Errai

Errai Reference Guide

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. 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.

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.

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. Messaging API Basics

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

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

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

2.2.1. Sending Messages with the Client Bus

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

```
public class HelloWorld implements EntryPoint {

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

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

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

        [...]
      }
   }
}
```

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

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

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. See the section on Errai IOC and Errai CDI for using ErraiBus from a client-side container.

2.2.2. Recieving Messages on the Server Bus / Server Services

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

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

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.

2.2.3. Sending Messages with the Server Bus

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

```
@Service
public class HelloWorldService implements MessageCallback {

  private RequestDispatcher dispatcher;

  @Inject
  public HelloWorldService(RequestDispatcher 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` .

2.2.4. Receiving Messages on the Client Bus/ Client Services

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

```
public class HelloWorld implements EntryPoint {

  private MessageBus bus = ErraiBus.get();

  public void onModuleLoad() {
     [...]

     /**
      * Declare a 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. Handling Errors

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

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

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

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

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

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

MessageBuilder.createMessage()
  .toSubject("HelloWorldService")
  .signalling()
  .with("msg", "Hi there!")
  .errorsHandledBy(error)
  .sendNowWith(dispatcher);
```

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

Errai further provides a subject to subscribe to for handling global errors on the client (such as a disconnected server bus or an invalid response code) that occur outside a regular application message exchange. Subscribing to this subject is useful to detect errors early (e.g. due to failing heartbeat requests). A use case that comes to mind here is activating your application's offline mode.

```
bus.subscribe(DefaultErrorCallback.CLIENT_ERROR_SUBJECT, new MessageCallback() {
  @Override
  public void callback(Message message) {
    try {
      caught = message.get(Throwable.class, MessageParts.Throwable);
```

```
      throw caught;
    }
    catch(TransportIOException e) {
       // thrown in case the server can't be reached or an unexpected status
 code was returned
    }
    catch (Throwable throwable) {
      // handle system errors (e.g response marshalling errors) - that of course
 should never happen :)
    }
  }
});
```

2.4. 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.

2.5. Broadcasting

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

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

```
    .noErrorHandling().sendGlobalWith(dispatcher);
```

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

2.6. Client-to-Client Communication

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

2.6.1. Relay Services

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

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

While you can probably imagine simply creating a broadcast-like service which accepts a message from one client and broadcasts it to the rest of the world, it may be less clear how to go about routing from one particular client to another particular client, so we'll focus on that problem. This is covered in *Section 2.10, “Message Routing Information”*

2.7. Asynchronous Message Tasks

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

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

```
MessageBuilder.createConversation(msg)
    .toSubject("FunSubject")
    .signalling()
    .noErrorHandling()
    .replyDelayed(TimeUnit.SECONDS, 5); // sends the message after 5 seconds.
```

or

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

2.8. Repeating Tasks

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

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

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

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

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

```
AsyncTask task = MessageBuilder.createConversation(message)
  .toSubject("TimeChannel").signalling()
  .withProvided(TimeServerParts.TimeString, new ResourceProvider<String>() {
     public String get() {
       return String.valueOf(System.currentTimeMillis());
     }
   }).defaultErrorHandling().replyRepeating(TimeUnit.MILLISECONDS, 100);

