Reference Guide

Errai Framework 1.3

by ToDo Dude ToDo Somewhere ToDo McPants

Preface

1. Document Conventions

2. Feedback

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. Installation

1.2.1. 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*

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.

2.1. Messaging Overview

It's important to understand the concept of how messaging works in ErraiBus. 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. In fact, 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 with the GWT client context, such as a way to advertise and expose components dynamically, to get around the lack of reflection in GWT.

In fact, ErraiBus was originally designed to run completely within the client but quickly evolved into having the capabilities it now has today. So keep that in mind when you run up against problems in the client space that could benefit from runtime federation.

2.2. MessageBuilder API

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 look at some use cases first.

Sending Messages with the Client BusIn 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`

Note

An astute observer will note that access to the `RequestDispatcher` differs within client code and server code. Because the client code does not run within a container, access to the `RequestDispatcher` and `MessageBus` is statically accessed using the `ErraiBus.get()` and `ErraiBus.getDispatcher()` methods. The server-side code, conversely, runs inside a dependency container for managing components. It currently uses Guice as the default container, but will more generically support the JSR-330 specification in the future.

Recieving Messages on the Server Bus / Server ServicesEvery 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!");
    }
  }
```

He 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.

Sending Messages with the Server BusIn 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 disaptcher) {
    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` .

Receiving Messages on the Client Bus/ Client ServicesMessages can be received asynchronously and arbitriraily 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 local 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.

ConversationsConversations 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 (2.4) and this is the use of the `createConversation()}}method with {{MessageBuilder .`

2.3. Single-Response Conversations & Psuedo-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.

Relay ServicesThe 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 *Section 21.14.13, "Message Routing Information"*

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.

The utility class `org.jboss.errai.bus.server.util.ServerBusUtils` contains a utility method for extracting the String-based SessionID which is used to identify the message queue associated with any particular client. You may use this method to extract the `SessionID` from a message so that you may use it for routing. For example:

```
...
  public void callback(Message message) {
    String sessionId = ServerBusUtils.getSessionId(message);

    // 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.

Now you're routing from client-to-client!

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

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);
```

A little nicer.

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

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.
```

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);
```

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 to 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.

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 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.

Making callsCalling 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 correpond 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.

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.

7.1. Scopes

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

7.1.1. 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");
  }
}
```

7.1.2. Session Scopes

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");
  }
```

```
}
```

7.2. 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.

Serialization

Serialization on the ErraiBus supports serialization within the same scope and limitations as the default GWT RPC serialization rules. In order to expose your domain objects to the bus so they can be exported across the bus, you must annotate them with the `org.jboss.errai.bus.server.annotations.ExposeEntity` annotation. The presence of this annotation will cause Errai's GWT compiler extensions to generate marshall/demarshall stubs for the annotated objects at compile-time.

For example:

```
@ExposeEntity
public class User implements java.io.Serializable {
  private int userId;

  public int getUserId() {
    return userId;
  }

  public void setUserId(int userId) {
    this.userId = userId;
  }
  [...]
}
```

Note

All exposed entities must follow Java Bean convensions, and must be in the classpath both at compile-time and at runtime. Compile-time access to the entities is required since the creation of the marshalling/demarshalling proxies involves code generation.

8.1. Serialization of external types

It may not be possible to annotate certain types you wish to expose to the bus for serialization if the entities are located in a third-party library that you do not maintain. As such, you can explicitly indicate in the configuration that you would like to have this entities made available by declaring them in the `ErraiApp.properties` of any module.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
```

```
                                org.foobie.Foobie
```

Wiring server side components

By default, ErraiBus uses Google Guice to wire components. However, we plan on standardizing on *JSR-330 Dependency Injection* specification in the near future. 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.

Bus Configuration

This section contains information on configuring the server-side bus.

10.1. web.xml and appserver configuration

Depending on what application server you are deploying on, you must provide an appropriate servlet implementation if you wish to use true, asynchronous I/O. See _section 6.5 _ for information on the available servlet implementations.

Here's a sample web.xml file:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <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>

  <context-param>
    <param-name>errai.properties</param-name>
    <param-value>/WEB-INF/errai.properties</param-value>
  </context-param>

  <context-param>
    <param-name>login.config</param-name>
    <param-value>/WEB-INF/login.config</param-value>
  </context-param>

  <context-param>
    <param-name>users.properties</param-name>
    <param-value>/WEB-INF/users.properties</param-value>
  </context-param>

</web-app>
```

10.2. ErraiService.properties

he ErraiService.properties file contains basic configuration for the bus itself.

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

##
## Specify the Authentication/Authorization Adapter to use
##
#errai.authentication_adapter=org.jboss.errai.persistence.server.security.HibernateAuthenticati
#errai.authentication_adapter=org.jboss.errai.bus.server.security.auth.JAASAdapter

##
## This property indicates whether or not authentication is required for all
 communication with the
bus. Set this
## to 'true' if all access to your application should be secure.
##
#errai.require_authentication_for_all=true
```

10.2.1. errai.dispatcher.implementation

The `errai.dispatcher_implementation` defines, as it's name quite succinctly implies, 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 section on Dispatchers for more information about the differences between the two.

