WEBSOCKETS CON JAVA

Introducción y guía de nuestra primera aplicación con WebSockets

Software Factory

2016-1

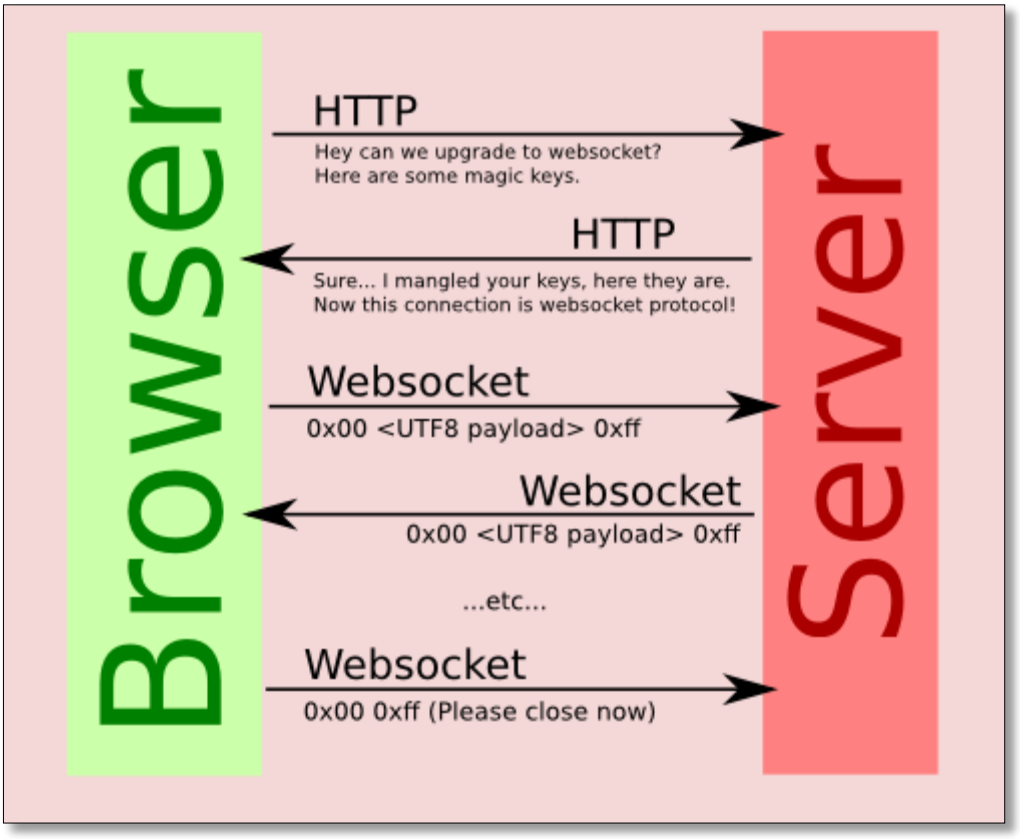
Roy Alejo Taza Rojas

U201213734@upc.edu.pe

Contenido

INTRODUCCIÓN

WebSockets es una especificación que va sonando con más fuerza de la mano de HTML5. Esta especificación se basa en un canal de comunicación bidireccional entre un cliente y un servidor, mediante el cual pueden enviarse mensajes de un sentido a otro en cualquier momento sin necesidad de que haya una petición de por medio.



Esta especificación está especialmente dirigida a las denominadas “aplicaciones en tiempo real”, que son aquellas en las que el cliente puede estar informado de todo aquello que sucede en el sistema desde el mismo momento en que se produce un cambio. Ejemplos de este tipo de aplicaciones pueden ser: juegos multijugador, aplicaciones de monitorización, chats, herramientas de trabajo colaborativo, etc…

En el caso de una herramienta de trabajo colaborativo, cuando un equipo de trabajo está delante de la pantalla (cada uno en su ordenador) y un miembro finaliza una tarea y actualiza su estado a “finalizada”, la aplicación notifica inmediatamente al resto de usuarios (o a uno, o a varios) de que esa tarea está cerrada e inmediatamente ven un cambio en el estado de esa tarea.

En un juego multijugador podríamos visualizar el movimiento de otro jugador. En un chat veríamos como nos llega un mensaje de otro usuario en el momento en que nos lo envía. En una herramienta de monitorización, la temperatura que marca el sensor de algún componente de un sistema, etc… La idea es que tenemos la información desde el mismo instante en que se genera.

En esta guía intentaremos explicar qué es la especificación WebSocket, en qué casos puede ser interesante su uso y cómo implementarla desde el lado del cliente y servidor.

¿Qué son los WebSockets?

