Software architecture

Week 11 - Web sockets and JMS

Requirements

1. Netbeans 11
2. Java Development Kit 8
3. Payara Server
4. Web socket API

Asynchronous methods

Web sockets

Web socket provides a persistent connection between a client and a server. It's an alternative to the limitation of efficient communication between the server and the web browser. It works on the underlying TCP/IP connections and provides **bi-directional**, **full-duplex**, **low-latency** and **real-time** client/server communications.

The Expert Group that defined the Java API for websocket (JSR) 356 wanted to support patterns and techniques that are common to Java EE developers. As a consequence, JSR 356 leverages annotations and injection.

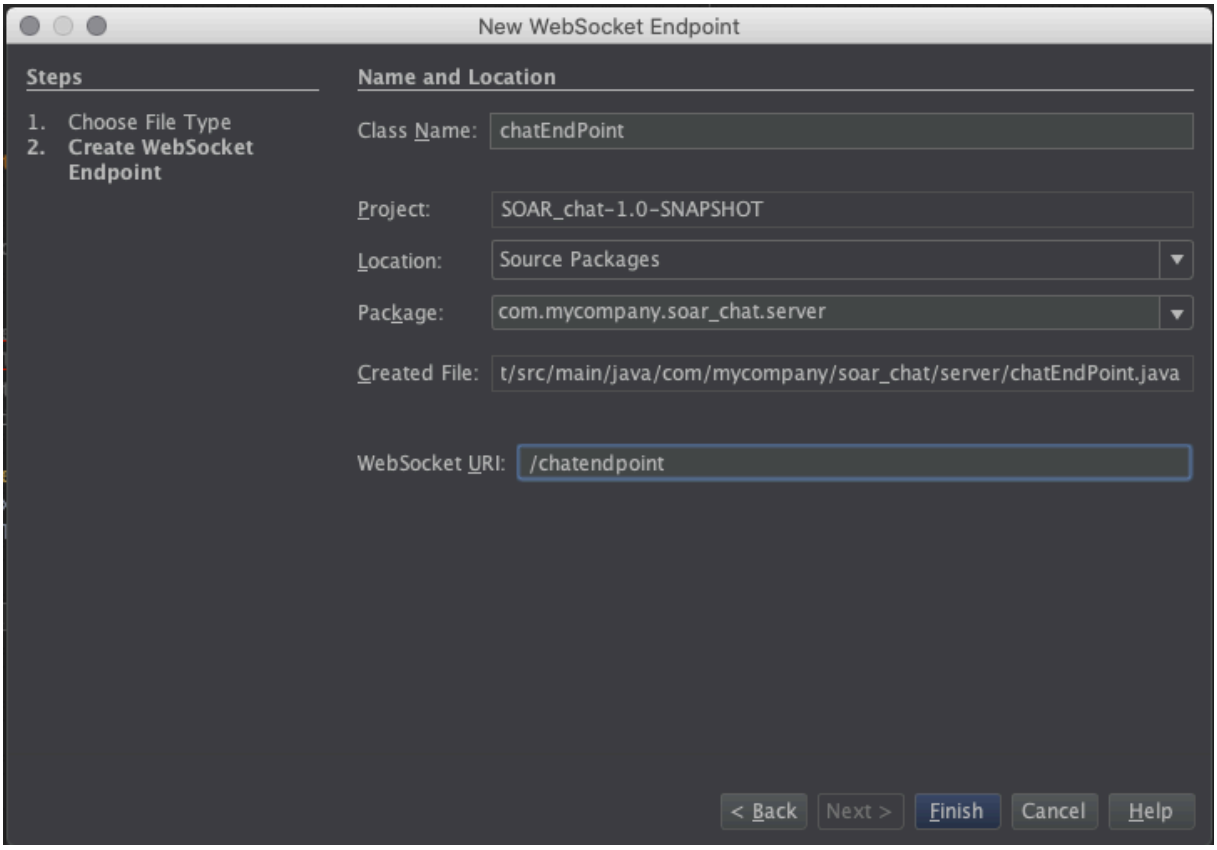
In this exercise session, we will create a real-time chat web application based on the Java API for Websockets. Through our messaging system, everybody will be able to log in and post messages in real-time.

