MAKE THE FUTURE JAVA

JavaOne™

September 30 – October 4, 2012
Hilton San Francisco

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

Martin Matula
Sr. Development Manager

Updated by Peter Doschkinow
Java Architect

ORACLE®

Change History

| Version	Date	Name	Notes
00.01	Oct, 2012	Martin Matula	Initial version
00.02	May, 2013	Peter Doschkinow	Document and code updated to match new APIs (Jersey, JSON, WebSocket, JAX-RS) JavaFX client added (exercise5)

Table of Contents

ORACLE®

ORACLE®

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:
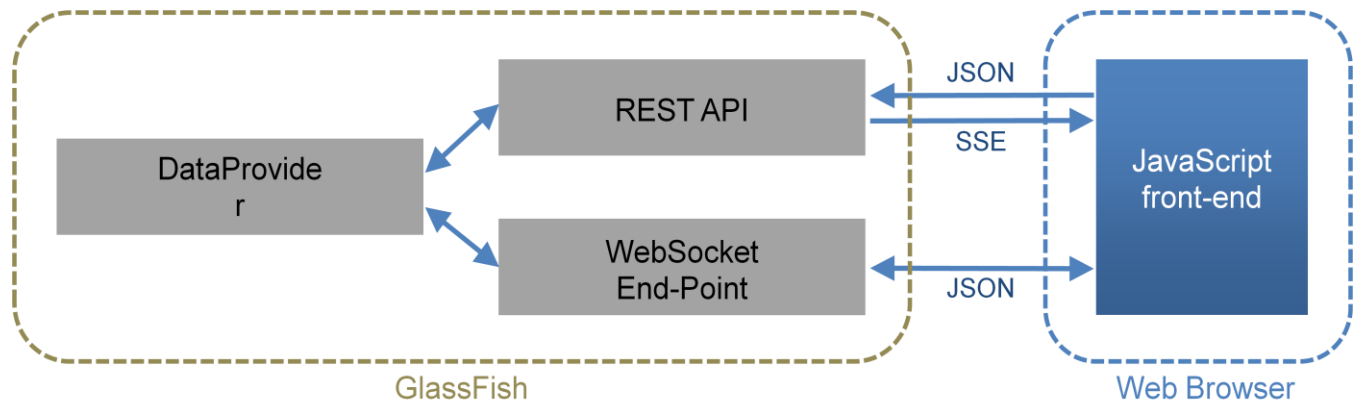
The application will be deployed to GlassFish, and will consist of a JavaScript front-end running in the browser, communicating with the Java back-end running on the server. The back-end is going to expose REST API and web sockets. The front-end is going to utilize these to access the data and render 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.

ORACLE®

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 4.0-b84", 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 a Jersey library and GlassFish 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.media</groupId>
        <artifactId>jersey-media-moxy</artifactId>
        <version>${jersey.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>${javaee7.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

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 "Application" class, which is the JAX-RS API configuration class.
NOTE: If REST Resources Configuration dialog appears while performing this step in NetBeans, simply click Cancel.
The resulting class should look as follows:

ORACLE®

```
@ApplicationPath("api")
public class JaxrsApplication extends javax.ws.rs.core.Application
{

    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(DrawingsResource.class);
        classes.add(MoxyJsonFeature.class);
        return classes;
    }
```

}As you can see, we register the DrawingResource class as a JAX-RS resource class and enable MoxyJsonFeature which takes care of the automatic conversion between Drawing objects and their JSON representation.

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.

ORACLE®

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))
```

ORACLE®

```
        .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 addEventOutput(EventOutput eo) {
    sseBroadcaster.add(eo);
}
```

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(SseFeature.SERVER_SENT_EVENTS)
public EventOutput getEvents() {
    EventOutput eo = new EventOutput();
    DataProvider.addEventOutput(eo);
    return eo;
}
```

As you can see, this method adds another sub-resource mapped to …/drawings/events URI, it produces a response of type text/event-stream (SseFeature.SERVER_SENT_EVENTS constant value), which is the standard media type for SSE and all it does is creating a new EventOutput instance, registering it to our broadcaster (through the DataProvider.addEventOutput() 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 configuration 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 and add SseFeature.classto the list of registered classes like this:

```
classes.add(SseFeature.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:

```javascript
// 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 (EventOutput) 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.

