateExternalizer">
      <include type="com.myapp.entity.Person"/>
    </externalizer>
  </externalizers>
</granite-config>
```

This instructs GraniteDS to externalize all classes named `com.myapp.entity.Person` by using the `org.granite.hibernate.HibernateExternalizer`. Note that the `HibernateClassGetter` configuration is necessary to detect Hibernate proxies (lazy-initialized beans). See more about this feature in the [JPA and lazy initialization](#) section.

If you use an abstract entity bean as a parent to all your entity beans you could use this declaration, but note that `type` in the example above is replaced by `instance-of`:

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

<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <class-getter type="org.granite.hibernate.HibernateClassGetter"/>

  <externalizers>
    <externalizer type="org.granite.hibernate.HibernateExternalizer">
```

```
<include instance-of="com.myapp.entity.AbstractEntity"/>
</externalizer>
</externalizers>
</granite-config>
```

This will avoid the need of writing externalization instructions for all your beans, and all instances of `AbstractEntity` will be automatically externalized.

You may also use an `annotated-with` attribute as follows:

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

<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <class-getter type="org.granite.hibernate.HibernateClassGetter"/>

  <externalizers>
    <externalizer type="org.granite.hibernate.HibernateExternalizer">
      <include annotated-with="javax.persistence.Entity"/>
      <include annotated-with="javax.persistence.MappedSuperclass"/>
      <include annotated-with="javax.persistence.Embeddable"/>
    </externalizer>
  </externalizers>
</granite-config>
```

Of course, you may mix these different attributes as you want. Note, however, that there are precedence rules for these three configuration options: `type` has precedence over `annotated-with` and `annotated-with` has precedence over `instance-of`. Playing with rule precedence provides a way to override general rules with more specific rules for particular classes.

### 4.3.3. Autoscan configuration

Instead of configuring externalizers with the above method, you may use the autoscan feature:

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

```
<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config scan="true"/>
```

With this very short configuration, GraniteDS will scan at startup all classes available in the classpath, actually all classes found in the classloader of the `GraniteConfig` class, and discover all externalizers (classes that implements the `GDS Externalizer` interface). The matching rule are defined implicitly by each externalizer, for example the `Hibernate` externalizer is defined to match all classes annotated with `@Entity`.

### 4.3.4. Built-in externalizers

GraniteDS comes with a set of built-in externalizers for the most usual kinds of Java classes:

- `org.granite.messaging.amf.io.util.externalizer.DefaultExternalizer`: this externalizer may be used with any POJO bean.
- `org.granite.messaging.amf.io.util.externalizer.EnumExternalizer`: this externalizer may be used with Java enum types. When `autoscan` is enabled, it will be automatically used for all enum types.
- `org.granite.hibernate.HibernateExternalizer`: This externalizer may be used with all JPA/Hibernate entities (i.e., all classes annotated with `@Entity`, `@MappedSuperclass` or `@Embeddable` annotations). Include `granite-hibernate.jar` in your classpath in order to use this feature.
- `org.granite.toplink.TopLinkExternalizer`: this externalizer may be used with all JPA/TopLink entities (i.e., all classes annotated with `@Entity`, `@MappedSuperclass` or `@Embeddable` annotations). Include `granite-toplink.jar` in your classpath in order to use this feature.
- `org.granite.eclipselink.EclipseLinkExternalizer`: this externalizer will be used with the new version of TopLink (renamed `EclipseLink`). Include `granite-eclipselink.jar` in your classpath in order to use this feature.
- `org.granite.openjpa.OpenJpaExternalizer`: this externalizer may be used with all JPA/OpenJPA (formerly `WebLogic Kodo`) entities (mainly in `WebLogic` environments). Include `granite-openjpa.jar` in your classpath in order to use this feature.
- `org.granite.datanucleus.DataNucleusExternalizer`: this externalizer may be used with all JPA/DataNucleus entities. Include `granite-datanucleus.jar` in your classpath in order to use this feature.
- `org.granite.tide.cdi.TideEventExternalizer`: this externalizer externalizes classes annotated with the `TideEvent` annotation.

- `org.granite.messaging.amf.io.util.externalizer.LongExternalizer`: externalizes Java long or Long values.
- `org.granite.messaging.amf.io.util.externalizer.BigIntegerExternalizer`: externalizes Java BigInteger values.
- `org.granite.messaging.amf.io.util.externalizer.BigDecimalExternalizer`: externalizes Java BigDecimal values.

### 4.3.5. Built-in client externalizers

GraniteDS provides support for JavaFX beans with the `org.granite.client.javafx.JavaFXExternalizer` externalizer. It will unpack JavaFX properties and serialize them to a normalized network form.

### 4.3.6. Custom externalizers

It is easy to write your own externalizer, you have to implement the `org.granite.messaging.amf.io.util.externalizer.Externalizer` interface, or extend the `DefaultExternalizer` class. There is no particular use case for this extension; it mostly depends on your specific needs and you should look at the standard externalizer implementations to figure out how to write your custom code.

If you use autoscan configuration, make sure your class is packaged in a jar accessible via the `GraniteConfig` class loader (`granite.jar` classpath), put a `META-INF/granite-config.properties` in your jar, even empty, and put relevant code in the `accept` method to define which classes your externalizer should process:

```
public int accept(Class<?> clazz) {  
    return clazz.isAnnotationPresent(MySpecialAnnotation.class) ? 1 : -1;  
}
```

You may, of course, use any kind of conditional expression, based on annotations, inheritance, etc. The returned value is a numeric weight used when GDS tries to figure out what externalizer it should use when it encounters a Java bean at serialization time: -1 means "do not use this externalizer", 0 or more means "use this externalizer if there is no other externalizer that returns a superior weight for this bean". `DefaultExternalizer` has a weight of 0, `EnumExternalizer` and the built-in JPA externalizers a weight of 1. If your class would normally be externalized by the `HibernateExternalizer`, you may, for example, use a weight of 2 when you want to replace the default serialization for some particular entities.



### Note

Creating your own externalizer generally means that you also need to write a corresponding template for the Gas3 generator with matching implementations of `readExternal` and `writeExternal`.

## 4.3.7. @ExternalizedBean and @ExternalizedProperty

Two standard annotations are available that give you more control over the externalization process:

- `@ExternalizedBean`: This class annotation may be used to instruct GDS to externalize the annotated bean with the `DefaultExternalizer` or any other externalizer specified in the type attribute. For example, you could annotate a Java class with:

```
@ExternalizedBean(type=path.to.MyExternalizer.class)
public class MyExternalizedBean {
    ...
}
```

- `@ExternalizedProperty`: This method annotation may be used on a public getter when you want to externalize a property with no corresponding field (i.e., a computed property). For example:

```
public class MyBean {

    private int value;

    ...

    @ExternalizedProperty
    public int getSquare() {
        return value * value;
    }
}
```

Of course, this annotation will only be used if the MyBean class is configured for externalization. Note that externalized properties are always read only: a `setSquare( . . . )` will never be used in the client to server serialization. Note also that Gfx uses this annotation when it generates JavaFX bean so you'll find an extra `square` member field in your generated `MyBean.java`.

### 4.3.8. Custom class getters

A problem with the default AMF3 serialization is to get the true class name of an object in special cases. For example, a simple `myObject.getClass().getName()` with a proxied entity bean would return `org.hibernate.proxy.HibernateProxy` instead of the underlying entity bean class name. In order to get through this kind of problem, you must configure a class getter. Other methods of `ClassGetter` are also used by Tide to determine some internal properties of the managed objects, such as their JPA initialization state.

Class getters are generally used in conjunction with externalizers. For example, the full configuration for an application using Hibernate entities would be (without `autoscan`):

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

<!DOCTYPE granite-config PUBLIC
    "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <class-getter type="org.granite.hibernate.HibernateClassGetter"/>

  <externalizers>
    <externalizer type="org.granite.hibernate.HibernateExternalizer">
      <include instance-of="test.granite.ejb3.entity.AbstractEntity"/>
    </externalizer>
  </externalizers>
</granite-config>
```

The `org.granite.hibernate.HibernateClassGetter` class is used in order to retrieve the correct entity class name from a proxy. You may write and plug your own class getter in a similar way.

### 4.3.9. Instantiators

At deserialization time, from client to server, GraniteDS must instantiate and populate new JavaBeans with serialized data. The population issue (strictly private field), as we have seen before, is addressed by externalizers. But there is still a problem with classes that do not

declare a default constructor. How do we instantiate those classes with meaningful parameters at deserialization time?

When GraniteDS encounters classes without a default constructor, it tries to instantiate them by using the Sun JVM `sun.reflect.ReflectionFactory` class that bypasses this limitation. Then, if it can successfully instantiate this kind of class, fields deserialization follows the standard process with or without externalization. This solution has three serious limitations however: it only works with a Sun JVM, it does not take care of complex initialization you may have put in your custom constructor, and it cannot simply work with classes that should be created via a static method, such as singletons.

With GraniteDS *instantiators*, you may control the instantiation process, delaying the actual instantiation of the class after all its serialized data has been read.

**Built-in instantiators.** Two instantiators come with GDS:

- `org.granite.messaging.amf.io.util.instantiator.EnumInstantiator`: This instantiator is used in order to get an Enum constant value from an Enum class and value (the String representation of the constant), by means of the `java.lang.Enum.valueOf(Class<? extends Enum> enumType, String name)` method.
- `org.granite.hibernate.HibernateProxyInstantiator`: It is used when GDS needs to recreate an `HibernateProxy`. See source code for details.

Note that those instantiators do not require an entry in `granite-config.xml`, they are respectively used by the `EnumExternalizer`, `HibernateExternalizer`, and `TopLinkExternalizer`.

**Custom instantiators.** Let's say you have a JavaBean like this one:

```
package org.test;

import java.util.Map;
import java.util.HashMap;
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;

public class MyBean {

    private final static Map<String, MyBean> beans = new HashMap<String, MyBean>();

    private final String name;
    private final String encodedName;

    protected MyBean(String name) {
        this.name = name;
```

```
try {
    this.encodedName = URLEncoder.encode(name, "UTF-8");
} catch (UnsupportedEncodingException e) {
    throw new RuntimeException(e);
}
}

public static MyBean getInstance(String name) {
    MyBean bean = null;
    synchronized (beans) {
        bean = beans.get(name);
        if (bean == null) {
            bean = new MyBean(name);
            beans.put(name, bean);
        }
    }
    return bean;
}

public String getName() {
    return name;
}

public String getEncodedName() {
    return encodedName;
}
}
```

With this kind of Java class, even with the help of the GDS `DefaultExternalizer` and the `Sun ReflectionFactory` facility, you will not be able to get the cached instance of your bean and the `encodedName` field will not be correctly initialized. Instead, a new instance of `MyBean` would be created with a simulated default constructor and the `name` field would be assigned with serialized data.

The solution is to write a custom instantiator that will be used at deserialization time:

```
package org.test;

import java.util.Collections;
import java.util.List;
```

```

import java.util.ArrayList;

import org.granite.messaging.amf.io.util.instantiator.AbstractInstantiator;

public class MyBeanInstantiator extends AbstractInstantiator<MyBean> {

    private static final long serialVersionUID = -1L;

    private static final List<String> orderedFields;
    static {
        List<String> of = new ArrayList<String>(1);
        of.add("name");
        orderedFields = Collections.unmodifiableList(of);
    }

    @Override
    public List<String> getOrderedFieldNames() {
        return orderedFields;
    }

    @Override
    public MyBean newInstance() {
        return MyBean.getInstance((String)get("name"));
    }
}

```

You should finally use a `granite-config.xml` file as follows in order to use your instantiator:

```

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

<!DOCTYPE granite-config PUBLIC
    "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <externalizers>
    <externalizer type="org.granite.messaging.amf.io.util.externalizer.DefaultExternalizer">
      <include type="org.test.MyBean"/>
    </externalizer>
  </externalizers>

```

```
<instanciators>
  <instanciator type="org.test.MyBean">org.test.MyBeanInstanciator</instanciator>
</instanciators>
</granite-config>
```

### 4.4. JPA and lazy initialization

In many Java EE applications, persistence is done by using a JPA provider (such as Hibernate). The application directly persists and fetch Java entities, so this could seem natural to transfer these same objects to the client layer instead of adding a extra conversion layer with data transfer objects. However this is not as simple as it seems, in particular when using the lazy loading feature of JPA (and most applications using JPA should use lazy loading).

Usual serialization providers (AMF or not) will either throw exceptions during serialization (because the lazy loaded associations are not available at this time), or load the complete object graph and thus limit the applicability of lazy loading (when using patterns such as *Open Session in View*).

GraniteDS on the other hand is able to reliably serialize JPA entities with its externalizer mechanism (even detached objects outside of a JPA session) and supports both kinds of associations: *proxy* (single-valued associations) and *collections* (such as List, Set, Bag and Map). As described in the previous section, it provides built-in support for Hibernate, TopLink/EclipseLink, OpenJPA and DataNucleus.



#### Note

It is important to note that as a JPA detached entity can be reliably serialized between the Java client and the Java EE service, it's perfectly possible (and even recommended) to directly persist or merge entities sent from the client application without any intermediate DTO layer.

#### 4.4.1. Single-valued associations (proxied or weaved associations)

In your JPA entity bean, you may have a single-valued association like this:

```
@Entity
public class MyEntity {

    @Id @GeneratedValue
    private Integer id;
```

```

@OneToOne(fetch=FetchType.LAZY)
private MyOtherEntity other;

// Skipped code...
}

@Entity
public class MyOtherEntity {

    @Id @GeneratedValue
    private Integer id;

    // Skipped code...
}

```

If you load a large collection of `MyEntity` and do not need other references, this kind of declaration prevents unnecessary performance and memory usage (please refer to Hibernate documentation in order to actually fetch these references when you need them). With GDS, you can keep those uninitialized references as is.

When the client deserializes your collection of `MyEntity`'s with lazy loaded `MyOtherEntity` references, it reads a `initialized` flag set to `true` when it encounters a `MyEntity` instance since `MyEntity`'s are all initialized; so it reads all `MyEntity` fields including the `_other` one. When it deserializes a `MyOtherEntity` instance referenced by a `MyEntity`, it reads a `initialized` flag set to `false` since `MyOtherEntity` is lazy loaded, so it only reads the `MyOtherEntity` `id`. Informations put in `__initialized`, `__detachedState` and `_id` are sufficient to restore a correct `HibernateProxy` instances when you give back `MyEntity` objects to the server for update.

#### 4.4.2. Collections (List, Set, Bag, Map)

GDS also provides a way to keep uninitialized collections as is. When the externalizer encounters an uninitialized collection, it does not try to serialize its content and marks it as uninitialized. This information is kept in client beans and when this bean is sent back to the server (e.g., for an update), the externalizer restores a lazy initialized collection in Java. This gives you a good control over serialization depth, as you do not face the risk of serializing the entire graph of your data, and prevents faulty updates (i.e., an empty collection is saved and deletes database data while it was only uninitialized).

For example, in this persistent set:

```

package com.myapp.entity;

```

```
import java.util.HashSet;
import java.util.Set;

...
import javax.persistence.CascadeType;
import javax.persistence.FetchType;
import javax.persistence.OneToMany;

@Entity
public class Person extends AbstractEntity {
    ...
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.LAZY, mappedBy="person")
    private Set<Contact> contacts = new HashSet<Contact>();
    ...
    public Set<Contact> getContacts() {
        return contacts;
    }
    public void setContacts(Set<Contact> contacts) {
        this.contacts = contacts;
    }
}

// code for Contact skipped...
```

```
package com.myapp.entity;

...
import javafx.collections.ObservableList;

@JavaFXObject
@RemoteClass("test.granite.ejb3.entity.Person")
public class Person implements Identifiable, Lazyable, DataNotifier {

    ...
    private ObservableList<Contact> contacts = new PersistentSet<Contact>();
    ...
    public void setContacts(ObservableList<Contact> contacts) {
        this.contacts = value;
    }
    public ObservableList<Contact> getContacts() {
        return this.contacts;
    }
}
```

```
}  
  
// code for Contact skipped...
```

The actual, persistence aware, `ObservableList` implementation is part of a GDS JavaFX client library (`granite-javafx-client.jar`) that contains all you need in order to use the lazy loaded collections feature.

If GDS encounters an uninitialized `Set`, it is serialized as a `org.granite.messaging.persistence.ExternalizablePersistentSet` that contains some extra data indicating its initialization state. Other persistent collections, such as `List`, `Bag`, and `Map`, are handled in a similar manner.

GDS/JPA uses the interface `Identifiable` that requires a readable property `uid` for all entity beans. See a long Hibernate discussion [here](#) about `equals/hashCode/collection` problems and the use of UUIDs. This is only an implementation choice and you are free to code whatever you want, for example generate the `uid` from a natural identifier or from the database key.



### Note

1. With standard configuration (scan set to false), you must use the appropriate class getter together with the persistence externalizer (eg. `org.granite.openjpa.OpenJpaClassGetter` with `org.granite.openjpa.OpenJpaExternalizer`).
2. With all persistence externalizers, provided that you have added the relevant jar to your application classpath, you may use the auto scan feature: `<granite-config scan="true">` without anything else (no class getter or externalizer configuration): your entities will be automatically externalized according to the underlying JPA engine.
3. If you put many persistence externalizers libraries in the same application, only the one corresponding to your JPA provider will be active. This can be useful to build portable application between Java EE servers. If you bundle both `granite-hibernate.jar` and `granite-eclipselink.jar`, the application should work under both JBoss (which bundles Hibernate) and GlassFish 3 (which bundles EclipseLink).

## 4.5. Securing remote destinations

Security in a Java client cannot simply rely on standard web-app security-constraints configured in `web.xml`. Generally, you have only one `channel-definition`, equivalent to a `url-pattern` in `web.xml`, and multiple destinations. So, the security must be destination-based

rather than URL-pattern based, and Java EE standard configuration in `web.xml` does not provide anything like that.

With a configured `SecurityService`, you will be able to use `Channel`'s `setCredentials` and `logout` methods.

Another important feature in security is to be able to create and expose a `java.security.Principal` to, for example, an EJB3 session bean backend so role-based security can be used.

At this time, GraniteDS provides security service implementations for Tomcat5+, Jetty6+, GlassFish V2+ and V3 and WebLogic 10+ servers. Because JBoss comes with Tomcat by default but may be configured to use Jetty instead, Tomcat or Jetty security services may work as well with JBoss.

When you are using Java Enterprise frameworks such as Seam or Spring together with GraniteDS, you may use specific Seam Security or Spring Security implementations instead of the previous container-based services: please refer to [Seam Services](#) or [Spring Services](#) for more information.

### 4.5.1. Configuration

To enable security, you simply put this kind of declaration in your `granite-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC
  "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">
<granite-config>
  ...
  <security type="org.granite.messaging.service.security.TomcatSecurityService"/>
  <!--
  Alternatively for Tomcat 7.x
  <security type="org.granite.messaging.service.security.Tomcat7SecurityService"/>
  Alternatively for Jetty 6.x
  <security type="org.granite.messaging.service.security.Jetty6SecurityService"/>
  For Jetty 7.x/8.x (available at eclipse.org)
  <security type="org.granite.messaging.service.security.Jetty7SecurityService"/>
  For GlassFish 2.x
  <security type="org.granite.messaging.service.security.GlassFishSecurityService"/>
  For GlassFish 3.x
  <security type="org.granite.messaging.service.security.GlassFishV3SecurityService"/>
  For WebLogic
  <security type="org.granite.messaging.service.security.WebLogicSecurityService"/>
  -->
</granite-config>
```

Some of these implementations (currently only TomcatSecurityService) accept an optional parameter. In the case of the Tomcat service, it's the name of the service that will be used to execute the authentication in case you have many services defined in your `server.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC
  "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">
<granite-config>
  ...
  <security type="org.granite.messaging.service.security.TomcatSecurityService">
    <param name="service" value="your-tomcat-service-name-here"/>
  </security>
</granite-config>
```

You may now use role-based security on destination in your `services-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <service id="granite-service"
      class="flex.messaging.services.RemotingService"
      messageTypes="flex.messaging.messages.RemotingMessage">
      <destination id="person">
        <channels>
          <channel ref="my-graniteamf"/>
        </channels>
        <properties>
          <scope>session</scope>
          <source>com.myapp.PersonService</source>
        </properties>
        <security>
          <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
              <role>user</role>
              <role>admin</role>
            </roles>
          </security-constraint>
        </security>
      </destination>
    </service>
  </services>
</services-config>
```

```
        </roles>
      </security-constraint>
    </security>
  </destination>

  <destination id="restrictedPerson">
    <channels>
      <channel ref="my-graniteamf"/>
    </channels>
    <properties>
      <scope>session</scope>
      <source>com.myapp.RestrictedPersonService</source>
    </properties>
    <security>
      <security-constraint>
        <auth-method>Custom</auth-method>
        <roles>
          <role>admin</role>
        </roles>
      </security-constraint>
    </security>
  </destination>
</service>
</services>
...
</services-config>
```

Here, the person destination can be used by authenticated users with user or admin roles, while the restrictedPerson destination can only be used by authenticated users with the admin role.

Please refer to Tomcat and JBoss documentation for setting up your users/roles configuration.

### 4.5.2. Fine-grained per-destination security

You may write and configure a specific `RemoteDestinationSecurizer` in order to add fine grained security checks for specific actions.