To create our project using Netbeans, we will follow the steps below:

1. Open Netbeans
2. Create a New Project (File > New Project > Java with Maven > Web Application)
3. Let's call it "SoAr_sockets_week11", click on *Finish*. Our project is ready!
4. Separate our source code into packages. In order to have a better understanding of our code, we will separate each part of into packages. For this project, we will use two different packages:
    - Server (com.mycompany.server)
    - Client (com.mycompany.client)

Server

The server Endpoint will consist in a simple POJO (Plain Old Java Object) with the adequate annotations. To create the server Endpoint, we have to add a new class to our project. Luckily, Netbeans allows us to generate new Endpoints. To do it, we have to Right click on our project > New > Other > Web > WebSocket Endpoint



Now, we can add the additional methods and mention their role through annotations

The Web Socket Endpoint represents an object that can handle websocket conversations. Developers may extend this class in order to implement a programmatic WebSocket endpoint. The Endpoint class holds lifecycle methods that may be overridden to intercept websocket open, error and close events. https://docs.oracle.com/javaee/7/api/javax/websocket/Endpoint.html
The required annotations for our project are the following:

- @serverEndpoint: This class level annotation declares that the class it decorates is a WebSocket endpoint that will be deployed and made available in the URI-space of a WebSocket server. The annotation allows the developer to define the URL (or URI template) which this endpoint will be published, and other important properties of the endpoint to the websocket runtime, such as the encoders it uses to send messages.

- @OnOpen: is used to annotate a method which will be called after WebSocket connection in opened. The method linked to this annotation takes two parameters:

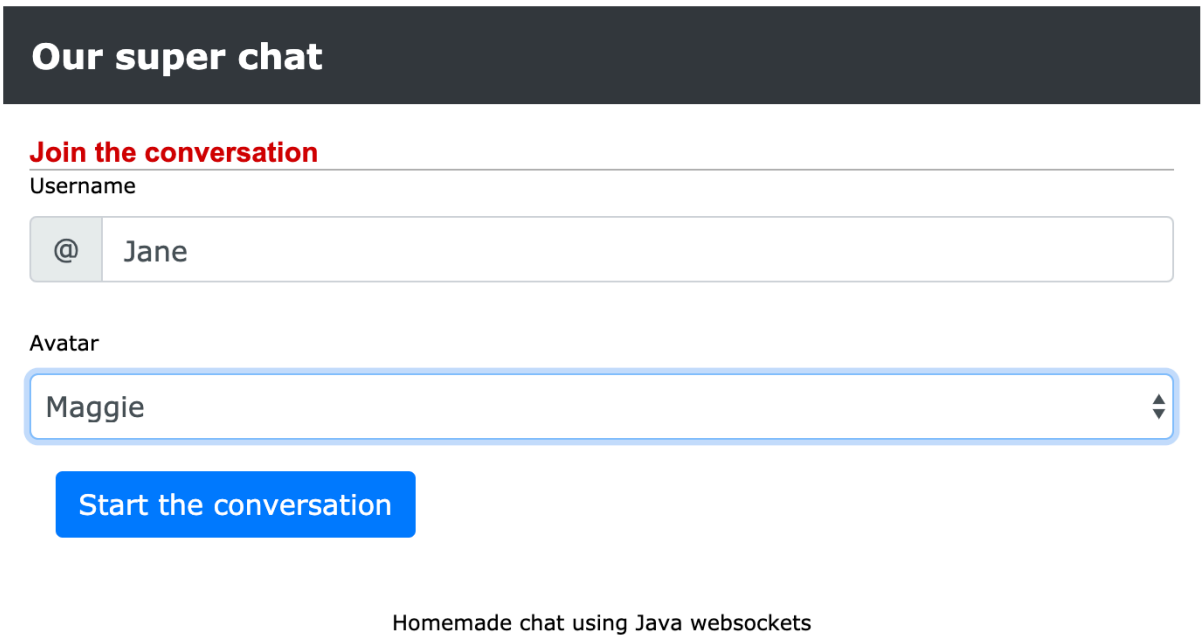
    - Session: the session that has just been activated

