Errai Reference Guide

Introduction

1.1. What is it?

Errai is a GWT-based framework for building rich web applications using next-generation web technologies. Built on-top of ErraiBus, the framework provides a unified federation and RPC infrastructure with true, uniform, asynchronous messaging across the client and server.

1.2. Required software

Errai requires a JDK version 6 or higher and depends on Apache Maven to build and run the examples, and for leveraging the quickstart utilities.

- JDK 6.0: *http://java.sun.com/javase/downloads/index.jsp*
- Apache Maven: *http://maven.apache.org/download.html*

1.3. Getting Started with Errai

Errai is a framework which combines a constellation of web and server-side technologies to assist you in developing large, scaleable rich web applications using a consistent, standardized programming model for client and server development.

1.3.1. Technology Primer

Since Errai is an end-to-end framework, in that, parts of the framework run and operate within the client and parts run and operate within the server, there is a set of various technologies upon which Errai relies. This section will detail the basic core technologies which you'll need to be familiar with.

1.3.1.1. Google Web Toolkit (GWT)

GWT is a toolkit built around a Java-to-JavaScript compiler. It provides a JRE emulation library, abstraction of browser quirks, a development mode runtime, and tools for native JavaScript integration.

Errai uses GWT to accomplish the translation of concepts such as CDI into the browser, which enables a consistent client and server programming experience.

1.3.1.2. Contexts and Dependency Injection (CDI)

CDI is a standard part of the Java EE 6.0 stack, and is defined in the *JSR-299* [http://jcp.org/en/jsr/detail?id=299] specification. CDI is the main programming model explored in this guide. As such, the basic concepts of CDI will be introduced in this guide, so pre-existing knowledge is not strictly necessary.

Errai's support for CDI is two-fold. For the server-side, Errai has integration with Weld, which is the reference implementation (RI) of the JSR-299 specification. The client-side integration for CDI is

provided by the Errai CDI extension. Errai CDI implements a subset of the JSR-299 specification to provide the CDI programming model within client code.

1.3.1.3. Java API for RESTful Web Services (JAX-RS)

JAX-RS is an API which provides a standardized programming model for specifying web services based around the concept of the Representational State Transfer (REST) architecture. REST has by and far become the preferred way of developing web services, and is used pervasively in modern web applications. Errai provides a set of tools to make working with JAX-RS easier.

1.3.1.4. ErraiBus

ErraiBus is an underlying transport technology which provides true, bidirectional, asynchronous messaging between the client and the server. It powers a myriad of technologies throughout the Errai framework, from RPC to CDI Events.

1.3.2. Creating your first project

Maven Required

The first thing you'll need to do if you have not already, is *install Maven* [http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html] . If you have not already installed Maven, do so now.

Warning: If you use maven2, you will run into this problem: *https://community.jboss.org/thread/177645*

You have two options to set up an Errai application. You can start by copying an existing example application (i.e. the *errai tutorial demo* [https://github.com/errai/errai-tutorial/archive/master.zip]) or by building an app with the Errai Forge Addon:

1.3.2.1. Start from a working example application

Simply download and unzip this *demo* [https://github.com/errai/errai-tutorial/archive/master.zip]. Check out the README file and continue with running the app in *GWT's development mode* and importing the project into *Eclipse* .

1.3.2.2. Starting with the Errai Forge Addon

Another way to start a new project with Errai is to use Forge and the Errai Forge Addon. To use this method, follow the instructions *here* [https://github.com/errai/errai/blob/master/errai-forge-addon/README.asciidoc] to install the Errai Forge Addon and create a new project.

In the upcomming sections, we will demonstrate how to run your app in GWT Development Mode through the command line and eclipse, so it would be nice to have something to run so that you are able to verify that everything is working. Here is a sample class you can use that displays an alert when the app loads:

```
// Add the package declaration here

import javax.annotation.PostConstruct;
import org.jboss.errai.ioc.client.api.EntryPoint;
import com.google.gwt.user.client.Window;

@EntryPoint
public class App {

  @PostConstruct
  public void onLoad() {
    Window.alert("Hello World!");
  }
}
```

For this code to run properly, you must use the Errai Forge Addon *Add Errai Features* command to install *Errai IOC*.

Create new subfolder, *client/local*, under the folder containing your GWT module file. Then create a file, *App.java*, in this new package and copy the above sample code (making sure to replace the top comment with the package declaration).

Tip

Plugin Tips. Keep an eye out for tips in the proceeding sections on how you can use the Errai Forge plugin to configure other Errai features for your new project.

1.3.3. Running and Debugging with GWT's Super Dev mode

As of GWT 2.7, we use Super Dev Mode to run your app in hosted mode during development. *GWT's Super Dev Mode* [http://www.gwtproject.org/doc/latest/DevGuideCompilingAndDebugging.html#DevGuideDevMode] allows for code-refresh development cycles. Simply change a client-side class and refresh the browser to see your changes. You can also debug client and server side code in your IDE of choice. These sections will describe how to use Super Dev Mode to develop and debug your web app.

1.3.3.1. Running the app in GWT

Change into your web app's project directory and type the following:

```
mvn clean gwt:run
```

This will begin the download of all the dependencies required to develop with and run Errai. It may take a few minutes to complete the download.

When it is finished, you should see the GWT Development Mode runtime window appear as shown in *Figure 1*.

Figure 1.1. GWT Development Mode

Next, click the *Launch Default Browser* button. You should now see the application load.

If you are using errai-tutorial, you should see a page with a complaint form.

If you followed the instructions for using the Errai Forge plugin, there should be a blank page with an alert saying "Hello World!".

That's it! You've got your first Errai Application up and running. The next sections will describe how to set up your app with your IDE.

1.3.3.2. Debugging the app in GWT

To debug your app using GWT's development mode, you will need to start a remote debugger separately and run the following command in your project directory:

```
mvn clean gwt:debug
```

1.3.4. Running and Debugging in Eclipse using Maven tooling

There are two ways to configure your project in your IDE. One option is to use the built-in Maven tooling to set up Maven run and debug configurations within your IDE. The other option is to download and use GWT tooling that is available for your IDE, and create custom run/debug configurations with that. Note that for Intellij IDEA, the GWT tooling is only available for users of the Ultimate Edition.

Read the previous section!

This next section assumes you have followed the instructions in the previous section. As such, we assume you have created an Errai project using the Errai Forge plugin or a copy of the errai-tutorial project, which we'll be importing into your IDE.

1.3.4.1. Prerequisites

1.3.4.2. Maven Integration for Eclipse (m2e)

This section will walk you through using Maven tooling for running and debugging your app within Eclipse. If you have not already installed m2e in Eclipse, you will want to do so now.

To install the Maven tooling, use the following steps:

1. Go to the *Eclipse Marketplace* under the *Help* menu in Eclipse.

Figure 1.2. Eclipse Marketplace

2. In the *Find* dialog enter the phrase *Maven* and hit enter.

Figure 1.3. Find Dialog

3. Find the *Maven Integration for Eclipse* plugin and click the *Install* button for that entry.

Figure 1.4. Maven Integration for Eclipse in Marketplace

4. Accept the defaults by clicking *Next* , and then accept the User License Agreement to begin the installation.

1.3.4.3. Importing your project

Once you have completed the installation of the prerequisites from the previous section, you will now be able to go ahead and import the Maven project you created in the first section of this guide. We will use the errai-tutorial project as an example.

Follow these steps to get the project setup:

1. From the *File* menu, select *Import…*

Figure 1.5. Import File in Eclipse

2. You will be presented with the Import dialog box. From here you want to select *Maven* → *Existing Maven Projects*

Figure 1.6. Import Existing Maven Project

3. From the *Import Maven Projects* dialog, you will need to select the directory location of the project you created in the first section of this guide. In the *Root Directory* field of the dialog, enter the path to the project, or click *Browse…* to select it from the file chooser dialog.

Figure 1.7. Select Folder

4. Click *Finish* to begin the import process.

5. When the import process has finished, you should see your project imported within the Eclipse *Project Explorer* . If you are using errai-tutorial, the `App` class should be visible within the `client` package.

Figure 1.8. App.java

1.3.4.4. Running Development Mode with Eclipse

1. Next you will need to setup a Maven Run Profile for Development Mode. To do so select *Run As… > Run Configurations…* from the toolbar.

Figure 1.9. Run Configurations

2. Select *Maven Build* from the sidebar and create a new launch configuration by pressing the *New* button in the top left corner.

Figure 1.10. New Configuration

3. Give the configuration a name, then click *Browse Workspace* and select the root directory of your new project.

Figure 1.11. Select Project Root Directory

4. In the *Goals* text box, type "clean gwt:run". Click *Apply* to save the configuration and then *Close*.

Figure 1.12. Run Configurations Goals

5. You can add this new configuration under the *Run As* button in your toolbar by selecting *Run As > Organize Favorites*, then clicking *Add* and selecting the run configuration.

Figure 1.13. Add Configuration to Favourites

6. At this point, you should try running your new configuration to make sure everything is in working order. To run your app, find the run configuration under the *Run As* menu in the toolbar.

Figure 1.14. Run Gwt Development Mode from Eclipse

This will start the GWT Development Mode exactly as running `mvn clean gwt:run` from the command line.

1.3.4.5. Debugging in Development Mode with Eclipse

1. To setup a debug run configuration for GWT Development Mode, repeate steps (2) and (3) from the section above, but this time use the *Goals* "clean gwt:debug".

Figure 1.15. Configure Maven Debug Configuration

2. Next we will need to setup our remote debugger configurations in Eclipse. Because the client and server code run on separate JVMs, we will need to setup two such configurations. To create a debug configuration, select *Debug As… > Debug Configurations…* from the toolbar.

Figure 1.16. Create Debug Configuration

3. In the sidebar, select *Remote Java Application* and click the *New* button in the top right corner.

Figure 1.17. Create Remote Debug Configuration

4. This new configuration will be for remote debugging your client-side code, so give it an appropriate name. If the name of your project is not already in the *Project* field, click *Browse* and select it. The *Host* and *Port* values should be *localhost* and *8000* respectively, such that your configuration looks like this:

Figure 1.18. Client Debug Configuration

   If everything is correct, click *Apply*.

5. Create another *Remote Java Application* run configuration with the steps just described for remote debugging server code. The only differences from the client configuration should be the name and the port, which is *8001*. Thus the server remote debug configuration should look like this:

Figure 1.19. Server Debug Configuration

6. That's it! You've successfully imported your Errai project into Eclipse. Now, on to *coding*!

1.3.5. Running and Debugging in your IDE using GWT tooling

Errai's *EmbeddedWildFlyLauncher* provides an embedded WildFly instance within Dev Mode to allow users to debug server- and client-side code simultaneously in a single JVM. Here are the instructions for using it in Eclipse and Intellij IDEA Ultimate Edition:

1.3.5.1. Configuring your app to use the Google Plugin for Eclipse

This method requires the Google Plugin for Eclipse (GPE). You can find the instructions to download and install it *here* [https://developers.google.com/eclipse/docs/download].

Here are the steps to setup a run/debug configuration using the embedded WildFly launcher:

1. Ensure that your app is using the Google Web Toolkit. Right-click on your project in Eclipse, and select Google → Web Toolkit Settings, and make sure that the "Use Google Web Toolkit" box is checked and the appropriate GWT SDK is selected.

Figure 1.20. Google Web Toolkit Settings

2. Set up a Web Application run/debug configuration as follows:

   Create a new Web Application run/debug configuration, select your Errai app in the Project field and set `com.google.gwt.dev.DevMode` as the main class:

Figure 1.21. Dev Mode Configuration: Main tab

3. In the *Server* tab, check the *Run built-in server* checkbox, and enter the port number `8888`.

Figure 1.22. Dev Mode Configuration: Server tab

4. Select *Super Development Mode* in the GWT tab, and add your GWT module under *Available Modules*.

Figure 1.23. Dev Mode Configuration: GWT tab

5. Under the *Arguments* tab, there are two fields, *Program arguments* and *VM arguments*. In the *Program arguments* field, amend the `server` flag as follows:

   Program Arguments.

   ```
   -server org.jboss.errai.cdi.server.gwt.EmbeddedWildFlyLauncher
   ```

   Make sure the following VM arguments are set:

   VM arguments.

   ```
   -Xmx2048m -XX:MaxPermSize=512M -Derrai.jboss.home=/home/ddadlani/errai-tutorial/target/wildfly-8.2.0.Final/
   ```

   where `errai.jboss.home` points to your own WildFly installation. This can either be your local WildFly installation directory, or in the *target/* directory of your app. For the Errai Tutorial app, `errai.jboss.home` points to the WildFly installation within the *target/* directory, which is redownloaded and installed as part of the build. See *below* for information on how to automatically download WildFly into the *target/* directory prior to running Super Dev Mode.

Figure 1.24. Dev Mode Configuration: Arguments tab

Copy of WildFly in the target/ directory

If you wish to automatically download and install WildFly as part of your build into the *target/* directory of your web app, you will need to run `mvn process-resources` prior to running or debugging your app. To do this, you can set up a Maven build configuration in Eclipse and specify *Goals*: "process-resources". You can then run the Maven configuration prior to running GWT's Super Dev Mode.

That's it! To debug your application within Eclipse, simply select your debug configuration and hit *Debug*. This will start GWT's dev mode as well as the debugger within Eclipse.

1.3.5.1.1. Debugging client-side JavaScript from within Eclipse

If you wish to debug your client-side JavaScript as Java code from within Eclipse, you will need to use the SDBG plugin for GWT Super Dev Mode. More information can be found *here* [http://sdbg.github.io/].

1.3.5.2. Configuring your app to use Intellij IDEA's GWT tooling

Intellij IDEA Community Edition users

The inbuilt GWT tooling for Intellij IDEA is only available in the Ultimate Edition. If you are using the Community Edition, you will not have access to the GWT plugin.

The Ultimate Edition for Intellij IDEA comes with a built-in GWT plugin that allows you to run and debug GWT apps specifically. We can configure the plugin to use the embedded WildFly launcher for debugging ease, in order to debug both server and client-side code in one debug session. This section describes how to set up a GWT run/debug configuration within Intellij IDEA:

1. If you have not already done so, add a GWT facet to your main module. Instructions to add a GWT facet to an existing module can be found *here* [https://www.jetbrains.com/idea/help/adding-a-gwt-facet-to-a-module.html].
2. On the top right hand corner of your Intellij IDEA session, select the dropdown box labeled *Edit Configurations* to create a new run/debug configuration.

Figure 1.25. Edit Run/Debug Configuration

3. Press the + button under GWT Configurations to create a new GWT configuration. Check the *Super Dev Mode* checkbox and select the name of your module. Fill out the following parameters in the corresponding boxes:

   VM Options.

   ```
   -Xmx2048m -XX:MaxPermSize=512M -Derrai.jboss.home=/home/ddadlani/errai-tutorial/target/wildfly-8.2.0.Final/
   ```

   where `errai.jboss.home` points to your WildFly installation directory. This can either be your local WildFly installation directory, or in the *target/* directory of your app. For the Errai Tutorial app, `errai.jboss.home` points to the WildFly installation within the *target/* directory, which is redownloaded and installed as part of the build. See *below* for information on how to automatically download WildFly into the *target/* directory prior to running Super Dev Mode.

   Dev Mode Parameters.

   ```
   -server org.jboss.errai.cdi.server.gwt.EmbeddedWildFlyLauncher
   ```

Figure 1.26. GWT Run/Debug Configuration for Intellij IDEA

4. Select the *Default* server and make sure the start page is *index.html*. Select your browser and check the *with JavaScript debugger* box to take advantage of Intellij IDEA's built-in JavaScript tooling.

5. Click on the + button under *Before Launch* and select the *Make* option. This will tell Intellij to compile your project before launching Dev Mode.

Copy of WildFly in the target/ directory

If you wish to automatically download and install WildFly as part of your build into the *target/* directory of your web app, you will need to run `mvn process-resources` prior to running or debugging your app. To do this, you can add a Maven goal to the *Before Launch* task list. Click the + button as before, select *Run Maven Goal* and in the *Command Line* field, enter "process-resources". Ensure that the working directory is the same as your project directory before clicking "OK". Intellij IDEA will then run the Maven goal prior to starting Super Dev Mode.

To run or debug your app, select this configuration in the top right corner of Intellij IDEA and click the *Run* or *Debug* buttons next to it. Your app should start up in Dev Mode within Intellij automatically and you should be able to use Intellij's own debugger to debug your code.

Now that you have everything set up, it's time to move on to coding!

1.3.6. A Gentle Introduction to CDI

This section is based on the previous guide sections

The project you created and setup in the previous two sections (ERRAI:Create your Project and ERRAI:Configuring your project for Eclipse) will be used as the basis for this section. So if you have not read them, do so now.

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai CDI* to follow along with this section.

Errai CDI as its namesake implies is based on, and is in fact, a partial implementation of the CDI (Contexts and Dependency Injection) specification. Errai CDI covers *most* of the programming model but omits the CDI SPI, instead replacing it with it a custom set of APIs which are more appropriate for the client programming model of Errai.

These differences aside, using Errai CDI in conjunction with CDI on the server will provide you with a uniform programming model across the client and server code of your application.

This guide does not assume any past experience with CDI. However, you may wish to consider reading the *Weld Documentation* [http://docs.jboss.org/weld/reference/1.1.5.Final/en-US/html/] in addition to this guide.

1.3.6.1. Your First Bean

A bean in CDI is merely a POJO (Plain Old Java Object), for the most part. In the context of CDI, any plain, default constructable class is a member of the *dependent scope* . Don't worry too much about what that means for now. Let's just go ahead and make one:

```
public class Foo {
  public String getName() {
    return "Mr. Foo";
  }
}
```

That was an easy, if uninteresting, exercise. But despite this class' worthy distinction as a dependent-scoped bean, it's actually quite a useless dependent scope beaned. Well, maybe not so much useless as it is unused.

Well, how would we use this bean? To answer that question we're going to need to introduce the concept of scopes in more detail.

1.3.6.2. Scopes

Scopes, put simply, are the context within which beans live. Some scopes are short-lived and some are long-lived. For instance, there are beans which you may only want to create during a request, and beans which you want to live for as long as the application is running.

It turns out that CDI includes a set of default scopes which represent these very things.

We'll start by taking a look at the *application scope*, which is lovingly represented by the annotation `@ApplicationScoped`. An application-scoped bean is a bean which will live for the entire duration of the application. In this sense, it is essentially like a singleton. And it's generally okay to think of it in that way.

So let's declare an application-scoped bean:

```
@ApplicationScoped
public class Bar {
  public String getName() {
    return "Mr. Bar";
  }
}
```

That was *almost* as easy as making the last bean. The difference between this bean and the last, is `Bar` will actually be instantiated by the container automatically, and `Foo` will not.

So what can we do with `Foo` ? Well, let's go ahead and get familiar with dependency injection, shall we?

```
@ApplicationScoped
public class Bar {
  @Inject Foo foo;

  public String getName() {
    return "Mr. Bar";
  }
}
```

We have added a field of the type `Foo` which we declared earlier, and we have annotated it with `javax.inject.Inject`. This tells the container to inject an instance of `Foo` into our bean. Since our `Foo` bean is of the dependent scope, the bean manager will actually create a *new* instance of `Foo` and pass it in.

This scope of the newly instantiated `Foo` is *dependent* on the scope that it was injected into. In this case, the application scope. On the other hand, if we were to turn around an inject `Bar` into `Foo` , the behaviour is quite different.

```
public class Foo {
  @Inject Bar bar;

  public String getName() {
    return "Mr. Foo";
  }
}
```

Here, every time a new instance of `Foo` is created, the *same* instance of `Bar` will be injected. That is to say: this pseudo-code assertion is now always true:

```
assert fooInstance.bar.foo == fooInstance
```

Note

This identity check will not *actually* be true at runtime due to the need to proxy the class in this scenario. But it is true, that `fooInstance` and `fooInstance.bar.foo` both point to the same *underlying* bean instance.

In the case of an Errai application, there are a bunch of application scoped beans which come built-in for common services like *ErraiBus*. Thus, in an Errai application which uses the message bus, we can inject a handle to the `MessageBus` service into any of our beans. Let's go ahead and do that in our `Bar` class:

```
@ApplicationScoped
public class Bar {
  @Inject Foo foo;
  @Inject MessageBus bus;

  public String getName() {
    return "Mr. Bar";
  }
}
```

If working with dependency injection is new to you, then this is where you'll start seeing some practical benefit. When you need a common service in your client code, you ask the container for it by *injecting* it. This frees you from worrying about the proper APIs to use in order to access a service; we need to use the message bus in our `Bar` bean, and so we inject it.

1.3.6.3. EntryPoints

Now that we're getting the gist of how dependency injection works, let's go back to our sample project.

In the `App` class that was created you may have noticed that the bean's scope is `@EntryPoint`.

The `@EntryPoint` annotation is an annotation which provides a an analogue to the GWT EntryPoint concept within the context of CDI in Errai. Basically you want to think of `@EntryPoint` beans as the Errai CDI-equalivalent of `main()` methods. But as of Errai 2.2., that might actually be going a little far. In fact, you might be asking what is the real difference between `@ApplicationScoped` and `@EntryPoint` in practice. The short answer is: nothing.

When Errai IOC, the technology which powers Errai's client-side CDI, was first built, it lacked the concept of scopes. To create entry point objects into the application which would automatically run, this annotation was added.

If you're not convinced, try running this example with the `mvn clean gwt:run` command (described *above*).

Launching maven the first time

Please note, that when launching maven the first time on your machine, it will fetch all dependencies from a central repository. This may take a while, because it

includes downloading large binaries like GWT SDK. However, subsequent builds are not required to go through this step and will be much faster.

Messaging

This section covers the core messaging concepts of the ErraiBus messaging framework.

ErraiBus provides a straight-forward approach to a complex problem space. Providing common APIs across the client and server, developers will have no trouble working with complex messaging scenarios such as building instant messaging clients, stock tickers, to monitoring instruments. There's no more messing with RPC APIs, or unwieldy AJAX or COMET frameworks. We've built it all in to one concise messaging framework. It's single-paradigm, and it's fun to work with.

2.1. Messaging Overview

It's important to understand the concept of how messaging works in ErraiBus. Here are some important facts you'll need to know:

- Service endpoints are given string-based names that are referenced by message senders.
- There is no difference between sending a message to a client-based service, or sending a message to a server-based service.
- Furthermore, a service of the same name may co-exist on both the client and the server and both will receive all messages bound for that service name, whether they are sent from the client or from the server.
- Services are lightweight in ErraiBus, and can be declared liberally and extensively within your application to provide a message-based infrastructure for your web application.

It can be tempting to think of ErraiBus simply as a client-server communication platform, but there is a plethora of possibilities for using ErraiBus purely within the GWT client context, such as a way to advertise and expose components dynamically, to get around the lack of reflection in GWT.

So keep that in mind when you run up against problems in the client space that could benefit from runtime federation.

Plugin Tip

Use the *Errai Forge Addon Add Errai Features* command and select *Errai Messaging* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add messaging to your project.

2.2. Messaging API Basics

The MessageBuilder is the heart of the messaging API in ErraiBus. It provides a fluent / builder API, that is used for constructing messages. All three major message patterns can be constructed from the `MessageBuilder`.

Components that want to receive messages need to implement the `MessageCallback` interface.

But before we dive into the details, let's look at some use cases.

2.2.1. Sending Messages with the Client Bus

In order to send a message from a client you need to create a `Message` and send it through an instance of `MessageBus` . In this simple example we send it to the subject *HelloWorldService*.

```
public class HelloWorld implements EntryPoint {

  // Get an instance of the RequestDispatcher
  private RequestDispatcher dispatcher = ErraiBus.getDispatcher();

  public void onModuleLoad() {
    Button button = new Button("Send message");

    button.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
        // Send a message to the 'HelloWorldService'.
        MessageBuilder.createMessage()
          .toSubject("HelloWorldService") // (1)
          .signalling() // (2)
          .noErrorHandling() // (3)
          .sendNowWith(dispatcher); // (4)
        });

        [...]
      }
   }
}
```

In the above example we build and send a message every time the button is clicked. Here's an explanation of what's going on as annotated above:

1. We specify the subject we wish to send a message to. In this case, " `HelloWorldService` ".
2. We indicate that we wish to only signal the service, meaning, that we're not sending a qualifying command to the service. For information on this, read the section on *Protocols*.
3. We indicate that we do not want to provide an `ErrorCallback` to deal with errors for this message.

4. We transmit the message by providing an instance to the `RequestDispatcher`

Important

An astute observer will note that access to the `RequestDispatcher` differs within client code and server code. Because this client code does not run within a container, access to the `RequestDispatcher` and `MessageBus` is provided statically using the `ErraiBus.get()` and `ErraiBus.getDispatcher()` methods. See the section on Errai IOC and Errai CDI for using ErraiBus from a client-side container.

When using Errai IOC or CDI, you can also use the Sender<T> interface to send messages.

2.2.2. Receiving Messages on the Server Bus / Server Services

Every message has a sender and at least one receiver. A receiver is as it sounds—it receives the message and does something with it. Implementing a receiver (also referred to as a service) is as simple as implementing our standard MessageCallback interface, which is used pervasively across, both client and server code. Let's begin with server side component that receives messages:

```
@Service
  public class HelloWorldService implements MessageCallback {
    public void callback(Message message) {
      System.out.println("Hello, World!");
    }
  }
```

Here we declare an extremely simple service. The `@Service` annotation provides a convenient, meta-data based way of having the bus auto-discover and deploy the service.

2.2.3. Sending Messages with the Server Bus

In the following example we extend our server side component to reply with a message when the callback method is invoked. It will create a message and address it to the subject '`HelloWorldClient`':

```
@Service
public class HelloWorldService implements MessageCallback {

  private RequestDispatcher dispatcher;

  @Inject
```

```
  public HelloWorldService(RequestDispatcher dispatcher) {
    dispatcher = dispatcher;
  }

  public void callback(CommandMessage message) {
    // Send a message to the 'HelloWorldClient'.
    MessageBuilder.createMessage()
      .toSubject("HelloWorldClient") // (1)
      .signalling()                  // (2)
      .with("text", "Hi There")      // (3)
      .noErrorHandling()             // (4)
      .sendNowWith(dispatcher);      // (5)
    });
  }
}
```

The above example shows a service which sends a message in response to receiving a message. Here's what's going on:

1. We specify the subject we wish to send a message to. In this case, " `HelloWorldClient` ". We are sending this message to all clients which are listening in on this subject. For information on how to communicate with a single client, see Section 2.6.
2. We indicate that we wish to only signal the service, meaning that we're not sending a qualifying command to the service. For information on this, read the section on Protocols.
3. We add a message part called "text" which contains the value "Hi there".
4. We indicate that we do not want to provide an `ErrorCallback` to deal with errors for this message.
5. We transmit the message by providing an instance of the `RequestDispatcher`.

2.2.4. Receiving Messages on the Client Bus/ Client Services

Messages can be received asynchronously and arbitrirally by declaring callback services within the client bus. As ErraiBus maintains an open COMET channel at all times, these messages are delivered in real time to the client as they are sent. This provides built-in push messaging for all client services.

```
public class HelloWorld implements EntryPoint {

  private MessageBus bus = ErraiBus.get();

  public void onModuleLoad() {
     [...]
```

```
    /*
     * Declare a service to receive messages on the subject
     * "BroadcastReceiver".
     */
    bus.subscribe("BroadcastReceiver", new MessageCallback() {
      public void callback(CommandMessage message) {
        /*
         * When a message arrives, extract the "text" field and
         * do something with it
         */
         String messageText = message.get(String.class, "text");
       }
    });

    [...]
  }
}
```

In the above example, we declare a new client service called `"BroadcastReceiver"` which can now accept both local messages and remote messages from the server bus. The service will be available in the client to receive messages as long the client bus is and the service is not explicitly de-registered.

2.2.5. Local Services

On the client or the server, you can create a local receiver which only receives messages that originated on the local bus. A local server-side service only receives messages that originate on that server, and a local client-side service only receives messages that originated on that client.

To create a local receiver using the declarative API, use the `@Local` annotation in conjunction with `@Service`:

```
@Local
@Service
  public class HelloIntrovertService implements MessageCallback {
    public void callback(Message message) {
      System.out.println("Hello, me!");
    }
  }
```

To create a local receiver using through programmatic service registration, use the `subscribeLocal()` method in place of `subscribe()` :

```
public void registerLocalService(MessageBus bus) {
  bus.subscribeLocal("LocalBroadcastReceiver", new MessageCallback() {
```

```
    public void callback(Message message) {
       String messageText = message.get(String.class, "text");
     }
  });
}
```

Both examples above work in client- and server-side code.

2.3. Single-Response Conversations & Pseudo-Synchronous Messaging

It is possible to contruct a message and a default response handler as part of the `MessageBuilder` API. It should be noted, that multiple replies will not be possible and will result an exception if attempted. Using this aspect of the API is very useful for doing simple psuedo-synchronous conversive things.

You can do this by specifying a `MessageCallback` using the `repliesTo()` method in the `MessageBuilder` API after specifying the error handling of the message.

```
MessageBuilder.createMessage()
  .toSubject("ConversationalService").signalling()
  .with("SomeField", someValue)
  .noErrorHandling()
  .repliesTo(new MessageCallback() {
    public void callback(Message message) {
      System.out.println("I received a response");
    }
   })
```

See the next section on how to build conversational services that can respond to such messages.

2.4. Sender Inferred Subjects

It is possible for the sender to infer, to whatever conversational service it is calling, what subject it would like the reply to go to. This is accomplished by utilizing the standard `MessageParts.ReplyTo` message part. Using this methodology for building conversations is generally encouraged.

Consider the following client side code:

```
  MessageBuilder.createMessage()
    .toSubject("ObjectService").signalling()
    .with(MessageParts.ReplyTo, "ClientEndpoint")
    .noErrorHandling().sendNowWith(dispatcher);
```

And the conversational code on the server (for service *ObjectService*):

```
MessageBuilder.createConversation(message)
  .subjectProvided().signalling()
  .with("Records", records)
  .noErrorHandling().reply();
```

In the above examples, assuming that the latter example is inside a service called "`ObjectService`" and is referencing the incoming message that was sent in the former example, the message created will automatically reference the `ReplyTo` subject that was provided by the sender, and send the message back to the subject desired by the client on the client that sent the message.

2.5. Broadcasting

Broadcasting messages to all clients listening on a specific subject is quite simple and involves nothing more than forgoing use of the reply API. For instance:

```
MessageBuilder.createMessage().
    .toSubject("MessageListener")
    .with("Text", "Hello, from your overlords in the cloud")
    .noErrorHandling().sendGlobalWith(dispatcher);
```

If sent from the server, all clients currently connected, who are listening to the subject "`MessageListener`" will receive the message. It's as simple as that.

2.6. Client-to-Client Communication

Communication from one client to another client is not directly possible within the bus federation, by design. This isn't to say that it's not possible. But one client cannot see a service within the federation of another client. We institute this limitation as a matter of basic security. But many software engineers will likely find the prospects of such communication appealing, so this section will provide some basic pointers on how to go about accomplishing it.

2.6.1. Relay Services

The essential architectural thing you'll need to do is create a relay service that runs on the server. Since a service advertised on the server is visible to all clients and all clients are visible to the server, you might already see where we're going with this.

By creating a service on the server which accepts messages from clients, you can create a simple protocol on-top of the bus to enable quasi peer-to-peer communication. (We say quasi, because it still needs to be routed through the server)

While you can probably imagine simply creating a broadcast-like service which accepts a message from one client and broadcasts it to the rest of the world, it may be less clear how to go about routing from one particular client to another particular client, so we'll focus on that problem. This is covered in *the next chapter*.

2.7. Message Routing Information

Every message that is sent between a local and remote (or server and client) buses contain session routing information. This information is used by the bus to determine what outbound queues to use to deliver the message to, so they will reach their intended recipients. It is possible to manually specify this information to indicate to the bus, where you want a specific message to go.

You can obtain the `SessionID` directly from a `Message` by getting the `QueueSession` resource:

```
   QueueSession sess = message.getResource(QueueSession.class, Resources.Session.name());
   String sessionId = sess.getSessionId();
```

You can extract the `SessionID` from a message so that you may use it for routing by obtaining the `QueueSession` resource from the `Message`. For example:

```
...
  public void callback(Message message) {
   QueueSession sess = message.getResource(QueueSession.class, Resources.Session.name());
   String sessionId = sess.getSessionId();

    // Record this sessionId somewhere.
    ...
  }
```

The `SessionID` can then be stored in a medium, say a Map, to cross-reference specific users or whatever identifier you wish to allow one client to obtain a reference to the specific `SessionID` of another client. In which case, you can then provide the `SessionID` as a MessagePart to indicate to the bus where you want the message to go.

```
  MessageBuilder.createMessage()
    .toSubject("ClientMessageListener")
    .signalling()
    .with(MessageParts.SessionID, sessionId)
    .with("Message", "We're relaying a message!")
    .noErrorHandling().sendNowWith(dispatcher);
```

By providing the `SessionID` part in the message, the bus will see this and use it for routing the message to the relevant queue.

It may be tempting however, to try and include destination `SessionIDs` at the client level, assuming that this will make the infrastructure simpler. But this will not achieve the desired results, as the bus treats `SessionIDs` as transient. Meaning, the `SessionID` information is not ever transmitted from bus-to-bus, and therefore is only directly relevant to the proximate bus.

2.8. Handling Errors

Asynchronous messaging necessitates the need for asynchronous error handling. Luckily, support for handling errors is built directly into the `MessageBuilder` API, utilizing the `ErrorCallback` interface. In the examples shown in previous exceptions, error handing has been glossed over with aubiquitous usage of the `noErrorHandling()` method while building messaging. We chose to require the explicit use of such a method to remind developers of the fact that they are responsible for their own error handling, requiring you to explicitly make the decision to forego handling potential errors.

As a general rule, you should *always handle your errors* . It will lead to faster and quicker identification of problems with your applications if you have error handlers, and generally help you build more robust code.

```
MessageBuilder.createMessage()
    .toSubject("HelloWorldService")
    .signalling()
    .with("msg", "Hi there!")
    .errorsHandledBy(new ErrorCallback() {
      public boolean error(Message message, Throwable throwable) {
        throwable.printStackTrace();
          return true;
      }
    })
    .sendNowWith(dispatcher);
```

The addition of error handling at first may put off developers as it makes code more verbose and less-readable. This is nothing that some good practice can't fix. In fact, you may find cases where the same error handler can appropriately be shared between multiple different calls.

```
ErrorCallback error = new ErrorCallback() {
  public boolean error(Message message, Throwable throwable) {
    throwable.printStackTrace();
    return true;
  }
}

MessageBuilder.createMessage()
  .toSubject("HelloWorldService")
  .signalling()
```

```
  .with("msg", "Hi there!")
  .errorsHandledBy(error)
  .sendNowWith(dispatcher);
```

The error handler is required to return a `boolean` value. This is to indicate whether or not Errai should perform the default error handling actions it would normally take during a failure. You will almost always want to return `true` here, unless you are trying to explicitly surpress some undesirably activity by Errai, such as automatic subject-termination in conversations. But this is almost never the case.

2.8.1. Handling global message transport errors

You may need to detect problems which occur on the bus at runtime. The client bus API provides a facility for doing this in the `org.jboss.errai.bus.client.framework.ClientMessageBus` using the `addTransportErrorHandler()` method.

A `TransportErrorHandler` is an interface which you can use to define error handling behavior in the event of a transport problem.

For example:

```
messageBus.addTransportErrorHandler(new TransportErrorHandler() {
  public void onError(TransportError error) {
    // error handling code.
  }
});
```

The `TransportError` interface represents the details of an an error from the bus. It contains a set of methods which can be used for determining information on the initial request which triggered the error, if the error occurred over HTTP or WebSockets, status code information, etc. See the JavaDoc for more information.

2.9. Asynchronous Message Tasks

In some applications, it may be necessary or desirable to delay transmission of, or continually stream data to a remote client or group of clients (or from a client to the server). In cases like this, you can utilize the `replyRepeating()` , `replyDelayed()` , `sendRepeating()` and `sendDelayed()` methods in the `MessageBuilder`.

Delayed TasksSending a task with a delay is straight forward. Simply utilize the appropriate method (either `replyDelayed()` or `sendDelayed()`).

```
  MessageBuilder.createConversation(msg)
    .toSubject("FunSubject")
    .signalling()
```

```
    .noErrorHandling()
    .replyDelayed(TimeUnit.SECONDS, 5); // sends the message after 5 seconds.
```

or

```
   MessageBuilder.createMessage()
     .toSubject("FunSubject")
     .signalling()
     .noErrorHandling()
                 .sendDelayed(requestDispatcher,   TimeUnit.SECONDS,   5);   /
/ sends the message after 5 seconds.
```

2.10. Repeating Tasks

A repeating task is sent using one of the MessageBuilder's `repeatXXX()` methods. The task will repeat indefinitely until cancelled (see next section).

```
  MessageBuilder.createMessage()
    .toSubject("FunSubject")
    .signalling()
    .withProvided("time", new ResourceProvider<String>() {
       SimpleDateFormat fmt = new SimpleDateFormat("hh:mm:ss");

       public String get() {
         return fmt.format(new Date(System.currentTimeMillis());
       }
     }
     .noErrorHandling()
           .sendRepeatingWith(requestDispatcher,   TimeUnit.SECONDS,   1);   //
sends a message every 1 second
```

The above example sends a message very 1 second with a message part called `"time"`, containing a formatted time string. Note the use of the `withProvided()` method; a provided message part is calculated at the time of transmission as opposed to when the message is constructed.

Cancelling an Asynchronous TaskA delayed or repeating task can be cancelled by calling the `cancel()` method of the `AsyncTask` instance which is returned when creating a task. Reference to the AsyncTask object can be retained and cancelled by any other thread.

```
AsyncTask task = MessageBuilder.createConversation(message)
  .toSubject("TimeChannel").signalling()
  .withProvided(TimeServerParts.TimeString, new ResourceProvider<String>() {
```

```
    public String get() {
      return String.valueOf(System.currentTimeMillis());
    }
  }).defaultErrorHandling().replyRepeating(TimeUnit.MILLISECONDS, 100);

   ...

// cancel the task and interrupt it's thread if necessary.
task.cancel(true);
```

2.11. Queue Sessions

The ErraiBus maintains it's own seperate session management on-top of the regular HTTP session management. While the queue sessions are tied to, and dependant on HTTP sessions for the most part (meaning they die when HTTP sessions die), they provide extra layers of session tracking to make dealing with complex applications built on Errai easier.

2.11.1. Lifecycle

The lifescyle of a session is bound by the underlying HTTP session. It is also bound by activity thresholds. Clients are required to send heartbeat messages every once in a while to maintain their sessions with the server. If a heartbeat message is not received after a certain period of time, the session is terminated and any resources are deallocated.

2.11.2. Scopes

One of the things Errai offers is the concept of session and local scopes.

2.11.2.1. Session Scope

A session scope is scoped across all instances of the same session. When a session scope is used, any parameters stored will be accessible and visible by all browser instances and tabs.

The SessionContext helper class is used for accessing the session scope.

```
public class TestService implements MessageCallback {
  public void callback(final Message message) {
    // obtain a reference to the session context by referencing the incoming
 message.
    SessionContext injectionContext = SessionContext.get(message);

    // set an attribute.
    injectionContext.setAttribute("MyAttribute", "Foo");
  }
}
```

2.11.2.2. Local Scope

A local scope is scoped to a single browser instance. But not to a single session.

In a browser a local scope would be confined to a tab or a window within a browser. You can store parameters inside a local scope just like with a session by using the `LocalContext` helper class.

```
public class TestService implements MessageCallback {
  public void callback(final Message message) {
   // obtain a reference to the local context by referencing the incoming message.
    LocalContext injectionContext = LocalContext.get(message);

   // set an attribute.
    injectionContext.setAttribute("MyAttribute", "Foo");
  }
}
```

2.12. Wire Protocol (J.REP)

ErraiBus implements a JSON-based wire protocol which is used for the federated communication between different buses. The protocol specification encompasses a standard JSON payload structure, a set of verbs, and an object marshalling protocol. The protocol is named J.REP. Which stands for JSON Rich Event Protocol.