10.2.2. 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.

10.2.3. errai.async.worker_timeout

Specifies the total amount of a time (in seconds) a service has to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value does not have an effect if you are using the SimpleDispatcher.

10.2.4. errai.authentication_adapter

Specifies the authentication modelAdapter the bus should use for determining whether calls should be serviced based on authentication and security principles.

10.2.5. errai.require_authentication_for_all

Indicates whether or not the bus should always require the use of authentication for all requests inbound for the bus. If this is turned on, an authentication model adapter must be defined, and any user must be authenticated before the bus will deliver any messages from the client to any service.

10.2.6. errai.auto_discover_services

A boolean indicating whether or not the Errai bootstrapper should automatically scan for services.

10.2.7. errai.auto_load_extensions

A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions.

10.3. ErraiApp.properties

The ErraiApp.properties acts as a marker file. When it is detected inside a JAR or at the top of any classpath, the subdirectories are scanned for deployable components. As such, all Errai application modules in a project should contain an ErraiApp.properties at the root of all classpaths that you wish to be scanned.

The file can also include explicitly declared serializable types (such as those from third-party code) that cannot be annotated for serialization. (See the section on serialization for more details)

errai.bus.serializableTypesDefines a list of serializable types to expose to the bus.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

Dispatchers

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.

11.1. 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.

11.2. 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.

Servlet Implementations

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 inlcuded implementations are packaged at: `org.jboss.errai.bus.server.servlet`

12.1. DefaultBlockingServlet

This is a universal, completely servlet spec (2.0) compliant, Servlet implementation. It provides purely synchronous request handling and should work in virtually any servlet container, unless there are restrictions on putting threads into sleep states.

12.2. TomcatCometServlet

The Tomcat AIO implementation of our servlet allows Errai to take advantage of Tomcat's event-based AIO APIs to improve scalability and reduce thread usage. The use of this implementation is dependant on the Tomcat container being configured to support AIO using either it's NIO or APR connectors. This servlet will NOT work with the regular HTTP and AJP connectors.

12.3. 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.

12.4. JBossCometServlet

The JBoss Comet support utilizes the JBoss Web AIO APIs (AS 5.0 and AS 6.0) to improve scalability and reduce thread usage. The HTTP, NIO, and AJP connectors are not supported. Use of this implementation requires use of the APR (Apache Portable Runtime).

12.5. GrizzlyCometServlet

Support for the comet API in the Grizzy HTTP server (used in Glassfish).

12.6. WeblogicAsyncServlet

Support for the Weblogic asynchronous APIs.

Debugging Errai Applications

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 13.1. TODO InformalFigure image title empty

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 13.2. TODO InformalFigure image title empty

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.

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* .

License and EULA

Errai is distributed under the terms of the the Apache License, Version 2.0. See *the full Apache license text* [http://www.apache.org/licenses/LICENSE-2.0] .

Introduction

17.1. 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.

17.2. Installation

17.2.1. 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*

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.

License and EULA

Errai is distributed under the terms of the the Apache License, Version 2.0. See *the full Apache license text* [http://www.apache.org/licenses/LICENSE-2.0] .

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* .

Errai Bus

21.1. Messaging

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

21.1.1. Messaging Overview

It's important to understand the concept of how messaging works in ErraiBus. 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. In fact, 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 with the GWT client context, such as a way to advertise and expose components dynamically, to get around the lack of reflection in GWT.

In fact, ErraiBus was originally designed to run completely within the client but quickly evolved into having the capabilities it now has today. So keep that in mind when you run up against problems in the client space that could benefit from runtime federation.

21.1.2. MessageBuilder API

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 look at some use cases first.

Sending Messages with the Client BusIn 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`

Note

An astute observer will note that access to the `RequestDispatcher` differs within client code and server code. Because the client code does not run within a container, access to the `RequestDispatcher` and `MessageBus` is statically accessed using the `ErraiBus.get()` and `ErraiBus.getDispatcher()` methods. The server-side code, conversely, runs inside a dependency container for managing components. It currently uses Guice as the default container, but will more generically support the JSR-330 specification in the future.