```
public interface RemotingDestinationSecurizer extends DestinationSecurizer {

    public void canExecute(ServiceInvocationContext context)
        throws SecurityServiceException;
}
```

You then have to tell GraniteDS where to use your securizer:

```
<services-config>
  <services>
    <service ...>
      <destination id="restrictedDestination">
        ...
        <properties>
          <securizer>path.to.MyDestinationSecurizer</securizer>
        </properties>
      </destination>
    </service>
  </services>
  ...
</services-config>
```

Note that securizers, if any, are always called before the standard `SecurityService.authorize()` method.

### 4.5.3. Deserialization protection

At Java side, AMF deserialization instantiates classes that are referenced in the binary-encoded request coming from the client. Thus, a malicious AMF3 request can be crafted in order to instantiate an arbitrary Java class (and execute its constructor and setters) that has nothing to do with the expected data exchanged between the client application and the server application.

GraniteDS' fix for this security issue relies on a new configurable option that you can put in your `granite-config.xml` file. If you don't configure anything, you will always see this warning at the startup of the application:

```
WARN [GraniteConfig] You should configure a deserializer securizer in your granite-config.xml
file in order to prevent potential security exploits!
```

In order to secure your application, you are strongly encouraged to configure a securizer as follows:

```
<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config scan="true">

  <amf3-deserializer-securizer param="
    org\granite\.* |
    flex\.messaging\.* |
    com\.myapp\.entity\.*
  ">

  ...
</granite-config>
```

By default, the securizer uses the `org.granite.messaging.amf.io.RegexAMF3DeserializerSecurizer` class that, uses a regular expression parameter. Only classes whose name match one of these patterns are allowed to be instantiated. Of course, all standard Java types are allowed by default and you don't have to explicitly add their package names expressions.

If this default regex-based implementation doesn't fit your needs, you may write your own securizer implementation. It only has to implement the `org.granite.messaging.amf.io.AMF3DeserializerSecurizer` interface and can be specified in `granite-config.xml`:

```
<!DOCTYPE granite-config PUBLIC
"-//Granite Data Services//DTD granite-config internal//EN"
"http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config scan="true">

  <amf3-deserializer-securizer type="com.myapp.MySecurizer"/>

  ...
</granite-config>
```

# JavaFX Code Generator

## 5.1. Overview

One of the main interests of using AMF remoting is that it can maintain a strongly typed bindable JavaFX data model in the client application. However that implies that you have to write a specific JavaFX class for each Java class that will be serialized. Writing and maintaining these JavaFX beans is tedious and a source of many errors. In order to solve this problem and accelerate the development of JavaFX/Java EE applications, GraniteDS comes with an code generator that writes JavaFX beans for all Java beans.

Additionally this generator specifically supports the externalization mechanism of GraniteDS and is able to generate corresponding JavaFX classes for externalized Java beans (typically JPA/ Hibernate entities) with specific templates.

Finally this generator is able to write typesafe client proxies for exposed remote services. Compared to the `RemoteService` API, this can greatly help development by bringing auto-completion and improved type-safety when using remote services.

Gfx may also replicate validation annotations in order to use the client side validation framework (see [Bean Validation \(JSR-303\)](#)).

The generator (named GFX) is implemented as an Ant task. This Ant task is packaged as an Eclipse 3.2+ Ant plugin but may also be used outside of Eclipse for command line Ant calls. It can also be used with Maven by means of the [Flexmojos](http://flexmojos.sonatype.org/) [http://flexmojos.sonatype.org/] Maven plugin.

## 5.2. Generated JavaFX Classes

A common problem with code generators is the potential loss of manual modifications made in generated files. A generated file must be either generated once and only once, allowing for safe manual modifications, but it will not be able to reflect the modifications made in its model (JavaBeans), or regenerated each time its model has been changed, thus preventing safe manual modifications.

Gfx uses the principle of "Base" and customizable inherited classes that let you add methods to generated classes without facing the risk of losing them when a new generation process is executed. For example, here are the two files generated for a given Java entity bean:

`Welcome.java`

```
package org.test;

import java.io.Serializable;
```

```
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Welcome implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    private Integer id;

    @Basic
    private String name;

    public Welcome() {
    }

    public Welcome(String name) {
        this.name = name;
    }

    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Welcome.java

```
/**
 * Generated by Gas3 v2.3.0 (Granite Data Services).
 *
 * NOTE: this file is only generated if it does not exist. You may safely put
```

```

* your custom code here.
*/

package org.test.client;

@JavaFXObject
@RemoteClass("org.test.Welcome")
public class Welcome extends WelcomeBase {
}

```

WelcomeBase.java

```

/**
 * Generated by Gas3 v2.3.0 (Granite Data Services).
 *
 * WARNING: DO NOT CHANGE THIS FILE. IT MAY BE OVERWRITTEN EACH TIME YOU USE
 * THE GENERATOR. INSTEAD, EDIT THE INHERITED CLASS (Welcome.as).
 */

package org.test.client;

@JavaFXObject
public class WelcomeBase implements IExternalizable {
    ...
}

```

The recommendations for manual editing are explicit in the header comments of each generated classes: while the "Base" class may be regenerated at any time, keeping it sync with its Java model class, the inherited one is only generated when it does not exist and you may safely add custom methods into it.

This two files generation principle is used for all generated classes except interface and enum: these classes are generated without any "Base" class and overwritten each time you have modified their Java counterparts.



## Warning

Note: Do not modify manually generated client interface or enum classes !

Here are the details for (re)generation conditions:

Note that for Java classes, relevant timestamp is the last modified time of the .class file, not the .java file.

| Templates                             | Conditions for (re)generation                                                                                                                                                                          |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dual templates (base + inherited)     | The inherited JavaFX class is generated only once if it does not exist. The JavaFX base one is generated if it does not exist or if its timestamp (last modified time) is less than the Java class one |
| Single template (enums or interfaces) | Like the base condition above, the JavaFX class is (re)generated if it does not exist or if its timestamp is less than the Java class one                                                              |

### 5.3. Java Classes and Corresponding Templates

Here is the summary of templates used by the generator depending on the kind of Java class it encounters:

| Type of Java Class                                                     | Template      | Base Template  |
|------------------------------------------------------------------------|---------------|----------------|
| Standard Java beans                                                    | bean.gsp      | beanBase.gsp   |
| JPA entities: all classes annotated with @Entity and @MappedSuperclass | entity.gsp    | entityBase.gsp |
| Java enums                                                             | enum.gsp      | (none)         |
| Java interfaces                                                        | interface.gsp | (none)         |
| Java services: all classes annotated with @RemoteDestination           | remote.gsp    | remoteBase.gsp |
| Java events (CDI): all classes annotated with @TideEvent               | bean.gsp      | beanBase.gsp   |

Note that all these templates are bundled in the granite-generator.jar archive, in the org.granite.generator.javaafx.template package and accessible as resources via the class loader.

### 5.4. Eclipse Plugin

TODO (see doc for Flex plugin).

JavaFX classes can also be generated with the Gas3 plugin for Flex. You just need to define a specific configuration to generate JavaFX classes :

- Use the org.granite.generator.javaafx.JavaFXGroovyTransformer transformer class
- Use the org.granite.generator.javaafx.DefaultJavaFXTypeFactory type factory class
- Select the JavaFX templates in the org/granite/generator/javaafx/template (for example class:org/granite/generator/template/entity.gsp)

## 5.5. Ant Task

**Installation in Eclipse.** Download `org.granite.builder_***.jar`, and drop it in your Eclipse plugins directory (remove any older version and restart Eclipse). The *Add GraniteDS Nature* option should now be available if you right-click on your Java project and the `gas3` Ant task should be ready to use in your `build.xml` file under Eclipse.

**Standalone Installation.** Download `org.granite.builder_***.jar` and unzip it somewhere (create a new directory, this jar doesn't contain a root folder). Move the `lib` directory somewhere else (say `gas3libs` at the root of you harddrive). In your `build.xml`, you must declare the `Gfx` ant task as follows:

```
<taskdef name="gas3" classname="org.granite.generator.javaafx.AntJavaFXTask"/>
```

To launch a build process with `Gfx` targets, you should go to your Java source root directory and type something like:

```
$ ant -lib /gfxlibs -f build.xml {target}
...
```

Just replace `{target}` with a valid target name and make sure Ant is correctly set up: set `ANT_HOME` variable and put `<ANT_HOME>/bin` in your `PATH` environment variable.

**Basic Usage.** After installation, you may use the `Gas3` Ant task in any target of an Ant build file. A working example of `Gas3` usage is available in the `examples/graniteds_ejb3` sample application. For example:

```
<target name="generate.fx">
  <gfx outputdir="java">
    <classpath>
      <pathelement location="classes"/>
    </classpath>
    <fileset dir="classes">
      <include name="com/myapp/entity/**/*.class"/>
    </fileset>
  </gas3>
</target>
```

As you can notice, Gfx generates JavaFX beans from Java compiled classes. You may use multiple Ant filesets in order to specify for which JPA classes you want to generate JavaFX beans. The `classpath` node is used for fileset class loading, and you may reference extra jars or classes needed by your beans class loading.

The `outputdir` attribute lets you instruct Gfx in which directory JavaFX beans will be generated (e.g., `./java`). This path is relative to your current project directory and Gfx will create subdirectories for packages. JavaFX beans will by default have the same package hierarchy as Java classes, with the same subdirectories as well. This may not be very convenient, so it is recommended that you use a package translation definition (see below [package translators](#)).

For each JPA entity (say `com.myapp.entity.MyEntity`), Gfx will generate two JavaFX beans:

- `org.entity.MyEntityBase.java`: This bean mainly contains fields, getters, setters, and extra methods. This file is generated if it does not exist or if it is outdated.
- `org.entity.MyEntity.java`: This bean inherits from the "Base" one and is only generated if it does not exist.

While you should not modify the "Base" file, since your modifications may be lost after another generation process, you may safely add your code to the inherited bean.

You can also use Ant `zipfilesets` if you want to generate JavaFX classes from an existing jar. Note that the jar must be in the `classpath`:

```
<target name="generate.fx">
  <gfx outputdir="java">
    <classpath>
      <pathelement location="lib/myclasses.jar"/>
    </classpath>
    <zipfileset src="lib/myclasses.jar">
      <include name="com/myapp/entity/**/*.class"/>
    </zipfileset>
  </gfx>
</target>
```

**Packages Translations.** You may tell Gfx to generate client classes with a different package and directory structure than the corresponding Java server classes. This is even highly recommended to avoid classpath conflicts or ambiguous autocompletions in the IDE.

```

<gfx ...>
  <classpath .../>
  <fileset .../>

  <translator
    java="path.to.my.java.class"
    client="path.to.my.client.class" />
  <translator
    java="path.to.my.java.class.special"
    client="otherpath.to.my.client.class.special" />
  ...
</gfx>

```

Gfx uses these translators with a "best match" principle; all Java classes within the `path.to.my.java.class` package, and subpackages as well, will be translated to `path.to.my.client.class`, while `path.to.my.java.class.special` will use a specific translation (`otherpath.to.my.client.class.special`).

**Groovy Templates.** Gfx generation relies on Groovy templates. You may plug your own templates in by using one of the advanced options attributes below. For example, you could add a `entitytemplate="/absolute/path/to/my/groovy/entityTemplate.gsp"` attribute to the `gfx` node. You can also specify paths to your custom templates relative to the current Ant project `basedir` directory. If you want to see the Groovy code of the default templates, just unpack `granite-generator.jar` in the `lib` directory of the plugin, and look for `org/granite/generator/template/*[Base].gsp` files.

**Advanced Options (Gfx XML Attributes).** Here is the complete list of Gfx node attributes:

- `outputdir` and `baseoutputdir`: We have already seen the `outputdir` attribute in basic usage. `baseoutputdir` lets you define a custom output directory for your "Base" generated files. The default is to use the same directory as specified by the `outputdir` attribute.
- `uid`: If you want your JavaFX to implement `Identifiable`, you must tell the generator the name of the Java field that contains this UID. By default, Gfx will search for a field named `uid`. You may change this by adding a `uid="myUid"` attribute to the `gfx` node. If Gfx does not find this `uid`, it will be silently ignored.
- `tide`: Should we use a Tide specific template instead of the standard base template used for entity beans (true or false, default is false). Setting this attribute has no effect if you use a custom entity base template. See below.
- `entitytemplate` and `entitybasetemplate`: Templates used for classes annotated with `@Entity` or `@MappedSuperclass`.
- `interfacetemplate`: Template used for Java interfaces.

- `beantemplate` and `beanbasetemplate`: Templates used for other Java classes including `@Embeddable`.
- `enumtemplate`: Template used for `java.lang.Enum` types.
- `remotetemplate` and `remotebasetemplate`: Templates used for server services (EJB3, Spring or Seam services).
- `clienttypefactory`: You can plug your own `org.granite.generator.as3.As3TypeFactory` implementation in order to add support for custom types. For example, if you have configured a custom Joda time converter, you may extend `Gfx` accordingly for this custom type. Just extend the `org.granite.generator.javaafx.DefaultJavaFXTypeFactory` class and return for example `com.myapp.custom.DATE` when you encounter a Joda `DateTime` instance. See [Handling custom data types](#) for a detailed example.
- `entityfactory`: Class used to introspect specific entity properties or metadata (default is `org.granite.generator.as3.DefaultEntityFactory`). You may also use the built-in `org.granite.generator.as3.BVEntityFactory` in order to replicate bean validation annotations into your AS3 model [Bean Validation \(JSR-303\)](#).
- `remotedestinationfactory`: Class used to introspect specific service properties or metadata (default is `org.granite.generator.as3.DefaultRemoteDestinationFactory`).
- `transformer`: Class used to control the generation process (very advanced use). Default for JavaFX is `org.granite.generator.javaafx.JavaFXGroovyTransformer`.

For example:

```
<target name="generate.fx">
  <gfx
    outputdir="java"
    baseoutputdir="base_java"
    uid="myUidFieldName"
    entitytemplate="/myEntityTemplate.gsp"
    entitybasetemplate="/myEntityBaseTemplate.gsp"
    interfacetemplate="/myInterfaceTemplate.gsp"
    beantemplate="/myBeanTemplate.gsp"
    beanbasetemplate="/myBeanBaseTemplate.gsp"
    enumtemplate="/myEnumTemplate.gsp"
    remotetemplate="/myRemoteTemplate.gsp"
    remotebasetemplate="/myRemoteBaseTemplate.gsp"
    tide="true"
    clienttypefactory="path.to.MyCustomTypeFactory"
    entityfactory="path.to.MyEntityFactory"
    remotedestinationfactory="path.to.MyRDFactory"
    transformer="path.to.MyTransformer"
```

```

externalizelong="true"
externalizebiginteger="true"
externalizebigdecimal="true">
  <classpath>
    <pathelement location="classes"/>
  </classpath>
  <fileset dir="classes">
    <include name="test/granite/ejb3/entity/**/*.class"/>
  </fileset>
</gas3>
</target>

```

Note that when using a custom `clienttypefactory`, `entityfactory`, `remotedestinationfactory` or `transformer` attribute, you must configure the `classpath` in order to make your custom classes available to the Gfx engine; either use the `classpath` attribute in the taskdef declaration or in the `gfx` call.

## 5.6. Maven Plugin (Flexmojos)

The Gfx generator is used as the default code generation tool in the Flexmojos plugin. To use it, you need to add the following part to your maven POM :

```

<build>
  ...
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonatype.flexmojos</groupId>
        <artifactId>flexmojos-maven-plugin</artifactId>
        <version>${flexmojos.version}</version>
      </plugin>
    </plugins>
  </pluginManagement>

  <plugins>
    <plugin>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-maven-plugin</artifactId>
      <version>${flexmojos.version}</version>
      <extensions>true</extensions>
      <executions>

```

```
<execution>
  <goals>
    <goal>generate</goal>
  </goals>
  <configuration>
    <generatorToUse>graniteds23</generatorToUse>
    <baseOutputDirectory>${project.build.directory}/generated-sources</
baseOutputDirectory>
    <outputDirectory>${basedir}/src/main/java</outputDirectory>
    <translators>
      <translator>com.myapp.server=com.myapp.client</translator>
    </translators>
    <extraOptions>
      <tide>true</tide>
      <uid>uid</uid>
      <transformer>org.granite.generator.javafx.JavaFXGroovyTransformer</
transformer>
      <as3typefactory>org.granite.generator.javafx.DefaultJavaFXTypeFactory</
as3typefactory>
      <entityFactory>org.granite.generator.as3.BVEntityFactory</entityFactory>
      <outputEnumToBaseOutputDirectory>false</outputEnumToBaseOutputDirectory>
    </extraOptions>
    <includeJavaClasses>
      <include>${package}.entities.**</include>
      <include>${package}.services.*Service</include>
    </includeJavaClasses>
    <templates>
      <base-bean-template>classpath:org/granite/generator/javafx/template/
tideBeanBase.gsp</base-bean-template>
      <bean-template>classpath:org/granite/generator/javafx/template/bean.gsp</bean-
template>
      <base-entity-template>classpath:org/granite/generator/javafx/template/
tideEntityBase.gsp</base-entity-template>
      <entity-template>classpath:org/granite/generator/javafx/template/entity.gsp</entity-
template>
      <base-remote-template>classpath:org/granite/generator/javafx/template/
tideRemoteBase.gsp</base-remote-template>
      <remote-template>classpath:org/granite/generator/javafx/template/
tideRemote.gsp</remote-template>
      <enum-template>classpath:org/granite/generator/javafx/template/enum.gsp</enum-
template>
    </templates>
  </configuration>
</execution>
```

```
</executions>
<dependencies>
  <dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.0-api</artifactId>
    <version>1.0.1.Final</version>
  </dependency>
  <dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
  </dependency>
  <dependency>
    <groupId>javax.jdo</groupId>
    <artifactId>jdo2-api</artifactId>
    <version>2.3-eb</version>
  </dependency>
  <dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy</artifactId>
    <version>1.6.0</version>
  </dependency>
  <dependency>
    <groupId>antlr</groupId>
    <artifactId>antlr</artifactId>
    <version>2.7.7</version>
  </dependency>
  <dependency>
    <groupId>asm</groupId>
    <artifactId>asm</artifactId>
    <version>2.2.3</version>
  </dependency>
  <dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.2.2</version>
  </dependency>
  <dependency>
    <groupId>org.sonatype.flexmojos</groupId>
    <artifactId>flexmojos-generator-graniteds-2.3.0</artifactId>
    <version>${flexmojos.version}</version>
  </dependency>
  <dependency>
    <groupId>org.graniteds</groupId>
```

```
        <artifactId>granite-core</artifactId>
        <version>${graniteds.version}</version>
    </dependency>
    <dependency>
        <groupId>org.graniteds</groupId>
        <artifactId>granite-generator-share</artifactId>
        <version>${graniteds.version}</version>
    </dependency>
    <dependency>
        <groupId>org.graniteds</groupId>
        <artifactId>granite-generator</artifactId>
        <version>${graniteds.version}</version>
    </dependency>
</dependencies>
</plugin>
...
</plugins>
...
</build>
```

### 5.7. Template Language

TODO: see documentation for Gas3 ActionScript 3 generator.

## Messaging (Gravity)

Granite Data Services provides a real-time messaging service, code name *Gravity*. It currently provides a [Comet](http://en.wikipedia.org/wiki/Comet_(programming)) [http://en.wikipedia.org/wiki/Comet\_(programming)]-like implementation with AMF3 data polling over HTTP and a [WebSocket](http://datatracker.ietf.org/doc/rfc6455/?include_text=1) [http://datatracker.ietf.org/doc/rfc6455/?include\_text=1] based implementation. Both can be used with the same producer/consumer based API.

The Comet implementation is freely inspired from the [Bayeux](http://cometd.com/bayeux/Bayeux) [http://cometd.com/bayeux/Bayeux] protocol specification and adapted from the Jetty 6.1.x implementation of a cometd server.

The WebSocket server implementation uses the native WebSocket capabilities of the deployment application server when available (Tomcat 7.0.29+, GlassFish 3.1.2+, Jetty 8.1.1+) or can alternatively use an embedded Jetty server.

The WebSocket client uses by default the Jetty WebSocket client library.

### 6.1. Example usage with Consumer/Producer

GraniteDS messaging relies on two main components on the client side: `org.granite.client.messaging.Consumer` and `org.granite.client.messaging.Producer`.

Here is a quick example of GDS Consumer/Producer usage with a Comet/long-polling channel:

```
...
import org.granite.client.messaging.Consumer;
import org.granite.client.messaging.Producer;
...

public void test() {
    HTTPTransport transport = new ApacheAsyncTransport();
    AMFMessagingChannel channel = new AMFMessagingChannel(transport, "gravityamf", new URI("http://localhost:8080/myapp/gravityamf/amf"));
    transport.start();

    Consumer consumer = new Consumer(channel, "chat", "discussion");
    consumer.addMessageListener(new TopicMessageListener() {
        @Override
        public void onMessage(TopicMessageEvent event) {
            System.out.println(event.getData());
        }
    });
};
```

```
ResponseMessageFuture future = consumer.subscribe(new ResultFaultIssuesResponseListener() {
    @Override
    public void onResult(ResultEvent event) {
        System.out.println("onSubscribeSuccess");
    }

    @Override
    public void onFault(FaultEvent event) {
        System.out.println("onSubscribeFault");
    }

    @Override
    public void onIssue(IssueEvent event) {
        System.out.println("onSubscribeIssue");
    }
});
future.get();

producer = new Producer(channel, "chat", "discussion");
producer.publish("Hello world").get();

Thread.sleep(1000);
}
...
```

In this code, the producer sends `String` messages, which could of course be of any type, and the producer receives `String` messages as well.

The same with a `WebSocket` channel:

```
...
import org.granite.client.messaging.Consumer;
import org.granite.client.messaging.Producer;
...

public void test() {
    WebSocketTransport transport = new JettyWebSocketTransport();
    AMFMessagingChannel channel = new AMFMessagingChannel(transport, new URI("http://
localhost:8080/myapp/websocketamf/amf"));
    transport.start();
}
```

```

Consumer consumer = new Consumer(channel, "chat", "discussion");
consumer.addListener(new TopicMessageListener() {
    @Override
    public void onMessage(TopicMessageEvent event) {
        System.out.println(event.getData());
    }
});

ResponseMessageFuture future = consumer.subscribe(new ResultFaultIssuesResponseListener() {
    @Override
    public void onResult(ResultEvent event) {
        System.out.println("onSubscribeSuccess");
    }

    @Override
    public void onFault(FaultEvent event) {
        System.out.println("onSubscribeFault");
    }

    @Override
    public void onIssue(IssueEvent event) {
        System.out.println("onSubscribeIssue");
    }
});
future.get();

producer = new Producer(channel, "chat", "discussion");
producer.publish("Hello world").get();

Thread.sleep(1000);
}
...