2.12.1. Payload Structure

All wire messages sent across are assumed to be JSON arrays at the outermost element, contained in which, there are *0..n* messages. An empty array is considered a no-operation, but should be counted as activity against any idle timeout limit between federated buses.

Example 2.1. Example J.REP Payload

```
[
  {"ToSubject" : "SomeEndpoint", "Value" : "SomeValue" },
  {"ToSubject" : "SomeOtherEndpoint", "Value" : "SomeOtherValue"}
]
```

Here we see an example of a J.REP payload containing two messages. One bound for an endpoint named `"SomeEndpoint"` and the other bound for the endpoint `"SomeOtherEndpoint"` . They both include a payload element `"Value"` which contain strings. Let's take a look at the anatomy of an individual message.

Example 2.2. An J.REP Message

```
{
  "ToSubject" : "TopicSubscriber",
  "CommandType" : "Subscribe",
  "Value " : "happyTopic",
  "ReplyTo" : "MyTopicSubscriberReplyTo"
}
```

The message shows a very vanilla J.REP message. The keys of the JSON Object represent individual *message parts* , with the values representing their corresponding values. The standard J.REP protocol encompasses a set of standard message parts and values, which for the purposes of this specification we'll collectively refer to as the protocol verbs.

The following table describes all of the message parts that a J.REP capable client is expected to understand:

Part	Required	JSON Type	Description
`ToSubject`	Yes	String	Specifies the subject within the bus, and its federation, which the message should be routed to.
`CommandType`	No	String	Specifies a command verb to be transmitted to the receiving subject. This is an optional part of a message contract, but is required for using management services
`ReplyTo`	No	String	Specifies to the receiver what subject it should reply to in response to this message.
`Value`	No	Any	A recommended but not required standard payload part for sending data to services

Part	Required	JSON Type	Description
`PriorityProcessing`	No	Number	A processing order salience attribute. Messages which specify priority processing will be processed first if they are competing for resources with other messages in flight. Note: the current version of ErraiBus only supports two salience levels (0 and >1). Any non-zero salience in ErraiBus will be given the same priority relative to 0 salience messages
`ErrorMessage`	No	String	An accompanying error message with any serialized exception
`Throwable`	No	Object	If applicable, an encoded object representing any remote exception that was thrown while dispatching the specified service

2.12.1.1. Built-in Subjects

The table contains a list of reserved subject names used for facilitating things like bus management and error handling. A bus should never allow clients to subscribe to these subjects directly.

Subject	Description
`ClientBus`	The self-hosted message bus endpoint on the client
`ServerBus`	The self-hosted message bus endpoint on the server

Subject	Description
`ClientBusErrors`	The standard error receiving service for clients

As this table indicates, the bus management protocols in J.REP are accomplished using self-hosted services. See the section on *Bus Management and Handshaking Protocols* for details.

2.12.2. Message Routing

There is no real distinction in the J.REP protocol between communication with the server, versus communication with the client. In fact, it assumed from an architectural standpoint that there is no real distinction between a client and a server. Each bus participates in a flat-namespaced federation. Therefore, it is possible that a subject may be observed on both the server and the client.

One in-built assumption of a J.REP-compliant bus however, is that messages are routed within the auspices of session isolation. Consider the following diagram:

Figure 2.1. Topology of a J.REP Messaging Federation

It is possible for *Client A* to send messages to the subjects *ServiceA* and *ServiceB* . But it is not possible to address messages to *ServiceC* . Conversely, *Client B* can address messages to *ServiceC* and *ServiceB* , but not *ServiceA* .

2.12.3. Bus Management and Handshaking Protocols

Federation between buses requires management traffic to negotiate connections and manage visibility of services between buses. This is accomplished through services named `ClientBus` and `ServerBus` which both implement the same protocol contracts which are defined in this section.

2.12.3.1. ServerBus and ClientBus commands

Both bus services share the same management protocols, by implementing verbs (or commands) that perform different actions. These are specified in the protocol with the `CommandType` message part. The following table describes these commands:

Table 2.1. Message Parts for Bus Commands:

Command / Verb	Message Parts	Description
`ConnectToQueue`	N/A	The first message sent by a connecting client to begin the handshaking process.
`CapabilitiesNotice`	`CapabilitiesFlags`	A message sent by one bus to another to notify it of its

Command / Verb	Message Parts	Description
		capabilities during handshake (for instance long polling or websockets)
`FinishStateSync`	N/A	A message sent from one bus to another to indicate that it has now provided all necessary information to the counter-party bus to establish the federation. When both buses have sent this message to each other, the federation is considered active.
`RemoteSubscribe`	`Subject` *or* `SubjectsList`	A message sent to the remote bus to notify it of a service or set of services which it is capable of routing to.
`RemoteUnsubscribe`	`Subject`	A message sent to the remote bus to notify it that a service is no longer available.
`Disconnect`	`Reason`	A message sent to a server bus from a client bus to indicate that it wishes to disconnect and defederate. Or, when sent from the client to server, indicates that the session has been terminated.
`SessionExpired`	N/A	A message sent to a client bus to indicate that its messages are no longer being routed because it no longer has an active session
`Heartbeat`	N/A	A message sent from one bus to another periodically to indicate it is still active.

Part	Required	JSON Type	Description
`CapabilitiesFlags`	Yes	String	A comma delimited string of capabilities the bus is capable of us

Part	Required	JSON Type	Description
`Subject`	Yes	String	The subject to subscribe or unsubscribe from
`SubjectsList`	Yes	Array	An array of strings representing a list of subjects to subscribe to

2.13. Conversations

Conversations are message exchanges which are between a single client and a service. They are a fundmentally important concept in ErraiBus, since by default, a message will be broadcast to all client services listening on a particular channel.

When you create a reply with an incoming message, you ensure that the message you are sending back is received by the same client which sent the incoming message. A simple example:

```
@Service
public class HelloWorldService implements MessageCallback {
  public void callback(CommandMessage message) {
    // Send a message to the 'HelloWorldClient' on the client that sent us the
    // the message.
    MessageBuilder.createConversation(message)
      .toSubject("HelloWorldClient")
      .signalling()
      .with("text", "Hi There! We're having a reply!")
      .noErrorHandling().reply();
    });
  }
}
```

Note that the only difference between the example in the previous section and this is the use of the `createConversation()` method with `MessageBuilder`.

2.14. WebSockets

ErraiBus has support for WebSocket-based communication. When WebSockets are enabled, capable web browsers will attempt to upgrade their COMET-based communication with the server-side bus to use a WebSocket channel.

There are three different ways the bus can enable WebSockets. The first uses a sideband server, which is a small, lightweight server which runs on a different port from the application server. The second is native JBoss AS 7-based integration and the third is to rely in JSR-356 support in WildFly. Of course, you only need to configure one of these three options!

2.14.1. Configuring the sideband server

Activating the sideband server is as simple as adding the following to the `ErraiService.properties` file:

```
errai.bus.enable_web_socket_server=true
```

The default port for the sideband server is `8085`. You can change this by specifying a port with the `errai.bus.web_socket_port` property in the `ErraiService.properties` file.

Netty Dependencies

Make sure to deploy the required Netty dependencies to your server. If you started with one of our demos or our tutorial project it should be enough to **NOT** set `netty-codec-http` to provided. All required transitive dependencies should then be part of your war file (WEB-INF/lib).

2.14.2. Deploying with JBoss AS 7.1.2 (or higher)

This is an alternative approach to the sideband server described in the previous chapter. Make sure to **NOT** configure both! It is currently necessary to use the native connector in JBoss AS for WebSockets to work. So the first step is to configure your JBoss AS instance(s) to use the native connector by changing the `domain/configuration/standalone.xml` or `domain/configuration/domain.xml` file as follows:

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-server="default-host" native="false">
```

to:

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-server="default-host" native="true">
```

Verify that the native APR connector is being used

To verify that the native connectors are being used check your console for the following log message: `INFO [org.apache.coyote.http11.Http11AprProtocol] (MSC service thread 1-6) Starting Coyote HTTP/1.1 on http-/127.0.0.1:8080`

The important part is `org.apache.coyote.http11.Http11AprProtocol`. You should **NOT** be seeing `org.apache.coyote.http11.Http11Protocol`. You might have to install the Tomcat native library if not already available on your system.

You will then need to configure the servlet in your application's `web.xml` which will provide WebSocket upgrade support within AS7.

Add the following to the `web.xml` :

```
<context-param>
  <param-name>websockets-enabled</param-name>
  <param-value>true</param-value>
</context-param>

<context-param>
  <param-name>websocket-path-element</param-name>
  <param-value>in.erraiBusWS</param-value>
</context-param>
```

This will tell the bus to enable web sockets support. The `websocket-path-element` specified the path element within a URL which the client bus should request in order to negotiate a websocket connection. For instance, specifying `in.erraiBusWS` as we have in the snippit above, will result in attempted negotiation at *`http://<your_server>:<your_port>/<context_path>/in.erraiBusWS`*. For this to have any meaningful result, we must add a servlet mapping that will match this pattern:

```
<servlet>
  <servlet-name>ErraiWSServlet</servlet-name>
   <servlet-class>org.jboss.errai.bus.server.servlet.JBossAS7WebSocketServlet</
servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>ErraiWSServlet</servlet-name>
  <url-pattern>*.erraiBusWS</url-pattern>
</servlet-mapping>
```

Do not remove the regular ErraiBus servlet mappings!

When configuring ErraiBus to use WebSockets on JBoss AS, you *do not* remove the existing servlet mappings for the bus. The WebSocket servlet is in *addition to*

your current bus servlet. This is because ErraiBus *always* negotiates WebSocket sessions over the COMET channel.

Important dependency

Also make sure to deploy the required errai-bus-jboss7-websocket.jar to your server. If you're using Maven simply add the following dependency to your pom.xml file:

```
    <dependency>
        <groupId>org.jboss.errai</groupId>
        <artifactId>errai-bus-jboss7-websocket</artifactId>
        <version>${errai.version}</version>
    </dependency>
```

2.14.3. JSR-356 WebSocket support (Deploying to WildFly 8.0 or higher)

Errai provides two implementations for this:

1. errai-bus-jsr356-websocket: A simple JSR-356 implementation, that does not rely on CDI or Weld.
2. errai-bus-jsr356-websocket-weld: Provides the possibility to use the builtin CDI scopes (javax.enterprise.context.RequestScoped, javax.enterprise.context.SessionScoped, javax.enterprise.context.ConversationScoped). This implementation uses WELD and is intended for JBoss WildFly 8 or higher.

Make sure to add the following project dependency:

```
<!-- For JSR-356 without depending on Weld -->
<dependency>
    <groupId>org.jboss.errai</groupId>
    <artifactId>errai-bus-jsr356-websocket</artifactId>
    <version>${errai.version}</version>
</dependency>

<!-- For JEE environment with Weld-->
<dependency>
    <groupId>org.jboss.errai</groupId>
    <artifactId>errai-bus-jsr356-websocket-weld</artifactId>
    <version>${errai.version}</version>
</dependency>
```

Dependency on Weld!

The JSR-356 specification is not addressing the integration with builtin CDI scopes which is what the `<artifactId>errai-bus-jsr356-websocket-weld</artifactId>` module provides. It therefore depends directly to JBoss Weld. If you use a non Weld-based middleware, you can use `<artifactId>errai-bus-jsr356-websocket</artifactId>` instead.

To configure ErraiBus that WebSocket communication should be used, define the following in your `web.xml`

```
<context-param>
    <param-name>websockets-enabled</param-name>
    <param-value>true</param-value>
</context-param>
```

Do not remove the regular ErraiBus servlet mappings!

When configuring ErraiBus to use JSR-356 WebSocket, you *do not* remove the existing servlet mappings for the bus. This is because ErraiBus *always* negotiates WebSocket sessions over the COMET channel.

You can also define filters when using the JSR-356 WebSocket implementation. These filters will be executed for each received ErraiBus message on the server. Your filters need to implement `org.jboss.errai.bus.server.websocket.jsr356.filter.WebSocketFilter` and must be configured in your appliations's `web.xml` as an ordered comma separated list:

```
<context-param>
    <param-name>errai-jsr-356-websocket-filter</param-name>
    <param-value>foo.bar.FooFilter,foo.bar.BarFilter</param-value>
</context-param>
```

Because one of the filter method parameters is the actual WebSocket session, you also have to add the following dependency to your `pom.xml`:

```
<dependency>
    <groupId>org.jboss.spec.javax.websocket</groupId>
    <artifactId>jboss-websocket-api_1.0_spec</artifactId>
    <scope>provided</scope>
    <version>1.0.0.Final</version>
```

```
</dependency>
```

Please take a look at the JavaDoc of the filter interface for more information about the method parameters.

2.14.4. WebSocket Security

Errai supports Websocket security (wss) for two deployment scenarios.

1. The servlet container is deployed behind a reverse-proxy or other SSL terminating appliance
2. The servlet container is deployed in front and takes care of SSL termination

2.14.4.1. Servlet container is deployed behind a reverse-proxy

All you need to do is configure the following context parameter in your `web.xml`

```
<context-param>
  <param-name>force-secure-websockets</param-name>
  <param-value>true</param-value>
</context-param>
```

This will work for the sideband-server as well as JBoss AS7 and WildFly based websocket support.

Dont forget to configure your reverse-proxy or appliance to handle websocket connections with SSL!

To use this kind of configuration you have to configure your reverse-proxy or appliance to use HTTPS (SSL / TLS).

2.14.4.2. Servlet container takes care of SSL termination

2.14.4.2.1. With JBoss AS 7.1.2 (or higher) & JSR-356 WebSocket (WildFly 8 or higher)

If the servlet container is configured to use HTTPS, the Errai WebSocket Servlet for JBoss AS 7 and / or the Errai JSR-356 WebSocket implementation for WildFly will use the WSS scheme automatically. So, there's is nothing else to do!

2.14.4.2.2. Using the sideband server

To tell the sideband server to use SSL and configure Errai to use the WSS scheme the following properties are mandatory in `ErraiService.properties`:

1. errai.bus.secure_web_socket_server=true → This tells the sideband server and Errai to use SSL / WSS.

2. errai.bus.web_socket_keystore=[full qualified path to the JKS or PKCS12 keystore] → Key store to use

3. errai.bus.web_socket_keystore_password=[password for the key store] → Password for the key store to use

Depending on your configuration and type of PKI container you can the add following addtional properties in the `ErraiService.properties`:

1. errai.bus.web_socket_keystore_type=[kind of key store] → When you want to use the PKCS12 format instead of JKS (default)

2. errai.bus.web_socket_key_password=[password of private key] → When the private key in the container has a different password than the keystore itself.

Sidband server needs a keystore.

In this deployment case, the sideband server needs a keystore with a server certificate and the corresponding private key.

2.15. Bus Lifecycle

2.15.1. Turning Server Communication On and Off

By default, Errai's client-side message bus attempts to connect to the server as soon as the ErraiBus module has been loaded. The bus will stay connected until a lengthy (about 45 seconds) communication failure occurs, or the web page is unloaded.

The application can affect bus communication through two mechanisms:

1. By setting a global JavaScript variable `erraiBusRemoteCommunicationEnabled = false` before the GWT scripts load, bus communication with the server is permanently disabled

2. By calling `((ClientMessageBus) ErraiBus.get()).stop()`, the bus disconnects from the server

To resume server communication after a call to `ClientMessageBus.stop()` or after communication with the server has exceeded the bus' retry timeout, call `((ClientMessageBus) ErraiBus.get()).init()`. You can use a `BusLifecycleListener` to monitor the success or failure of this attempt. See the next section for details.

2.15.2. Observing Bus Lifecycle State and Communication Status

In a perfect world, the client message bus would always be able to communicate with the server message bus. But in the real world, there's a whole array of reasons why the communication link between the server and the client might be interrupted.

On its own, the client message bus will attempt to reconnect with the server whenever communication has been disrupted. Errai applications can monitor the status of the bus' communication link (whether it is disconnected, attempting to connect, or fully connected) through the `BusLifecycleListener` interface:

```
class BusStatusLogger implements BusLifecycleListener {

  @Override
  public void busAssociating(BusLifecycleEvent e) {
    GWT.log("Errai Bus trying to connect...");
  }

  @Override
  public void busOnline(BusLifecycleEvent e) {
    GWT.log("Errai Bus connected!");
  }

  @Override
  public void busOffline(BusLifecycleEvent e) {
    GWT.log("Errai Bus trying to connect...");
  }

  @Override
  public void busDisassociating(BusLifecycleEvent e) {
    GWT.log("Errai Bus going into local-only mode.");
  }
}
```

To attach such a listener to the bus, make the following call in client-side code:

```
ClientMessageBus bus = (ClientMessageBus) ErraiBus.get();
bus.addLifecycleListener(new BusStatusLogger());
```

2.16. Shadow Services

Shadow Services is a Service that will get invoked when there is no longer a connection with the server. This is particular helpful when developing an application for mobile. To create a Shadow

Service for a specific Services all you have to do is annotate a new client side implementation with the @ShadowService:

```
@ShadowService
public class SignupShadowService implements MessageCallback {
   @Override
   public void callback(Message message) {
   }
}
```

Also when you have a RPC based Service you can just add @ShadowService on a client side implementation to configure it to be the service to get called when there is no network:

```
@ShadowService
public class SignupServiceShadow implements SignupService {
   @Override
   public User register(User newUserObject, String password) throws RegistrationException {
   }
}
```

In this shadow service we can create logic that will deal with the temporary connection loss. For instance you could save the data that needs to get send to the server with JPA on the client and then when the bus get online again sent the data to the server.

2.17. Debugging Messaging Problems

Errai includes a bus monitoring application, which allows you to monitor all of the message exchange activity on the bus in order to help track down any potential problems It allows you to inspect individual messages to examine their state and structure.

To utilize the bus monitor, you'll need to include the _errai-tools _ package as part of your application's dependencies. When you run your application in development mode, you will simply need to add the following JVM options to your run configuration in order to launch the monitor: `-Derrai.tools.bus_monitor_attach=true`

Figure 2.2. ErraiBus Monitor

The monitor provides you a real-time perspective on what's going on inside the bus. The left side of the main screen lists the services that are currently available, and the right side is the service-explorer, which will show details about the service.

To see what's going on with a specific service, simply double-click on the service or highlight the service, then click "Monitor Service…". This will bring up the service activity monitor.

Figure 2.3. ErraiBus Monitor details

The service activity monitor will display a list of all the messages that were transmitted on the bus since the monitor became active. You do not need to actually have each specific monitor window open in order to actively monitor the bus activity. All activity on the bus is recorded.

The monitor allows you select individual messages, an view their individual parts. Clicking on a message part will bring up the object inspector, which will allow you to explore the state of any objects contained within the message, not unlike the object inspectors provided by debuggers in your favorite IDE. This can be a powerful tool for looking under the covers of your application.

Dependency Injection

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai IOC* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai IOC to your project.

The core Errai IOC module implements the *JSR-330 Dependency Injection* specification for in-client component wiring.

Dependency injection (DI) allows for cleaner and more modular code, by permitting the implementation of decoupled and type-safe components. By using DI, components do not need to be aware of the implementation of provided services. Instead, they merely declare a contract with the container, which in turn provides instances of the services that component depends on.

Classpath Scanning and ErraiApp.properties

Errai only scans the contents of classpath locations (JARs and directories) that have *a file called ErraiApp.properties* at their root. If dependency injection is not working for you, double-check that you have an `ErraiApp.properties` in every JAR and directory that contains classes Errai should know about.

A simple example:

```
public class MyLittleClass {
  private final TimeService timeService;

  @Inject
  public MyLittleClass(TimeService timeService) {
    this.timeService = timeService;
  }

  public void printTime() {
    System.out.println(this.timeService.getTime());
  }
}
```

In this example, we create a simple class which declares a dependency using `@Inject` for the interface `TimeService`. In this particular case, we use constructor injection to establish the contract between the container and the component. We can similarly use field injection to the same effect:

```
public class MyLittleClass {
  @Inject
  private TimeService timeService;

  public void printTime() {
    System.out.println(this.timeService.getTime());
  }
}
```

In order to inject `TimeService` , you must annotate it with `@ApplicationScoped` or the Errai DI container will not acknowledge the type as a bean.

```
@ApplicationScoped
public class TimeService {
}
```

Best Practices

Although field injection results in less code, a major disadvantage is that you cannot create immutable classes using the pattern, since the container must first call the default, no-argument constructor, and then iterate through its injection tasks, which leaves the potential albeit remote that the object could be left in an partially or improperly initialized state. The advantage of constructor injection is that fields can be immutable (final), and invariance rules applied at construction time, leading to earlier failures, and the guarantee of consistent state.

3.1. Container Wiring

In contrast to *Gin* [http://code.google.com/p/google-gin/] , the Errai IOC container does not provide a programmatic way of creating and configuring injectors. Instead, container-level binding rules are defined by implementing a `Provider` , which is scanned for and auto-discovered by the container.

A `Provider` is essentially a factory which produces type instances within in the container, and defers instantiation responsibility for the provided type to the provider implementation. Top-level providers use the standard `javax.inject.Provider<T>` interface.

Types made available as *top-level* providers will be available for injection in any managed component within the container.

Out of the box, Errai IOC implements these default top-level providers, all defined in the `org.jboss.errai.ioc.client.api.builtin` package:

- `CallerProvider` : Makes RPC `Caller<T>` objects available for injection.
- `DisposerProvider` : Makes Errai IoC `Disposer<T>` objects available for injection.
- `InitBallotProvider` : Makes instances of `InitBallot` available for injection.
- `IOCBeanManagerProvider` : Makes Errai's client-side bean manager, `ClientBeanManager` , available for injection.
- `MessageBusProvider` : Makes Errai's client-side `MessageBus` singleton available for injection.
- `RequestDispatcherProvider` : Makes an instance of the `RequestDispatcher` available for injection.
- `RootPanelProvider` : Makes GWT's `RootPanel` singleton injectable.
- `SenderProvider` : Makes MessageBus `Sender<T>` objects available for injection.

Implementing a `Provider` is relatively straight-forward. Consider the following two classes:

TimeService.java

```
public interface TimeService {
  public String getTime();
}
```

TimeServiceProvider.java

```
@IOCProvider
@Singleton
public class TimeServiceProvider implements Provider<TimeService> {
  @Override
  public TimeService get() {
    return new TimeService() {
      public String getTime() {
        return "It's midnight somewhere!";
      }
    };
  }
}
```

If you are familiar with Guice, this is semantically identical to configuring an injector like so:

```
Guice.createInjector(new AbstractModule() {
  public void configure() {
    bind(TimeService.class).toProvider(TimeServiceProvider.class);
  }
 }).getInstance(MyApp.class);
```

As shown in the above example code, the annotation `@IOCProvider` is used to denote top-level providers.

The classpath will be searched for all annotated providers at compile time.

Important

Top-level providers are regular beans, so they can inject dependencies particularly from other top-level providers as necessary.

3.2. Wiring server side components

By default, Errai uses Google Guice to wire server-side components. When deploying services on the server-side, it is currently possible to obtain references to the `MessageBus`, `RequestDispatcher`, the `ErraiServiceConfigurator`, and `ErraiService` by declaring them as injection dependencies in Service classes, extension components, and session providers.

Alternatively, supports CDI based wiring of server-side components. See the chapter on Errai CDI for more information.

3.3. Scopes

Out of the box, the IOC container supports three bean scopes, `@Dependent`, `@Singleton` and `@EntryPoint`. The singleton and entry-point scopes are roughly the same semantics.

3.3.1. Dependent Scope

In Errai IOC, all client types are valid bean types if they are default constructable or can have construction dependencies satisfied. These unqualified beans belong to the dependent pseudo-scope. See: *Dependent Psuedo-Scope from CDI Documentation* [http://docs.jboss.org/weld/reference/latest/en-US/html/scopescontexts.html#d0e1997]

Additionally, beans may be qualified as `@ApplicationScoped`, `@Singleton` or `@EntryPoint`. Although `@ApplicationScoped` and `@Singleton` are supported for completeness and conformance, within the client they effectively result in behavior that is identical.

Example 3.1. Example dependent scoped bean

```
public void MyDependentScopedBean {
  private final Date createdDate;

  public MyDependentScopedBean {
    createdDate = new Date();
  }
}
```

Example 3.2. Example ApplicationScoped bean

```
@ApplicationScoped
public void MyClientBean {
  @Inject MyDependentScopedBean bean;

  // ... //
}
```

Availability of dependent beans in the client-side BeanManager

As is mentioned in the *bean manager documentation*, only beans that are *explicitly* scoped will be made available to the bean manager for lookup. So while it is not necessary for regular injection, you must annotate your dependent scoped beans with `@Dependent` if you wish to dynamically lookup these beans at runtime.

3.4. Built-in Extensions

3.4.1. Bus Services

As Errai IOC provides a container-based approach to client development, support for Errai services are exposed to the container so they may be injected and used throughout your application where appropriate. This section covers those services.

3.4.1.1. @Service

The `org.jboss.errai.bus.server.annotations.Service` annotation is used for binding service endpoints to the bus. Within the Errai IOC container you can annotate services and have them published to the bus on the client (or on the server) in a very straight-forward manner:

Example 3.3. A simple message receiving service

```
@Service
public class MyService implements MessageCallback {
  public void callback(Message message) {
    // ... //
  }
}
```

As with server-side use of the annotation, if a service name is not explicitly specified, the underlying class name or field name being annotated will be used as the service name.

3.4.1.2. @Local

The `org.jboss.errai.bus.server.api.Local` annotation is used in conjunction with the `@Service` annotation to advertise a service only for visibility on the local bus and thus, cannot receive messages across the wire for the service.

Example 3.4. A local only service

```
@Service @Local
public class MyLocalService implements MessageCallback {
  public void callback(Message message) {
    // ... //
  }
}
```

3.4.1.3. Lifecycle Impact of Services

Services which are registered with ErraiBus via the bean manager through use of the `@Service` annotation, have de-registration hooks tied implicitly to the destruction of the bean. Thus, *destruction of the bean* implies that these associated services are to be dereferenced.

3.4.2. Client Components

The IOC container, by default, provides a set of default injectable bean types. They range from basic services, to injectable proxies for RPC. This section covers the facilities available out-of-the-box.

3.4.2.1. MessageBus

The type `org.jboss.errai.bus.client.framework.MessageBus` is globally injectable into any bean. Injecting this type will provide the instance of the active message bus running in the client.

Example 3.5. Injecting a MessageBus

```
@Inject MessageBus bus;
```

3.4.2.2. RequestDispatcher

The type `org.jboss.errai.bus.client.framework.RequestDispatcher` is globally injectable into any bean. Injecting this type will provide a `RequestDispatcher` instance capable of delivering any messages provided to it, to the `MessageBus`.

Example 3.6. Injecting a RequestDispatcher

```
@Inject RequestDispatcher dispatcher;
```

3.4.2.3. Caller<?>

The type `org.jboss.errai.common.client.api.Caller<?>` is a globally injectable RPC proxy. RPC proxies may be provided by various components. For example, JAX-RS or Errai RPC. The proxy itself is agnostic to the underlying RPC mechanism and is qualified by it's type parameterization.

For example:

Example 3.7. An example Caller<?> proxy

```
public void MyClientBean {
  @Inject
  private Caller<MyRpcInterface> rpcCaller;

  // ... ///

  @EventHandler("button")
  public void onButtonClick(ClickHandler handler) {
    rpcCaller.call(new RemoteCallback<Void>() {
      public void callback(Void void) {
        // put code here that should execute after RPC response arrives
      }
    ).callSomeMethod();
  }
}
```

The above code shows the injection of a proxy for the RPC remote interface, `MyRpcInterface`. For more information on defining RPC proxies see *Remote Procedure Calls (RPC)*.

3.4.2.4. Sender<?>

The `org.jboss.errai.ioc.support.bus.client.Sender<?>` interface is the lower-level counterpart to the `Caller<?>` interface described above. You can inject a `Sender` to send low-level ErraiBus messages directly to subscribers on any subject.

For example:

```
@Inject
@ToSubject("ListCapitializationService")
Sender<List<String>> listSender;

// ... ///

@EventHandler("button")
public void onButtonClick(ClickHandler handler) {
  List<String> myListOfStrings = getSelectedCitiesFromForm();
  listSender.send(myListOfStrings, new MessageCallback() {
    public void callback(Message reply) {
      // do stuff with reply
    }
  );
}
```

The `Sender.send()` method is overloaded. The variant demonstrated above takes a value and a MessageCallback to reply receive a reply (assuming the subscriber sends a conversational reply). The following variants are available:

- `send(T)`
- `send(T, ErrorCallback)`
- `send(T, MessageCallback)`
- `send(T, MessageCallback, ErrorCallback)`

The reply-to service can also be specified declaratively using the `@ReplyTo` annotation. This allows the app to receive conversational replies even when using the `send()` variants that do not take a `MessageCallback`:

```
@Inject
@ToSubject("ListCapitializationService")
@ReplyTo("ClientListService")
Sender<List<String>> listSender;

// ... ///
```

```
  @EventHandler("button")
  public void onButtonClick(ClickHandler handler) {
    List<String> myListOfStrings = getSelectedCitiesFromForm();
    listSender.send(myListOfStrings);
  }

  @Singleton
  @Service
  public static class ClientListService implements MessageCallback {
    @Override
    public void callback(Message message) {
      // do stuff with message
    }
  }
```

These `Sender<?>` features are just convenient wrappers around the full-featured programmatic ErraiBus API. See *Messaging API Basics* and *Conversations* for full information about low-level ErraiBus communication.

3.4.3. Lifecycle Tools

A problem commonly associated with building large applications in the browser is ensuring that things happen in the proper order when code starts executing. Errai IOC provides you tools which permit you to ensure things happen before initialization, and forcing things to happen after initialization of all of the Errai services.

3.4.3.1. Controlling Startup

In order to prevent initialization of the bus and it's services so that you can do necessary configuration, especially if you are writing extensions to the Errai framework itself, you can create an implicit startup dependency on your bean by injecting an `org.jboss.errai.ioc.client.api.InitBallot<?>`.

Example 3.8. Using an InitBallot to Control Startup

```
@Singleton
public class MyClientBean {
  @Inject InitBallot<MyClientBean> ballot;

  @PostConstruct
  public void doStuff() {
    // ... do some work ...

    ballot.voteForInit();
  }
}
```

3.4.3.2. Performing Tasks After Initialization

Sending RPC calls to the server from inside constructors and `@PostConstruct` methods in Errai is not always reliable due to the fact that the bus and RPC proxies initialize asynchronously with the rest of the application. Therefore it is often desirable to have such things happen in a post-initialization task, which is exposed in the `ClientMessageBus` API. However, it is much cleaner to use the `@AfterInitialization` annotation on one of your bean methods.

Example 3.9. Using @AfterInitialization to do something after startup

```
@Singleton
public class MyClientBean {
  @AfterInitialization
  public void doStuffAfterInit() {
    // ... do some work ...
  }
}
```

3.4.3.3. Increase the Initialization Timeout

For some very large applications it is possible for initialization to timeout in Development Mode because of deferred code generation. If you experience this problem, you can adjust the initialization timeout value by setting the `erraiInitTimeout` variable in your GWT Host Page to a value in milliseconds.

3.4.4. Timed Methods

The `@Timed` annotation allows scheduling method executions on managed client-side beans. Timers are automatically scoped to the scope of the corresponding managed bean and participate in the same lifecycle (see *Bean Lifecycle* for details).

In the following example the `updateTime` method is invoked repeatedly every second.

```
@Timed(type = TimerType.REPEATING, interval = 1, timeUnit = TimeUnit.SECONDS)
private void updateTime() {
  timeWidget.setTime(System.currentTimeMillis);
}
```

For delayed one-time execution of methods `type = TimerType.DELAYED` can be used instead.

3.5. Client-Side Bean Manager

It may be necessary at times to manually obtain instances of beans managed by Errai IOC from outside the container managed scope or creating a hard dependency from your

bean. Errai IOC provides a simple client-side bean manager for handling these scenarios: `org.jboss.errai.ioc.client.container.ClientBeanManager`.

As you might expect, you can inject a bean manager instance into any of your managed beans. If you use Errai IOC in its default mode you will need to inject the synchronous bean manager (`org.jboss.errai.ioc.client.container.SyncBeanManager`).

If you have asynchronous IOC mode enabled simply inject the asynchronous bean manager (`org.jboss.errai.ioc.client.container.async.AsyncBeanManager`) instead. Asynchronous IOC brings support for *code splitting* [http://www.gwtproject.org/doc/latest/DevGuideCodeSplitting.html] . That means that any bean annotated with `@LoadAsync` can be compiled into a separate JavaScript file that's downloaded when the bean is first needed on the client. `@LoadAsync` also allows to specify a fragment name using a class literal. Using GWT 2.6.0 or higher, all types with the same fragment name will be part of the same JavaScript file.

Example 3.10. Injecting the client-side bean manager

```
public MyManagedBean {
  @Inject SyncBeanManager manager;

  // class body
}
```

If you need to access the bean manager outside a managed bean, such as in a unit test, you can access it by calling `org.jboss.errai.ioc.client.container.IOC.getBeanManager()`

3.5.1. Looking up beans

Looking up beans can be done through the use of the `lookupBeans()` method. Here's a basic example:

Example 3.11. Example lookup of a bean

```
public MyManagedBean {
  @Inject SyncBeanManager manager;

  public void lookupBean() {
    IOCBeanDef<SimpleBean> bean = manager.lookupBean(SimpleBean.class);

    if (bean != null) {
      // get the instance of the bean
      SimpleBean inst = bean.getInstance();
    }
  }
}
```

In this example we lookup a bean class named `SimpleBean` . This example will succeed assuming that `SimpleBean` is unambiguous. If the bean is ambiguous and requires qualification, you can do a qualified lookup like so:

Example 3.12. Looking up beans with qualifiers

```
MyQualifier qual = new MyQualifier() {
  public annotationType() {
    return MyQualifier.class;
  }
}

MyOtherQualifier qual2 = new MyOtherQualifier() {
  public annotationType() {
    return MyOtherQualifier.class;
  }
}

// pass qualifiers to ClientBeanManager.lookupBeans
IOCBeanDef<SimpleBean> bean = beanManager.lookupBean(SimpleBean.class, qual, qual2);
```

In this example we manually construct instances of qualifier annotations in order to pass it to the bean manager for lookup. This is a necessary step since there's currently no support for annotation literals in Errai client code.

3.5.2. Availability of beans

Not all beans that are available for injection are available for lookup from the bean manager by default. Only beans which are *explicitly* scoped are available for dynamic lookup. This is an intentional feature to keep the size of the generated code down in the browser.

3.6. Alternatives and Mocks

3.6.1. Alternatives

It may be desirable to have multiple matching dependencies for a given injection point with the ability to specify which implementation to use at runtime. For instance, you may have different versions of your application which target different browsers or capabilities of the browser. Using alternatives allows you to share common interfaces among your beans, while still using dependency injection, by exporting consideration of what implementation to use to the container's configuration.

Consider the following example:

```
@Singleton @Alternative
```

```
public class MobileView implements View {
  // ... //
}
```

and

```
@Singleton @Alternative
public class DesktopView implements View {
  // ... //
```

In our controller logic we in turn inject the `View` interface:

```
@EntryPoint
public class MyApp {
  @Inject
  View view;

  // ... //
}
```

This code is unaware of the implementation of `View`, which maintains good separation of concerns. However, this of course creates an ambiguous dependency on the `View` interface as it has two matching subtypes in this case. Thus, we must configure the container to specify which alternative to use. Also note, that the beans in both cases have been annotated with `javax.enterprise.inject.Alternative`.

In your `ErraiApp.properties` for the module, you can simply specify which active alternative should be used:

```
errai.ioc.enabled.alternatives=org.foo.MobileView
```

You can specify multiple alternative classes by white space separating them:

```
errai.ioc.enabled.alternatives=org.foo.MobileView \
                               org.foo.HTML5Orientation \
                               org.foo.MobileStorage
```

You can only have one enabled alternative for a matching set of alternatives, otherwise you will get ambiguous resolution errors from the container.

3.6.2. Test Mocks

Similar to alternatives, but specifically designed for testing scenarios, you can replace beans with mocks at runtime for the purposes of running unit tests. This is accomplished simply by annotating a bean with the `org.jboss.errai.ioc.client.api.TestMock` annotation. Doing so will prioritize consideration of the bean over any other matching beans while running unit tests.

Consider the following:

```
@ApplicationScoped
public class UserManagementImpl implements UserManagement {
  public List<User> listUsers() {
     // do user listy things!
  }
}
```

You can specify a mock implementation of this class by implementing its common parent type (`UserManagement`) and annotating that class with the `@TestMock` annotation inside your test package like so:

```
@TestMock @ApplicationScoped
public class MockUserManagementImpl implements UserManagement {
  public List<User> listUsers() {
     // return only a test user.
     return Collections.singletonList(TestUser.INSTANCE);
  }
}
```

In this case, the container will replace the `UserManagementImpl` with the `MockUserManagementImpl` automatically when running the unit tests.

The `@TestMock` annotation can also be used to specify alternative providers during test execution. For example, it can be used to mock a `Caller<T>`. `Callers` are used to invoke RPC or JAX-RS endpoints. During tests you might want to replace theses callers with mock implementations. For details on providers see *Container Wiring*.

```
@TestMock @IOCProvider
public class MockedHappyServiceCallerProvider implements ContextualTypeProvider<Caller<HappySer

  @Override
  public Caller<HappyService> provide(Class<?>[] typeargs, Annotation[] qualifiers) {
    return new Caller<HappyService>() {
      ...
    }
```

```
}
```

3.7. Bean Lifecycle

All beans managed by the Errai IOC container support the `@PostConstruct` and `@PreDestroy` annotations.

Beans which have methods annotated with `@PostConstruct` are guaranteed to have those methods called before the bean is put into service, and only after all dependencies within its graph has been satisfied.

Beans are also guaranteed to have their `@PreDestroy` annotated methods called before they are destroyed by the bean manager.

Important

This cannot be guaranteed when the browser DOM is destroyed prematurely due to: closing the browser window; closing a tab; refreshing the page, etc.

3.7.1. Destruction of Beans

Beans under management of Errai IOC, of any scope, can be explicitly destroyed through the client bean manager. Destruction of a managed bean is accomplished by passing a reference to the `destroyBean()` method of the bean manager.

Example 3.13. Destruction of bean

```
public MyManagedBean {
  @Inject SyncBeanManager manager;

  public void createABeanThenDestroyIt() {
    // get a new bean.
    SimpleBean bean = manager.lookupBean(SimpleBean.class).getInstance();

    bean.sendMessage("Sorry, I need to dispose of you now");

    // destroy the bean!
    manager.destroyBean(bean);
  }
}
```

When the bean manager "destroys" the bean, any pre-destroy methods the bean declares are called, it is taken out of service and no longer tracked by the bean manager. If there are references on the bean by other objects, the bean will continue to be accessible to those objects.

Important

Container managed resources that are dependent on the bean such as bus service endpoints or CDI event observers will also be automatically destroyed when the bean is destroyed.

Another important consideration is the rule, "all beans created together are destroyed together." Consider the following example:

Example 3.14. SimpleBean.class

```
@Dependent
public class SimpleBean {
   @Inject @New AnotherBean anotherBean;

   public AnotherBean getAnotherBean() {
     return anotherBean;
   }

   @PreDestroy
   private void cleanUp() {
     // do some cleanup tasks
   }
}
```

Example 3.15. Destroying bean from subgraph

```
public MyManagedBean {
  @Inject SyncBeanManager manager;

  public void createABeanThenDestroyIt() {
    // get a new bean.
    SimpleBean bean = manager.lookupBean(SimpleBean.class).getInstance();

    // destroy the AnotherBean reference from inside the bean
    manager.destroyBean(bean.getAnotherBean());
  }
}
```

In this example we pass the instance of `AnotherBean`, created as a dependency of `SimpleBean`, to the bean manager for destruction. Because this bean was created at the same time as its parent, its destruction will also result in the destruction of `SimpleBean`; thus, this action will result in the `@PreDestroy`[code]`cleanUp()` method of `SimpleBean` being invoked.