Recieving Messages on the Server Bus / Server ServicesEvery 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!");
    }
  }
```

He 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.

Sending Messages with the Server BusIn 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 disaptcher) {
    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` .

Receiving Messages on the Client Bus/ Client ServicesMessages can be received asynchronously and arbitriraily 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 local 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.

ConversationsConversations 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 (2.4) and this is the use of the `createConversation()}}method with {{MessageBuilder` .

21.1.3. Single-Response Conversations & Psuedo-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.

21.1.4. 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.

21.1.5. 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.

21.1.6. 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.

Relay ServicesThe 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 *Section 21.14.13, "Message Routing Information"*

21.1.7. 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.

The utility class `org.jboss.errai.bus.server.util.ServerBusUtils` contains a utility method for extracting the String-based SessionID which is used to identify the message queue associated with any particular client. You may use this method to extract the `SessionID` from a message so that you may use it for routing. For example:

```
...
  public void callback(Message message) {
    String sessionId = ServerBusUtils.getSessionId(message);

    // 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.

Now you're routing from client-to-client!

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

21.2. 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);
```

A little nicer.

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

21.3. 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.
```

21.4. 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);
```

21.5. 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 to 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.

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 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.

Making callsCalling 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 correpond 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.

21.6. 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.

21.6.1. Scopes

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

21.6.1.1. 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");
  }
}
```

21.6.1.2. Session Scopes

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");
  }
}
```

21.6.2. 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.

21.7. Serialization

Serialization on the ErraiBus supports serialization within the same scope and limitations as the default GWT RPC serialization rules. In order to expose your domain objects to the bus so they can be exported across the bus, you must annotate them with the `org.jboss.errai.bus.server.annotations.ExposeEntity` annotation. The presence of this annotation will cause Errai's GWT compiler extensions to generate marshall/demarshall stubs for the annotated objects at compile-time.

For example:

```
@ExposeEntity
public class User implements java.io.Serializable {
  private int userId;

  public int getUserId() {
    return userId;
  }

  public void setUserId(int userId) {
```

```
    this.userId = userId;
  }
  [...]
}
```

Note

All exposed entities must follow Java Bean convensions, and must be in the classpath both at compile-time and at runtime. Compile-time access to the entities is required since the creation of the marshalling/demarshalling proxies involves code generation.

21.7.1. Serialization of external types

It may not be possible to annotate certain types you wish to expose to the bus for serialization if the entities are located in a third-party library that you do not maintain. As such, you can explicitly indicate in the configuration that you would like to have this entities made available by declaring them in the `ErraiApp.properties` of any module.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.8. Wiring server side components

By default, ErraiBus uses Google Guice to wire components. However, we plan on standardizing on *JSR-330 Dependency Injection* specification in the near future. 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.

21.9. Bus Configuration

This section contains information on configuring the server-side bus.

21.9.1. web.xml and appserver configuration

Depending on what application server you are deploying on, you must provide an appropriate servlet implementation if you wish to use true, asynchronous I/O. See _section 6.5 _ for information on the available servlet implementations.

Here's a sample web.xml file:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <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>

  <context-param>
    <param-name>errai.properties</param-name>
    <param-value>/WEB-INF/errai.properties</param-value>
  </context-param>

  <context-param>
    <param-name>login.config</param-name>
    <param-value>/WEB-INF/login.config</param-value>
  </context-param>

  <context-param>
    <param-name>users.properties</param-name>
    <param-value>/WEB-INF/users.properties</param-value>
  </context-param>

</web-app>
```

21.9.2. ErraiService.properties

he ErraiService.properties file contains basic configuration for the bus itself.

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

##
## Specify the Authentication/Authorization Adapter to use
##
#errai.authentication_adapter=org.jboss.errai.persistence.server.security.HibernateAuthenticati
#errai.authentication_adapter=org.jboss.errai.bus.server.security.auth.JAASAdapter