    ...
```

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

2.9. 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.

2.10. Message Routing Information

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

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

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

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

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

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

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

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

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

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

2.11. Queue Sessions

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

2.11.1. Lifecycle

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

2.11.2. Scopes

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

2.11.2.1. Session Scope

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

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

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

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

2.11.2.2. Local Scope

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

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

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

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

2.12. Client Logging and Error Handling

2.13. Wire Protocol (J.REP)

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

2.13.1. Payload Structure

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

Example 2.1. Figure 1 - Example J.REP Payload

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

In **Figure 1** , we see an example of a J.REP payload containing two messages. One bound for an endpoint named `"SomeEndpoint"` and the other bound for the endpoint `"SomeOtherEndpoint"` . They both include a payload element `"Value"` which contain strings. Let's take a look at the anatomy of an individual message.

Example 2.2. Figure 2 - An J.REP Message

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

The message shown in **Figure 2** shows a very vanilla J.REP message. The keys of the JSON Object represent individual *message parts* , with the values representing their corresponding values. The standard J.REP protocol encompasses a set of standard message parts and values, which for the purposes of this specification we'll collectively refer to as the protocol verbs.

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

Part	Required	JSON Type	Description
`ToSubject`	Yes	String	Specifies the subject within the bus, and its federation, which the message should be routed to.
`CommandType`	No	String	Specifies a command verb to be transmitted to the receiving subject. This is an optional part of a message contract, but is required for using management services
`ReplyTo`	No	String	Specifies to the receiver what subject it should reply to in response to this message.
`Value`	No	Any	A recommended but not required standard payload part for sending data to services
`PriorityProcessing`	No	Number	A processing order salience attribute. Messages which specify priority processing will be processed first if they are competing for resources with other messages in flight. Note: the current version of ErraiBus only supports two salience levels (0 and >1). Any non-zero salience in ErraiBus will be given the same priority relative to 0 salience messages

Part	Required	JSON Type	Description
`ErrorMessage`	No	String	An accompanying error message with any serialized exception
`Throwable`	No	Object	If applicable, an encoded object representing any remote exception that was thrown while dispatching the specified service

2.13.1.1. Built-in Subjects

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

Subject	Description
`ClientBus`	The self-hosted message bus endpoint on the client
`ServerBus`	The self-hosted message bus endpoint on the server
`ClientBusErrors`	The standard error receiving service for clients

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

2.13.2. Message Routing

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

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

Figure 2.1. Figure 3 - Topology of a J.REP Messaging Federation

In **Figure 3** , is is possible for *Client A* to send messages to the subjects *ServiceA* and *ServiceB* . But it is not possible to address messages to *ServiceC* . Conversely, *Client A* can address messages to *ServiceC* and *ServiceB* , but not *ServiceA* .

2.13.3. Bus Management and Handshaking Protocols

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

2.13.3.1. ServerBus and ClientBus commands

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

Table 2.1. Message Parts for Bus Commands:

Command / Verb	Message Parts	Description
`ConnectToQueue`	N/A	The first message sent by a connecting client to begin the handshaking process.
`CapabilitiesNotice`	`CapabilitiesFlags`	A message sent by one bus to another to notify it of its capabilities during handshake (for instance long polling or websockets)
`FinishStateSync`	N/A	A message sent from one bus to another to indicate that it has now provided all necessary information to the counter-party bus to establish the federation. When both buses have sent this message to each other, the federation is considered active.
`RemoteSubscribe`	`Subject` *or* `SubjectsList`	A message sent to the remote bus to notify it of a service or set of services which it is capable of routing to.
`RemoteUnsubscribe`	`Subject`	A message sent to the remote bus to notify it that a service is no longer available.
`Disconnect`	`Reason`	A message sent to a server bus from a client bus to indicate that it wishes to disconnect and defederate. Or, when sent from the client

Command / Verb	Message Parts	Description
		to server, indicates that the session has been terminated.
`SessionExpired`	N/A	A message sent to a client bus to indicate that its messages are no longer being routed because it no longer has an active session
`Heartbeat`	N/A	A message sent from one bus to another periodically to indicate it is still active.

Part	Required	JSON Type	Description
`CapabilitiesFlags`	Yes	String	A comma delimited string of capabilities the bus is capable of us
`Subject`	Yes	String	The subject to subscribe or unsubscribe from
`SubjectsList`	Yes	Array	An array of strings representing a list of subjects to subscribe to

Dependency Injection

The core Errai IOC module implements 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());
  }
}
```

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

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

Best Practices

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

3.1. Container Wiring

In contrast to *Gin* [http://code.google.com/p/google-gin/] , the Errai IOC container does not provide a programmatic way of creating and configuring injectors. Instead, container-level binding rules are defined by implementing a *`Provider`* `[http://download.oracle.com/javaee/6/api/javax/inject/Provider.html]` , which is scanned for an auto-discovered by the container.