3.7.1.1. Disposers

Another way which beans can be destroyed is through the use of the injectable `org.jboss.errai.ioc.client.api.Disposer<T>` class. The class provides a straight forward way of disposing of bean type.

For instance:

Example 3.16. Destroying bean with disposer

```
public MyManagedBean {
  @Inject @New SimpleBean myNewSimpleBean;
  @Inject Disposer<SimpleBean> simpleBeanDisposer;

  public void destroyMyBean() {
    simpleBeanDisposer.dispose(myNewSimpleBean);
  }
}
```

Errai CDI

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai CDI* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai CDI to your project.

CDI (Contexts and Dependency Injection) is the Jave EE standard (JSR-299) for handling dependency injection. In addition to dependency injection, the standard encompasses component lifecycle, application configuration, call-interception and a decoupled, type-safe eventing specification.

The Errai CDI extension implements a subset of the specification for use inside of client-side applications within Errai, as well as additional capabilities such as distributed eventing.

Errai CDI does not currently implement all life cycles specified in JSR-299 or interceptors. These deficiencies may be addressed in future versions.

Important

Errai CDI is implemented as an extension on top of the Errai IOC Framework (see *Dependency Injection*), which itself implements JSR-330. Inclusion of the CDI module your GWT project will result in the extensions automatically being loaded and made available to your application.

Classpath Scanning and ErraiApp.properties

Errai CDI only scans the contents of classpath locations (JARs and directories) that have *a file called ErraiApp.properties* at their root. If CDI features such as dependency injection, event observation, and `@PostConstruct` are not working for your classes, double-check that you have an `ErraiApp.properties` at the root of every JAR and directory tree that contains classes Errai should know about.

4.1. Features and Limitations

Beans that are deployed to a CDI container will automatically be registered with Errai and exposed to your GWT client application. So, you can use Errai to communicate between your GWT client components and your CDI backend beans.

Errai CDI based applications use the same annotation-driven programming model as server-side CDI components, with some notable limitations. Many of these limitations will be addressed in future releases.

1. There is no support for CDI interceptors in the client. Although this is planned in a future release.
2. Passivating scopes are not supported.
3. The JSR-299 SPI is not supported for client side code. Although writing extensions for the client side container is possible via the Errai IOC Extensions API.
4. The `@Typed` annotation is unsupported.
5. The `@Interceptor` annotation is unsupported.
6. The `@Decorator` annotation is unsupported.

4.1.1. Other features

The CDI container in Errai is built around the *Errai IOC module*, and thus is a superset of the existing functionality in Errai IOC. Thus, all features and APIs documented in Errai IOC are accessible and usable with this Errai CDI programming model.

4.2. Events

Any CDI managed component may produce and consume *events* [http://docs.jboss.org/weld/reference/latest/en-US/html/events.html] . This allows beans to interact in a completely decoupled fashion. Beans consume events by registering for a particular event type and optional qualifiers. The Errai CDI extension simply extends this concept into the client tier. A GWT client application can simply register an `Observer` for a particular event type and thus receive events that are produced on the server-side. Likewise and using the same API, GWT clients can produce events that are consumed by a server-side observer.

Let's take a look at an example.

Example 4.1. FraudClient.java

```
public class FraudClient extends LayoutPanel {

  @Inject
  private Event<AccountActivity> event; (1)

  private HTML responsePanel;
```

```
  public FraudClient() {
    super(new BoxLayout(BoxLayout.Orientation.VERTICAL));
  }

  @PostConstruct
  public void buildUI() {
    Button button = new Button("Create activity", new ClickHandler() {
      public void onClick(ClickEvent clickEvent) {
        event.fire(new AccountActivity());
      }
    });
    responsePanel = new HTML();
    add(button);
    add(responsePanel);
  }

  public void processFraud(@Observes @Detected Fraud fraudEvent) { (2)
    responsePanel.setText("Fraud detected: " + fraudEvent.getTimestamp());
  }
}
```

Two things are noteworthy in this example:

1. Injection of an `Event` dispatcher proxy
2. Creation of an `Observer` method for a particular event type

The event dispatcher is responsible for sending events created on the client-side to the server-side event subsystem (CDI container). This means any event that is fired through a dispatcher will eventually be consumed by a CDI managed bean, if there is an corresponding `Observer` registered for it on the server side.

In order to consume events that are created on the server-side you need to declare an client-side observer method for a particular event type. In case an event is fired on the server this method will be invoked with an event instance of type you declared.

To complete the example, let's look at the corresponding server-side CDI bean:

Example 4.2. AccountService.java

```
@ApplicationScoped
public class AccountService {

  @Inject @Detected
  private Event<Fraud> event;

  public void watchActivity(@Observes AccountActivity activity) {
```

```
    Fraud fraud = new Fraud(System.currentTimeMillis());
    event.fire(fraud);
 }
}
```

4.2.1. Conversational events

A server can address a single client in response to an event annotating event types as `@Conversational`. Consider a service that responds to a subscription event.

Example 4.3. SubscriptionService.java

```
@ApplicationScoped
public class SubscriptionService {

  @Inject
  private Event<Documents> welcomeEvent;

  public void onSubscription(@Observes Subscription subscription) {
    Document docs = createWelcomePackage(subscription);
    welcomeEvent.fire(docs);
 }
}
```

And the `Document` class would be annotated like so:

Example 4.4. Document.java

```
@Conversational @Portable
public class Document {
   // code here
}
```

As such, when `Document` events are fired, they will be limited in scope to the initiating conversational contents which are implicitly inferred by the caller. So only the client which fired the `Subscription` event will receive the fired `Document` event.

4.2.2. Local Events

The simplest way to stop a CDI Event from being broadcast over the wire is to avoid annotating the type with `@Portable`. But in some cases you may wish to send a type over the network with Errai RPC or the Message Bus, but only fire it locally as a CDI Event.

This can be accomplished by annotating a type with `@LocalEvent`, as in this example:

```
@Portable @LocalEvent
public DocumentChange {
  private String diff;

  public String getDiff() {
    return diff;
  }

  public void setDiff(String diff) {
    this.diff = diff;
  }
}
```

Because of the `@Portable` annotation instances of `DocumentChange` can be sent over the wire via RPC calls or bus messages, but because of the `@LocalEvent` annotation they will not be sent over the network if fired via a CDI Event.

4.2.3. Client-Server Event Example

A key feature of the Errai CDI framework is the ability to federate the CDI eventing bus between the client and the server. This permits the observation of server produced events on the client, and vice-versa.

Example server code:

Example 4.5. MyServerBean.java

```
@ApplicationScoped
public class MyServerBean {
  @Inject
  Event<MyResponseEvent> myResponseEvent;

  public void myClientObserver(@Observes MyRequestEvent event) {
    MyResponseEvent response;

    if (event.isThankYou()) {
      // aww, that's nice!
      response = new MyResponseEvent("Well, you're welcome!");
    }
    else {
      // how rude!
      response = new MyResponseEvent("What? Nobody says 'thank you' anymore?");
    }

    myResponseEvent.fire(response);
  }
```

```
}
```

Domain-model:

Example 4.6. MyRequestEvent.java

```
@Portable
public class MyRequestEvent {
  private boolean thankYou;

  public MyRequestEvent(boolean thankYou) {
    setThankYou(thankYou);
  }

  public void setThankYou(boolean thankYou) {
    this.thankYou = thankYou;
  }

  public boolean isThankYou() {
    return thankYou;
  }
}
```

Example 4.7. MyResponseEvent.java

```
@Portable
public class MyResponseEvent {
  private String message;

  public MyRequestEvent(String message) {
    setMessage(message);
  }

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

  public String getMessage() {
    return message;
  }
}
```

Client application logic:

Example 4.8. MyClientBean.java

```
@EntryPoint
public class MyClientBean {
   @Inject
   Event<MyRequestEvent> requestEvent;

   public void myResponseObserver(@Observes MyResponseEvent event) {
     Window.alert("Server replied: " + event.getMessage());
   }

   @PostConstruct
   public void init() {
     Button thankYou = new Button("Say Thank You!");
     thankYou.addClickHandler(new ClickHandler() {
       public void onClick(ClickEvent event) {
         requestEvent.fire(new MyRequestEvent(true));
       }
     }

     Button nothing = new Button("Say nothing!");
     nothing.addClickHandler(new ClickHandler() {
       public void onClick(ClickEvent event) {
         requestEvent.fire(new MyRequestEvent(false));
       }
     }

     VerticalPanel vPanel = new VerticalPanel();
     vPanel.add(thankYou);
     vPanel.add(nothing);

     RootPanel.get().add(vPanel);
   }
}
```

4.3. Producers

Producer methods and fields act as sources of objects to be injected. They are useful when additional control over object creation is needed before injections can take place e.g. when you need to make a decision at runtime before an object can be created and injected.

Example 4.9. App.java

```
@EntryPoint
public class App {
  ...
```

```
  @Produces @Supported
  private MyBaseWidget createWidget() {
    return (Canvas.isSupported()) ? new MyHtml5Widget() : new MyDefaultWidget();
  }
}
```

Example 4.10. MyComposite.java

```
@ApplicationScoped
public class MyComposite extends Composite {

  @Inject @Supported
  private MyBaseWidget widget;

  ...
}
```

Producers can also be scoped themselves. By default, producer methods are dependent-scoped, meaning they get called every time an injection for their provided type is requested. If a producer method is scoped `@Singleton` for instance, the method will only be called once, and the bean manager will inject the instance from the first invokation of the producer into every matching injection point.

Example 4.11. Singleton producer

```
public class App {
  ...

  @Produces @Singleton
  private MyBean produceMyBean() {
    return new MyBean();
  }
}
```

For more information on CDI producers, see the *CDI specification* [http://docs.jboss.org/cdi/spec/1.0/html/] and the *WELD reference documentation* [http://seamframework.org/Weld/WeldDocumentation] .

4.4. Safe dynamic lookup

As an alternative to using the bean manager to dynamically create beans, this can be accomplished in a type-safe way by injecting a `javax.enterprise.inject.Instance<T>`.

For instance, assume you have a dependent-scoped bean `Bar` and consider the following:

```
public class Foo {
  @Inject Instance<Bar> barInstance;

  public void pingNewBar() {
    Bar bar = barInstance.get();
    bar.ping();
  }
}
```

In this example, calling `barInstance.get()` returns a new instance of the dependent-scoped bean `Bar`.

4.5. Deploying Errai CDI

The CDI integration is a plugin to the Errai core framework and represents a CDI portable extension. Which means it is discovered automatically by both Errai and the CDI container. In order to use it, you first need to understand the different runtime models involved when working GWT, Errai, and CDI.

Typically a GWT application lifecycle begins in *Development Mode* [http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html] and finally a web application containing the GWT client code will be deployed to a target container (Servlet Engine, Application Server). This is no way different when working with CDI components to back your application.

What's different however is availability of the CDI container across the different runtimes. In GWT development mode and in a pure servlet environment you need to provide and bootstrap the CDI environment on your own. While any Java EE 6 Application Server already provides a preconfigured CDI container. To accomodate these differences, we need to do a little trickery when executing the GWT Development Mode and packaging our application for deployment.

Plugin Tip

Use the *Errai Forge Addon* *Add Errai in Project* command to setup development mode, usable for any Errai application, including one with Errai CDI.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually setup Errai CDI in Development Mode.

Marshalling

Errai includes a comprehensive marshalling framework which permits the serialization of domain objects between the browser and the server. From the perspective of GWT, this is a complete replacement for the provided GWT serialization facilities and offers a great deal more flexibility. You can use it with your own application-specific domain models as well as preexisting models, including classes from third-party libraries using configuration files or the custom definitions API.

Plugin Tip

If you used the *Errai Forge Addon* *Add Errai Features* command to add *Errai Messaging* or *Errai CDI* then Marshalling is already available to you.

5.1. Mapping Your Domain

All classes that you intend to be marshalled between the client and the server must be exposed to the marshalling framework explicitly. There are several ways to do that, and this section will take you through the different approaches.

5.1.1. @Portable and @NonPortable

The easiest way to make a Java class eligible for serialization with Errai Marshalling is to mark it with the `org.jboss.errai.common.client.api.annotations.Portable` annotation. This tells the marshalling system to generate marshalling and demarshalling code for the annotated class and all of its nested classes.

The mapping strategy that will be used depends on how much information you provide about your model up-front. If you simply annotate a domain type with `@Portable` and do nothing else, the marshalling system will use an exhaustive strategy to determine how to serialize and deserialize instances of that type and its nested types.

The Errai marshalling system works by enumerating all of the Portable types it can find (by any of the three methods discussed in this section of the reference guide), eliminating all the non-portable types it can find (via `@NonPortable` annotations and entries in `ErraiApp.properties`), then enumerating the marshallable properties that make up each remaining portable entity type. The rules that Errai uses for enumerating the properties of a portable entity type are as follows:

- If an entity type has a field called `foo`, then that entity has a property called `foo` unless the field is marked `static` or [code]+transient.

Note that the existence of methods called `getFoo()`, `setFoo()`, or both, *does not* mean that the entity has a property called `foo`. Errai Marshalling always works from fields when discovering properties.

When reading a field `foo`, Errai Marshalling will call the method `getFoo()` in preference to direct field access if the `getFoo()` method exists.

When writing a field `foo`, Errai Marshalling gives first preference to a parameter of the *mapping constructor* (defined below) annotated with `@MapsTo("foo")`. Failing this, Errai Marshalling will the method `setFoo()` if it exists. As a last resort, Errai Marshalling will use direct field access to write the value of `foo`.

Each field is mapped independently according to the above priority rules. Portable classes can rely on a mix of constructor, setter/getter, and field access.

For de-marshalling a `@Portable` type, Errai must know how to obtain a new instance of that type. To do this, it selects a *mapping constructor* (which could literally be a constructor, but could also be a static factory method) using the following rules:

- If the entity has a constructor where every argument is annotated with `@MapsTo`, then this is the mapping constructor. The constructor doesn't have to be public.
- Otherwise, if the entity has a public static method whose return type matches the entity type and every argument is annotated with `@MapsTo`, then it is the mapping constructor. Unlike a constructor, such a method is free to return an instance of a subtype of the marshalled type, or resolve an instance from a cache. In this case, do keep in mind that left-over properties not covered by the method's `@MapsTo` parameters will still be written by setters and direct field access.
- Otherwise, if the entity has a public no-arguments constructor (including the one the Java compiler provides in the absence of explicit constructors), it is the mapping constructor.

If no suitable mapping constructor can be found on a type marked `@Portable`, it is a compile-time error.

Now let's take a look at some common examples of how this works.

5.1.1.1. Example: A Common Mutable Bean

```
@Portable
public class Person {
  private String name;
  private int age;

  public String getName() {
    return name;
  }

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

```
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }
}
```

This class is a straightforward mutable bean. Errai will read and write the name and age properties via the setter and getter methods. It will create new instances of the type using the default no-args public constructor that the Java compiler generated implicitly.

5.1.1.2. Example: An Immutable Entity with a Public Constructor

It's always good to aim for truly immutabile value types wherever practical, and Errai's marshalling system does not force you to compromise on this ideal.

```
@Portable
public final class Person {
  private final String name;
  private final int age;

  public Person(@MapsTo("name") String name, @MapsTo("age") int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }
}
```

Both fields are final, so they cannot be written by setter methods or by direct field access. But that's okay, because we have given Errai a way to set them using the annotated constructor parameters.

5.1.1.3. Example: An Immutable Entity with a Factory Method

Another good practice is to use a factory pattern to enforce invariance. Once again, let's modify our example.

```
@Portable
```

```
public class Person {
  private final String name;
  private final int age;

  private Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public static Person createPerson(@MapsTo("name") String name, @MapsTo("age") int age) {
    return new Person(name, age);
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }
}
```

Here we have made our only declared constructor private, and created a static factory method. Notice that we've simply used the same `@MapsTo` annotation in the same way we did on the constructor from our previous example. The marshaller will see this method and know that it should use it to construct the object.

5.1.1.4. Example: An Immutable Entity with a Builder

For types with a large number of optional attributes, a builder is often the best approach.

```
@Portable
public class Person {
  private final String name;
  private final int age;

  private Person(@MapsTo("name") String name, @MapsTo("age") int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
```

```
  }

  @NonPortable
  public static class Builder {
    private String name;
    private int age;

    public Builder name(String name) {
      this.name = name;
      return this;
    }

    public Builder age(int age) {
      this.age = age;
      return this;
    }

    public Person build() {
      return new Person(name, age);
    }
  }
}
```

In this example, we have a nested `Builder` class that implements the Builder Pattern and calls the private `Person` constructor. Hand-written code will always use the builder to create `Person` instances, but the `@MapsTo` annotations on the private `Person` constructor tell Errai Marshalling to bypass the builder and construct instances of Person directly.

One final note: as a nested type of `Person` (which is marked `@Portable`), the builder itself would normally be portable. However, we do not intend to move instances of `Person.Builder` across the network, so we mark `Person.Builder` as `@NonPortable`.

5.1.2. Manual Mapping

Some classes may be out of your control, making it impossible to annotate them for auto-discovery by the marshalling framework. For cases such as this, there are two approaches to include these classes in your application.

The first approach is the easiest, but is contingent on whether or not the class is directly exposed to the GWT compiler. That means, the classes must be part of a GWT module and within the GWT client packages. See the GWT documentation on *Client-Side Code* [http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsClient.html] for information on this.

5.1.2.1. Mapping Existing Client Classes

If you have client-exposed classes that cannot be annotated with the `@Portable` annotation, you may manually map these classes so that the marshaller framework will comprehend and produce marshallers for them and their nested types.

To do this, specify them in *ErraiApp.properties*, using the `errai.marshalling.serializableTypes` attribute with a whitespace separated list of classes to make portable.

Example 5.1. Example ErraiApp.properties defining portable classes.

```
errai.marshalling.serializableTypes=org.foo.client.UserEntity \
                                    org.foo.client.GroupEntity \
                                    org.abcinc.model.client.Profile
```

If any of the serializable types have nested classes that you wish to make non-portable, you can specify them like this:

Example 5.2. Example ErraiApp.properties defining nonportable classes.

```
errai.marshalling.nonserializableTypes=org.foo.client.UserEntity$Builder \
                                       org.foo.client.GroupEntity$Builder
```

5.1.2.2. Aliased Mappings of Existing Interface Contracts

The marshalling framework supports and promotes the concept of marshalling by interface contract, where possible. For instance, the framework ships with a marshaller which can marshall data to and from the `java.util.List` interface. Instead of having custom marshallers for classes such as `ArrayList` and `LinkedList`, by default, these implementations are merely aliased to the `java.util.List` marshaller.

There are two distinct ways to go about doing this. The most straightforward is to specify which marshaller to alias when declaring your class is `@Portable` .

```
package org.foo.client;

@Portable(aliasOf = java.util.List.class)
public MyListImpl extends ArrayList {
  // .. //
}
```

In the case of this example, the marshaller will not attempt to comprehend your class. Instead, it will merely rely on the `java.util.List` marshaller to dematerialize and serialize instances of this type onto the wire.

If for some reason it is not feasible to annotate the class, directly, you may specify the mapping in the *ErraiApp.properties* file using the `errai.marshalling.mappingAliases` attribute.

```
errai.marshalling.mappingAliases=org.foo.client.MyListImpl->java.util.List \
                                 org.foo.client.MyMapImpl->java.util.Map
```

The list of classes is whitespace-separated so that it may be split across lines.

The example above shows the equivalent mapping for the `MyListImpl` class from the previous example, as well as a mapping of a class to the `java.util.Map` marshaller.

The syntax of the mapping is as follows: `<class_to_map>`#[code]`<contract_to_map_to>`.

Aliases do not inherit fields!

When you alias a class to another marshalling contract, extended fields of the aliased class will not be available upon deserialization. For this you must provide custom marshallers for those classes.

5.1.3. Manual Class Mapping

Although the default marshalling strategies in Errai Marshalling will suit the vast majority of use cases, there may be situations where it is necessary to manually map your classes into the marshalling framework to teach it how to construct and deconstruct your objects.

This is accomplished by specifying `MappingDefinition` classes which inform the framework exactly how to read and write state in the process of constructing and deconstructing objects.

5.1.3.1. MappingDefinition

All manual mappings should extend the `org.jboss.errai.marshalling.rebind.api.model.MappingDefinition` class. This is base metadata class which contains data on exactly how the marshaller can deconstruct and construct objects.

Consider the following class:

```
public class MySuperCustomEntity {
   private final String mySuperName;
   private String mySuperNickname;

   public MySuperCustomEntity(String mySuperName) {
     this.mySuperName = mySuperName;;
   }

   public String getMySuperName() {
     return this.mySuperName;
   }
```

```
    public void setMySuperNickname(String mySuperNickname) {
      this.mySuperNickname = mySuperNickname;
    }

    public String getMySuperNickname() {
      return this.mySuperNickname;
    }
}
```

Let us construct this object like so:

```
  MySuperCustomEntity entity = new MySuperCustomEntity("Coolio");
  entity.setSuperNickname("coo");
```

It is clear that we may rely on this object's two getter methods to extract the totality of its state. But due to the fact that the `mySuperName` field is final, the only way to properly construct this object is to call its only public constructor and pass in the desired value of `mySuperName` .

Let us consider how we could go about telling the marshalling framework to pull this off:

```
@CustomMapping
public MySuperCustomEntityMapping extends MappingDefinition {
  public MySuperCustomEntityMapping() {
   super(MySuperCustomEntity.class);                                    //
  (1)

     SimpleConstructorMapping cnsMapping = new SimpleConstructorMapping();
   cnsMapping.mapParmToIndex("mySuperName", 0, String.class);           //
  (2)

     setInstantiationMapping(cnsMapping);

addMemberMapping(newWriteMapping("mySuperNickname",String.class,"setMySuperNickname"));//
  (3)

 addMemberMapping(newReadMapping("mySuperName",String.class,"getMySuperName"));//
  (4)
addMemberMapping(newReadMapping("mySuperNickname",String.class,"getMySuperNickname"));//
  (5)
   }
}
```

And that's it. This describes to the marshalling framework how it should go about constructing and deconstructing `MySuperCustomEntity`.

Paying attention to our annotating comments, let's describe what we've done here.

1. Call the constructor in `MappingDefinition` passing our reference to the class we are mapping.
2. Using the `SimpleConstructorMapping` class, we have indicated that a custom constructor will be needed to instantiate this class. We have called the `mapParmToIndex` method with three parameters. The first, `"mySupername"` describes the class field that we are targeting. The second parameter, the integer `0` indicates the parameter index of the constructor arguments that we'll be providing the value for the aforementioned field in this case the first and only, and the final parameter `String.class` tells the marshalling framework which marshalling contract to use in order to de-marshall the value.
3. Using the `WriteMapping` class, we have indicated to the marshaller framework how to write the `"mySuperNickname"` field, using the `String.class` marshaller, and using the setter method `setMySuperNickname` .
4. Using the `ReadMapping` class, we have indicated to the marshaller framework how to read the `"mySuperName"` field, using the `String.class` marshaller, and using the getter method `getMySuperName` .
5. Using the `ReadMapping` class, we have indicated to the marshaller framework how to read the `"mySuperNickname"` field, using the `String.class` marshaller, and using the getter method `getMySuperNickname` .

5.1.4. Custom Marshallers

There is another approach to extending the marshalling functionality that doesn't involve mapping rules, and that is to implement your own `Marshaller` class. This gives you complete control over the parsing and emission of the JSON structure.

The implementation of marshallers is made relatively straight forward by the fact that both the server and the client share the same JSON parsing API.

Consider the included `java.util.Date` marshaller that comes built-in to the marshalling framework:

Example 5.3. DataMarshaller.java from the built-in marshallers

```
@ClientMarshaller(Date.class)
@ServerMarshaller(Date.class)
public class DateMarshaller extends AbstractNullableMarshaller<Date> {
  @Override
  public Date[] getEmptyArray() {
    return new Date[0];
  }

  @Override
```

```
  public Date doNotNullDemarshall(final EJValue o, final MarshallingSession ctx) {
     if (o.isObject() != null) {
       EJValue qualifiedValue = o.isObject().get(SerializationParts.QUALIFIED_VALUE);
       if (!qualifiedValue.isNull() && qualifiedValue.isString() != null) {
       return new Date(Long.parseLong(qualifiedValue.isString().stringValue()));
       }
      EJValue numericValue = o.isObject().get(SerializationParts.NUMERIC_VALUE);
       if (!numericValue.isNull() && numericValue.isNumber() != null) {
         return new Date(new Double(numericValue.isNumber().doubleValue()).longValue());
       }
       if (!numericValue.isNull() && numericValue.isString() != null) {
         return new Date(Long.parseLong(numericValue.isString().stringValue()));
       }
     }

     return null;
  }

  @Override
  public String doNotNullMarshall(final Date o, final MarshallingSession ctx) {
              return    "{\""    +    SerializationParts.ENCODED_TYPE    +    "\":
\"" + Date.class.getName() + "\"," +
          "\"" + SerializationParts.OBJECT_ID + "\":\"" + o.hashCode() + "\"," +
                          "\""  +  SerializationParts.QUALIFIED_VALUE  +  "\":
\"" + o.getTime() + "\"}";
  }
}
```

The class is annotated with both `@ClientMarshaller` and `@ServerMarshaller` indicating that this class should be used for both marshalling on the client and on the server.

The `doNotNullDemarshall()` method is responsible for converting the given JSON object (which has already been parsed and verified non-null) into a Java object.

The `doNotNullMarshall()` method does roughly the inverse: it converts the given Java object into a String (which must be parseable as a JSON object) for transmission on the wire.

Remote Procedure Calls (RPC)

ErraiBus supports a high-level RPC layer to make typical client-server RPC communication easy on top of the bus. While it is possible to use ErraiBus without ever using this API, you may find it to be a more useful and concise approach for exposing services to the clients.

Please note that this API has changed since version 1.0. RPC services provide a way of creating type-safe mechanisms to make client-to-server calls. Currently, this mechanism only support client-to-server calls, and not vice-versa.

Plugin Tip

Using the *Errai Forge Addon* *Add Errai Features* command, Errai RPC can be used if *Errai Messaging* or *Errai CDI* have been installed.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai Messaging or Errai CDI to a project.

6.1. Creating an RPC Interface

Creating a service is straight forward. It requires the definition of a remote interface, and a service class which implements it. See the following:

```
@Remote
public interface MyRemoteService {
  public boolean isEveryoneHappy();
}
```

The `@Remote` annotation tells Errai that we'd like to use this interface as a remote interface. The remote interface must be part of of the GWT client code. It cannot be part of the server-side code, since the interface will need to be referenced from both the client and server side code. That said, the implementation of a service is relatively simple to the point:

```
@Service
public class MyRemoteServiceImpl implements MyRemoteService {

  public boolean isEveryoneHappy() {
    // blatently lie and say everyone's happy.
    return true;
```

```
  }
}
```

That's all there is to it. You use the same `@Service` annotation as described in Section 2.4. The presence of the remote interface tips Errai off as to what you want to do with the class.

Warning

Beginning with Errai 2.0.CR1, the default for automatic service discovery has changed in favour of CDI based applications, meaning RPC service discovery must be explicitly turned on in case Errai CDI is not used (the `weld-integration.jar` is not on the classpath). This can be done using an init-param in the servlet config of your web.xml:

```
<servlet>
  <servlet-name>ErraiServlet</servlet-name>
     <servlet-class>org.jboss.errai.bus.server.servlet.DefaultBlockingServlet</
servlet-class>
  <init-param>
    <param-name>auto-discover-services</param-name>
    <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

6.2. Making calls

Calling a remote service involves use of the `MessageBuilder` API. Since all messages are asynchronous, the actual code for calling the remote service involves the use of a callback, which we use to receive the response from the remote method. Let's see how it works:

```
MessageBuilder.createCall(new RemoteCallback<Boolean>() {
  public void callback(Boolean isHappy) {
    if (isHappy) Window.alert("Everyone is happy!");
  }
 }, MyRemoteService.class).isEveryoneHappy();
```

In the above example, we declare a remote callback that receives a Boolean, to correspond to the return value of the method on the server. We also reference the remote interface we are calling, and directly call the method. However, *don't be tempted to write code like this* :

```
boolean bool = MessageBuilder.createCall(..., MyRemoteService.class).isEveryoneHappy();
```

The above code will never return a valid result. In fact, it will always return null, false, or 0 depending on the type. This is due to the fact that the method is dispatched asynchronously, as in, it does not wait for a server response before returning control. The reason we chose to do this, as opposed to emulate the native GWT-approach, which requires the implementation of remote and async interfaces, was purely a function of a tradeoff for simplicity.

6.2.1. Proxy Injection

An alternative to using the `MessageBuilder` API is to have a proxy of the service injected.

```
@Inject
private Caller<MyRemoteService> remoteService;
```

For calling the remote service, the callback objects need to be provided to the `call` method before the corresponding interface method is invoked.

```
remoteService.call(callback).isEveryoneHappy();
```

The Errai IOC GWT module needs to be inherited to make use of caller injection. To do this, the following line needs to be added to the application's `*.gwt.xml` file. It is important that this line comes `after` the Errai Bus module:

```
  <inherits name="org.jboss.errai.ioc.Container"/>
```

6.3. Handling exceptions

Handling remote exceptions can be done by providing an `ErrorCallback` on the client:

```
MessageBuilder.createCall(
  new RemoteCallback<Boolean>() {
    public void callback(Boolean isHappy) {
      if (isHappy) Window.alert("Everyone is happy!");
    }
  },
  new ErrorCallback() {
    public boolean error(Message message, Throwable caught) {
      try {
        throw caught;
```

```
      }
      catch (NobodyIsHappyException e) {
        Window.alert("OK, that's sad!");
      }
      catch (Throwable t) {
        GWT.log("An unexpected error has occurred", t);
      }
      return false;
    }
  },
  MyRemoteService.class).isEveryoneHappy();
```

As remote exceptions need to be serialized to be sent to the client, the `@Portable` annotation needs to be present on the corresponding exception class (see *Marshalling*). Further the exception class needs to be part of the client-side code. For more details on `ErrorCallbacks` see *Handling Errors*.

6.3.1. Global RPC exception handler

In a scenario where many different remote calls potentially throw the same exception types (e.g. exceptions related to authentication or authorization) it can be easier to register a global exception handler instead of providing error callbacks at each RPC invocation. This global exception handler is called in case an exception occurs in the process of a remote call that has no error callback associated with it. So, it will handle an otherwise uncaught exception.

```
@UncaughtException
private void onUncaughtException(Throwable caught) {
  try {
    throw caught;
  }
  catch (UserNotLoggedInException e) {
    // navigate to login dialog
  }
  catch (Throwable t) {
    GWT.log("An unexpected error has occurred", t);
  }
}
```

6.4. Client-side Interceptors

Client-side remote call interceptors provide the ability to manipulate or bypass the remote call before it's being sent. This is useful for implementing crosscutting concerns like caching, for example when the remote call should be avoided if the data is already cached locally.

6.4.1. Annotating the Remote Interface

To have a remote call intercepted, either an interface method or the remote interface type has to be annotated with `@InterceptedCall`. If the type is annotated, all interface methods will be intercepted.

```
@Remote
public interface CustomerService {

  @InterceptedCall(MyCacheInterceptor.class)
  public Customer retrieveCustomerById(long id);
}
```

Note that an ordered list of interceptors can be used for specifying an interceptor chain e.g.

```
@InterceptedCall({MyCacheInterceptor.class, MySecurityInterceptor.class})
public Customer retrieveCustomerById(long id);
```

6.4.2. Implementing an Interceptor

Implementing an interceptor is easy:

```
public class MyCacheInterceptor implements RpcInterceptor {

  @Override
  public void aroundInvoke(final RemoteCallContext context) {
     // e.g check if the result is cached and carry out the actual call only
 in case it's not.
      context.proceed() // executes the next interceptor in the chain or the
 actual remote call.
     // context.setResult() // sets the result directly without carrying out
 the remote call.
   }
}
```

The `RemoteCallContext` passed to the `aroundInvoke` method provides access to the intercepted method's name and read/write access to the parameter values provided at the call site.

Calling `proceed` executes the next interceptor in the chain or the actual remote call if all interceptors have already been executed. If access to the result of the (asynchronous) remote call is needed in the interceptor, one of the overloaded versions of `proceed` accepting a `RemoteCallback` has to be used instead.

The result of the remote call can be manipulated by calling `RemoteCallContext.setResult()`.

Not calling `proceed` in the interceptor bypasses the actual remote call, passing `RestCallContext.getResult()` to the `RemoteCallBack` provided at the call site.

6.4.3. Annotating the Interceptor (alternative)

If you cannot (or do not wish to) annotate the remote interface you may instead define remote call interceptors by annotating the interceptor class itself with `@InterceptsRemoteCall`. This annotation requires the developer to specify the remote interface that should be intercepted. The interceptor will then be applied to all methods in that interface. If the interface method is annotated with `InterceptedCall`, that annotation will win out.

```
@InterceptsRemoteCall({ MyRemoteInterface.class, MyOtherRemoteInterface.class })
public class CustomRpcInterceptor implements RpcInterceptor {

  @Override
  public void aroundInvoke(final RemoteCallContext context) {
    // interceptor logic goes here
  }
}
```

This approach sacrifices granularity (you cannot intercept individual methods on the remote interface). However, it does allow method interception without modification to the remote interface (which is particularly useful when the developer is not in control of the remote interface).

6.4.4. Interceptors and IOC

It is worth noting that interceptors **may** be defined as managed beans using the `@Dependent`, `@Singleton`, or `@ApplicationScoped` annotations. If the Errai application is using IOC (i.e. imports the IOC Errai module) **and** the interceptor is annotated as a managed bean, then the IOC container will be used to get/create the interceptor instance. This allows developers to `@Inject` dependencies into interceptors. If IOC is not being used, or else the interceptor is not properly annotated, then the interceptor class will simply be instantiated via **new**.

6.5. Session and request objects in RPC endpoints

Before invoking an endpoint method Errai sets up an `RpcContext` that provides access to message resources that are otherwise not visible to RPC endpoints.

```
@Service
public class MyRemoteServiceImpl implements MyRemoteService {

  public boolean isEveryoneHappy() {
    HttpSession session = RpcContext.getHttpSession();
```

```
    ServletRequest request = RpcContext.getServletRequest();
    ...
    return true;
  }
}
```

6.6. Batching remote calls

Some use cases require multiple interactions with the server to complete. Errai's RPC mechanism allows for batched invocations of remote methods that will be executed using a single server round-trip. This is useful for reducing the number of simultaneous HTTP connections and at the same time allows for reusing and combining fine-grained remote services.

Injecting a BatchCaller instead of a Caller<T> is all it takes to make use of batched remote procedure calls.

```
@EntryPoint
public class MyBean {

 @Inject
 private BatchCaller batchCaller;

 private void someMethod() {
    // ...
    batchCaller.call(remoteCallback1, RemoteService1.class).method1();
    batchCaller.call(remoteCallback2, RemoteService2.class).method2();

    // Invokes the accumulated remote requests using a single server round-trip.
    batchCaller.sendBatch();
 }

}
```

The remote methods will get executed only after `sendBatch()` was called. The method `sendBatch` accepts an additional `RemoteCallback` instance as a parameter which will we invoked when all remote calls have completed in success. Consequently, an `ErrorCallback` can also be provided which will get executed for all remote calls that have completed in failure.

6.7. Asynchronous handling of RPCs on the server

If computing the result of an RPC call takes a significant amount of time (i.e. because a third party service needs to be contacted or a long running query needs to be executed) it might be preferable to release the request-processing thread so it can perform other work and provide the result in a different execution context farther in the future. So, the RPC endpoint method can return immediately and the thread handling the incoming request doesn't need to stay active until the

result is available. Computing and setting the result can be done in a different thread (i.e. from a smaller thread pool provided by a library).

Errai provides a special return type `CallableFuture` to indicate to the RPC system that the result of the remote method call will be provided asynchronously (after the remote method call has returned).

Here's an example returning a result of type String:

```
@Remote
public interface LongRunningService {
  public CallableFuture<String> someLongRunningTask();
}
```

```
@Service
public class LongRunningServiceImpl implements LongRunningService {
  @Override
  public CallableFuture<String> someLongRunningTask() {
    final CallableFuture<String> future = CallableFutureFactory.get().createFuture(String.class

   final ExecutorService executorService = Executors.newSingleThreadExecutor();
    executorService.submit(new Runnable() {
      @Override
      public void run() {
        try {
          Thread.sleep(5000);
          future.setValue("result");
        }
        catch (Throwable t) {
          t.printStackTrace();
        }
      }
    });
    executorService.shutdown();
    return future;
  }
}
```

Note that the client-side code does not change when using this feature and will work exactly as described in *Section 6.2, "Making calls"* i.e.:

```
MessageBuilder.createCall(new RemoteCallback<String>() {
  @Override
  public void callback(String response) {
    assertEquals("foobar", response);
```

```
    finishTest();
  }
}, LongRunningService.class).someLongRunningTask();
```

Errai JAX-RS

JAX-RS (Java API for RESTful Web Services) is a Java EE standard (JSR-311) for implementing REST-based Web services in Java. Errai JAX-RS brings this standard to the browser and simplifies the integration of REST-based services in GWT client applications. Errai can generate proxies based on JAX-RS interfaces which will handle all the underlying communication and serialization logic. All that's left to do is to invoke a Java method.

Errai's JAX-RS support consists of the following:

- A client-side API to communicate with JAX-RS endpoints
- A code generator that runs at your project's build time, providing proxy implementations for each JAX-RS resource interfaces visible within the GWT module
- Errai IoC and CDI providers that allow you to `@Inject` instances of {{Caller<T>} (the same API used in Errai RPC)}
- Integration with either Errai Marshalling or Jackson to translate request and response data between Java object and a string-based wire format

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai JAXRS* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add JAX-RS to your project. You can also go to the *Errai tutorial project* [https://github.com/errai/errai-tutorial/archive/master.zip] or the *demo collection* [https://github.com/errai/errai/tree/master/errai-demos] for an example of JAX-RS.

7.1. Server-Side JAX-RS Implementation

Errai's JAX-RS support consists mostly of features that make the client side easier and more reliable to maintain. You will need to use an existing third-party JAX-RS implementation on the server side. All Java EE 6 application servers include such a module out-of-the-box. If you are developing an application that you intend to deploy on a plain servlet container, you will have to choose a JAX-RS implementation (for example, RestEasy) and configure it properly in your web.xml.

Alternatively, you could keep your REST resource layer in a completely separate web application hosted on the same server (perhaps build an Errai JAX-RS client against an existing REST service

you developed previously). In this case, you could factor out the shared JAX-RS interface into a shared library, leaving the implementation in the non-Errai application.

Finally, you can take advantage of the cross-origin resource sharing (CoRS) feature in modern browsers and use Errai JAX-RS to send requests to a third-party server. The third-party server would have to be configured to allow cross-domain requests. In this case, you would write a JAX-RS-Annotated interface describing the remote REST resources, but you would not create an implementation of that interface.

7.2. Shared JAX-RS Interface

Errai JAX-RS works by leveraging standard Java interfaces that bear JAX-RS annotations. You will also want these interfaces visible to server-side code so that your JAX-RS resource classes can implement them (and inherit the annotations). This keeps the whole setup typesafe, and reduces duplication to the bare minimum. The natural solution, then is to put the JAX-RS interfaces under the client.shared package within your GWT module:

- project
  - src/main/java
    - com.mycompany.myapp
      - MyApp.gwt.xml *[the app's GWT module]*
    - com.mycompany.myapp.client.local
      - MyAppClientStuff.java *[code that @Injects Caller<MyAppRestResource>]*
    - com.mycompany.myapp.client.shared
      - CustomerService.java *[the JAX-RS interface]*
    - com.mycompany.myapp.server
      - CustomerServiceImpl.java *[the server-side JAX-RS resource implementation]*

The contents of the server-side files would be as follows:

Example 7.1. CustomerService.java

```
@Path("customers")
public interface CustomerService {
  @GET
  @Produces("application/json")
  public List<Customer> listAllCustomers();

  @POST
  @Consumes("application/json")
```

```
  @Produces("text/plain")
  public long createCustomer(Customer customer);
}
```

The above interface is visible both to server-side code and to client-side code. It is used by client-side code to describe the available operations, their parameter types, and their return types. If you use your IDE's refactoring tools to modify this interface, both the server-side and client-side code will be updated automatically.