##
## This property indicates whether or not authentication is required for all
 communication with the
bus. Set this
## to 'true' if all access to your application should be secure.
##
#errai.require_authentication_for_all=true
```

21.9.2.1. errai.dispatcher.implementation

The `errai.dispatcher_implementation` defines, as it's name quite succinctly implies, 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 section on Dispatchers for more information about the differences between the two.

21.9.2.2. 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.

21.9.2.3. errai.async.worker_timeout

Specifies the total amount of a time (in seconds) a service has to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value does not have an effect if you are using the SimpleDispatcher.

21.9.2.4. errai.authentication_adapter

Specifies the authentication modelAdapter the bus should use for determining whether calls should be serviced based on authentication and security principles.

21.9.2.5. errai.require_authentication_for_all

Indicates whether or not the bus should always require the use of authentication for all requests inbound for the bus. If this is turned on, an authentication model adapter must be defined, and any user must be authenticated before the bus will deliver any messages from the client to any service.

21.9.2.6. errai.auto_discover_services

A boolean indicating whether or not the Errai bootstrapper should automatically scan for services.

21.9.2.7. errai.auto_load_extensions

A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions.

21.9.3. ErraiApp.properties

The ErraiApp.properties acts as a marker file. When it is detected inside a JAR or at the top of any classpath, the subdirectories are scanned for deployable components. As such, all Errai application modules in a project should contain an ErraiApp.properties at the root of all classpaths that you wish to be scanned.

The file can also include explicitly declared serializable types (such as those from third-party code) that cannot be annotated for serialization. (See the section on serialization for more details)

errai.bus.serializableTypesDefines a list of serializable types to expose to the bus.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.10. Dispatchers

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.

21.10.1. 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.

21.10.2. 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.

21.11. Servlet Implementations

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 inlcuded implementations are packaged at: `org.jboss.errai.bus.server.servlet`

21.11.1. DefaultBlockingServlet

This is a universal, completely servlet spec (2.0) compliant, Servlet implementation. It provides purely synchronous request handling and should work in virtually any servlet container, unless there are restrictions on putting threads into sleep states.

21.11.2. TomcatCometServlet

The Tomcat AIO implementation of our servlet allows Errai to take advantage of Tomcat's event-based AIO APIs to improve scalability and reduce thread usage. The use of this implementation is dependant on the Tomcat container being configured to support AIO using either it's NIO or APR connectors. This servlet will NOT work with the regular HTTP and AJP connectors.

21.11.3. 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.

21.11.4. JBossCometServlet

The JBoss Comet support utilizes the JBoss Web AIO APIs (AS 5.0 and AS 6.0) to improve scalability and reduce thread usage. The HTTP, NIO, and AJP connectors are not supported. Use of this implementation requires use of the APR (Apache Portable Runtime).

21.11.5. GrizzlyCometServlet

Support for the comet API in the Grizzy HTTP server (used in Glassfish).

21.11.6. WeblogicAsyncServlet

Support for the Weblogic asynchronous APIs.

21.12. Debugging Errai Applications

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 21.1. TODO InformalFigure image title empty

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 21.2. TODO InformalFigure image title empty

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.

21.13. What is Errai Bus?

ErraiBus forms the backbone of the Errai framework's approach to application design. Most importantly, it 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 from building instant messaging clients, stock tickers, to monitoring instruments. There's no more messing with RPC APIs, or unweildy AJAX or COMET frameworks. We've built it all in to one, consice messaging framework. It's single-paradigm, and it's fun to work with.

21.14. Messaging

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

21.14.1. Messaging Overview

It's important to understand the concept of how messaging works in ErraiBus. 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. In fact, 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 with the GWT client context, such as a way to advertise and expose components dynamically, to get around the lack of reflection in GWT.

In fact, ErraiBus was originally designed to run completely within the client but quickly evolved into having the capabilities it now has today. So keep that in mind when you run up against problems in the client space that could benefit from runtime federation.

21.14.2. MessageBuilder API

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 look at some use cases first.

Sending Messages with the Client BusIn 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`

Note

An astute observer will note that access to the `RequestDispatcher` differs within client code and server code. Because the client code does not run within a container, access to the `RequestDispatcher` and `MessageBus` is statically accessed using the `ErraiBus.get()` and `ErraiBus.getDispatcher()` methods. The server-side code, conversely, runs inside a dependency container for managing components. It currently uses Guice as the default container, but will more generically support the JSR-330 specification in the future.

Recieving Messages on the Server Bus / Server ServicesEvery 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!");
    }
  }
```

He 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.

Sending Messages with the Server BusIn 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 disaptcher) {
    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` .

Receiving Messages on the Client Bus/ Client ServicesMessages can be received asynchronously and arbitriraily 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 local 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.

ConversationsConversations 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 (2.4) and this is the use of the `createConversation()}}method with {{MessageBuilder` .

21.14.3. Single-Response Conversations & Psuedo-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.

21.14.4. 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.

21.14.5. 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.

21.14.6. 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.

Relay ServicesThe 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 *Section 21.14.13, "Message Routing Information"*

21.14.7. 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.