```

## 6.2. Topics and Selectors

By default all messages sent by a producer are transmitted to all subscribed consumers. In most cases you will want to more finely control how the messages are routed. There are two main ways of doing this: the easiest is the topic and the most advanced is by using selectors.

Topics are a way to divide the destination in many parts. When a producer sends a message on a particular topic, only the consumers attached to this topic will receive the message. For example, if you have a destination for quotes, you could have a topic for each country:

```
Producer producer = new Producer(channel, "quotes", "/germany");
producer.publish(message);

Consumer consumerGermany = new Consumer(channel, "quotes", "/germany");
consumerGermany.subscribe(new ResponseListener() { ... }).get();

Consumer consumerFrance = new Consumer(channel, "quotes", "/france");
consumerFrance.subscribe(new ResponseListener() { ... }).get();
```

Here only `consumerGermany` will receive the messages published by the producer. Note the slash (/) to start the name of the topic. You can define more sections for the topic name and use wildcards (\*) and (\*\*) to match a part of the topic. For example you could define a hierarchy `/europe/germany`, `/europe/france`, `/america/US`, and define a consumer for the topic `/europe/*` that will receive only messages for Germany and France. Finally a consumer with `/**` will receive everything, whatever topic is used by the producer.



### Note

The JMS adapter currently does not support this filtering by topic as this is not a standard feature of JMS and many JMS providers do not have this concept. The ActiveMQ adapter however does support it.

Topics are a simple way of filtering the message, but in some cases you may want to use more sophisticated rules to route the messages from producers to consumers. Gravity uses the concept of message selectors from JMS to do this. It works by defining a SQL-like select string that will define the criteria that a consumer wants on the message headers.

A consumer can specify its message selector before it subscribes to the destination:

```
Consumer consumerFrance = new Consumer(channel, "quotes", null);
consumerFrance.setSelector("COUNTRY = 'France'");
consumerFrance.subscribe(new ResponseListener() { ... }).get();
```

This consumer will receive all messages that have a header named `COUNTRY` with the value `France`. Many header values can be combined in the selector with `AND` and `OR`, and you can use operators. See [JMS documentation](http://download.oracle.com/javase/1.4/api/javax/jms/Message.html) [http://download.oracle.com/javase/1.4/api/javax/jms/Message.html] for details.

## 6.3. Common configuration

There are three main steps to configure Gravity in an application:

- Declare the Gravity servlet implementation for your target server in `web.xml`
- Declare a messaging service and destination in `services-config.xml`, mapped to a specific channel definition of type `GravityChannel`

```
<web-app version="2.5" ...>
  ...
  <listener>
    <listener-class>org.granite.config.GraniteConfigListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>GravityServlet</servlet-name>
    <servlet-class>org.granite.gravity.tomcat.GravityTomcatServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>GravityServlet</servlet-name>
    <url-pattern>/gravityamf/*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

This declaration is the one specific to the Tomcat application server. See below for all available Gravity servlet implementations.



### Note

The servlet listener definition is important to ensure proper startup and shutdown of the Gravity services, in particular cleanup of used resources.

```
<services-config>
  <services>
    <service id="messaging-service"
      class="flex.messaging.services.MessagingService"
```

```
messageTypes="flex.messaging.messages.AsyncMessage">
  <adapters>
    <adapter-definition
      id="default"
      class="org.granite.gravity.adapters.SimpleServiceAdapter"
      default="true"/>
    </adapters>

    <destination id="topic">
      <channels>
        <channel ref="my-gravityamf"/>
      </channels>
    </destination>
  </service>
</services>

<channels>
  <channel-definition
    id="my-gravityamf"
    class="org.granite.gravity.channels.GravityChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/gravityamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>
```

Here, we define a GravityChannel (my-gravityamf) and we use it in the destination named topic. See above destination usage in [Consumer/Producer usage](#).

The topic we have defined uses the default Gravity adapter SimpleServiceAdapter that is a simple fast in-memory message bus. If you need more advanced features such as persistent messages or clustering, you should consider using a dedicated messaging implementation such as [Apache ActiveMQ](http://activemq.apache.org/) [http://activemq.apache.org/].

The simple adapter exposes two configuration properties:

- no-local: default is true, if set to false the client producing messages will receive their own messages
- session-selector: this is an advanced option and instructs Gravity to store the message selector string in the user session. This allows the server part of the application to override the selector string defined by the client Consumer. The selector is stored and read from the session attribute named org.granite.gravity.selector.{destinationId}.

### 6.3.1. Supported application servers for Comet/long polling

GraniteDS provides a generic servlet implementation that can work in any compliant servlet container. However it will use blocking IO and thus will provide relatively limited scalability.

Before the release of the Servlet 3.0 specification, there was no standard way of writing asynchronous non blocking servlets and each server provided its own specific API (for example Tomcat CometProcessor or Jetty continuations). GraniteDS thus provides implementations of non blocking messaging for the most popular application servers.

Here is the table of the supported implementations:

| Application server     | Servlet class                                        | Specific notes                                         |
|------------------------|------------------------------------------------------|--------------------------------------------------------|
| Tomcat 6.0.18+         | org.granite.gravity.tomcat.GravityCometProcessor     | Only with APR/NIO enabled (APR highly recommended)     |
| JBoss 4.2.x            | org.granite.gravity.tomcat.GravityCometProcessor     | APR/NIO Servlet disable CommonHeadersFilter            |
| Jetty 6.1.x            | org.granite.gravity.jetty.GravityContinuationServlet | Jetty 7 not supported, Jetty 8 using Servlet 3 API     |
| JBoss 5+               | org.granite.gravity.jbossweb.GravityCometProcessor   | Only with APR/NIO enabled (APR highly recommended)     |
| WebLogic 9.1+          | org.granite.gravity.weblogic.GravityCometProcessor   | See Weblogic documentation for configuration tuning    |
| GlassFish 3.x          | org.granite.gravity.servlet3.GravityCometProcessor   | Using Servlet 3.0, requires async-supported in web.xml |
| Tomcat 7.x / Jetty 8.x | org.granite.gravity.servlet3.GravityCometProcessor   | Using Servlet 3.0, requires async-supported in web.xml |
| Any other              | org.granite.gravity.generic.GravityBlockingServlet   | Using blocking I/O (no asynchronous support)           |

### 6.3.2. Supported application servers for WebSocket

There is no standard way (yet) to use WebSockets in Java EE application servers thus GraniteDS provides support for native WebSocket implementations on some application servers.

Here is the table of the supported implementations:

| Application server | Servlet class                              | Specific notes                                     |
|--------------------|--------------------------------------------|----------------------------------------------------|
| Tomcat 7.0.29+     | org.granite.gravity.tomcat.TomcatWebSocket | Only with APR/NIO enabled (APR highly recommended) |
| Jetty 8.1.1+       | org.granite.gravity.jetty8.JettyWebSocket  | Jetty 7 not supported                              |

| Application server | Servlet class                                           | Specific notes                                  |
|--------------------|---------------------------------------------------------|-------------------------------------------------|
| GlassFish 3.1.2+   | org.granite.gravity.glassfish.GlassFishWebSocketServlet |                                                 |
| Any other          | Embedded Jetty 8.1.1+                                   | Requires another TCP port, not webapp dependent |

### 6.3.3. Advanced configuration

Whichever Gravity servlet implementation is used in your application, the advanced configuration is done in `granite-config.xml`. Here is a sample Gravity configuration with all default options:

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

<!DOCTYPE granite-config PUBLIC "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>

    <gravity
        factory="org.granite.gravity.DefaultGravityFactory"
        channel-idle-timeout-millis="1800000"
        long-polling-timeout-millis="20000"
        reconnect-interval-millis="30000"
        reconnect-max-attempts="60">

        <thread-pool
            core-pool-size="5"
            maximum-pool-size="20"
            keep-alive-time-millis="10000"
            queue-capacity="2147483647" />

        </gravity>

    </granite-config>
```

This `<gravity>` section is purely optional and you may omit it if you accept default values.

Some explanations about these options:

- `channel-idle-timeout-millis`: the elapsed time after which an idle channel (pure producer or dead client) may be silently unsubscribed and removed by Gravity. Default is 30 minutes.

- `long-polling-timeout-millis`: the elapsed time after which an idle connect request is closed, asking the client to reconnect. Default is 20 seconds. Note that setting this value isn't supported in Tomcat/APR configurations.
- `thread-pool` attributes: all options are standard parameters for the Gravity `ThreadPoolExecutor` instance.

All other configuration options are for advanced use only and you should keep default values.

### 6.3.4. Tomcat and JBoss/Tomcat specific configuration tips

GraniteDS messaging for Tomcat relies on the `org.apache.catalina.CometProcessor` interface. In order to enable Comet support in Tomcat, you must configure an [APR or NIO connector](http://tomcat.apache.org/tomcat-6.0-doc/aio.html) [http://tomcat.apache.org/tomcat-6.0-doc/aio.html].

At least for now, APR is the easiest to configure and the most reliable. To configure APR, see documentation [here](http://tomcat.apache.org/tomcat-6.0-doc/apr.html) [http://tomcat.apache.org/tomcat-6.0-doc/apr.html]. On Windows®, it's simply a matter of downloading a native *dll* [http://tomcat.heanet.ie/native/] and putting it in your `WINDOWS/system32` directory – while other and better configurations are possible. For more recent versions of Tomcat such as the one embedded in JBoss 5 or 6, you will need the latest APR library, see [here](http://tomcat.apache.org/download-native.cgi) [http://tomcat.apache.org/download-native.cgi].

For JBoss 4.2.\*, you must comment out a specific filter in the default global `web.xml` (`<JBoss_HOME>/server/default/deploy/jboss-web.deployer/conf/web.xml`):

```
...
<!-- Comment this out!
<filter>
  <filter-name>CommonHeadersFilter</filter-name>
  <filter-class>org.jboss.web.tomcat.filters.ReplyHeaderFilter</filter-class>
  <init-param>
    <param-name>X-Powered-By</param-name>
    <param-value>...</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>CommonHeadersFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
-->
...
```

See above for Tomcat configuration.

For JBoss 5+ servers, you must use a specific servlet. JBoss 5 implements its own version of Tomcat, named JBossWeb:

```
<web-app version="2.4" ...>
  ...
  <servlet>
    <servlet-name>GravityServlet</servlet-name>
    <servlet-class>org.granite.gravity.jbossweb.GravityJBossWebServlet</servlet-class>
    ... (see Tomcat configuration above for options)
  </servlet>
  ...
</web-app>
```

Note that you do not need to comment out the `CommonHeadersFilter` with JBoss 5, but you still need to enable APR.

### 6.4. Integration with JMS

The default messaging engine of GraniteDS is embedded in `SimpleServiceAdapter` and has many limitations. In particular it does not support clustering. For more robust messaging, it is possible and recommended to integrate with a robust messaging engine such as Apache ActiveMQ. When deploying your application in a full Java EE application server, you may also want to configure Gravity to integrate with the built-in messaging engine of your application server (such as HornetQ in JBoss AS 7).

Here is a sample configuration for a default JBoss installation with a brief description of the different options:

```
<adapters>
  <adapter-definition id="jms" class="org.granite.gravity.adapters.JMSServiceAdapter"/>
</adapters>

<destination id="chat-jms">
  <properties>
    <jms>
      <destination-type>Topic</destination-type>
      <!-- Optional: forces usage of simple text messages
      <message-type>javax.jms.TextMessage</message-type>
      -->
    </jms>
  </properties>
  <connection-factory>ConnectionFactory</connection-factory>
</destination>
```

```

<destination-jndi-name>topic/testTopic</destination-jndi-name>
<destination-name>TestTopic</destination-name>
<acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
<transacted-sessions>>false</transacted-sessions>
<!-- Optional JNDI environment. Specify the external JNDI configuration to access
a remote JMS provider. Sample for a remote JBoss server.
-->
<initial-context-environment>
  <property>
    <name>Context.SECURITY_PRINCIPAL</name>
    <value>guest</value>
  </property>
  <property>
    <name>Context.SECURITY_CREDENTIALS</name>
    <value>guest</value>
  </property>
  <property>
    <name>Context.PROVIDER_URL</name>
    <value>http://my.host.com:1099</value>
  </property>
  <property>
    <name>Context.INITIAL_CONTEXT_FACTORY</name>
    <value>org.jnp.interfaces.NamingContextFactory</value>
  </property>
  <property>
    <name>Context.URL_PKG_PREFIXES</name>
    <value>org.jboss.naming:org.jnp.interfaces</value>
  </property>
</initial-context-environment>
</jms>
...
</properties>
...
<adapter ref="jms"/>
</destination>

```

Comments on configuration options:

- destination-type must be Topic for the moment. Queues may be supported later.
- message-type may be forced to simple text messages by specifying `javax.jms.TextMessage`.
- connection-factory and destination-jndi-name are the JNDI names respectively of the JMS ConnectionFactory and of the JMS topic.

- `destination-name` is just a label but still required.
- `acknowledge-mode` can have the standard values accepted by any JMS provider: `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`.
- `transacted-sessions` allows the use of transactions in sessions when set to `true`.
- `initial-context-environment`: The `initial-context` parameters allow to access a remote JMS server by setting the JNDI context options.



### Note

The JMS headers are always copied between client and JMS messages



### Note

Durable subscriptions are not yet supported

## 6.5. Using an Embedded ActiveMQ

In the case of a simple Tomcat/Jetty installation without JMS provider, or to allow client-to-client messaging with advanced capabilities such as durable messages, Gravity can be integrated with an embedded Apache ActiveMQ instance.

To enable ActiveMQ, just put the `activemq-xx.jar` in your `WEB-INF/lib` directory. The necessary topic will be lazily created on first use, except if the property `create-broker` is set to `false`. The uri of the created ActiveMQ broker will be `vm://adapterId`.

Here is a sample configuration to use an embedded ActiveMQ provider:

```
<adapters>
  <adapter-definition
    id="activemq"
    class="org.granite.gravity.adapters.ActiveMQServiceAdapter"/>
</adapters>

<destination id="chat-activemq">
  <properties>
    <jms>
      <destination-type>Topic</destination-type>
      <!-- Optional: forces usage of simple text messages
      <message-type>javax.jms.TextMessage</message-type>
```

```

-->
<connection-factory>ConnectionFactory</connection-factory>
<destination-jndi-name>topic/testTopic</destination-jndi-name>
<destination-name>TestTopic</destination-name>
<acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
<transacted-sessions>>false</transacted-sessions>
</jms>

<server>
  <durable>>true</durable>
  <file-store-root>/var/activemq/data</file-store-root>
  <create-broker>>true</create-broker>
  <wait-for-start>>false</wait-for-start>
</server>
</properties>
...
<adapter ref="activemq"/>
</destination>

```

And a sample configuration to use an external ActiveMQ provider:

```

<adapters>
  <adapter-definition
    id="activemq"
    class="org.granite.gravity.adapters.ActiveMQServiceAdapter"/>
</adapters>

<destination id="chat-activemq">
  <properties>
    <jms>
      <destination-type>Topic</destination-type>
      <!-- Optional: forces usage of simple text messages
      <message-type>javax.jms.TextMessage</message-type>
    -->
      <connection-factory>ConnectionFactory</connection-factory>
      <destination-jndi-name>topic/testTopic</destination-jndi-name>
      <destination-name>TestTopic</destination-name>
      <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
      <transacted-sessions>>false</transacted-sessions>
    </jms>
  </properties>
</destination>

```

```
<server>
  <broker-url>tcp://activemq-server:61616</broker-url>
</server>
</properties>
...
<adapter ref="activemq"/>
</destination>
```

Comments on configuration options:

- The main parameters (<jms>...</jms>) are identical to those used in the default JMS configuration. See [above](#).
- durable, if set to true, allows for durable messages, stored in the filesystem. The data store directory of ActiveMQ can be specified by the file-store-root parameter.
- create-broker is optional, as well as the dependant wait-for-start attribute. When create-broker is false, creation of the broker is not automatic and has to be done by the application itself. In this case, wait-for-start set to true tells the ActiveMQConnectionFactory to wait for the effective creation of the broker. Please refer to the ActiveMQ documentation for more details on these options.

### 6.6. Server to client publishing

There are mostly two kinds of requirements for messaging: client-to-client interactions, that can be easily handled by the Consumer/Producer pattern, and server-to-client push that can be done with either the low-level Gravity API or directly using the JMS API when the JMS adapter is used.

**Server to client messaging with the low-level Gravity API.** If you use the SimpleAdapter, the message sending will have to be done at a lower level and you will need a compilation dependency on the Gravity API. It's also possible but not recommended to use this low-level API with the JMS and ActiveMQ adapters. It first requires to get the Gravity object from the ServletContext. It is set as an attribute named org.granite.gravity.Gravity. When using Spring, Seam 2 or CDI, you can also get this object by injection (see the corresponding documentation). Then you can send messages of type flex.messaging.messages.Message by calling the method gravity.publish(message);.

```
Gravity gravity = GravityManager.getGravity(servletContext);
AsyncMessage message = new AsyncMessage();
message.setDestination("my-gravity-destination");
message.setHeader(AsyncMessage.SUBTOPIC_HEADER, "my-topic");
message.setBody("Message content");
gravity.publishMessage(message);
```

If you need to simulate a publish from the client subscribed in the current session, you can get the `clientId` in the session attribute named `org.granite.gravity.channel.clientId`. `{destination}` and set it in the message.

**Server to Client Messaging with JMS.** Sending messages from the server to clients simply consists of sending JMS messages to the corresponding JMS topic. Text messages are received as simple text on the client side, object messages are serialized in AMF3 and deserialized and received as typed objects. The Gravity messaging channel supports lazily loaded collections and objects, exactly as the remoting channel.

Here is an example on an EJB3 sending a message:

```
@Stateless
@Local(Test.class)
public class TestBean implements Test {

    @Resource
    SessionContext ctx;

    @Resource(mappedName="java:/ConnectionFactory")
    ConnectionFactory jmsConnectionFactory;

    @Resource(mappedName="topic/testTopic")
    Topic jmsTopic;

    public TestBean() {
        super();
    }

    public void notifyClient(Object object) {
        try {
            Connection connection = jmsConnectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            javax.jms.Message jmsMessage = session.createObjectMessage(person);
            MessageProducer producer = session.createProducer(jmsTopic);
            producer.send(jmsMessage);
            session.close();
            connection.close();
        }
        catch (Exception e) {
```

```
        log.error("Could not publish notification", e);
    }
}
```

Here is an example on a Seam 2 component sending a message:

```
@Stateless
@Local(Test.class)
@Name("test")
public class TestBean implements Test {

    private static Logger log = Logger.getLogger(TestBean.class.getName());

    @In
    private TopicPublisher testTopicPublisher;
    @In
    private TopicSession topicSession;

    public void notifyClient(Serializable object) {
        try {
            testTopicPublisher.publish(topicSession.createObjectMessage(object));
        }
        catch (Exception e) {
            log.error("Could not publish notification", e);
        }
    }
}
```

**Server to client messaging with Embedded ActiveMQ.** The only difference with standard JMS is that you can get a `ConnectionFactory` more easily. Also ActiveMQ supports subtopics. The name of the topic is built with the following rule:

- Without subtopic, the name of the ActiveMQ destination should be the same as defined in the `jms/destination-name` configuration parameter.
- With subtopic, the name is the concatenation of the `destination-name` parameter with the subtopic. Wildcards are supported in the subtopic following Flex convention and are converted to the ActiveMQ format (see [here](http://activemq.apache.org/wildcards.html) [http://activemq.apache.org/wildcards.html]), meaning that `toto.**` is converted to `toto.>`.

```

public class Test throws JMSException {
    // adapterId should be the id of the JMS adapter as defined in services-config.xml
    ConnectionFactory f = new ActiveMQConnectionFactory("vm://adapterId");
    Connection connection = jmsConnectionFactory.createConnection();
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

    ActiveMQTopic activeMQTopic= new ActiveMQTopic("destination");
    javax.jms.Message jmsMessage = session.createObjectMessage(person);
    MessageProducer producer = session.createProducer(activeMQTopic);
    producer.send(jmsMessage);

    session.close();
    connection.close();
}

```

## 6.7. Securing Messaging Destinations

Securing messaging destination is very similar to security remoting destinations (see [here](#)) and most concepts apply to messaging services as well as remoting services.

You can for example setup role-based security on a Gravity destination with the following definition in `services-config.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <service id="messaging-service"
      class="flex.messaging.services.MessagingService"
      messageTypes="flex.messaging.messages.AsyncMessage">
      <adapters>
        <adapter-definition
          id="default"
          class="org.granite.gravity.adapters.SimpleServiceAdapter"
          default="true"/>
      </adapters>

      <destination id="restrictedTopic">
        <channels>
          <channel ref="my-gravityamf"/>

```

```
</channels>
<security>
  <security-constraint>
    <auth-method>Custom</auth-method>
    <roles>
      <role>admin</role>
    </roles>
  </security-constraint>
</security>
</destination>
</service>
</services>
...
</services-config>
```

In this case, only users with the role `admin` will be able to subscribe to the topic `restrictedTopic`.

**Fine-grained per-destination security.** You may write and configure a specific `GravityDestinationSecurizer` in order to add fine grained security checks for specific actions. In particular you can control who can subscribe or publish messages to a particular topic.