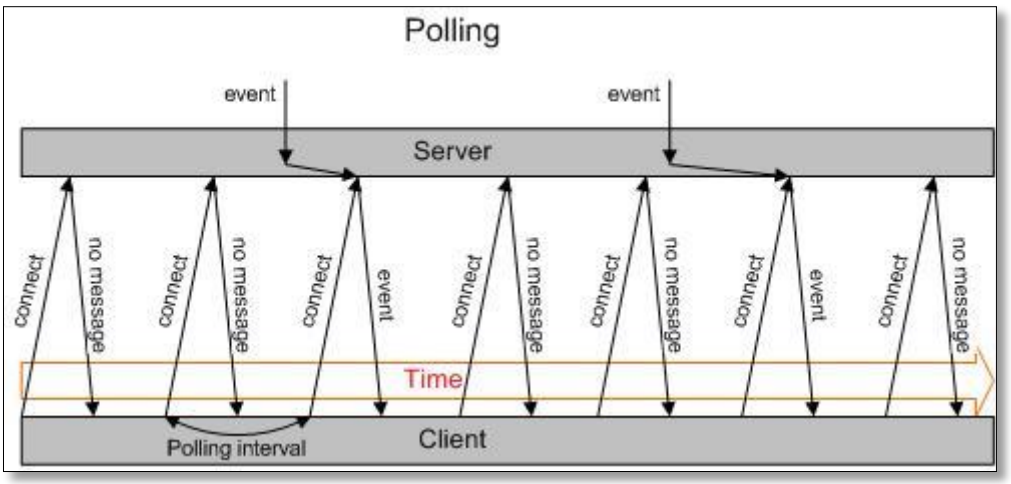
La especificación

La especificación WebSocket (llevada a cabo por la IETF) provee un canal de comunicación bi-direccional entre el navegador (cliente) y el servidor, enviando y recibiendo mensajes de manera simultánea. El cliente puede enviar datos al servidor por este canal pero, lo más interesante, es que el servidor puede enviar datos al cliente sin necesidad de que éste realice una petición para solicitar datos. Se establece una única conexión entre cliente y servidor que permite que cualquiera de los dos actores pueda enviar mensajes al otro en cualquier momento.

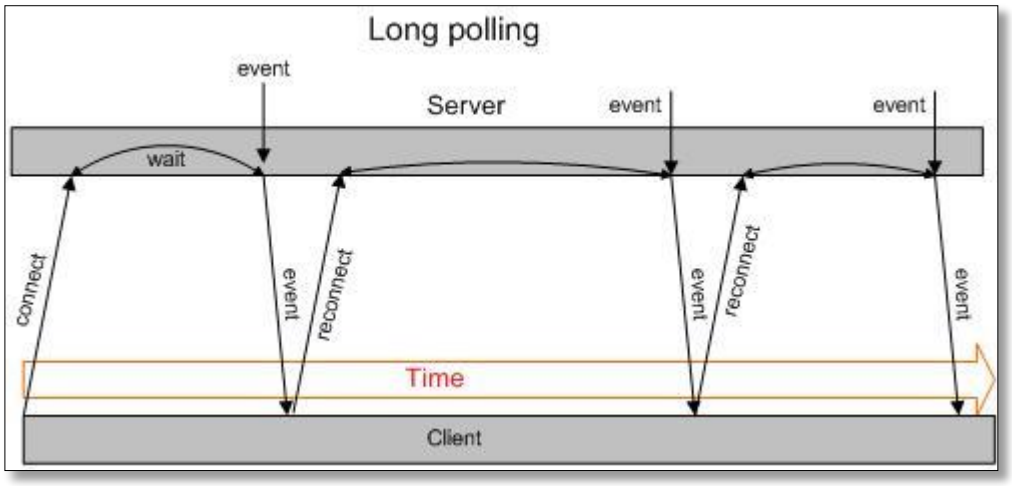
¿Y esto para qué vale? Pues como hemos comentado para aplicaciones que requieren constantes actualizaciones en el front-end debido a la interacción de terceras personas (o sistemas). Son las denominadas "aplicaciones en tiempo real".