The utility class `org.jboss.errai.bus.server.util.ServerBusUtils` contains a utility method for extracting the String-based SessionID which is used to identify the message queue associated with any particular client. You may use this method to extract the `SessionID` from a message so that you may use it for routing. For example:

```
...
  public void callback(Message message) {
    String sessionId = ServerBusUtils.getSessionId(message);

    // 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.

Now you're routing from client-to-client!

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

21.14.8. Messaging Overview

It's important to understand the concept of how messaging works in ErraiBus. 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. In fact, 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 with the GWT client context, such as a way to advertise and expose components dynamically, to get around the lack of reflection in GWT.

In fact, ErraiBus was originally designed to run completely within the client but quickly evolved into having the capabilities it now has today. So keep that in mind when you run up against problems in the client space that could benefit from runtime federation.

21.14.9. MessageBuilder API

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 look at some use cases first.

Sending Messages with the Client BusIn 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`

Note

An astute observer will note that access to the `RequestDispatcher` differs within client code and server code. Because the client code does not run within a container, access to the `RequestDispatcher` and `MessageBus` is statically

> accessed using the `ErraiBus.get()` and `ErraiBus.getDispatcher()` methods. The server-side code, conversely, runs inside a dependency container for managing components. It currently uses Guice as the default container, but will more generically support the JSR-330 specification in the future.

Recieving Messages on the Server Bus / Server ServicesEvery 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!");
    }
  }
```

He 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.

Sending Messages with the Server BusIn 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 disaptcher) {
    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` .

Receiving Messages on the Client Bus/ Client ServicesMessages can be received asynchronously and arbitriraily 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 local 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.

ConversationsConversations 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 (2.4) and this is the use of the `createConversation()}}method with {{MessageBuilder` .

21.14.10. Single-Response Conversations & Psuedo-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.

21.14.11. 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.

21.14.12. 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.

21.14.12.1. 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.

Relay ServicesThe 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 *Section 21.14.13, “Message Routing Information”*

21.14.12.2. 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.

Relay ServicesThe 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 *Section 21.14.13, “Message Routing Information”*

21.14.13. 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.

The utility class `org.jboss.errai.bus.server.util.ServerBusUtils` contains a utility method for extracting the String-based SessionID which is used to identify the message queue associated with any particular client. You may use this method to extract the `SessionID` from a message so that you may use it for routing. For example:

```
...
  public void callback(Message message) {
    String sessionId = ServerBusUtils.getSessionId(message);

    // 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.

Now you're routing from client-to-client!

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

21.15. 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);
```

A little nicer.

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

21.16. 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.
```

21.17. 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);
```

21.18. 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 to 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.

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 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.

Making callsCalling 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 correpond 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.

21.19. 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.

21.19.1. Scopes

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

21.19.1.1. 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");
  }
}
```

21.19.1.2. Session Scopes

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");
  }
}
```

21.19.2. 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.

21.19.3. 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.

21.19.4. Scopes

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

21.19.4.1. 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");
  }
}
```

21.19.4.2. Session Scopes

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");
  }
}
```

21.19.4.3. 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");
  }
}
```

21.19.4.4. 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");
  }
}
```

21.20. Serialization

Serialization on the ErraiBus supports serialization within the same scope and limitations as the default GWT RPC serialization rules. In order to expose your domain objects to

the bus so they can be exported across the bus, you must annotate them with the `org.jboss.errai.bus.server.annotations.ExposeEntity` annotation. The presence of this annotation will cause Errai's GWT compiler extensions to generate marshall/demarshall stubs for the annotated objects at compile-time.

For example:

```
@ExposeEntity
public class User implements java.io.Serializable {
  private int userId;

  public int getUserId() {
    return userId;
  }

  public void setUserId(int userId) {
    this.userId = userId;
  }
  [...]
}
```

Note

All exposed entities must follow Java Bean convensions, and must be in the classpath both at compile-time and at runtime. Compile-time access to the entities is required since the creation of the marshalling/demarshalling proxies involves code generation.

21.20.1. Serialization of external types

It may not be possible to annotate certain types you wish to expose to the bus for serialization if the entities are located in a third-party library that you do not maintain. As such, you can explicitly indicate in the configuration that you would like to have this entities made available by declaring them in the `ErraiApp.properties` of any module.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.20.2. Serialization of external types

It may not be possible to annotate certain types you wish to expose to the bus for serialization if the entities are located in a third-party library that you do not maintain. As such, you can explicitly

indicate in the configuration that you would like to have this entities made available by declaring them in the `ErraiApp.properties` of any module.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.21. Wiring server side components

By default, ErraiBus uses Google Guice to wire components. However, we plan on standardizing on *JSR-330 Dependency Injection* specification in the near future. 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.

21.22. Bus Configuration

This section contains information on configuring the server-side bus.

21.22.1. web.xml and appserver configuration

Depending on what application server you are deploying on, you must provide an appropriate servlet implementation if you wish to use true, asynchronous I/O. See _section 6.5 _ for information on the available servlet implementations.