Example 7.2. CustomerServiceImpl.java

```
public class CustomerServiceImpl implements CustomerService {

  @Override
  public List<Customer> listAllCustomers() {
    // Use a database API to look up all customers in back-end data store
    // Return the resulting list
  }

  @Override
  public long createCustomer(Customer customer) {
    // Store new Customer instance in back-end data store
  }
}
```

The above class implements the shared interface. Since it performs database and/or filesystem operations to manipulate the persistent data store, it is not GWT translatable, and it's therefore kept in a package that is not part of the GWT module.

Save typing and reduce duplication

Note that all JAX-RS annotations (`@Path`, `@GET`, `@Consumes`, and so on) can be inherited from the interface. You do not need to repeat these annotations in your resource implementation classes.

7.3. Creating Requests

This section assumes you have already set up the CustomerService JAX-RS endpoint as described in the previous section.

To create a request on the client, all that needs to be done is to invoke `RestClient.create()`, thereby providing the JAX-RS interface, a response callback and to invoke the corresponding interface method:

Example 7.3. App.java

```
...
Button create = new Button("Create", new ClickHandler() {
  public void onClick(ClickEvent clickEvent) {
    Customer customer = new Customer(firstName, lastName, postalCode);
   RestClient.create(CustomerService.class, callback).createCustomer(customer);
  }
});
...
```

For details on the callback mechanism see *Handling Responses*.

7.3.1. Proxy Injection

Injectable proxies can be used as an alternative to calling `RestClient.create()`.

```
@Inject
private Caller<CustomerService> customerService;
```

To create a request, the callback objects need to be provided to the `call` method before the corresponding interface method is invoked.

```
customerService.call(callback).listAllCustomers();
```

To use caller injection, your application needs to inherit the Errai IOC GWT module. To do this, just add this line to your application's `*.gwt.xml` file and make sure it comes `after` the Errai JAX-RS module (see *Getting Started*):

```
  <inherits name="org.jboss.errai.ioc.Container"/>
```

Note

The JAX-RS interfaces need to be visible to the GWT compiler and must therefore reside within the client packages (e.g. client.shared).

7.4. Handling Responses

An instance of Errai's `RemoteCallback<T>` has to be passed to the `RestClient.create()` call, which will provide access to the JAX-RS resource method's result. `T` is the return type of the JAX-RS resource method. In the example below it's just a `Long` representing a customer ID, but it can be any serializable type (see *Marshalling*).

```
RemoteCallback<Long> callback = new RemoteCallback<Long>() {
  public void callback(Long id) {
    Window.alert("Customer created with ID: " + id);
  }
};
```

A special case of this `RemoteCallback` is the `ResponseCallback` which can be used as an alternative. It provides access to the `Response` object representing the underlying HTTP response. This is useful when more details of the HTTP response are needed, such as headers and the status code. The `ResponseCallback` can also be used for JAX-RS interface methods that return a `javax.ws.rs.core.Response` type. In this case, the `MarshallingWrapper` class can be used to manually demarshall the response body to an entity of the desired type.

```
ResponseCallback callback = new ResponseCallback() {
  public void callback(Response response) {
    Window.alert("HTTP status code: " + response.getStatusCode());
    Window.alert("HTTP response body: " + response.getText());
  }
};
```

7.4.1. Handling Errors

For handling errors, Errai's error callback mechanism can be reused and an instance of `ErrorCallback` can optionally be passed to the `RestClient.create()` call. In case of an HTTP error, the `ResponseException` provides access to the `Response` object. All other `Throwables` indicate a communication problem.

```
ErrorCallback errorCallback = new RestErrorCallback() {
  public boolean error(Request request, Throwable throwable) {
    try {
      throw throwable;
    }
    catch (ResponseException e) {
      Response response = e.getResponse();
      // process unexpected response
      response.getStatusCode();
```

```
    }
    catch (Throwable t) {
      // process unexpected error (e.g. a network problem)
    }
    return false;
  }
};
```

To provide more customized error handling, Errai also defines client side exception handling via the `ClientExceptionMapper` interface. The client exception mapper allows developers to process a REST `Response` into a `Throwable` prior to the error being delivered to the `ErrorCallback` described above. The exception mapper class must be annotated with `javax.ws.rs.ext.Provider` as well as implement the `ClientExceptionMapper` interface.

```
@Provider
public class MyAppExceptionMapper implements ClientExceptionMapper {

  /**
   * @see org.jboss.errai.enterprise.client.jaxrs.ClientExceptionMapper#fromResponse(com.google
   */
  @Override
  public Throwable fromResponse(Response response) {
    if (response.getStatusCode() == 404)
      return new MyAppNotFoundException();
    else
      return new MyAppServerError(response.getStatusText());
  }

}
```

Must be used in conjunction with `RestErrorCallback`

It is important to note that the `ClientExceptionMapper` will only be invoked when the callback passed to the `Caller` is an instance of `RestErrorCallback`.

The `ClientExceptionMapper` will, by default, be invoked for every error response. However, Errai also provides the `org.jboss.errai.enterprise.shared.api.annotations.MapsFrom` annotation which provides for additional granularity. An exception mapper can be annotated so that it is only invoked for methods on specific REST interfaces.

```
@Provider
@MapsFrom({ SomeRestInterface.class })
public class SpecificClientExceptionMapper implements ClientExceptionMapper {
```

```
  /**
   * @see org.jboss.errai.enterprise.client.jaxrs.ClientExceptionMapper#fromResponse(com.google
   */
  @Override
  public Throwable fromResponse(Response response) {
     ... // Do something specific here
  }

}
```

7.5. Accesssing and aborting requests

An instance of Errai's `RequestCallback` can optionally be passed to the `RestClient.create()` call, which will provide access to the underlying HTTP request. The callback is invoked synchronously after the HTTP request has been initiated. This means that you will have access to the request immediately after the call.

```
RequestCallback requestCallback = new RequestCallback() {
  @Override
  public void callback(Request request) {
    this.request = request;
  }
};

RestClient.create(SearchService.class, callback, requestCallback).find(query);
```

This allows you to cancel a pending request. Alternatively, a `RequestHolder` can be used instead.

```
RequestHolder searchRequestHolder = new RequestHolder();
RestClient.create(SearchService.class, callback, searchRequestHolder).find(query);
...
if (searchRequestHolder.isAlive()) {
  searchRequestHolder.getRequest().cancel();
}
```

7.6. Client-side Interceptors

Client-side remote call interceptors provide the ability to manipulate or bypass the request before it's being sent. This is useful for implementing crosscutting concerns like caching or security features e.g:

- avoiding the request when the data is cached locally

- adding special HTTP headers or parameters to the request

7.6.1. Annotating the JAX-RS Interface

To have a JAX-RS remote call intercepted, either an interface method or the JAX-RS implementation class method has to be annotated with `@InterceptedCall`. If the type is annotated, all interface methods will be intercepted.

```
@Path("customers")
public interface CustomerService {

  @GET
  @Path("/{id}")
  @Produces("application/json")
  @InterceptedCall(MyCacheInterceptor.class)
  public Customer retrieveCustomerById(@PathParam("id") long id);
}
```

Note that an ordered list of interceptors can be used for specifying an interceptor chain e.g.

```
@InterceptedCall({MyCacheInterceptor.class, MySecurityInterceptor.class})
public Customer retrieveCustomerById(@PathParam("id") long id);
```

7.6.2. Implementing an Interceptor

Implementing an interceptor is easy:

```
public class MyCacheInterceptor implements RestClientInterceptor {

  @Override
  public void aroundInvoke(final RestCallContext context) {
    RequestBuilder builder = context.getRequestBuilder();
    builder.setHeader("headerName", "value");
    context.proceed();
  }
}
```

The `RestCallContext` passed to the `aroundInvoke` method provides access to the context of the intercepted JAX-RS (REST) remote call. It allows to read and write the parameter values provided at the call site and provides read/write access to the `RequestBuilder` instance which has the URL, HTTP headers and parameters set.

Calling `proceed` executes the next interceptor in the chain or the actual remote call if all interceptors have already been executed. If access to the result of the (asynchronous) remote call is needed in the interceptor, one of the overloaded versions of `proceed` accepting a `RemoteCallback` has to be used instead.

The result of the remote call can be manipulated by calling `RestCallContext.setResult()`.

Not calling `proceed` in the interceptor bypasses the actual remote call, passing `RestCallContext.getResult()` to the `RemoteCallBack` provided at the call site.

7.6.3. Annotating the Interceptor (alternative)

If you cannot (or do not wish to) annotate the JAX-RS interface you may instead define remote call interceptors by annotating the interceptor class itself with `@InterceptsRemoteCall`. This annotation requires the developer to specify the JAX-RS interface that should be intercepted. The interceptor will then be applied to all methods in that interface. If the interface method is annotated with `InterceptedCall`, that annotation will win out.

```
@InterceptsRemoteCall({ MyJaxrsInterface.class, MyOtherJaxrsInterface.class })
public class MyCacheInterceptor implements RestClientInterceptor {

  @Override
  public void aroundInvoke(final RestCallContext context) {
    // Do interceptor logic here
    context.proceed();
  }
}
```

This approach sacrifices granularity (you cannot intercept individual methods on the JAX-RS interface). However, it does allow method interception without modification to the JAX-RS interface (which is particularly useful when the developer is not in control of the JAX-RS interface).

7.6.4. Interceptors and IOC

It is worth noting that interceptors **may** be defined as managed beans using the `@Dependent`, `@Singleton`, or `@ApplicationScoped` annotations. If the Errai application is using IOC (i.e. imports the IOC Errai module) **and** the interceptor is annotated as a managed bean, then the IOC container will be used to get/create the interceptor instance. This allows developers to `@Inject` dependencies into interceptors. If IOC is not being used, or else the interceptor is not properly annotated, then the interceptor class will simply be instantiated via **new**.

7.7. Wire Format

Errai's JSON format will be used to serialize/deserialize your custom types. See *Marshalling* for details.

Alternatively, a Jackson compatible JSON format can be used on the wire. See *Configuration* for details on how to enable Jackson marshalling.

7.8. Path

All paths specified using the `@Path` annotation on JAX-RS interfaces are by definition relative paths. Therefore, by default, it is assumed that the JAX-RS endpoints can be found at the specified paths relative to the GWT client application's context path.

Configuring the Path

To learn more about configuring the path, checkout *the JAX-RS Configuration Section*.

Errai JPA

Starting with Errai 2.1, Errai implements a subset of JPA 2.0. With Errai JPA, you can store and retrieve entity objects on the client side, in the browser's local storage. This allows the reuse of JPA-related code (both entity class definitions and procedural logic that uses the EntityManager) between client and server.

Errai JPA implements the following subset of JPA 2.0:

- Annotation-based configuration
- Entity Types with
- Identifiers of any numeric type (int, long, short, etc.)
- Generated identifiers
- Regular attributes of any JPA Basic type (Java primitive types, boxed primitives, enums, BigInteger, BigDecimal, String, Date, Time, and Timestamp)
- Singular and Plural (collection-valued) attributes of other entity types
- All association types (one-to-one, one-to-many, many-to-one, many-to-many)
- All association cascade rules (ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH)
- Circular and self references work properly
- Polymorphic queries and collections (queries for a base entity type can result in instances of its subtypes)
- Property access by field or get/set methods
- Named, typed JPQL queries that select exactly one entity type
- With cascading fetch of related entities
- With or without `WHERE` clause
- All boolean, arithmetic, and string operators supported
- All String manipulation functions supported
- With or without `ORDER BY` clause
- Lifecycle events and entity lifecycle listeners
- Much of the Metamodel API (`Metamodel`, `EntityType`, `SingularAttribute`, `PluralAttribute` , etc.)

It's all client-side

Errai JPA is a declarative, typesafe interface to the web browser's `localStorage` object. As such it is a *client-side implementation* of JPA. Objects are stored and fetched from the browser's local storage, *not* from the JPA provider on the server side.

8.1. Getting Started

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai JPA* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai JPA to your project.

8.1.1. INF/persistence.xml

Errai ignores META-INF/persistence.xml for purposes of client-side JPA. Instead, Errai scans all Java packages that are part of your GWT modules for classes annotated with `@Entity`. This allows you the freedom of defining a persistence.xml that includes both shared entity classes that you use on the client and the server, plus server-only entities that are defined in a server-only package.

8.1.2. Declaring an Entity Class

Classes whose instances can be stored and retrieved by JPA are called *entities*. To declare a class as a JPA entity, annotate it with `@Entity`.

JPA requires that entity classes conform to a set of rules. These are:

- The class must have an ID attribute
- The class must have a public or protected constructor that takes no arguments
- The class must be public and nonfinal
- No methods or persistent fields of the class may be final
- The class must be a top-level type (not a nested or inner class)

Here is an example of a valid entity class with an ID attribute (`id`) and a String-valued persistent attribute (`name`):

```
@Entity
public class Genre {

  @Id @GeneratedValue
  private int id;

  private String name;

  // This constructor is used by JPA
  public Genre() {}

  // This constructor is not used by JPA
  public Genre(String name) {
    this();
    this.name = name;
  }


  // These getter and Setter methods are optional:

  public int getId() { return id; }
  public void setId(int id) { this.id = id; }

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

8.1.2.1. Entity Attributes

The state of fields and JavaBeans properties of entities are generally persisted with the entity instance. These persistent things are called *attributes* .

JPA Attributes are subdivided into two main types: *singular* and *plural* . Singular attributes are scalar types like `Integer` or `String`. Plural attributes are collection values, such as `List<Integer>` or `Set<String>`.

The values of singular attributes (and the elements of plural attributes) can be of any application-defined entity type or a JPA Basic type. The JPA basic types are all of the Java primitive types, all boxed primitives, enums, BigInteger, BigDecimal, String, Date (`java.util.Date` or `java.sql.Date`), Time, and Timestamp.

You can direct JPA to read and write your entity's attributes by direct field access or via JavaBeans property access methods (that is, "getters and setters"). Direct field access is the default. To request property access, annotate the class with `@Access(AccessType.PROPERTY)`. If using direct field access, attribute-specific JPA annotations should be on the fields themselves; when using property access, the attribute-specific annotations should be on the getter method for that property.

8.1.2.2. ID Attributes and Auto-Generated Identifiers

Each entity class must have exactly one ID attribute. The value of this attribute together with the fully-qualified class name uniquely identifies an instance to the entity manager.

ID values can be assigned by the application, or they can be generated by the JPA entity manager. To declare a generated identifier, annotate the field with `@GeneratedValue`. To declare an application-assigned identifier, leave off the `@GeneratedValue` annotation.

Generated identifier fields must not be initialized or modified by application code. Application-assigned identifier fields must be initialized to a unique value before the entity is persisted by the entity manager, but must not be modified afterward.

8.1.2.3. Single-valued Attributes

By default, every field of a JPA basic type is a persistent attribute. If a basic type field should not be presistent, mark it with `transient` or annotate it with `@Transient`.

Single-valued attributes of entity types must be annotated with `@OneToOne` or `@ManyToOne`.

Single-valued types that are neither entity types nor JPA Basic types are not presently supported by Errai JPA. Such attributes must be marked transient.

Here is an example of an entity with single-valued basic attributes and a single-valued relation to another entity type:

```
@Entity
public class Album {

  @GeneratedValue
  @Id
  private Long id;

  private String name;

  @ManyToOne
  private Artist artist;

  private Date releaseDate;

  private Format format;

  public Long getId() { return id; }
  public void setId(Long id) { this.id = id; }

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

```
  public Artist getArtist() { return artist; }
  public void setArtist(Artist artist) { this.artist = artist; }

  public Date getReleaseDate() { return releaseDate; }
 public void setReleaseDate(Date releaseDate) { this.releaseDate = releaseDate; }

  public Format getFormat() { return format; }
  public void setFormat(Format format) { this.format = format; }
}
```

8.1.2.4. Plural (collection-valued) Attributes

Collection-valued types `Collection<T>`, `Set<T>`, and `List<T>` are supported. JPA rules require that all access to the collections are done through the collection interface method; never by specific methods on an implementation.

The element type of a collection attribute can be a JPA basic type or an entity type. If it is an entity type, the attribute must be annotated with `@OneToMany` or `@ManyToMany`.

Here is an example of an entity with two plural attributes:

```
@Entity
public class Artist {

  @Id
  private Long id;

  private String name;

  // a two-way relationship (albums refer back to artists)
  @OneToMany(mappedBy="artist", cascade=CascadeType.ALL)
  private Set<Album> albums = new HashSet<Album>();

  // a one-way relationship (genres don't reference artists)
  @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE})
  private Set<Genre> genres = new HashSet<Genre>();

  public Long getId() { return id; }
  public void setId(Long id) { this.id = id; }

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

  public Set<Album> getAlbums() { return albums; }
  public void setAlbums(Set<Album> albums) { this.albums = albums; }

  public Set<Genre> getGenres() { return genres; }
  public void setGenres(Set<Genre> genres) { this.genres = genres; }
```

```
}
```

8.1.3. Entity Lifecycle States

8.1.3.1. Cascade Rules

When an entity changes state (more on this later), that state change can be cascaded automatically to related entity instances. By default, no state changes are cascaded to related entities. To request cascading of entity state changes, use the `cascade` attribute on any of the relationship quantifiers `@OneToOne`, `@ManyToOne`, `@OneToMany`, and `@ManyToMany`.

CascadeType value	Description
`PERSIST`	Persist the related entity object(s) when this entity is persisted
`MERGE`	Merge the attributes of the related entity object(s) when this entity is merged
`REMOVE`	Remove the related entity object(s) from persistent storage when this one is removed
`REFRESH`	Not applicable in Errai JPA
`DETACH`	Detach the related entity object(s) from the entity manager when this object is detached
`ALL`	Equivalent to specifying all of the above

For an example of specifying cascade rules, refer to the `Artist` example above. In that example, the cascade type on `albums` is `ALL`. When a particular `Artist` is persisted or removed, detached, etc., all of that artist's albums will also be persisted or removed, or detached correspondingly. However, the cascade rules for `genres` are different: we only specify `PERSIST` and `MERGE`. Because a `Genre` instance is reusable and potentially shared between many artists, we do not want to remove or detach these when one artist that references them is removed or detached. However, we still want the convenience of automatic cascading persistence in case we persist an `Artist` which references a new, unmanaged `Genre`.

8.1.4. Obtaining an instance of EntityManager

The entity manager provides the means for storing, retrieving, removing, and otherwise affecting the lifecycle state of entity instances.

To obtain an instance of EntityManager on the client side, use Errai IoC (or CDI) to inject it into any client-side bean:

```
@EntryPoint
public class Main {
  @Inject EntityManager em;
```

```
}
```

8.1.4.1. Storing and Updating Entities

To store an entity object in persistent storage, pass that object to the `EntityManager.persist()` method. Once this is done, the entity instance transitions from the *new* state to the *managed* state.

If the entity references any related entities, these entities must be in the managed state already, or have cascade-on-persist enabled. If neither of these criteria are met, an `IllegalStateException` will be thrown.

See an example in the following section.

8.1.4.2. Fetching Entities by ID

If you know the unique ID of an entity object, you can use the `EntityManager.find()` method to retrieve it from persistent storage. The object returned from the `find()` method will be in the managed state.

Example:

```
    // make it
    Album album = new Album();
    album.setArtist(null);
    album.setName("Abbey Road");
    album.setReleaseDate(new Date(-8366400000L));

    // store it
    EntityManager em = getEntityManager();
    em.persist(album);
    em.flush();
    em.detach(album);
    assertNotNull(album.getId());

    // fetch it
    Album fetchedAlbum = em.find(Album.class, album.getId());
    assertNotSame(album, fetchedAlbum);
    assertEquals(album.toString(), fetchedAlbum.toString());
```

8.1.4.3. Removing Entities from Persistent Storage

To remove a persistent managed entity, pass it to the `EntityManager.remove()` method. As the cascade rules specify, related entities will also be removed recursively.

Once an entity has been removed and the entity manager's state has been flushed, the entity object is unmanaged and back in the *new* state.

8.1.4.3.1. Clearing all Local Storage

Errai's EntityManager class provides a `removeAll()` method which removes everything from the browser's persistent store for the domain of the current webpage.

This method is not part of the JPA standard, so you must down-cast your client-side `EntityManager` instance to `ErraiEntityManager`. Example:

```
@EntryPoint
public class Main {

  @Inject EntityManager em;

  void resetJpaStorage() {
    ((ErraiEntityManager) em).removeAll();
  }
}
```

8.1.4.4. Detaching Entity Instances from the Entity Manager

For every entity instance in the managed state, changes to the attribute values of that entity are persisted to local storage whenever the entity manager is flushed. To prevent this automatic updating from happening, you can *detach* an entity from the entity manager. When an instance is detached, it is not deleted. All information about it remains in persistent storage. The next time that entity is retrieved, the entity manager will create a new and separate managed instance for it.

To detach one particular object along with all related objects whose cascade rules say so, call `EntityManager.detach()` and pass in that object.

To detach all objects from the entity manager at once, call `EntityManager.detachAll()` .

8.1.4.5. Testing if an Entity is in the Managed State

To check if a given object is presently managed by the entity manager, call `EntityManager.contains()` and pass in the object of interest.

8.1.5. Named Queries

To retrieve one or more entities that match a set of criteria, Errai JPA allows the use of JPA *named queries*. Named queries are declared in annotations on entity classes.

8.1.5.1. Declaring Named Queries

Queries in JPA are written in the JPQL language. As of Errai 2.1, Errai JPA does not support all JPQL features. Most importantly, implicit and explicit joins in queries are not yet supported. Queries of the following form generally work:

```
SELECT et FROM EntityType et WHERE [expression with constants, named parameters and attributes
```

Here is how to declare a JPQL query on an entity:

```
@NamedQuery(name="selectAlbumByName",  query="SELECT  a  FROM  Album  a  WHERE
 a.name=:name")
@Entity
public class Album {
  ... same as before ...
}
```

To declare more than one query on the same entity, wrap the `@NamedQuery` annotations in `@NamedQueries` like this:

```
@NamedQueries({
  @NamedQuery(name="selectAlbumByName", query="SELECT a FROM Album a WHERE a.name
 = :name"),
   @NamedQuery(name="selectAlbumsAfter",  query="SELECT  a  FROM  Album  a  WHERE
 a.releaseDate >= :startDate")
})
@Entity
public class Album {
  ... same as before ...
}
```

8.1.5.2. Executing Named Queries

To execute a named query, retrieve it by name and result type from the entity manager, set the values of its parameters (if any), and then call one of the execution methods `getSingleResult()` or `getResultList()`.

Example:

```
   TypedQuery<Album> q = em.createNamedQuery("selectAlbumByName", Album.class);
    q.setParameter("name", "Let It Be");
    List<Album> fetchedAlbums = q.getResultList();
```

8.1.6. Entity Lifecycle Events

To receive a notification when an entity instance transitions from one lifecycle state to another, use an entity lifecycle listener.

These annotations can be applied to methods in order to receive notifications at certain points in an entity's lifecycle. These events are delivered for direct operations initiated on the EntityManager as well as operations that happen due to cascade rules.

Annotation	Meaning
`@PrePersist`	The entity is about to be persisted or merged into the entity manager.
`@PostPersist`	The entity has just been persisted or merged into the entity manager.
`@PreUpdate`	The entity's state is about to be captured into the browser's localStorage.
`@PostUpdate`	The entity's state has just been captured into the browser's localStorage.
`@PreRemove`	The entity is about to be removed from persistent storage.
`@PostRemove`	The entity has just been removed from persistent storage.
`@PostLoad`	The entity's state has just been retrieved from the browser's localStorage.

JPA lifecycle event annotations can be placed on methods in the entity type itself, or on a method of any type with a public no-args constructor.

To receive lifecycle event notifications directly on the affected entity instance, create a no-args method on the entity class and annotate it with one or more of the lifecycle annotations in the above table.

For example, here is a variant of the Album class where instances receive notification right after they are loaded from persistent storage:

```
@Entity
public class Album {

  ... same as before ...

  @PostLoad
  public void postLoad() {
    System.out.println("Album " + getName() + " was just loaded into the entity
 manager");
  }
}
```

To receive lifecycle methods in a different class, declare a method that takes one parameter of the entity type and annotate it with the desired lifecycle annotations. Then name that class in the `@EntityListeners` annotation on the entity type.

The following example produces the same results as the previous example:

```
@Entity
@EntityListeners(StandaloneLifecycleListener.class)
public class Album {

  ... same as always ...

}

public class StandaloneLifecycleListener {

  @PostLoad
  public void albumLoaded(Album a) {
  public void postLoad() {
     System.out.println("Album " + a.getName() + " was just loaded into the
 entity manager");
  }
}
```

8.1.7. JPA Metamodel

Errai captures structural information about entity types at compile time and makes them available in the GWT runtime environment. The JPA metamodel includes methods for enumerating all known entity types and enumerating the singular and plural attributes of those types. Errai extends the JPA 2.0 Metamodel by providing methods that can create new instances of entity classes, and read and write attribute values of existing entity instances.

As an example of what is possible, this functionality could be used to create a reusable UI widget that can present an editable table of any JPA entity type.

To access the JPA Metamodel, call the `EntityManager.getMetamodel()` method. For details on what can be done with the stock JPA metamodel, see the API's javadoc or consult the JPA specification.

8.1.7.1. Errai Extensions to JPA Metamodel API

Wherever you obtain an instance of `SingularAttribute` from the metamodel API, you can down-cast it to `ErraiSingularAttribute`. Likewise, you can down-cast any `PluralAttribute` to `ErraiPluralAttribute` .

In either case, you can read the value of an arbitrary attribute by calling `ErraiAttribute.get()` and passing in the entity instance. You can set any attribute's value by calling `ErraiAttribute.set()`, passing in the entity instance and the new value.

In addition to `get()` and `set()`, `ErraiPluralAttribute` also has the `createEmptyCollection()` method, which creates an empty collection of the correct interface type for the given attribute.

8.1.8. JPA Features Not Implemented in Errai 2.4

The following features are not yet implemented, but could conceivably be implemented in a future Errai JPA release:

- Flush modes other than immediate
- Transactions, including EntityManager.getTransaction()
- In named queries:
- Joins and nested attribute paths (`a.b.c`) do not yet work, although single-step attribute paths (`a.b`) do.
- The `SELECT` clause must specify exactly one entity type. Selection of individual attributes is not yet implemented.
- Embedded collections
- Compound identifiers (presently, only basic types are supported for entity IDs)
- `EntityManager.refresh()` to pick up changes made in localStorage from a different browser window/tab.
- Criteria Queries
- The generated static Metamodel
- The `@PersistenceContext` annotation currently has no effect in client-side code (use `@Inject` instead)

The following may never be implemented due to limitations and restrictions in the GWT client-side environment:

- `EntityManager.createQuery(String, …)` (that is, unnamed queries) are impractical because JPQL queries are parsed at compile time, not in the browser.
- `EntityManager.createNativeQuery(String, …)` don't make sense because the underlying database is just a hash table. It does not have a query language.
- Persistent attributes of type `java.util.Calendar` because the `Calendar` class is not in GWT's JRE emulation library.

8.1.9. Other Caveats for Errai 2.1 JPA

We hope to remedy these shortcomings in a future release.

- In Dev Mode, changes to entity classes are not discovered on page refresh. You need to restart Dev Mode.
- The local data stored in the browser is not encrypted

8.2. Errai JPA Data Sync

Traditional JPA implementations allow you to store and retrieve entity objects on the server side. Errai's JPA implementation allows you to store and retrieve entity objects in the web browser using the same APIs. All that's missing is the ability to synchronize the stored data between the server side and the client side.

This is where Errai JPA Data Sync comes in: it provides an easy mechanism for two-way synchronization of data sets between the client and the server.

8.2.1. How To Use It

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai JPA Datasync* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai JPA Datasync to your project.

8.2.1.1. A Running Example

For the rest of this chapter, we will refer to the following Entity classes, which are defined in a `shared` package that's visible to client and server code:

```
@Portable
@Entity
@NamedQuery(name = "allUsers", query = "SELECT u FROM User u")
public class User {

  @Id
  @GeneratedValue
  private long id;

  private String name;

  // getters and setters omitted
```

```
}
```

```
@Portable
@Entity
@NamedQuery(name = "groceryListsForUser", query = "SELECT gl FROM GroceryList
 gl WHERE gl.owner=:user")
public class GroceryList {

  @Id
  @GeneratedValue
  private long id;

  @ManyToOne
  private User owner;

  @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REFRESH })
  private List<Item> items = new ArrayList<Item>();

  // getters and setters omitted
}
```

```
@Portable
@Entity
@NamedQuery(name = "allItems", query = "SELECT i FROM Item i")
public class Item {

  @Id
  @GeneratedValue
  private long id;

  private String name;
  private String department;
  private String comment;
  private Date addedOn;

  @ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REFRESH })
  private User addedBy;

  // getters and setters omitted
}
```

To summarize: there are three entity types: `User`, `GroceryList`, and `Item` . Each `GroceryList` belongs to a `User` and has a list of `Item` objects.

Note

All the entities involved in the data synchronization request must be marshallable via Errai Marshalling. This is normally accomplished by adding the `@Portable` annotation to each JPA entity class, but it is also acceptable to list them in `ErraiApp.properties` . See the *Marshalling* section for more details.

Now let's say we want to synchronize the data for all of a user's grocery lists. This will make them available for offline use through Errai JPA, and at the same time it will update the server with the latest changes made on the client. Ultimately, the sync operation is accomplished via an annotated method or an asynchronous call into `ClientSyncManager`, but first we have to prepare a few things on the client and the server.

8.2.1.2. Server Side DataSyncServiceImpl

A data sync operation begins when the client-side sync manager sends an *Errai RPC* request to the server. Although a server-side implementation of the remote interface is provided, you are responsible for implementing a thin wrapper around it. This wrapper serves two purposes:

1. It allows you to determine how to obtain a reference to the JPA EntityManager (and to choose which persistence context the server-side data sync will operate on)
2. It allows you to inspect the contents of each sync request and make security decisions about access to particular entities

If you are deploying to a container that supports CDI and EJB 3, you can use this DataSyncServiceImpl as a template for your own:

```
@Stateless @org.jboss.errai.bus.server.annotations.Service
public class DataSyncServiceImpl implements DataSyncService {

  @PersistenceContext
  private EntityManager em;

  private final JpaAttributeAccessor attributeAccessor = new JavaReflectionAttributeAccessor();

  @Inject private LoginService loginService;

  @Override
  public <X> List<SyncResponse<X>> coldSync(SyncableDataSet<X> dataSet, List<SyncRequestOperati

    // Ensure a user is logged in
    User currentUser = loginService.whoAmI();
    if (currentUser == null) {
      throw new IllegalStateException("Nobody is logged in!");
```

```
    }

    // Ensure user is accessing their own data!
    if (dataSet.getQueryName().equals("groceryListsForUser")) {
      User requestedUser = (User) dataSet.getParameters().get("user");
      if (!currentUser.getId().equals(requestedUser.getId())) {
        throw new AccessDeniedException("You don't have permission to sync user
 " + requestedUser.getId());
      }
    }
    else {
        throw new IllegalArgumentException("You don't have permission to sync
 dataset " + dataSet.getQueryName());
    }

    DataSyncService dss = new org.jboss.errai.jpa.sync.server.DataSyncServiceImpl(em, attribute
    return dss.coldSync(dataSet, remoteResults);
  }
}
```

If you are not using EJB 3, you will not be able to use the `@PersistenceContext` annotation. In this case, obtain a reference to your EntityManager the same way you would anywhere else in your application.

8.2.1.3. Client Side — Declarative

Like many Errai features, Errai JPA DataSync provides an annotation-driven programming model and a programmatic API. You can choose which to use based on your needs and preferences.

The declarative data sync API is driven by the `@Sync` annotation. Consider the following example client-side class:

```
 // This injected User object could have been set up in a @PostConstruct method
instead of being injected.
 @Inject
 private User syncThisUser;

 @Sync(query = "groceryListsForUser", params = { @SyncParam(name = "user", val = "{syncThisUse
 private void onDataSyncComplete(SyncResponses<GroceryList> responses) {
   Window.alert("Data Sync Complete!");
 }
```

By placing the above code snippet in a client-side bean, you tell Errai JPA Data Sync that, as long as a managed instance of the bean containing the `@Sync` method exists, the Data Sync system should keep all grocery lists belonging to the `syncThisUser` user in sync between the client-side JPA EntityManager and the server-side EntityManager. Right now, the data sets are kept in sync

using a sync request every 5 seconds. In the future, this may be optimised to an incremental approach that pushes changes as they occur.

The annotated method needs to have exactly one parameter of type `SyncResponses` and will be called each time a data sync operation has completed. All sync operations passed to the method will have already been applied to the local EntityManager, with conflicts resolved in favour of the server's version of the data. The original client values are available in the `SyncResponses` object, which gives you a chance to implement a different conflict resolution policy.

The `query` attribute on the `@Sync` annotation must refer to an existing JPA Named Query that is defined on a shared JPA entity class.

The `params` attribute is an array of `@SyncParam` annotations. There must be exactly one `@SyncParam` for each named parameter in the JPA query (positional parameters are not supported). If the `val` argument is surrounded with brace brackets (as it is in the example aboce) then it is interpreted as a reference to a declared or inherited field in the containing class. Otherwise, it is interpreted as a literal String value.

Note

Field-reference sync params are captured just after the bean's `@PostConstruct` method is invoked. This means that values of referenced fields can be provided using `@Inject` (which in turn could come from a CDI Producer method) or by code in the `@PostConstruct` method.

Transport (network) errors are logged to the slf4j logger channel `org.jboss.errai.jpa.sync.client.local.ClientSyncWorker`. As of Errai 3.0.0.M4, it is not possible to specify a custom error handler using the declarative API. See the next section for information about the programmatic API.

8.2.1.4. Client Side — Programmatic

```
@Inject private ClientSyncManager syncManager;
@Inject private EntityManager em;

public void syncGroceryLists(User forUser) {
  RemoteCallback<List<SyncResponse<GroceryList>>> onCompletion = new RemoteCallback<List<Sync
    @Override
    public void callback(List<SyncResponse<GroceryList>> response) {
      Window.alert("Data Sync Complete!");
    }
  };

  ErrorCallback<?> onError = new BusErrorCallback() {

    @Override
```

```
      public boolean error(Message message, Throwable throwable) {
        Window.alert("Data Sync failed!");
        return false;
      }
    };

    Map<String, Object> queryParams = new HashMap<String, Object>();
    queryParams.put("user", forUser);

    syncManager.coldSync("groceryListsForUser", GroceryList.class, queryParams, onCompletion, o
  }
```

Important

The `onCompletion` and `onError` callbacks are optional. In the unlikely case that your application doesn't care if a data sync request completed successfully, you can pass `null` for either callback.

Once your `onCompletion` callback has been notified, the server and client will have the same entities stored in their respective databases for all entities reachable from the given query result.

8.2.1.5. Dealing With Conflicts

When the client sends the sync request to the server, it includes information about the state it expects each entity to be in. If an entity's state on the server does not match this expected state on the client, the server ignores the client's change request and includes a `ConflictResponse` object in the sync reply.

When the client processes the sync responses from the server, it applies the new state from the server to the local data store. This overwrites the change that was initially requested from the client. In short, you could call this the "server wins" conflict resolution policy.

In some cases, your application may be able to do something smarter: apply domain-specific knowledge to merge the conflict automatically, or prompt the user to perform a manual merge. In order to do this, you will have to examine the server response from inside the `onCompletion` callback you provided to the `coldSync()` method:

```
    RemoteCallback<List<SyncResponse<GroceryList>>> onCompletion = new RemoteCallback<List<Sync
      @Override
      public void callback(List<SyncResponse<GroceryList>> responses) {
        for (SyncResponse<GroceryList> response : responses) {
          if (response instanceof ConflictResponse) {
            ConflictResponse<GroceryList> cr = (ConflictResponse<GroceryList>) response;
            List<Item> expectedItems = cr.getExpected().getItems();
            List<Item> serverItems = cr.getActualNew().getItems();
```

```
            List<Item> clientItems = cr.getRequestedNew().getItems();

            // merge the list of items by comparing each to expectedItems
            List<Item> merged = ...;

            // update local storage with the merged list
        em.find(GroceryList.class, cr.getActualNew().getId()).setItems(merged);
            em.flush();
          }
        }
      }
    };
```

Remember, because of Errai's default "server wins" resolution policy, the call to `em.find(GroceryList.class, cr.getActualNew().getId())` will return a GroceryList object that has already been updated to match the state present in `serverItems`.

Note

Searching for ConflictResponse objects in the `onCompletion` callback is the only way to recover client data that was clobbered in a conflict. If you do not merge this data back into local storage, or at least retain a reference to the `cr.getRequestedNew()` object, this conflicting client data will be lost forever.

In a future release of Errai JPA, we plan to provide a client-side callback mechanism for custom conflict handling. If such a callback is registered, it will override the default behaviour.

Data Binding

Errai's data binding module provides the ability to bind model objects to UI fields/widgets. The bound properties of the model and the UI components will automatically be kept in sync for as long as they are bound. So, there is no need to write code for UI updates in response to model changes and no need to register listeners to update the model in response to UI changes.

9.1. Getting Started

The data binding module is directly integrated with *Errai UI* and *Errai JPA* but can also be used as a standalone project in any GWT client application:

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai Data Binding* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai Data Binding to your project.

9.1.1. Bindable Objects

Objects that should participate in data bindings have to be marked as `@Bindable` and must follow Java bean conventions. All editable properties of these objects are then bindable to UI widgets.

Example 9.1. Customer.java

```
@Bindable
public class Customer {
  ...
  private String name;

  public String getName() {
    return name;
  }

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

```
}
```

Important

If you cannot or prefer not to annotate your classes with `@Bindable`, you can alternatively specify bindable types in your ErraiApp.properties using a whitespace-separated list of fully qualified class names: `errai.ui.bindableTypes=org.example.Model1 org.example.Model2`

9.1.2. Initializing a DataBinder

An instance of `DataBinder` is required to create bindings. It can either be

injected into a client-side bean:

```
public class CustomerView {
  @Inject
  private DataBinder<Customer> dataBinder;
}
```

or created manually:

```
DataBinder<Customer> dataBinder = DataBinder.forType(Customer.class);
```

In both cases above, the `DataBinder` instance is associated with a new instance of the model (e.g. a new `Customer` object). A `DataBinder` can also be associated with an already existing object:

```
DataBinder<Customer> dataBinder = DataBinder.forModel(existingCustomerObject);
```

In case there is existing state in either the model object or the UI components before the they are bound, initial state synchronization can be carried out to align the model and the corresponding UI fields.

For using the model object's state to set the initial values in the UI:

```
DataBinder<Customer> dataBinder = DataBinder.forModel(existingCustomerObject, InitialState.FROM
```

For using the UI values to set the initial state in the model object:

```
DataBinder<Customer> dataBinder = DataBinder.forModel(existingCustomerObject, InitialState.FROM
```

9.2. Creating Bindings

Bindings can be created by calling the `bind` method on a `DataBinder` instance, thereby specifying which widgets should be bound to which properties of the model. It is possible to use property chains for bindings, given that all nested properties are of bindable types. When binding to `customer.address.streetName`, for example, both `customer` and `address` have to be of a type annotated with `@Bindable`.

```
public class CustomerView {
  @Inject
  private DataBinder<Customer> dataBinder;

  private Customer customer;
  private TextBox nameTextBox = new TextBox();
  // more UI widgets...

