EDUARDO MORANCHEL AND EDGAR MARTINEZ

BIO

Java EE 7

Embracing HTML5

Explore new features for creating next-generation internet applications.

Java EE 7 supports new and evolving standard technologies such as WebSocket, HTML5, and JSON, providing an API for each one.

This article first presents some background on WebSocket and then offers a Web application to demonstrate the new platform features.

Note: The source code for the example described in this article can be downloaded as a Maven project here.

WebSocket Background

Modern Web applications have grown in interactivity, demanding greater communication between the browser and the Web server. At first, constant communication was achieved by servlets with long polling in the browser, but problems in the server appeared because the threading model of the Web container demanded that every request be handled by a thread until the response is written. Then, asynchronous servlets came with Java EE 6 providing a more efficient approach that allowed requests to be processed asynchronously.

The Polling, long pooling, and Comet/Ajax approaches simulate a full-duplex communication between the server and the browser. WebSocket is a full-duplex and bidirectional API, which not only involves the server but the Web browser, too.

WebSocket changes the way the Web server reacts to client requests: instead of closing the connection, it sends a 101 status and leaves the connection open, expecting messages to be written on the stream and to be able to write messages, as well. The client also expects messages to be written and is also able to write messages at its end of the stream. This allows both ends to read and write to the stream and communicate in real time.

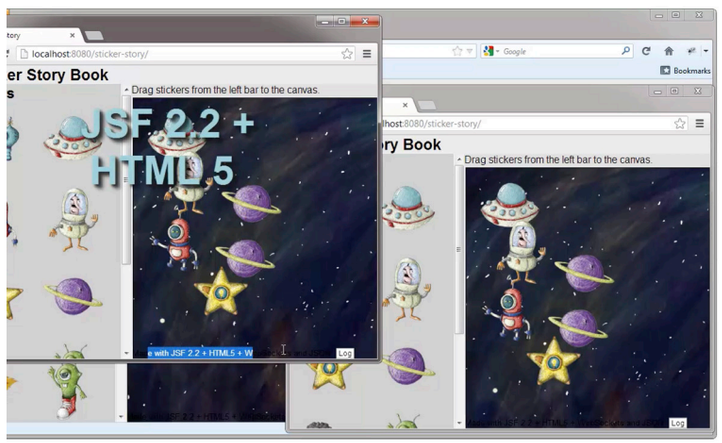
Before Java EE 7, implementing a WebSocket was Web container–dependent: you could write a WebSocket for Tomcat, but if you wanted to migrate to Jetty, you had to reimplement it for the new server.

Note: As with every new technology, WebSocket is being adopted by different Web browsers at different times. You can check this page for more information about browser compatibility.

Overview of the Sticker Story Application

Let's start with a simple example to see how the things we have discussed so far work.

This article showcases a sticker story Web application that children can use to create a story collaboratively by dragging stickers into a book or canvas. At the instant a sticker is placed into a child's



This video provides a demo of the sticker story application.

Figure 1

book, the sticker is drawn within other children's books. Watch the video to see the application in action.

Figure 1 shows the home page for the sticker story Web application. When you drag a sticker from the sidebar on the left to the canvas and drop it, the sticker is rendered into all the Web browsers that are open at that time, as shown in the video.

Obtaining and Running the Sticker Story Application

1. Download GlassFish v4 promoted build 79 (glassfish-4.0-b79.zip). GlassFish can be obtained here as a zip archive.
2. Unzip GlassFish into a directory of your choice.
3. Download and install NetBeans. The version used in this article is NetBeans 7.3.
4. Download the sticker story Maven project link to sticker-story.zip.
5. Expand the project into a directory of your choice.
6. Start NetBeans, and on the **Services** tab, right-click the **Servers** node, and then click **Add Server**.
7. From the Server list, select **GlassFish Server 3+**, and then click **Next**.
8. In the **Server Location** field, enter the path to the directory where you unzipped GlassFish v4, and then click **Finish**.
9. Select **File** -> **Open Project**.
10. From the Open Project dialog box, open the sticker story Maven project.
11. Right-click the project and select **Run**. Select the recently installed GlassFish server (GlassFish Server 3+) when prompted.