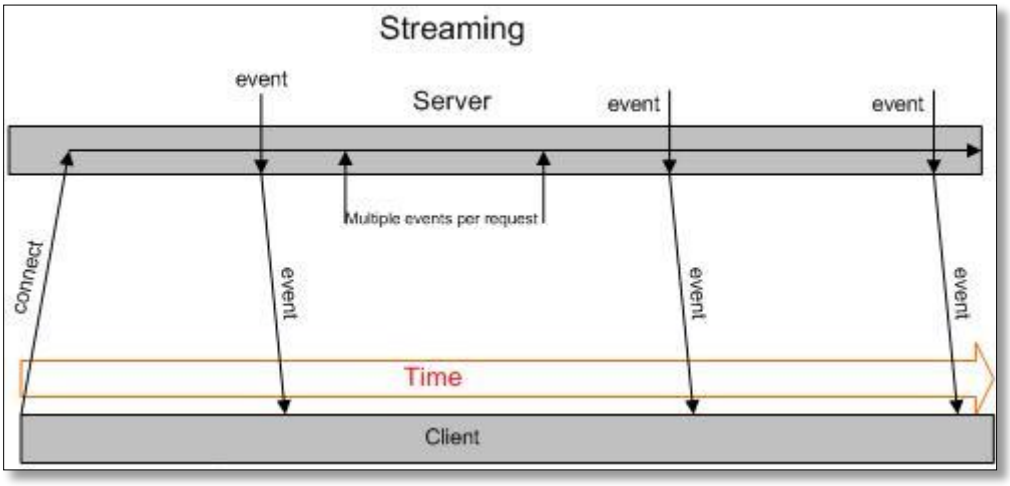
Antes de la aparición de WebSockets se utilizaban dos técnicas para implementar este comportamiento:

- **AJAX polling:** Consiste en realizar constantemente peticiones al servidor preguntándole si se ha producido algún evento que requiera una actualización en la vista. Por ejemplo: en un chat, desde la pantalla donde llegan los posibles mensajes que recibe el usuario por parte de otros usuarios, se estarían enviando peticiones HTTP al servidor (cada 2 segundos, 5, 10, o lo que sea) preguntándole ¿tengo mensajes?, ¿tengo mensajes?, ¿tengo mensajes?, etc… Esta es una solución bastante ineficiente debido a que se genera una gran cantidad de tráfico con el servidor sobre todo en aplicaciones con un elevado número de usuarios e intervalo entre peticiones pequeño. Además, pensemos que la mayoría de las veces el servidor responderá lo mismo: "no, no hay cambios".



- **Comet o "long polling":** Es una técnica muy parecida a la que utilizan los WebSockets. Consiste en realizar una única petición al servidor de forma que éste responde diciendo que va a devolver la respuesta en "trozos" (streaming). La petición queda abierta hasta que el servidor responda con todas las porciones de respuesta que solicita el cliente, que no son otra cosa que eventos que se producen en el servidor notificando un cambio de estado en el cliente. Como digo, es parecido a lo que hacen los WebSockets, sin embargo su implementación es bastante más compleja.





API Javascript

El constructor: Recibe un parámetro obligatorio y otro opcional. El primero es la URL del servidor con el que estableceremos la conexión. El segundo (opcional) es una lista de subprotocolos de conexión. El WebSocket se construiría así.

```
var ws = new WebSocket(URL /* requerido */, subprotocolos /* opcional */);
```

El manejador de eventos **onopen**: manejador que es invocado cuando se abre una conexión con el servidor.

```
ws.onopen = function () {
                // hacemos lo que sea al abrir la conexión
        };
```

El manejador de eventos **onclose**: manejador que es invocado cuando se cierra conexión con el servidor.

```
ws.onclose = function () {
                // hacemos lo que sea al cerrar la conexión
        };
```

El manejador de eventos **onmessage**: manejador que es invocado cuando llega un mensaje desde el servidor.

```
ws.onmessage = function (event) {
                var message = event.data;
                //      hacemos lo que sea con el mensaje
        };
```

El manejador de eventos **onerror**: manejador que es invocado cuando se produce un error.

```
ws.onmessage = function (event) {
                //      hacemos lo que sea con el error
        };
```

El método **send**: envía un mensaje de texto o datos binarios al servidor.

```
ws.send('Mensaje');
```

El método close: cierra la conexión con el servidor.

```
ws.close();
```

Api Java EE 7

El API Java para WebSocket (JSR-356) simplifica la integración de WebSocket en aplicaciones Java EE 7.

Estas son algunas de las características de la API Java para WebSocket:

- La programación de la anotación en la que los desarrolladores utilizan POJOs para interactuar con los eventos del ciclo de vida WebSocket
- Interfaz de programación en la que los desarrolladores implementan interfaces y métodos para interactuar con los eventos del ciclo de vida WebSocket
- La integración con otras tecnologías Java EE (se puede inyectar Enterprise JavaBeans mediante el uso de componentes como Contextos y la inyección de dependencias).

Propósito de la Guía

En esta guía se muestra cómo crear una aplicación que utiliza la API de WebSocket para la comunicación en tiempo real entre un cliente y un servidor. Se aprenderá:

- Crear una aplicación de la plataforma Java EE 7, que utiliza la API WebSocket.
- Usar los eventos del ciclo de vida WebSocket OnOpen y onMessage para llevar a cabo diferentes acciones en la aplicación Java EE.
- Definir un punto final WebSocket del lado del cliente mediante el uso de JavaScript.
- Operar en Plain Old Java Objects (POJO), en tiempo real, con acciones invocadas desde un cliente de navegador web.