  @PostConstruct
  private void init() {
    customer = dataBinder
        .bind(nameTextBox, "name")
        .bind(idLabel, "id")
        .getModel();
  }
}
```

After the call to `dataBinder.bind()` in the example above, the customer object's name property and the `nameTextBox` are kept in sync until either the `dataBinder.unbind()` method is called or the `CustomerView` bean is destroyed.

That means that a call to `customer.setName()` will automatically update the value of the TextBox and any change to the TextBox's value in the browser will update the customer object's name property. So, `customer.getName()` will always reflect the currently displayed value of the `TextBox`.

Note

It's important to retrieve the model instance using dataBinder.getModel() before making changes to it as the data binder will provide a proxy to the model to ensure that changes will update the corresponding UI components.

Tip

Errai also provides a *declarative binding API* that can be used to create bindings automatically based on matching field and model property names.

9.3. Specifying Converters

Errai has built-in conversion support for all Number types as well as Boolean and Date to java.lang.String and vice versa. However, in some cases it will be necessary to provide custom converters (e.g. if a custom date format is desired). This can be done on two levels.

9.3.1. Registering a global default converter

```
@DefaultConverter
public class MyCustomDateConverter implements Converter<Date, String> {

  private static final String DATE_FORMAT = "YY_DD_MM";

  @Override
  public Date toModelValue(String widgetValue) {
    return DateTimeFormat.getFormat(DATE_FORMAT).parse(widgetValue);
  }

  @Override
  public String toWidgetValue(Date modelValue) {
    return DateTimeFormat.getFormat(DATE_FORMAT).format((Date) modelValue);
  }
}
```

All converters annotated with `@DefaultConverter` will be registered as global defaults calling `Convert.registerDefaultConverter()`. Note that the `Converter` interface specifies two type parameters. The first one represents the type of the model field, the second one the type held by the widget (e.g. `String` for widgets implementing `HasValue<String>`). These default converters will be used for all bindings with matching model and widget types.

9.3.2. Providing a binding-specific converter

Alternatively, converter instances can be passed to the `dataBinder.bind()` calls.

```
dataBinder.bind(textBox, "name", customConverter);
```

Converters specified on the binding level take precedence over global default converters with matching types.

9.4. Property Change Handlers

In some cases keeping the model and the UI in sync is not enough. Errai's `DataBinder` allows for the registration of `PropertyChangeHandlers` for specific properties, property expressions or all properties of a bound model. A property expression can be a property chain such as customer.address.street. It can end in a wildcard to indicate that changes of any property of the corresponding bean should be observed (e.g `"customer.address.*"`). A double wildcard can be used at the end of a property expression to register a cascading change handler for any nested property (e.g `"customer.**"` or just `"**"`) .

This provides a uniform notification mechanism for model and UI value changes. `PropertyChangeHandlers` can be used to carry out any additional logic that might be necessary after a model or UI value has changed.

```
dataBinder.addPropertyChangeHandler(new PropertyChangeHandler() {
  @Override
  public void onPropertyChange(PropertyChangeEvent event) {
   Window.alert(event.getPropertyName() + " changed to:" + event.getNewValue());
  }
});
```

```
dataBinder.addPropertyChangeHandler("name", new PropertyChangeHandler() {
  @Override
  public void onPropertyChange(PropertyChangeEvent event) {
    Window.alert("name changed to:" + event.getNewValue());
  }
});
```

9.5. Declarative Binding

Programmatic binding as described above (see *Creating Bindings*) can be tedious when working with UI components that contain a large number of input fields. Errai provides an annotation-driven binding API that can be used to create bindings automatically which cuts a lot of boilerplate code. The declarative API will work in any *Errai IOC* managed bean (including *Errai UI* templates). Simply inject a data binder or model object and declare the bindings using `@Bound` .

Here is a simple example using an injected model object provided by the `@Model` annotation (field injection is used here, but constructor and method injection are supported as well):

```
@Dependent
public class CustomerView {
  @Inject @Model
  private Customer customer;
```

```
  @Inject @Bound
  private TextBox name;

  @Bound
  private Label id = new Label();

  ....
}
```

Here is the same example injecting a `DataBinder` instead of the model object. This is useful when more control is needed (e.g. the ability to register property change handlers). The `@AutoBound` annotation specifies that this `DataBinder` should be used to bind the model to all enclosing widgets annotated with `@Bound`. This example uses field injection again but constructor and method injection are supported as well.

```
@Dependent
public class CustomerView {
  @Inject @AutoBound
  private DataBinder<Customer> customerBinder;

  @Inject @Bound
  private TextBox name;

  @Bound
  private Label id = new Label();

  ...
}
```

In both examples above an instance of the `Customer` model is automatically bound to the corresponding UI widgets based on matching field names. The model object and the UI fields will automatically be kept in sync. The widgets are inferred from all enclosing fields and methods annotated with `@Bound` of the class that defines the `@AutoBound DataBinder` or `@Model` and all its super classes.

9.5.1. Default, Simple, and Chained Property Bindings

By default, bindings are determined by matching field names to property names on the model object. In the examples above, the field `name` was automatically bound to the JavaBeans property `name` of the model (`user` object). If the field name does not match the model property name, you can use the `property` attribute of the `@Bound` annotation to specify the name of the property. The property can be a simple name (for example, "name") or a property chain (for example, `user.address.streetName`). When binding to a property chain, all properties but the last in the chain must refer to @Bindable values.

The following example illustrates all three scenarios:

```
@Bindable
public class Address {
  private String line1;
  private String line2;
  private String city;
  private String stateProv;
  private String country;

  // getters and setters
}

@Bindable
public class User {
  private String name;
  private String password;
  private Date dob;
  private Address address;
  private List<Role> roles;

  // getters and setters
}

@Templated
public class UserWidget {
  @Inject @AutoBound DataBinder<User> user;
  @Inject @Bound TextBox name;
  @Inject @Bound(property="dob") DatePicker dateOfBirth;
  @Inject @Bound(property="address.city") TextBox city;
}
```

In `UserWidget` above, the `name` text box is bound to `user.name` using the default name matching; the `dateOfBirth` date picker is bound to `user.dob` using a simple property name mapping; finally, the `city` text box is bound to `user.address.city` using a property chain. Note that the `Address` class is required to be `@Bindable` in this case.

9.5.2. Data Converters

The `@Bound` annotation further allows to specify a converter to use for the binding (see *Specifying Converters* for details). This is how a binding specific converter can be specified on a data field:

```
@Inject
@Bound(converter=MyDateConverter.class)
@DataField
```

```
private TextBox date;
```

9.5.3. Updating model values on UI text changes

By default, Errai DataBinding updates model values when the corresponding Widget fires a ValueChangeEvent. For text-based widgets, this means that the model values are updated only once the widget loses focus. However in some cases, you may also need to update a model's value as soon as the text changes in the Widget.

Errai allows you to do this by setting the `onKeyUp` flag in the `@Bound` annotation. **Note that this parameter can only be used on Widgets that extend GWT's ValueBoxBase widget**. Setting this parameter to `true` will cause the model value to update on a KeyUpEvent as well as the default ValueChangeEvent. This will result in the model object being updated as soon as any text is entered/removed from the Widget. You can specify this as follows:

```
@Inject
@Bound(onKeyUp = true)
private TextBox name;
```

You can also achieve the same effect using programmatic bindings. To do this, you can use the method `DataBinder.bind(Widget widget, String property, Converter converter, boolean bindOnKeyUp)`. As an example, take a look at the following code snippet:

```
@Inject
DataBinder<Model> binder;

@Inject
private TextBox nameTextBox;

...

@PostConstruct
public void onLoad() {
  binder.bind(nameTextBox, "name", converter, true);
}
```

Text Based Widgets

Binding on key events is only valid for text-based Widgets, i.e. those that extend ValueBoxBase. Errai DataBinding will throw an exception if the onKeyUp parameter is set on any non-ValueBoxBased Widgets.

9.5.4. Replacing a model object

The injected model objects in the examples above are always proxies to the actual model since method invocations on these objects need to trigger additional logic for updating the UI. Special care needs to be taken in case a model object should be replaced.

When working with an `@AutoBound DataBinder` , simply calling `setModel()` on the `DataBinder` will be enough to replace the underlying model instance. However, when working with `@Model` the instance cannot be replaced directly. Errai provides a special method level annotation `@ModelSetter` that will allow replacing the model instance. Here's an example:

```
@Dependent
public class CustomerView {
  @Inject @Model
  private Customer customer;

  @Inject @Bound
  private TextBox name;

  @Bound
  private Label id = new Label();

  @ModelSetter
  public void setModel(Customer customer) {
    this.customer = customer;
  }
}
```

The `@ModelSetter` method is required to have a single parameter. The parameter type needs to correspond to the type of the managed model.

9.6. Bean validation

Java bean validation (JSR 303) provides a declarative programming model for validating entities. More details and examples can be found *here* [http://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/] . Errai provides a bean validation module that makes `Validator` instances injectable and work well with Errai's data binding module. The following line needs to be added to the GWT module descriptor to inherit Errai's bean validation module:

Example 9.2. App.gwt.xml

```
<inherits name="org.jboss.errai.validation.Validation" />

<inherits name="org.hibernate.validator.HibernateValidator" />
```

To use Errai's bean validation module, you must add the module, the javax.validation API and an implementation such as hibernate validator to your classpath. If you are using Maven for your build, add these dependencies:

```
    <dependency>
      <groupId>org.jboss.errai</groupId>
      <artifactId>errai-validation</artifactId>
      <version>${errai.version}</version>
    </dependency>

    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <classifier>sources</classifier>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.2.0.Final</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.2.0.Final</version>
      <scope>provided</scope>
      <classifier>sources</classifier>
    </dependency>
```

Now it is as simple as injecting a `Validator` instance into an *Errai IOC* managed bean and calling the `validate` method.

```
@Inject
private Validator validator;
```

```
Set<ConstraintViolation<Customer>> violations  = validator.validate(customer);
// display violations
```

9.6.1. Excluding Classes from Validation

By default, Errai scans the entire classpath for classes with constraints. But sometimes it is necessary or desirable to exclude some shared classes from being validated on the client side. This can be done by adding a list of classes and package masks to the ErraiApp.properties file like so:

```
# The following blacklists the class some.fully.qualified.ClassName and all
 classes in some.package.mask (and subpackages thereof).
errai.validation.blacklist = some.fully.qualified.ClassName \
                             some.package.mask.*
```

Errai UI

One of the primary complaints of GWT to date has been that it is difficult to use "pure HTML" when building and skinning widgets. Inevitably one must turn to Java-based configuration in order to finish the job. Errai, however, strives to remove the need for Java styling. HTML template files are placed in the project source tree, and referenced from custom "Composite components" (Errai UI Widgets) in Java. Since Errai UI depends on Errai IOC and Errai CDI, dependency injection is supported in all custom components. Errai UI provides rapid prototyping and HTML5 templating for GWT.

10.1. Get started

The Errai UI module is directly integrated with *Data Binding* and Errai JPA but can also be used as a standalone project in any GWT client application by simply inheriting the Errai UI GWT module, and ensuring that you have properly using *Errai CDI's @Inject* to instantiate your widgets:

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai UI* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai UI to your project. If you work better by playing with a finished product, you can see a simple client-server project *implemented using Errai UI here* [https://github.com/errai/summit-demo-2013] .

10.2. Use Errai UI Composite components

Before explaining how to create Errai UI components, it should be noted that these components behave no differently from any other GWT Widget once built. The primary difference is in A) their construction, and B) their instantiation. As with most other features of Errai, dependency injection with CDI is the programming model of choice, so when interacting with components defined using Errai UI, you should always `@Inject` references to your Composite components.

10.2.1. Inject a single instance

```
@EntryPoint
public class Application {
   @Inject
```

```
   private ColorComponent comp;

   @PostConstruct
   public void init() {
      comp.setColor("blue");
      RootPanel.get().add(comp);
   }
}
```

10.2.2. Inject multiple instances (for iteration)

```
@EntryPoint
public class Application {
   private String[] colors = new String[]{"Blue", "Yellow", "Red"};

   @Inject
   private Instance<ColorComponent> instance;

   @PostConstruct
   public void init() {
      for(String color: colors) {
        ColorComponent comp = instance.get();
        comp.setColor(c);
        RootPanel.get().add();
      }
   }
}
```

10.3. Create a @Templated Composite component

Custom components in Errai UI are single classes extending from `com.google.gwt.user.client.ui.Composite` , and must be annotated with @Templated.

10.3.1. Basic component

```
@Templated
public class LoginForm extends Composite {
   /* looks for LoginForm.html in LoginForm's package */
}
```

10.3.2. Custom template names

With default values, @Templated informs Errai UI to look in the current package for a parallel "`.html`" template next to the Composite component Class; however, the template name may be overridden by passing a String into the @Templated annotation, like so:

```
@Templated("my-template.html")
public class LoginForm extends Composite {
   /* looks for my-template.html in LoginForm's package */
}
```

Fully qualified template paths are also supported, but must begin with a leading /:

```
@Templated("/org/example/my-template.html")
public class LoginForm extends Composite {
   /* looks for my-template.html in package org.example */
}
```

10.4. Create an HTML template

Templates in Errai UI may be designed either as an HTML snippet or as a full HTML document. You can even take an existing HTML page and use it as a template. With either approach, the `id` , `class` , and `data-field` attributes in the template identify elements by name. These elements and their children are used in the Composite component to add behavior, and use additional components to add functionality to the template. There is no limit to how many component classes may share a given HTML template.

We will begin by creating a simple HTML login form to accompany our `@Templated LoginForm` composite component.

```
<form>
  <legend>Log in to your account</legend>

  <label for="username">Username</label>
  <input id="username" type="text" placeholder="Username">

  <label for="password">Password</label>
  <input id="password" type="password" placeholder="Password">

  <button>Log in</button>
  <button>Cancel</button>
</form>
```

10.4.1. Select a template from a larger HTML file

Or as a full HTML document which may be more easily previewed during design without running the application; however, in this case we must also specify the location of our component's root DOM Element using a `"data-field"` , `id` , or `class` attribute matching the value of the @Templated annotation. There is no limit to how many component classes may share a given HTML template.

```
@Templated("my-template.html#login-form")
public class LoginForm extends Composite {
   /* Specifies that <... id="login-form"> be used as the root Element of this
 Widget  */
}
```

Notice the corresponding HTML `id` attribute in the form Element below (we could have used `data-field` or `class` instead). Note that multiple components may use the same template provided that they specify a corresponding `data-field` , `id` , or `class` attribute. Also note that two or more components may share the same DOM elements; there is no conflict since components each receive a unique copy of the template DOM rooted at the designated element at runtime (or from the root element if a fragment is not specified.)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>A full HTML snippet</title>
</head>
<body>
    <div>
        <form id="login-form">
          <legend>Log in to your account</legend>

          <label for="username">Username</label>
          <input id="username" type="text" placeholder="Username">

          <label for="username">Password</label>
          <input id="password" type="password" placeholder="Password">

          <button>Log in</button>
          <button>Cancel</button>
        </form>
    </div>

    <hr>
    <footer id="theme-footer">
        <p>(c) Company 2012</p>
```

```
    </footer>
</body>
</html>
```

For example's sake, the component below could also use the same template. All it needs to do is reference the template name, and specify a fragment.

```
@Templated("my-template.html#theme-footer")
public class Footer extends Composite {
    /* Specifies that <... id="theme-footer"> be used as the root Element of
 this Widget  */
}
```

10.5. Use other Widgets in a composite component

Now that we have created the @Templated Composite component and an HTML template, we can start wiring in functionality and behavior; this is done by annotating fields and methods to replace specific sub-elements of the template DOM with other Widgets. We can even replace portions of the template with other Errai UI Widgets!

10.5.1. Annotate Widgets in the template with @DataField

In order to composite Widgets into the template DOM, you annotate fields in your @Templated Composite component with @DataField, and mark the HTML template Element with a correspondingly named `data-field` , `id` , or `class` attribute. This informs Errai UI which element in the template the Widget should replace. All replacements happen while the @Templated Composite component is being constructed; thus, fields annotated with @DataField must either be {{@Inject}}ed or provide their own Widget or Element instances in field initializers.

```
@Templated
public class LoginForm extends Composite {
   // This element must be initialized manually because Element is not @Inject-
able*/
   @DataField
   private Element form = DOM.createForm();

  // If not otherwise specified, the name to match in the HTML template defaults
 to the name of the field; in this case, the name would be "username"
   @Inject
   @DataField
   private TextBox username;

   // The name to reference in the template can also be specified manually
   @Inject
```

```
   @DataField("pass")
   private PasswordTextBox password;

  // We can also choose to instantiate our own Widgets. Injection is not required.
   @DataField
   private Button submit = new Button();
}
```

Important

Note: Field, method, and constructor injection are all supported by @DataField.

10.5.2. Add corresponding attributes to the HTML template

Each @DataField reference in the Java class must match an element in the HTML template. The matching of Java references to HTML elements is performed as follows:

1. A *name* for the Java reference is determined. If the `@DataField` annotation has a value argument, that is used as the reference name. For fields, the default reference name is the field name. Method and constructor parameters have no default name, so they must always specify a value.
2. If there is an element in the HTML template with attribute `data-field=name` , the Java reference will point to this element. If there is more than one such element, the Java reference points to the first.
3. Otherwise, if there is an element in the HTML template with attribute `id=name` , the Java reference will point to this element. If there is more than one such element, the Java reference points to the first.
4. Otherwise, if there is an element in the HTML template with a CSS style class `name` , the Java reference will point to this element. If there is more than one such element, the Java reference points to the first. For elements with more than one CSS style, each style name is considered individually. For example:

```
<div class="eat drink be-merry">
```

matches Java references named `eat` , `drink` , or `be-merry` .

1. If no matching element is found by this point, it is an error.

If more than one Java reference matches the same HTML element in the template, it is an error. For example, given a template containing the element `<div class="eat drink be-merry">`, the following Java code is in error:

```
@Templated
public class ErroneousTemplate extends Composite {
 @Inject @DataField
 private Label eat;

 @Inject @DataField
 private Label drink;
}
```

because both fields `eat` and `drink` refer to the same HTML `div` element.

So now we must ensure there are `data-field`, `id`, or `class` attributes in the right places in our template HTML file. This, combined with the @DataField annotation in our Composite component allow Errai UI to determine where and what should be composited when creating component instances.

```
<form id="form">
  <legend>Log in to your account</legend>

  <label for="username">Username</label>
  <input id="username" type="text" placeholder="Username">

  <label for="password">Password</label>
 <input data-field="pass" id="password" type="password" placeholder="Password">

  <button id="submit">Log in</button>
  <button>Cancel</button>
</form>
```

Now, when we run our application, we will be able to interact with these fields in our Widget.

10.6. How HTML templates are merged with Components

Three things are merged or modified when Errai UI creates a new Composite component instance:

1. Element attributes are merged from the template to the component

2. DOM Elements are merged from the component to the template

3. Template element inner text and inner HTML are preserved when the given `@DataField Widget` implements `HasText` or `HasHTML`

10.6.1. Example

10.6.1.1. Composite component class:

```
@Templated
public class StyledComponent extends Composite {
   @Inject
   @DataField("field-1")
   private Label div = new Label();

   public StyledComponent() {
    div.getElement().setAttribute("style", "position: fixed; top: 0; left: 0;");
      this.getElement().setId("outer-id");
   }
}
```

10.6.1.2. Template:

```
<form>
                                                <span                        data-
field="field-1" style="display:inline;"> This element will become a div </span>
</form>

This text will be ignored.
```

10.6.1.3. Output / result:

```
<form id="outer-id">
                                                      <div                   data-
field="field-1" style="display:inline;"> This element will become a div </div>
</form>
```

But why does the output look the way it does? Some things happened that may be unsettling at first, but we find that once you understand why these things occur, you'll find the mechanisms extremely powerful.

10.6.2. Element attributes (template wins)

When styling your templates, you should keep in mind that all attributes defined in the template file will take precedence over any preset attributes in your Widgets. This "attribute merge" occurs

only when the components are instantiated; subsequent changes to any attributes after Widget construction will function normally. In the example we defined a Composite component that applied several styles to a child Widget in its constructor, but we can see from the output that the styles from the template have overridden them. If styles must be applied in Java, instead of the template, `@PostConstruct` or other methods should be favored over constructors to apply styles to fully-constructed Composite components.

10.6.3. DOM Elements (component field wins)

Element composition, however, functions inversely from attribute merging, and the `<span>` defined in our template was actually be replaced by the `<div>` Label in our Composite component field. This does not, however, change the behavior of the attribute merge - the new `<div>` was still be rendered inline, because we have specified this style in our template, and the template always wins in competition with attributes set programatically before composition occurs. In short, whatever is inside the `@DataField` in your class will replace the children of the corresponding element in your template.

10.6.4. Inner text and inner HTML (preserved when component implements HasText or HasHTML)

Additionally, because `Label` implements both `HasText` and `HasHTML` (only one is required,) the contents of this "field-1" Element in the template were preserved; however, this would not have been the case if the `@DataField` specified for the element did not implement `HasText` or `HasHTML` . In short, if you wish to preserve text or HTML contents of an element in your template, you can do one of two things: do not composite that Element with a `@DataField` reference, or ensure that the Widget being composited implements `HasText` or `HasHTML`.

10.7. Event handlers

Dealing with User and DOM Events is a reality in rich web development, and Errai UI provides several approaches for dealing with all types of browser events using its "quick handler" functionality. It is possible to handle:

1. GWT events on Widgets
2. GWT events on DOM Elements
3. Native DOM events on Elements

Important

It is not possible to handle Native DOM events on Widgets because GWT overrides native event handlers when Widgets are added to the DOM. You must programatically configure such handlers after the Widget has been added to the DOM.

10.7.1. Concepts

Each of the three scenarios mentioned above use the same basic programming model for event handling: Errai UI wires methods annotated with `@EventHandler("my-data-field")` (*event handler methods*) to handle events on the corresponding `@DataField("my-data-field")` in the same component. Event handler methods annotated with a bare `@EventHandler` annotation (no annotation parameter) are wired to receive events on the @Templated component itself.

10.7.2. GWT events on Widgets

Probably the simplest and most common use-case, this approach handles GWT Event classes for Widgets that explicitly handle the given event type. If a Widget does not handle the Event type given in the `@EventHandler` method's signature, the application will fail to compile and appropriate errors will be displayed.

```
@Templated
public class WidgetHandlerComponent extends Composite {

   @Inject
   @DataField("b1")
   private Button button;

   @EventHandler("b1")
   public void doSomethingC1(ClickEvent e) {
     // do something
   }
}
```

10.7.3. GWT events on DOM Elements

Errai UI also makes it possible to handle GWT events on native Elements which are specified as a `@DataField` in the component class. This is useful when a full GWT Widget is not available for a given Element, or for GWT events that might not normally be available on a given Element type. This could occur, for instance, when clicking on a `<div>` , which would normally not have the ability to receive the GWT `ClickEvent` , and would otherwise require creating a custom DIV Widget to handle such an event.

```
@Templated
public class ElementHandlerComponent extends Composite {

   @DataField("div-1")
   private DivElement button = DOM.createDiv();

   @EventHandler("div-1")
   public void doSomethingC1(ClickEvent e) {
```

```
      // do something
    }
}
```

10.7.4. Native DOM events on Elements

The last approach is handles the case where native DOM events must be handled, but no such GWT event handler exists for the given event type. Alternatively, it can also be used for situations where Elements in the template should receive events, but no handle to the Element the component class is necessary (aside from the event handling itself.) Native DOM events do not require a corresponding `@DataField` be configured in the class; only the HTML `data-field` , `id` , or `class` template attribute is required.

```
<div>
   <a id="link" href="/page">this is a hyperlink</a>
   <div data-field="div"> Some content </div>
</div>
```

The `@SinkNative` annotation specifies (as a bit mask) which native events the method should handle; this sink behaves the same in Errai UI as it would with `DOM.sinkEvents(Element e, int bits)`. Note that a `@DataField` reference in the component class is optional.

Important

Only one @EventHandler may be specified for a given template element when @SinkNative is used to handle native DOM events.

```
@Templated
public class QuickHandlerComponent extends Composite {

  @DataField
  private AnchorElement link = DOM.createAnchor().cast();

  @EventHandler("link")
  @SinkNative(Event.ONCLICK | Event.ONMOUSEOVER)
  public void doSomething(Event e) {
    // do something
  }

  @EventHandler("div")
  @SinkNative(Event.ONMOUSEOVER)
  public void doSomethingElse(Event e) {
    // do something else
```

```
  }
}
```

10.8. HTML Form Support

Using asynchronous Javascript calls often make realizing the benefits of modern browsers difficult when it comes to form submission. But there is now a base class in Errai UI for creating `@Templated` form widgets that are perfect for tasks such as creating a login form.

10.8.1. A Login Form that Triggers Browsers' "Remember Password" Feature

Here is a sample `@Templated` login form class. This form has:

- a `username` text field
- a `password` field
- a button that with a click handler that attempts to login asynchronously

```
@Dependent
@Templated
public class LoginForm extends AbstractForm { (1)

  @Inject
  private Caller<AuthenticationService> authenticationServiceCaller;

  @Inject
  @DataField
  private TextBox username;

  @Inject
  @DataField
  private PasswordTextBox password;

  @DataField
  private final FormElement form = DOM.createForm(); (2)

  @Inject
  @DataField
  private Button login; (3)

  @Override
  protected FormElement getFormElement() {
```

```
    return form; (4)
  }

  @EventHandler("login")
  private void loginClicked(ClickEvent event) {
    authenticationServiceCaller.call(new RemoteCallback<User>() {

      @Override
      public void callback(User response) {
        // Now that we're logged in, submit the form
        submit(); (5)
      }
    }).login(username.getText(), password.getText());
  }

}
```

The key things that you should take from this example:

1. The class extends `org.jboss.errai.ui.client.widget.AbstractForm`.
2. The `form` field is a `@DataField` but it is not injected.
3. The login button is a regular button widget, with a click handling method below.
4. The `getFormElement` method inherited from `AbstractForm` must return the `FormElement` that will be submitted.
5. After the user has successfully logged in asynchronously we call `submit()`. This causes form submission to happen in a way that will not cause the page to refresh, but will still properly notify the browser of a form submission.

When a user successfully logs in via this example, the web browser should prompt them to remember the username and password (assuming this is a feature of the browser being used).

10.8.2. Using the Correct Elements in the Template

The most likely way to go wrong is to accidentally use the wrong types of elements in your template. It is very important that you use a proper `from` element with `input` elements **with the exception of the submit button**. Here is an html template that could accompany the `LoginForm.java` example above:

```
<div>
  <form data-field="form">
    <input type="text" name="username" data-field="username">
    <input type="password" name="password" data-field="password">
    <button data-field="login">Sign In</button>
  </form>
</div>
```

To reiterate, notice that the `username` and `password` fields are legitimate `input` elements. This is because we want these values to be submitted when `AbstractForm.submit()` is called (so that the browser notices them). However, we do not want there to be any way to submit the form other than calling `AbstractForm.submit()`, so the `button` element is notably missing the `type="submit"` attribute pair.

10.9. Data Binding

A recurring implementation task in rich web development is writing event handler code for updating model objects to reflect input field changes in the user interface. The requirement to update user interface fields in response to changed model values is just as common. These tasks require a significant amount of boilerplate code which can be alleviated by Errai. Errai's *data binding module* provides the ability to bind model objects to user interface fields, so they will automatically be kept in sync. While the module can be used on its own, it can cut even more boilerplate when used together with Errai UI.

In the following example, all `@DataFields` annotated with `@Bound` have their contents bound to properties of the data model (a `User` object). The model object is injected and annotated with `@Model`, which indicates automatic binding should be carried out. Alternatively, the model object could be provided by an injected `DataBinder` instance annotated with `@AutoBound` , see *Declarative Binding* for details.

```
@Templated
public class LoginForm extends Composite {

   @Inject
   @Model
   private User user;

   @Inject
   @Bound
   @DataField
   private TextBox name;

   @Inject
   @Bound
   @DataField
   private PasswordTextBox password;

   @DataField
   private Button submit = new Button();
}
```

Now the user object and the `username` and `password` fields in the UI are automatically kept in sync. No event handling code needs to be written to update the user object in response to input field changes and no code needs to be written to update

the UI fields when the model object changes. So, with the above annotations in place, it will always be true that `user.getUsername().equals(username.getText())` and `user.getPassword().equals(password.getText())`.

10.9.1. Default, Simple, and Chained Property Bindings

By default, bindings are determined by matching field names to property names on the model object. In the example above, the field `name` was automatically bound to the JavaBeans property `name` of the model (`user` object). If the field name does not match the model property name, you can use the `property` attribute of the `@Bound` annotation to specify the name of the property. The property can be a simple name (for example, "name") or a property chain (for example, `user.address.streetName`). When binding to a property chain, all properties but the last in the chain must refer to @Bindable values.

The following example illustrates all three scenarios:

```
@Bindable
public class Address {
  private String line1;
  private String line2;
  private String city;
  private String stateProv;
  private String country;

  // getters and setters
}

@Bindable
public class User {
  private String name;
  private String password;
  private Date dob;
  private Address address;
  private List<Role> roles;

  // getters and setters
}

@Templated
public class UserWidget extends Composite {
  @Inject @AutoBound DataBinder<User> user;
  @Inject @Bound TextBox name;
  @Inject @Bound(property="dob") DatePicker dateOfBirth;
  @Inject @Bound(property="address.city") TextBox city;
}
```

In `UserWidget` above, the `name` text box is bound to `user.name` using the default name matching; the `dateOfBirth` date picker is bound to `user.dob` using a simple property name mapping; finally, the `city` text box is bound to `user.address.city` using a property chain. Note that the `Address` class is required to be `@Bindable` in this case.

10.9.2. Binding of Lists

Often you will need to bind a list of model objects so that every object in the list is bound to a corresponding widget. This task can be accomplished using Errai UI's `ListWidget` class. Here's an example of binding a list of users using the `UserWidget` class from the previous example. First, we need to enhance `UserWidget` to implement `HasModel` .

```
@Templated
public class UserWidget extends Composite implements HasModel<User> {
  @Inject @AutoBound DataBinder<User> userBinder;
  @Inject @Bound TextBox name;
  @Inject @Bound(property="dob") DatePicker dateOfBirth;
  @Inject @Bound(property="address.city") TextBox city;

  public User getModel() {
    userBinder.getModel();
  }

  public void setModel(User user) {
    userBinder.setModel(user);
  }
}
```

Now we can use `UserWidget` to display items in a list.

```
@Templated
public class MyComposite extends Composite {

  @Inject @DataField ListWidget<User, UserWidget> userListWidget;

  @PostConstruct
  public void init() {
    List<User> users = .....
    userListWidget.setItems(users);
  }
}
```

Calling `setItems` on the `userListWidget` causes an instance of `UserWidget` to be displayed for each user in the list. The `UserWidget` is then bound to the corresponding user object. By default, the widgets are arranged in a vertical panel. However, `ListWidget` can also be subclassed to

provide alternative behaviour. In the following example, we use a horizontal panel to display the widgets.

```
public class UserListWidget extends ListWidget<User, UserWidget> {

  public UserList() {
    super(new HorizontalPanel());
  }

  @PostConstruct
  public void init() {
    List<User> users = .....
    setItems(users);
  }

  @Override
  public Class<UserWidget> getItemWidgetType() {
    return UserWidget.class;
  }
}
```

10.9.2.1. Binding lists with @Bound

An instance of `ListWidget` can also participate in automatic bindings using `@Bound` . In this case, `setItems` never needs to be called manually. The bound list property and displayed items will automatically be kept in sync. In the example below a list of user roles is bound to a `ListWidget` that displays and manages a `RoleWidget` for each role in the list. Every change to the list returned by `user.getRoles()` will now trigger a corresponding update in the UI.

```
@Templated
public class UserDetailView extends Composite {

   @Inject
   @Bound
   @DataField
   private TextBox name;

   @Inject
   @Bound
   @DataField
   private PasswordTextBox password;

   @Inject
   @Bound
   @DataField
   private ListWidget<Role, RoleWidget> roles;
```

```
   @DataField
   private Button submit = new Button();

   @Inject @Model
   private User user;
}
```

10.9.3. Data Converters

The `@Bound` annotation further allows to specify a converter to use for the binding (see *Specifying Converters* for details). This is how a binding specific converter can be specified on a data field:

```
@Inject
@Bound(converter=MyDateConverter.class)
@DataField
private TextBox date;
```

Errai's `DataBinder` also allows to register `PropertyChangeHandlers` for the cases where keeping the model and UI in sync is not enough and additional logic needs to be executed (see *Property Change Handlers* for details).

10.10. Nest Composite components

Using Composite components to build up a hierarchy of widgets functions exactly the same as when building hierarchies of GWT widgets. The only distinction might be that with Errai UI, `@Inject` is preferred to manual instantiation.

```
@Templated
public class ComponentOne extends Composite {

   @Inject
   @DataField("other-comp")
   private ComponentTwo two;
}
```

10.11. Extend Composite components

Templating would not be complete without the ability to inherit from parent templates, and Errai UI also makes this possible using simple Java inheritance. The only additional requirement is that Composite components extending from a parent Composite component must also be annotated with @Templated, and the path to the template file must also be specified in the child component's annotation. Child components may specify `@DataField` references that were omitted in the parent

class, and they may also override `@DataField` references (by using the same `data-field` name) that were already specified in the parent component.

10.11.1. Template

Extension templating is particularly useful for creating reusable page layouts with some shared content (navigation menus, side-bars, footers, etc.) where certain sections will be filled with unique content for each page that extends from the base template; this is commonly seen when combined with the MVP design pattern traditionally used in GWT applications.

```
<div class="container">
   <div id="header"> Default header </div>
   <div id="content"> Default content </div>
   <div id="footer"> Default footer </div>
</div>
```

10.11.2. Parent component

This component provides the common features of our page layout, including header and footer, but does not specify any content. The missing @DataField "content" will be provided by the individual page components extending from this parent component.

```
@Templated
public class PageLayout extends Composite {

   @Inject
   @DataField
   private HeaderComponent header;

   @Inject
   @DataField
   private FooterComponent footer;

   @PostConstruct
   public final void init() {
      // do some setup
   }
}
```

10.11.3. Child component

We are free to fill in the missing "content" @DataField with a Widget of our choosing. Note that it is not required to fill in all omitted @DataField references.

```
@Templated("PageLayout.html")
public class LoginLayout extends PageLayout {

   @Inject
   @DataField
   private LoginForm content;

}
```

We could also have chosen to override one or more `@DataField` references defined in the parent component, simply by specifying a `@DataField` with the same name in the child component, as is done with the "footer" data field below.

```
@Templated("PageLayout.html")
public class LoginLayout extends PageLayout {

   @Inject
   @DataField
   private LoginForm content;

   /* Override footer defined in PageLayout */
   @Inject
   @DataField
   private CustomFooter footer;

}
```

10.12. Stylesheet binding

When developing moderately-complex web applications with Errai, you may find yourself needing to do quite a bit of programmatic style changes. One common case is showing or enabling controls only if a user has the necessary permissions to use them. One part of the problem is securing those features from being used, and the other part which is an important usability consideration is communicating that state to the user.

RestrictedAccess in Errai Security

Errai Security contains a `RestrictedAccess` annotation that uses style sheet binding to implement a feature similar in nature to this example.

Let's start with the example case I just described. We have a control that we only want to be visible if the user is an admin. So the first thing we do is create a style binding annotation.

```
@StyleBinding
@Retention(RetentionPolicy.RUNTIME)
public @interface Admin {
}
```

This defines `Admin` as a stylebinding now we can use it like this:

```
@EntryPoint
@Templated
public class HelloWorldForm extends Composite {
  @Inject @Admin @DataField Button deleteButton;
  @Inject SessionManager sessionManager;


  @EventHandler("deleteButton")
  private void handleSendClick(ClickEvent event) {
    // do some deleting!
  }

  @Admin
  private void applyAdminStyling(Style style) {
    if (!sessionManager.isAdmin()) {
      style.setVisibility(Style.Visibility.HIDDEN);
    }
  }
}
```

Now before the form is shown to the user the `applyAdminStyling` method will be executed where the `sessionManager` is queried to see if the user is an admin if not the delete button that is also annotated with `@Admin` will be hidden from the view.

The above example took at `Style` object as a parameter, but it is also possible to use an `Element`. So the `applyAdminStyling` method above could have also been written like this:

```
  @Admin
  private void applyAdminStyling(Element element) {
    if (!sessionManager.isAdmin()) {
      element.addClassName("disabled");
    }
  }
```

The CSS class "disabled" could apply the same style as before ("visibility: hidden") or it could have more complex behaviour that is dependent on the element type.

10.12.1. Usage with Data Binding

In addition when using this in conjunction with Errai Databinding. Any Errai UI component which uses @AutoBound, will get live updating of the style rules for free, anytime the model changes. Allowing dynamic styling based on user input and other state changes.

10.13. Internationalization (i18n)

User interfaces often need to be available in different languages. Errai's i18n support makes it easier for you to publish your web app in multiple languages. This section explains how to use this feature in your application.

10.13.1. HTML Template Translation

To get started with Errai's internationalization support, simply put the `@Bundle("bundle.json")` annotation on your entry point and add an empty `bundle.json` file to your classpath (e.g. to src/main/java or src/main/resources). Of course, you can name it differently.

Errai will scan your HTML templates and process all text elements to generate key/value pairs for translation. It will generate a file called `errai-bundle-all.json` and put it in your `.errai` directory. You can copy this generated file and use it as a starting point for your custom translation bundles. If the text value is longer than 128 characters the key will get cut off and a hash appended at the end.

The translation bundle files use the same naming scheme as Java (e.g. `bundle_nl_BE.json` for Belgian Dutch and `bundle_nl.json` for plain Dutch). Errai will also generate a file called `errai-bundle-missing.json` in the `.errai` folder containing all template values for which no translations have been defined. You can copy the key/value pairs out of this file to create our own translations:

```
{
"StoresPage.Stores!" : "Stores!",
"WelcomePage.As_you_move_toward_a_more_and_more_declarative_style,_you_allow_the_compiler_and_t
 you move toward a more and more declarative style, you allow the compiler and the
 framework to catch more mistakes up front. Broken links? A thing of the past!"
}
```

If you want to use your own keys instead of these generated ones you can specify them in your templates using the `data-i18n-key` attribute:

```
<html>
<body>
  <div id="content">
  <p data-i18n-key="welcome">Welcome to errai-ui i18n.</p>
<div>
```

```
...
```

By adding this attribute in the template you can translate it with the following:

```
{
    "Widget.welcome": "Willkommen bei Errai-ui i18n."
}
```

Because your templates are designer templates and can contain some mock data that doesn't need to be translated, Errai has the ability to indicate that with an attribute `data-role=dummy` :

```
<div id=navbar data-role=dummy>
  <div class="navbar navbar-fixed-top">
    <div class=navbar-inner>
      <div class=container>
        <span class=brand>Example Navbar</span>
        <ul class=nav>
          <li><a>Item</a>
          <li><a>Item</a>
        </ul>
      </div>
    </div>
  </div>
</div>
```

Here the template fills out a navbar with dummy elements, useful for creating a design, adding `data-role=dummy` will not only exclude it form being translated it will also strip the children nodes from the template that will be used by the application.

When you have setup a translation of your application Errai will look at the browser locale and select the locale, if it's available, if not it will use the default (`bundle.json`). If the users of your application need to be able to switch the language manually, Errai offers a pre build component you can easily add to your page: `LocaleListBox` will render a Listbox with all available languages. If you want more control of what this language selector looks like there is also a `LocaleSelector` that you can use to query and select the locale for example:

```
@Templated
public class NavBar extends Composite {

  @Inject
  private LocaleSelector selector;

  @Inject @DataField @OrderedList
```

```
  ListWidget<Locale, LanguageItem> language;

  @AfterInitialization
  public void buildLanguageList() {
    language.setItems(new ArrayList<Locale>(selector.getSupportedLocales()));
  }

...
// in LanguageItem we add a click handler on a link

  @Inject
  Navigation navigation;

  @Inject
  private LocaleSelector selector;

  link.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        selector.select(model.getLocale());
        navigation.goTo(navigation.getCurrentPage().name());
      }
    });
```

10.13.2. TranslationKey and TranslationService

The `@TranslationKey` annotation and `TranslationService` class extend Errai's i18n support to Java code. They provide a mechanism for developers to declare translation strings from within their GWT application code (as opposed to the HTML templates).

