Developing JAX-RS Web Applications Utilizing Server-Sent Events and WebSocket

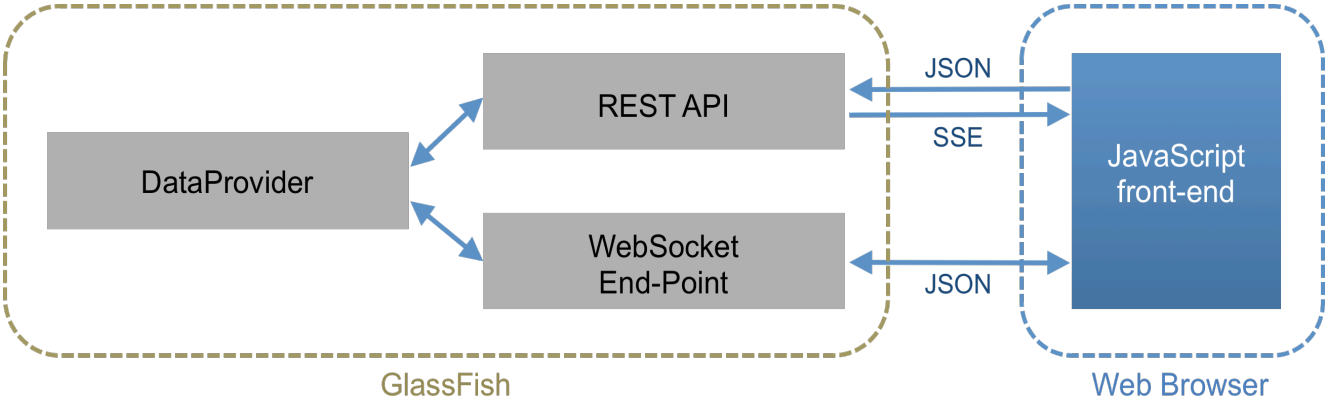
Martin Matula
Sr. Development Manager

ORACLE

Table of Contents

Introduction

In this lab we are going to demonstrate some of the upcoming features of JavaEE 7 as well as value-add features we are working on for GlassFish, the JavaEE reference implementation. We will develop a web application that allows users to collaboratively draw simple pictures. Here is how the architecture of our application is going to look like:



The application will be deployed to GlassFish, and will consist of a JavaScript front-end running in the browser, communicating with the Java back-end running on the server. The back-end is going to expose REST API and web sockets. The front-end is going to utilize these to access the data and render them in the browser. This lab focuses on the Java back-end development, so we won't dive into how the JavaScript portion of the application is implemented. Also keep in mind this is not a "best practices" kind of lab. The main focus is on demonstrating the technology, rather than discussing the best practices when developing HTML5-based web applications.

Here are the projects we are going to utilize:

- GlassFish – open source application server, reference implementation of JavaEE (includes all of the other projects mentioned below)
- Jersey – open source framework for building RESTful web services in Java, reference implementation of JAX-RS
- Tyrus – open source web socket runtime, reference implementation of Java API for WebSocket
- JSON Processing – open source implementation of Java API for JSON Processing

Development of this application is split into three exercises. In the first one, we are going to develop a simple RESTful API for our application, in the second one we will add server-sent events and the third exercise is focused on web sockets.

We are also going to briefly look at the client programming model by developing a simple console application. This will be done in the last (fourth) exercise of this lab. This application will interact with the Drawing Board application we will develop in the first three exercises.

If you ever get stuck while following the lab guide, you can look at the solution for the exercise. Solutions are located in <lab_root>/solutions directory (<lab_root> being the location of the lab files where also this lab guide document resides).

If you need to install the lab environment yourself, the Appendix at the end of this document describes how to do that.

Once you finish the lab, you can find out more about the related technologies we used in this lab by following the links in the Summary section of this document.

Exercise 1: Exposing RESTful API

In this first exercise, we are going to create and expose a simple RESTful API for CRUD (create, retrieve, update, delete) operations on top of drawings. To save time, instead of starting from scratch, there is an existing project in the <lab_root>/drawingboard we are going to add this functionality to.

Step 1: Exploring the Initial Project

Let's first look at what's already in the project:

1. Start NetBeans and open the project **drawingboard** from **<lab_root>** directory.
2. Expand "Drawing Board Web Application"/"Web Pages" folder in the project view. This folder contains the front-end of our application. It utilizes AngularJS JavaScript framework. The main entry point to the application is the index.html, which loads the framework scripts and style sheets. We define two views – main.html for working with the list of drawings and drawing.html for working with a single drawing. Most of the application-specific front-end logic is implemented in controller.js file, where we define controllers for these two views. The controllers take care of interacting with the backend, receiving the server-sent events as well as opening WebSocket connections. Detailed description of the JavaScript part of the application is beyond the scope of this hands-on-lab, as we are focusing on building the back-end part in Java.
3. Expand "Source Packages" node of the project. You can see the project currently has one Java package with two classes. DataProvider class is a simple utility class serving as an in-memory data store for drawings. It defines operations for creating, retrieving, updating and deleting the drawings. The second class in the package defines Java representation of a drawing.
4. Let's run the project to see how the initial page looks like. To do that, first right-click on the project node and choose "Clean and Build" in the pop-up menu, then right-click again and choose "Run".
   NOTE: When you click on Run, a dialog may appear asking you to choose which application server you want to run the application with. Select "GlassFish Server 4.0-b57", you can check "Remember Permanently" and click OK.
   The NetBeans will start GlassFish, deploy our application and open a web browser at the application URL (in this case http://localhost:8080/drawingboard/).
5. The initial screen of the application has a text field where you can enter a name for a new drawing and hit Enter or click the New button to issue a command to create a new drawing with that name. This will not work at the moment, as we haven't exposed the RESTful API the front-end tries to use to create the new drawing.
6. You can confirm that the front-end makes HTTP requests to the back-end by enabling the network monitoring in Chrome browser. To do that, choose View->Developer->Developer Tools in the Chrome application menu. The Developer Tools will show up at the bottom of the browser window. Switch to the Network tab.