ORACLE®

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 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. Similarly to JAX-RS, the web socket API is annotation-based. To expose a class as a server web socket end-point, you need to annotate it with the standard @ServerEndpoint annotation. So, create a new class named DrawingWebSocket and annotate it with @ServerEndpoint, setting the path-value parameter of the annotation to "/websockets/{id}", as that is the URI at which the class should listen for incoming connections:

    ```
    @ServerEndpoint(
        value = "/websockets/{id}"
    )
    public class DrawingWebSocket {
    }
    ```

    In this way we will have the drawing id injected as parameter into our handler methods by the web socket runtime automatically. 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 @OnOpen annotation (the name of the method is not significant, but let's call it onOpen):

    ```
    @OnOpen
    public void onOpen(@PathParam("id") Integer drawingId, Session session) {
    ```

    } The web socket runtime passes the drawing id and a session object as parameters to this method. The session 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.

2. As part of onOpen() method we are going to add the new WebSocket session to the map of drawing sessions that we maintain in the DataProvider class:

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

3. Now we can implement a method for handling incoming messages on an existing connection. That's what @OnMessage 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:

```
@OnMessage
public void shapeCreated(@PathParam("id") Integer drawingId,
Drawing.Shape shape, Session session) {
    DataProvider.addShape(drawingId, shape);
}
```

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

```
@OnClose
public void onClose(@PathParam("id") Integer drawingId, Session
session) {
    DataProvider.removeWebSocket(drawingId, 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 API coming in JavaEE 7 – Java API for JSON Processing.
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:

```
public Drawing.Shape decode(String s) throws DecodeException {
    // temporary workaround for a web socket implementation issue

Thread.currentThread().setContextClassLoader(getClass().getClassLo
ader());

    Drawing.Shape shape = new Drawing.Shape();
     try (JsonReader reader = Json.createReader(new
StringReader(s))) {
        JsonObject object = reader.readObject();
        shape.x = object.getInt("x");
        shape.y = object.getInt("y");
        shape.type = Drawing.ShapeType.valueOf(
                object.getString("type"));
        shape.color = Drawing.ShapeColor.valueOf(
                object.getString("color"));
    }

    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:

```java
@Override
public String encode(Drawing.Shape object) throws EncodeException
{
    // temporary workaround for a web socket implementation issue

Thread.currentThread().setContextClassLoader(getClass().getClassLo
ader());

    StringWriter result = new StringWriter();

    try (JsonGenerator gen = Json.createGenerator(result)) {
            gen.writeStartObject()
                .write("x", object.x)
                .write("y", object.y)
                .write("type", object.type.toString())
                .write("color", object.color.toString())
                .writeEnd();
    }

    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 @ServerEndpoint annotation as follows:

```java
@ServerEndpoint(
        value = "/websockets/{id}",
        decoders = ShapeCoding.class,
        encoders = ShapeCoding.class)
```

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:

ORACLE®

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.getBasicRemote().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.getBasicRemote().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 in DrawingWebSocket.onOpen():

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

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

```
@OnClose
public void onClose(Session 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:
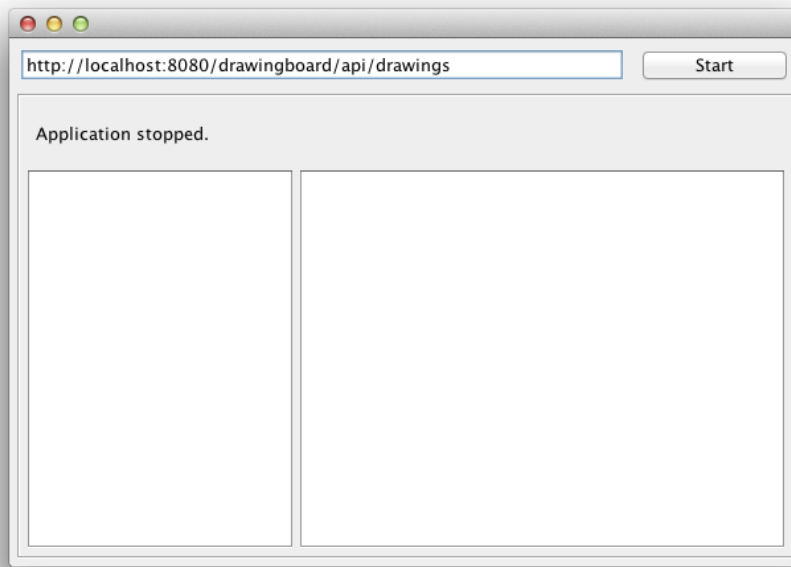
5. The initial application has a text field containing the URL of the drawing resource, a "Start"/"Stop" button, a status label for displaying application status messages, a drawing list view and a text area for displaying the information about the received SSE events.
6. You can click the "Start"/"Stop" 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