Escenario

En este tutorial, se crea una Java WebSocket Home, una aplicación web de control del hogar inteligente basado en Java EE 7. Java WebSocket Home tiene una interfaz de usuario para conectar y controlar dispositivos ficticios desde un navegador web para una aplicación Java. Esta aplicación proporciona actualizaciones en tiempo real a todos los clientes que están conectados al servidor Java WebSocket Home.

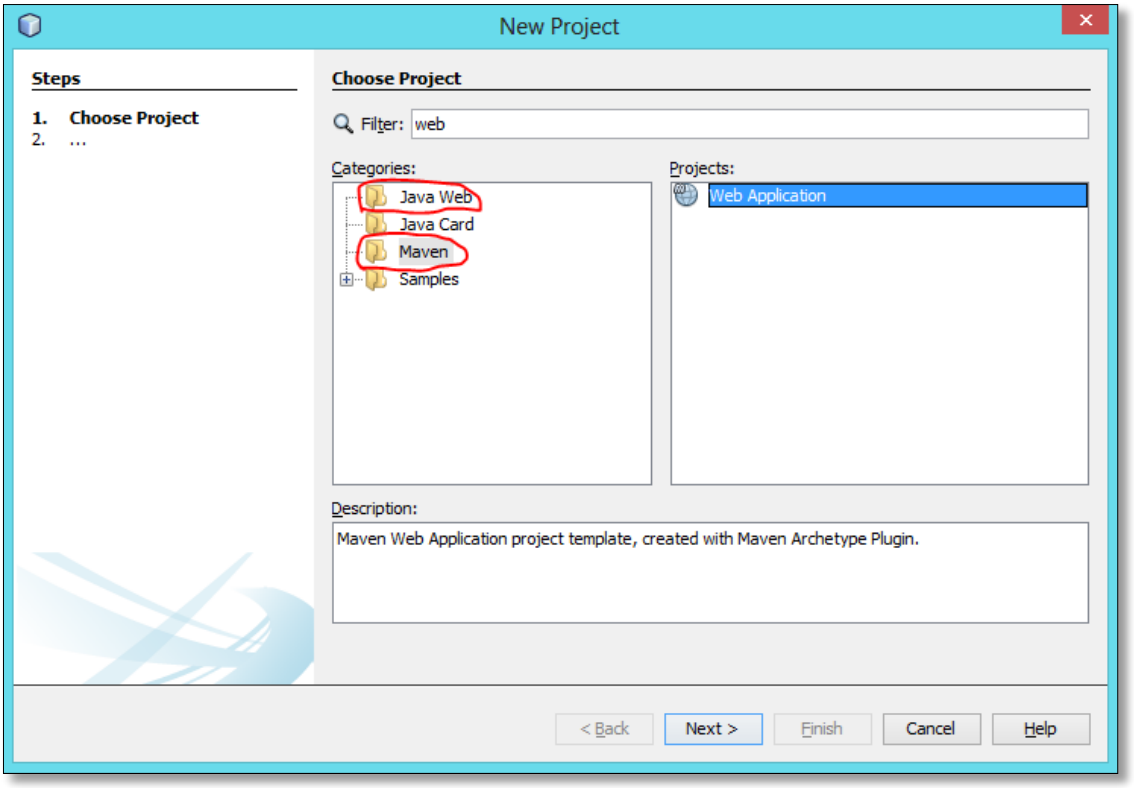