7. Now, type something into the Drawing Name text field of our application (e.g. “test”) and hit Enter. In the Network tab of Developer Tools you should see the frond-end made HTTP POST request to /drawingboard/api/drawings URL, but the server responded with “404 – Not Found” status code, since we haven’t exposed anything at that URI yet.
8. Once done, close the Developer Tools pane.

Step 2: Adding JAX-RS Resources

Now we are going to expose the RESTful API. Here is how we want the API to look like:

URI	HTTP Method	Description
<app_context>/api/drawings	POST	Creates a new drawing
<app_context>/api/drawings	GET	Retrieves the list of all drawings
<app_context>/api/drawings/{id}	GET	Retrieves a drawing with id = {id}
<app_context>/api/drawings/{id}	DELETE	Deletes a drawing with id = {id}

1. First, we need to add a dependency on Jersey libraries to our project. Double-click “Project Files”->pom.xml file to open it and add the following dependencies just before the closing </project> tag:

```
<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.core</groupId>
        <artifactId>jersey-server</artifactId>
        <version>${jersey.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-moxy</artifactId>
        <version>${jersey.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

   The dependency on jersey-server is needed so that we can use the JAX-RS and Jersey server-side API in our application. We are going to use MOXy to map Java objects (particularly the Drawing bean) to/from JSON, hence the dependency on jersey-media-moxy. Both dependencies have the scope set to "provided", which means the respective jars won't be bundled in the application war file. This is because GlassFish (our target deployment server) already contains these libraries out of the box, so no need to include them in the war.
2. Let’s rebuild the project (right-click on the project, click on “Build” in the pop-up menu) to get these new dependencies downloaded from maven.

3. Now, create a new DrawingsResource class (right-click on the com.mycompany.drawingboard package and choose New->Java Class in the pop-up menu).
4. We will expose this class at “drawings” URI (relative to the JAX-RS application URI). To do that, annotate the class with @Path(“drawings”) (add import for javax.ws.rs.Path).
5. Attach also @Consumes and @Produces annotations to the class to indicate the class expects/returns JSON messages:

```
@Path("drawings")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class DrawingsResource {
}
```

NOTE: Whenever copy-pasting the code from this document, fix imports Java imports after the code is copied, either by following the NetBeans hints (Alt+Enter) on each line, or by using the Optimize Imports feature (Ctrl+Shift+I or Cmd+Shift+I when on MacOS) which fixes all imports at once. You can also use Alt+Shift+F (Ctrl+Shift+F on MacOS) to reformat the code.

6. Let’s add a method named “create()” that will be used to create new drawings. We’ll map it to HTTP POST. The method will return "201 – Created" response with "Location" HTTP header set to the URI of the newly created drawing:

```
@POST
public Response create(@Context UriInfo uriInfo, Drawing drawing)
{
    return Response.created(uriInfo.getBaseUriBuilder()
            .path(DrawingsResource.class).path("{drawingId}")
            .build(DataProvider.createDrawing(drawing))
        ).build();
}
```

Note we are using JAX-RS @Context annotation to inject UriInfo, which provides contextual request-specific information about the request URI. The class provides us with the base URI of our application. We use it to construct the full URI of the newly created drawing. The second method parameter will receive the content of the HTTP request converted to an instance of Drawing object (using JSON un-marshaller provided by MOXy library).

7. Now, add a "getAll()" method that returns the list of all drawings (mapped to HTTP GET):

```
@GET
public List<Drawing> getAll() {
    return DataProvider.getAllDrawings();
}
```

As you can see, the method simply returns List<Drawing> - this is possible thanks to the concept of JAX-RS message body writers/readers that you can plug in to implement mapping to/from a specific media type to a java type (in this case we will be utilizing MOXy message body writer that knows how to convert Java objects into JSON strings).

8. The third resource method we are going to add is the "get()" method. It is actually going to be what we call a **sub-resource** method, because it will be exposed at a URI containing one additional path element – the drawing ID. So the method itself is going to be annotated with @Path annotation:

```
@Path("{id:[0-9]+}")
@GET
public Drawing get(@PathParam("id") int drawingId) {
    Drawing result = DataProvider.getDrawing(drawingId);
    if (result == null) {
        throw new NotFoundException();
    }
    return result;
}
```

As you can see, we are utilizing so called path parameter in this method. We use regular expression in the definition of the path parameter named "id" to indicate the parameter should only match path elements that contain numbers. So, a HTTP GET request to a URI like "…/drawings/1234" would match this resource method taking "1234" as the value of the "id" path parameter, however "…/drawings/abc" will not match.
Also note that the method utilizes NotFoundException() which is new to JAX-RS 2.0 and when thrown, produces "404 – Not Found" response code. Our method throws it when no drawing with the given ID was found.

9. Another sub-resource method we will add is "delete()". It will be mapped to the same path as "get()":