Here's a sample web.xml file:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <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>
```

```
  <context-param>
    <param-name>errai.properties</param-name>
    <param-value>/WEB-INF/errai.properties</param-value>
  </context-param>

  <context-param>
    <param-name>login.config</param-name>
    <param-value>/WEB-INF/login.config</param-value>
  </context-param>

  <context-param>
    <param-name>users.properties</param-name>
    <param-value>/WEB-INF/users.properties</param-value>
  </context-param>

</web-app>
```

21.22.2. ErraiService.properties

he ErraiService.properties file contains basic configuration for the bus itself.

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
```

```
##
## Specify the Authentication/Authorization Adapter to use
##
#errai.authentication_adapter=org.jboss.errai.persistence.server.security.HibernateAuthenticati
#errai.authentication_adapter=org.jboss.errai.bus.server.security.auth.JAASAdapter

##
## This property indicates whether or not authentication is required for all
 communication with the
bus. Set this
## to 'true' if all access to your application should be secure.
##
#errai.require_authentication_for_all=true
```

21.22.2.1. errai.dispatcher.implementation

The `errai.dispatcher_implementation` defines, as it's name quite succinctly implies, 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 section on Dispatchers for more information about the differences between the two.

21.22.2.2. 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.

21.22.2.3. errai.async.worker_timeout

Specifies the total amount of a time (in seconds) a service has to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value does not have an effect if you are using the SimpleDispatcher.

21.22.2.4. errai.authentication_adapter

Specifies the authentication modelAdapter the bus should use for determining whether calls should be serviced based on authentication and security principles.

21.22.2.5. errai.require_authentication_for_all

Indicates whether or not the bus should always require the use of authentication for all requests inbound for the bus. If this is turned on, an authentication model adapter must be defined, and any user must be authenticated before the bus will deliver any messages from the client to any service.

21.22.2.6. errai.auto_discover_services

A boolean indicating whether or not the Errai bootstrapper should automatically scan for services.

21.22.2.7. errai.auto_load_extensions

A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions.

21.22.3. ErraiApp.properties

The ErraiApp.properties acts as a marker file. When it is detected inside a JAR or at the top of any classpath, the subdirectories are scanned for deployable components. As such, all Errai application modules in a project should contain an ErraiApp.properties at the root of all classpaths that you wish to be scanned.

The file can also include explicitly declared serializable types (such as those from third-party code) that cannot be annotated for serialization. (See the section on serialization for more details)

errai.bus.serializableTypesDefines a list of serializable types to expose to the bus.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.22.4. ErraiApp.properties

The ErraiApp.properties acts as a marker file. When it is detected inside a JAR or at the top of any classpath, the subdirectories are scanned for deployable components. As such, all Errai application modules in a project should contain an ErraiApp.properties at the root of all classpaths that you wish to be scanned.

The file can also include explicitly declared serializable types (such as those from third-party code) that cannot be annotated for serialization. (See the section on serialization for more details)

errai.bus.serializableTypesDefines a list of serializable types to expose to the bus.

```
errai.bus.serializableTypes=org.foo.Foo \
                            org.bar.Bar \
                            org.foobie.Foobie
```

21.22.5. ErraiService.properties

he ErraiService.properties file contains basic configuration for the bus itself.

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

##
## Specify the Authentication/Authorization Adapter to use
##
#errai.authentication_adapter=org.jboss.errai.persistence.server.security.HibernateAuthenticati
#errai.authentication_adapter=org.jboss.errai.bus.server.security.auth.JAASAdapter

##
## This property indicates whether or not authentication is required for all
 communication with the
bus. Set this
## to 'true' if all access to your application should be secure.
##
#errai.require_authentication_for_all=true
```

21.22.5.1. errai.dispatcher.implementation

The `errai.dispatcher_implementation` defines, as it's name quite succinctly implies, 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 section on Dispatchers for more information about the differences between the two.

21.22.5.2. 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.

21.22.5.3. errai.async.worker_timeout

Specifies the total amount of a time (in seconds) a service has to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value does not have an effect if you are using the SimpleDispatcher.

21.22.5.4. errai.authentication_adapter

Specifies the authentication modelAdapter the bus should use for determining whether calls should be serviced based on authentication and security principles.

21.22.5.5. errai.require_authentication_for_all

Indicates whether or not the bus should always require the use of authentication for all requests inbound for the bus. If this is turned on, an authentication model adapter must be defined, and any user must be authenticated before the bus will deliver any messages from the client to any service.

21.22.5.6. errai.auto_discover_services

A boolean indicating whether or not the Errai bootstrapper should automatically scan for services.

21.22.5.7. errai.auto_load_extensions

A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions.

21.22.5.8. 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.

21.22.5.9. errai.async.worker_timeout

Specifies the total amount of a time (in seconds) a service has to finish processing an incoming message before the pool interrupts the thread and returns an error. Adjusting this value does not have an effect if you are using the SimpleDispatcher.

21.22.5.10. errai.authentication_adapter

Specifies the authentication modelAdapter the bus should use for determining whether calls should be serviced based on authentication and security principles.

21.22.5.11. errai.auto_discover_services

A boolean indicating whether or not the Errai bootstrapper should automatically scan for services.

21.22.5.12. errai.auto_load_extensions

A boolean indicating whether or not the Errai bootstrapper should automatically scan for extensions.

21.22.5.13. errai.dispatcher.implementation

The `errai.dispatcher_implementation` defines, as it's name quite succinctly implies, 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 section on Dispatchers for more information about the differences between the two.

21.22.5.14. errai.require_authentication_for_all

Indicates whether or not the bus should always require the use of authentication for all requests inbound for the bus. If this is turned on, an authentication model adapter must be defined, and any user must be authenticated before the bus will deliver any messages from the client to any service.

21.22.6. web.xml and appserver configuration

Depending on what application server you are deploying on, you must provide an appropriate servlet implementation if you wish to use true, asynchronous I/O. See _section 6.5 _ for information on the available servlet implementations.

Here's a sample web.xml file:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <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>

  <context-param>
```

```
    <param-name>errai.properties</param-name>
    <param-value>/WEB-INF/errai.properties</param-value>
  </context-param>

  <context-param>
    <param-name>login.config</param-name>
    <param-value>/WEB-INF/login.config</param-value>
  </context-param>

  <context-param>
    <param-name>users.properties</param-name>
    <param-value>/WEB-INF/users.properties</param-value>
  </context-param>

</web-app>
```