Software Utilizado

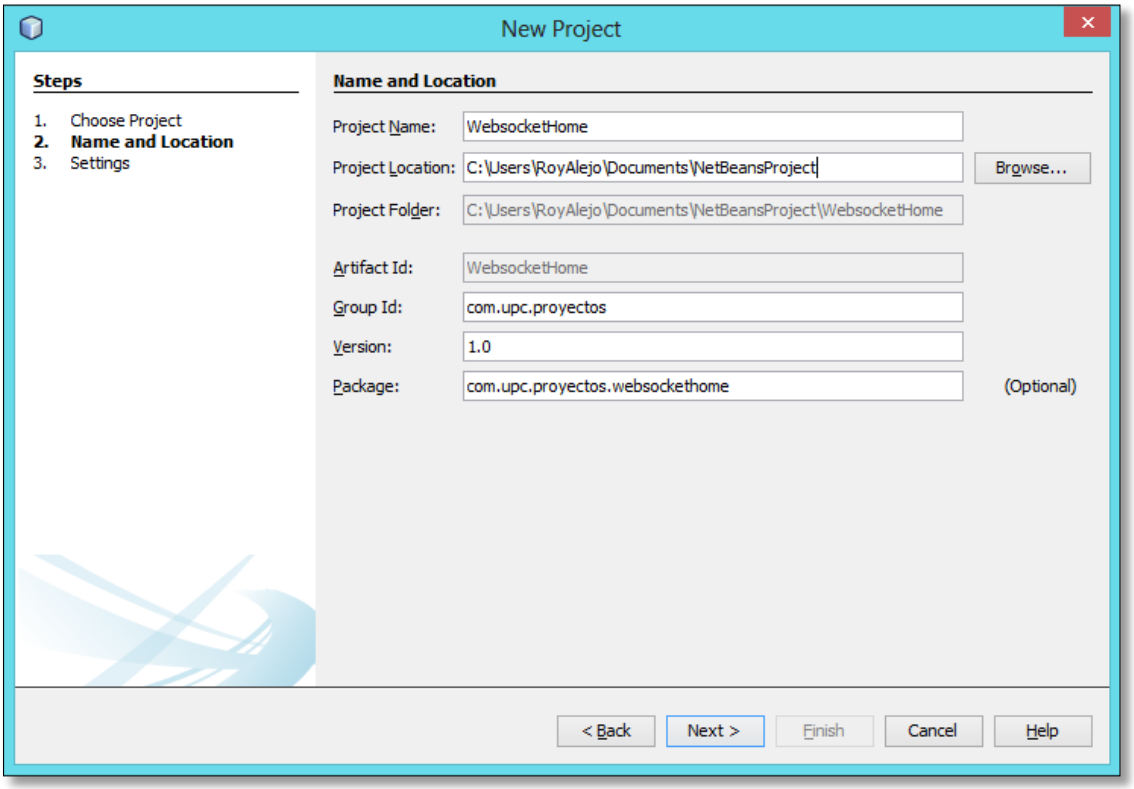
- JDK 7/JDK 8
- Netbeans 7.3.1 / Netbeans 8.0.2
- Oracle GlassFish Server 4.0

IMPLEMENTACIÓN

Creación del proyecto

Creamos un proyecto Web, ya sea Java Web o Maven. En este caso será Maven. Le ponemos de nombre **WebsocketHome.**

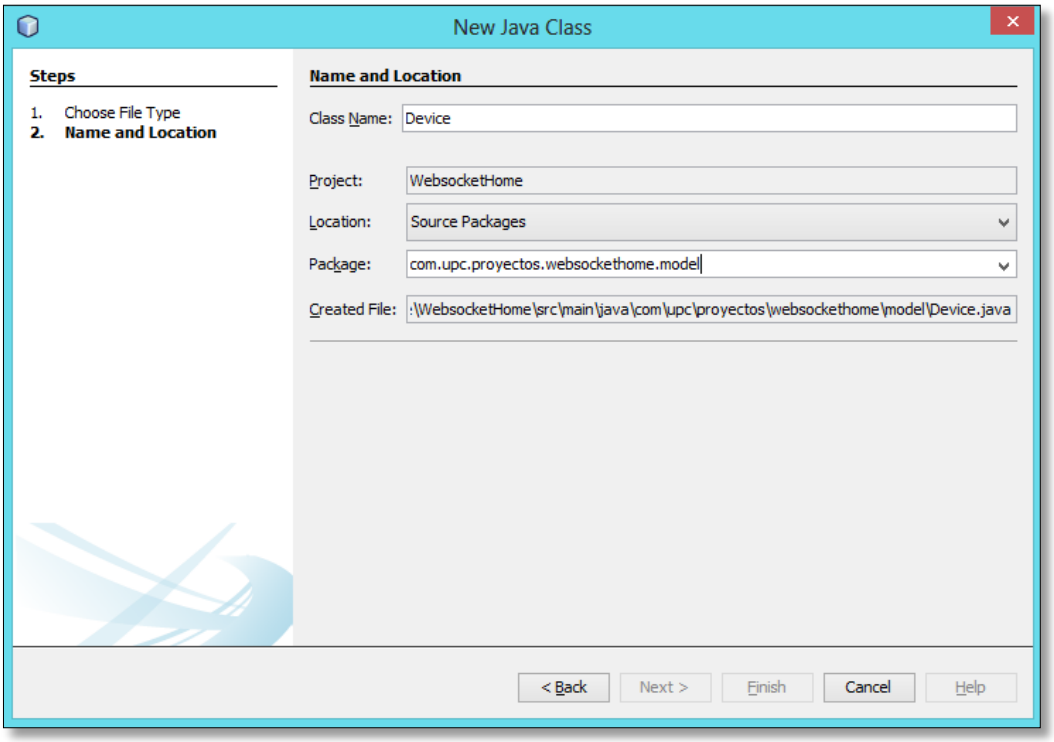




Creando el modelo Device

Se creará la clase que contenga los atributos de un Device (dispositivo).