To do this, developers must annotate a field which represents the translation key with `@TranslationKey` annotation. This key will then map to a value in the translation bundle file. Once the field is annotated appropriately, the developer must directly invoke the TranslationService's format() method. This method call will perform a lookup in the translation service of the value mapped to the provided key. Note that value substitution using the `{N}` format is supported.

As an example, consider the following code:

```
package org.example.ui.client.local;

public class AppMessages {

   @TranslationKey(defaultValue = "I guess something happened!")
   public static final String CUSTOM_MESSAGE = "app.custom-message";

  @TranslationKey(defaultValue = "Hey {0}, I just told you something happened!")
```

```
    public static final String CUSTOM_MESSAGE_WITH_NAME = "app.custom-message-
with-name";
}
```

```
package org.example.ui.client.local;

@Dependent
@Templated
public class CustomComponent extends Composite {

   @Inject
   private TranslationService translationService;

   @Inject
   @DataField
   private Button someAction;

   @EventHandler("someAction")
   private void doLogin(ClickEvent event) {

      // do some action that may require a notification sent to the user

      String messageToUser = translationService.format(AppMessages.CUSTOM_MESSAGE);
      Window.alert(messageToUser);

      String username = getCurrentUserName();
      String messageToUserWithName = translationService.format(AppMessages.CUSTOM_MESSAGE_WITH_
      Window.alert(messageToUserWithName);
   }
}
```

10.14. Extended styling with LESS

Errai also supports *LESS* [http://lesscss.org] stylesheets. To get started using these you'll have to create a LESS stylesheet and place it on the classpath of your project and declare their ordering with the `StyleDescriptor` annotation. Every application should have 0 or 1 classes annotated with `StyleDescriptor` like the following example:

```
package org.jboss.errai.example;

@StyleDescriptor({ "/main.less", "other.css" })
public class MyStyleDescriptor {
}
```

The two files listed above, `main.less` and `other.css`, will be compiled into a single stylesheet by Errai. The relative path for `other.css` will be loaded relative to the package `org.jboss.errai.example`.

Do Not Declare Imported Stylesheets

It is only necessary to declare top-level stylesheets with the `StyleDescriptor`. If a CSS or LESS resource is only meant to be imported by another LESS stylesheet, then it need only be on the classpath.

Errai will convert the LESS stylesheet to css, perform optimisations on it, and ensure that is get injected into the pages of your application. It will also obfuscate the class selectors and replace the use of those in your templates. To be able to use the selectors in your code you can use:

```
public class MyComponent extends Component {
  @Inject
  private LessStyle lessStyle;

  ...

  @PostConstruct
  private void init() {
    textBox.setStyleName(lessStyle.get("input"));
  }
}
```

Finally it will also add any deferred binding properties to the top of your LESS stylesheet, so for example you could use the user.agent in LESS like this:

```
.mixin (@a) when (@a = "safari") {
  background-color: black;
}

.mixin (@a) when (@a = "gecko1_8") {
  background-color: white;
}

.class1 { .mixin(@user_agent) }
```

Because a dot is not allowed in LESS variables it's replaced with an underscore, so in the example above class1 will have a black background on Safari and Chrome and white on Firefox. On the top of this LESS stylesheet @user_agent: "safari" will get generated.

Errai UI Navigation

Starting in version 2.1, Errai offers a system for creating applications that have multiple bookmarkable pages. This navigation system has the following features:

- Declarative, statically-analyzable configuration of pages and links
- Compile time referential safety (i.e. “no broken links”)
- Generates a storyboard of the application’s navigation flow at compile time
- Decentralized configuration
- Create a new page by creating a new annotated class. No need to edit a second file.
- Make navigational changes in the natural place in the code
- Integrates cleanly with Errai UI templates, but also works well with other view technologies
- Builds on Errai IoC & CDI
- Provides support for HTML5 pushState and path-parameter based URLs

11.1. Getting Started

Plugin Tip

Use the *Errai* *Forge* *Addon* *Add* *Errai* *Features* command and select *Errai Navigation* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai Navigation to your project.

11.2. How it Works

Errai Navigation has these main parts:

- The `@Page` annotation marks any widget as a page.
- The `TransitionTo<P>`, `TransitionAnchor<P>`, and `TransitionToRole<R>` classes are injectable types that provide links to other pages.
- The `Navigation` singleton offers control over the navigation system as a whole.

The `Navigation` singleton owns a GWT Panel called the *navigation panel* . This panel always contains a widget corresponding to the fragment ID (the part after the # symbol) in the browser's location bar. Whenever the fragment ID changes for any reason (for example, because the user pressed the back button, navigated to a bookmarked URL, or simply typed a fragment ID by hand), the widget in the navigation panel is replaced by the widget associated with that fragment ID. Likewise, when the application asks the navigation system to follow a link, the fragment ID in the browser's location bar is updated to reflect the new current page.

11.2.1. Declaring a Page

To declare a page, annotate any subclass of Widget with the `@Page` annotation:

```
@Page
public class ItemListPage extends Composite {
  // Anything goes...
}
```

By default, the name of a page is the simple name of the class that declares it. In the above example, the `ItemListPage` will fill the navigation panel whenever the browser's location bar ends with `#ItemListPage`. If you prefer a different page name, use the `@Page` annotation's `path` attribute:

```
@Page(path="items")
public class ItemListPage extends Composite {
  // Anything goes...
}
```

Navigation and Errai UI

Any widget can be a page. This includes Errai UI `@Templated` classes! Simply annotate any Errai UI templated class with `@Page`, and it will become a page that can be navigated to.

11.2.1.1. The Default (Starting) Page

Each application must have exactly one *default page*. This requirement is enforced at compile time. This default page is displayed when there is no fragment ID present in the browser's location bar.

Use the `role = DefaultPage.class` attribute to declare the default starting page, like this:

```
@Page(role = DefaultPage.class)
```

```
public class WelcomePage extends Composite {
  // Anything goes...
}
```

Pages are looked up as CDI beans, so you can inject other CDI beans into fields or a constructor. Pages can also have `@PostConstruct` and `@PreDestroy` CDI methods.

11.2.1.2. Page Roles

`DefaultPage` is just one example of a page role. A page role is simply an interface used to mark `@Page` types. The main uses for page roles:

- Using the `Navigation` singleton, you can look up all pages that have a specific role.
- If a role is unique (as is the case with `DefaultPage`) then it should extend `UniquePageRole`, making it possible to navigate to the page by its role.

11.2.2. Page Lifecycle

There are four annotations related to page lifecycle events: `@PageShowing`, `@PageShown`, `@PageHiding`, and `@PageHidden`. These annotations designate methods so a page widget can be notified when it is displayed or hidden:

```
@Page
public class ItemPage extends VerticalPanel {

  @PageShowing
  private void preparePage() {
  }

  @PageHiding
  private void unpreparePage() {
  }

  // Anything goes...
}
```

11.2.2.1. Lifecycle Phases

1. The fragment identifier in the URL changes
2. The `@PageHiding` method on the current (about-to-be-navigated-away-from) page is invoked
3. The current page is removed from the browser's DOM

4. The `@PageHidden` method on the just-removed page is invoked
5. The navigation system looks up the corresponding `@Page` bean in the client-side bean manager (we'll call this bean "the new page")
6. The navigation system writes to all `@PageState` fields in the new page bean (more on this in the next section)
7. The `@PageShowing` method of the new page is invoked
8. The new page widget is added to the DOM (as a direct child of the navigation content panel)
9. The `@PageShown` method of the new page is invoked.

11.2.2.2. Optional Parameters

The `@PageShowing` and `@PageShown` methods are permitted one optional parameter of type `HistoryToken` — more on this in the next section.

The `@PageHiding` method is also permitted one optional parameter of type `NavigationControl`. If the parameter is present, the page navigation will not be carried out until `NavigationControl.proceed()` is invoked. This is useful for interrupting page navigations and then resuming at a later time (for example, to prompt the user to save their work before transitioning to a new page).

11.2.2.3. Page Instance Lifespan

The lifespan of a Page instance is governed by CDI scope: Dependent and implict-scoped page beans are instantiated each time the user navigates to them, whereas Singleton and ApplicationScoped beans are created only once over the lifetime of the application. If a particular page is slow to appear because its UI takes a lot of effort to build, try marking it as a singleton.

11.2.3. Page State Parameters

A page widget will often represent a view on on instance of a class of things. For example, there might be an ItemPage that displays a particular item available at a store. In cases like this, it's important that the bookmarkable navigation URL includes not only the name of the page but also an identifier for the particular item being displayed.

This is where page state parameters come in. Consider the following page widget:

```
@Page
public class ItemPage extends VerticalPanel {

  @PageState
  private int itemId;

  // Anything goes...
```

```
}
```

This page would be reachable at a URL like `http://www.company.com/store/#ItemPage;itemId=4`, assuming `www.company.com` was the host address and `store` was the application context. Before the page was displayed, the Errai UI Navigation framework would write the `int` value `4` into the `itemId` field.

Page state parameters can also be accessed using URLs with path parameters. In this case, you have to declare the template of the page's path in the `path` field of the `@Page` annotation. As an example, consider the following code:

```
@Page(path="item/{itemID}/{customerID}")
public class ItemPage extends VerticalPanel {

  @PageState
  private int itemID;

  @PageState
  private String customerID;

  // Anything goes...
}
```

There are three ways to pass state information to a page: by passing a Multimap to `TransitionTo.go()` ; by passing a Multimap to `Navigation.goTo()` , or by including the state information in the path parameter or fragment identifier of a hyperlink as illustrated in the previous paragraph (use the `HistoryToken` class to construct such a URL properly.)

A page widget can have any number of `@PageState` fields. The fields can be of any primitive or boxed primitive type (except `char` or `Character`), `String`, or a `Collection`, `List`, or `Set` of the allowable scalar types. Nested collections are not supported.

`@PageState` fields can be private, protected, default access, or public. They are always updated by direct field access; never via a setter method. The updates occur just before the `@PageShowing` method is invoked.

In addition to receiving page state information via direct writes to `@PageState` fields, you can also receive the whole Multimap in the `@PageShowing` and `@PageShown` methods through a parameter of type `HistoryToken` . Whether or not a lifecycle method has such a parameter, the `@PageState` fields will still be written as usual.

Page state values are represented in the URL in place of the corresponding parameter variables declared in the URL template (the `path` field of the `@Page` annotation. See *Declaring a Page*). If a parameter variable is declared in the URL template and is missing from the actual typed URL, it will cause a navigation error as Errai will not be able to match the typed URL to any template.

Any additional path parameters not found in the URL template are appended as key=value pairs separated by the ampersand (`&`) character. Multi-valued page state fields are represented by repeated occurrences of the same key. If a key corresponding to a `@PageState` field is absent from the state information passed to the page, the framework writes a default value: `null` for scalar Object fields, the JVM default (0 or false) for primitives, and an empty collection for collection-valued fields. To construct and parse state tokens programmatically, use the `HistoryToken` class.

To illustrate this further, consider the following example:

```
@Page(path="item/{itemID}/{customerID}")
public class ItemPage extends VerticalPanel {

  @PageState
  private int itemID;

  @PageState
  private String customerID;

  @PageState
  private int storeID;

  // Anything goes...
}
```

Given the host `"www.company.com"`, the context `store`, and a state map with the values `itemID=4231` , `customerID=9364` , and `storeID=0032` , the following URL will be generated: `www.company.com/store/#item/4231/9364;storeID=0032`

If the value for storeID is undefined, the URL will be `www.company.com/store/#item/4231/9364;storeID=0` .

If the URL typed into the browser is `www.company.com/store/#item/4231;storeID=0032`, it will cause a navigation error (assuming there is no other page by this url) because there is a missing path parameter.

11.2.4. PushState Functionality

Errai now comes with support for pushState and path-parameter-based URLs. If HTML5 pushState is enabled Errai Navigation urls will not use the fragment-identifier (#). Thus the non-pushState url from the previous section, `www.company.com/store/#item/4231/9364`, would become `www.company.com/store/item/4231/9364`.

HTML5 pushState can be enabled by adding the following lines to your GWT host page:

```
<script type="text/javascript">
    var erraiPushStateEnabled = true;
```

```
</script>
```

The application context must be the same as the application's servlet web context deployed on the server. Errai attempts to infer the application context upon the first page load, but it can also be set manually. To explicitly declare the application context, you can use the `setApplicationContext` method in the Navigation class, or set the `erraiApplicationWebContext` variable in your GWT host page as follows:

```
<script type="text/javascript">
    var erraiApplicationWebContext = "store";
</script>
```

In the event that the browser does not support HTML5, Errai automatically disables pushState functionality and reverts to a #-based URL format. That is, Errai uses fragment identifiers to refer to particular resources.

If the page that the user is trying to navigate to cannot be found, a 404 - Not Found page is displayed. You can override this functionality and display a custom page in the case of a page not found error. For example, to navigate to the GWT host page by default, add the following lines to your web.xml file:

```
<error-page>
    <error-code>404</error-code>
    <location>/</location>
</error-page>
```

11.2.5. Declaring a Link with TransitionAnchor

The easiest way to declare a link between pages is to inject an instance of `TransitionAnchor<P>`, where `P` is the class of the target page.

Here is an example declaring an anchor link from the templated welcome page to the item list page. The first code sample would go in WelcomePage.java while the second would go in the WelcomePage.html, the associated html template.

```
@Page(role = DefaultPage.class)
@Templated
public class WelcomePage extends Composite {

  @Inject @DataField TransitionAnchor<ItemListPage> itemLink;

}
```

```
<div>
  <a data-field="itemLink">Go to Item List Page</a>
</div>
```

You can inject any number of links into a page. The only restriction is that the target of the link must be a Widget type that is annotated with `@Page` . When the user clicks the link Errai will transition to the item list page.

11.2.6. Declaring a Manual Link

Sometimes it is necessary to manually transition between pages (such as in response to an event being fired). To declare a manual link from one page to another, inject an instance of `TransitionTo<P>` , where `P` is the class of the target page.

This code declares a manual transition from the welcome page to the item list page:

```
@Page(role = DefaultPage.class)
public class WelcomePage extends Composite {

  @Inject TransitionTo<ItemListPage> startButtonClicked;

}
```

You do not need to implement the `TransitionTo` interface yourself; the framework creates the appropriate instance for you.

As with `TransitionAnchor`, the only restriction is that the target of the link must be a Widget type that is annotated with `@Page` .

11.2.7. Following a Manual Link

To follow a manual link, simply call the `go()` method on an injected `TransitionTo` object. For example:

```
@Page(role = DefaultPage.class)
public class WelcomePage extends Composite {

  @Inject TransitionTo<ItemListPage> startButtonClicked;

  public void onStartButtonPressed(ClickEvent e) {
    startButtonClicked.go();
  }
}
```

11.2.8. Declaring a Link By UniquePageRole

For convenience, it is also possible to transition to a page by its role using an injected `TransitionToRole<R>` where `R` is an interface extending `UniquePageRole`. This type is used exactly as the `TransitionTo`: just inject a parameterized instance and invoke the `go()` method.

By injecting a `TransitionToRole` into a `@Page`, Errai will verify the existence of a single page with this role at compile-time.

11.2.9. Installing the Navigation Panel into the User Interface

Beginning in version 2.4, Errai will automatically attach the Navigation Panel to the Root Panel, but it is possible to override this behaviour by simply adding the Navigation Panel to another component manually. The best time to do this is during application startup, for example in the `@PostConstruct` method of your `@EntryPoint` class. By using the default behaviour you can allow Errai Navigation to control the full contents of the page, or you can opt to keep some parts of the page (headers, footers, and sidebars, for example) away from Errai Navigation by choosing an alternate location for the Navigation Panel.

The following example reserves space for header and footer content that is not affected by the navigation system:

```
@EntryPoint
public class Bootstrap {

  @Inject
  private Navigation navigation;

  @PostConstruct
  public void clientMain() {
    VerticalPanel vp = new VerticalPanel();
    vp.add(new HeaderWidget());
    vp.add(navigation.getContentPanel());
    vp.add(new FooterWidget());

    RootPanel.get().add(vp);
  }
}
```

This last example demonstrates a simple approach to defining the page structure with an Errai UI template. The final product is identical to the above example, but in this case the overall page structure is declared in an HTML template rather than being defined programmatically in procedural logic:

```
@Templated
```

```
@EntryPoint
public class OverallPageStructure extends Composite {

  @Inject
  private Navigation navigation;

  @Inject @DataField
  private HeaderWidget header;

  @Inject @DataField
  private SimplePanel content;

  @Inject @DataField
  private FooterWidget footer;

  @PostConstruct
  public void clientMain() {

    // give over the contents of this.content to the navigation panel
    content.add(navigation.getContentPanel());

    // add this whole templated widget to the root panel
    RootPanel.get().add(this);
  }

}
```

11.2.10. Overriding the default Navigating Panel type

By default Errai uses `com.google.gwt.user.client.ui.SimplePanel` as a container for navigation panel. Sometimes this is not sufficient and users would prefer using another implementation. For example a `com.google.gwt.user.client.ui.SimpleLayoutPanel` that manages child size state.

To provide your own implementation of the navigation panel you must implement `org.jboss.errai.ui.nav.client.local.NavigatingContainer`. For example:

```
public class NavigatingPanel implements NavigatingContainer {

  SimplePanel panel = new SimpleLayoutPanel();

  public void clear() {
    this.panel.clear();
  }

  public Widget asWidget() {
    return panel.asWidget();
```

```
  }

  public Widget getWidget() {
    return panel.getWidget();
  }

  public void setWidget(Widget childWidget) {
    panel.add(childWidget);
  }

  public void setWidget(IsWidget childWidget) {
    panel.add(childWidget);
  }

}
```

Then in your GWT module descriptor you need to override the default navigation panel (`org.jboss.errai.ui.nav.client.local.NavigatingContainer`) by adding:

```
<replace-with class="com.company.application.client.NavigatingPanel">
                                                                    <when-type-
is class="org.jboss.errai.ui.nav.client.local.NavigatingContainer"/>
</replace-with>
```

11.2.11. Handling Navigation Errors

When a user enters a url for an Errai page that does not exist an error is logged and the app navigates to the `DefaultPage`. It is possible to override this behaviour by setting an error handler on `Navigation`.

Here is an example of a class that registers a navigation error handler that redirects the user to a special `PageNotFound` page:

```
@ApplicationScoped
public class NavigationErrorHandlerSetter {

  @Inject
  private Navigation navigation;

  @PostConstruct
  public void setErrorHandler() {
    navigation.setErrorHandler(new PageNavigationErrorHandler() {

      @Override
      public void handleError(Exception exception, String pageName) {
```

```
        navigation.goTo("PageNotFound");
      }

      @Override
      public void handleError(Exception exception, Class<? extends PageRole> pageRole) {
        navigation.goTo("PageNotFound");
      }
    });
  }

}
```

1. Note that this method signature is for errors that occur from calls to `Navigation.goToWithRole(Class<? extends UniquePageRole>)`. These kinds of errors can be avoided at compile-time by injecting `TransitionToRole` instances into your `@Page` classes instead of directly calling that method.

11.2.12. Viewing the Generated Navigation Graph

Because the pages and links in an Errai Navigation application are declared structurally, the framework gets a complete picture of the app's navigation structure at compile time. This knowledge is saved out during compilation (and at page reload when in Dev Mode) to the file `.errai/navgraph.gv` . You can view the navigation graph using any tool that understands the GraphViz (also known as DOT) file format.

One popular open source tool that can display GraphViz/DOT files is *GraphViz* [http://www.graphviz.org/] . Free downloads are available for all major operating systems.

When rendered, a navigation graph looks like this:

Figure 11.1. Navigation Graph

In the rendered graph, the pages are nodes (text surrounded by an ellipse). The starting page is drawn with a heavier stroke. The links are drawn as arrows from one page to another. The labels on these arrows come from the Java field names the TransitionTo objects were injected into.

Errai Cordova (Mobile Support)

Starting with version 2.4.0, Errai now supports mobile development. One of the modules that makes this feasible is the Cordova module. It offers a way to integrate with native hardware in an Errai way.

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai Cordova* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai Cordova to your project.

12.1. Integrate with native hardware

When the Cordova module is included you can integrate with native hardware by injecting the native components into your code:

```
@Templated("#main")
public class KitchenSinkClient extends Composite {
  @Inject
  Camera camera;

  @Inject
  @DataField
  Button takePicture;

  @EventHandler("takePicture")
  public void onTakePicktureClicked(ClickEvent event) {
    PictureOptions options = new PictureOptions(25);
    options.setDestinationType(PictureOptions.DESTINATION_TYPE_DATA_URL);
    options.setSourceType(PictureOptions.PICTURE_SOURCE_TYPE_CAMERA);

    camera.getPicture(options, new PictureCallback() {

      @Override
      public void onSuccess(String data) {
     image.setUrl(UriUtils.fromSafeConstant("" + data));
      }
```

```
      @Override
      public void onFailure(String error) {
        setGeneralErrorMessage("Could not take picture: " + error);
      }
    });
  }
```

The components that are supported come from the *gwt-phonegap* [https://code.google.com/p/gwt-phonegap/] project have a look there form more documentation.

Here are the native hardware components you can inject:

- Camera
- Accelerometer
- Contacts
- Capture (Provides access to the audio, image, and video capture capabilities of the device).
- Compass
- Notification (*see documentation on phonegap site* [http://docs.phonegap.com/en/edge/cordova_notification_notification.md.html#Notification])
- File create a native file
- Device Get general information about the device.

So to integrate with these things all we have to do is `@Inject` these classes. There are also a couple of CDI events one can observe to be informed about hardware state:

- BackButtonEvent
- BatteryCriticalEvent
- BatteryEvent
- BatteryLowEvent
- BatteryStatusEvent
- EndCallButtonEvent
- MenuButtonEvent
- OffLineEvent
- OnlineEvent

- PauseEvent
- ResumeEvent
- SearchButtonEvent
- StartCallButtonEvent
- VolumeDownButtonEvent
- VolumeUpButtonEvent

Example of how to use these events:

```
  private void batteryIsLow(@Observes BatteryLowEvent event) {
    //mission accomplished. we can stop the infinite loop now.
  }
```

Errai Security

Errai Security provides a lightweight security API for declaring RPC services and client-side UI elements which require authentication or authorization.

Plugin Tip

Use the *Errai Forge Addon* *Add Errai Features* command and select *Errai Security* to follow along with this section.

Manual Setup

Checkout the *Manual Setup Section* for instructions on how to manually add Errai Security to your project.

13.1. Basic Model

Errai Security provides two main concepts:

- **Users**
  - A User corresponds to a single person.
  - It is usually associated with a username, full name, and email address.
- **Roles**
  - A Role represents a priveleged group within your system.
  - A User can have several roles, and a role can be had by many users.
  - Roles are the primary way of defining authorization in Errai Security.

By default the server-side Errai Security module uses *PicketLink* [http://www.picketlink.org/] for authentication. Later on we will explain how to use an alternative backend.

13.2. Getting Started

13.2.1. Making Users

The simplest way to begin experimenting with Errai Security is to add Users and Roles to PicketLink programmatically. Here is some sample server-side code from the *Errai Security Demo* [https://github.com/errai/errai/blob/master/errai-demos/errai-security-demo/src/main/java/org/jboss/errai/security/demo/server/PicketLinkDefaultUsers.java].

```
@Singleton
@Startup
public class PicketLinkDefaultUsers {

  @Inject
  private PartitionManager partitionManager; (1)

  /**
   * <p>Loads some users during the first construction.</p>
   */
  @PostConstruct
  public void create() {
    final IdentityManager identityManager = partitionManager.createIdentityManager();
    final RelationshipManager relationshipManager = partitionManager.createRelationshipManager(

    User john = new User("john");

    john.setEmail("john@doe.com");
    john.setFirstName("John");
    john.setLastName("Doe");

    User hacker = new User("hacker");

    hacker.setEmail("hacker@illegal.ru");
    hacker.setFirstName("Hacker");
    hacker.setLastName("anonymous");

    identityManager.add(john); (2)
    identityManager.add(hacker);
    final Password defaultPassword = new Password("123");
    identityManager.updateCredential(john, defaultPassword);
    identityManager.updateCredential(hacker, defaultPassword);

    Role roleDeveloper = new Role("simple");
    Role roleAdmin = new Role("admin");

    identityManager.add(roleDeveloper);
    identityManager.add(roleAdmin);

    relationshipManager.add(new Grant(john, roleDeveloper)); (3)
    relationshipManager.add(new Grant(john, roleAdmin));
  }

}
```

Here are the important things that are happening here:

1. PicketLink uses the concept of partitions, which are sections that can contain different users and roles. What we really need to make users and roles are the `IdentityManager` and `RelationshipManager`, but these objects are `@RequestScoped` so in order to access them when the application starts we must `@Inject` the `PartitionManager`.
2. Here we add are new users to the `IdentityManager`. It is also used below to give passwords to the new users, and to add the *simple* and *admin* roles.
3. The `RelationshipManager` defines relationships between entities. In this case, it is used to specify that a user belongs to a role.

13.2.2. Authentication from the Client

Once you've created some users and roles, you're ready to write some client-side code. Authentication is performed with the `org.jboss.errai.security.shared.service.AuthenticationService` via Errai RPC.

Here is some sample code involving the user *john* from the previous Security Demo excerpt.

- Injecting the `Caller<AuthenticationService>`:

```
@Inject Caller<AuthenticationService> authServiceCaller;
```

- Logging in:

```
authServiceCaller.call(new RemoteCallback<User>() {

    @Override
    public void callback(User user) {
      // handle successful login
    }
  }, new ErrorCallback<Message>() {

    @Override
    public boolean error(Message message, Throwable t) {
      if (t instanceof AuthenticationException) {
        // handle authentication failure
      }

      // Returning true causes the error to propogate to top-level handlers
      return true;
    }
  }).login("john", "123");
```

- Getting the currently authenticated User:

```
authServiceCaller.call(new RemoteCallback<User>() {
```

```
    @Override
    public void callback(User user) {
      if (!user.equals(User.ANONYMOUS)) {
        // Do something because we're logged in.
      }
      else {
        // Do something else because we're not logged in.
      }
    }
  }).getUser();
```

- Logging out:

```
authServiceCaller.call().logout();
```

AuthenticationService Caching

Client-side interceptors are used for caching so that generally only calls to `login` and `logout` must be sent over the wire. The cache is automatically invalidated when a service throws an `UnauthenticatedException`, but it can also be invalidated manually via the `SecurityContext`.

13.3. RestrictedAccess

The annotation `@RestrictedAccess` is the only annotation necessary to secure a resource or UI element. In general, `@RestrictedAccess` blocks a resource from users who are either not logged in or who lack required roles. Roles are defined through the `@RestrictedAccess` annotation in one of the following two ways.

13.3.1. Simple Roles as Strings

Simple roles are roles that can be directly mapped to Strings (the String value being the role name). Simple roles are defined by defining an array of Strings in the `roles` parameter of `@RestrictedAccess`. Two simple roles are equivalent if they have the same name.

Here is an example usage of `@RestrictedAccess` with two simple roles, "user" and "admin":

```
@RestrictedAccess(roles = { "user", "admin" })
```

13.3.2. Provided Roles

Conceptually, a provided can be used to implement a more complex security system. In practice, a provided role is some concrete type that implements the `Role` interface and overrides `Object.equals(Object)`. Provided roles are declared on a resource by creating a `RequiredRolesProvider` that produces these roles and assigning the type to the `providers` parameter of `@RestrictedAccess`.

Here is an sample of a `RequireRolesProvider` and its usage with `@RestrictedAccess`. This example defines equivalent roles to the above example using simple roles.

```
@Dependent (1)
public class AdminRolesProvider implements RequiredRolesProvider {

  @Override
  public Set<Role> getRoles() {
    return new HashSet<Role>(Arrays.asList(
      new RoleImpl("user"), (2)
      new RoleImpl("admin")
    ));
  }
}
```

```
@RestrictedAccess(providers = { AdminRolesProvider.class })
```

(1) The role provider implementation must be a CDI bean so that it can be looked up dynamically on the client and server.

(2) `RoleImpl` is the internal implementation used for simple roles. A `RoleImpl` equals another role if they are both instances of `RoleImpl` and have matching names.

13.3.3. RPC Services

To secure an Errai RPC service, simply annotate the RPC interface (either the entire type or just a method) with one of the security annotations.

For example:

- All methods on this interface require an authenticated user to access:

```
@Remote
@RestrictedAccess
public interface UserOnlyStuff {
  public void someMethod();
  public void otherMethod();
```

```
}
```

- Here the first method requires an authenticated user, and the second requires a user with the *admin* role:

```
@Remote
public interface MixedService {

  @RestrictedAccess
  public void userService();

  @RestrictedAccess(roles = {"admin"})
  public void adminService();
}
```

Using Role Providers with RPC Services

If a `RequiredRolesProvider` is used on an RPC interface, the provider type must be located in a shared package. Security checks for RPCs are performed on the client and the server, so placing the type in a client- or server-only package will result in run-time errors.

13.3.3.1. Error Callbacks

When access to a secured RPC service is denied an `UnauthenticatedException` or `UnauthorizedException` is thrown. This error is then transmitted back to the client, where it can be caught with an `ErrorCallback` (provided when the RPC is invoked).

Here is how we would invoke the previous `MixedService` example with error handling:

```
MessageBuilder.createCall(new RemoteCallback<Void>() {

    @Override
    public void callback(Void response) {
      // ...
    }
  }, new ErrorCallback<Message>() {

    @Override
    public boolean error(Message message, Throwable t) {
      if (t instanceof UnauthenticatedException) {
        // User is not logged in.
        return false;
      }
```

```
      else if (t instanceof UnauthorizedException) {
        // User is logged in but lacked sufficient roles.
        return false;
      }
      else {
        // Some other error has happened. Let it propogate.
        return true;
      }
    }
  }, MixedService.class).adminService();
```

1. This `ErrorCallback` is parameterized with the type `Message` because it is an Errai Bus RPC. In the next section we will demonstrate the use of a JAX-RS RPC.

DefaultBusSecurityErrorCallback

Errai Security provides a default global Bus RPC handler that catches any thrown `UnauthenticatedException` or `UnauthorizedException` and navigates to the page with the `LoginPage` or `SecurityError` role respectively.

13.3.3.2. JAX-RS RPC

JAX-RS RPCs are secured exactly as bus RPCs. Here is the first example from the previous section, but converted to use JAX-RS instead of the Errai Bus.

```
@Path("/rest-endpoint")
@RestrictedAccess
public interface UserOnlyStuff {

  @Path("/some-method")
  @GET
  public void someMethod();

  @Path("/other-method")
  @GET
  public void otherMethod();
}
```

There are two important differences when calling a secured JAX-RS RPC (in contrast to an Errai Bus RPC):

- JAX-RS RPC calls use the `RestErrorCallback` (an interface extending `ErrorCallback<Request>`).
- There is now global error-handling for JAX-RS.

Because there is no global error-handling, you should always pass a `RestErrorCallback` when using a JAX-RS RPC. Errai provides the `DefaultRestSecurityErrorCallback` that provides the same default behaviour as the `DefaultBusSecurityErrorCallback` mentioned above. It can also optionally wrap a provided callback as demonstrated below:

- Injecting a callback `Instance`:

```
@Inject
private Instance<DefaultRestSecurityErrorCallback> defaultCallbackInstance;
```

- Wrapping a custom callback in a default callback:

```
void callSomeService() {
  userOnlyStuffService.call(new RemoteCallback<Void>() {

    @Override
    public void callback(Void response) {
      // Handle success...
    }
  }, defaultCallbackInstance.get()
        .setWrappedErrorCallback(new RestErrorCallback() {

          @Override
          public boolean error(Request request, Throwable t) {
            // Handle error...

           // Returning true means the default navigation behaviour will occur
            return true;
          }
        }
  )).someMethod();
}
```

- Using the default callback without a wrapped callback:

```
void callSomeService() {
  userOnlyStuffService.call(new RemoteCallback<Void>() {

    @Override
    public void callback(Void response) {
      // Handle success...
    }
  }, defaultCallbackInstance.get()).someMethod();
}
```

13.3.4. Page Navigation

Any class annotated with `@Page` can also be marked with `@RestrictedAccess`. By doing so, users will be prevented from navigating to the given page if they are not logged in or lack authorization.

Here are two simple examples:

- This page is only for logged in users:

```
@Page
@RestrictedAccess
public class UserProfilePage extends SimplePanel {

  @Inject private Caller<AuthenticationService> authServiceCaller;
  private User user;

  @PageShowing
  private void setupPage() {
    authServiceCaller.call(new RemoteCallback<User>() {
        @Override
        public void callback(User response) {
           // We don't have to check if this is a valid user, since the page
 requires authentication.
          user = response;
          // do setup...
        }
    }).getUser();
  }

}
```

- This page requires the *user* and *admin* roles:

```
@Page
@RestrictedAccess(roles = {"admin", "user"})
public class AdminManagementPage extends SimplePanel {
}
```

Redirection

When a user is denied access to a page they will be redirected to a `LoginPage` (`@Page(role = LoginPage.class)`) or `SecurityError` (`@Page(role = SecurityError.class)`) page. To direct a user to the page they were trying

to reach after successful login, `@Inject` the `SecurityContext` and invoke the `navigateBackOrHome` method.

13.3.4.1. Page Redirection and Caching

Security checks performed before page navigation do not use any RPC calls, but are instead performed from a cached (in-memory) instance of the `org.jboss.errai.security.shared.api.identity.User`. This prevents the possibility of lengthy delays between page navigation while waiting for RPC return values.

But the drawback is that any attempts to navigate to a secured `@Page` before the cache is populated will result in redirection to the `LoginPage` — even if the user is in fact logged in.

In practice, this is only likely to happen if a user starts an Errai app with a URL to a secure page while still logged in on the server from a previous session.

One option offered by Errai is to persist the `org.jboss.errai.security.shared.api.identity.User` object in a cookie. This can be done by adding the following to `ErraiApp.properties`:

```
errai.security.user_cookie_enabled=true
```

With this option enabled the `User` will be persisted in a browser cookie, which is loaded quickly enough to avoid the described navigation issue. This feature can also be used to allow an application to work offline, or allow the server to log in a user on an initial page request.

User is stored in plain text

The `errai.security.user_cookie_enabled=true` setting causes the `User` to be stored in **plain text**. That includes the following information:

- The user's login name.
- The user's full name.
- The user's email address.
- The user's security roles.

If you do not wish to use this feature you will likely want to handle this case in the `@PageShowing` method of your `LoginPage`. Here is an outline of what you might want to do:

```
@Page(role = LoginPage.class)
@Templated
public class ExampleLoginPage extends Composite {
```

```
  @Inject
  private SecurityContext securityContext;

  @Inject
  private Caller<AuthenticationService> authService;

  @Inject
  @DataField
  private Label status;

  @PageShowing
  public void checkForPendingCache() {
    // Check if cache is invalid.
    if (!securityContext.isUserCacheValid()) {
      // Update the status.
      status.setText("loading...");

      // Force cache to update by calling getUser
      authService.call(new RemoteCallback<User> {
        @Override
        public void callback(User user) {
          /* An interceptor will have updated the cache by now.
             So check if we are logged in and redirect if necessary.
          */
          if (!user.equals(User.ANONYMOUS)) {
            /* This is a special transition that takes us back to
               a secure page from which we were redirected. */
            securityContext.navigateBackOrHome();
          }
          else {
            status.setText("You are not logged in.");
          }
        }
      }).getUser();
    }
  }

}
```

13.3.5. Hiding UI Elements

Errai Security annotations can also be used to hide Errai UI template fields. When a user is not logged in or lacks required roles the annotated field will have the CSS class "errai-restricted-access-style" added to it. By defining this style (for example with `visibility: none`) you can hide or otherwise modify the display of the element for unautorized users.

Here is an example of an Errai UI templated class using this feature:

```
@Templated
public class NavBar extends Composite {

  @Inject
  @DataField
  @RestrictedAccess
  private Button logoutButton;

  @Inject
  @DataField
  @RestrictedAccess(roles = {"admin"})
  private Button dropAllTablesButton;

}
```

13.4. Form Based Login

If you do enable the Errai Security cookie, it is possible to use a form-based login from outside your GWT/Errai app. The `errai-security-server` jar contains a servlet filter for encoding the currently authenticated user as a cookie in the http response. Here are the steps for setting this up:

1. Create a login page using an html form that posts to a servlet-filter. If you are using Errai Security with PicketLink you will want to use the `org.picketlink.authentication.web.AuthenticationFilter` servlet-filter. Otherwise, you will need to implement one yourself that authenticates the user by calling `AuthenticationService.login(String, String)` method.

2. Add this filter-mapping for setting the Errai Security user cookie:

```
<filter-mapping>
  <filter-name>ErraiUserCookieFilter</filter-name>
  <url-mapping>/gwt-host-page.html</url-mapping>
</filter-mapping>
```

   The mapped URL should be that of your GWT Host Page.

This mapping must come after the filter that authenticates the user

If this filter maps to the same URL as the filter for authentication, this filter must come after the authentication filter or else it will set the cookie before the user has logged in.

3. Make sure this is in your ErraiApp.properties file: `errai.security.user_cookie_enabled=true`

13.5. Using an Alternative to PicketLink

All Errai Security authentication is implemented with Errai Remote Procedure Calls to the `AuthenticationService`. A default implementation of this interface using PicketLink is provided in the `errai-security-picketlink` jar. But it is possible to use a different sever-side security framework by providing your own custom implementation of `AuthenticationService` and annotating it with `@Service`. In that case your project should not depend on `errai-security-picketlink`.

13.6. Using Keycloak for Authentication

Keycloak [http://keycloak.jboss.org/] is is a new project that provides integrated SSO and IDM for browser apps and RESTful web services. By using Keycloak it is possible to outsource the responsibility of authentication and account management from your application entirely. Errai Security provides an optional `errai-security-keycloak` jar that provides an implementation of the `AuthenticationService` that works with Keycloak.

13.6.1. How It Works (Overview)

From the perspective of a visitor, here is what happens when she attempts to log in:

- The visitor is redirected to a Keycloak login page.
- The visitor submits her credentials through the Keycloak login page.
- Assuming the credentials are valid, the visitor is redirected back to the web app, where she is now logged in.

13.6.1.1. Keycloak Token

Behind the scenes, when the visitor successfully submits credentials she is redirected back to the web app with a Keycloak Access Token, which contains information that is configurable from within Keycloak. A servlet filter is used to extract the token from the request and assign it to the `AuthenticationService` implementation. At this point the User is now logged in to your application.

13.6.2. Setup

Make sure to check out the Errai Security Demo.

This demo can be configured to work with Keycloak in just a few simple steps as outlined in the *README file* [https://github.com/errai/errai/tree/master/errai-demos/errai-security-demo]!