```
public interface GravityDestinationSecurizer extends DestinationSecurizer {

    public void canSubscribe(GravityInvocationContext context)
        throws SecurityServiceException;
    public void canPublish(GravityInvocationContext context)
        throws SecurityServiceException;
}
```

You then have to tell GraniteDS where to use your securizer:

```
<services-config>
  <services>
    <service ...>
      <destination id="restrictedDestination">
        ...
      <properties>
        <securizer>path.to.MyDestinationSecurizer</securizer>
      </properties>
    </service>
  </services>
</services-config>
```

```

        </properties>
    </destination>
</service>
</services>
...
</services-config>

```

Your custom implementation of this interface is expected to throw a `SecurityServiceException` when the user has no right to execute the requested action (subscription or publishing). You can also override the subscription message in the method `canSubscribe` if for example you want to force a particular subtopic or selector depending on the user access rights and not only rely on the client to define the subscription parameters.

```

public class CustomDestinationSecurizer implements GravityDestinationSecurizer {

    public void canSubscribe(GravityInvocationContext context) throws SecurityServiceException {
        String profile = getProfileForCurrentUser();
        if (profile.equals("limited"))
            throws new SecurityServiceException("Access denied");

        if (profile.equals("restricted"))
            ((CommandMessage)context.getMessage()).getHeaders().put("DSSubtopic", "forcedCustomTopic");
    }

    public void canPublish(GravityInvocationContext context) throws SecurityServiceException {
        String profile = getProfileForCurrentUser();
        if (profile.equals("limited"))
            throws new SecurityServiceException("Access denied");
    }
}

```

If you have configured a security service, the current thread has already been authenticated at this point, so you are able to get user information depending your security implementation. For example, with Spring Security, you can use `SecurityContextHolder.getContext().getAuthentication()`.



## Integration with EJB3

EJB 3 are an important part of the [Java EE 5 platform](http://www.oracle.com/technetwork/java/javasee/tech/javasee5-jsp-135162.html) [http://www.oracle.com/technetwork/java/javasee/tech/javasee5-jsp-135162.html]. They provide a powerful framework for managing and securing enterprise services in an application server (session beans) as well as an powerful persistence and query language system (JPA).

GraniteDS provides access to EJB 3 services via either the RemoteService API or the Tide API for Session Beans methods calls, and fully supports serialization of JPA entities from and to your Java client application, taking care of lazily loaded associations; both collections and proxies. This support for JPA entity beans is covered in the section [JPA and lazy initialization](#), so this section will only describe how to call remotely stateless and stateful session beans from a Java client application. GraniteDS also integrates with container security for authentication and role-based authorization.

### 7.1. Using the RemoteService API

The client-side usage of the RemoteService API is completely independent of the server technology, so everything described in the [Remoting](#) chapter applies for EJBs. This section will only describe the particular configuration required in various use cases of EJB services.

Configuring remoting for EJB 3 services simply requires adding the `org.granite.messaging.service.EjbServiceFactory` service factory in `services-config.xml` and specifying its JNDI lookup string property.

#### 7.1.1. Basic Remoting Example

All remoting examples from the [Remoting](#) chapter apply for EJBs, here is a basic example:

```
public interface HelloService {

    public String hello(String name);
}

@Stateless
@Local(HelloService.class)
@RemoteDestination(id="helloService")
public class HelloServiceBean implement HelloService {

    public String hello(String name) {
        return "Hello " + name;
    }
}
```

```
AMFRemotingChannel channel = new AMFRemotingChannel(transport, "graniteamf",
    new URI("http://localhost:8080/helloworld/graniteamf/amf.txt"));
RemoteService srv = new RemoteService(channel, "hello");

srv.newInvocation("hello", "Barack").setTimeToLive(5, TimeUnit.SECONDS)
    .addListener(new ResultFaultIssuesResponseListener() {

        @Override
        public void onResult(ResultEvent event) {
            System.out.println("Result: " + event.getResult());
        }

        @Override
        public void onFault(FaultEvent event) {
            System.err.println("Fault: " + event.toString());
        }

        @Override
        public void onIssue(IssueEvent event) {
            System.err.println("Issue: " + event.toString());
        }
    }).invoke();
```

### 7.1.2. Common configuration

The main part of the configuration is the factory declaration in the file `services-config.xml` :

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

<services-config>

    <services>
        <service
            id="granite-service"
            class="flex.messaging.services.RemotingService"
            messageTypes="flex.messaging.messages.RemotingMessage">
```

```

<destination id="personService">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
  <properties>
    <factory>ejbFactory</factory>
  </properties>
</destination>
</service>
</services>

<factories>
  <factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
    <properties>
      <lookup>myapp.ear/{capitalized.destination.id}Bean/local</lookup>
    </properties>
  </factory>
</factories>

<channels>
  <channel-definition id="my-graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>

```

Two elements are important in this configuration :

- The EJB service factory declaration and the reference to it in our destination
- The JNDI lookup string defined in the lookup property of the factory



### Note

In Java EE 6 compliant application servers such as JBoss 6 and GlassFish 3, you can use the standard global naming specification : `java:global/{context.root}/{capitalized.destination.id}Bean`.

The JNDI lookup string is common for all EJB 3 destinations, and thus contains placeholders that will be replaced at runtime depending on the destination that is called.

{capitalized.destination.id} will be replaced by the destination id with the first letter in capital, for example personService will become myApp/PersonServiceBean/local. {destination.id} can alternatively be used. Note that some Java EE servers do not expose EJB local interfaces in the global JNDI context, so you will have to use a local JNDI reference and add an ejb-local-ref section in web.xml for each EJB exposed to JNDI.

```
<ejb-local-ref>
  <ejb-ref-name>myapp.ear/PeopleServiceBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home/>
  <local>com.myapp.service.PeopleService</local>
</ejb-local-ref>
```

```
<factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
  <properties>
    <lookup>java:comp/env/myapp.ear/{capitalized.destination.id}Bean</lookup>
  </properties>
</factory>
```

Of course you can share the same factory with many EJB destinations.

```
<destination id="person">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
  <properties>
    <factory>ejbFactory</factory>
  </properties>
</destination>

<destination id="product">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
  <properties>
```

```

    <factory>ejbFactory</factory>
  </properties>
</destination>

```

### 7.1.3. Configuration for Remote EJBs

By default GraniteDS will lookup the bean in JNDI with the default `InitialContext`. To access remote EJB services you have to specify the JNDI context environment that will be used for remote lookup in the factory definition of `services-config.xml`.

The parameters generally depend on the remote application server. Please refer to the standard [JNDI Context API documentation](http://java.sun.com/j2se/1.5.0/docs/api/javax/naming/Context.html) [http://java.sun.com/j2se/1.5.0/docs/api/javax/naming/Context.html] and to the documentation of your application server for more details.

```

...
<factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
  <properties>
    <lookup>myApp/{capitalized.destination.id}Bean/local</lookup>

    <!-- InitialContext parameters -->
    <initial-context-environment>
      <property>
        <name>Context.PROVIDER_URL</name>
        <value>...</value>
      </property>
      <property>
        <name>Context.INITIAL_CONTEXT_FACTORY</name>
        <value>...</value>
      </property>
      <property>
        <name>Context.URL_PKG_PREFIXES</name>
        <value>...</value>
      </property>
      <property>
        <name>Context.SECURITY_PRINCIPAL</name>
        <value>...</value>
      </property>
      <property>
        <name>Context.SECURITY_CREDENTIALS</name>
        <value>...</value>
      </property>
    </initial-context-environment>
  </properties>
</factory>

```

```
</properties>
</factory>
...
```

For JBoss Application Server for example this declaration looks like this:

```
...
<factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
  <properties>
    <lookup>myApp/{capitalized.destination.id}Bean/local</lookup>

    <!-- InitialContext parameters -->
    <initial-context-environment>
      <property>
        <name>Context.PROVIDER_URL</name>
        <value>jnp://remotehostname:1099</value>
      </property>
      <property>
        <name>Context.INITIAL_CONTEXT_FACTORY</name>
        <value>org.jnp.interfaces.NamingContextFactory</value>
      </property>
      <property>
        <name>Context.URL_PKG_PREFIXES</name>
        <value>org.jboss.naming:org.jnp.interfaces</value>
      </property>
    </initial-context-environment>
  </properties>
</factory>
...
```

### 7.1.4. Automatic Configuration of EJB Destinations

This is annoying to have to declare each and every EJB exposed to Flex remoting in `services-config.xml`. To avoid this step, it is possible to instruct GraniteDS to search EJB services in the application classpath.

Note however that this cannot work with remote EJBs as GraniteDS will obviously not have access to the remote classpath.

To enable automatic destination discovery, you simply have to enable the scan property in `granite-config.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC
    "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config scan="true">
    ...
</granite-config>
```

Then you have to add a simple marker file (even empty) `META-INF/services-config.properties` in every EJB jar (or in `WEB-INF/classes` if you use EJB 3.1 packaged in a war). Then GraniteDS will scan these jars at startup and look for EJB classes annotated with `@RemoteDestination`. The annotation can be put either on the EJB interface or on the EJB implementation, but it's recommended to put it on the EJB interface.

```
@Stateless
@Local(PersonService.class)
@RemoteDestination(id="person", securityRoles={"user","admin"})
public class PersonServiceBean implements PersonService {
    ...
}
```

The `@RemoteDestination` annotation additionally supports the following attributes:

- `id` is mandatory and is the destination name
- `service` is optional if there is only one service for `RemotingMessage` defined in `services-config.xml`. Otherwise this should be the name of the service.
- `channel` is optional if there is only one channel defined in `services-config.xml`. Otherwise this should be the id of the target channel.
- `channels` may be used instead of `channel` to define a failover channel.
- `factory` is optional if there is only one factory in `services-config.xml`. Otherwise this should be the factory id.

- `securityRoles` is an array of role names for securing the destination.

As shown below, the `service`, `factory` and `channel` sections are still required in your `services-config.xml` file, but the `service` part will not contain any destination. So, with any number of EJBs annotated this way, the `services-config.xml` file may be defined as follows:

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

<services-config>

  <services>
    <service
      id="granite-service"
      class="flex.messaging.services.RemotingService"
      messageTypes="flex.messaging.messages.RemotingMessage">
      <!-- no need to declare destinations here -->
    </service>
  </services>

  <factories>
    <factory id="ejbFactory" class="org.granite.messaging.service.EjbServiceFactory">
      <properties>
        <lookup>myApp/{capitalized.destination.id}Bean/local</lookup>
      </properties>
    </factory>
  </factories>

  <channels>
    <channel-definition id="my-graniteamf" class="mx.messaging.channels.AMFChannel">
      <endpoint
        uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
      </channel-definition>
    </channels>

</services-config>
```

As the destinations are not defined in `services-config.xml` any more, you will have to setup the `RemoteObject` endpoint manually in ActionScript (see [here](#) for details).

### 7.1.5. Configuration for Stateful EJBs

Most of what has been described for stateless beans also applies for stateful beans, however stateful beans have a different lifecycle.

GraniteDS stores the reference of stateful EJBs retrieved from JNDI in the HTTP session so it can keep the correct instance between remote calls. Take care that the timeout for HTTP session expiration should be consistent with the timeout for EJB3 stateful beans expiration.

GraniteDS has to know a bit more information about stateful beans than for stateless beans, here is an example of `services-config.xml` for the following EJB:

```
package com.myapp.services;

import javax.ejb.Local;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
@Local(PositionService.class)
public class PositionServiceBean implements PositionService {

    int x = 300;

    public int getX() {
        return x;
    }

    public void saveX(int x) {
        this.x = x;
    }

    @Remove
    public void remove() {
    }
}
```

```
<destination id="position">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
</destination>
```

```
</channels>
<properties>
  <factory>ejbFactory</factory>

  <!-- Specific for stateful beans -->
  <ejb-stateful>
    <remove-method>
      <signature>remove</signature>
      <retain-if-exception>false</retain-if-exception>
    </remove-method>
  </ejb-stateful>
</properties>
</destination>
```

The configuration of the destination is similar to the one used for stateless beans, except for the additional `ejb-stateful` subsection. The presence of this `ejb-stateful` node, even empty, informs GDS that this EJB 3 is stateful and should be managed as such. Otherwise, the bean will be considered stateless and only one instance will be shared between all users.

The inner `remove-method` node contains information about the `remove()` methods of your stateful bean:

- **signature:** This is the name of the method, optionally followed by a parameter list. If your `remove()` method has arguments, the signature should follow the conventions used in `java.lang.reflect.Method.toString()`. For example, with the following `remove()` method:

```
@Remove
public int remove(boolean arg1, Integer arg2, String[] arg3) {...}
```

... you should write this signature:

```
<signature>remove(boolean,java.lang.Integer,java.lang.String[])</signature>
```

- **retain-if-exception (optional):** This is the equivalent of the `@Remove` annotation attribute; the default is `false`.

You may of course add multiple `remove-method` nodes in the same `ejb-stateful` node if necessary.

When using automatic configuration with classpath scanning, stateful EJBs are automatically detected with the `@Stateful` annotation and properly configured.

### 7.1.6. Security

You can easily protect access to your EJB destinations with destination-based security. Please refer to the [security chapter](#).

GraniteDS will then pass the user credentials from the client `RemoteChannel` to the EJB security context, making possible to use role-based authorization with the EJB destination.

Here is an example configuration in `services-config.xml`:

```
<destination id="personService">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
  <properties>
    <factory>ejbFactory</factory>
  </properties>
  <security>
    <security-constraint>
      <auth-method>Custom</auth-method>
      <roles>
        <role>user</role>
        <role>admin</role>
      </roles>
    </security-constraint>
  </security>
</destination>
```

```
@Stateless
@Local(PersonService.class)
public class PersonServiceBean implements PersonService {

  @PersistenceContext
  protected EntityManager manager;

  public List<Person> findAllPersons() {
    return manager.createQuery("select distinct p from Person p").getResultList();
  }
}
```

```
@RolesAllowed({"admin"})
public Person createPerson(Person person) {
    return manager.merge(person);
}

@RolesAllowed({"admin"})
public Person modifyPerson(Person person) {
    return manager.merge(person);
}

@RolesAllowed({"admin"})
public void deletePerson(Person person) {
    person = manager.find(Person.class, person.getId());
    manager.remove(person);
}
}
```

With this configuration, only authenticated users having either the user or admin roles will be able to call the EJB remotely from the client. Then the EJB container will enforce the particular access on each method due to the `@RolesAllowed` annotation and may throw a `EJBAccessException`.

## 7.2. Using the Tide API

Most of what is described in the [Tide Remoting](#) section applies for EJB 3, however GraniteDS also provides an improved integration with EJB 3 services.

### 7.2.1. Configuration

There are a few noticeable differences in the configuration in this case.

- It is *mandatory* to use automatic classpath scanning as Tide needs to have access to the actual implementation of the EJB and not only to its interface. Consequently this is currently not possible to use remote EJBs as Tide-enabled destinations.
- You can define in the `tide-annotations` section of `granite-config.xml` the conditions used to enable remote access to EJB destinations (for example all EJBs annotated with a particular annotation).
- You have to configure the specific Tide/EJB3 `org.granite.tide.ejb.EjbServiceFactory` service factory in `services-config.xml`.
- You have to configure a unique Tide/EJB3 destination named `ejb` in `services-config.xml`

- You have to retrieve the Tide context in Flex with `Ejb.getInstance().getEjbContext()` instead of `Tide.getInstance().getContext()`.

Here is a default configuration suitable for most cases:

```
<granite-config scan="true">
  ...

  <tide-components>
    <tide-component annotated-
with="org.granite.messaging.service.annotations.RemoteDestination"/>
  </tide-components>

</granite-config>
```

```
<services-config>

  <services>
    <service id="granite-service"
      class="flex.messaging.services.RemotingService"
      messageTypes="flex.messaging.messages.RemotingMessage">
      <!--
      ! Use "tideEjbFactory" and "my-graniteamf" for "ejb" destination (see below).
      ! The destination must be "ejb" when using Tide with default configuration.
      !-->
      <destination id="ejb">
        <channels>
          <channel ref="my-graniteamf"/>
        </channels>
        <properties>
          <factory>tideEjbFactory</factory>
          <entity-manager-factory-jndi-name>java:/DefaultEMF</entity-manager-factory-jndi-
name>
        </properties>
      </destination>
    </service>
  </services>

  <!--
  ! Declare tideEjbFactory service factory.
```

```
!-->
<factories>
  <factory id="tideEjbFactory" class="org.granite.tide.ejb.EjbServiceFactory">
    <properties>
      <lookup>myapp.ear/{capitalized.component.name}Bean/local</lookup>
    </properties>
  </factory>
</factories>

<!--
! Declare my-graniteamf channel.
!-->
<channels>
  <channel-definition id="my-graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>

</services-config>
```

The destination named `ejb` will be the one and only destination required for all EJB destinations.

The property `lookup` of the factory defines the lookup string used by Tide to lookup the EJBs in JNDI. The example above is suitable for JBoss, please refer to your application server documentation for other servers. Placeholders can be defined in this lookup string that will be replaced at runtime for each EJB: `{capitalized.component.name}` is the name used on the client.



### Note

In Java EE 6 compliant application servers such as JBoss 6 and GlassFish 3, you can use the standard global naming specification : `java:global/{context.root}/{capitalized.component.name}Bean`.



### Note

In many JEE servers (GlassFish v2 for example, but not JBoss), EJB local interfaces are not published in the global JNDI. To be able to call them through

Tide, you will have to specify `ejb-local-ref` definitions for each EJB in `web.xml` and use a `java:comp/env` local JNDI name.

```
<ejb-local-ref>
  <ejb-ref-name>myapp/PeopleServiceBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home/>
  <local>com.myapp.service.PeopleService</local>
</ejb-local-ref>
```

```
<factory id="tideEjbFactory" class="org.granite.tide.ejb.EjbServiceFactory">
  <properties>
    <lookup>java:comp/env/myapp/{capitalized.component.name}Bean</lookup>
  </properties>
</factory>
```

The property `entity-manager-factory-name` is necessary only when using transparent remote lazy loading of collections. It should be the JNDI name that GraniteDS can use to lookup the `EntityManagerFactory` in JNDI. Alternatively you can instead specify `entity-manager-name`, then GraniteDS will lookup for an `EntityManager`. JBoss server can expose these two elements in the global JNDI by adding these lines in `persistence.xml`:

```
<persistence-unit name="ejb-pu">
  ...
  <properties>
    ...
    <property name="jboss.entity.manager.factory.jndi.name" value="java:/DefaultEMF"/>
    <property name="jboss.entity.manager.jndi.name" value="java:/DefaultEM"/>
  </properties>
</persistence-unit>
```

For other application servers that does not expose the persistence unit in JNDI, you will have to use a local name and add `persistence-unit-ref` in `web.xml`.

```
<persistence-unit-ref>
  <persistence-unit-ref-name>ejb-pu</persistence-unit-ref-name>
</persistence-unit-ref>
```

```
<destination id="ejb">
  <channels>
    <channel ref="graniteamf"/>
  </channels>
  <properties>
    <factory>tideEjbFactory</factory>
    <entity-manager-factory-jndi-name>java:comp/env/ejb-pu</entity-manager-factory-jndi-name>
  </properties>
</destination>
```

### 7.2.2. Basic remoting with dependency injection

When using EJB3, the only difference on the client is that you have to use the destination named `ejb` to build the `ServerSession`. Here is a simple example of remoting with an Spring-injected client proxy for an EJB service:

```
public class HelloController {

    @Inject @Qualifier("helloService")
    private Component helloService;

    public void hello(String to) {
        // Asynchronous call using handlers
        helloService.call("hello", to, new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }
        });
    }
}
```

```

    }

    @Override
    public void fault(TideFaultEvent fault) {
        System.err.println("Fault: " + fault.getFault());
    }
};
}

public String helloSync(String to) {
    // Synchronous wait of Future result
    Future<String> futureResult = helloService.call("hello", to);
    String result = futureResult.get();
    System.out.println("Sync result: " + result);
    return result;
}
}

```

If you have generated typed client proxies, it can be further simplified to something like this:

```

public class HelloController {

    @Inject
    private HelloService helloService;

    public void hello(String to) {
        // Asynchronous call using handlers
        helloService.hello(to, new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }
        });

        @Override
        public void fault(TideFaultEvent fault) {
            System.err.println("Fault: " + fault.getFault());
        }
    };
}

public String helloSync(String to) {

```

```
// Synchronous wait of Future result
Future<String> futureResult = helloService.hello(to);
String result = futureResult.get();
System.out.println("Sync result: " + result);
return result;
}
}
```

This is almost identical to the standard Tide API described in the [Tide remoting](#) section, and all other methods apply for EJB.

### 7.2.3. Typesafe remoting with dependency injection

You can benefit from the capability of the Gfx code generator (see [here](#)) to generate a strongly typed Java client proxy from the EJB3 interface when it is annotated with `@RemoteDestination`. In this case, you can inject a typesafe reference to your service and get better compile time error checking and auto completion in your IDE:

```
public class HelloController {

    @Inject @Qualifier("helloService")
    private HelloService helloService;

    // Asynchronous call using handlers
    helloService.hello("Barack", new TideResponder<String>() {
        @Override
        public void result(TideResultEvent<String> result) {
            System.out.println("Async result: " + result.getResult());
        }

        @Override
        public void fault(TideFaultEvent fault) {
            System.err.println("Fault: " + fault.getFault());
        }
    });

    // Synchronous wait of Future result
    Future<String> futureResult = helloService.hello("Barack");
    String result = futureResult.get();
    System.out.println("Sync result: " + result);
}
```

Note that as there is only one instance of `HelloService`, you may also omit the `Qualifier` annotation and use typesafe injection with `@Inject` only.