ORACLE®

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 = ClientBuilder.newClient().register(MoxyJsonFeature.class);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

ORACLE®

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 exercise 4 of this lab, which served as a quick introduction to the basics of the JAX-RS and Jersey client-side programming model.

Exercise 5: Implementing a JavaFX based client

In this last exercise of our lab we will develop a JavaFX-based client with the same functionality as the AngularJS-based browser client. We will use the JavaFX WebView component (which wraps a WebKit implementation) to display the html pages and execute the embedded JavaScript. Since the underlying WebKit does not support WebSocket and SSE, we will implement this in Java. We will only minimally modify the JavaScript code to replace the WebSocket/SSE handling with hooks to the Java code using the Java – JavaScript bridge as documented at http://docs.oracle.com/javafx/2/api/javafx/scene/web/WebEngine.html and http://docs.oracle.com/javafx/2/webview/jfxpub-webview.htm. The HTML and CSS code will not be modified. Exercise 5 is a nice example of a hybrid Java/JavaScript application showing how JavaFX can be used to extend existing and non-trivial JavaScript applications.

Step 1: Explore the initial JavaFX project

1. Open the project **drawingboard-javafx** from **<lab_root>** directory.
2. Expand the "Drawing Board JavaFX"/"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 for a client-side WebSocket and SSE implementation:

```
<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>tyrus-client</artifactId>
```

```
        <version>${tyrus.version}</version>
</dependency>
<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>tyrus-container-grizzly</artifactId>
    <version>${tyrus.version}</version>
</dependency>
<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. To keep the pom.xml simple we will do a little hack to get the JavaFX runtime on the Java classpath, so that we don't have to define a special dependency for JavaFX: just copy <jdk_home>/jre/lib/jfxrt.jar to <jdk_home>/jre/lib/ext. By the way, in JDK8 jfxrt.jar is located exactly there. If you do the HOL on a VirtualBox image, this is probably already done for you.

4. Expand the "Drawing Board JavaFX/Source Packages" and "Drawing Board JavaFX/Other Sources" nodes of the project. You can see four classes below `com.mycompany.websocket.draw.sample.javafx` and two resources below `src.main.resources`. `JavaFXDrawing` and `DrawingController` together with `drawing.xml` and `drawing.css` implemnt the skeleton of the JavaFX application, that was built with the JavaFX SceneBuilder. Explore the application's artefacts by double-clicking on them. Note that a double-click on `drawing.xml` starts the SceneBuilder.

5. 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. Make sure that the web application from exercise 3 is still deployed and GlassFish is running. Open a browser against it and add a couple of drawings. Then right-click on the project node again and choose "Run". You will note that the JavaFX client shows the list of drawings created in the browser, but when you click on a drawing name, you see something like:
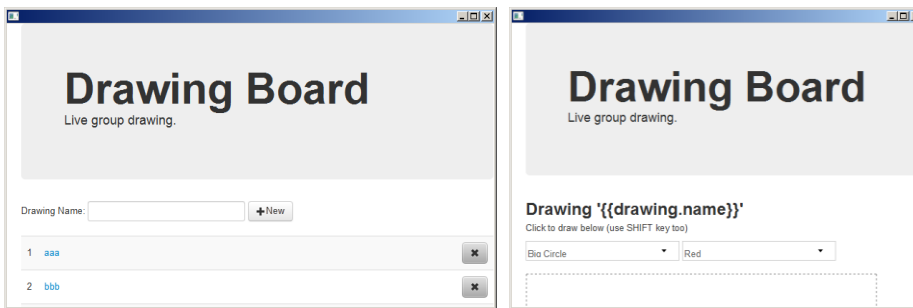
which shows that something is broken. Also when you go back to the drawing list in the JavaFX app and add some more drawings, they will appear in the browser, but won't be reflected in the JavaFX view. This is because for now only JAX-RS is working(the AngularJS code in WebView) but the WebSocket and SSE implementations are not there yet.

6. Once done, close the application.

Step 2: Ajust the Drawing Board Web Application

1. Starting point is the web application we developed in exercise 3. The WebKit integrated in WebView treats JavaScript code differently. While most browsers today do not stop the execution of JavaScript when referencing non existing functions or properties, this seem not to be the case with the JavaScript engine in WebView. That is why we need to introduce a variable in controllers.js that is used to tell us if the the JavaScript code is executing in a normal browser or JavaFX, so that we can avoid the problem of a missing WebSocket/SSE implementation in JavaFX/WebKit:

```
var javaFXClient = navigator.appVersion.indexOf("JavaFX") > 0;
```

2. This results in the following modifications in the MainController function of controllers.js :

```
// listens to server-sent events for the list of drawings
if (!javaFXClient)
    $scope.eventSource = new
EventSource("/drawingboard/api/drawings/events");
…
if (!javaFXClient) {
    $scope.eventSource.addEventListener("create", eventHandler,
false);
    $scope.eventSource.addEventListener("update", eventHandler,
false);
    $scope.eventSource.addEventListener("delete", eventHandler,
false);
}

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

```
    if (!javaFXClient)
        $scope.eventSource.close();
});
```

3. In DrawingController from controllers.js we need first to escape the WebSocket API in JavaScript for JavaFX clients and next to redirect its usage to Java by using new JavaScript functions(starting with window.) calling into Java, that will be registered when we setup the WebView component in JavaFX:

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

// clean up
$scope.$on("$destroy", function(event) {
    if (!javaFXClient){
        // sometimes when this function is called, the websocket
is already closed
        if ($scope.websocket.readyState > 0)
            $scope.websocket.close();
    }
    else {
        // sometimes when this function is called,
$scope.drawing.id is undefinded
        if ($scope.drawing.id > 0)
            window.webSocketClose.close($scope.drawing.id);
    }
});
```

4. Run the modified Drawing Board Web Application. The browser clients should work as in exercise 3. The Drawing Board JavaFX client won't work better then in Step 1, since we still have to implement a WebSocket- and SSE-Client in JavaFX and to register and implement in Java the new JavaSctipt functions(window.*) we used above.

Step 3: Implement a WebSocket- and SSE-client in the Drawing Board JavaFX project

1. We are going to implement the WebSocket client in WSClient.java in the same package as DrawingController and JavaFXDrawing. It will expose a WebSocket session for sending of drawings in JSON format trough the Java – JavaScript bridge from controllers.js running in the WebView component. WSClient.java will also receive JSON representations of drawings through WebSocket broadcasts from the server and call JavaScript to make them appear in the WebView. Note that the JavaScript code has to be executed on the JavaFX application thread, hence the usage of Platform.runLater(). Here is the code for WSClient.java:

```java
package com.mycompany.websocket.draw.sample.javafx;


/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
import javafx.application.Platform;
import javafx.scene.web.WebView;
import javax.websocket.ClientEndpoint;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;

/**
 *
 * @author pdos
 */
@ClientEndpoint
public class WSClient {

    private Session session;
    private WebView webview;

    public WebView getWebview() {
        return webview;
    }

    public Session getSession() {
        return session;
    }

    public WSClient(WebView webview) {
        this.webview = webview;
    }
```

```
    @OnOpen
    public void onOpen(Session session) {
        this.session = session;
        System.out.println("Connection had opened.");
    }