21.23. Dispatchers

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.

21.23.1. 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.

21.23.2. 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.

21.23.3. 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.

21.23.4. 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.

21.24. Servlet Implementations

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 inlcuded implementations are packaged at: `org.jboss.errai.bus.server.servlet`

21.24.1. DefaultBlockingServlet

This is a universal, completely servlet spec (2.0) compliant, Servlet implementation. It provides purely synchronous request handling and should work in virtually any servlet container, unless there are restrictions on putting threads into sleep states.

21.24.2. TomcatCometServlet

The Tomcat AIO implementation of our servlet allows Errai to take advantage of Tomcat's event-based AIO APIs to improve scalability and reduce thread usage. The use of this implementation is dependant on the Tomcat container being configured to support AIO using either it's NIO or APR connectors. This servlet will NOT work with the regular HTTP and AJP connectors.

21.24.3. 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.

21.24.4. JBossCometServlet

The JBoss Comet support utilizes the JBoss Web AIO APIs (AS 5.0 and AS 6.0) to improve scalability and reduce thread usage. The HTTP, NIO, and AJP connectors are not supported. Use of this implementation requires use of the APR (Apache Portable Runtime).

21.24.5. GrizzlyCometServlet

Support for the comet API in the Grizzy HTTP server (used in Glassfish).

21.24.6. WeblogicAsyncServlet

Support for the Weblogic asynchronous APIs.

21.24.7. DefaultBlockingServlet

This is a universal, completely servlet spec (2.0) compliant, Servlet implementation. It provides purely synchronous request handling and should work in virtually any servlet container, unless there are restrictions on putting threads into sleep states.

21.24.8. GrizzlyCometServlet

Support for the comet API in the Grizzy HTTP server (used in Glassfish).

21.24.9. JBossCometServlet

The JBoss Comet support utilizes the JBoss Web AIO APIs (AS 5.0 and AS 6.0) to improve scalability and reduce thread usage. The HTTP, NIO, and AJP connectors are not supported. Use of this implementation requires use of the APR (Apache Portable Runtime).

21.24.10. 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.

21.24.11. TomcatCometServlet

The Tomcat AIO implementation of our servlet allows Errai to take advantage of Tomcat's event-based AIO APIs to improve scalability and reduce thread usage. The use of this implementation is dependant on the Tomcat container being configured to support AIO using either it's NIO or APR connectors. This servlet will NOT work with the regular HTTP and AJP connectors.

21.24.12. WeblogicAsyncServlet

Support for the Weblogic asynchronous APIs.

21.25. Debugging Errai Applications

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 21.3. TODO InformalFigure image title empty

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 21.4. TODO InformalFigure image title empty

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.

Development Proxy

Proxied access to external containersUsually GWT developement happens in hosted mode and then, later on, the GWT app is turned into a webapp (*.war) that can be deployed on a target container (app server, servlet engine). This works quiet well for closed systems that don't depend on additional resources the target container provides. A typical resource would be a DataSource for access to a relational database.