---

## Integration with Spring

The [Spring framework](http://www.springframework.org) [http://www.springframework.org] is one of the most popular Java enterprise frameworks. It integrates on a common platform all the necessary services for building enterprise applications: persistence, transactions, security...

GraniteDS provides out-of-the-box integration with Spring 2.5+ and 3.0+ via either the RemoteService API or the Tide API to remotely call Spring services, and fully supports serialization of JPA entities from and to your Java client application, taking care of lazily loaded associations. The support for JPA entity beans is covered in the section [JPA and lazy initialization](#), so this section will only describe how to call Spring beans from a Java application. GraniteDS also fully supports Acegi Security / Spring Security 2.x / Spring Security 3.x.

The support for Spring is included in the library `granite-spring.jar`, so you always have to include this library in either `WEB-INF/lib` or `lib` for an ear packaging.

Note that to provide a more native experience for Spring developers, the Spring support in GraniteDS can be configured directly in the Spring configuration files (`applicationContext.xml`). Most features of GraniteDS can be configured this way, and it is still possible to fall back to the default GraniteDS configuration files `services-config.xml` and `granite-config.xml` for unsupported features.

### 8.1. Spring MVC setup

It is perfectly possible to use the default setup for the GraniteDS servlet in `web.xml`, but the recommended way when using Spring is to configure a Spring MVC dispatcher servlet and handle incoming AMF requests. This will in particular allow configuring GraniteDS in the Spring application context. You also need to setup the Spring request and application listeners but this is standard Spring configuration. Note that this works only for the remoting servlet, but you still have to configure the Gravity servlet in the default way because the Spring MVC dispatcher servlets cannot support non blocking I/O.

```
<!-- Path to Spring config file -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/conf/application-context.xml
  </param-value>
</context-param>

<!-- Spring application listener -->
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
```

```
</listener>

<!-- Spring listener for web-scopes (request, session) -->
<listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-
class>
</listener>

<!-- Spring MVC dispatcher servlet for AMF remoting requests -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/graniteamf/*</url-pattern>
</servlet-mapping>
```

You also have to add an empty file `WEB-INF/dispatcher-servlet.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
</beans>
```

## 8.2. Using the RemoteService API

The client-side usage of the RemoteService API is completely independent of the server technology, so everything described in the [Remoting](#) chapter applies for Spring beans. This section will only describe the particular configuration required in various use cases of Spring services.

### 8.2.1. Basic remoting example

All remoting examples from the [Remoting](#) chapter apply for Spring beans, here is a basic example with an annotated Spring service:

```
public interface HelloService {

    public String hello(String name);
}

@Service("helloService")
@RemoteDestination(id="helloService", source="helloService")
public class HelloServiceImpl implement HelloService {

    public String hello(String name) {
        return "Hello " + name;
    }
}
```

```
AMFRemotingChannel channel = new AMFRemotingChannel(transport, "graniteamf",
    new URI("http://localhost:8080/helloworld/graniteamf/amf.txt"));
RemoteService srv = new RemoteService(channel, "helloService");

srv.newInvocation("hello", "Barack").setTimeToLive(5, TimeUnit.SECONDS)
    .addListener(new ResultFaultIssuesResponseListener() {

        @Override
        public void onResult(ResultEvent event) {
            System.out.println("Result: " + event.getResult());
        }

        @Override
        public void onFault(FaultEvent event) {
            System.err.println("Fault: " + event.toString());
        }

        @Override
        public void onIssue(IssueEvent event) {
            System.err.println("Issue: " + event.toString());
        }
    }).invoke();
```

### 8.2.2. Configuration with a MVC setup

Besides configuring the dispatcher servlet (see [here](#)), configuring GraniteDS in the Spring context just requires adding the `graniteds` namespace and adding a `server-filter` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:graniteds="http://www.graniteds.org/config"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.0.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/aop http://www.springframework.org/schema/aop/
spring-aop-3.0.xsd
    http://www.graniteds.org/config http://www.graniteds.org/public/dtd/3.0.0/granite-
config-3.0.xsd">

  ...

  <graniteds:server-filter url-pattern="/*"/>

</beans>
```

The actual url that will be listened to by the AMF processor is the combination of the `url-pattern` in the Spring context and the `servlet-mapping` of the dispatcher servlet in `web.xml`. The configuration described here maps GraniteDS on `/graniteamf/*` and is suitable in almost all cases.

When necessary, this configuration can be overridden or completed by the default configuration in `services-config.xml` described in the [next section](#). In this case, the implicit configuration created by the MVC setup contains the following elements :

- a remoting service named `granite-service`

- a remoting service factory named `spring-factory`
- a remoting channel named `graniteamf`

The MVC setup automatically enables component scanning, so you can just annotate your Spring services with `@RemoteDestination` and put an empty `META-INF/services-config.properties` file in your services jar or folder to tell GraniteDS where to look for services. See last paragraph [Automatic Configuration of Destinations](#).

Alternatively you can also declare the remote destinations manually in the Spring context:

```
<graniteds:remote-destination id="personService" source="personService"/>
```

You can also specify a secure destination by adding the list of roles required to access the destination:

```
<graniteds:remote-destination id="personService" source="personService">
  <graniteds:roles>
    <graniteds:role>ROLE_ADMIN</graniteds:role>
  </graniteds:roles>
</graniteds:remote-destination>
```

The support for Spring Security is automatically enabled when Spring Security is installed in the application and the version of Spring Security is automatically detected. However if you have configured more than one `AuthenticationManagers`, it will be necessary to instruct GraniteDS which one should be used for authentication by adding the following line to your Spring configuration :

```
<graniteds:security-service authentication-manager="myAuthenticationManager"/>
```

With this declaration you can also provide various configuration elements for the Spring 3 security service implementation :

```
<graniteds:security-service
  authentication-manager="myAuthenticationManager"
```

```
allow-anonymous-access="true"
authentication-trust-resolver="com.myapp.MyAuthenticationTrustResolver"
session-authentication-strategy="com.myapp.MySessionAuthenticationStrategy"
security-context-repository="com.myapp.MySecurityContextRepository"
security-interceptor="com.myapp.MySecurityInterceptor"
password-encoder="com.myapp.MyPasswordEncoder"
/>
```

### 8.2.3. Default configuration

Configuring remoting for Spring services simply requires using the `org.granite.spring.SpringServiceFactory` service factory in `services-config.xml`:

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

<services-config>
  <services>
    <service
      id="granite-service"
      class="flex.messaging.services.RemotingService"
      messageTypes="flex.messaging.messages.RemotingMessage">
      <destination id="testBean">
        <channels>
          <channel ref="graniteamf"/>
        </channels>
        <properties>
          <factory>springFactory</factory>
          <source>springBean</source>
        </properties>
        <security>
          <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
              <role>ROLE_USER</role>
              <role>ROLE_ADMIN</role>
            </roles>
          </security-constraint>
        </security>
      </destination>
    </service>
  </services>
```

```

<factories>
  <factory id="springFactory" class="org.granite.spring.SpringServiceFactory" />
</factories>

<channels>
  <channel-definition id="graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>

```

The only thing that should be noted for Spring destinations is that you have to specify a source property specifying the name of the remote Spring bean.

### 8.2.4. Automatic configuration of destinations

It is possible to instruct GraniteDS to automatically search for Spring destinations in the classpath by:

- Enabling scanning in `granite-config.xml` (scanning is always enabled with a MVC setup).

```
<granite-config scan="true"/>
```

- Adding an empty `META-INF/services-config.properties` marker file in all jars containing Spring services
- Annotating the Spring service (or preferably its interface) with `org.granite.messaging.service.annotations.RemoteDestination`

```

@RemoteDestination(id="personService", source="personService", securityRoles={"user","admin"})
public interface PersonService {
}

@Service("personService")

```

```
public class PersonServiceBean implements PersonService {  
    ...  
}
```

The annotation supports the following attributes:

- `id` is mandatory and is the name of the destination as used from Flex
- `source` is mandatory and should be the name of the Spring bean
- `service` is optional when there is only one service for `RemotingMessage` defined in `services-config.xml`. Otherwise this should be the name of the service.
- `channel` is optional if there is only one channel defined in `services-config.xml`. Otherwise this should be the id of the target channel.
- `channels` may be used instead of `channel` to define a failover channel.
- `factory` is optional if there is only one factory in `services-config.xml`. Otherwise this should be the factory id.
- `securityRoles` is an array of role names for securing the destination.

Using scanning allows simplifying your `services-config.xml` file, however it is recommended to use the MVC setup, so you don't even need one !

### 8.2.5. Integration with Spring Security

When not using the Spring MVC setup, you have to manually configure the integration of Spring Security in `granite-config.xml`. Depending on the version of Spring Security you are using, you can use one of the 3 available security services:

Spring Security 3.x

```
<granite-config>  
    ...  
    <!--  
        ! Use Spring based security service.  
    !-->  
    <security type="org.granite.spring.security.SpringSecurity3Service"/>  
  
</granite-config>
```

## Spring Security 2.x

```
<granite-config>
...
<!--
! Use Spring based security service.
!-->
<security type="org.granite.messaging.service.security.SpringSecurityService"/>

</granite-config>
```

## Acegi Security

```
<granite-config>
...
<!--
! Use Spring based security service.
!-->
<security type="org.granite.messaging.service.security.AcegiSecurityService"/>

</granite-config>
```

You may then secure your GraniteDS destinations as shown earlier. Please refer to [Acegi](http://www.springframework.org/) [http://www.springframework.org/] or [Spring Security](http://static.springframework.org/spring-security/site/) [http://static.springframework.org/spring-security/site/] documentation for specific configuration details.

Note however that there are two main ways of securing the GraniteDS AMF endpoint:

- Apply the Spring Security Web filter on the dispatcher servlet. This is the most secure and can be necessary if you share the same web application between a rich client and an HTML client or if you want to use a HTML login form to protect access to the swf resource, but note that as the request credentials are encoded in the AMF request and decoded by the servlet, the request will have to be authenticated as anonymous between the Spring Security filter and the AMF service processor. That means that you have to enable the anonymous support in Spring Security, and that other Web filters will not have access to the authenticated user.
- Let GraniteDS handle security and simply configure a secure remoting destination. This is the recommended way if your application only has a rich client.

## 8.3. Using the Tide API

Most of what is described in the [Tide Remoting](#) section applies for Spring, however GraniteDS also provides an improved integration with Spring services.

### 8.3.1. Configuration with a MVC setup

This is by far the easiest way to use Tide with Spring, it just consists in declaring the GraniteDS flex filter in the Spring context:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:graniteds="http://www.graniteds.org/config"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.0.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/aop http://www.springframework.org/schema/aop/
spring-aop-3.0.xsd
    http://www.graniteds.org/config http://www.graniteds.org/public/dtd/3.0.0/granite-
config-3.0.xsd">

  ...

  <graniteds:server-filter url-pattern="/*" tide="true"/>
</beans>
```

The `server-filter` declaration will setup an AMF processor for the specified url pattern, and the `tide` attribute specifies that you want a Tide-enabled service factory. Note that the actual url that will be listened to by GraniteDS is the combination of this `url-pattern` with the `servlet-mapping` defined in `web.xml` for the dispatcher servlet.

Other configurations can be done with `server-filter`:

- `tide-annotations` is equivalent to `tide-component annotated-with=""` in `granite-config.xml`. It allows to define the list of annotation names that enable remote access to Spring beans. `@RemoteDestination` is always declared by default, but you can use any other one if you don't want a compilation dependency on the GraniteDS libraries.
- `tide-roles` allows to define a list of security roles that are required to access the Tide remote destination. In general it is not necessary to define this destination-wide security and only rely on Spring security for fine-grained access to individual beans.
- `security-service` allows to specify the security service implementation.
- `exception-converters` allows to define a list of server-side exception converters. It's the equivalent to `exception-converters` in `granite-config.xml`.
- `amf3-message-interceptor` allows to define a message interceptor that will be called before and after the processing of each incoming message. You have to define the bean name of an existing bean implementing `AMFMessageInterceptor`.

Additional elements can also be configured in the Spring beans file:

- `tide-identity` allows to declare the identity bean. When using Spring Security ACL you can define here the necessary attributes `acl-service`, `sid-retrieval-strategy` and `object-identity-retrieval-strategy`.
- `tide-persistence` allows to declare the persistence implementation for your application. It is not necessary when you have only one Spring `transactionManager`, otherwise just specify the name of the transaction manager to use. Tide/Spring will automatically determine the kind of transaction management it should use (JTA, JPA or Hibernate API).

Note that in addition to these manual elements, any Spring bean implementing one of the GraniteDS interfaces `SecurityService`, `ExceptionHandler`, `AMFMessageInterceptor` or `TidePersistenceManager` will be automatically picked up and registered in the GraniteDS configuration.

### 8.3.2. Default configuration

If you don't use the MVC setup, you will have to use the standard GraniteDS configuration files instead of the Spring context, and setup these elements manually. You can safely skip this section if you chose the recommended MVC setup.

- You can define in the `tide-annotations` section of `granite-config.xml` the conditions used to enable remote access to Spring destinations (for example all beans annotated with a particular annotation).
- You have to configure the specific Tide/Spring `org.granite.tide.spring.SpringServiceFactory` service factory in `services-config.xml`.
- You have to configure a unique Tide/Spring destination named `spring` in `services-config.xml`

- You must use the destination named `spring` when creating the `ServerSession`.

Here is a default configuration suitable for most cases:

```
<granite-config scan="true">
  ...

  <tide-components>
    <tide-component annotated-
with="org.granite.messaging.service.annotations.RemoteDestination"/>
  </tide-components>

</granite-config>
```

```
<services-config>

<services>
  <service id="granite-service"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <!--
      ! Use "tideSpringFactory" and "my-graniteamf" for "ejb" destination (see below).
      ! The destination must be "spring" when using Tide with default configuration.
    !-->
    <destination id="spring">
      <channels>
        <channel ref="my-graniteamf"/>
      </channels>
      <properties>
        <factory>tideSpringFactory</factory>
      </properties>
    </destination>
  </service>
</services>

<!--
  ! Declare tideSpringFactory service factory.
!-->
<factories>
  <factory id="tideSpringFactory" class="org.granite.tide.spring.SpringServiceFactory"/>
</factories>
```

```

</factories>

<!--
! Declare my-graniteamf channel.
!-->
<channels>
  <channel-definition id="graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>

```

The destination named `spring` will be the one and only destination required for all Spring destinations.

You should also define the correct Spring security service in `granite-config.xml`, see [here](#) for details.

You can use the property `entity-manager-factory-bean-name` to specify an `EntityManagerFactory` bean that will be used for transparent remote lazy loading of collections.

Here is an example with a Spring JPA/Hibernate configuration:

```

<persistence-unit name="spring-pu">
  ...
</persistence-unit>

```

```

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>org.hsqldb.jdbcDriver</value>
  </property>
  <property name="url">
    <value>jdbc:hsqldb:mem:springds</value>
  </property>
  <property name="username">
    <value>sa</value>
  </property>
</bean>

```

```
</property>
<property name="password">
  <value></value>
</property>
</bean>

<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="persistenceUnitName" value="spring-pu" />
  <property name="jpaVendorAdapter">
    <bean
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="false" />
      <property name="generateDdl" value="true" />
      <property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect" />
    </bean>
  </property>
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
  <property name="dataSource" ref="dataSource" />
</bean>
```

If you use a plain Hibernate session instead of JPA, you cannot use `entity-manager-factory-bean-name`, you have to configure a specific Tide persistence manager in the Spring context (assuming the bean name of the Hibernate session factory is `sessionFactory`):

```
<!-- All this AOP stuff is to ensure the Tide persistence manager will be transactional -->
<aop:config>
  <aop:pointcut id="tidePersistenceManagerMethods"
    expression="execution(* org.granite.tide.ITidePersistenceManager.*(..))"/>
  <aop:advisor advice-ref="tidePersistenceManagerMethodsTxAdvice"
    pointcut-ref="tidePersistenceManagerMethods"/>
</aop:config>

<tx:advice id="tidePersistenceManagerMethodsTxAdvice"
  transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" read-only="true"/>
  </tx:attributes>
</tx:advice>
```

```

</tx:attributes>
</tx:advice>

<bean id="tidePersistenceManager"
      class="org.granite.tide.hibernate.HibernateSessionManager" scope="request">
  <constructor-arg>
    <ref bean="sessionFactory"/>
  </constructor-arg>
</bean>

```

### 8.3.3. Basic remoting with dependency injection

When using Spring, the only difference on the client is that you must use the `spring` destination to build the `ServerSession`. Here is a simple example of remoting with an injected client proxy for a Spring service:

```

public class HelloController {

    @Inject @Qualifier("helloService")
    private Component helloService;

    public void hello(String to) {
        // Asynchronous call using handlers
        helloService.call("hello", to, new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }
        });

        @Override
        public void fault(TideFaultEvent fault) {
            System.err.println("Fault: " + fault.getFault());
        }
    };
}

public String helloSync(String to) {
    // Synchronous wait of Future result
    Future<String> futureResult = helloService.call("hello", to);
    String result = futureResult.get();
    System.out.println("Sync result: " + result);
}

```

```
        return result;
    }
}
```

This is almost identical to the standard Tide API described in the [Tide remoting](#) section, and all other methods apply for Spring.

### 8.3.4. Typesafe remoting with dependency injection

You can benefit from the capability of the Gfx code generator (see [here](#)) to generate a strongly typed Java client proxy from the Spring interface when it is annotated with `@RemoteDestination`. In this case, you can inject a typesafe reference to your service and get better compile time error checking and auto completion in your IDE:

```
public class HelloController {

    @Inject @Qualifier("helloService")
    private HelloService helloService;

    // Asynchronous call using handlers
    helloService.hello("Barack", new TideResponder<String>() {
        @Override
        public void result(TideResultEvent<String> result) {
            System.out.println("Async result: " + result.getResult());
        }

        @Override
        public void fault(TideFaultEvent fault) {
            System.err.println("Fault: " + fault.getFault());
        }
    });

    // Synchronous wait of Future result
    Future<String> futureResult = helloService.hello("Barack");
    String result = futureResult.get();
    System.out.println("Sync result: " + result);
}
```

Note that as there is only one instance of `HelloService`, you may also omit the `Qualifier` annotation and use typesafe injection with `@Inject` only.

### 8.3.5. Integration with Spring Security

GraniteDS provides a client-side JavaFX component named `identity` which ensures the integration between the client channel credentials and the server-side container security. It additionally includes an easy-to-use API to define runtime authorization checks on the UI.

Enabling support for the client `identity` component requires to configure the corresponding server-side component in the Spring context:

```
<graniteds:tide-identity/>
```

If you want to integrate with Spring Security ACL authorizations, you will have to specify the name of the ACL service and optionally the Object ID retrieval strategy and SID retrieval strategy (see details on Spring Security ACL [here](http://static.springsource.org/spring-security/site/docs/3.0.x/reference/domain-acls.html) [http://static.springsource.org/spring-security/site/docs/3.0.x/reference/domain-acls.html]):

```
<graniteds:tide-identity acl-service="myAclService"
  object-identity-retrieval-strategy="myObjectIdentityRetrievalStrategy"
  sid-retrieval-strategy="mySIDRetrievalStrategy"/>
```

The client `Identity` component for Spring (of class `org.granite.client.tide.javafx.spring.Identity`) predictably provides two methods `login()` and `logout()` that can be used as any Tide remote call:

```
@Inject
private Identity identity;

public function login(String username, String password) {
    identity.login(username, password, new TideResponder<String>() {
        @Override
        public void result(TideResultEvent<String> event) {
            System.out.println("Logged in as " + event.getResult());
        }
    })

    @Override
    public void fault(TideFaultEvent event) {
        System.out.println("Could not log in");
    }
}
```

```
    }  
    });  
}  
  
public function logout() {  
    identity.logout(new TideResponder<Void>() {  
        @Override  
        public void result(TideResultEvent<Void> event) {  
            System.out.println("Logged out");  
        }  
  
        @Override  
        public void fault(TideFaultEvent event) {  
            System.out.println("Could not log out");  
        }  
    });  
}
```

The `identity` component also exposes the bindable property `loggedIn` that represents the current authentication state. As it is bindable, it can be used to choose between different views, for example to switch between a login form and the application:

```
identity.loggedInProperty().addListener(new ChangeListener<Boolean>() {  
    @Override  
    public void changed(ObservableValue<? extends Boolean> property, Boolean oldValue, Boolean newValue) {  
        if (newValue)  
            showView("applicationView");  
        else  
            showView("loginForm");  
    }  
});
```

Finally the `identity` component is integrated with server-side role-based security and can be used to get information or show/hide UI depending on the user access rights. It provides methods similar to the Spring Security jsp tags `sec:ifAllGranted`, `sec:ifAnyGranted`, `sec:ifNotGranted` and `sec:hasPermission`.

```

Button deleteCategoryButton = new Button();
deleteCategoryButton.setText("Delete Category");
deleteCategoryButton.disableProperty().bind(Bindings.not(identity.ifAllGranted("ROLE_ADMIN")));

Button deleteProductButton = new Button();
deleteProductButton.setText("Delete Product");
deleteProductButton.visibleProperty().bind(identity.hasPermission(productTable.getSelectionModel().getSelectedItem(), "DELETE"));

```

With these declarations, the button labeled *Delete Category* will be enabled only if the user has the role `ROLE_ADMIN` and the button *Delete Product* visible only if the user has the ACL permissions `DELETE` (code 8) or `ADMINISTER` (code 16) for the selected product. Of course any other property can be bound to these observable elements.