You can now point your Web browser to http://localhost:8080/sticker-story to see a page similar to that of **Figure 1**. Make sure your browser is compatible with WebSocket by checking this page.

Application Contents

The application contains the following files and Java classes:

- index.html is the home page and contains JavaServer Faces (JSF) 2.2 and HTML5 code.
- The org.sticker.jsf.StickerSheet class is the managed bean used by the home page to display all the available stickers.
- story-page.js contains the JavaScript code for the WebSocket client and the drag-and-drop functionality.
- The org.sticker.websocket.Sticker class is the object being sent and received by the WebSocket.
- The org.sticker.websocket.StickerEncoder and org.sticker.websocket.StickerDecoder classes transform the data sent by a WebSocket into objects.
- The org.sticker.websocket.StoryWebSocket class is the WebSocket handler. It also stores the stickers in the current story book.

Combining HTML5 with the JSF 2.2–Friendly HTML Markup

Just as Ed Burns, spec lead of JSF 2.2 (JSR 344), mentioned in his blog, JSF 2.2 allows page authors to have complete control of the HTML rendering by using the Friendly Markup feature of JSF 2.2. It also facilitates the separation of markup and business logic.

Let's see an example. The view of the sticker story application is provided using HTML5 and JSF. **Listing 1** shows the content of the index.xhtml file, which is mostly HTML5; JSF is used to render the sticker images in the sidebar of the browser using the <h:graphicImage> tag.

- JSF 2.2 lets you pass through HTML attributes. There are two ways to do this: with the namespace for the pass-through attributes, which is http://java.sun.com/jsf/passthrough, or by using the child TagHandler f:passThroughAttribute (or f:passThroughAttributes for multiple attributes). In our example, we use the former.

In **Listing 1**, notice the <h:graphicImage>tag. There are three attributes (draggable, ondragstart, and data-sticker) in our graphicImage tag that are not part of the attributes for graphicImage but that we need in order to make our stickers work. Therefore, we use the prefix p: so that JSF pass-through adds these three attributes. **Note:** Because graphicImage is a JSF markup, we have used the namespace http://java.sun.com/jsf/passthrough. If you want to pass through elements into a non-JSF-aware markup, use the namespace http://java.sun.com/jsf. Look for some examples in **Listing 1**, such as head or script tags.

Listing 2 shows the managed bean, which provides the names of the stickers. The @Named annotation exposes the bean to JSF, and later the bean method is called using the #{stickerSheet.allStickers} expression as the value in the JSF ui:repeat tag in the index.xhtml file shown in **Listing 1**.

DID YOU KNOW?

WebSocket is a full-duplex and bidirectional API, which not only involves the server but the Web browser, too.

Writing Drag-and-Drop Code

In HTML5, drag-and-drop functionality is simplified and requires specific events that are handled using JavaScript code that is divided into two parts: the dragged-object events and the receiving-object events.

The dragged object (that is, the sticker) must define the ondragstart event (see **Listing 1**) and handle it using JavaScript. As we discussed earlier, in JSF 2.2, you can use the pass-through parameters to add this event. The JavaScript code in **Listing 3** specifies what will be executed when the drag starts.

In drag-and-drop interactions, you might have to use an intermediate object to store the data being transferred between components. In JavaScript, you do this by storing the data inside the drag event.dataTransfer property. Because this property allows only strings to be stored, the easiest way to store data is using JSON.