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.

3.2. Wiring server side components

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

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

3.3. Scopes

Out of the box, the IOC container supports two bean scopes, `@Singleton` and `@EntryPoint` . Both of these scopes are roughly the same semantics. Since the client code is single-user and single-scope in nature, there is no general support for passivation scopes in the client.

Note

The Errai CDI extension module adds support for the `@Dependent` pseudo-scope and `@ApplicationScope` .

3.4. Built-in Extensions

3.4.1. Bus Services

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

3.4.1.1. @Service

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

Example 3.1. A simple message receiving service

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

Or like so ...

Example 3.2. Mapping a callback from a field of a bean

```
@Singleton
```

```
public class MyAppBean {
   @Service("MyService")
   private final MessageCallback myService = new MesageCallback() {
     public void callback(Message message) {
       // ... //
     }
   }
}
```

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

3.4.1.2. @Local

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

Example 3.3. A local only service

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

3.4.1.3. Lifecycle Impact of Services

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

3.4.2. Client Components

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

3.4.2.1. MessageBus

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

Example 3.4. Injecting a MessageBus

```
@Inject MessageBus bus;
```

3.4.2.2. RequestDispatcher

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

Example 3.5. Injecting a RequestDispatcher

```
@Inject RequestDispatcher dispatcher;
```

3.4.2.3. Caller<?>

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

For example:

Example 3.6. An example Caller<?> proxy

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

  // ... ///

  @UiHandler("button")
  public void onButtonClick(ClickHandler handler) {
    rpcCaller.call(new RemoteCallback<Void>() {
      public void callback(Void void) {
      }
    ).callSomeMethod();
  }
}
```

The above code shows the injection of a proxy for the RPC remote interface, `MyRpcInterface` . For more information on defining RPC proxies see *Chapter 5, Remote Procedure Calls (RPC)* and *Section 7.1, “Creating Requests”* in Errai JAX-RS.

3.4.3. Lifecycle Tools

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

3.4.3.1. Controlling Startup

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

Example 3.7. Using an InitBallot to Control Startup

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

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

    ballot.voteForInit();
  }
}
```

3.4.3.2. Performing Tasks After Initialization

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

Example 3.8. Using @AfterInitialization to do something after startup

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

```
}
```

3.5. Client-Side Bean Manager

It may be necessary at times to obtain instances of beans managed by Errai IOC from outside the container managed scope or creating a hard dependency from your bean. Errai IOC provides a simple client-side bean manager for handling these scenarios: `org.jboss.errai.ioc.client.container.IOCBeanManager` .

As you might expect, you can inject the bean manager into any of your managed beans.

Example 3.9. Injecting the client-side bean manager

```
public MyManagedBean {
  @Inject IOCBeanManager manager;

  // class body
}
```

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

3.5.1. Looking up beans

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

Example 3.10. Example lookup of a bean

```
public MyManagedBean {
  @Inject IOCBeanManager manager;

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

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

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

Example 3.11. Looking up beans with qualifiers

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

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

// pass qualifiers to IOCBeanManager.lookupBean
IOCBean<SimpleInterface> bean = beanManager.lookupBean(SimpleBean.class, qual, qual2);
```

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

3.5.2. Availability of beans

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

3.6. Alternatives and Mocks

3.6.1. Alternatives

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

Consider the following example:

```
@Singleton @Alternative
```

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

and

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

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

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

  // ... //
}
```

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

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

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

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

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

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

3.6.2. Test Mocks

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

Consider the following:

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

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

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

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

The `@TestMock` annotation can also be used to specify alternative providers during test execution. For example, it can be used to mock a `Caller<T>` . `Callers` are used to invoke RPC or JAX-RS endpoints. During tests you might want to replace theses callers with mock implementations. For details on providers see *Section 3.1, “Container Wiring”* .

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

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