The available elements are:

- `ifAllGranted/ifAnyGranted`: the user should have the specified role
- `ifNotGranted`: the user should not have the specified role
- `hasPermission`: the user should have the specified permission for the specified entity

This can also be used as any remote class with result and fault handlers:

```

public void checkRole(final String role) {
    identity.ifAllGranted(role).get(new TideResponder<Boolean>() {
        @Override
        public void result(TideResultEvent<Boolean> event) {
            if (role.equals("ROLE_ADMIN")) {
                if (event.getResult())
                    System.out.println("User has admin role");
                else
                    System.out.println("User does not have admin role");
            }
        }
    });

    @Override
    public void fault(TideFaultEvent event) {
        System.err.println("Error getting role access for role " + role);
    }
}

```



### Warning

`identity.ifAllGranted()` will issue a remote call when it is called the first time, thus its return value cannot be used reliably to determine if the use has the required role. It will always return `false` until the remote call result is received.

It is important to note that `identity` caches the user access rights so only the first call to `ifAllGranted()` will be remote. If the user rights are changed on the server, or if you want to enforce security more than once per user session, you can clear the security cache manually with `identity.clearSecurityCache()`, for example periodically with a `Timer`.

## 8.4. Messaging with Spring (Gravity)

It is possible to configure the three kinds of Gravity topics directly in the Spring context instead of `services-config.xml`:

Simple Topic:

```
<graniteds:messaging-destination id="myTopic"/>
```

This declaration supports the properties `no-local` and `session-selector` (see the [Messaging Configuration section](#)).

You can also define a secure destination by specifying a list of roles required to access the topic:

```
<graniteds:messaging-destination id="myTopic">
  <graniteds:roles>
    <graniteds:role>ROLE_ADMIN</graniteds:role>
  </graniteds:roles>
</graniteds:messaging-destination/>
```

JMS Topic:

```
<graniteds:jms-messaging-destination id="myTopic"
  connection-factory="ConnectionFactory"
  destination-jndi-name="topic/MyTopic"
  transacted-sessions="true"
  acknowledge-mode="AUTO_ACKNOWLEDGE"/>
```

This declaration supports all properties of the default JMS declaration in `services-config.xml` except for non local initial context environments (see the [JMS Integration](#) section).

ActiveMQ Topic:

```
<graniteds:activemq-messaging-destination id="myTopic"
  connection-factory="ConnectionFactory"
  destination-jndi-name="topic/MyTopic"
  transacted-sessions="true"
  acknowledge-mode="AUTO_ACKNOWLEDGE"
  broker-url="vm://localhost"
  create-broker="true"
  wait-for-start="true"
  durable="true"
  file-store-root="/opt/activemq/data"/>
```

This declaration supports all properties of the default ActiveMQ declaration in `services-config.xml` except for non local initial context environments (see the [ActiveMQ Integration](#) section).

Finally note that the Gravity singleton that is needed to push messages from the server (see [here](#)) is available as a bean in the Spring context and can be autowired by type with `@Inject` or `@Autowired` :

```
@Inject
private Gravity gravity;
```



## Integration with CDI

The [Context and Dependency Injection](http://www.jcp.org/en/jsr/detail?id=299) [http://www.jcp.org/en/jsr/detail?id=299] specification is a powerful new feature of Java EE 6. It integrates on a common programming model all the services provided by Java EE.

GraniteDS provides out-of-the-box integration with CDI via the Tide API. You can remotely call CDI beans, and it fully supports serialization of JPA entities from and to your client application, taking care of lazily loaded associations. The support for JPA entity beans is covered in the section [JPA and lazy initialization](#), so this section will only describe how to call CDI components from a Java client. GraniteDS also integrates with container security for authentication and role-based authorization.

The support for CDI is included in the library `granite-cdi.jar`, so you always have to include this library in either `WEB-INF/lib` or `lib` for an ear packaging.



### Note

Only the reference implementation [Weld](http://seamframework.org/Weld) [http://seamframework.org/Weld] is supported for now because of some inconsistencies in a few parts of the spec (notably conversations). This is the one used in JBoss 6 and GlassFish v3.

To provide a more native experience for CDI developers when used in a Servlet 3 compliant container, the CDI support in GraniteDS can be configured with a simple annotated class. The most important features of GraniteDS can be configured this way, and it is still possible to fall back to the default GraniteDS configuration files `services-config.xml` and `granite-config.xml` for unsupported features.

### 9.1. Configuration with Servlet 3

On Servlet 3 compliant containers, GraniteDS can use the new APIs to automatically register its own servlets and filters and thus does not need any particular configuration in `web.xml`. This automatic setup is triggered when GraniteDS finds a class annotated with `@ServerFilter` in one of the application archives:

```
@ServerFilter(configProvider=CDIConfigProvider.class)
public class GraniteConfig {
}
```

The `ConfigProvider` class defines suitable default values for the CDI integration. It is possible however to override these values by setting them in the annotation properties :

```
@ServerFilter(  
    tide=true,  
    type="cdi",  
    factoryClass=CDIServiceFactory.class,  
    tideInterfaces={Identity.class}  
)  
public class GraniteConfig {  
}
```

As for any CDI application, don't forget to add a file `WEB-INF/beans.xml`, even empty. Note that only the Tide API is currently supported out-of-the-box with CDI (there is no basic service factory for `RemoteService`).

The `@ServerFilter` declaration will setup an AMF processor for the specified url pattern, and the `tide` attribute specifies that you want a Tide-enabled service factory. The default url pattern for remoting is `/graniteamf/amf.txt` and messaging is `/gravityamf/amf.txt`.

Other configurations can be done with `@ServerFilter`:

- `tideAnnotations` is equivalent to `tide-component annotated-with=""` in `granite-config.xml`. It allows to define the list of annotation names that enable remote access to CDI beans. `@RemoteDestination` and `@TideEnabled` are always declared by default, but you can use any other one if you don't want a compilation dependency on the GraniteDS libraries.
- `tideInterfaces` is equivalent to `tide-component instance-of=""` in `granite-config.xml`. It allows to define the list of interface/class names that enable remote access to CDI beans.
- `tideRoles` allows to define a list of security roles that are required to access the Tide remote destination. In general it is not necessary to define this destination-wide security and you can only rely on Java EE security for fine-grained access to individual beans.
- `exceptionConverters` allows to define a list of server-side exception converters. It's the equivalent to `exception-converters` in `granite-config.xml`.
- `amf3MessageInterceptor` allows to define a message interceptor. You have to define a class implementing `AMFMessageInterceptor`. It's highly recommended to subclass `org.granite.cdi.CDIInterceptor` and call `super.before` and `super.after` in your implementation.

When using the `ConfigProvider` allows Tide to search in the CDI context for some of its configuration elements. For now, it will lookup beans that implement `ExceptionConverter`, `AMF3MessageInterceptor` or `SecurityService` and use the existing beans.

## 9.2. Default Configuration

If you don't use the Servlet 3 configuration, you will have to use the standard GraniteDS configuration files instead, and setup these elements manually. You can safely skip this section if you choose Servlet 3 configuration.

- You can define in the `tide-annotations` section of `granite-config.xml` the conditions used to enable remote access to Seam destinations (for example all beans annotated with a particular annotation).
- You have to configure the specific Tide/CDI `org.granite.tide.cdi.CDIServiceFactory` service factory in `services-config.xml`.
- You have to configure a unique Tide/CDI destination named `cdi` in `services-config.xml`
- You have to retrieve the Tide context in Flex with `Cdi.getInstance().getCdiContext()` instead of `Tide.getInstance().getContext()`.

Here is a default configuration suitable for most cases:

```
<granite-config scan="true">
  ...

  <tide-components>
    <tide-component annotated-
with="org.granite.messaging.service.annotations.RemoteDestination"/>
    <tide-component annotated-with="org.granite.tide.annotations.TideEnabled"/>
  </tide-components>

</granite-config>
```

```
<services-config>

<services>
  <service id="granite-service"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <!--
      ! Use "tideCdiFactory" and "my-graniteamf" for "cdi" destination (see below).
      ! The destination must be "cdi" when using Tide with default configuration.
    !-->
```

```
<destination id="cdi">
  <channels>
    <channel ref="my-graniteamf"/>
  </channels>
  <properties>
    <factory>tideCdiFactory</factory>
  </properties>
</destination>
</service>
</services>

<!--
! Declare tideCdiFactory service factory.
!-->
<factories>
  <factory id="tideCdiFactory" class="org.granite.tide.cdi.CdiServiceFactory"/>
</factories>

<!--
! Declare my-graniteamf channel.
!-->
<channels>
  <channel-definition id="graniteamf" class="mx.messaging.channels.AMFChannel">
    <endpoint
      uri="http://{server.name}:{server.port}/{context.root}/graniteamf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>
</services-config>
```

The destination named `cdi` will be the one and only destination required for all CDI destinations.

### 9.3. Using the Tide API

Most of what is described in the [Tide Remoting](#) section applies for CDI, however GraniteDS also provides a much improved integration with CDI when using the Tide client API.

#### 9.3.1. Basic remoting with dependency injection

When using CDI, the only difference on the client is that you must use the `cdi` destination to build the `ServerSession`. Here is a simple example of remoting with an injected client proxy for a CDI service:

```

public class HelloController {

    @Inject @Qualifier("helloService")
    private Component helloService;

    public void hello(String to) {
        // Asynchronous call using handlers
        helloService.call("hello", to, new TideResponder<String>() {
            @Override
            public void result(TideResultEvent<String> result) {
                System.out.println("Async result: " + result.getResult());
            }

            @Override
            public void fault(TideFaultEvent fault) {
                System.err.println("Fault: " + fault.getFault());
            }
        });
    }

    public String helloSync(String to) {
        // Synchronous wait of Future result
        Future<String> futureResult = helloService.call("hello", to);
        String result = futureResult.get();
        System.out.println("Sync result: " + result);
        return result;
    }
}

```

This is almost identical to the standard Tide API described in the [Tide remoting](#) section, and all other methods apply for Spring.

### 9.3.2. Typesafe remoting with dependency injection

You can benefit from the capability of the Gfx code generator (see [here](#)) to generate a strongly typed Java client proxy from the CDI interface when it is annotated with `@RemoteDestination`. In this case, you can inject a typesafe reference to your service and get better compile time error checking and auto completion in your IDE:

```

public class HelloController {

```

```
@Inject
private HelloService helloService;

// Asynchronous call using handlers
helloService.hello("Barack", new TideResponder<String>() {
    @Override
    public void result(TideResultEvent<String> result) {
        System.out.println("Async result: " + result.getResult());
    }

    @Override
    public void fault(TideFaultEvent fault) {
        System.err.println("Fault: " + fault.getFault());
    }
});

// Synchronous wait of Future result
Future<String> futureResult = helloService.hello("Barack");
String result = futureResult.get();
System.out.println("Sync result: " + result);
}
```

Note that if there are more than one instance of `HelloService`, you may add the `Qualifier` annotation to disambiguate the actual server bean name (meaning that the server beans also have to be annotated with `@Named`).

### 9.3.3. Integration with Events

The Tide client context can register listeners for CDI events triggered on the server-side. The interesting events are sent back along the server response and dispatched at the end of the processing of the result so that the context is correctly synchronized when the event is dispatched.

Here is a simple example:

```
@Stateful
public class HotelBookingAction implements HotelBooking {
    ...
    @Inject
    @Confirmed
    private Event<BookingEvent> bookingConfirmedEventSrc;
    ...
}
```

```

public void confirm() {
    em.persist(booking);
    bookingConfirmedEventSrc.fire(new BookingEvent(booking));
    conversation.end();
}
}

```

```

[Observer(remote="true")]
public function bookingConfirmedHandler(event:BookingEvent):void {
    Alert.show("Booking confirmed: " + event.booking);
}

```

## 9.4. Messaging with CDI (Gravity)

As with EJB 3 and when using a servlet 3 compliant container, it is possible to configure the three kinds of Gravity topics in the configuration class annotated with `@ServerFilter`. You can simply add variables to your configuration class annotated with `@MessagingDestination`, `@JmsTopicDestination` or `@ActiveMQTopicDestination`, the name of the variable will be used as destination id.

Simple Topic:

```

@FlexFilter()
public class MyConfig {

    @MessagingDestination(noLocal=true, sessionSelector=true)
    AbstractMessagingDestination myTopic;
}

```

This declaration supports the properties `no-local` and `session-selector` (see the [Messaging Configuration section](#)).

You can also define a secure destination by specifying a list of roles required to access the topic:

```
@MessagingDestination(noLocal=true, sessionSelector=true, roles={ "admin", "user" })
AbstractMessagingDestination myTopic;
```

JMS Topic:

```
@JMSTopicDestination(noLocal=true,
    sessionSelector=true,
    connectionFactory="ConnectionFactory",
    topicJndiName="topic/myTopic",
    transactedSessions=true,
    acknowledgeMode="AUTO_ACKNOWLEDGE",
    roles={ "admin", "user" })
AbstractMessagingDestination myTopic;
```

This declaration supports all properties of the default JMS declaration in `services-config.xml` except for non local initial context environments (see the [JMS Integration](#) section).

ActiveMQ Topic:

```
@ActiveMQTopicDestination(noLocal=true,
    sessionSelector=true,
    connectionFactory="ConnectionFactory",
    topicJndiName="topic/myTopic",
    transactedSessions=true,
    acknowledgeMode="AUTO_ACKNOWLEDGE",
    brokerUrl="vm://localhost",
    createBroker=true,
    waitForStart=true,
    durable=true,
    fileStoreRoot="/opt/activemq/data",
    roles={ "admin", "user" })
AbstractMessagingDestination myTopic;
```

This declaration supports all properties of the default ActiveMQ declaration in `services-config.xml` except for non-local initial context environments (see the [ActiveMQ Integration](#) section).

Finally note that the Gravity singleton that is needed to push messages from the server (see [here](#)) is available as a CDI bean and can be injected in any component :

```
@Inject  
private Gravity gravity;
```



## Client-Side Validation API (JSR 303)

The "Bean Validation" [specification](http://jcp.org/en/jsr/detail?id=303) [http://jcp.org/en/jsr/detail?id=303] (aka JSR-303) standardizes an annotation-based validation framework for Java. It provides an easy and powerful way of processing bean validations, with a pre-defined set of constraint annotations, allowing to arbitrarily extend the framework with user specific constraints.

It's of course possible to use it in a Java client application, and Bean Validation constraint annotations can be put on any bean with property accessors. JavaFX however doesn't provide any simple way to integrate data binding and validation. GraniteDS provide a simple client component named `FormValidator` that helps bridging Bean Validation and JavaFX data binding.

### 10.1. Integration with code generation tools (Gfx)

The Bean Validation specification was primarily intended to be used with Java entity beans. GraniteDS code generation tools replicate your Java model into a JavaFX-enabled model and may be configured in order to copy validation annotations. All you have to do is to change the default `org.granite.generator.as3.DefaultEntityFactory` to `org.granite.generator.as3.BVEntityFactory`.

With the Ant task, use the `entityfactory` attribute as follow in your `build.xml`:

```
<gfx entityfactory="org.granite.generator.as3.BVEntityFactory" ...>
  ...
</gfx>
```

With the Maven plugin, add the `entityfactory` option in the plugin configuration:

```
<configuration>
  <generatorToUse>graniteds23</generatorToUse>
  <baseOutputDirectory>${project.build.directory}/generated-sources</baseOutputDirectory>
  <outputDirectory>${basedir}/src/main/java</outputDirectory>
  <translators>
    <translator>com.wineshop.admin=com.wineshop.admin.client</translator>
  </translators>
  <extraOptions>
    <tide>true</tide>
    <uid>uid</uid>
    <transformer>org.granite.generator.javaafx.JavaFXGroovyTransformer</transformer>
    <as3typefactory>org.granite.generator.javaafx.DefaultJavaFXTypeFactory</as3typefactory>
```

```
<entityFactory>org.granite.generator.as3.BVEntityFactory</entityFactory>
<outputEnumToBaseOutputDirectory>>false</outputEnumToBaseOutputDirectory>
</extraOptions>
...
</configuration>
```

Then, provided that you have a Java entity bean like this one:

```
@Entity
public class Person {

    @Id @GeneratedValue
    private Integer id;

    @Basic
    @Size(min=1, max=50)
    private String firstname;

    @Basic
    @NotNull(message="You must provide a lastname")
    @Size(min=1, max=255)
    private String lastname;

    // getters and setters...
}
```

... you will get this generated ActionScript3 code:

```
@JavaFXObject
public class PersonBase implements Identifiable, Lazyable, DataNotifier {

    ...

    private StringProperty firstnameProperty = new SimpleStringProperty(this, "firstname");
    private StringProperty lastnameProperty = new SimpleStringProperty(this, "lastname");

    public void setFirstname(String value) {
```

```

    this.firstname = value;
}
@Size(min=1, max=50, message="{javax.validation.constraints.Size.message}")
public String getFirstname() {
    return this.firstname;
}

public void setLastname(String value) {
    this.lastname = value;
}
@NotNull(message="You must provide a last name")
@Size(min=1, max=255, message="{javax.validation.constraints.Size.message}")
public String getLastname() {
    return this.lastname;
}

....
}

```

You may then use the standard Bean Validation mechanism to validate your client JavaFX bean.

This works for plain Java beans and entity beans.

## 10.2. Using the FormValidator class

With the FormValidator component, you can easily add validation to any part of a UI form: the FormValidator performs validation on the fly whenever the user enters data into user inputs and automatically displays error messages when these data are incorrect, based on constraint annotations placed on the bean properties. This however requires that the form uses JavaFX data binding to propagate updates between UI components and data beans.

Example (using the Person bean introduced above and bidirectional bindings):

```

private Person person = new Person();
private VBox personForm;
private FormValidator personFormValidator;

public void buildForm() {
    person = new Person();

    personForm = new VBox();
    TextField textFirstname = new TextField();

```

```
TextField textLastname = new TextField();

personForm.getChildren().add(textFirstname);
personForm.getChildren().add(textLastname);

texteFirstname.textProperty().bindBidirectional(person.firstnameProperty());
texteLastname.textProperty().bindBidirectional(person.lastnameProperty());

personFormValidator = new FormValidator();
personFormValidator.setForm(formPerson);
}

public void validate() {
    if (!personFormValidator.validate(person)) {
        // Data is invalid
        return;
    }

    // Data is valid, do something useful...
}
```

In the above sample, the personForm form uses two bidirectional bindings between the text inputs and the person bean. Each time the user enter some text in an input, the value of the input is copied into the bean and triggers a validation.

Note that JavaFX does not provide any standard way of displaying the error messages, so you are basically on your own to choose whatever look & feel you prefer (tooltip, basic text...).

To allow displaying these messages at the right time, the form validator dispatches two particular events on the target form: `ValidationResultEvent.VALID` and `ValidationResultEvent.INVALID`. The event also contains a list of more detailed error messages of type `ValidationResult`.

Here a very basic example that simply changes the border color of the inputs to red when the input data is invalid.

```
personForm.addEventHandler(ValidationResultEvent.ANY, new EventHandler<ValidationResultEvent>() {
    @Override
    public void handle(ValidationResultEvent event) {
        if (event.getEventType() == ValidationResultEvent.INVALID)
            ((Node)event.getTarget()).setStyle("-fx-border-color: red");
        else if (event.getEventType() == ValidationResultEvent.VALID)
```

```
((Node)event.getTarget()).setStyle("-fx-border-color: null");  
}  
});
```

The global validation of the person bean will be performed when `FormValidator.validateEntity()` is called. However, class-level constraint violations cannot be automatically associated to an input, and these violations prevent the `fValidator.validateEntity()` call to succeed while nothing cannot be automatically displayed to the user.

To solve this problem, two options are available:

1. Get unhandled violations with the `FormValidator.unhandledViolations` property
2. Listen to validation events of type `ValidationResultEvent.UNHANDLED`

The second option let you do whatever you want with these unhandled violations. You can display the error messages anywhere and get any useful information from the `ConstraintViolation` objects.



# Data Management

GraniteDS provides various features that simplify the handling of data between the client and Java EE, in particular when using JPA or Hibernate as a persistence mechanism.

## 11.1. JPA and Managed Entities

Tide provides an integration between the concept of a client persistence context and the server persistence context (JPA or Hibernate).

In particular, Tide maintains a client-side cache of entity instances and ensures that every instance is unique in the Java client context. To achieve this, it requires a unique identifier on each entity class. This is why GraniteDS supports the concept of managed entities.

All entities implementing `Identifiable` are considered as corresponding to Hibernate/JPA managed entities on the server. The managed entities should always use JavaFX bindable properties so the client entity manager can track changes on their values.

It is highly recommended to use JPA optimistic locking in a multi-tier environment (`@Version` annotation). Note that Tide currently only supports `Integer` or `Long` version fields, not timestamps and that the field must be nullable (entity instances with a `null/NaN` version field will be considered as unsaved). It is also *highly* recommended to add a persistent `uid` field (generally typed as a 36 bytes `String`) to have a consistent identifier through all application layers, see the explication below.

Below is a `AbstractEntity` class that can be used as a JPA mapped superclass for your application entities. The entity listener ensures that the entity always has an initialized `uid` field, but in general this identifier will be initialized from the client.

```
@MappedSuperclass
@EntityListeners({AbstractEntity.AbstractEntityListener.class})
public abstract class AbstractEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    private Long id;

    /* "UUID" and "UID" are Oracle reserved keywords -> "ENTITY_UID" */
    @Column(name="ENTITY_UID", unique=true, nullable=false, updatable=false, length=36)
    private String uid;

    @Version
```

```
private Integer version;

public Long getId() {
    return id;
}

public Integer getVersion() {
    return version;
}

@Override
public boolean equals(Object o) {
    return (o == this || (o instanceof AbstractEntity && uid().equals(((AbstractEntity)o).uid())));
}

@Override
public int hashCode() {
    return uid().hashCode();
}

public static class AbstractEntityListener {
    @PrePersist
    public void onPrePersist(AbstractEntity abstractEntity) {
        abstractEntity.uid();
    }
}

private String uid() {
    if (uid == null)
        uid = UUID.randomUUID().toString();
    return uid;
}
}
```



### Tip

The easiest and recommended way for getting Tide enabled managed entities is to generate them from JPA/Java classes with Gfx using the `tide="true"` option.