    - EndpointConfig(optional): the configuration used to configure the endpoint

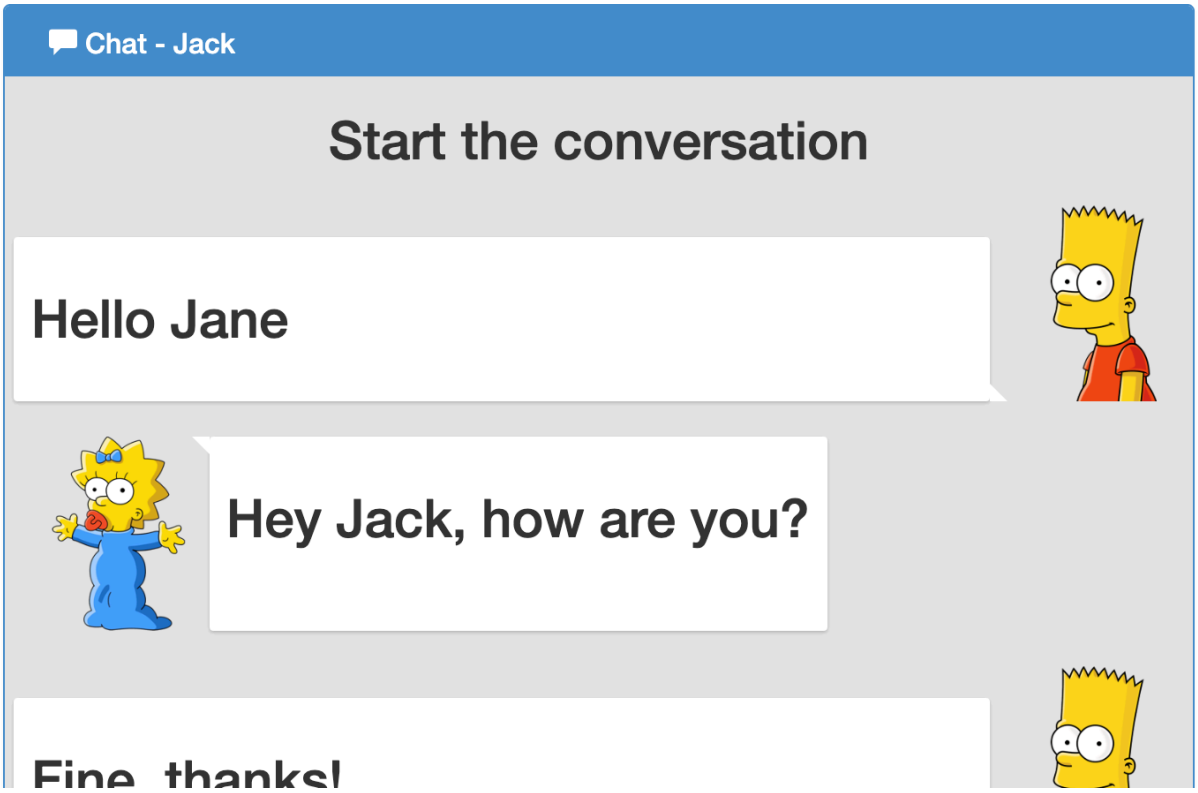
- @OnClose: this method is called immediately prior to the session with the remote peer being closed. It is called whether the session is being closed because the remote peer initiated a close and sent a close frame, or whether the local websocket container or this endpoint requests to close the session. The method linked to this annotation takes two parameters:
    - Session: the session about to be closed
    - closeReason: the reason the session was closed.
- @OnMessage: this method-level annotation can be used for a Java method to receive incoming WebSocket messages.
- @OnError: a method with @OnError is invoked when there is a problem with the communication

Client

To communicate with the WebSocket server, the client has to initiate the WebSocket connection by sending an HTTP request to a server. This is called the *Handshake phase*. Except for this phase, the WebSocket protocol is totally independent from *HTTP*. For this exercise session, we will use Javascript to create a WebSocket client. But first of all, we need to create the User Interface for our chat. To make it simple, we will only create two JSP pages and a servlet. The JSP pages will consist of a homepage (registration form) and the chatroom. The homepage should have a form containing 2 form fields: Name and Avatar.



Home page



Chatroom
To create the JSP pages and the servlet, you can refer to the instructions of week 9.
You can find on this link a sample web page for our chatroom.