```
@Path("{id:[0-9]+}")
@DELETE
@Consumes("*/*")
public void delete(@PathParam("id") int drawingId) {
    if (!DataProvider.deleteDrawing(drawingId)) {
        throw new NotFoundException();
    }
}
```

10. Finally we need to add a JAX-RS application class that encapsulates the JAX-RS runtime configuration for our project. Add a new class named "JaxrsApplication", annotate it with @ApplicationPath("api"), make it extend "ResourceConfig" class, which is a Jersey API class that provides some useful functionality additional to what the default JAX-RS API Application class provides (such as package scanning, notion of properties, etc.).
NOTE: If REST Resources Configuration dialog appears while performing this step in

NetBeans, simply click Cancel.
The resulting class should look as follows:

```
@ApplicationPath("api")
public class JaxrsApplication extends ResourceConfig {
    public JaxrsApplication() {
        super(DrawingsResource.class);
        addBinders(new MoxyJsonBinder());
    }
}
```

As you can see, in the constructor of the class we are passing the resource class to the constructor of the super class – this tells the JAX-RS application what classes it should recognize as resources or providers. We are registering the JSON-related MOXy providers by a single call addBinders() – binder is a Jersey proprietary concept that can be used for registering a set of co-related JAX-RS providers as a single "feature".

11. We are done with the implementation part, let's rebuild and run the application to try it out (right-click on the project and click Run).
12. Once the application page opens in the browser, try entering some text into the Drawing Name text field and hit Enter. It will still look like nothing happened, since we haven't implemented the server-sent events part that would notify the JavaScript front-end that a new drawing has been added, however, if you hit Refresh in your browser, you should see the new drawing is there. This confirms our RESTful API works and the front-end is able to use it to create and retrieve drawings. You can try clicking the "x" next to the drawing to delete it and refresh again to see the changes.
13. We can also test the API directly (instead of using the front-end of our application). To retrieve the list of drawings in JSON format, you can enter the URI of our DrawingsResource (http://localhost:8080/drawingboard/api/drawings) directly into the address bar of the browser and hit enter. That will send an HTTP GET request to our resource and you should see the JSON string representing the list of drawings.
14. To directly make POST and DELETE requests to our REST API you can utilize the Postman Chrome add-on that's installed on your machine. To try adding a new drawing, you can enter http://localhost:8080/drawingboard/api/drawings address into the request URL field of Postman, switch the method to POST, click on the Headers button and add "Content-Type" header set to "application/json", switch to "raw" view of the message entity to be able to enter a JSON string and type in the following for example:

```
{"name" : "my drawing"}
```

After you click the Send button, this will create a new drawing named "my drawing". Feel free to try DELETE as well and to experiment further.

This concludes the first exercise where you learned how to expose simple RESTful API from your application using JAX-RS and Jersey. In the following exercise we are going to add support for change notifications using Jersey's implementation of the HTML5 concept called Server-Sent Events.

Exercise 2: Adding Server-Sent Events

Now that we have the basic REST API working, it is time to add the SSE notifications, so that the front-end gets automatically updated whenever someone adds/removes a drawing.

1. The SSE support for Jersey resides in jersey-media-sse maven module, so let's add this dependency to our application pom.xml file by copy-pasting the following into the <dependencies> section of that file:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-sse</artifactId>
    <version>${jersey.version}</version>
    <scope>provided</scope>
</dependency>
```

2. Rebuild the project so that the dependency gets downloaded (right-click the project and choose Build).
3. Jersey defines a class named SseBroadcaster, that can be used for broadcasing server-sent events. Let's add a broadcaster instance to the DataProvider class and use it to send events whenever any changes are made to the collection of drawings. Open the DataProvider class and add the following field declaration:

```
private static SseBroadcaster sseBroadcaster
        = new SseBroadcaster();
```

4. Now, find the createDrawing() method and insert the following just before the line with the return statement:

```
sseBroadcaster.broadcast(new OutboundEvent.Builder()
        .name("create")
        .data(Drawing.class, result)
        .mediaType(MediaType.APPLICATION_JSON_TYPE)
        .build());
```

   This will create a new event named "create", sending the newly created drawing object as JSON in the data field of the event, and broadcasts it to all the clients registered in the broadcaster instance (we'll add the client registration shortly).
5. Next, we are going to add event notification to DataProvider.deleteDrawing() method. Add the following code before the return statement:

```
sseBroadcaster.broadcast(new OutboundEvent.Builder()
        .name("delete")
        .data(String.class, String.valueOf(drawingId))
```

```
        .build());
```

As you can see, this generates an event named "delete" and data field containing the ID of the drawing being deleted.

6. We've just added the event notification code, but how is the client registration to the broadcaster going to work? Jersey has a concept of an "event channel", which is essentially a long-running running connection established by the client the server uses to send the event data to. Events sent from the server are a long-running response (typically to a HTTP GET request made by the client) being sent from the server in "chunks". EventChannel is a class in the Jersey API that represents this SSE connection. Let's add a method to the DataProvider class for registering a new EventChannel to the broadcaster:

```
static void addEventChannel(EventChannel ec) {
    sseBroadcaster.add(ec);
}
```

7. Now we have to add a resource method to our DrawingsResource class responding to HTTP GET request, establishing the SSE EventChannel connection. Open the DrawingsResource class and add the following method to it:

```
@GET
@Path("events")
@Produces(EventChannel.SERVER_SENT_EVENTS)
public EventChannel getEvents() {
    EventChannel ec = new EventChannel();
    DataProvider.addEventChannel(ec);
    return ec;
}
```

As you can see, this method adds another sub-resource mapped to …/drawings/events URI, it produces a response of type text/event-stream (EventChannel.SERVER_SENT_EVENTS constant value), which is the standard media type for SSE and all it does is creating a new EventChannel instance, registering it to our broadcaster (through the DataProvider.addEventChannel() method we added in the previous step), and returning it. Jersey keeps the connection open, and releases the container thread for processing other requests (i.e. open SSE connections don't block container threads).