```
}
```

3.7. Bean Lifecycle

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

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

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

Important

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

3.7.1. Destruction of Beans

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

Example 3.12. Destruction of bean

```
public MyManagedBean {
  @Inject IOCBeanManager manager;

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

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

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

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

Important

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

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

Example 3.13. SimpleBean.class

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

   public AnotherBean getAnotherBean() {
     return anotherBean;
   }

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

Example 3.14. Destroying bean from subgraph

```
public MyManagedBean {
  @Inject IOCBeanManager manager;

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

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

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

3.7.1.1. Disposers

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

For instance:

Example 3.15. Destroying bean with disposer

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

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

Marshalling

Errai includes a comprehensive marshalling framework which permits the serialization of domain objects between the browser and the server. From the perspective of GWT, this is a complete replacement for the provided GWT serialization facilities and offers a great deal more flexibility. You are be able to map both application-specific domain model, as well as preexisting model, including model from third-party libraries using the custom definitions API.

4.1. Mapping Your Domain

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

4.1.1. @Portable

The simplest and most straight-forward way to map your domain entities is to annotate it with the `org.jboss.errai.common.client.api.annotations.Portable` annotation. Doing so will cause the marshalling system to discover the entities at both compile-time and runtime to produce the marshalling code and definitions to marshal and de-marshal the objects.

The mapping strategy that will be used depends on how much information you provide about your model up-front. If you simply annotate a domain object with `@Portable` and do nothing else, the marshalling system will use and exhaustive strategy to determine how to construct and deconstruct the object.

Let's take a look at how this works.

4.1.1.1. A Simple Entity

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

  public Person() {
  }

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

  public String getName() {
    return name;
  }
```

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

This is a pretty vanilla domain object. Note the default, public, no-argument constructor. In this case, it will be necessary to have one explicitly declared. But notice we have no setters. In this case, the marshaler will rely on private field access to write the values on each side of the marshalling transaction. For simple domain objects, this is both nice and convenient. But you may want to make the class immutable and have a constructor enforce invariance. See the next section for that.

4.1.1.2. Immutable Objects

Immutability is almost always a good practice, and the marshalling system provides you a straight forward way to tell it how to marshal and de-marshal objects which enforce an immutable contract. Let's modify our example from the previous section.

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

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

  public String getName() {
    return name;
  }

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

Here we have set both of the class fields final. By doing so, we had to remove our default constructor. But that's okay, because we have annotated the remaining constructor's parameters using the `org.jboss.errai.marshalling.client.api.annotations.MapsTo` annotation.

By doing this, we have told the marshaling system, for instance, that the first parameter of the constructor maps to the property `name` . Which in this case, defaults to the name of the corresponding field. This may not always be the case – as will be explored in the section on custom definitions. But for now that's a safe assumption.

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

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

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

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

  public String getName() {
    return name;
  }

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

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

4.1.1.3. Manual Mapping

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

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

4.1.1.3.1. Mapping Existing Client Classes

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

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

Example 4.1. Example ErraiApp.properties defining portable classes.

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

4.1.1.3.2. Aliased Mappings of Existing Interface Contracts

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

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

```
package org.foo.client;

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

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

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

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

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

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

The syntax of the mapping is as follows: `<class_to_map>` **->** `<contract_to_map_to>` .

Aliases do not inherit functionality!

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

4.1.2. Manual Class Mapping

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

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

4.1.2.1. MappingDefinition

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

Consider the following class:

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

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

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

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

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

Let us construct this object like so:

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

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

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

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

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

    setInstantiationMapping(cnsMapping);

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

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

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

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

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

3. Using the `WriteMapping` class, we have indicated to the marshaller framework how to write the `"mySuperNickname"` field, using the `String.class` marshaller, and using the setter method `setMySuperNickname` .

4. Using the `ReadMapping` class, we have indicated to the marshaller framework how to read the `"mySuperName"` field, using the `String.class` marshaller, and using the getter method `getMySuperName` .

5. Using the `ReadMapping` class, we have indicated to the marshaller framework how to read the `"mySuperNickname"` field, using the `String.class` marshaller, and using the getter method `getMySuperNickname` .

4.1.3. Custom Marshallers

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

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

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

Example 4.2. DataMarshaller.java from the built-in marshallers