Now, we can focus on the WebSocket client (written in Javascript). We have to create a new Javascript file and link it to our JSP by using the following code:

```
<script src="${pageContext.request.contextPath}/js/main.js"></script>
```

The Javascript file will be used to :

- Map the WebSocket server endpoint to the URI defined, in our case, it should be `ws://localhost:8080/<project-name>/<server-EndPoint>`.
- Capture the event of sending a new message and transfer the message to the server
- Update the UI according to incoming messages.

```
//Connection the server
var webSocket = new WebSocket('ws://localhost:8080/<project-name>/<server-
EndPoint>');

// Triggers an action when the user clicks on the "Send button"
    $("#btn-chat").click(function(){
        var message = $("#message_input").val();
        //Encode the message in a JSON format
        message = JSON.stringify({
            name: name,
            message: message,
            avatar: avatar
        });
        webSocket.send(message);
    });
```

```
//Triggered when there is a new message from the server
webSocket.onmessage = function (event) {
    onMessage(event);
};

function onMessage(event) {
  var incoming_message = JSON.parse(event.data);
  updateUI(incoming_message);
}

//Method used to update the User Interface
function updateUI(message){
    //Decode the JSON
    name = message.name;
    message = message.message;
    alert("New message from"+name+": "+message);

}
```

JMS

1. Open **NetBeans IDE**
2. Start the **Payara Server**
3. Once Payara Server started, right-click on Payara Server and click on **View Domain Admin Console**
4. You'll see the **Payara Server Console** on your browser
5. Click on **JMS Resource >> Connection Factories >> New**
6. Write **jms/MyJMSExerciseConnectionFactory** to JNDI Name and select **javax.jms.ConnectionFactory** from Resource Type list. Then, just click on OK
7. Click on **JMS Resource >> Destination Resources >> New**
8. Select **javax.jms.Queue** from Resource Type, write **jms/MyJMSExerciseQueue** to JNDI Name and **myQueue** to Pyhsical Destination Name. Then, just click on OK
9. Create a new **Enterprise Application Client**

As you implement a JMS sender and a JMS receiver classes, you will need to import packages. Don't forget to check correct import statements! Here are the import statements you will need:

For JMSSender.java class

- javax.annotation.Resource
- javax.jms.ConnectionFactory
- javax.jms.JMSProducer
- javax.jms.JMSContext
- javax.jms.Queue

For JMSReceiver.java class

- javax.annotation.Resource
- javax.jms.ConnectionFactory
- javax.jms.JMSConsumer
- javax.jms.JMSContext
- javax.jms.Queue

Implementing JMS Sender (JMSSender.java)

1. Add two private class variables called **connectionFactory** type of **ConnectionFactory** and *_queue* type of **Queue**
2. Add @Resource annotation for the class variable **connectionFactory**, define a mappedName and it's value should be the name of the connection factory (**jms/MyJMSExerciseConnectionFactory**)
3. Add @Resource annotation for the class variable **queue**, define a mappedName and it's value should be the name of the queue (**jms/MyJMSExerciseQueue**)
4. Write a main method
5. Create a local variable called **jmsContext** (type of **JMSContext**) and initialize it by calling **createContext()** method of **connectionFactory** instance
6. Create a local variable called **jmsProducer** (type of **JMSProducer**) and initialize it by calling **createProducer()** method of **jmsContext** instance
7. Create a local String variable called **message** and initialize it as you wish (i.e. "Hello JMS!")
8. Tell the user that you're sending a message (Hint: print it!)
9. Call the **send(...)** method **jmsProducer** using **queue** and **message**
10. Tell the user that the message is sent

Implementing JMS Receiver (JMSReceiver.java)

1. Add two private class variables called **connectionFactory** type of **ConnectionFactory** and *_queue* type of **Queue**
2. Add @Resource annotation for the class variable **connectionFactory**, define a mappedName and it's value should be the name of the connection factory (**jms/MyJMSExerciseConnectionFactory**)
3. Add @Resource annotation for the class variable **queue**, define a mappedName and it's value should be the name of the queue (**jms/MyJMSExerciseQueue**)
4. Write a main method

5. Create a local variable called **jmsContext** (type of **JMSContext**) and initialize it by calling **createContext()** method of **connectionFactory** instance
6. Create a local variable called **jmsConsumer** (type of **JMSConsumer**) and initialize it by calling **createConsumer** method of **jmsContext** instance using **queue**
7. Tell the user that you're receiving message from JMS
8. Create a local String variable called **message** and initialize it by calling **receiveBody(String.class)** method of **jmsConsumer** instance
9. Tell the user that you received the message and show the message

Clean and Build your project, then Run JMSSender and JMSReceiver respectively!