Instead of pulling these resources into the hosted mode servlet engine (jetty, read-only JNDI) or creating mock objects for any resources that cannot be run in hosted mode, we offer you a much more simple way to work with external resources: Simply proxy all requests that occur in hosted mode to an external target container:

Figure 22.1. Development Proxy

The proxy is implemented a yet another servlet that you need to add to the web.xml that's being sed in hosted mode:

```
<servlet>
                    <servlet-name>erraiProxy</servlet-name>
                    <description>Errai Proxy</description>
                <servlet-class>org.jboss.errai.tools.proxy.XmlHttpProxyServlet</
servlet-class>
                    <init-param>
                    <param-name>config.name</param-name>
                    <param-value>errai-proxy.json</param-value> (1)
                    </init-param>
                    <load-on-startup>1</load-on-startup>
                    </servlet>

                    <servlet-mapping>
                    <servlet-name>erraiProxy</servlet-name>
                    <url-pattern>/app/proxy/*</url-pattern> (2)
                    </servlet-mapping>
```

1. Proxy configuration. See details below.

2. The proxy url pattern convetion. The bus bootstraps on this URL.

The web.xml proxy declaration contains two notable elements: A reference to the proxy configuration file and a URL pattern, where the proxy can found. While the later shouldn't be changed (the bus bootstraps on this URL), you need to change the proxy config according to your needs:

```
{"xhp": {
                    "version": "1.1",
                    "services": [
                    {"id": "default",
                    "url":"http://127.0.0.1:8080/my-gwt-app/in.erraiBus",
                    "passthrough":true
                    },
                    ]
                    }
                    }
```

You would need to _change the host, port and webcontext _ ('my-gwt-app' in this case) to reflect the location of the external container. 'passthrough' simply means that any request to 'proxy/in.erraiBus' will go to 'container/my-gwt-app/in.erraiBus'. This already indicates that you need to have the server side part of your GWT application, already running on the target container. The most simple way to achieve this, is to build a the complete webapp, deploy it and ignore the UI parts that may be available on the server.

Errai IOC

The Errai IOC (Inversion-of-Control) module is a central feature of the Errai Framework, providing client-side service location, component lifecycle, and injection services. The framework is a modular and extensible system for building reusable client-side components.

23.1. Dependency Injection

The core Errai IOC module implements a subset of the *JSR-330 Dependency Injection [http://download.oracle.com/otndocs/jcp/dependency_injection-1.0-final-oth-JSpec/]* 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.

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` [http://download.oracle.com/javaee/6/api/javax/inject/Inject.html] 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());
  }
```

```
}
```

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.

`@Inject` [http://download.oracle.com/javaee/6/api/javax/inject/Inject.html]

23.2. 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`* [http://download.oracle.com/javaee/6/api/javax/inject/Provider.html] , which is scanned for an auto-discovered by the container.

23.2.1. level Providers

A `Provider` is essentially a factory which produces dependent types in the container, which 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 three default top-level providers:

- `org.jboss.errai.ioc.client.api.builtin.MessageBusProvider` : Makes an instance of `MessageBus` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.RequestDispatchProvider` : Makes an instance of the `RequestDispatcher` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.ConsumerProvider` : Makes event `Consumer<?>` 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 treated as regular beans. And as such may inject dependencies – particularly from other top-level providers – as necessary.

23.3. 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`* `[http://download.oracle.com/javaee/6/api/javax/inject/Provider.html]` , which is scanned for an auto-discovered by the container.

23.3.1. level Providers

A `Provider` is essentially a factory which produces dependent types in the container, which 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 three default top-level providers:

- `org.jboss.errai.ioc.client.api.builtin.MessageBusProvider` : Makes an instance of `MessageBus` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.RequestDispatchProvider` : Makes an instance of the `RequestDispatcher` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.ConsumerProvider` : Makes event `Consumer<?>` 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 treated as regular beans. And as such may inject dependencies – particularly from other top-level providers – as necessary.

23.3.2. level Providers

A `Provider` is essentially a factory which produces dependent types in the container, which 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 three default top-level providers:

- `org.jboss.errai.ioc.client.api.builtin.MessageBusProvider` : Makes an instance of `MessageBus` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.RequestDispatchProvider` : Makes an instance of the `RequestDispatcher` available for injection.
- `org.jboss.errai.ioc.client.api.builtin.ConsumerProvider` : Makes event `Consumer<?>` 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 treated as regular beans. And as such may inject dependencies – particularly from other top-level providers – as necessary.

23.4. Dependency Injection

The core Errai IOC module implements a subset of the *JSR-330 Dependency Injection [http://download.oracle.com/otndocs/jcp/dependency_injection-1.0-final-oth-JSpec/]* 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.

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* [http://download.oracle.com/javaee/6/api/javax/inject/Inject.html] 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());
  }
}
```

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.

@Inject [http://download.oracle.com/javaee/6/api/javax/inject/Inject.html]

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*

Appendix A. Revision History

Revision History

Revision ToDo 0-0 ToDo Wed Jan 19 2011 ToDo DudeToDo McPants<ToDo Dude.McPants@example.com>

ToDo Initial creation of book