```
@ClientMarshaller @ServerMarshaller
public class DateMarshaller extends AbstractNullableMarshaller<Date> {
  @Override
  public Class<Date> getTypeHandled() {
    return Date.class;
  }

  @Override
  public Date demarshall(EJValue o, MarshallingSession ctx) {
    // check if the JSON element is null
    if (o.isNull() != null) {
      // if the JSON element is null, so is our object!
      return null;
    }

    // instantiate our Date!
    return new Date(Long.parseLong(o.isObject().get(SerializationParts.QUALIFIED_VALUE).isStrin
  }

  @Override
  public String marshall(Date o, MarshallingSession ctx) {
    // if the object is null, we encode "null"
```

```
    if (o == null) { return "null"; }

    // return the JSON representation of the object
              return    "{\""    +    SerializationParts.ENCODED_TYPE    +    "\":
\"" + Date.class.getName() + "\"," +
           "\"" + SerializationParts.OBJECT_ID + "\":\"" + o.hashCode() + "\"," +
                              "\""   +   SerializationParts.QUALIFIED_VALUE   +   "\":
\"" + o.getTime() + "\"}";
  }
}
```

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

The `demarshall()` method does what its name implies: it is responsible for demarshalling the object from JSON and turning it back into a Java object.

The `marshall()` method does the opposite, and encodes the object into JSON for transmission on the wire.

Remote Procedure Calls (RPC)

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

```
@Service
public class MyRemoteServiceImpl implements MyRemoteService {

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

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

5.1. Making calls

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

```
MessageBuilder.createCall(new RemoteCallback<Boolean>() {
  public void callback(Boolean isHappy) {
```

```
    if (isHappy) Window.alert("Everyone is happy!");
  }
 }, MyRemoteService.class).isEveryoneHappy();
```

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

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

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

5.1.1. Proxy Injection

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

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

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

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

5.2. Handling exceptions

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

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

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

As remote exceptions need to be serialized to be sent to the client, the `@ExposeEntity` annotation needs to be present on the corresponding exception class (see *Chapter 4, Marshalling*). Further the exception class needs to be part of the client-side code. For more details on `ErrorCallbacks` see *Section 2.3, “Handling Errors”* .

5.3. Session and request objects in RPC endpoints

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

```
@Service
public class MyRemoteServiceImpl implements MyRemoteService {

  public boolean isEveryoneHappy() {
    HttpSession session = RpcContext.getHttpSession();
    ServletRequest request = RpcContext.getServletRequest();
    ...
    return true;
  }
}
```

Errai CDI

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

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

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

Important

The Errai CDI extension itself is implemented on top of the Errai IOC Framework (see *Chapter 3, Dependency Injection*), which itself implements the JSR-330 specification. Inclusion of the CDI module your GWT project will result in the extensions automatically being loaded and made available to your application.

6.1. Features and Limitations

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

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

1. There is no support for CDI interceptors in the client. Although this is planned in a future release.
2. Passivating scopes are not supported.
3. The JSR-299 SPI is not supported for client side code. Although writing extensions for the client side container is possible via the Errai IOC Extensions API.
4. The `@Typed` annotation is unsupported.
5. The `@Disposes` annotation is unsupported.
6. The `@Specializes` annotation is unsupported.
7. Qualifier attributes are not currently supported. (eg. `@MyQualifier(foo=BAR)` and `@MyQualifier(foo=FOO)` will be considered equivalent in the client).