8. Finally we need to add a special MessageBodyWriter for SSE to our JaxrsApplication (so that Jersey knows how to convert the event objects to the stream of data sent on the wire). Doing that is simple – just go to the JaxrsApplication class constructor and add OutboundEventWriter.class to the list of classes passed to the super constructor, so now the call to super() should look like this:

```
super(DrawingsResource.class, OutboundEventWriter.class);
```

9. Let's run the project to see if it works (right-click on the project and click Run). Once the browser opens, try adding a new drawing again – this time you should see the new drawing is displayed in the list of drawings right away. Since the front-end now receives the events.
10. Try opening another browser window, so that you have the application in two windows side-by-side. Try adding/deleting drawings in one window and watch how the list of drawings gets automatically updated in both.
11. Before we move on to the next exercise, let's quickly look at how the event listening is implemented on the JavaScript side. Open controller.js file in the Web Pages folder of the project. Between lines 22 and 35 you can see the following code:

```
// listens to server-sent events for the list of drawings
$scope.eventSource = new
EventSource("/drawingboard/api/drawings/events");

var eventHandler = function (event) {
    $scope.drawings = DrawingService.query();
};

$scope.eventSource.addEventListener("create", eventHandler,
false);
$scope.eventSource.addEventListener("delete", eventHandler,
false);

// clean up
$scope.$on("$destroy", function (event) {
    $scope.eventSource.close();
});
```

As you can see, on the JavaScript side, EventSource object (available in HTML5-compliant browsers) is used to establish an SSE connection – it makes an HTTP GET request to the URI passed to it in the constructor. The request hits our DrawingsResource.getEvents() resource method, which keeps the connection open and registers the stream (EventChannel) to the broadcaster. You can register event handlers on the EventSource object based on the event name. In the code above we are adding listeners for the two types of events we are firing (create and delete) – both use the same event handler, which simply reads an updated list of drawings from the server.

This concludes the second exercise. You've learned how you can leverage Jersey API to enable server-sent events in your server-side application. In the following exercise we are going to look at how to utilize another HTML5 technology – web sockets – to do bi-directional communication between the server and the clients.

Exercise 3: Adding Web Sockets

So far we have been dealing with the main page of our application, showing the list of drawings. In this exercise we are going to implement the functionality for the drawing detail page. This page opens when a user clicks on a particular drawing. We are going to use web socket to transmit the list of existing shapes of a particular drawing to the client and then continue using the open web socket connection to broadcast and receive any changes this user and other users make to the drawing. This will enable collaborative editing of the drawing by multiple users in real time.

To implement the web socket functionality on the server side, we are going to use an early version of the new Java API for WebSocket that is bundled in nightly builds of GlassFish 4.0 and is coming as part of Java EE 7.

The JavaScript front-end is trying to establish a web socket connection at the following URI:

```
ws://host:port/drawingboard/websockets/{drawingId}
```

So, we will have to make our web-socket endpoint handle that URI space.

Step 1: Implementing Web Socket End-Point

1. Like in the previous exercises, the first thing we need to do to be able to use the new API is adding the corresponding dependencies to our maven pom.xml file. The web socket API implementation is provided by project Tyrus. So, open pom.xml and add the following to the dependencies section:

```
<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>websocket-impl</artifactId>
    <version>${websocket-version}</version>
    <scope>provided</scope>
</dependency>
```

2. Rebuild the project so that the dependency gets downloaded (right-click on the project and choose Build).
3. Similarly to JAX-RS, the web socket API is annotation-based. To expose a class as a web socket end-point, you need to annotate it with the standard @WebSocketEndpoint annotation. So, create a new class named DrawingWebSocket and annotate it with @WebSocketEndpoint, setting the path parameter of the annotation to "/websockets/", as that is the URI prefix at which the class should listen for incoming connections:

```
@WebSocketEndpoint(
    path = "/websockets/"
)
```

```
public class DrawingWebSocket {
}
```

Ideally, we should be able to define the path as "/websockets/{drawingId}" and have the drawingId injected into our handler methods by the web socket runtime automatically. However, at the time of writing this lab guide such functionality is not yet available in the Web Socket API (but hopefully will be there soon). Defining the path as "/websockets/" will cause every request coming to URI starting with …/websockets/ to be handled by this web socket end-point. We have to extract the drawingId from the URI manually.

4. Let's implement the method to handle opening of a new web socket connection. To do that, add a new method and annotate it with @WebSocketOpen annotation (the name of the method is not significant, but let's call it onOpen):

```
@WebSocketOpen
public void onOpen(Session session) {
}
```

The web socket runtime passes session object as a parameter to this method. The object contains contextual information for the connection being opened (such as the request URI for example) and methods enabling to send messages to the connection peer.