    @OnMessage
    public void onMessage(String message) {
        final String js_script =
            "var elem =
angular.element(document.getElementsByClassName(\"form-
horizontal\"));"
            + "var sc = elem.scope();"
            + "sc.drawShape(eval(" + message + "));";
        System.out.println("script to execute: " + js_script);
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                webview.getEngine().executeScript(js_script);
            }
        });
    }

    @OnClose
    public void closeConnection(Session session) {
        System.out.println("Connection had closed.");
    }
}
```

2. Next we implement a SSE client in DrawingsEventSource.java using Jersey's client API for SSE. It receives SSE broadcasts from the server about adding and removing of drawings and calls JavaScript code in controllers.js to refresh the drawings model in AngularJS. Note that the JavaScript code has to be executed on the JavaFX application thread, hence the usage of Platform.runLater().

```
package com.mycompany.websocket.draw.sample.javafx;

import java.io.IOException;
import javafx.application.Platform;
import javafx.scene.web.WebEngine;
import javax.ws.rs.client.WebTarget;
import org.glassfish.jersey.media.sse.EventSource;
```

```java
import org.glassfish.jersey.media.sse.InboundEvent;

/**
 *
 * @author pdos
 */
class DrawingsEventSource extends EventSource {

    private WebEngine engine;
    private String js_script =
            "var elem =
angular.element(document.getElementsByClassName(\"table\"));"
            + "var sc = elem.scope();"
            + "var svc = elem.injector().get('DrawingService');"
            + "sc.drawings = svc.query();"
            + "sc.$apply();";

    public DrawingsEventSource(WebTarget target, WebEngine engine)
throws NullPointerException {
        super(target);
        this.engine = engine;
    }

    @Override
    public void onEvent(InboundEvent inboundEvent) {
        try {
            System.out.println("Event "
                    + inboundEvent.getName() + ": "
                    + inboundEvent.getData());
            System.out.println("script to execute: " + js_script);
            Platform.runLater(new Runnable() {
                @Override
                public void run() {
                    engine.executeScript(js_script);
                }
            });
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Step 4: Implement the Java – JavaScript bridge in the Drawing Board JavaFX project

1. Extend DrawingController.java with a couple of private fields to wire references to the WebSocket- and SSE clients we implemented above. We will also need a WebSocketContainer for bootstrapping of a WebSocket communication and a HashMap of active WebSocket sessions that correspond to drawings that are being edited():

```
private DrawingsEventSource eventSource;
private WSClient wsClient;
private WebSocketContainer container =
ContainerProvider.getWebSocketContainer();
private HashMap<String, WSClient> webSocketSessions = new
HashMap<>();
```

2. Implement setupEngine() method. Here we define the Java – JavaScript Bridge for JavaScript → Java callbacks that come from controllers.js and initialize the SSE client side. That means that we map JavaScript functions to Java methods following specific conventions, as described in http://docs.oracle.com/javafx/2/api/javafx/scene/web/WebEngine.html. Essentially we define a public Java class and a global JavaScript function that correspond to each other and have a single method with the same signature. It is important that this code is executed after the page that contains the JavaScript code was loaded(WorkerStateSUCCEDED):

```
private void setupEngine() {
    engine.getLoadWorker().stateProperty().addListener(
            new ChangeListener<Worker.State>() {
        @Override
        public void changed(ObservableValue<? extends
Worker.State> ov,
                Worker.State oldState, Worker.State newState) {
            if (newState == Worker.State.SUCCEEDED) {
                JSObject global =
                        (JSObject) engine.executeScript("window");
                global.setMember("webSocketOpen", new
WebSocketOpen());
                global.setMember("webSocketClose", new
WebSocketClose());
                global.setMember("webSocketSend", new
WebSocketSend());
                initSSE();
            }
        }
    });
}
```

3. Implement initSSE():

```java
private void initSSE() {
    Client client = ClientBuilder.newClient();
    // create a web target pointing to the drawings resource
    WebTarget t =
client.target("http://localhost:8080/drawingboard/api/drawings");
    // start listening to SSE
    eventSource = new DrawingsEventSource(t.path("events"),
engine);
}
```