6.1.1. Other features

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

6.2. Beans and Scopes

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

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

Example 6.1. Example dependent scoped bean

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

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

Example 6.2. Example ApplicationScoped bean

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

  // ... //
}
```

Availability of dependent beans in the client-side BeanManager

As is mentioned in the *bean manager documentation [29]* , only beans that are *explicitly* scoped will be made available to the bean manager for lookup. So while

it is not necessary for regular injection, you must annotate your dependent scoped beans with `@Dependent` if you wish to dynamically lookup these beans at runtime.

6.3. Events

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

Let's take a look at an example.

Example 6.3. FraudClient.java

```
public class FraudClient extends LayoutPanel {

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

  private HTML responsePanel;
 #
  public FraudClient() {
    super(new BoxLayout(BoxLayout.Orientation.VERTICAL));
  }

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

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

Two things are noteworthy in this example:

1. Injection of an `Event` dispatcher proxy

2. Creation of an `Observer` method for a particular event type

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

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

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

Example 6.4. AccountService.java

```
@ApplicationScoped
public class AccountService {
 #
  @Inject @Detected
  private Event<Fraud> event;

  public void watchActivity(@Observes AccountActivity activity) {
    Fraud fraud = new Fraud(System.currentTimeMillis());
    event.fire(fraud);
 }
}
```

6.3.1. Conversational events

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

Example 6.5. SubscriptionService.java

```
@ApplicationScoped
public class SubscriptionService {
 #
  @Inject
  private Event<Documents> welcomeEvent;

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

```
}
```

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

Example 6.6. Document.java

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

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

6.3.2. Client-Server Event Example

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

Example server code:

Example 6.7. MyServerBean.java

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

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

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

    myResponseEvent.fire(response);
  }
}
```

Domain-model:

Example 6.8. MyRequestEvent.java

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

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

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

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

Example 6.9. MyResponseEvent.java

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

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

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

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

Client application logic:

Example 6.10. MyClientBean.java

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

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

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

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

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

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

6.4. Producers

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

Example 6.11. App.java

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

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

Example 6.12. MyComposite.java

```
@ApplicationScoped
public class MyComposite extends Composite {
 #
  @Inject @Supported
  private MyBaseWidget widget;

  ...
}
```

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

Example 6.13. Singleton producer

```
public class App {
  ...

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

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

6.5. Deploying Errai CDI

If you do not care about the deployment details for now and just want to get started take a look at the *Quickstart Guide* [https://docs.jboss.org/author/pages/viewpage.action?pageId=5833096] .

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

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

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

6.5.1. Deployment in Development Mode

In development mode we need to bootstrap the CDI environment on our own and make both Errai and CDI available through JNDI (common denominator across all runtimes). Since GWT uses Jetty, that only supports read only JNDI, we need to replace the default Jetty launcher with a custom one that will setup the JNDI bindings:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven plugin</artifactId>
  <version>${gwt.maven}</version>

  <configuration>
  ...
    <server>org.jboss.errai.cdi.server.gwt.JettyLauncher</server>
  </configuration>
  <executions>
  ...
  </executions>
</plugin>
```

Starting Development Mode from within your IDE

Consequently, when starting Development Mode from within your IDE the following program argument has to be provided: -server org.jboss.errai.cdi.server.gwt.JettyLauncher

Once this is set up correctly, we can bootstrap the CDI container through a servlet listener:

```
<web-app>
  ...
  <listener>
   <listener-class>org.jboss.errai.container.CDIServletStateListener</listener-
class>
  </listener>

  <resource-env-ref>
    <description>Object factory for the CDI Bean Manager</description>
    <resource-env-ref-name>BeanManager</resource-env-ref-name>
      <resource-env-ref-type>javax.enterprise.inject.spi.BeanManager</resource-
env-ref-type>
  </resource-env-ref>
  ...
</web-app>
```

Errai-CDI maven archetype

Sounds terribly complicated, no? Don't worry we provide a maven archetype that takes care of all these setup steps and configuration details.