Example build file for ant:

```

<gfx outputdir="java" tide="true">
  <classpath>
    <pathelement location="classes"/>
  </classpath>
  <fileset dir="classes">
    <include name="com/myapp/entity/**/*.class"/>
  </fileset>
</gfx>

```

**Important things on ID/UID.** In a typical client/app server/database application, an entity lives in three layers:

- the Java client
- the Hibernate/JPA persistence context
- the database

During the entity lifecycle, the only invariant is the id. The id reliably links the different existing versions of the entity in the three layers. When updating existing entities coming from the database, there are, in general, no problems because the id is defined and is maintained in the three layers during the different serialization/persistence operations. A problem arises when a new entity is being created in any of the two upper layers (client/JPA). The new entity has no id until it has been persisted to the database. This means that between the initial creation and the final stored entity, the id has changed from null to a real value. It is thus impossible to have a reliable link between the original entity that has been created and the entity that has been stored. This is even more complex if you try to add two or more new entities to a collection because, in this case, there will be absolutely no way to determine which one has been persisted with which id because they all had null ids at the beginning. The problem already exists outside of a rich client when you use a database generated id with Hibernate/JPA. The most common solution is to have a second persisted id, the uid, which is created by the client and persisted along with the entity (but is NOT the database key). On the client, we have the same problem because the entities are often serialized/deserialized between client and the server, so we cannot check object instances equality. When there is a uid field in the Java entity, the Gfx Tide template will generate a uid property on the JavaFX object. In other cases, the Tide template tries to build a convenient uid property from the entity id. This second mode is, of course, vulnerable to the initial null id problem. In conclusion, the recommended approach to avoid any kind of subtle problems is to have a real uid property which will be persisted in the database but is NOT a primary key for efficiency concerns. If it is not possible to add a uid property due to a legacy database schema or Java classes, it will work most of the time but you will then have to be very careful when creating new entities from the client application. You will then have to take care that hashCode() and equals() are implemented based on this property uid.

### 11.2. Transparent lazy loading

All uninitialized lazy collections coming from the server are transparently wrapped on the client side by the Tide context in a `ManagedPersistentCollection` or `ManagedPersistentMap`. This collection can be used as a data provider for any JavaFX UI component that is able to handle `ObservableList/ObservableMap` (all JavaFX components, such as `Table` and `List` do). When data is requested by the UI component, the collection asks the server for the real collection content. This lazy loading functionality is completely transparent but will happen only if the collection is bound to a UI component.

On the server Tide will try different means to determine the correct `EntityManager`/Hibernate session to use. The whole collection and owning entity are then retrieved from a newly created persistence context. If you have a deep object graph, it will then be possible to get entities from different persistence contexts in the same client context, and it can lead to inconsistencies in the client data and issues with optimistic locking/versioning.

Depending on the server framework of the application (Spring, EJB 3, Seam, CDI...), Tide will lookup an `EntityManager` or an Hibernate session in JNDI, in the Spring context or any other relevant way, and will try to determine the correct transaction management (JTA, JPA...). With Spring or Seam, it is possible to override the default persistence manager if you have particular requirements: with Spring you just have to configure a bean implementing `TidePersistenceManager` in the application context, with Seam you can override the component named `org.granite.tide.seam.seamInitializer` with a component extending the class `org.granite.tide.seam.seamInitializer`. Using a custom persistence manager can be useful for example if you have multiple `EntityManagerFactories` and want to be able to select one of them depending on the entity whose collection has to be fetched.

**Manual fetching of lazy collections.** In some cases you may need to trigger manually the loading of a lazy loaded collection. As told earlier, all collections are wrapped in a `PersistentCollection` or `PersisteneMap`. These two classes expose a method `withInitialized` that can take a function callback that can do something once the collection is populated:

```
((ManagedPersistentCollection<Object>)myEntity.getMyCollection()).withInitialized(new InitializationCallback() {
    @Override
    public void call(ManagedPersistentCollection<Object> collection) {
        // Do something with the content of the list
        Object obj = collection.get(0);
    }
});
```

## 11.3. Dirty checking and conflict handling

The Tide framework includes a client-side entity cache where each managed entity exists only once for each Tide context. Besides maintaining this cache, Tide tracks all changes made on managed entities and on their associations and saves these changes for each modification. This flag is always reset to `false` when the same instance is received from the server, so this flag is indeed an indication that the user has changed something since the last remote call.

A particular entity instance can be in two states :

- Stable: the instance has not been modified until it was received from the server
- Dirty : the instance has been modified since last received from the server

The current state of an entity can be accessed with :

```
entity.isDirty()
```

The property `dirty` is bindable, so it could be used for example to enable/disable a Save button.

Note that this dirty flag only indicates if a direct property or collection of the entity has been changed, it does not indicate if something has changed deeper in the object graph (that would not make sense anyway for circular graphs). The correct way of knowing if any object has been changed in the context, is to use the property `dirtyProperty()` of the whole Tide context/entity manager.

```
@Inject
private JavaFXDataManager dataManager;

public void createButton() {
    Button saveButton = new Button();
    saveButton.setText("Save");
    saveButton.disableProperty().bind(Bindings.not(dataManager.dirtyProperty()));
}
```



### Warning

This dirty flag is a lot more reliable when using optimistic locking. The best way for Tide to know that an entity instance has actually been changed on the server is to check that its `@version` field has been incremented.

In a typical client/server interaction, here is what happens :

1. The client application retrieves an entity instance from the server, for example with a version number 0. This instance is considered stable.
2. The user modifies data on the client, possibly with bidirectional data binding. The version number stays 0, other properties are modified and the client state becomes dirty.
3. The user clicks on a save button. The client application calls a service and retrieves the result. The server has incremented the version number to 1, so Tide overwrites the cached instance on the client and it is considered as stable again.

Note that if you retrieve the same instance without version increment, the local changes won't be overwritten. In the previous example, if the server returns the same instance with an unchanged version number of 0, the local instance will still be dirty. That means that you can still issue queries that return a locally changed entity without losing the user changes.

One nice possibility with this programming model is that you can easily implement a cancel button after step 2. If you use bidirectional data binding, the client view of the entity instance has already become dirty. As Tide always saves the local changes, it also provides a simple way of restoring the last stable state :

```
@Inject
private EntityManager entityManager;

private void restore() {
    entityManager.resetEntity(entity);
}
```

You can also reset all entities in the context to their last stable state with :

```
@Inject
private EntityManager entityManager;

private void restoreAll() {
    entityManager.resetAllEntities();
}
```

If you look at the previous process in 3 steps, we assume that nobody else has changed the data the user has been working on between 1 and 3. In concurrent environments with read-write data,

there are possibilities that someone else has modified the entity on the server between step 1 and step 3.

There are two ways of managing this: either you just rely on optimistic locking and intercept the corresponding server exceptions to display a message to the user, or you use data push (see section [Data Push](#)) so all clients are updated in near real-time. Note however that even with data push, there can still be conflicts between changes made by a user and updates received from the server.

With normal optimistic locking, the remote service call at step 3 will trigger a `OptimisticLockException`. Tide provides a built-in exception handler to handle this case: it will extract the entity argument of the exception, compare its state with the client state and dispatch a conflict event `TideDataConflictEvent` on the Tide context when it's not identical. The exception handler can be enabled with :

```
ContextManager.getContext().set(new OptimisticLockExceptionHandler());
```

Or when using Spring on the client, by simply declaring a Spring bean of type `OptimisticLockExceptionHandler`

When data push is used, an entity instance can be updated with data received from the server at any time. If the current user was working on this instance, it is obviously not desirable that his work is overwritten without notice. Similarly to the previous case, Tide will determine that an incoming data from another user session is in conflict with the local data and call `DataConflictListener` from the Tide entity manager.

What can you do with this event ? Basically there are two possibilities : accept the server-side state or keep the client state. Here is an example of a conflict listener defined in a client application, generally in the main application class :

```
@Inject
private EntityManager entityManager;

public void init() {
    entityManager.addListener(new DataConflictListener() {
        @Override
        public void onConflict(EntityManager entityManager, Conflicts conflicts) {
            conflicts.acceptAllClient();
        }
    });
}
```

The `Conflicts` class exposes a few properties that give more details about the conflicts and make possible to present a better alert message to the user.

When using the Hibernate native API (`Session`), the `optimistick lock exception StaleObjectStateException` is unfortunately missing a critical information to allow for correct conflict handling (that is present in the JPA `OptimistickLockException`). In this case, you should use the provided Hibernate GraniteDS-provided event wrappers that add the missing data to the Hibernate exception. Here is what it will look like when configuring the `SessionFactory` with Spring :

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
      <prop key="hibernate.show_sql">false</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
  <property name="eventListeners">
    <map>
      <entry key="merge"><bean class="org.granite.tide.hibernate.HibernateMergeListener"/>
    </entry>
      <entry key="create"><bean class="org.granite.tide.hibernate.HibernatePersistListener"/>
    </entry>
      <entry key="create-onflush"><bean class="org.granite.tide.hibernate.HibernatePersistOnFlushListener"/></entry>
      <entry key="delete"><bean class="org.granite.tide.hibernate.HibernateDeleteListener"/>
    </entry>
      <entry key="update"><bean class="org.granite.tide.hibernate.HibernateSaveOrUpdateListener"/>
    </entry>
      <entry key="save-update"><bean class="org.granite.tide.hibernate.HibernateSaveOrUpdateListener"/></entry>
      <entry key="save"><bean class="org.granite.tide.hibernate.HibernateSaveOrUpdateListener"/>
    </entry>
      <entry key="lock"><bean class="org.granite.tide.hibernate.HibernateLockListener"/></entry>
    </map>
  </property>
</bean>
```

```

        <entry key="flush"><bean class="org.granite.tide.hibernate.HibernateFlushListener"/>
    </entry>

        <entry key="auto-
flush"><bean class="org.granite.tide.hibernate.HibernateAutoFlushListener"/></entry>
    </map>
</property>
...
</bean>

```

## 11.4. Data validation

Tide integrates with Hibernate Validator 3.x and the Bean Validation API (JSR 303) implementations, and propagate the server validation errors to the client UI components.

The server support for Hibernate Validator 3 is available in `granite-hibernate.jar`, and the support for Bean Validation is available in `granite-beanvalidation.jar`. You will have to add one of these jars in your application `lib` folder.

The validator integration is based on the GraniteDS exception handling framework. A server exception converter is registered to handle the `InvalidStateException`, and a client exception handler can be registered with:

```
ContextManager.getContext().set(new ValidationExceptionHandler());
```

Or when using Spring on the client, by simply declaring a Spring bean of type `ValidationExceptionHandler`

This exception handler intercepts all server validation faults and dispatches a validation event on the context. The `ValidationExceptionHandler` also integrates with the GraniteDS `FormValidator` so both client and server-detected constraint violations can be handled transparently and propagated to the UI.

## 11.5. Data paging

GraniteDS provides the `PagedQuery` component which is an implementation of `ObservableList` and can be used as a data provider for most UI components such a tables or lists.

This component supports paging and can be mapped to a server component which execute queries. The collection is completely paged and keeps in memory only the data needed for the current display. In fact, it keeps in memory two complete pages to avoid too many server calls.

PagedQuery also supports automatic remote sorting and filtering. The server-side part of the paging depends on the server technology and is described in the next paragraphs.

On the client-side, you first need to register the client component with:

```
PagedQuery people = new PagedQuery(serverSession);
people.setMethodName("list");
people.setRemoteComponentClass(PeopleService.class);
people.setElementClass(Person.class);
ContextManager.getContext().set("people", people);
```

This registers a client component with a page size defined by the server. It's also possible to define the page size on the client with :

```
people.setMaxResults(25);
```

When using Spring on the client, you can simply declare a PagedQuery bean in your Spring context.

That's all. Just bind the component as a data provider for any component and it should work as expected (here in a FXML):

```
<TableView fx:id="peopleView" id="peopleList" layoutX="10" layoutY="40" items="$people">
  <columns>
    <TableColumn fx:id="firstnameColumn" id="firstnameColumn" text="First
name" sortable="true"/>
    <TableColumn fx:id="lastnameColumn" id="lastnameColumn" text="Last
name" sortable="true"/>
  </columns>
</TableView>
```

To handle sorting automatically when the user click on a column header, you can attach a sorting adapter:

```
people.setSort(new TableViewSort<Person>(peopleView, new Person()));
```

The `TableViewSort` adapter requires an instance of the element type of the table view/paged query.

**Server-side implementation.** The `PagedQuery` components expects that the corresponding server component implements a specific method to fetch elements. There are two ways of handling filtering, either with an untyped map or with a typesafe filter object:

For untyped filters, the server component should implement the following method:

```
public Map find(Map<?, ?> filter, int first, int max, String order, boolean desc);
```

`first`, `max`, `order` and `desc` are straightforward. `filter` is a map containing the parameter values of the query. These values can be set on the client by:

```
pagedQuery.getFilterMap().put("param1", "value1");
pagedQuery.getFilterMap().put("param2", "value2");
...
```

Alternatively you can use a typesafe filter object by setting the property `filterClass` on `PagedQuery`. Usually the filter class can be the same as the element class, so any property of the elements can be used to filter the results. In more complex cases, you may use any other specific filter class.

```
pagedQuery.setFilterClass(Person.class);
pagedQuery.getFilter().setLastname("Bar");
...
```

The return object must be a map containing four properties:

- `firstResult`: Should be exactly the same as the argument passed in (`int first`).

- `maxResults`: Should be exactly the same as the argument passed in (int max), except when its value is 0, meaning that the client component is initializing and needs a max value. In this case, you have to set the page value, which must absolutely be greater than the maximum expected number of elements displayed simultaneously in a table.
- `resultCount`: Count of results.
- `resultList`: List of results.

Alternatively you can also return a result of type `org.granite.tide.data.model.Page`. That implies a compile dependency of your services on a GraniteDS API, which may not be suitable. If necessary you can define your own page class and use a converter to translate from your server class to the client Page class.

If you are using Spring Data, you can simply return an instance of the Page class of Spring Data and use the built-in `PageableConverter` to translate between GraniteDS Page and Spring Data Page.

The following code snippet is a quick and dirty implementation and can be used as a base for other implementations (here this is a Spring service but the equivalent implementations for EJB3 or CDI would be extremely similar):

```
@Service("people")
@Transactional(readOnly=true)
public class PeopleServiceImpl implements PeopleService {

    @PersistenceContext
    protected EntityManager manager;

    public Map<String, Object> find(Map<String, Object> filter, int first, int max, String order, boolean desc) {
        Map<String, Object> result = new HashMap<String, Object>(4);

        String from = "from Person e ";
        String where = "where lower(e.lastName) like '% ' || lower(:lastName) || '% ' ";
        String orderBy = (
            order != null ? "order by e." + order + (desc ? " desc" : "") : ""
        );
        String lastName = (
            filter.containsKey("lastName") ? (String)filter.get("lastName") : ""
        );

        Query qc = manager.createQuery("select count(e) " + from + where);
        qc.setParameter("lastName", lastName);
        long resultCount = (Long)qc.getSingleResult();
```

```

    if (max == 0)
        max = 36;

    Query ql = manager.createQuery("select e " + from + where + orderBy);
    ql.setFirstResult(first);
    ql.setMaxResults(max);
    ql.setParameter("lastName", lastName);
    List resultList = ql.getResultList();

    result.put("firstResult", first);
    result.put("maxResults", max);
    result.put("resultCount", resultCount);
    result.put("resultList", resultList);

    return result;
}
}

```

Or with typesafe arguments:

```

@Service("people")
@Transactional(readOnly=true)
public class PeopleServiceImpl implements PeopleService {

    @PersistenceContext
    protected EntityManager manager;

    public Page find(Person filter, PageInfo pageInfo) {
        Page result = new Page();

        String from = "from Person e ";
        String where = "where lower(e.lastName) like '%" || lower(:lastName) || '%" ";
        String orderBy = (
            pageInfo.getSortInfo().getOrder() != null ? "order by
e." + pageInfo.getSortInfo().getOrder()[0] + (pageInfo.getSortInfo().getDesc()[0] ? " desc" : "") : ""
        );
        String lastName = (
            filter.getLastname() != null ? filter.getLastname() : ""
        );

        Query qc = manager.createQuery("select count(e) " + from + where);
    }
}

```

```
qc.setParameter("lastName", lastName);
long resultCount = (Long)qc.getSingleResult();

if (max == 0)
    max = 36;

Query ql = manager.createQuery("select e " + from + where + orderBy);
ql.setFirstResult(first);
ql.setMaxResults(max);
ql.setParameter("lastName", lastName);
List resultList = ql.getResultList();

result.setFirstResult(first);
result.setMaxResults(max);
result.setResultCount(resultCount);
result.setResultList(resultList);

return result;
}
```

It is also possible to define on the client side an alternative remote component name and method name that will implement the querying :

```
pagedQuery.setRemoteComponentName("peopleService");
pagedQuery.setMethodName("list");
```

### 11.6. Data push

In classic client applications using remoting, data is updated only when the user does an action that triggers a call to the server. As it is possible to do many things purely on the client without involving the server at all, that can lead to stale client state if someone else has modified something between updates.

Optimistic locking ensures that the data will keep consistent on the server and in the database, but it would be better if data updates were pushed in real-time to all connected clients.

Tide makes this possible by integrating with the JPA provider and the Gravity messaging broker to dispatch data updates to subscribed clients.

This requires a bit of configuration :

1. Define a Gravity topic
2. Add the Tide JPA listener DataPublishListener on entities that should be tracked
3. Add the Tide annotation DataEnabled on all server components involved in modifications of these data
4. Subscribe to the topic on the client with the DataObserver component

Let's see all this in details :

Define a Gravity topic: in the standard case, it can be done in `services-config.xml`:

```
<service id="gravity-service"
  class="flex.messaging.services.MessagingService"
  messageTypes="flex.messaging.messages.AsyncMessage">
  <adapters>
    <adapter-definition id="simple"
      class="org.granite.gravity.adapters.SimpleServiceAdapter"/>
  </adapters>

  <destination id="dataTopic">
    <properties>
      <no-local>true</no-local>
      <session-selector>true</session-selector>
    </properties>
    <channels>
      <channel ref="gravityamf"/>
    </channels>
    <adapter ref="simple"/>
  </destination>
</service>

...
<channel-definition id="gravityamf" class="org.granite.gravity.channels.GravityChannel">
  <endpoint
    uri="http://{server.name}:{server.port}/{context.root}/gravityamf/amf"
    class="flex.messaging.endpoints.AMFEndpoint"/>
</channel-definition>
```

With Spring or Seam, this can be done more easily in the respective configuration files `application-context.xml` or `components.xml`:

Spring context:

```
<graniteds:messaging-destination id="dataTopic" no-local="true" session-selector="true"/>
```

This example configuration defines a simple Gravity destination but it's also possible to use the JMS, ActiveMQ or any custom adapter if you need transactional behaviour or better scalability.

The two important parameters for the topic definition are :

- `no-local` should be set to `true`. That means that the client that triggers the modification will not receive the result of the update twice : first by the remoting call, then by the messaging update.
- `session-selector` must be set to `true`. Tide uses JMS-style selectors to determine which data will be sent to which clients and thus needs to store the current messaging selector state in the user session.

Add the Tide JPA publishing listener on the entities that should be tracked:

```
@Entity
@EntityListeners({DataPublishListener.class})
public abstract class MyEntity {
    ...
}
```

When using the Hibernate native API instead of JPA, you can use the following listener configuration:

```
Configuration configuration = new Configuration();
...
configuration.setListener("post-insert", new HibernateDataPublishListener());
configuration.setListener("post-update", new HibernateDataPublishListener());
configuration.setListener("post-delete", new HibernateDataPublishListener());
```

With Hibernate XML config:

```

<hibernate-configuration>
  <session-factory>
    ...
    <event type="post-insert">
      <listener class="org.granite.tide.hibernate.HibernateDataPublishListener"/>
    </event>
    <event type="post-update">
      <listener class="org.granite.tide.hibernate.HibernateDataPublishListener"/>
    </event>
    <event type="post-delete">
      <listener class="org.granite.tide.hibernate.HibernateDataPublishListener"/>
    </event>
  </session-factory>
</hibernate-configuration>

```

And with Spring:

```

<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
      <prop key="hibernate.show_sql">>false</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
  <property name="eventListeners">
    <map>
      <entry key="post-insert">
        <list><bean class="org.granite.tide.hibernate.HibernateDataPublishListener"/></list>
      </entry>
      <entry key="post-update">
        <list><bean class="org.granite.tide.hibernate.HibernateDataPublishListener"/></list>
      </entry>
      <entry key="post-delete">
        <list><bean class="org.granite.tide.hibernate.HibernateDataPublishListener"/></list>
      </entry>
    </map>
  </property>
  ...

```

```
</bean>
```

Then add the Tide data annotation on all services, example here with a Spring service:

```
@DataEnabled(topic="dataTopic", publish=PublishMode.ON_SUCCESS)
public interface MyService {
    ...
}
```

It's generally recommended to put the annotation on the service interface but it can also work when defined on the service implementation. Note that even services that only read data should be annotated this `@DataEnabled` because they also participate in the construction of the message selector.

The attributes of this annotations are :

- `topic`: the name of the messaging topic that will be used to dispatch updates. Obviously this is the one we declared just before in `services-config.xml`.
- `publish`: the publishing mode. `PublishMode.MANUAL` means that you will have to trigger the dispatch manually, for example in an interceptor. `PublishMode.ON_SUCCESS` means that Tide will automatically dispatch the updates on any successful call. `PublishMode.ON_COMMIT` is not yet implemented.
- `params`: a filter class that will define to which updates are sent to which clients. By default there is no filtering, otherwise see below for a more detailed explanation.

**Publishing filters.** It is possible to tell the Tide engine how it should dispatch each update (i.e. to which clients).

It works in two phases : at each remote call from a client, Tide calls the `observes` method of the `params` class and builds the current message selector. Next at each update it calls `publishes` to set the message headers that will be filtered by the selector. Let's see it on an example to be more clear :