The JavaScript code for the drag event creates the sticker representation, gets from the screen the coordinates of the location to which the sticker was dragged, and—using another HTML5 feature—gets the sticker image filename from the tag's data-sticker

LISTING 1 LISTING 2 / LISTING 3

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:jsf="http://java.sun.com/jsf"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p=" http://java.sun.com/jsf/passthrough">
 <head jsf:id="head">
  <title>Sticker Story</title>
  <script jsf:target="body" jsf:name="story-page.js"/>
  <link jsf:name="styles.css" rel="stylesheet" type="text/css" />
 </head>
 <body jsf:id="body">
  <header>
   <h1>Sticker Story Book</h1>
</header>
  <nav>
  Drag stickers from the left bar to the canvas.
 </nav>
 <aside>
  <h2>Stickers</h2>
  <div id="stickerContainer">
   <ui:repeat var="imgName"
      value="#{stickerSheet.allStickers}">
    <h:graphicImage library="stickers" name="#{imgName}"
          style="float:left" p:draggable="true"
          p:ondragstart="drag(event)" p:data-
          sticker="#{imgName}" />
   </ui:repeat>
  </div>
 </aside>
 ...
 </body>
</html>
```

Download all listings in this issue as text

attribute. In HTML5, you can use custom attributes to store application-specific data by adding the data- prefix to the attribute; in JSF 2.2, you use the pass-through attributes to add such attributes.

For the drop action, the drop target object (that is, the receiving end) must define two event handlers.

One is the ondragover event handler, which must handle the drag-over (hover action) of the mouse to determine whether a drag is valid. To allow drops with further validation, the code just calls ev.preventDefault(), since the default behavior is to deny drops.

The second event handler is ondrop, which handles the drop event when the mouse button is released while something is being dragged. The JavaScript code that handles this event (see **Listing 4**) begins by preventing the default behavior (denying the drop) and then gets the data from dataTransfer. Because we are transferring text, we have to convert it to an object using the JSON.parse method, which is a standard JavaScript method. Next, we convert the dragged object to the actual object that we are going to send to the server. The drop method in **Listing 4** also calculates the coordinate for the position of the upper left corner of the sticker on the canvas and adds the name of the sticker to the object. The object then is converted to a JSON string and finally sent using the WebSocket.send method.

A WebSocket Client with JavaScript

Using WebSocket, the sticker story application sends data to the server when a drag-and-drop action occurs. So how is a WebSocket connection established? Let's see.

After the page loads, the initialize method is called (see **Listing 3**), and the first thing this method does is draw the background for the application using the HTML5 canvas. Next, it opens the WebSocket connection by using the statement socket = new WebSocket("ws://localhost:8080/sticker-story/story/notifications") to establish and open the communication. Note that it uses the WebSocket protocol instead of HTTP and the path is absolute.

Note: If you are running the Web server in a different port, you will need to change this line to use your server's port.

The next line in **Listing 3** defines what the client will call when the server pushes a message to the browser.

Now, take a look at the onSocketMessage method in **Listing 4**. This method will handle

LISTING 4

```
function drop(ev) {
 ev.preventDefault();
 var bounds = document.getElementById("board")
.getBoundingClientRect();
 var draggedText = ev.dataTransfer.getData("text");
 var draggedSticker = JSON.parse(draggedText);
 var stickerToSend = {
  action: "add",
  x: ev.clientX - draggedSticker.offsetX - bounds.left,
  y: ev.clientY - draggedSticker.offsetY - bounds.top,
  sticker: draggedSticker.sticker
 };
 socket.send(JSON.stringify(stickerToSend));
 log("Sending Object " + JSON.stringify(stickerToSend));
}