6.5.2. Deployment to a Servlet Engine

Deployment to servlet engine has basically the same requirements as running in development mode. You need to include the servlet listener that bootstraps the CDI container and make sure both Errai and CDI are accessible through JNDI. For Jetty you can re-use the artefacts we ship with the archetype. In case you want to run on tomcat, please consult the *Apache Tomcat Documentation* [http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html] .

6.5.3. Deployment to an Application Server

We provide integration with the *JBoss Application Server* [http://jboss.org/jbossas] , but the requirements are basically the same for other vendors. When running a GWT client app that leverages CDI beans on a Java EE 6 application server, CDI is already part of the container and accessible through JNDI (`java:/BeanManager`).

6.5.4. Configuration Options

Since the discovery of service implementations (beans) is delegated to the CDI container, we need to disable Errai's own service discovery mechanism. In order to do so, simply turn off the auto-discovery feature in `ErraiService.properties`

```
errai.auto_discover_services=false
```

Errai JAX-RS

JAX-RS (Java API for RESTful Web Services) is a Java EE standard (JSR-311) for implementing REST-based Web services in Java. Errai JAX-RS brings this standard to the browser and simplifies the integration of REST-based services in GWT client applications. Errai can generate proxies based on JAX-RS interfaces which will handle all the underlying communication and serialization logic. All that's left to do is to invoke a Java method. We have provided a Maven archetype which will create a fully function CRUD application using JAX-RS. See the *Quickstart Guide* [https://docs.jboss.org/author/pages/viewpage.action?pageId=5833096] for details.

7.1. Creating Requests

The JAX-RS interface needs to be visible to the GWT compiler and should therefore reside within the client packages (e.g. client.shared).

Assuming the following simple JAX-RS interface should be used:

Example 7.1. CustomerService.java

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

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

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

Example 7.2. App.java

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

```
...
```

For details on the callback mechanism see *Section 7.2, "Handling Responses"* .

7.1.1. Proxy Injection

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

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

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

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

7.2. Handling Responses

An instance of Errai's `RemoteCallback<T>` has to be passed to the `RestClient.create()` call, which will provide access to the JAX-RS resource method's result. `T` is the return type of the JAX-RS resource method. In the example below it's just a `Long` representing a customer ID, but it can be any serializable type (see *Chapter 4, Marshalling*).

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

A special case of this `RemoteCallback` is the `ResponseCallback` which provides access to the `Response` object representing the underlying HTTP response. This is useful when more details of the HTTP response are needed, such as headers, the status code, etc. This `ResponseCallback` can be provided as an alternative to the `RemoteCallback` for the method result.

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

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

```
ErrorCallback errorCallback = new ErrorCallback() {
  public boolean error(Message message, Throwable throwable) {
    try {
      throw throwable;
    }
    catch (ResponseException e) {
      Response response = e.getResponse();
      // process unexpected response
      response.getStatusCode();
    }
    catch (Throwable t) {
      // process unexpected error (e.g. a network problem)
    }
    return false;
  }
};
```

7.3. Wire Format

Errai's JSON format will be used to serialize/deserialize your custom types. See *Chapter 4, Marshalling* for details. A future extension to Errai's marshaller capabilities will support pluggable/custom serializers. So in the near future you will have the flexibility to use your own wire format.

7.4. Errai JAX-RS Configuration

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

To configure a relative or absolute root path, the following JavaScript variable can be set in either

the host HTML page

```
<script type="text/javascript">
  erraiJaxRsApplicationRoot = "/MyJaxRsEndpointPath";
</script>
```

or by using a JSNI method:

```
private native void setMyJaxRsAppRoot(String path) /*-{
  $wnd.erraiJaxRsApplicationRoot = path;
}-*/;
```

or by simply invoking:

```
RestClient.setApplicationRoot("/MyJaxRsEndpointPath");
```

The root path will be prepended to all paths specified on the JAX-RS interfaces. It serves as the base URL for all requests sent from the client.

Configuration

This section contains information on configuring Errai.