```
public class AddressBookParams implements DataTopicParams {

    public void observes(DataObserveParams params) {
        params.addValue("user", Identity.instance().getCredentials().getUsername());
        params.addValue("user", "__public__");
    }
}
```

```

    }

    public void publishes(DataPublishParams params, Object entity) {
        if (((AbstractEntity)entity).isRestricted())
            params.setValue("user", ((AbstractEntity)entity).getCreatedBy());
        else
            params.setValue("user", "__public__");
    }
}

```

The method observes here adds two values to the current selector: the current user name (here retrieved by Seam Identity but could be any other means) and the value `__public__`. From these values Tide will define a message selector (`user = 'username' OR user = '__public__'`) meaning that we only want to be notified of updates concerning public data or data that we own.

During the publishing phase, Tide will call the method `publishes` for each updated entity and build the message headers with the provided values. In the example, an update message will have a user header with either `__public__` or the entity owner for restricted data. These headers are then matched with the current message selector for each subscribed client.

Here we have used only one header parameter but it's possible to define as many as you want. Just take care that the match between observed and published values can become very complex and difficult to predict with too many criteria. When having many header values, the resulting selector is an AND of all criteria :

```

public void observes(DataObserveParams params) {
    params.addValue("user", Identity.instance().getCredentials().getUsername());
    params.addValue("user", "__public__");
    params.addValue("group", "admin");
    params.addValue("group", "superadmin");
}

```

Will generate the following selector :

```
(user = 'username' OR user = '__public__') AND (group = 'admin' OR group = 'superadmin')
```

**Publishing Modes.** There are three publishing modes :

- `PublishMode.ON_SUCCESS` is the easiest to use, and dispatch updates after each successful remote call, regardless of the actual result of the transaction (if there is one).
- `PublishMode.ON_COMMIT` allows for a transactional behaviour, and does the dispatch only on transaction commit.
- `PublishMode.MANUAL` lets you do the dispatch manually in your services when you want, giving the most control.

By default only GraniteDS remoting calls are able to dispatch update messages with `ON_SUCCESS` or `MANUAL` modes. If you need the `ON_COMMIT` mode, or need that services that are not called from a client also trigger the dispatch, then you will have to enable the Tide data dispatcher interceptor that will handle to updates in threads that are not managed by GraniteDS.

To enable the interceptor, it is necessary to indicate on the `@DataEnabled` annotation that there is one with the `useInterceptor` attribute :

```
@DataEnabled(topic="dataTopic", publishMode=PublishMode.ON_COMMIT, useInterceptor=true)
public class MyService {
}
```

There are versions of the interceptor available for each supported framework : EJB3, Spring, CDI.

For Spring, add the advice to your context (take care that you need to reference the latest GraniteDS XSD version 2.3 to allow this) :

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:graniteds="http://www.graniteds.org/config"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/
    beans/spring-beans-3.0.xsd
    http://www.graniteds.org/config http://www.graniteds.org/public/dtd/3.0.0/granite-
    config-3.0.xsd">
  ...
</graniteds:tide-data-publishing-advice/>
```

For CDI, enable the interceptor in `beans.xml` :

```

<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
beans_1_0.xsd">
  <interceptors>
    <class>org.granite.tide.cdi.TideDataPublishingInterceptor</class>
  </interceptors>
</beans>

```

For EJB 3, you can define a global interceptor in `ejb-jar.xml` :

```

<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>org.granite.tide.ejb.TideDataPublishingInterceptor</interceptor-class>
  </interceptor-binding>
  ...
</assembly-descriptor>

```

Or alternatively configure the interceptor on each EJB 3 :

```

@Stateless
@Local(MyService.class)
@Interceptors(TideDataPublishingInterceptor.class)
@DataEnabled(topic="myTopic", publish=PublishMode.ON_COMMIT, useInterceptor=true)
public class MyServiceBean {
  ...
}

```

**Manual publishing.** If you need full control on the publishing process, you can create your own interceptor or use the following API in your services :

```
@DataEnabled(topic="dataTopic", params=DefaultDataTopicParams.class, publishMode=PublishMode.MANUAL,
public class MyService {

    @Inject
    private Gravity gravity;

    public void doSomething() {
        DataContext.init(gravity, "dataTopic", DefaultDataTopicParams.class, PublishMode.MANUAL);
        try {
            Object result = invocation.proceed();
            DataContext.publish(PublishMode.MANUAL);
            return result;
        }
        finally {
            DataContext.remove();
        }
    }
}
```

```
@Interceptor
public class CustomPublishInterceptor {

    @Inject
    private Gravity gravity;

    @AroundInvoke
    public Object aroundInvoke(InvocationContext invocation) throws Exception {
        DataContext.init(gravity, "dataTopic", DefaultDataTopicParams.class, PublishMode.MANUAL);
        try {
            Object result = invocation.proceed();
            DataContext.publish(PublishMode.MANUAL);
            return result;
        }
        finally {
            DataContext.remove();
        }
    }
}
```

**Transactional publishing.** You can setup a fully transactional dispatch by using the ON\_COMMIT mode with a JMS transport. When using JMS transacted sessions with the ON\_COMMIT mode, you will ensure that only successful database updates will be dispatched.

```
<destination id="dataTopic">
  <properties>
    <jms>
      <destination-type>Topic</destination-type>
      <connection-factory>ConnectionFactory</connection-factory>
      <destination-jndi-name>topic/dataTopic</destination-jndi-name>
      <destination-name>dataTopic</destination-name>
      <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
      <transacted-sessions>true</transacted-sessions>
      <no-local>true</no-local>
    </jms>
    <no-local>true</no-local>
    <session-selector>true</session-selector>
  </properties>
  <channels>
    <channel ref="gravityamf"/>
  </channels>
  <adapter ref="jms"/>
</destination>
```



# Extensibility

## 12.1. Writing a security service

GraniteDS implements security based on the following `SecurityService` interface. Note that the term `Service` in `SecurityService` has nothing to do with a true Flex destination, since security services are not exposed to outside calls:

```
package org.granite.messaging.service.security;

import java.util.Map;

public interface SecurityService {

    public void configure(Map<String, String> params);

    public void login(Object credentials) throws SecurityServiceException;

    public void login(Object credentials, String charset) throws SecurityServiceException;

    public Object authorize(AbstractSecurityContext context) throws Exception;

    public void logout() throws SecurityServiceException;

    public void handleSecurityException(SecurityServiceException e);
}
```

An implementation of this interface must be thread safe, i.e., only one instance of this service is used in the entire web-app and will be called by concurrent threads.

- `configure`: This method is called at startup time and gives a chance to pass parameters to the security service.
- `login`: This method is called when you call one of the `setCredentials` or `setRemoteCredentials` `RemoteObject`'s method. Note that these method calls do not fire any request by themselves but only pass credentials on the next destination service method call. The `login` method is responsible for creating and exposing a `java.security.Principal` or throwing an appropriate `org.granite.messaging.service.security.SecurityServiceException` if credentials are

invalid. Note that credentials are a Base64 string with the common "username:password" format. An additional login method with an extra charset parameter is available, so you can use the `RemoteObject.setCredentials(user, pass, charset)` method and specify the charset used in the username/password string (default is ISO-8859-1).

- `authorize`: This method is called upon each and every service method call invocations (`RemoteObject`) or subscribe/publish actions (`Consumer/Producer`). When used with `RemoteObjects`, the `authorize` method is responsible for checking security, calling the service method, and returning the corresponding result. When used with `Consumers/Producers`, it is simply responsible for checking security; no service method invocation, no result. If authorization fails, either because the user is not logged in or because it doesn't have required rights, it must throw an appropriate `org.granite.messaging.service.security.SecurityServiceException`.
- `logout`: This method is called when you call the `RemoteObject`'s `logout` method. Note that the `RemoteObject.logout` method fires a remote request by itself.
- `handleSecurityException`: This method is called whenever a `SecurityServiceException` is thrown by a login or logout operation. The default implementation of this method in `AbstractSecurityService` is to do nothing, but you may add extra care for these security exceptions if you need so.

## 12.2. Custom exception handlers

The default exception handling mechanism of GraniteDS already provides a lot of flexibility with exception converters that can transform the exceptions caught on the server to meaningful errors on the Flex side. However if you need even more flexibility, you can completely replace the handling mechanism and provide you own exception handler. This is however not recommended with Tide as some features rely on proper exception conversions to work, but in this case you can simply extend the `ExtendedExceptionHandler` and add you custom behaviour.

If you need special service exception handling, either to add extra informations or to mask implementation details, you may configure a custom implementation of `ServiceExceptionHandler` in `services-config.xml`:

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

<services-config>
  ...
  <factories>
    <factory id="..." class="...">
      <properties>
        <service-exception-handler>
          path.to.my.CustomServiceExceptionHandler
        </service-exception-handler>
      </properties>
    </factory>
  </factories>
</services-config>
```

```

...
</properties>
</factory>
</factories>
...
</services-config>

```

Your custom service exception handler must implement the `org.granite.messaging.service.ServiceExceptionHandler` interface. Note that it can of course extend the `org.granite.messaging.service.DefaultServiceExceptionHandler` class:

```

public ServiceException handleNoSuchMethodException(
    Message request,
    Destination destination,
    Object invokee,
    String method,
    Object[] args,
    NoSuchMethodException e
);

public ServiceException handleInvocationException(
    ServiceInvocationContext context,
    Throwable t
);

```

The first method is called whenever the service invoker cannot find any suitable method with the supplied name and arguments.

The second one is called whenever the method invocation throws an exception. Note that `java.lang.reflect.InvocationTargetException` are unwrapped (`getTargetException`) before `handleInvocationException` is called.

In both cases, the returned `ServiceException` will be thrown and serialized in a `Flex ErrorMessage` instead of the raw `NoSuchMethodException e` or `Throwable t` one.

## 12.3. Server message interceptors

If you need to do some actions before and after each remote call, such as setting or accessing message headers, or doing some setup before request handling, you can configure a custom `AMF3MessageInterceptor` in `granite-config.xml`:

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

<granite-config>
    ...
    <amf3-message-interceptor type="com.myapp.MyMessageInterceptor"/>
</granite-config>
```

When using configuration scanning, you can also put this in `META-INF/granite-config.properties` of your application jar archive :

```
amf3MessageInterceptor=com.myapp.MyMessageInterceptor
```

When using Spring or CDI, you will just have to declare a bean implementing `AMF3MessageInterceptor` in the framework context (with CDI, that just means adding an implementation in the application archive).

Take care that some of the GraniteDS server frameworks integrations (CDI and Seam) already provide their own message interceptors. If you need to do something else, you will have to override the existing interceptor and call `super.before` and `super.after`.

### 12.4. Custom AMF3 (De)Serializers (Advanced use only)

You may plug your own AMF3 serializer/deserializer. A custom AMF3 serializer must implement `java.io.ObjectOutput` and have a special constructor signature:

```
public class MyAMF3Serializer implements java.io.ObjectOutput {

    public MyAMF3Serializer(java.io.OutputStream out) {
        // ...
    }

    // ObjectOutput implementation...
}
```

Then, you must register this serializer in `granite-config.xml`:

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

<!DOCTYPE granite-config PUBLIC
  "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <amf3-serializer type="path.to.MyAMF3Serializer"/>
</granite-config>
```

A custom AMF3 deserializer must implement `java.io.ObjectInput` and have a special constructor signature:

```
public class MyAMF3Deserializer implements java.io.ObjectInput {

    public MyAMF3Deserializer(java.io.InputStream in) {
        // ...
    }

    // ObjectInput implementation...
}
```

Then, you have to register this deserializer in `granite-config.xml`:

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

<!DOCTYPE granite-config PUBLIC
  "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <amf3-deserializer type="path.to.MyAMF3Deserializer"/>
</granite-config>
```

You may of course extend `org.granite.messaging.amf.io.AMF3Serializer` or `org.granite.messaging.amf.io.AMF3Deserializer` to override only some parts of the default AMF3 (de)serialization process, as all methods in those classes are public or protected.

### 12.5. ServiceInvocationListener (Advanced use only)

If you need to listen to each service invocation method call, you may plugin a `org.granite.messaging.service.ServiceInvocationListener` implementation like this:

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

<!DOCTYPE granite-config PUBLIC
  "-//Granite Data Services//DTD granite-config internal//EN"
  "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">

<granite-config>
  <invocation-listener type="path.to.MyServiceInvocationListener"/>
</granite-config>
```

Your class must implement the `org.granite.messaging.service.ServiceInvocationListener` interface containing the following methods:

```
public Object[] beforeMethodSearch(Object invokee, String methodName, Object[] args);
public void beforeInvocation(ServiceInvocationContext context);
public void afterInvocationError(ServiceInvocationContext context, Throwable t);
public Object afterInvocation(ServiceInvocationContext context, Object result);
```



#### Warning

Be very careful with those listeners as you may break the entire invocation process if you do not return proper args (`beforeMethodSearch`), if you modify the `ServiceInvocationContext` (`beforeInvocation`) or if you return a different object than the service method call result (`afterInvocation`)!

## Configuration Reference

The two main files used to configure GraniteDS are `granite-config.xml` and `services-config.xml`. By default these files should be present in the web archive in `WEB-INF/granite/granite-config.xml` and `WEB-INF/flex/services-config.xml`.

If absolutely needed, they can be placed in another location, but then you will have to specify two servlet parameters in `web.xml` to indicate GraniteDS where to look for them:

```
<context-param>
  <param-name>servicesConfigPath</param-name>
  <param-value>/WEB-INF/flex/services-config.xml</param-value>
</context-param>
<context-param>
  <param-name>graniteConfigPath</param-name>
  <param-value>/WEB-INF/granite/granite-config.xml</param-value>
</context-param>
```

### 13.1. Framework Configuration

`granite-config.xml` contains all the internal configuration of the framework. It can contain the following sections:

- `<granite scan="true">`: instructs GraniteDS to scan application archives and to automatically register the configuration elements it discovers.
- `<amf3-deserializer type="com.myapp.custom.CustomAMF3Deserializer">`: registers a custom deserializer that should implement the interface `java.io.ObjectInput`. The default is `org.granite.messaging.amf.io.AMF3Deserializer`.
- `<amf3-serializer type="com.myapp.custom.CustomAMF3Serializer">`: registers a custom serializer that should implement the interface `java.io.ObjectOutput`. The default is `org.granite.messaging.amf.io.AMF3Serializer`.
- `<amf3-message-interceptor type="">`: registers an optional message interceptor that should implement `org.granite.messaging.amf.process.AMF3MessageInterceptor`.
- `<class-getter type="">`: registers a class getter that should implement `org.granite.messaging.amf.io.util.ClassGetter`.
- `<converters>`: registers a list of data converters that should implement `org.granite.messaging.amf.io.convert.Converter`.

- `<descriptors>`: registers a list of type descriptors that should extend either `org.granite.messaging.amf.io.util.ActionScriptClassDescriptor` or `org.granite.messaging.amf.io.util.JavaClassDescriptor`.
- `<exception-converters>`: registers a list of exception converters that should implement `org.granite.messaging.service.ExceptionConverter`.
- `<externalizers>`: registers custom externalizers that should implement `org.granite.messaging.amf.io.util.externalizer.Externalizer`. See also [here](#).

```
<externalizers>
  <configuration>
  </configuration>
  <externalizer type=""/>
</externalizers>
```

- `<gravity>`: configures the Gravity internal parameters. See [here](#).
- `<instantiators>`: registers custom instantiators that should implement `org.granite.messaging.amf.io.util.instantiator.AbstractInstantiator`.
- `<invocation-listener type="">`: registers an invocation listener that will be called at each invocation and should implement `org.granite.messaging.service.ServiceInvocationListener`.
- `<message-selector>`: registers a custom message selector implementation that should implement `org.granite.gravity.selector.MessageSelector`. 3 implementations are available, the default is `GravityMessageSelector`.
- `<method-matcher type="">`: registers a custom method matcher that should implement `org.granite.messaging.service.MethodMatcher`.
- `<security>`: registers a custom security service that should implement `org.granite.messaging.service.security.SecurityService`.
- `<tide-components>`: registers a list of component matchers to enable remote access for Tide service factories. There are 4 ways of enabling or disabling access to Tide components:

```
<tide-components>
  <tide-component annotated-with=""/>
  <tide-component instance-of=""/>
  <tide-component name=""/>
```

```
<tide-component type="" disabled="true"/>
</tide-components>
```

annotated-with: component class is annotated with the specified annotation class. instance-of: component class extends or implements the specified interface or class. name: component name matches the specified name regular expression. type: component class matches the specified class name regular expression.

## 13.2. Application Configuration

services-config.xml contains all the remoting and messaging configuration of the application. There are three main sections: channels, factories and services.

### 13.2.1. Factories

A factory defines a way to tell GraniteDS how to route incoming remoting calls to a server component. A factory should implement `org.granite.messaging.service.ServiceFactory`. The factory definition can also have configuration options in the section properties:

```
<factory id="myFactory" class="com.myapp.custom.MyServiceFactory">
  <properties>
    <service-exception-handler>com.myapp.custom.MyServiceExceptionHandler</service-exception-handler>
    <enable-exception-logging>true</enable-exception-logging>
  </properties>
</factory>
```

service-exception-handler: an exception handler should implement `org.granite.messaging.service.ServiceExceptionHandler` and will be called when an exception is thrown by the remote service. The default is `DefaultServiceExceptionHandler` for standard factories and `ExtendedServiceExceptionHandler` for Tide factories.

enable-exception-logging: enables (true) or disable (false) the logging of exceptions thrown by remote services. This can avoid double logging if the server application already logs everything. Default is true.

Other properties exist for the built-in service factories. You will get more details in the corresponding sections. For example EJB3 factories have a lookup and initial-context-environment properties.

### 13.2.2. Remoting destinations

Remoting destinations can be defined in a service definition with the class property value `flex.messaging.services.RemotingService` and the `messageTypes` value `flex.messaging.messages.RemotingMessage`. Destinations can also have a `properties` section and in general they will define at least the `factory` and the `channels` they are attached to.

```
<services>
  <service
    id="granite-service"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <destination id="cars">
      <channels>
        <channel ref="my-graniteamf"/>
      </channels>
      <properties>
        <factory>guiceFactory</factory>
        <source>test.granite.guice.services.Cars</source>
      </properties>
    </destination>
  </service>
</services>
```

A destination can also define a list of security roles that are allowed to access the remote component. See [Remoting security](#).

### 13.2.3. Messaging destinations

Messaging destinations can be defined in a service definition with the class property value `flex.messaging.services.MessagingService` and the `messageTypes` value `flex.messaging.messages.AsyncMessage`. Destinations can also have a `properties` section that is used for example with the JMS adapter.

A messaging service can also define a list of service adapters that define how messages are routed and each destination can reference one of the configured adapters.

```
<service id="gravity-service"
  class="flex.messaging.services.MessagingService"
  messageTypes="flex.messaging.messages.AsyncMessage">
```

```

<adapters>
  <adapter-definition id="simple" class="org.granite.gravity.adapters.SimpleServiceAdapter"/>
  <!--adapter-definition id="jms" class="org.granite.gravity.adapters.JMSServiceAdapter"/-->
</adapters>

<destination id="addressBookTopic">
  <properties>
    <!--jms>
      <destination-type>Topic</destination-type>
      <connection-factory>ConnectionFactory</connection-factory>
      <destination-jndi-name>topic/testTopic</destination-jndi-name>
      <destination-name>dataTopic</destination-name>
      <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
      <transacted-sessions>true</transacted-sessions>
      <no-local>true</no-local>
    </jms-->
    <no-local>true</no-local>
    <session-selector>true</session-selector>
  </properties>
  <channels>
    <channel ref="gravityamf"/>
  </channels>
  <adapter ref="simple"/>
  <!--adapter ref="jms"/-->
</destination>
</service>

```

You can define multiple channels for the same destination to handle failover. When the first channel cannot be accessed, the remote object will try the next one in the list.

A destination can also define a list of security roles that are allowed to access the remote component. See [Messaging Security](#).



# Appendix

## 14.1. GraniteDS config DTD

You can reference the GraniteDS Configuration DTD with:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE granite-config PUBLIC
    "-//Granite Data Services//DTD granite-config internal//EN"
    "http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd">
<granite-config>
    ...
</granite-config>
```

The DTD is online and can be found [here](http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd) [http://www.graniteds.org/public/dtd/3.0.0/granite-config.dtd].

## 14.2. GraniteDS Spring/Seam Configuration XSD

You can reference the GraniteDS Configuration XSD in your Spring or Seam configuration with:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    ...
    xmlns:graniteds="http://www.graniteds.org/config"
    xsi:schemaLocation="
        ...
        http://www.graniteds.org/config http://www.graniteds.org/public/dtd/3.0.0/granite-
        config-3.0.xsd">
    ...
</beans>
```

The XSD is online and can be found [here](http://www.graniteds.org/public/dtd/3.0.0/granite-config-3.0.xsd) [http://www.graniteds.org/public/dtd/3.0.0/granite-config-3.0.xsd].

### 14.3. Release notes

See the [GraniteDS JIRA](http://www.graniteds.org/jira/browse/GDS?report=com.atlassian.jira.plugin.system.project:roadmap-panel#selectedTab=com.atlassian.jira.plugin.system.project%3Achangelog-panel) [http://www.graniteds.org/jira/browse/GDS?report=com.atlassian.jira.plugin.system.project:roadmap-panel#selectedTab=com.atlassian.jira.plugin.system.project%3Achangelog-panel].