Crearemos una clase **Device** y lo pondremos en el paquete model del proyecto.



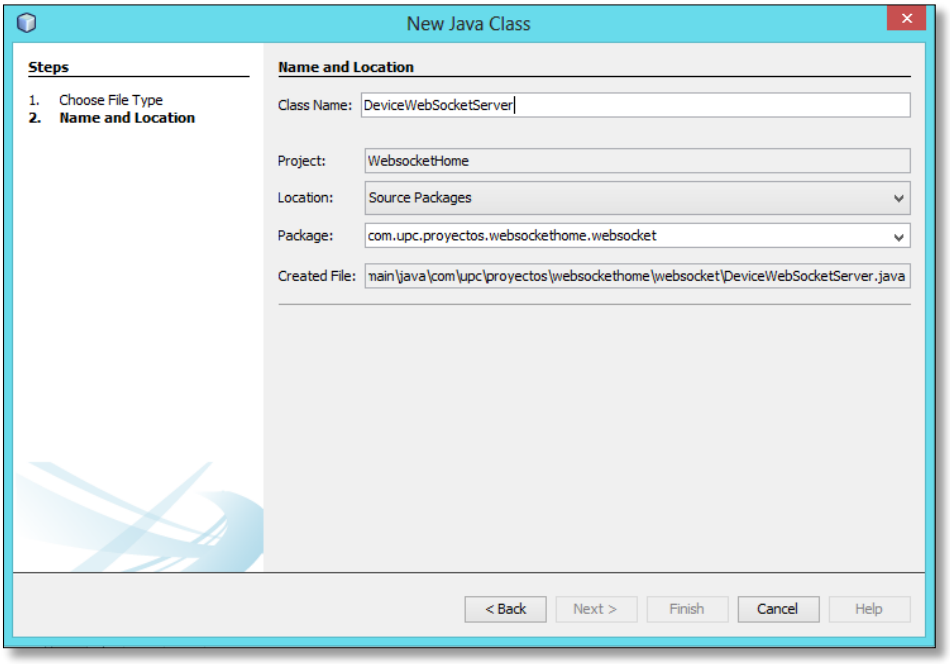
Añadimos el siguiente código a la clase Device que acabamos de crear para definir el constructor, los getter y setter:

```
public class Device {

   private int id;
   private String name;
   private String status;
   private String type;
   private String description;

   public Device() {
   }

   public int getId() {
      return id;
   }

   public String getName() {
      return name;
   }

   public String getStatus() {
      return status;
   }
```

```
    public String getType() {
        return type;
    }

    public String getDescription() {
        return description;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public void setType(String type) {
        this.type = type;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Creando el WebSocket Server Endpoint

Creamos una clase **DeviceWebSocketServer** y lo pondremos en el paquete websocket de nuestro proyecto.



Definimos la ruta de endpoint de nuestro WebSocket server añadiendo el siguiente código:

```
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/actions")
public class DeviceWebSocketServer {

}
```

Definimos las anotaciones del ciclo de vida de un WebSocket añadiendo los métodos e importaciones a la clase:

```
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/actions")
public class DeviceWebSocketServer {

   @OnOpen
      public void open(Session session) {
   }

   @OnClose
      public void close(Session session) {
   }

   @OnError
      public void onError(Throwable error) {
   }

   @OnMessage
      public void handleMessage(String message, Session session) {
   }
}
```

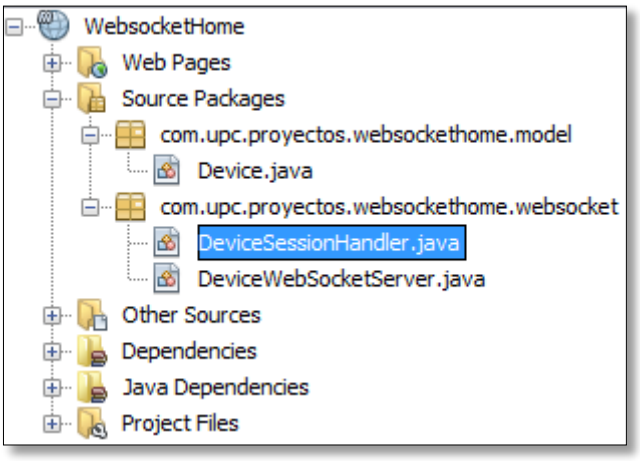
Con esto, el ciclo de vida del WebSocket está mapeado en los métodos Java de nuestra clase DeviceWebSocketServer.

Especificamos que la clase es ámbito de la aplicación añadiendo la etiqueta ApplicationScoped:

```
...
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
@ServerEndpoint("/actions")
public class DeviceWebSocketServer {
..
```

Creando Session Handler

Crearemos el gestor de sesiones. Añadiremos una clase java **DeviceSessionHandler** al paquete websocket de nuestro proyecto.



Especificamos que la clase es application-scoped y declaramos los HashSet para guardar la lista de Devices añadidos a la aplicación y las sesiones activas en la aplicación:

```
import javax.enterprise.context.ApplicationScoped;
import java.util.HashSet;
import java.util.Set;
import javax.websocket.Session;
import org.example.model.Device;

@ApplicationScoped
public class DeviceSessionHandler {
   private final Set sessions = new HashSet<>();
   private final Set devices = new HashSet<>();
}
```

Nota: cada cliente conectado a la aplicación tiene su propia sesión.

Definimos los siguientes métodos para añadir y quitar sesiones del servidor:

```
...

@ApplicationScoped
public class DeviceSessionHandler {

   ...

   public void addSession(Session session) {
      sessions.add(session);
   }

   public void removeSession(Session session) {
      sessions.remove(session);
   }
}
```

Añadimos los métodos que operaran el objeto Device:

```
...
public class DeviceSessionHandler {

    ...

    public List getDevices() {
        return new ArrayList<>(devices);
    }

    public void addDevice(Device device) {
    }

    public void removeDevice(int id) {
    }

    public void toggleDevice(int id) {
    }

    private Device getDeviceById(int id) {
        return null;
    }

    private JsonObject createAddMessage(Device device) {
        return null;
    }

    private void sendToAllConnectedSessions(JsonObject message) {
    }

    private void sendToSession(Session session, JsonObject message) {
    }
}
```

Construyendo la interfaz del usuario

Abrimos el archivo index.html que viene por defecto al crear el proyecto y reemplazamos su contenido por lo siguiente:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script src="websocket.js"></script>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>

        <div id="wrapper">
            <h1>Java Websocket Home</h1>
            <p>Welcome to the Java WebSocket Home. Click the Add a device button to start adding devices.</p>
```

```
        <br />
        <div id="addDevice">
          <div class="button"> <a href="#" OnClick="showForm()">Add a device</a> </div>
          <form id="addDeviceForm">
            <h3>Add a new device</h3>
            <span>Name: <input type="text" name="device_name"
id="device_name"></span>
            <span>Type:
              <select id="device_type">
                <option name="type" value="Appliance">Appliance</option>
                <option name="type" value="Electronics">Electronics</option>
                <option name="type" value="Lights">Lights</option>
                <option name="type" value="Other">Other</option>
              </select></span>

            <span>Description:<br />
              <textarea name="description" id="device_description" rows="2"
cols="50"></textarea>
            </span>

            <input type="button" class="button" value="Add" onclick=formSubmit()>
            <input type="reset" class="button" value="Cancel" onclick=hideForm()>
          </form>
        </div>
        <br />
        <h3>Currently connected devices:</h3>
        <div id="content">
        </div>
      </div>

  </body>
</html>
```

Creamos un archivo CSS en la raíz del proyecto, lo nombramos **style.css** y contendrá lo siguiente:

```
body {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 80%;
  background-color: #1f1f1f;
}

#wrapper {
  width: 960px;
  margin: auto;
  text-align: left;
  color: #d9d9d9;
}

p {
  text-align: left;
}

.button {
```

```
    display: inline;
    color: #fff;
    background-color: #f2791d;
    padding: 8px;
    margin: auto;
    border-radius: 8px;
    -moz-border-radius: 8px;
    -webkit-border-radius: 8px;
    box-shadow: none;
    border: none;
}

.button:hover {
    background-color: #ffb15e;
}
.button a, a:visited, a:hover, a:active {
    color: #fff;
    text-decoration: none;
}

#addDevice {
    text-align: center;
    width: 960px;
    margin: auto;
    margin-bottom: 10px;
}

#addDeviceForm {
    text-align: left;
    width: 400px;
    margin: auto;
    padding: 10px;
}

#addDeviceForm span {
    display: block;
}

#content {
    margin: auto;
    width: 960px;
}

.device {
    width: 180px;
    height: 110px;
    margin: 10px;
    padding: 16px;
    color: #fff;
    vertical-align: top;
    border-radius: 8px;
    -moz-border-radius: 8px;
```

```
    -webkit-border-radius: 8px;
    display: inline-block;
}

.device.off {
    background-color: #c8cccf;
}

.device span {
    display: block;
}

.deviceName {
    text-align: center;
    font-weight: bold;
    margin-bottom: 12px;
}

.removeDevice {
    margin-top: 12px;
    text-align: center;
}

.device.Appliance {
    background-color: #5eb85e;
}

.device.Appliance a:hover {
    color: #a1ed82;
}

.device.Electronics {
    background-color: #0f90d1;
}

.device.Electronics a:hover {
    color: #4badd1;
}

.device.Lights {
    background-color: #c2a00c;
}

.device.Lights a:hover {
    color: #fad232;
}

.device.Other {
    background-color: #db524d;
}

.device.Other a:hover {
```

```
    color: #ff907d;
}

.device a {
    text-decoration: none;
}

.device a:visited, a:active, a:hover {
    color: #fff;
}

.device a:hover {
    text-decoration: underline;
}
```