8.1. 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 8.6, "Servlet Implementations"* 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>
```

8.2. Client Configuration

In some cases it might be desirable to prevent the client bus from communicating with the server. One use case for this is when all communication with the server is handled using JAX-RS and the constant long polling requests for message exchange are not needed.

To turn off remote communication in the client bus the following JavaScript variable can be set in the HTML host page:

```
<script type="text/javascript">
  erraiBusRemoteCommunicationEnabled = false;
</script>
```

8.3. 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.

8.4. 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
```

8.4.1. 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 8.5, "Dispatcher Implementations"* for more information about the differences between the two.

8.4.2. 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.

8.4.3. 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.

8.4.4. errai.authentication_adapter

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

8.4.5. 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.

8.4.6. errai.auto_discover_services

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

8.4.7. errai.auto_load_extensions

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

8.5. Dispatcher Implementations

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.

8.5.1. 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.

8.5.2. 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.

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

8.6.1. 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.

8.6.2. 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).

8.6.3. 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.

8.6.4. StandardAsyncServlet

This implementation leverages asynchronous support in Servlet 3.0 to allow for threadless pausing of port connections. Note that `<async-supported>true</async-supported>` has to be added to the servlet definition in `web.xml` .

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 9.1. 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 9.2. 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.

Upgrade Guide

This chapter contains important information for migrating to newer versions of Errai. If you experience any problems, don't hesitate to get in touch with us. See *Chapter 13, Reporting problems* .

10.1. Upgrading from 1.x to 2.0

The first issues that will arise after replacing the jars or after changing the version numbers in the `pom.xml` are unresolved package imports. This is due to refactorings that became necessary when the project grew. Most of these import problems can be resolved automatically by modern IDEs (Organize Imports). So, this should replace `org.jboss.errai.bus.client.protocols.*` with `org.jboss.errai.common.client.protocols.*` for example.

The following is a list of manual steps that have to be carried out when upgrading:

- @ExposedEntity became @Portable (`org.jboss.errai.common.client.api.annotations.Portable`). See *Chapter 4, Marshalling* for details.

- Errai CDI projects must now use the `SimpleDispatcher` instead of the `AsynDispatcher` . This has to be configured in *Section 8.4, “ErraiService.properties”* .

- The bootstrap listener (configured in `WEB-INF/web.xml`) for Errai CDI has changed (`org.jboss.errai.container.DevModeCDIBootstrap` is now `org.jboss.errai.container.CDIServletStateListener`).

- gwt 2.3.0 or newer must be used and replace older versions.

- mvel2 2.1.Beta8 or newer must be used and replace older versions.

- weld 1.1.5.Final or newer must be used and replace older versions.

- slf4j 1.6.1 or newer must be used and replace older versions.

- This step can be skipped if Maven is used to build the project. If the project is NOT built using Maven, the following jar files have to be added manually to project's build/class path: errai-common-2.x.jar, errai-marshalling-2.x.jar, errai-codegen-2.x.jar, netty-4.0.0.Alpha1.errai.r1.jar.

- If the project was built using an early version of an Errai archetype the configuration of the maven-gwt-plugin has to be modified to contain the `<hostedWebapp>path-to-your-standard-webapp-folder</hostedWebapp>` . This is usually either `war` or `src/main/webapp` .

Downloads

The distribution packages can be downloaded from jboss.org *http://jboss.org/errai/Downloads.html*

Sources

Errai is currently managed using Github. You can clone our repositories from *http://github.com/errai* .

Reporting problems

If you run into trouble don't hesitate to get in touch with us:

- JIRA Issue Tracking: *https://jira.jboss.org/jira/browse/ERRAI*
- User Forum: *http://community.jboss.org/en/errai?view=discussions*
- Mailing List: *http://jboss.org/errai/MailingLists.html*
- IRC: *irc://irc.freenode.net/errai*

Errai License

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

Appendix A. Revision History

Revision History

Revision <>