To start from scratch and add Keycloak integration to your application:

- Setup a Keycloak server. Please consult the *Keycloak documentation* [http://keycloak.jboss.org/docs.html] for details on how this is achieved.
- Start the Keycloak server.
- Go to the *Keycloak Administrative Console* [https://github.com/errai/errai/tree/master/errai-demos/errai-security-demo] (i.e. *http://localhost:8080/auth/admin/)* (the username and password are both **admin** on first use).
- Click **Add Realm** and create a custom realm for your application.
- Select the **Clients** tab and click **Create**, then fill in the following to add the client application to this realm:
  - `Client ID`: the name of your client application (i.e. errai-security-demo)
  - `Access Type`: public
  - `Redirect URI`: the url of your application (i.e. `http://localhost:8080/[your-application]/*`)
- After saving your application, choose the new application in the menu and make sure the following are set:
  - In the **Roles** tab add your custom roles.
  - In the **Installation** tab choose the format option `keycloak.json` and copy the contents in your `WEB-INF/keycloak.json` file.
- Click on **Users** on the side-panel to add a user:
  - Fill out the `Username`, `Email`, `First Name`, and `Last Name` with any values.
  - After saving go to the **Credentials** tab and set a password.
  - Go to the **Role Mappings** tab. Add `at least one` role to the `Assigned Roles` for your application (scroll down to `Application Roles` and select your application to do this).
- Add the `errai-security-keycloak` jar to your project and make sure it's being deployed to the server. In maven, the dependency is `org.jboss.errai:errai-security-keycloak`.
- Configure the `ErraiUserCookieFilter` in your `web.xml`. All that is necessary is adding a filter-mapping for your GWT host page like so:

```
  <filter-mapping>
    <filter-name>ErraiUserCookieFilter</filter-name>
    <url-pattern>/index.html</url-pattern>
```

```
  </filter-mapping>
```

- Configure the `ErraiLoginRedirectFilter` in your `web.xml`.
  - Create a filter-mapping of this filter onto a path that will act as a url to the Keycloak login page. For example, if your deployed app is called `my-app` and you wanted `<server-uri>/my-app/app-login` as your login url then you would add the following:

```
<filter-mapping>
  <filter-name>ErraiLoginRedirectFilter</filter-name>
  <url-pattern>/app-login</url-pattern>
</filter-mapping>
```

  - Add a security-constraint to login url. This is what actually causes the redirection to Keycloak. All the filter does is redirect back to your app (which happens after the login completes). For the previous example, the constraint would look like this:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Login</web-resource-name>
    <url-pattern>/app-login</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

  - Optionally configure the URL that the `ErraiLoginRedirectFilter` redirects to. You can do this with the `redirectLocation` param, which takes a path relative to the app context:

```
<filter>
  <filter-name>ErraiLoginRedirectFilter</filter-name>
  <init-param>
    <param-name>redirectLocation</param-name>
    <param-value>/index.jsp</param-value>
  </init-param>
</filter>
```

- Set the login method to use Keycloak in you `web.xml`:

```
<login-config>
  <auth-method>KEYCLOAK</auth-method>
  <realm-name>[your-realm-name]</realm-name>
```

```
  </login-config>
```

- Add roles available to users in your application to the `web.xml`. Here is an example declaration of a "user" role:

```
<security-role>
  <role-name>user</role-name>
</security-role>
```

All users must have at least one role

With this configuration all users must have at least a single role, or else they will not be redirected propertly. Unfortunately, there is no way to define a security-constraint that only requires authentication. The simplest solution is to add a default role to your realm.

Logging

Errai now supports using the *slf4j* [http://www.slf4j.org/] logging api on the server and client. This gives you the flexibility of choosing your own logging back-end for your server-side code, while still allowing a uniform logging interface that can be used in shared packages.

14.1. What is slf4j?

sl4j is logging abstraction. Using the slf4j api, you can add log statements to your code using a fixed api while maintaining the ability to switch the logging implementation at run-time. For example, the slf4j api can be used with java.util.logging (JUL) as the back-end.

14.2. Client-Side Setup

The client-side slf4j code uses the *GWT Logging* [http://www.gwtproject.org/doc/latest/DevGuideLogging.html] as the back-end. Using slf4j in client-side code has three steps:

- Add the errai-common artifact as a maven dependency to your project
- Inherit the gwt module `org.jboss.errai.common.ErraiCommon`

```
<inherits name="org.jboss.errai.common.ErraiCommon" />
```

- Enable logging and configure the log level in your gwt.xml module descriptor:

```
<set-property name="gwt.logging.enabled" value="TRUE"/>
<set-property name="gwt.logging.logLevel" value="ALL"/>
```

14.2.1. Errai Client-Side Log Handlers

In the ErraiCommon module, we have disabled the built-in GWT log handlers and provided four handlers of our own:

- *ErraiSystemLogHandler* : prints log statements to the terminal in Development Mode
- *ErraiConsoleLogHandler* : prints statements to the web console in the browser
- *ErraiDevelopmentModeLogHandler* : prints statements in the Development Mode window
- *ErraiFirebugLogHandler* : prints statements to the console in Firefox These loggers are all enabled by default and set to handle all log levels.

14.2.2. Configuring Errai Client-Side Log Handlers

Log handler levels can be changed at run-time through Java or Javascript. To do so through Java, use the `LoggingHandlerConfigurator` in Errai Common. Here's an example:

Example 14.1. HandlerLevelAdjuster.java

```
import org.jboss.errai.common.client.logging.LoggingHandlerConfigurator;
import org.jboss.errai.common.client.logging.handlers.ErraiSystemLogHandler;
import java.util.logging.Level;

public class HandlerLevelAdjuster {

  public static void logAll() {
    LoggingHandlerConfigurator config = LoggingHandlerConfigurator.get();
    ErraiSystemLogHandler handler = config.getHandler(ErraiSystemLogHandler.class);
    handler.setLevel(Level.ALL);
  }

  public static void disableLogging() {
    LoggingHandlerConfigurator config = LoggingHandlerConfigurator.get();
    ErraiSystemLogHandler handler = config.getHandler(ErraiSystemLogHandler.class);
    handler.setLevel(Level.OFF);
  }

}
```

Each handler has a native Javascript variable associated with its log level:

Handler	Variable Name
ErraiSystemLogHandler	erraiSystemLogHandlerLevel
ErraiConsoleLogHandler	erraiConsoleLogHandlerLevel
ErraiDevelopmentModeLogHandler	erraiDevelopmentModeLogHandlerLevel
ErraiFirebugLogHandler	erraiFirebugLogHandlerLevel

Since these are native Javascript variables, they can easily be set in a script tag on your host page:

```
<script type="text/javascript">
  erraiSystemLoghandlerLevel = "INFO";
</script>
```

The possible log levels correspond to those in `java.util.logging.Level`.

Logging Levels

If you are increasing the logging level of an Errai log handler, you will also need to increase the `gwt.logging.logLevel` (set in your *.gwt.xml). Handlers will not receive log records that are lower than the GWT log level, which is set to **INFO** in `ErraiCommon.gwt.xml`.

14.2.3. Format String

The Errai log handlers use `ErraiSimpleFormatter` to format log output. The format string is similar to that used in by `java.util.SimpleFormatter` (for precise differences please see the javadocs for `ErraiSimpleFormatter` and `StringFormat`).

As with handler settings, these can be configured in Java or Javascript. To do so in Java, use `ErraiSimpleFormmater.setSimpleFormatString(String)`. In Javascript, just set the variable `erraiSimpleFormatString` to the desired value.

14.3. Server-Side Setup

On the server you are free to use any logging back-end that has slf4j bindings (or to make your own). Just make sure to add dependencies for the slf4j-api artifact and the slf4j binding you choose. *Note:* Some application servers provide their own slf4j bindings (such as JBoss AS), in which case you should add your binding dependency as provided scope.

To learn more about how to setup slf4j for your server-side code, see *their website* [http://www.slf4j.org/].

14.4. Example Usage

Here is sample usage of the slf4j code (which with the above setup can be run on the client or server):

Example 14.2. LogExample.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import javax.inject.Inject;

public class LogExample {

  public void logStuff() {
    // Get a logger for this class
    @Inject Logger logger;

    // Logging going from most to least detailed
```

```
    logger.trace("this is extremely specific!");
    logger.debug("this is still pretty specific");
    logger.info("this is an average log message");
    logger.warn("there might be something fishy here...");
      logger.error("uh oh... abandon ship!", new Exception("I am a logged
 exception"));
  }
}
```

14.5. Logger Names

By default, the above example with provide a logger with the fully qualified class name of the enclosing class. To inject a logger with an alternate name, use the `NamedLogger` annotation:

Example 14.3. NamedLogExample.java

```
import org.slf4j.Logger;
import javax.inject.Inject;
import org.jboss.errai.common.client.api.NamedLogger;

public class NamedLogExample {

  // Get a logger with the name "Logger!"
  @Inject @NamedLogger("Logger!") logger;

  // Get the root logger
  @Inject @NamedLogger rootLogger;

}
```

Configuration

This section contains information on manually setting up Errai and describes additional configurations and settings which may be adjusted.

15.1. Errai Development Mode Configuration

15.1.1. Deployment in Development Mode (JBossLauncher)

In development mode we need to bootstrap the CDI environment on our own and make both Errai and CDI available through JNDI (common denominator across all runtimes). GWT by default uses Jetty, that only supports read only JNDI. The current solution for this is to use a custom launcher to control a JBoss AS 7 or Wildfly 8 instance instead of GWT's built-in Jetty.

To do this, requires the following configurations in the gwt-maven-plugin configuration:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>${gwt.version}</version>

  <configuration>
  ...
                          <extraJvmArgs>-Derrai.jboss.home=$JBOSS_HOME       -
Derrai.jboss.javaagent.path=${settings.localRepository}/org/jboss/errai/errai-
client-local-class-hider/$ERRAI_VERSION/errai-client-local-class-hider-
$ERRAI_VERSION.jar</extraJvmArgs>
    <noServer>false</noServer>
    <server>org.jboss.errai.cdi.server.gwt.JBossLauncher</server>
  </configuration>
  <executions>
  ...
  </executions>
</plugin>
```

What does all this mean?

- `<noServer>false</noServer>`: Tells GWT to lauch a server for us.
- `<server>org.jboss.errai.cdi.server.gwt.JBossLauncher</server>`: Tells GWT to use a custom launcher instead of it's default JettyLauncher.
- `<extraJvmArgs>...</extraJvmArgs>`

- `-Derrai.jboss.home=$JBOSS_HOME`: Tells the JBossLauncher the location of the JBoss or Wildfly instance to use. Note that `$JBOSS_HOME` should be replaced with a literal path (or pom property) to a JBoss or Wildfly instance you have installed.
- `-Derrai.jboss.javaagent.path=${settings.localRepository}/org/jboss/errai/errai-client-local-class-hider/${ERRAI_VERSION}/errai-client-local-class-hider-${ERRAI_VERSION}.jar`: This scary looking line is necessary so that the JBoss instance does not see client-only dependencies. Note that `$ERRAI_VERSION` should be replaced with the literal version of Errai (or a pom property).

15.1.2. Additional JBossLauncher Arguments

Here are some additional JVM arguments that can be passed to the JBossLauncher:

- `errai.dev.context`: Sets the context under which your app will be deployed (defaults to "webapp").
- `errai.jboss.debug.port`: Sets the port for debugging server-side code (defaults to 8001).
- `errai.jboss.config.file`: Sets the configuration file (in `$JBOSS_HOME/configuration`) used by the JBoss/Wildfly instance (defaults to standalone-full.xml).
- `errai.jboss.javaopts`: Sets additional java opts used by the JVM running JBoss/Wildfly.

15.1.3. Deployment to an Application Server

We provide integration with the *JBoss Application Server* [http://jboss.org/jbossas], but the requirements are basically the same for other vendors. When running a GWT client app that leverages CDI beans on a Java EE 6 application server, CDI is already part of the container and accessible through JNDI (`java:/BeanManager`).

15.2. Errai Offline Mode Configuration

Errai provides special support for HTML5's application caching mechanism that enables applications to work offline. If you're not familiar with the HTML5 application cache you can find all the details *here* [https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache].

As GWT compiles separate browser-specific JavaScript permutations for your application, it is not enough to manually create a cache manifest file and simply list all generated JavaScript files. This would cause every browser to download and cache JavaScript files that it doesn't need in the first place (i.e. Safari would download and cache JavaScript files that were generated for Internet Explorer only). Errai solves this problem by using a custom linker to generate user-agent specific cache manifest files.

The following steps are necessary to activate this linker:

- Define the linker in your gwt.xml module descriptor:

```
<define-
linker name="offline" class="org.jboss.errai.offline.linker.DefaultCacheManifestLinker"
>
<add-linker name="offline" />
```

- Add the manifest (`your_module_name`/errai.appcache) to the `html` tag in your host page:

```
<html manifest="your_module_name/errai.appcache">
```

- Add a mime-mapping to your web.xml file (you can skip this step if you deploy the errai-javaee-all.jar as part of your application):

```
<mime-mapping>
  <extension>manifest</extension>
  <mime-type>text/cache-manifest</mime-type>
</mime-mapping>
```

- Make sure the `errai-common.jar` file is deployed as part of your application. It contains a servlet that will provide the correct user-agent specific manifest file in response to requests to your_module_name/errai.appcache
- To obtain manifests that contain other files in addition to those generated by the `DefaultCacheManifestLinker`, create a subclass that overrides `otherCachedFiles()`, and use this subclass as a linker instead:

```
@Shardable
@LinkerOrder(Order.POST)
public class MyCacheManifestLinker extends DefaultCacheManifestLinker {
  @Override
  protected String[] otherCachedFiles() {
    return new String[] {"/my-app/index.html","/my-app/css/application.css"};
  }
}
```

15.3. ErraiApp.properties

ErraiApp.properties acts both as a marker file for JARs that contain Errai-enabled GWT modules, and as a place to put configuration settings for those modules in the rare case that non-default configuration is necessary.

15.3.1. As a Marker File

An `ErraiApp.properties` *file must appear at the root of each classpath location that contains an Errai module.* The contents of JAR and directory classpath entries that do not contain an `ErraiApp.properties` are effectively invisible to Errai's classpath scanner.

15.3.2. As a Configuration File

ErraiApp.properties is usually left empty, but it can contain configuration settings for both the core of Errai and any of its extensions. Configuration properties defined and used by Errai components have keys that start with " `errai.` ". Third party extensions should each choose their own prefix for keys in ErraiApp.properties.

15.3.2.1. Configuration Merging

In a non-trivial application, there will be several instances of ErraiApp.properties on the classpath (one per JAR file that contains Errai modules, beans, or portable classes).

Before using the configuration information from ErraiApp.properties, Errai reads the contents of every ErraiApp.properties on the classpath. The configuration information in all these files is merged together to form one set of key=value pairs.

If the same key appears in more than one ErraiApp.properties file, only one of the values will be associated with that key. The other values will be ignored. In future versions of Errai, this condition may be made into an error. It's best to avoid specifying the same configuration key in multiple ErraiApp.properties files.

15.3.2.2. Errai Marshalling Configuration

- *errai.marshalling.use_static_marshallers* when set to `false`, Errai will not use the precompiled server-side marshallers even if the generated `ServerMarshallingFactoryImpl` class is found on the classpath. This is useful when using Dev Mode in conjunction with an external server such as JBoss AS 7 or EAP 6.
- *errai.marshalling.force_static_marshallers* when set to `true`, Errai will not use dynamic marshallers. If the generated `ServerMarshallingFactoryImpl` cannot be loaded (possibly after an attempt to generate it on-the-fly), the Errai web app will fail to start.

Errai also supports configuring portable types in `ErraiApp.properties` as an alternative to the `@Portable` annotation. See *the Errai Marshalling section on Manual Mapping* for details.

15.3.2.3. Errai IoC Configuration

- *errai.ioc.QualifyingMetaDataFactory* specifies the fully-qualified class name of the QualifyingMetadataFactory implementation to use with Errai IoC.
- *errai.ioc.enabled.alternatives* specifies a whitespace-separated list of fully-qualified class names for *alternative beans* . See *Alternatives and Mocks* for details.

- *errai.ioc.async_bean_manager* a boolean property that when set to true (defaults to false) will activate asynchronous IOC to allow for *code splitting* [http://www.gwtproject.org/doc/latest/DevGuideCodeSplitting.html]. The code of types annotated with `@LoadAsync` will be downloaded the first time it is needed. `@LoadAsync` also allows to specify a fragment name using a class literal. Using GWT 2.6.0 or higher, all types with the same fragment name will be part of the same split point.
- *errai.ioc.blacklist* specifies a whitespace-separated list of classes that should be hidden from Errai IOC and that will be excluded when generating the bean graph and wiring components. Wildcards are supported to exclude all types underneath a package e.g. org.jboss.myapp.exclude.* (all types under the exclude package will be hidden from ERRAI IOC).
- *errai.ioc.whitelist* when this property is present all types in your application are hidden from Errai IOC by default. It specifies a whitespace-separated list of classes that should be visible to IOC and that will be included when generating the bean graph and wiring components. Wildcards are supported to include all types underneath a package e.g. org.jboss.myapp.include.* (all types under the include package will be visible to ERRAI IOC).

15.3.2.4. Errai JPA Configuration

- *errai.jpa.whitelist* specifies a whitespace-separated list of fully-qualified class names that should be included in Errai JPA's scanning and in the Entity Manager. Any entities that are not part of the whitelist are ignored by Errai JPA. This property supports wildcards to exclude everything within a package (e.g. org.jboss.myapp.exclude.*).
- *errai.jpa.blacklist* specifies a whitespace-separated list of fully-qualified class names that should be excluded from Errai JPA's scanning. This can be used for client-side entities that do not use Errai JPA. This property supports wildcards to exclude everything within a package (e.g. org.jboss.myapp.exclude.*). The blacklist has priority over the whitelist. This means that if a class is part of both the blacklist and the whitelist, the class will be blacklisted, i.e. ignored by Errai JPA.

15.4. Messaging (Errai Bus) Configuration

15.4.1. Compile-time Dependencies

The following compile-time dependency is required for Errai Messaging:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-bus</artifactId>
  <version>${errai.version}</version>
</dependency>
```

Or if you are not using Maven, have `errai-bus-${errai.version}.jar` on the classpath.

If you are also using Errai IOC or Errai CDI and wish to use inject Errai Messaging dependencies, you will also want this dependency:

```
  <dependency>
    <groupId>org.jboss.errai</groupId>
    <artifactId>errai-ioc-bus-support</artifactId>
    <version>${errai.version}</version>
  </dependency>
```

Or if you are not using Maven, have `errai-ioc-bus-support-${errai.version}.jar` on the classpath.

15.4.2. Disabling remote communication

In some cases it might be desirable to prevent the client bus from communicating with the server. One use case for this is when all communication with the server is handled using JAX-RS and the constant long polling requests for message exchange are not needed.

To turn off remote communication in the client bus the following JavaScript variable can be set in the HTML host page:

```
<script type="text/javascript">
  erraiBusRemoteCommunicationEnabled = false;
</script>
```

15.4.3. Configuring an alternative remote remote bus endpoint

By default the remote bus is expected at the GWT web application's context path. In case the remote bus is part of a different web application or deployed on a different server, the following configuration can be used in the HTML host page to configure the remote bus endpoint used on the client.

```
<script type="text/javascript">
  erraiBusApplicationRoot = "/MyRemoteMessageBusEnpoint";
</script>
```

15.4.4. ErraiService.properties

The ErraiService.properties file contains basic configuration for the bus itself. Unlike ErraiApp.properties, there should be at most one ErraiService.properties file on the classpath of a deployed application. If you do not need to set any properties to their non-default values, this file can be omitted from the deployment entirely.

15.4.4.1. Message Dispatching

Dispatchers encapsulate the strategy for taking messages that need to be delivered somewhere and seeing that they are delivered to where they need to go. There are two primary implementations that are provided with Errai, depending on your needs.

SimpleDispatcher:

SimpleDispatcher is basic implementation that provides no asychronous delivery mechanism. Rather, when you configure the Errai to use this implementation, messages are delivered to their endpoints synchronously. The incoming HTTP thread will be held open until the messages are delivered.

While this sounds like it has almost no advantages, especially in terms of scalablity. Using the SimpleDispatcher can be far preferable when you're developing your application, as any errors and stack traces will be far more easily traced and some cloud services may not permit the use of threads in any case.

AsyncDispatcher:

The AsyncDispatcher provides full asynchronous delivery of messages. When this dispatcher is used, HTTP threads will have control immediately returned upon dispatch of the message. This dispatcher provides far more efficient use of resources in high-load applications, and will significantly decrease memory and thread usage overall.

- *errai.dispatcher.implementation* specifies the dispatcher implementation to be used by the bus. There are two implementations which come with Errai out of the box: the `SimpleDispatcher` and the `AsyncDispatcher` . See ERRAI:Dispatcher Implementations for more information about the differences between the two.

15.4.4.2. Threading

- *errai.async_thread_pool_size* specifies the total number of worker threads in the worker pool for handling and delivering messages. Adjusting this value does not have any effect if you are using the SimpleDispatcher.
- *errai.async.worker_timeout* specifies the total amount of time (in seconds) that a service is given to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value has no effect if you are using the SimpleDispatcher.

15.4.4.3. Buffering

- *errai.bus.buffer_size* The total size of the transmission buffer, in megabytes. If this attribute is specified along with `errai.bus.buffer_segment_count` , then the segment count is inferred by the calculation `buffer_segment_count / buffer_size}. If {{errai.bus.buffer_segment_count` is also defined, it will be ignored in the presence of this property. Default value: 32.

- *errai.bus.buffer_segment_size* The transmission buffer segment size in bytes. This is the minimum amount of memory each message will consume while stored within the buffer. Defualt value: 8.
- *errai.bus.buffer_segment_count* The number of segments in absolute terms. If this attribute is specified in the absence of `errai.bus.buffer_size` , the buffer size is inferred by the calculation `buffer_segment_size / buffer_segment_count` .
- *errai.bus.buffer_allocation_mode* Buffer allocation mode. Allowed values are `direct` and `heap` . Direct allocation puts buffer memory outside of the JVM heap, while heap allocation uses buffer memory inside the Java heap. For most situations, heap allocation is preferable. However, if the application is data intensive and requires a substantially large buffer, it is preferable to use a direct buffer. From a throughput perspective, current JVM implementations pay about a 20% performance penalty for direct-allocated memory access. However, your application may show better scaling characteristics with direct buffers. Benchmarking under real load conditions is the only way to know the optimal setting for your use case and expected load. Default value: `direct` .

15.4.4.4. Clustering

- *errai.bus.enable_clustering* A boolean indicating whether or not Errai's server side bus should attempt to orchestrate with its peers. The orchestration mechanism is dependent on the configured clustering provider (e.g. UDP based multicast discovery in case of the default JGroups provider). The default value is `false` .
- *errai.bus.clustering_provider* The fully qualified class name of the clustering provider implementation. A class that implements `org.jboss.errai.bus.server.cluster.ClusteringProvider` . Currently the only build-in provider is the `org.jboss.errai.bus.server.cluster.jgroups.JGroupsClusteringProvider` .

15.4.4.5. Startup Configuration

- *errai.auto_discover_services* A boolean indicating whether or not the Errai bootstrapper should automatically scan for services. *This property must be set to true if and only if Errai CDI is not on the classpath* . The default value is `false`.
- *errai.auto_load_extensions* A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions. The default value is `true`.

15.4.4.6. Example Configuration

```
##
## Request dispatcher implementation (default is SimpleDispatcher)
##
```

```
#errai.dispatcher_implementation=org.jboss.errai.bus.server.SimpleDispatcher
errai.dispatcher_implementation=org.jboss.errai.bus.server.AsyncDispatcher

#
## Worker pool size. This is the number of threads the asynchronous worker pool
 should provide for
processing
## incoming messages. This option is only valid when using the AsyncDispatcher
 implementation.
##
errai.async.thread_pool_size=5

##
## Worker timeout (in seconds). This defines the time that a single asychronous
 process may run,
before the worker pool
## terminates it and reclaims the thread. This option is only valid when using
 the AsyncDispatcher
implementation.
##
errai.async.worker.timeout=5
```

15.4.5. Servlet Configuration

Errai has several different implementations for HTTP traffic to and from the bus. We provide a universally-compatible blocking implementation that provides fully synchronous communication to/from the server-side bus. Where this introduces scalability problems, we have implemented many webserver-specific implementations that take advantage of the various proprietary APIs to provide true asynchrony.

These included implementations are packaged at: `org.jboss.errai.bus.server.servlet` .

One is Enough!

You should use just one of the options below. Configuring multiple ErraiServlet implementations in the same application will lead to unpredictable behaviour!

Remember that all Errai demos and archetypes are preconfigured with DefaultBlockingServlet as a servlet. You will need to remove this default setup if you choose to use a different ErraiServlet implementation in your app.

Rolling your own security? Beware!

All of the following examples use a wildcard mapping for `\*.erraiBus` with no path prefix. This allows Errai Bus to communicate from any point in your application's

URI hierarchy, which allows bus communication to work properly no matter where you choose to put your GWT host page.

For example, all of the following are equivalent from Errai's point of view:

- /in.erraiBus
- /foo/bar/in.erraiBus
- /long/path/to/get/to.erraiBus

If you rely on your own security rules or a custom security filter to control access to Errai Bus (rather than the security framework within Errai Bus,) ensure you use the same mapping pattern for that `filter-mapping` or `security-constraint` as you do for the Errai Servlet itself.

15.4.5.1. DefaultBlockingServlet

This ErraiServlet implementation should work in virtually any servlet container that supports Java Servlets 2.0 or higher. It provides purely synchronous request handling. The one scenario where this servlet will not work is in servers that put restrictions on putting threads into sleep states.

The default DefaultBlockingServlet which provides the HTTP-protocol gateway between the server bus and the client buses.

As its name suggests, DefaultBlockingServlet is normally configured as an HTTP Servlet in the `web.xml` file:

```
<servlet>
  <servlet-name>ErraiServlet</servlet-name>
     <servlet-class>org.jboss.errai.bus.server.servlet.DefaultBlockingServlet</
servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>ErraiServlet</servlet-name>
  <url-pattern>*.erraiBus</url-pattern>
</servlet-mapping>
```

15.4.5.2. DefaultBlockingServlet configured as Filter

Alternatively, the DefaultBlockingServlet can be deployed as a Servlet Filter. This may be necessary in cases where an existing filter is configured in the web application, and that filter interferes with the Errai Bus requests. In this case, configuring DefaultBlockingServlet to handle `\*.erraiBus` requests ahead of other filters in web.xml will solve the problem:

```
<filter>
  <filter-name>ErraiServlet</filter-name>
      <filter-class>org.jboss.errai.bus.server.servlet.DefaultBlockingServlet</
filter-class>
</filter>

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

15.4.5.3. JettyContinuationsServlet

The Jetty implementation leverages Jetty's continuations support, which allows for threadless pausing of port connections. This servlet implementation should work without any special configuration of Jetty.

```
<servlet>
  <servlet-name>ErraiServlet</servlet-name>
  <servlet-class>org.jboss.errai.bus.server.servlet.JettyContinuationsServlet</
servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>ErraiServlet</servlet-name>
  <url-pattern>*.erraiBus</url-pattern>
</servlet-mapping>
```

15.4.5.4. StandardAsyncServlet

This implementation leverages asynchronous support in Servlet 3.0 to allow for threadless pausing of port connections. Note that `<async-supported>true</async-supported>` has to be added to the servlet definition in `web.xml` .

```
<servlet>
  <servlet-name>ErraiServlet</servlet-name>
       <servlet-class>org.jboss.errai.bus.server.servlet.StandardAsyncServlet</
servlet-class>
  <load-on-startup>1</load-on-startup>
  <async-supported>true</async-supported>
</servlet>

<servlet-mapping>
```

```
  <servlet-name>ErraiServlet</servlet-name>
  <url-pattern>*.erraiBus</url-pattern>
</servlet-mapping>
```

15.4.5.5. Automatic Service Discovery

By default Errai relies on a provided CDI container to do server-side service discovery. But if you intend to use Errai Messaging without a CDI container, Errai can scan for services on its own if the following initialization parameter is added to the servlet configuration:

```
    <init-param>
      <param-name>auto-discover-services</param-name>
      <param-value>true</param-value>
    </init-param>
```

Warning

This configuration will cause issues (such as duplicate services) if it is set to true *and* a server-side CDI container is available.

15.5. Errai JAX-RS Setup

15.5.1. Compile-time dependency

To use Errai JAX-RS, you must include it on the compile-time classpath. If you are using Maven for your build, add this dependency:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-jaxrs-client</artifactId>
  <version>${errai.version}</version>
  <scope>provided</scope>
</dependency>
```

Or if you are not using Maven for dependency management, add `errai-jaxrs-client-${errai.version}.jar` to your classpath.

If you intend to use Errai's JSON format on the wire you will need to add Errai's JAX-RS JSON provider to your classpath and make sure it gets deployed to the server.

```
<dependency>
```

```
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-jaxrs-provider</artifactId>
  <version>${errai.version}</version>
</dependency>
```

Or manually add `errai-jaxrs-provider-${errai.version}.jar` in case you're not using Maven. If your REST service returns Jackson generated JSON you do not need the errai-jaxrs-provider (see *Configuration*) .

15.5.2. GWT Module

Once you have Errai JAX-RS on your classpath, ensure your application inherits the GWT module as well. Add this line to your application's `*.gwt.xml` file:

```
<inherits name="org.jboss.errai.enterprise.Jaxrs"/>
```

15.5.3. Configuration

15.5.3.1. Configuring the default root path of JAX-RS endpoints

All paths specified using the `@Path` annotation on JAX-RS interfaces are by definition relative paths. Therefore, by default, it is assumed that the JAX-RS endpoints can be found at the specified paths relative to the GWT client application's context path.

To configure a relative or absolute root path, the following JavaScript variable can be set in either:

- The host HTML page;

```
<script type="text/javascript">
  erraiJaxRsApplicationRoot = "/MyJaxRsEndpointPath";
</script>
```

- By using a JSNI method;

```
private native void setMyJaxRsAppRoot(String path) /*-{
  $wnd.erraiJaxRsApplicationRoot = path;
}-*/;
```

- Or by simply invoking.

```
RestClient.setApplicationRoot("/MyJaxRsEndpointPath");
```

The root path will be prepended to all paths specified on the JAX-RS interfaces. It serves as the base URL for all requests sent from the client.

15.5.3.2. Enabling Jackson marshalling

The following options are available for activating Jackson marshalling on the client. Note that this is a client-side configuration, the JAX-RS endpoint is assumed to already return a Jackson representation (Jackson is supported by all JAX-RS implementations). The `errai-jaxrs-provider-${errai.version}.jar` does not have to be deployed on the server in this case!

To use the Jackson marshaller add on of these configurations:

- Set a Javascript variable in the GWT Host Page;

```
<script type="text/javascript">
  erraiJaxRsJacksonMarshallingActive = true;
</script>
```

- Use a JSNI method;

```
private native void setJacksonMarshallingActive(boolean active) /*-{
  $wnd.erraiJaxRsJacksonMarshallingActive = active;
}-*/;
```

- Or invoke a method in RestClient.

```
RestClient.setJacksonMarshallingActive(true);
```

15.6. Errai JPA

15.6.1. Compile-time Dependencies

To use Errai JPA, you must include it on the compile-time classpath. If you are using Maven for your build, add this dependency:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-jpa-client</artifactId>
  <version>${errai.version}</version>
</dependency>
```

If you are not using Maven for dependency management, add `errai-jpa-client-${errai.version}.jar` , Hibernate 4.1.1, and Google Guava for GWT 12.0 to your compile-time classpath.

15.6.2. GWT Module Descriptor

Once you have Errai JPA on your classpath, ensure your application inherits the GWT module as well. Add this line to your application's `*.gwt.xml` file:

```
<inherits name="org.jboss.errai.jpa.JPA"/>
```

15.7. Errai JPA Data Sync

15.7.1. Compile-time Dependencies

First, ensure your `pom.xml` includes a dependency on the Data Sync module. This module must be packaged in your application's WAR file, so include it with the default scope (compile):

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-jpa-datasync</artifactId>
  <version>${errai.version}</version>
</dependency>
```

15.7.2. GWT Module Descriptor

Then, ensure your project's `*.gwt.xml` module descriptor includes a dependency on the Data Sync GWT module:

```
<inherits name="org.jboss.errai.jpa.sync.DataSync"/>
```

15.8. Errai Data Binding

15.8.1. Compile-time Dependencies

To use Errai's data binding module, you must include it on the compile-time classpath. If you are using Maven for your build, add this dependency:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-data-binding</artifactId>
```

```
  <version>${errai.version}</version>
</dependency>
```

If you are not using Maven for dependency management, add `errai-data-binding-${errai.version}.jar` to your classpath.

15.8.2. GWT module descriptor

You must also inherit the Errai data binding module by adding the following line to your GWT module descriptor (gwt.xml).

Example 15.1. App.gwt.xml

```
<inherits name="org.jboss.errai.databinding.DataBinding" />
```

15.8.3. Bootstrapping Data Binding without Errai IOC

In case you don't want to or cannot use Errai's IOC container you will have to manually bootstrap Errai Data Binding and inherit the Errai Common GWT module:

```
BindableProxyLoader proxyLoader = GWT.create(BindableProxyLoader.class);
proxyLoader.loadBindableProxies();
```

```
<inherits name="org.jboss.errai.common.ErraiCommon"/>
```

15.9. Errai UI

15.9.1. Compile-time dependency

The easiest way to get Errai UI on your classpath is to depend on the special `errai-javaee-all` artifact, which brings in most Errai modules:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-javaee-all</artifactId>
  <version>${errai.version}</version>
</dependency>
```

Or if you prefer to manage your project's dependency in a finer-grained way, you can depend on `errai-ui` directly:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-ui</artifactId>
  <version>${errai.version}</version>
</dependency>
```

15.9.2. GWT Module Descriptor

Once you have Errai UI on your classpath, ensure your application inherits the GWT module as well. Add this line to your application's `*.gwt.xml` file:

```
<inherits name="org.jboss.errai.ui.UI" />
```

15.10. Errai UI Navigation

15.10.1. Compile-time Dependencies

To use Errai UI Navigation, you must include it on the compile-time classpath. If you are using Maven for your build, add these dependencies:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-navigation</artifactId>
  <version>${errai.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-cdi-client</artifactId>
  <version>${errai.version}</version>
  <scope>provided</scope>
</dependency>
```

If you are not using Maven for dependency management, add `errai-navigation-${errai.version}.jar` to the compile-time classpath of a project that's already set up for Errai UI templating.

15.10.2. GWT Module Descriptor

Once you have Errai UI Navigation on your classpath, ensure your application inherits the GWT module as well. Add this line to your application's `*.gwt.xml` file:

```
<inherits name="org.jboss.errai.ui.nav.Navigation"/>
```

15.11. Errai Cordova (Mobile Support)

15.11.1. Compile-time Dependencies

Using Errai Cordova requires the following compile-time dependency:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-cordova</artifactId>
  <version>${errai.verison}</version>
</dependency>
```

15.11.2. Cordova Maven Plugin

Errai Cordova allows you build an Errai app to natively run on a device. In order to make this as easy as possible we have a maven plugin that will create a native binary that you can install on a device. It will put the html and javascript of you application in a *cordova* [http://cordova.apache.org/] application.

```
<build>
 ...
    <plugins>
      <plugin>
        <groupId>org.jboss.errai</groupId>
        <artifactId>cordova-maven-plugin</artifactId>
        <version>${errai.version}</version>
      </plugin>
```

15.11.3. GWT Module Descriptor

Add the following to your application's `*.gwt.xml` module file:

```
<inherits name="org.jboss.errai.ui.Cordova"/>
```

Because the client is no longer served by the server the client will need to know how it can reach the server to do that place the following in your gwt.xml:

```
<replace-with class="com.company.application.Config">
```

```
   <when-type-is class="org.jboss.errai.bus.client.framework.Configuration" />
</replace-with>
```

This class must implement `org.jboss.errai.bus.client.framework.Configuration` and return the url where the server is configured.

```
import org.jboss.errai.bus.client.framework.Configuration;

public class Config implements Configuration {
    @Override
    public String getRemoteLocation() {
         // you probably want to do something environment specify here instead
 of something like this:
       return "https://grocery-edewit.rhcloud.com/errai-jpa-demo-grocery-list";
    }
}
```

15.11.4. Building with Errai Cordova

Now you can execute a native build with the following maven command:

```
#will build all supported platforms for now only ios and android
mvn cordova:build-project

#only build android
mvn cordova:build-project -Dplatform=android

#start the ios emulator with the deployed application
mvn cordova:emulator -Dplatform=ios
```

Important

For these to work you'll need to have the SDK's installed and on your path! In case of android you will additionally have to have ANDROID_HOME environment variable set.