5. As part of onOpen() method we are going to extract the drawingId from the request URI (to see which drawing should this connection be associated with). We are going to keep session->drawingId map, so that we don't have to extract the drawingId from the session object for every message coming on the same connection. Add the following final static variables to the DrawingWebSocket class – one for the regular expression we are going to use to extract the drawing ID from the URI and another one for the session-to-drawingId map:

```
private static final Pattern URI_PATTERN =
Pattern.compile("(?:.*)/websockets/([0-9]+)");
private static final ConcurrentHashMap<Session, Integer>
sessionToId = new ConcurrentHashMap<>();
```

6. Let's extract the drawing ID when the connection opens. Update the onOpen() method as follows:

```
@WebSocketOpen
public void onOpen(Session session) {
    // see if the request URI matches the regular expression
    Matcher matcher =
URI_PATTERN.matcher(session.getRequestURI().toString());
    if (!matcher.matches()) {
        // if not, close the connection (invalid drawing ID)
        try {
            session.close(new
CloseReason(CloseReason.Code.CANNOT_ACCEPT, "Not found."));
```

```
        } catch (IOException ex) {

Logger.getLogger(DrawingWebSocket.class.getName()).log(Level.SEVER
E, null, ex);
        }
    } else {
        // if it does match, extract the drawing ID
        int drawingId = Integer.parseInt(matcher.group(1));
        // store it in the map
        sessionToId.put(session, drawingId);
    }
}
```

7. Now we can implement a method for handling incoming messages on an existing connection. That's what @WebSocketMessage annotation is used for. The body of the message gets passed to the method as a parameter. Since the front-end sends a JSON representation of a shape that was added to the drawing, it would be nice if we could somehow map that to Drawing.Shape object automatically. Luckily, the web socket API has a notion of decoders and encoders that are similar to JAX-RS message body readers/writers. They allow mapping of Java objects to/from messages. Since we will define a decoder and encoder for Drawing.Shape object, we can declare the parameter of our message hander to be of type Drawing.Shape, and the web socket runtime will automatically call our decoder to do the conversion:

```
@WebSocketMessage
public void shapeCreated(Drawing.Shape shape, Session session) {
    // get the drawing ID corresponding to this web socket session
    int drawingId = sessionToId.get(session);
    // add a new shape to the drawing
    DataProvider.addShape(drawingId, shape);
}
```

8. To complete the web socket end-point implementation, we should do some clean-up (remove session-to-drawingId mapping) when a session closes. To do that, add onClose() method:

```
@WebSocketClose
public void onClose(Session session){
    sessionToId.remove(session);
}
```

Step 2: Implementing Message Encoding/Decoding

Now that we have the basic implementation of the end-point, we need to add the encoder and decoder for Shape objects:

1. To process JSON, we are going to try out another early access API coming in JavaEE 7 – Java API for JSON Processing. Let's add the following dependency to the pom.xml file:

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.json</artifactId>
    <version>1.0-b01</version>
    <scope>provided</scope>
</dependency>
```

   Rebuild the project so that the dependency gets downloaded.
2. Add a new class named ShapeCoding and make it implement Decoder.Text and Encoder.Text interfaces:

```
public class ShapeCoding implements Decoder.Text<Drawing.Shape>,
Encoder.Text<Drawing.Shape> {
    @Override
    public Drawing.Shape decode(String message) throws
DecodeException {
    }

    @Override
    public boolean willDecode(String message) {
        // all messages will be decoded by this decoder
        return true;
    }

    @Override
    public String encode(Drawing.Shape shape) throws
EncodeException {
    }
}
```

3. Implement the decode() method as follows:

```
@Override
public Drawing.Shape decode(String message) throws DecodeException
{
    // -- workaround for a web socket implementation issue
    Thread.currentThread().setContextClassLoader(
            getClass().getClassLoader());
    // -- end of wokaround

    Drawing.Shape shape = new Drawing.Shape();
```

```
    try (JsonReader reader = new JsonReader(new
StringReader(message))) {
        JsonObject object = reader.readObject();
        shape.x = object.getValue("x", JsonNumber.class)
                .getIntValue();
        shape.y = object.getValue("y", JsonNumber.class)
                .getIntValue();
        shape.type = Drawing.ShapeType.valueOf(
                object.getValue("type", JsonString.class)
                .getValue());
        shape.color = Drawing.ShapeColor.valueOf(
                object.getValue("color", JsonString.class)
                .getValue());
    }

    return shape;
}
```

As you can see, the JSON Processing API is quite low level – it does not provide direct Java binding to POJO's (that's what another upcoming JSR – Java API for JSON Binding will be targeting), anyway, it is quite easy to work with in this case. Since we are always expecting just one type of object – Drawing.Shape.

4. We've seen how you can read JSON using the JSON Processing API. Let's see how you can produce JSON. Implement the encode() method as follows:

```
@Override
public String encode(Drawing.Shape shape) throws EncodeException {
    // -- workaround for a web socket implementation issue
    Thread.currentThread().setContextClassLoader(
            getClass().getClassLoader());
    // -- end of wokaround

    StringWriter result = new StringWriter();

    try (JsonGenerator gen = Json.createGenerator(result)) {
        gen.beginObject()
                .add("x", shape.x)
                .add("y", shape.y)
                .add("type", shape.type.toString())
                .add("color", shape.color.toString())
                .endObject();
    }

    return result.toString();
}
```

Again, the code should be quite self-descriptive.

5. Now we need to register the ShapeCoding class as the decoder and encoder on the DrawingWebSocket end-point. Open the DrawingWebSocket class and update the @WebSocketEndpoint annotation as follows:

```
@WebSocketEndpoint(
        decoders = ShapeCoding.class,
        encoders = ShapeCoding.class,
        path = "/websockets/"
)
```

Step 3: Broadcasting Web Socket Messages

OK, so we have the web socket end-point to receive the web socket messages. But if we ran the application now, it would still not work. That is because all the front-end does when you try to draw a shape on the canvas is it sends a web socket message describing that shape to the server. It is then server's responsibility to add the shape to the actual drawing object and broadcast that change back to all the clients (including the one that sent the original message). The front-end adds the shape to the canvas only as a result of receiving the web socket message from the server. But we are not sending anything yet, so nothing will be drawn. In this section we are going to complete the puzzle. We will add methods to the DataProvider for registering and unregistering of web socket sessions. And then in the addShape() method we need to broadcast the change to all sessions registered for that particular drawing.

Let's start:

1. Open DataProvider class and add the following static field. We are going to use it to store the web socket session registrations:

```
private static final MultivaluedHashMap<Integer, Session>
        webSockets = new MultivaluedHashMap<>();
```

   It is a "multi-valued" map, which maps a drawing ID to a list of web socket sessions that are associated with that drawing ID.

2. Now we can add a method that associates a new web socket session with a drawing ID. We will be calling this method from DrawingWebSocket.onOpen() when a new connection opens and so, as part of this method we should let the connecting client know, what shapes already exist in the drawing, so that it can correctly render the current state of the drawing. Add the following method to the DataProvider which does that:

```
static synchronized void addWebSocket(int drawingId,
        Session session) {
    // associate the session with the drawing ID
    webSockets.add(drawingId, session);

    Drawing drawing = getDrawing(drawingId);
```

```
    // if the drawing exists and has shapes,
    // send all these shapes to the client
    // so that it can draw them
    if (drawing != null && drawing.shapes != null) {
        for (Drawing.Shape shape : drawing.shapes) {
            try {
                session.getRemote().sendObject(shape);
            } catch (IOException | EncodeException ex) {
                Logger.getLogger(DataProvider.class.getName())
                        .log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

In this code you can nicely see, how you can use the session object to send messages to the remote peer (i.e. the other end of the web socket connection).

3. When the web socket connection closes, we should remove the session from the webSockets map. So, let's add a method for unregistering the session:

```
static synchronized void removeWebSocket(int drawingId,
        Session session) {
    List<Session> sessions = webSockets.get(drawingId);
    if (sessions != null) {
        sessions.remove(session);
    }
}
```

4. Now we'll add a helper method for broadcasting a shape to all web socket sessions associated with a given drawing:

```
private static void wsBroadcast(int drawingId,
        Drawing.Shape shape) {
    List<Session> sessions = webSockets.get(drawingId);
    if (sessions != null) {
        for (Session session : sessions) {
            try {
                session.getRemote().sendObject(shape);
            } catch (IOException | EncodeException ex) {
                Logger.getLogger(DataProvider.class.getName())
                        .log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

5. Let's call this method from DataProvider.addShape(). Add the following right before the line with "return true;" in the addShape() method:

```
wsBroadcast(drawingId, shape);
```

6. Finally we will call DataProvider.addWebSocket() from DrawingWebSocket.onOpen() and DataProvider.removeWebSocket() from DrawingWebSocket.onClose(). Add the following code right after the line with "sessionToId.put(session, drawingId);" in DrawingWebSocket.onOpen():

```
DataProvider.addWebSocket(drawingId, session);
```

   And update the DrawingWebSocket.onClose() method as follows:

```
@WebSocketClose
public void onClose(Session session){
    int drawingId = sessionToId.remove(session);
    DataProvider.removeWebSocket(drawingId, session);
}
```

7. That's it. Let's try to run the application to confirm everything works. Once the browser comes up, open another browser window at the same URI like in the last exercise. Add a new drawing, open it in both windows and start drawing in one of the windows to see if the same is drawn in the other one. You can try hitting the Back button to get back to the list of drawings and click on the drawing again to see that when opening the drawing again, all the existing shapes will get drawn (thanks to the code in DataProvider.addWebSocket() method). Try switching between the browser windows (draw in one, then the other), change the shapes and colors.
8. Before we move on to the next exercise, let's also quickly go through how the web socket communication is implemented on the JavaScript side. Open the controller.js file (under Web Pages node) again. The code that opens the web socket connection and listens to web socket messages is between lines 46 and 51:

```
// open a web socket connection for a given drawing
$scope.websocket = new WebSocket("ws://" + document.location.host
    + "/drawingboard/websockets/" + $routeParams.drawingId);
$scope.websocket.onmessage = function (evt) {
    $scope.drawShape(eval("(" + evt.data + ")"));
};
```

   The front-end sends web socket messages to the server in the mouseDown event handler on line 106:

```
$scope.websocket.send(
    '{"x" : ' + posx +
    ', "y" : ' + posy +
```

```
    ', "color" : "' + $scope.shapeColor +
    '", "type" : "' + $scope.shapeType + '"}');
```

This concludes the third exercise. At this point our application is complete. In the next chapter we will develop a very simple client using the new client API in JAX-RS 2.0 and the Jersey client API for receiving Server-Sent Events.

Exercise 4: Implementing a Java-based SSE Client

In this last exercise of our lab we will develop a Java-based restful client application and see the basics of JAX-RS 2.0 client API and the proprietary client-side SSE API in Jersey.

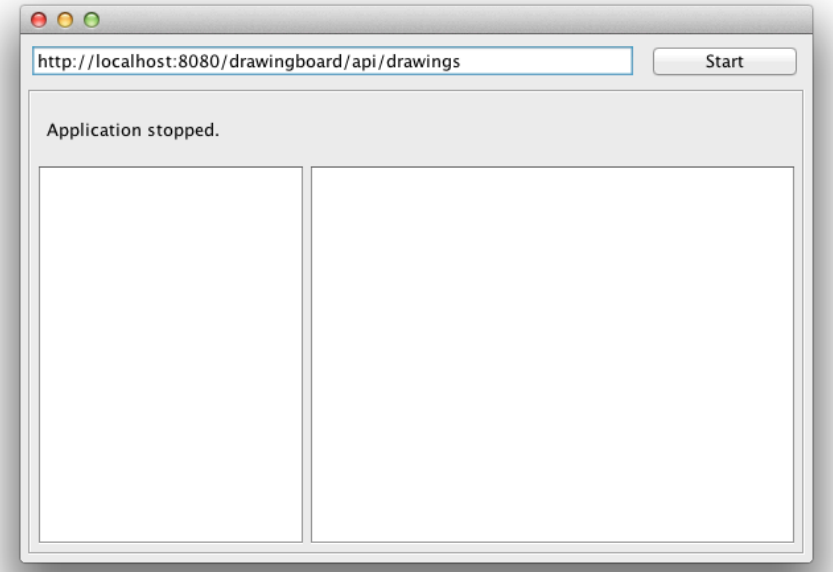
Step 1: Explore the initial SSE Client project

1. Open the project **drawingboard-client** from **<lab_root>** directory.
2. Expand the "Drawing Board SSE Client"/"Project Files" node of the project and double-click the `pom.xml` file to open it.
   The POM file of the project already contains all the necessary dependencies required to implement the client-side logic:

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>${jersey.version}</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
    <version>${jersey.version}</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-sse</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

3. Expand the "Drawing Board SSE Client"/"Source Packages" node of the project. You can see that the project contains a single `com.mycompany.drawingboardclient` package with two classes.
   `Drawing` class is the class you should be familiar with from the previous exercises. This class defines a Java representation of a drawing. We will use this class to process the drawing data returned from the server.
   `SseClientApp` is the second class in the package. The class represents a Drawing Board Swing GUI application envelope that has been provided for your convenience. You will be filling the empty methods in this class to provide the necessary logic that connects the Swing client application to the Drawing Board Web Application developed in previous exercises.
4. Let's run the project to see how the initial client application looks like.
   To do that, first right-click on the project node and choose "Clean and Build" in the pop-up menu. Then right-click on the project node again and choose "Run".

The NetBeans will start the application and the Main application window should open:



5. The initial application has a text field containing the URL of the drawing resource, a "Start"/"Stop" button, a status label for displaying application status messages, a drawing list view and a text area for displaying the information about the received SSE events.
6. You can click the "Start"/"Stop" button a few times to see that the status of the application is changing accordingly. Other than changing the status, the application does not do anything useful at the moment, as we haven't implemented the necessary client-side logic yet.
7. Once done, close the application.

Step 2: Exploring the details of SSeClientApp class

1. In the Projects view, double-click the `SseClientApp` class to open it in the editor pane. You should see the class open in "Design" view. Click the "Source" button in toolbar located at the top of the opened `SseClientApp` class editor pane to switch to the source view of the class.
2. The `SseClientApp` class code starts with a no-arg constructor that initializes the GUI components as well as an executor service that is internally used to execute long-running tasks in a thread separate from the Swing event dispatcher thread.
3. The next section contains a generated `initComponents()` method containing the actual Swing component assembly and initialization logic, followed by two event handling methods – `onStartStop(…)` and `onClose(…)`.
   Let's look at the `onStartStop(…)` method in more detail. The method uses the `started` boolean variable to manage the main application state. Whenever a button is clicked, the value of the variable is inverted and content of GUI components is updated based on the new value of the application state. The method contains an `if … else` clause for

separate handling of logic specific to starting and stopping the application. Notice the two most important pieces of this methods code the calls to `connect(url)` and `disconnect(eventSource)` methods. In this exercise we'll focus on implementing those two methods to provide the business logic for the client application.

4. The next section of the class contains the `main(…)` method definition followed by the declaration of the class fields. Let's skip that section and move to the last section that contains a set of methods we need to implement as well as a few helper methods that provide a more convenient manipulation of the state of application's GUI components.
5. We have already introduced the first two methods in this last section – `connect(…)` and `disconnect(…)`. One other method we will be implementing is the `getDrawings(…)` method that is used from within the `upateDrawings(…)` method bellow to update the list of individual drawing nodes in the application's GUI.
6. Feel free to explore the rest of the helper methods. Since we're not going to use them directly in our implementation, we're not going to describe these methods in more detail here.

Step 3: Retrieving the list of drawings using JAX-RS client API

1. First we need to create the JAX-RS client we will use to make the HTTP requests. Open the `SseClientApp` class add a new private field definition:

```
private final Client jaxrsClient;
```

You may place the field definition anywhere in the class, but to keep things organized, you should consider adding the new field right to the top of the existing field declarations.

2. Then update the `SseClientApp()` constructor by adding the initialization code for the newly introduced `jaxrsClient` field right below the line initializing the `executor` field:

```
this.jaxrsClient = ClientFactory.newClient(
                   new ClientConfig().binders(new MoxyJsonBinder()));
```

We'll be re-using the same client instance in our `connect(…)` method whenever the application "Start" button is clicked.

3. Now, we'll proceed with implementing the first part of the logic in the `connect(…)` method that will retrieve the list of drawings from the server-side drawings resource and update the `drawingsListModel` that is backing up the `drawingsList` GUI component.
First, create a new `WebTarget` pointing to the list of drawings and pass the created `WebTarget` instance to the `updateDrawings(…)` method. Add the following to the beginning of the `connect(…)` method:

```
final WebTarget drawingsResource = jaxrsClient.target(drawingsUrl);
updateDrawings(drawingsResource);
```

4. The `updateDrawings(…)` method is passing the created `WebTarget` instance to the `getDrawings(…)` method. We can now implement the `getDrawings(…)` method by using the `WebTarget` instance we just created to make client requests to the drawings resource. We are going to make an `HTTP GET` request to retrieve a list of drawings. Replace the body of the `getDrawings(…)` method with the following code:

```
return drawingsResource.request(MediaType.APPLICATION_JSON)
                       .get(new GenericType<List<Drawing>>() {});
```

   In the above code we are saying we want to make HTTP GET request (hence calling the `get()` method), accepting a response of "application/json" media type from the server (that's what the `request()` method indicates) and we want to unmarshall the response into `List<Drawing>` type. Note we are wrapping this in `GenericType` when passing it to the `get()` method. This is needed to make the information about the generic type parameter available to Jersey during the runtime (due to type erasure in Java, unless we use `GenericType`, Jersey would see only `List` during the runtime, without the `Drawing` type parameter. It would thus not know it should be a list of `Drawing` objects).
5. Let's run the application, click “Start” button and see if it correctly displays the list of drawings.
   Note that, at this point, in case you modify the list of drawings using the browser UI after the application is started, you will need to stop and start the application again to see the refreshed list of drawings.

Step 4: Listening to SSE

One last thing to add is the client code that listens to the server-sent events. Let's add that to our client:

1. Replace the `return null;` line in the `connect(…)` method with the following code:

```
final WebTarget eventsResource = drawingsResource.path("events");
final EventSource eventSource = new EventSource(eventsResource) {
    @Override
    public void onEvent(InboundEvent inboundEvent) {
        String eventData;
        try {
            eventData = "Event "
                        + inboundEvent.getName() + ": "
                        + inboundEvent.getData();
        } catch (IOException ex) {
            eventData = "Failed to process event: " + ex.getMessage();
        }
        appendEventData(eventData);
        updateDrawings(drawingsResource);
    }
};
updateStatus("Listening to the SSE...");
return eventSource;
```

`EventSource` is a Jersey API class similar to the one that's available to JavaScript in HTML5. As the constructor parameter we are passing `WebTarget` pointing to the events URI and we are implementing `onEvent()` method, which gets called for every incoming event. `EventSource` object automatically establishes the connection (by making a `HTTP GET` request to the passed web target) and calls the `onEvent` method and all the registered listeners (besides implementing the `onEvent()` method you can also implement event listeners and register them to the event source object) for every event. This is all asynchronous – i.e. happens on a separate thread – the call to `EventSource` constructor returns immediately.

2. To make sure our application closes the event source when stopped, implement the `disconnect(…)` method to simply close the event source instance as follows:

```
eventSource.close();
```

3. At this point we are done. Run the application and try adding/removing drawings using the web browser – you should see events being printed out to the application GUI and the list of drawings should get updated as you modify the list of drawings through the web interface.

This concludes the last exercise of this lab, which served as a quick introduction to the basics of the JAX-RS and Jersey client-side programming model.

Summary

In this lab you got a sneak peek of some of the new features coming in JavaEE 7, such as Java API for WebSocket, Java API for JSON Processing and JAX-RS 2.0 Client API. We've also seen how to utilize server-sent events support that comes with Jersey – the JAX-RS reference implementation. Here are some additional resources that can help you get more information on these technologies and build your own applications utilizing these:

GlassFish:

- Project website: http://glassfish.java.net
- Community blog: http://blogs.oracle.com/theaquarium

Jersey/JAX-RS:

- Project website: http://jersey.java.net
- JAX-RS project website: http://jax-rs-spec.java.net

Tyrus/WebSocket API

- Project website: http://tyrus.java.net
- JSR project website: http://websocket-spec.java.net

JSON Processing

- Implementation project website: http://jsonp.java.net
- Specification project website: http://json-processing-spec.java.net

Appendix: Setting up the Lab Environment

This lab was developed and tested with the following configuration:

- JavaSE 7 (http://www.oracle.com/technetwork/java/javase/downloads/index.html)
- Chrome web browser (https://www.google.com/intl/en/chrome/browser/)
- Postman REST Client extension for Chrome (https://chrome.google.com/webstore/detail/fdmmgilgnpjigdojojpjoooidkmcomcm)
- NetBeans 7.2 (http://netbeans.org/downloads/index.html)
- GlassFish 4.0-b57 promoted build (http://dlc.sun.com.edgesuite.net/glassfish/4.0/promoted/glassfish-4.0-b57.zip)

To be able to easily deploy and run the application project from NetBeans, you need to register the GlassFish 4.0-b57 promoted build in NetBeans as follows:

1. Click on the Services tab in NetBeans.
2. Right-click on Servers, choose Add Server… in the pop-up menu.
3. Select GlassFish Server 3+ in the Add Server Instance wizard, set the name to GlassFish 4.0-b57 and click Next.
4. Browse to where you installed the GlassFish build (point to the glassfish3 directory that got created when you unzipped the above archive), click Next.
5. Click Finish on the next screen.