// Web socket on message received:
function onSocketMessage(event) {
 if (event.data) {
  var receivedSticker = JSON.parse(event.data);
  log("Received Object: " + JSON.stringify(receivedSticker));
  if (receivedSticker.action === "add") {
   var imageObj = new Image();
   imageObj.onload = function() {
    var canvas = document.getElementById("board");
    var context = canvas.getContext("2d");
    context.drawImage(imageObj,
receivedSticker.x, receivedSticker.y);
   };
   imageObj.src =
"resources/stickers/" + receivedSticker.sticker;
  }
 }
```

Download all listings in this issue as text

WebSocket messages from the server side.

The sticker story application sends JSON objects as text using the WebSocket connection. Therefore, when a message is received, it is transformed to a Sticker object and the onSocketMessage method does the following:

- Transforms the data
- Gets the image for the sticker
- Draws the image on the canvas using the coordinates from the sticker

And because the connection is open at this time, sending a message to the server is done using the send method in the WebSocket JavaScript object. It's just that simple.

WebSocket for Real-Time Communication

Creating a WebSocket in Java EE 7 is very easy. You just need to create a couple of annotations and your WebSocket will be configured.

In the sticker story application, the class configured as a WebSocket is the StoryWebSocket class (see **Listings 5a** and **5b**).

Notice the @ServerEndpoint annotation at the top of the class declaration. Just as with servlets, you have to define a mapping. For a WebSocket, the value of the @ServerEndpoint defines the mapping of the WebSocket. This means that you can declare a WebSocket just by using the following and the server will wait for connections at the /websocketMapping URL to connect to the WebSocket:

```
@ServerEndpoint(
"/websocketMapping")
```

For now, skip the encoders and decoders attributes and head over to the methods of the class. To define the methods that will handle the WebSocket events, you use annotations as well.

The @OnOpen, @OnClose, and @OnError annotations are related to the lifecycle of the WebSocket connection. You can annotate a method with any of them and the method will be called when a connection opens or closes or when there is an error in the connection (an exception was not handled).

Our application uses the @OnOpen annotation to push the stickers to the connected user. This way, a newly connected user receives all the stickers previously posted on the canvas by other users.

The @OnMessage annotation is the core of the WebSocket implementation. The annotated method will be called whenever the client sends a message; in the appli-

LISTING 5a LISTING 5b

```
package org.sticker.websocket;

import java.io.IOException;
import java.util.*;
import java.util.logging.*;
import javax.websocket.EncodeException;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;

@ServerEndpoint(
    value = "/story/notifications",
    encoders = {StickerEncoder.class},
    decoders = {StickerDecoder.class})
public class StoryWebSocket {

 private static final List<Sticker> stickers =
Collections.synchronizedList(new LinkedList<Sticker>());

@OnOpen
 public void onOpen(Session session) {
  synchronized (stickers) {
   for (Sticker sticker : stickers) {
    try {
     session.getBasicRemote().sendObject(sticker);
    } catch (IOException | EncodeException ex) {
     Logger.getLogger(StoryWebSocket.class.getName()).log
(Level.SEVERE, null, ex);
    }
   }
  }
 }
```

Download all listings in this issue as text

cation, the client sends stickers and then the WebSocket stores the added sticker and pushes the sticker to all the connected peers.

Notice that the @onMessage method receives a Sticker object (see **Listing 6**) and a Sticker object is written to the other peers using the following method:

```
session.getBasicRemote()
.sendObject()
```

We discussed the fact that WebSocket sends and receives text. So, why do the methods receive a Sticker object, and why are we writing Sticker objects to the sessions? The answer lies in the @ServerEndpoint declaration, specifically in the encoders and decoders attributes.

A WebSocket may use a decoder to transform the text message into an object and then handle it in the matching @OnMessage method, and whenever an object is written to the session, a WebSocket may use an encoder to convert the object to text and send it to the client.

URL path parameters. You can set URL path parameters by adding them to the @ServerEndpoint mapping. Do so by enclosing the parameter name inside brackets:

```
/pathToWebSocket/
{parameter1}
```

To use it, add a string parameter to any annotated method and add the @PathParam("parameterName") annotation to the method parameter. In the example in **Listings 5a** and **5b**, you would add the @PathParam("parameter1") string parameter. You can add as many parameters as you need.

And while we are at it, for the methods annotated using any WebSocket–related annotation, you can receive the WebSocket session as a parameter. In the application, we add it as a parameter, but it is not needed and it can be omitted completely.

The @OnMessage annotation applies to a method that can have an extra parameter. This parameter is the message being received and can take the form of any object decoded using the declared decoders in the @ServerEndpoint annotation or String, if no decoders are specified.

EASY AS 1, 2, 3

Creating a WebSocket in Java EE 7 is very easy. You just need to create a couple of annotations and your WebSocket will be configured.

LISTING 6

```
package org.sticker.websocket;

public class Sticker {
 private int x;
 private int y;
 private String image;

 public Sticker() {
 }
 public int getX() {
  return x;
 }

 public void setX(int x) {
  this.x = x;
 }

 public int getY() {
  return y;
 }

 public void setY(int y) {
  this.y = y;
 }
 public String getImage() {
  return image;
 }
 public void setImage(String image) {
  this.image = image;
 }
}
```

Download all listings in this issue as text

Using the New JSON API with WebSocket

The new Java API for JSON Processing (JSR 353) is part of Java EE 7, and we are using it in our example.

The sticker story application communicates with the client and the WebSocket server by using a JSON string representation of objects. The WebSocket uses decoders and encoders to transparently convert the text sent to the WebSocket into a Sticker object and to convert Sticker objects sent to the client into text.

In the StickerDecoder and StickerEncoder classes (**Listing 7** and **Listing 8**), the application uses the JSON API for the following:

- Reading from the socket stream to get objects
- Transforming objects into JSON string representation and writing them to the socket

You can create appropriate decoders depending on the format in which you want to read the data from the WebSocket. Similarly, you can create encoders depending on how you want to write the data to the socket.

As shown in **Listing 7**, a decoder must implement any interface inside the parent Decoder interface. There are several types of decoders and encoders, and they vary in the parameters that the decode and encode methods take.

The StickerDecoder class in **Listing 7** uses the Decoder.TextStream interface, which allows you to read the data from the socket using a Reader. To convert a string into a JsonObject you use a JsonReader, which conveniently takes a Reader to be constructed.

Note: The JsonReader object should not be created; instead, use the JsonProvider class to create readers and writers.

Listing 8 shows the StickerEncoder class that writes the object to the WebSocket stream.

Conclusion

WebSocket technology, JSF's HTML5 pass-through parameters, and JSON object serialization are powerful additions to Java EE 7 that greatly simplify the development of applications that require constant communication between the browser and the Web server.

In this article, we've built a Web application to show how to combine HTML5 with the JSF 2.2–friendly HTML markup. We also wrote a WebSocket client and drag-and-drop functionality for the application with JavaScript. Lastly, we built a WebSocket server by using the new Java APIs for WebSocket and JSON processing. The example shown in this article barely touches on their potential.

LISTING 7 LISTING 8

```
package org.sticker.websocket;

import java.io.IOException;
import java.io.Reader;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.spi.JsonProvider;
import javax.websocket.DecodeException;
import javax.websocket.Decoder;

public class StickerDecoder implements
Decoder.TextStream<Sticker> {

 @Override
 public Sticker decode(Reader reader) throws DecodeException,
IOException {
  JsonProvider provider = JsonProvider.provider();
  JsonReader jsonReader = provider.createReader(reader);
  JsonObject jsonSticker = jsonReader.readObject();
  Sticker sticker = new Sticker();
  sticker.setX(jsonSticker.getInt("x"));
  sticker.setY(jsonSticker.getInt("y"));
  sticker.setImage(jsonSticker.getString("sticker"));
  return sticker;

 }
}
```

Download all listings in this issue as text

Acknowledgements. *The authors would like to thank Marco Velasco, a young Mexican artist, for the illustration of the sticker story application.* `</article>`

LEARN MORE

- JSF 2.2 JSR
- Java API for WebSocket JSR
- Java API for JSON Processing JSR
- Arun Gupta's video about Java EE 7 and the WebSocket API