15.12. Errai Security

15.12.1. Compile-time dependency

Errai Security requires to modules to be included in a project:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-security-server</artifactId>
  <version>${errai.version}</version>
</dependency>
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-security-client</artifactId>
  <scope>provided</scope>
  <version>${errai.version}</version>
</dependency>
```

If you are using picketlink for authentication, you should also include this:

```
<dependency>
  <groupId>org.jboss.errai</groupId>
  <artifactId>errai-security-picketlink</artifactId>
  <version>${errai.version}</version>
</dependency>
```

15.12.2. GWT Module Descriptor

Once you have Errai Security Client on your classpath, ensure your application inherits the GWT module as well. Add this line to your application's `*.gwt.xml` file:

```
<inherits name="org.jboss.errai.security.Security" />
```

15.12.3. CDI and Interceptor Bindings

Errai security requires a CDI container to intercept calls to remote services. In particular, the following interceptor must be added to your application's `beans.xml`:

```
<interceptors>
  <class>org.jboss.errai.security.server.ServerSecurityRoleInterceptor</class>
</interceptors>
```

15.13. Errai Project Dependencies

For those not using maven, here is the dependency tree of Errai project jars.

15.13.1. Errai Messaging

- org.jboss.errai:errai-bus:jar
  - org.jboss.errai:errai-common:jar:compile
    - org.jboss.errai.reflections:reflections:jar:compile
    - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
  - org.jboss.errai:errai-config:jar:compile
  - org.jboss.errai:errai-marshalling:jar:compile
    - org.jboss.errai:errai-codegen:jar:compile
    - org.jboss.errai:errai-codegen-gwt:jar:compile
    - javax.annotation:jsr250-api:jar:1.0:compile
    - javax.enterprise:cdi-api:jar:1.0-SP4:compile
      - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
  - org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec:jar:1.0.0.Final:provided
  - com.google.gwt:gwt-user:jar:2.5.1:provided
    - javax.validation:validation-api:jar:1.0.0.GA:provided
    - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
    - org.json:json:jar:20090211:provided
  - com.google.gwt:gwt-dev:jar:2.5.1:provided
  - com.google.inject:guice:jar:3.0:compile
    - aopalliance:aopalliance:jar:1.0:compile
  - javax.inject:javax.inject:jar:1:compile
  - org.mvel:mvel2:jar:2.1.7.Final:compile
  - org.slf4j:slf4j-api:jar:1.7.2:compile
  - org.mortbay.jetty:jetty:jar:6.1.25:provided
    - org.mortbay.jetty:jetty-util:jar:6.1.25:provided
    - org.mortbay.jetty:servlet-api:jar:2.5-20081211:provided

- org.jboss:jboss-vfs:jar:3.0.1.GA:provided
- junit:junit:jar:4.10:compile
  - org.hamcrest:hamcrest-core:jar:1.1:compile
- org.javassist:javassist:jar:3.15.0-GA:compile
- io.netty:netty-codec-http:jar:4.0.12.Final:compile
  - io.netty:netty-codec:jar:4.0.12.Final:compile
    - io.netty:netty-transport:jar:4.0.12.Final:compile
  - io.netty:netty-handler:jar:4.0.12.Final:compile
    - io.netty:netty-buffer:jar:4.0.12.Final:compile
      - io.netty:netty-common:jar:4.0.12.Final:compile
- com.google.guava:guava:jar:14.0.1:compile
- javax:javaee-api:jar:6.0:provided
- org.jgroups:jgroups:jar:3.2.10.Final:compile

15.13.2. Errai CDI

- org.jboss.errai:errai-weld-integration:jar
  - org.jboss.errai:errai-common:jar:compile
    - org.jboss.errai.reflections:reflections:jar:compile
      - dom4j:dom4j:jar:1.6.1:compile
        - xml-apis:xml-apis:jar:1.4.01:compile
    - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
  - org.jboss.errai:errai-bus:jar:compile
    - org.jboss.errai:errai-marshalling:jar:compile
    - com.google.inject:guice:jar:3.0:compile
      - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile
    - org.mvel:mvel2:jar:2.1.7.Final:compile

- org.javassist:javassist:jar:3.15.0-GA:compile
- io.netty:netty-codec-http:jar:4.0.12.Final:compile
  - io.netty:netty-codec:jar:4.0.12.Final:compile
    - io.netty:netty-transport:jar:4.0.12.Final:compile
  - io.netty:netty-handler:jar:4.0.12.Final:compile
    - io.netty:netty-buffer:jar:4.0.12.Final:compile
    - io.netty:netty-common:jar:4.0.12.Final:compile
- com.google.guava:guava:jar:14.0.1:compile
- org.jgroups:jgroups:jar:3.2.0.Final:compile

- org.jboss.errai:errai-config:jar:compile
- org.jboss.errai:errai-ioc:jar:provided
  - org.jboss.errai:errai-codegen:jar:compile
  - org.jboss.errai:errai-codegen-gwt:jar:compile
  - javax.annotation:jsr250-api:jar:1.0:compile
- org.jboss.errai:errai-ioc-bus-support:jar:provided
- javax.enterprise:cdi-api:jar:1.0-SP4:provided
  - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:provided
- org.jboss.errai:errai-cdi-client:jar:compile
- org.slf4j:slf4j-api:jar:1.7.2:provided
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - org.json:json:jar:20090211:provided
- com.google.gwt:gwt-dev:jar:2.5.1:provided
- javax.validation:validation-api:jar:1.0.0.GA:provided
- javax.validation:validation-api:jar:sources:1.0.0.GA:provided
- org.jboss.spec.javax.ejb:jboss-ejb-api_3.1_spec:jar:1.0.2.Final:provided
- org.quartz-scheduler:quartz:jar:2.1.6:provided

- c3p0:c3p0:jar:0.9.1.1:provided
- junit:junit:jar:4.10:provided
  - org.hamcrest:hamcrest-core:jar:1.1:provided
- org.jboss:jboss-common-core:jar:2.2.17.GA:provided
  - org.jboss.logging:jboss-logging-spi:jar:2.1.0.GA:provided
- org.jboss.errai:errai-javax-enterprise:jar:provided
- org.jboss.errai:errai-data-binding:jar:provided
  - com.google.guava:guava-gwt:jar:14.0.1:provided
    - com.google.code.findbugs:jsr305:jar:1.3.9:provided
- org.jboss.errai:errai-cdi-client:jar
  - org.jboss.errai:errai-javax-enterprise:jar:provided
  - org.jboss.errai:errai-bus:jar:compile
    - org.jboss.errai:errai-common:jar:compile
      - org.jboss.errai.reflections:reflections:jar:compile
        - dom4j:dom4j:jar:1.6.1:compile
        - xml-apis:xml-apis:jar:1.4.01:compile
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
    - org.jboss.errai:errai-config:jar:compile
    - org.jboss.errai:errai-marshalling:jar:compile
    - com.google.inject:guice:jar:3.0:compile
      - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile
    - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.slf4j:slf4j-api:jar:1.7.2:compile
    - org.javassist:javassist:jar:3.15.0-GA:compile
    - io.netty:netty-codec-http:jar:4.0.12.Final:compile

    - io.netty:netty-codec:jar:4.0.12.Final:compile
        - io.netty:netty-transport:jar:4.0.12.Final:compile
    - io.netty:netty-handler:jar:4.0.12.Final:compile
        - io.netty:netty-buffer:jar:4.0.12.Final:compile
        - io.netty:netty-common:jar:4.0.12.Final:compile
  - com.google.guava:guava:jar:14.0.1:compile
  - org.jgroups:jgroups:jar:3.2.10.Final:compile
- org.jboss.errai:errai-ioc-bus-support:jar:compile
  - org.jboss.errai:errai-codegen:jar:compile
  - javax.annotation:jsr250-api:jar:1.0:compile
  - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
- org.jboss.spec.javax.ejb:jboss-ejb-api_3.1_spec:jar:1.0.2.Final:provided
- org.quartz-scheduler:quartz:jar:2.1.6:provided
  - c3p0:c3p0:jar:0.9.1.1:provided
- org.jboss.errai:errai-ioc:jar:compile
  - org.jboss.errai:errai-codegen-gwt:jar:compile
- javax:javaee-api:jar:6.0:provided
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - org.json:json:jar:20090211:provided
- com.google.gwt:gwt-dev:jar:2.5.1:provided
- junit:junit:jar:4.10:provided
  - org.hamcrest:hamcrest-core:jar:1.1:provided
- javax.validation:validation-api:jar:1.0.0.GA:provided
- javax.validation:validation-api:jar:sources:1.0.0.GA:provided

15.13.3. Errai IOC

- org.jboss.errai:errai-ioc:jar
  - org.jboss.errai:errai-config:jar:compile
  - org.jboss.errai:errai-codegen:jar:compile
    - org.jboss.errai:errai-common:jar:compile
      - org.jboss.errai.reflections:reflections:jar:compile
        - com.google.guava:guava:jar:14.0.1:compile
        - org.javassist:javassist:jar:3.15.0-GA:compile
        - dom4j:dom4j:jar:1.6.1:compile
        - xml-apis:xml-apis:jar:1.4.01:compile
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
    - org.mvel:mvel2:jar:2.1.7.Final:compile
  - org.jboss.errai:errai-codegen-gwt:jar:compile
  - com.google.inject:guice:jar:3.0:compile
    - aopalliance:aopalliance:jar:1.0:compile
  - javax.inject:javax.inject:jar:1:compile
  - org.jboss.errai:errai-javax-enterprise:jar:compile
  - javax.annotation:jsr250-api:jar:1.0:compile
  - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
  - com.google.gwt:gwt-user:jar:2.5.1:provided
    - javax.validation:validation-api:jar:1.0.0.GA:provided
    - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
    - org.json:json:jar:20090211:provided
  - com.google.gwt:gwt-dev:jar:2.5.1:provided
  - junit:junit:jar:4.10:provided
    - org.hamcrest:hamcrest-core:jar:1.1:provided

15.13.4. Errai UI

- org.jboss.errai:errai-uibinder:jar
  - org.jboss.errai:errai-bus:jar:provided
    - org.jboss.errai:errai-common:jar:provided
      - org.jboss.errai.reflections:reflections:jar:provided
        - dom4j:dom4j:jar:1.6.1:provided
        - xml-apis:xml-apis:jar:1.4.01:provided
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:provided
    - org.jboss.errai:errai-config:jar:provided
    - org.jboss.errai:errai-marshalling:jar:provided
    - org.mvel:mvel2:jar:2.1.7.Final:provided
    - org.slf4j:slf4j-api:jar:1.7.2:provided
    - org.javassist:javassist:jar:3.15.0-GA:provided
    - io.netty:netty-codec-http:jar:4.0.12.Final:compile
      - io.netty:netty-codec:jar:4.0.12.Final:compile
        - io.netty:netty-transport:jar:4.0.12.Final:compile
      - io.netty:netty-handler:jar:4.0.12.Final:compile
        - io.netty:netty-buffer:jar:4.0.12.Final:compile
        - io.netty:netty-common:jar:4.0.12.Final:compile
    - com.google.guava:guava:jar:14.0.1:provided
    - org.jgroups:jgroups:jar:3.2.10.Final:provided
  - org.jboss.errai:errai-ioc:jar:provided
    - org.jboss.errai:errai-codegen:jar:provided
    - org.jboss.errai:errai-codegen-gwt:jar:provided
    - org.jboss.errai:errai-javax-enterprise:jar:provided
    - javax.annotation:jsr250-api:jar:1.0:provided
    - javax.enterprise:cdi-api:jar:1.0-SP4:provided

        - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:provided
    - com.google.inject:guice:jar:3.0:provided
        - aopalliance:aopalliance:jar:1.0:provided
    - javax.inject:javax.inject:jar:1:compile
    - com.google.gwt:gwt-user:jar:2.5.1:provided
        - javax.validation:validation-api:jar:1.0.0.GA:provided
        - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
        - org.json:json:jar:20090211:provided
    - com.google.gwt:gwt-dev:jar:2.5.1:provided

15.13.5. Errai Navigation

- org.jboss.errai:errai-navigation:jar
    - org.jboss.errai:errai-cdi-client:jar:provided
        - org.jboss.errai:errai-bus:jar:provided
            - org.mvel:mvel2:jar:2.1.7.Final:provided
            - org.slf4j:slf4j-api:jar:1.7.2:provided
            - org.javassist:javassist:jar:3.15.0-GA:provided
            - io.netty:netty-codec-http:jar:4.0.12.Final:compile
                - io.netty:netty-codec:jar:4.0.12.Final:compile
                - io.netty:netty-transport:jar:4.0.12.Final:compile
                - io.netty:netty-handler:jar:4.0.12.Final:compile
                - io.netty:netty-buffer:jar:4.0.12.Final:compile
                - io.netty:netty-common:jar:4.0.12.Final:compile
            - org.jgroups:jgroups:jar:3.2.10.Final:provided
        - org.jboss.errai:errai-ioc-bus-support:jar:provided
    - org.jboss.errai:errai-ioc:jar:provided
        - org.jboss.errai:errai-config:jar:provided

- org.jboss.errai:errai-codegen:jar:provided
- org.jboss.errai:errai-codegen-gwt:jar:provided
- com.google.inject:guice:jar:3.0:provided
  - aopalliance:aopalliance:jar:1.0:provided
- javax.inject:javax.inject:jar:1:provided
- javax.annotation:jsr250-api:jar:1.0:provided
- javax.enterprise:cdi-api:jar:1.0-SP4:provided
  - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:provided

- org.jboss.errai:errai-javax-enterprise:jar:provided
- org.jboss.errai:errai-marshalling:jar:provided
  - org.jboss.errai:errai-common:jar:provided
    - org.jboss.errai.reflections:reflections:jar:provided
      - dom4j:dom4j:jar:1.6.1:provided
      - xml-apis:xml-apis:jar:1.4.01:provided
    - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:provided
- com.google.guava:guava-gwt:jar:14.0.1:compile
  - com.google.code.findbugs:jsr305:jar:1.3.9:compile
  - com.google.guava:guava:jar:14.0.1:compile
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - javax.validation:validation-api:jar:1.0.0.GA:provided
  - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
  - org.json:json:jar:20090211:provided
- com.google.gwt:gwt-dev:jar:2.5.1:provided
- org.jboss.weld.se:weld-se-core:jar:1.1.6.Final:provided
  - org.jboss.weld:weld-spi:jar:1.1.Final:provided
    - org.jboss.weld:weld-api:jar:1.1.Final:provided

- org.jboss.weld:weld-core:jar:1.1.13.Final:provided
  - org.slf4j:slf4j-ext:jar:1.7.2:provided
  - ch.qos.cal10n:cal10n-api:jar:0.7.4:provided
- javax.el:el-api:jar:2.2:provided
- org.jboss.errai:errai-weld-integration:jar:provided

15.13.6. Errai DataBinding

- org.jboss.errai:errai-data-binding:jar
  - org.jboss.errai:errai-ioc:jar:provided
    - org.jboss.errai:errai-config:jar:compile
    - org.jboss.errai:errai-codegen:jar:compile
      - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.jboss.errai:errai-codegen-gwt:jar:compile
    - com.google.inject:guice:jar:3.0:provided
      - aopalliance:aopalliance:jar:1.0:provided
    - javax.inject:javax.inject:jar:1:compile
    - org.jboss.errai:errai-javax-enterprise:jar:provided
    - javax.annotation:jsr250-api:jar:1.0:compile
    - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - * org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
  - org.jboss.errai:errai-marshalling:jar:compile
    - org.jboss.errai:errai-common:jar:compile
      - org.jboss.errai.reflections:reflections:jar:compile
        - org.javassist:javassist:jar:3.15.0-GA:compile
        - org.slf4j:slf4j-api:jar:1.7.2:compile
        - dom4j:dom4j:jar:1.6.1:compile
        - xml-apis:xml-apis:jar:1.4.01:compile

    - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
- com.google.gwt:gwt-user:jar:2.5.1:provided
    - org.json:json:jar:20090211:provided
- com.google.gwt:gwt-dev:jar:2.5.1:provided
- com.google.guava:guava-gwt:jar:14.0.1:compile
    - com.google.code.findbugs:jsr305:jar:1.3.9:compile
    - com.google.guava:guava:jar:14.0.1:compile
- junit:junit:jar:4.10:provided
    - org.hamcrest:hamcrest-core:jar:1.1:provided
- javax.validation:validation-api:jar:1.0.0.GA:provided
- javax.validation:validation-api:jar:sources:1.0.0.GA:provided

15.13.7. Errai JPA Client

- org.jboss.errai:errai-jpa-client:jar
    - org.hibernate:hibernate-entitymanager:jar:4.2.0.Final:compile
        - org.jboss.logging:jboss-logging:jar:3.1.2.GA:compile
        - org.hibernate:hibernate-core:jar:4.2.0.Final:compile
            - antlr:antlr:jar:2.7.7:compile
        - org.jboss.spec.javax.transaction:jboss-transaction-api_1.1_spec:jar:1.0.1.Final:compile
        - dom4j:dom4j:jar:1.6.1:compile
        - org.javassist:javassist:jar:3.15.0-GA:compile
        - org.hibernate.common:hibernate-commons-annotations:jar:4.0.1.Final:compile
    - org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.1.Final:compile
    - org.jboss.errai:errai-ioc:jar:compile
        - org.jboss.errai:errai-config:jar:compile
        - org.jboss.errai:errai-codegen:jar:compile
            - org.jboss.errai:errai-common:jar:compile

        - org.jboss.errai.reflections:reflections:jar:compile
        - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
      - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.jboss.errai:errai-codegen-gwt:jar:compile
    - com.google.inject:guice:jar:3.0:compile
      - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile
    - org.jboss.errai:errai-javax-enterprise:jar:compile
    - javax.annotation:jsr250-api:jar:1.0:compile
    - javax.enterprise:cdi-api:jar:1.0-SP4:compile
      - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
  - org.jboss.errai:errai-data-binding:jar:compile
    - org.jboss.errai:errai-marshalling:jar:compile
  - com.google.gwt:gwt-user:jar:2.5.1:compile
    - javax.validation:validation-api:jar:1.0.0.GA:compile
    - javax.validation:validation-api:jar:sources:1.0.0.GA:compile
    - org.json:json:jar:20090211:compile
  - com.google.gwt:gwt-dev:jar:2.5.1:provided
  - com.google.guava:guava-gwt:jar:14.0.1:compile
    - com.google.code.findbugs:jsr305:jar:1.3.9:compile
    - com.google.guava:guava:jar:14.0.1:compile

15.13.8. Errai JPA Datasync

- org.jboss.errai:errai-jpa-datasync:jar
  - org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.1.Final:provided
  - org.jboss.errai:errai-jpa-client:jar:compile
    - org.hibernate:hibernate-entitymanager:jar:4.2.0.Final:compile

- org.jboss.logging:jboss-logging:jar:3.1.2.GA:compile
- org.hibernate:hibernate-core:jar:4.2.0.Final:compile
  - antlr:antlr:jar:2.7.7:compile
- org.jboss.spec.javax.transaction:jboss-transaction-api_1.1_spec:jar:1.0.1.Final:compile
- dom4j:dom4j:jar:1.6.1:compile
- org.hibernate.common:hibernate-commons-annotations:jar:4.0.1.Final:compile

- org.jboss.errai:errai-ioc:jar:compile
  - org.jboss.errai:errai-codegen:jar:compile
  - org.jboss.errai:errai-codegen-gwt:jar:compile
  - org.jboss.errai:errai-javax-enterprise:jar:compile
  - javax.annotation:jsr250-api:jar:1.0:compile
  - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
- org.jboss.errai:errai-data-binding:jar:compile
- com.google.gwt:gwt-user:jar:2.5.1:compile
  - javax.validation:validation-api:jar:1.0.0.GA:compile
  - javax.validation:validation-api:jar:sources:1.0.0.GA:compile
- com.google.guava:guava-gwt:jar:14.0.1:compile
  - com.google.code.findbugs:jsr305:jar:1.3.9:compile

- org.jboss.errai:errai-bus:jar:compile
  - org.jboss.errai:errai-common:jar:compile
    - org.jboss.errai.reflections:reflections:jar:compile
    - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
  - org.jboss.errai:errai-config:jar:compile
  - org.jboss.errai:errai-marshalling:jar:compile
  - com.google.inject:guice:jar:3.0:compile

        - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile
    - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.javassist:javassist:jar:3.15.0-GA:compile
    - io.netty:netty-codec-http:jar:4.0.12.Final:compile
        - io.netty:netty-codec:jar:4.0.12.Final:compile
            - io.netty:netty-transport:jar:4.0.12.Final:compile
        - io.netty:netty-handler:jar:4.0.12.Final:compile
            - io.netty:netty-buffer:jar:4.0.12.Final:compile
            - io.netty:netty-common:jar:4.0.12.Final:compile
    - com.google.guava:guava:jar:14.0.1:compile
    - org.jgroups:jgroups:jar:3.2.10.Final:compile
- org.jboss.errai:errai-cdi-client:jar:compile
    - org.jboss.errai:errai-ioc-bus-support:jar:compile

15.13.9. Errai JAXRS

- org.jboss.errai:errai-jaxrs-client:jar
    - org.jboss.errai:errai-marshalling:jar:compile
        - org.jboss.errai:errai-common:jar:compile
            - org.jboss.errai.reflections:reflections:jar:compile
                - org.javassist:javassist:jar:3.15.0-GA:compile
                - org.slf4j:slf4j-api:jar:1.7.2:compile
                - dom4j:dom4j:jar:1.6.1:compile
                - xml-apis:xml-apis:jar:1.4.01:compile
            - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
        - org.jboss.errai:errai-config:jar:compile
        - org.jboss.errai:errai-codegen:jar:compile

        - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.jboss.errai:errai-codegen-gwt:jar:compile
    - javax.inject:javax.inject:jar:1:compile
    - javax.annotation:jsr250-api:jar:1.0:compile
    - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - * org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
  - com.google.gwt:gwt-user:jar:2.5.1:provided
    - org.json:json:jar:20090211:provided
  - com.google.gwt:gwt-dev:jar:2.5.1:provided
  - junit:junit:jar:4.10:provided
    - org.hamcrest:hamcrest-core:jar:1.1:provided
  - javax.validation:validation-api:jar:1.0.0.GA:provided
  - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
  - org.jboss.resteasy:jaxrs-api:jar:2.3.6.Final:compile
  - com.google.guava:guava-gwt:jar:14.0.1:compile
    - com.google.code.findbugs:jsr305:jar:1.3.9:compile
    - com.google.guava:guava:jar:14.0.1:compile
- org.jboss.errai:errai-jaxrs-provider:jar
  - org.jboss.resteasy:jaxrs-api:jar:2.3.6.Final:compile
  - org.jboss.errai:errai-marshalling:jar:compile
    - org.jboss.errai:errai-common:jar:compile
      - org.jboss.errai.reflections:reflections:jar:compile
        - com.google.guava:guava:jar:14.0.1:compile
        - org.javassist:javassist:jar:3.15.0-GA:compile
        - org.slf4j:slf4j-api:jar:1.7.2:compile
        - dom4j:dom4j:jar:1.6.1:compile

        - xml-apis:xml-apis:jar:1.4.01:compile
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
    - org.jboss.errai:errai-config:jar:compile
    - org.jboss.errai:errai-codegen:jar:compile
      - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.jboss.errai:errai-codegen-gwt:jar:compile
    - javax.inject:javax.inject:jar:1:compile
    - javax.annotation:jsr250-api:jar:1.0:compile
    - javax.enterprise:cdi-api:jar:1.0-SP4:compile
      - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile

15.13.10. Errai Cordova

- org.jboss.errai:errai-cordova:jar
  - org.jboss.errai:errai-bus:jar:compile
    - org.jboss.errai:errai-common:jar:compile
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
    - org.jboss.errai:errai-config:jar:compile
    - org.jboss.errai:errai-marshalling:jar:compile
      - org.jboss.errai:errai-codegen:jar:compile
      - org.jboss.errai:errai-codegen-gwt:jar:compile
      - javax.annotation:jsr250-api:jar:1.0:compile
      - javax.enterprise:cdi-api:jar:1.0-SP4:compile
        - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
    - com.google.inject:guice:jar:3.0:compile
      - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile

- org.mvel:mvel2:jar:2.1.7.Final:compile
- org.slf4j:slf4j-api:jar:1.7.2:compile
- org.javassist:javassist:jar:3.15.0-GA:compile
- io.netty:netty-codec-http:jar:4.0.12.Final:compile
  - io.netty:netty-codec:jar:4.0.12.Final:compile
    - io.netty:netty-transport:jar:4.0.12.Final:compile
  - io.netty:netty-handler:jar:4.0.12.Final:compile
    - io.netty:netty-buffer:jar:4.0.12.Final:compile
    - io.netty:netty-common:jar:4.0.12.Final:compile
- com.google.guava:guava:jar:14.0.1:compile
- org.jgroups:jgroups:jar:3.2.10.Final:compile

- org.jboss.errai:errai-cdi-client:jar:compile
  - org.jboss.errai:errai-ioc-bus-support:jar:compile
    - org.jboss.errai:errai-javax-enterprise:jar:compile
  - org.jboss.errai:errai-ioc:jar:compile
- org.jboss.errai:errai-jaxrs-client:jar:compile
  - org.jboss.resteasy:jaxrs-api:jar:2.3.6.Final:compile
  - com.google.guava:guava-gwt:jar:14.0.1:compile
    - com.google.code.findbugs:jsr305:jar:1.3.9:compile
- org.jboss.errai:errai-html5:jar:compile
  - org.jboss.errai.reflections:reflections:jar:compile
    - dom4j:dom4j:jar:1.6.1:compile
      - xml-apis:xml-apis:jar:1.4.01:compile
- org.jboss.errai:errai-data-binding:jar:compile
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - javax.validation:validation-api:jar:1.0.0.GA:provided

- javax.validation:validation-api:jar:sources:1.0.0.GA:provided
- org.json:json:jar:20090211:provided

- com.google.gwt:gwt-dev:jar:2.5.1:provided
- com.googlecode.gwtphonegap:gwtphonegap:jar:2.4.0.0:compile
  - commons-io:commons-io:jar:2.4:compile
- junit:junit:jar:4.10:provided
  - org.hamcrest:hamcrest-core:jar:1.1:provided

15.13.11. Errai Security

- org.jboss.errai:errai-security-server:jar:3.0-SNAPSHOT
  - org.jboss.errai:errai-bus:jar:3.0-SNAPSHOT:compile
    - org.jboss.errai:errai-common:jar:3.0-SNAPSHOT:compile
      - org.jboss.errai.reflections:reflections:jar:3.0-SNAPSHOT:compile
        - dom4j:dom4j:jar:1.6.1:compile
        - xml-apis:xml-apis:jar:1.4.01:compile
      - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
    - org.jboss.errai:errai-config:jar:3.0-SNAPSHOT:compile
    - org.jboss.errai:errai-marshalling:jar:3.0-SNAPSHOT:compile
      - org.jboss.errai:errai-codegen:jar:3.0-SNAPSHOT:compile
      - org.jboss.errai:errai-codegen-gwt:jar:3.0-SNAPSHOT:compile
    - com.google.inject:guice:jar:3.0:compile
      - aopalliance:aopalliance:jar:1.0:compile
    - javax.inject:javax.inject:jar:1:compile
    - org.mvel:mvel2:jar:2.1.7.Final:compile
    - org.slf4j:slf4j-api:jar:1.7.2:compile
    - org.javassist:javassist:jar:3.15.0-GA:compile
    - io.netty:netty-codec-http:jar:4.0.12.Final:compile

    - io.netty:netty-codec:jar:4.0.12.Final:compile
        - io.netty:netty-transport:jar:4.0.12.Final:compile
    - io.netty:netty-handler:jar:4.0.12.Final:compile
        - io.netty:netty-buffer:jar:4.0.12.Final:compile
        - io.netty:netty-common:jar:4.0.12.Final:compile
  - com.google.guava:guava:jar:14.0.1:compile
  - org.jgroups:jgroups:jar:3.2.10.Final:compile
- org.jboss.errai:errai-data-binding:jar:3.0-SNAPSHOT:provided
  - com.google.guava:guava-gwt:jar:14.0.1:provided
    - com.google.code.findbugs:jsr305:jar:1.3.9:provided
- org.jboss.errai:errai-ui:jar:3.0-SNAPSHOT:provided
  - org.jboss.errai:errai-ioc:jar:3.0-SNAPSHOT:provided
    - org.jboss.errai:errai-javax-enterprise:jar:3.0-SNAPSHOT:provided
  - org.codehaus.jackson:jackson-mapper-asl:jar:1.9.12:provided
    - org.codehaus.jackson:jackson-core-asl:jar:1.9.9:provided
  - org.jsoup:jsoup:jar:1.7.1:provided
  - org.apache.stanbol:org.apache.stanbol.enhancer.engines.htmlextractor:jar:0.10.0:provided
    - org.apache.clerezza:rdf.core:jar:0.12-incubating:provided
        - org.osgi:org.osgi.core:jar:4.2.0:provided
        - org.osgi:org.osgi.compendium:jar:4.2.0:provided
        - org.apache.clerezza:utils:jar:0.1-incubating:provided
        - commons-codec:commons-codec:jar:1.4:provided
        - org.apache.httpcomponents:httpcore:jar:4.1:provided
        - org.wymiwyg:wymiwyg-commons-core:jar:0.7.6:provided
        - commons-logging:commons-logging-api:jar:1.1:provided
        - javax.activation:activation:jar:1.1.1:provided

- org.lesscss:lesscss:jar:1.3.3:provided
  - commons-io:commons-io:jar:2.4:provided
  - commons-logging:commons-logging:jar:1.1.1:provided
  - org.apache.commons:commons-lang3:jar:3.1:provided
  - org.mozilla:rhino:jar:1.7R4:provided
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - javax.validation:validation-api:jar:1.0.0.GA:provided
  - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
  - org.json:json:jar:20090211:provided
- org.picketlink:picketlink-api:jar:2.5.3.Final:compile
  - org.picketlink:picketlink-idm-api:jar:2.5.3.Final:compile
- org.picketlink:picketlink-impl:jar:2.5.3.Final:compile
  - org.picketlink:picketlink-common:jar:2.5.3.Final:compile
  - org.picketlink:picketlink-idm-impl:jar:2.5.3.Final:compile
- org.apache.deltaspike.core:deltaspike-core-api:jar:0.4:compile
- org.jboss.weld.se:weld-se-core:jar:1.1.6.Final:provided
  - org.jboss.weld:weld-spi:jar:1.1.Final:provided
    - org.jboss.weld:weld-api:jar:1.1.Final:provided
  - org.jboss.weld:weld-core:jar:1.1.13.Final:provided
    - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile
    - org.slf4j:slf4j-ext:jar:1.7.2:provided
    - ch.qos.cal10n:cal10n-api:jar:0.7.4:provided
  - javax.enterprise:cdi-api:jar:1.0-SP4:compile
  - javax.el:el-api:jar:2.2:provided
  - javax.annotation:jsr250-api:jar:1.0:compile
- org.jboss.errai:errai-security-client:jar:3.0-SNAPSHOT

- org.jboss.errai:errai-bus:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai:errai-config:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai:errai-marshalling:jar:3.0-SNAPSHOT:compile
  - com.google.inject:guice:jar:3.0:compile
    - aopalliance:aopalliance:jar:1.0:compile
  - javax.inject:javax.inject:jar:1:compile
  - org.mvel:mvel2:jar:2.1.7.Final:compile
  - org.slf4j:slf4j-api:jar:1.7.2:compile
  - org.javassist:javassist:jar:3.15.0-GA:compile
  - io.netty:netty-codec-http:jar:4.0.12.Final:compile
    - io.netty:netty-codec:jar:4.0.12.Final:compile
      - io.netty:netty-transport:jar:4.0.12.Final:compile
    - io.netty:netty-handler:jar:4.0.12.Final:compile
      - io.netty:netty-buffer:jar:4.0.12.Final:compile
      - io.netty:netty-common:jar:4.0.12.Final:compile
  - com.google.guava:guava:jar:14.0.1:compile
  - org.jgroups:jgroups:jar:3.2.10.Final:compile
- org.jboss.errai:errai-common:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai.reflections:reflections:jar:3.0-SNAPSHOT:compile
    - dom4j:dom4j:jar:1.6.1:compile
      - xml-apis:xml-apis:jar:1.4.01:compile
  - de.benediktmeurer.gwt-slf4j:gwt-slf4j:jar:0.0.2:compile
- org.jboss.errai:errai-ui:jar:3.0-SNAPSHOT:compile
  - org.codehaus.jackson:jackson-mapper-asl:jar:1.9.12:compile
    - org.codehaus.jackson:jackson-core-asl:jar:1.9.9:compile
  - org.jsoup:jsoup:jar:1.7.1:compile

- org.apache.stanbol:org.apache.stanbol.enhancer.engines.htmlextractor:jar:0.10.0:compile
  - org.apache.clerezza:rdf.core:jar:0.12-incubating:compile
    - org.osgi:org.osgi.core:jar:4.2.0:compile
    - org.osgi:org.osgi.compendium:jar:4.2.0:compile
    - org.apache.clerezza:utils:jar:0.1-incubating:compile
    - commons-codec:commons-codec:jar:1.4:compile
    - org.apache.httpcomponents:httpcore:jar:4.1:compile
    - org.wymiwyg:wymiwyg-commons-core:jar:0.7.6:compile
    - commons-logging:commons-logging-api:jar:1.1:compile
    - javax.activation:activation:jar:1.1.1:compile
  - org.lesscss:lesscss:jar:1.3.3:compile
    - commons-io:commons-io:jar:2.4:compile
    - commons-logging:commons-logging:jar:1.1.1:compile
    - org.apache.commons:commons-lang3:jar:3.1:compile
    - org.mozilla:rhino:jar:1.7R4:compile
- org.jboss.errai:errai-data-binding:jar:3.0-SNAPSHOT:compile
  - com.google.guava:guava-gwt:jar:14.0.1:compile
    - com.google.code.findbugs:jsr305:jar:1.3.9:compile
- org.jboss.errai:errai-navigation:jar:3.0-SNAPSHOT:compile
- org.jboss.errai:errai-ioc:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai:errai-codegen:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai:errai-codegen-gwt:jar:3.0-SNAPSHOT:compile
  - org.jboss.errai:errai-javax-enterprise:jar:3.0-SNAPSHOT:compile
  - javax.annotation:jsr250-api:jar:1.0:compile
  - javax.enterprise:cdi-api:jar:1.0-SP4:compile
    - org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1:compile

- org.jboss.errai:errai-ioc-bus-support:jar:3.0-SNAPSHOT:compile
- org.jboss.errai:errai-cdi-client:jar:3.0-SNAPSHOT:compile
- org.jboss.errai:errai-security-server:jar:3.0-SNAPSHOT:compile
  - org.picketlink:picketlink-api:jar:2.5.3.Final:compile
    - org.picketlink:picketlink-idm-api:jar:2.5.3.Final:compile
  - org.picketlink:picketlink-impl:jar:2.5.3.Final:compile
    - org.picketlink:picketlink-common:jar:2.5.3.Final:compile
    - org.picketlink:picketlink-idm-impl:jar:2.5.3.Final:compile
  - org.apache.deltaspike.core:deltaspike-core-api:jar:0.4:compile
- com.google.gwt:gwt-user:jar:2.5.1:provided
  - javax.validation:validation-api:jar:1.0.0.GA:provided
  - javax.validation:validation-api:jar:sources:1.0.0.GA:provided
  - org.json:json:jar:20090211:provided
- com.google.gwt:gwt-dev:jar:2.5.1:provided

Troubleshooting & FAQ

This section explains the cause of and solution to some common problems that people encounter when building applications with Errai.

Of course, when lots of people trip over the same problem, it's probably because there is a deficiency in the framework! A FAQ list like this is just a band-aid solution. If you have suggestions for permanent fixes to these problems, please get in touch with us: file an issue in our issue tracker, chat with us on IRC, or post a suggestion on our forum.

But for now, on to the FAQ:

16.1. Why does it seem that Errai can't see my class at compile time?

Possible symptoms:

- uncaught exception: java.lang.RuntimeException: No proxy provider found for type: *my.fully.qualified.ServiceName*

Answer: Make sure the *ErraiApp.properties* file is actually making it into your runtime classpath.

One common cause of this problem is a <resources> section in pom.xml that includes src/main/java (to expose .java sources to the GWT compiler) that does not also include src/main/resources as a resource path. You must include both explicitly:

```
<resources>
  <resource>
    <directory>src/main/java</directory>
  </resource>
  <resource>
    <directory>src/main/resources</directory>
  </resource>
</resources>
```

16.2. Why am I getting "java.lang.ClassFormatError: Illegal method name "<init>$" in class org/xyz/package/MyClass"?

Answer: This error message means that your project has a (direct or indirect) subclass of JavaScriptObject that lacks a protected no-args constructor. All subtypes of JavaScriptObject (also known as *overlay types*) must declare a protected no-args constructor, but the error

message could be much clearer. There is an issue filed in the GWT project's bug tracker for improving the error message: *GWT issue 3383* [http://code.google.com/p/google-web-toolkit/issues/detail?id=3383] .

16.3. I'm getting "java.lang.RuntimeException: There are no proxy providers registered yet." in my @PostConstruct method!

Answer: You can't invoke RPC methods via `Caller<?>` or by other means until after the Errai Bus has finished its initial handshake. Try changing your `@PostConstruct` annotation to `@AfterInitialization` . This will cause your method to be invoked later after the bus handshake has completed.

If this doesn't help, it is also possible that the proxies were never generated in the first place. Check in `.errai/RpcProxyLoaderImpl.java` to see if proxy code exists for the `@Remote` and/or `@Path` interface in question. If not, your `@Remote` interfaces were not present on the GWT compiler's classpath when your application module was compiled. Double-check your GWT compilation classpath: all `@Remote` interfaces must be visible to (in or inherited by) the GWT module that contains the `Caller<?>` types. Pay special attention that your `@Remote` and `@Path` interfaces are not in a package excluded from the GWT module (by default, every subpackage other than `client` and `shared` is invisible to the GWT compiler).

16.4. Why do I get a "404 - Not Found" page if I try to navigate to my web page by typing in the URL or refreshing the page?

Answer: There are two reasons that could cause this behaviour:

You may not have declared a default page for Errai pushState to navigate to in the case of a page not found error. For example, to navigate to the GWT host page by default, add the following lines to your web.xml file. See *Errai UI Navigation - How it Works # PushState Functionality* .

If that doesn't work, check to see if you have explicitly declared the application web context in your GWT host page. See *Errai UI Navigation - How it Works # PushState Functionality* .

Upgrade Guide

This chapter contains important information for migrating to newer versions of Errai. If you experience any problems, don't hesitate to get in touch with us. See *Reporting problems* .

17.1. Upgrading from 1.* to 2.0

The first issues that will arise after replacing the jars or after changing the version numbers in the `pom.xml` are unresolved package imports. This is due to refactorings that became necessary when the project grew. Most of these import problems can be resolved automatically by modern IDEs (Organize Imports). So, this should replace `org.jboss.errai.bus.client.protocols.*` with `org.jboss.errai.common.client.protocols.\*` for example.

The following is a list of manual steps that have to be carried out when upgrading:

- @ExposedEntity became @Portable (`org.jboss.errai.common.client.api.annotations.Portable`). See *Marshalling* for details.
- The @Conversational annotation must now target the event objects themselves, not the observer methods of the events. So an *event type* is either conversational or not; you no longer specify that listeners receive arbitrary events in a conversational context. See the *Conversational Events* section of the CDI chapter for details.
- Errai CDI projects must now use the `SimpleDispatcher` instead of the `AsynDispatcher` . This has to be configured in *Messaging (Errai Bus) Configuration* .
- The bootstrap listener (configured in `WEB-INF/web.xml`) for Errai CDI has changed (`org.jboss.errai.container.DevModeCDIBootstrap` is now `org.jboss.errai.container.CDIServletStateListener`).
- gwt 2.3.0 or newer must be used and replace older versions.
- mvel2 2.1.Beta8 or newer must be used and replace older versions.
- weld 1.1.5.Final or newer must be used and replace older versions.
- slf4j 1.6.1 or newer must be used and replace older versions.
- This step can be skipped if Maven is used to build the project. If the project is NOT built using Maven, the following jar files have to be added manually to project's build/class path: errai-common-2.x.jar, errai-marshalling-2.x.jar, errai-codegen-2.x.jar, netty-4.0.0.Alpha1.errai.r1.jar.
- If the project was built using an early version of an Errai archetype the configuration of the maven-gwt-plugin has to be modified to contain the `<hostedWebapp>path-to-your-standard-webapp-folder</hostedWebapp>` . This is usually either `war` or `src/main/webapp` .

17.2. Upgrading from 2.0.Beta to 2.0.*.Final

The following is a list of manual steps that have to be carried out when upgrading from a 2.0.Beta version to 2.0.CR1 or 2.0.Final:

- Starting with 2.0.CR1 the default for automatic service discovery has been changed in favour of CDI based applications. That means it has to be explicitly turned on for plain bus applications (Errai applications that do not use Errai-CDI). Not doing so will result in `NoSubscribersToDeliverTo` exceptions. The snippet below shows how to activate automatic service discovery: .web.xml

```
<servlet>
  <servlet-name>ErraiServlet</servlet-name>
     <servlet-class>org.jboss.errai.bus.server.servlet.DefaultBlockingServlet</
servlet-class>
  <init-param>
    <param-name>auto-discover-services</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- The `jboss7-support` module was deleted and is no longer needed as a dependency.

17.3. Upgrading from Errai 2.2.x to 2.4 or 3.0

There are some breaking API changes in the update from Errai 2.2.x to Errai 2.4.x and 3.0.x.

Here are the steps you'll need to take to get your project compiling after you update:

- Starting with Errai 2.3.0, GWT 2.5.0 or higher is required.
- Use your IDE to organize imports at the top level. In eclipse, you'd click in the Project Explorer, press Ctrl-A (select all) and then Ctrl-O (Organize Imports). Other IDEs have similar features.
- The `ErrorCallback` interface has been made more general so the same type can be shared between Errai modules. This allows you reuse your own generic error handler class for, eg, Errai JAX-RS and ErraiBus callbacks. If you want to use a generic error handler throughout your app, change your `ErrorCallback` implementations to `ErrorCallback<?>` and change the first argument type of your error() method to Object. Otherwise, if you have use-case-specific error callbacks, implement the interfaces `RestErrorCallback` or `BusErrorCallback` as appropriate.
- `IOCBeanManager` was replaced by two new types `SyncBeanManager` and `AsyncBeanManager` that need to be used instead. See *Client-side Bean Manager* for details.

Note: Errai 3 is still changing rapidly, so this section is a work in progress. Please add any additional steps you had to take in upgrading your own codebase.

17.4. Upgrading to Errai 3.0

Here are the steps you'll need to take to get your project running after you update:

- Errai's custom jetty launcher (org.jboss.errai.cdi.erver.gwt.JettyLauncher) is no longer needed and has been deleted. Simply remove the corresponding -server parameter from your GWT launch configuration if you still use it.
- The whole artifact errai-cdi-jetty has been deleted and is no longer required. Delete the JAR file from your project or remove the corresponding dependency in your pom.xml

17.5. Upgrading to Errai 3.1 from 3.0

Here are the steps you'll need to take to get your project running after you update:

- If you are using Errai to compile LESS/CSS files you will need to declare a `StyleDescriptor` listing the resources to compile in the desired order. This change was made to provide a deterministic and configurable order of styles in the generated CSS.
- Errai 3.1+ targets CDI 1.1 containers by default (i.e. WildFly 8+). If you wish to deploy to a CDI 1.0 container (i.e AS7, EAP6) you will have to downgrade the errai-weld-integration.jar to 3.0.4.Final. You can still use all of Errai's latest features and use the 3.1+ versions of all other Errai jars.

17.6. Upgrading to Errai 3.2 from 3.1

Starting from Errai 3.2.0.Final, Errai Security only supports Keycloak 1.2.0.Final and higher. Errai Security is no longer compatible with earlier versions of Keycloak.

Downloads

The distribution packages can be downloaded from jboss.org *http://jboss.org/errai/Downloads.html*

Sources

Errai is currently managed using Github. You can clone our repositories from *http://github.com/errai* .

Reporting problems

If you run into trouble don’t hesitate to get in touch with us:

- JIRA Issue Tracking: *https://jira.jboss.org/jira/browse/ERRAI*
- User Forum: *http://community.jboss.org/en/errai?view=discussions*
- Mailing List: *http://jboss.org/errai/MailingLists.html*
- IRC: *irc://irc.freenode.net/errai*

Errai License

Errai is distributed under the terms of the Apache License, Version 2.0. See *the full Apache license text* [http://www.apache.org/licenses/LICENSE-2.0].