4. Implement the callback methods corresponding to the Java – JavaScript bridge established in step 4.2 taking into account how they are called in DrawingController from controllers.js. Note that the WebSocket code is completely standard based and implementation independent():

```java
public class WebSocketSend {

    public void send(String drawing) {
        try {
            System.out.println("sending drawing: " + drawing);

wsClient.getSession().getBasicRemote().sendText(drawing);
        } catch (IOException ex) {

Logger.getLogger(DrawingController.class.getName()).log(Level.SEVE
RE, null, ex);
        }
    }
}

public class WebSocketOpen {

    public void open(String baseURL, String drawingId) {
        try {
            System.out.println("Setting WebSocket to " + baseURL +
drawingId);
            wsClient = new WSClient(webview);
            URI clientURI = new URI(baseURL + drawingId);
            container.connectToServer(wsClient, clientURI);
            webSocketSessions.put(drawingId, wsClient);
        } catch (URISyntaxException | DeploymentException |
IOException ex) {
```

ORACLE®

```
Logger.getLogger(DrawingController.class.getName()).log(Level.SEVE
RE, null, ex);
        }
    }
}

public class WebSocketClose {

    public void close(String drawingId) {
        System.out.println("Closing socket for drawing " +
drawingId);
        WSClient client = (WSClient)
webSocketSessions.get(drawingId);
        if (client == null) return;
        try {
            client.getSession().close();
        } catch (IOException ex) {

Logger.getLogger(DrawingController.class.getName()).log(Level.SEVE
RE, null, ex);
        }
    }
}
```
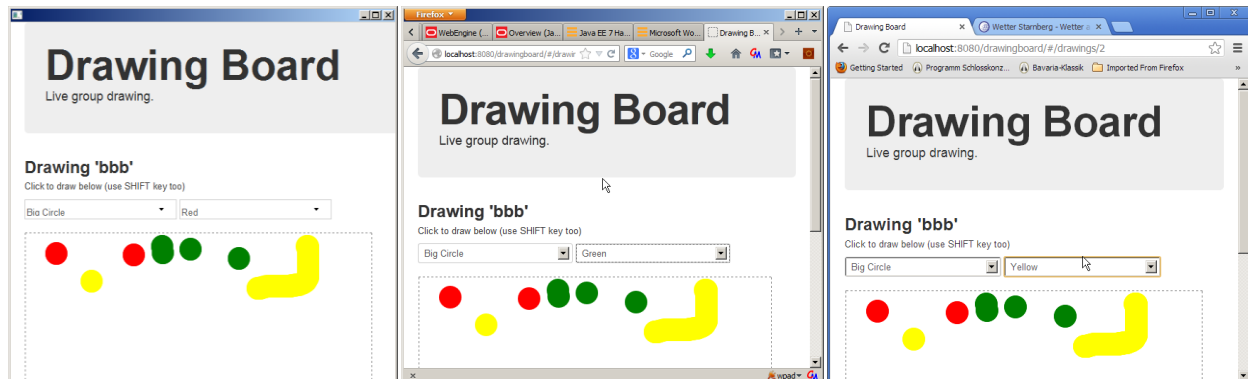
5. Build and run the Drawing Board JavaFX project. Play with different concurrently running browser or JavaFX clients. The experience should be the same:

This concludes exercise 5 of this lab, which served as a quick introduction to a hybrid(Java-JavaScript) JavaFX application using the client-side programming model of WebSocket and Jersey's Server-Sent-Events and an AngularJS application.

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

JavaFX

- JavaFX documentation: http://docs.oracle.com/javafx/index.html

ORACLE®

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 7.3 (http://netbeans.org/downloads/index.html)
- GlassFish 4.0-b84 promoted build (http://dlc.sun.com.edgesuite.net/glassfish/4.0/promoted/glassfish-4.0-b84.zip)

To be able to easily deploy and run the application project from NetBeans, you need to register the GlassFish 4.0-b84 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-b84 and click Next.
4. Browse to where you installed the GlassFish build (point to the glassfish4 directory that got created when you unzipped the above archive), click Next.
5. Click Finish on the next screen.
6.