Creando el WebSocket Client Endpoint

Creamos el endpoint del cliente WebSocket utilizando JavaScript. Añadimos un archivo JavaScript **websocket.js** a la raíz del proyecto y ponemos lo siguiente:

```
window.onload = init;
var socket = new WebSocket("ws://localhost:8080/WebsocketHome/actions");
socket.onmessage = onMessage;

function onMessage(event) {
    var device = JSON.parse(event.data);
    if (device.action === "add") {
        printDeviceElement(device);
    }
    if (device.action === "remove") {
        document.getElementById(device.id).remove();
        //device.parentNode.removeChild(device);
    }
    if (device.action === "toggle") {
        var node = document.getElementById(device.id);
        var statusText = node.children[2];
        if (device.status === "On") {
            statusText.innerHTML = "Status: " + device.status + " (<a href=\"#\"
OnClick=toggleDevice(" + device.id + ")>Turn off</a>)";
        } else if (device.status === "Off") {
            statusText.innerHTML = "Status: " + device.status + " (<a href=\"#\"
OnClick=toggleDevice(" + device.id + ")>Turn on</a>)";
        }
    }
}

function addDevice(name, type, description) {
    var DeviceAction = {
        action: "add",
        name: name,
        type: type,
        description: description
    };
```

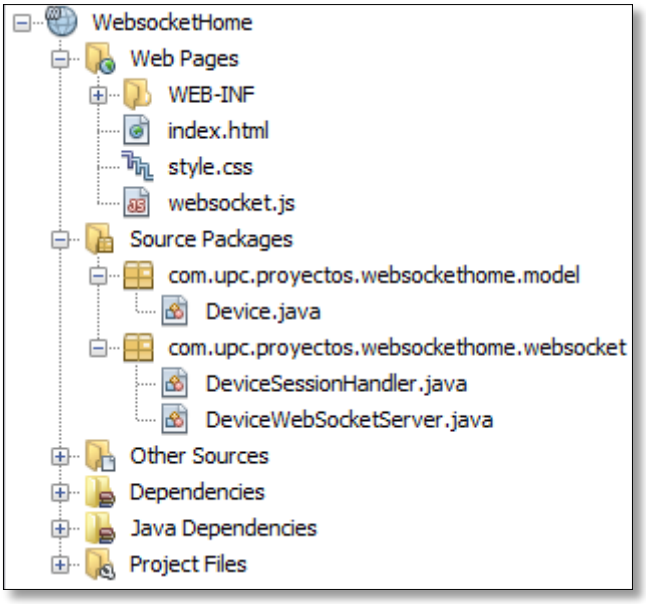
```
    socket.send(JSON.stringify(DeviceAction));
}

function removeDevice(element) {
    var id = element;
    var DeviceAction = {
        action: "remove",
        id: id
    };
    socket.send(JSON.stringify(DeviceAction));
}

function toggleDevice(element) {
    var id = element;
    var DeviceAction = {
        action: "toggle",
        id: id
    };
    socket.send(JSON.stringify(DeviceAction));
}

function printDeviceElement(device) {
    var content = document.getElementById("content");

    var deviceDiv = document.createElement("div");
    deviceDiv.setAttribute("id", device.id);
    deviceDiv.setAttribute("class", "device " + device.type);
    content.appendChild(deviceDiv);

    var deviceName = document.createElement("span");
    deviceName.setAttribute("class", "deviceName");
    deviceName.innerHTML = device.name;
    deviceDiv.appendChild(deviceName);

    var deviceType = document.createElement("span");
    deviceType.innerHTML = "<b>Type:</b> " + device.type;
    deviceDiv.appendChild(deviceType);

    var deviceStatus = document.createElement("span");
    if (device.status === "On") {
        deviceStatus.innerHTML = "<b>Status:</b> " + device.status + " (<a href=\"#\"
OnClick=toggleDevice(" + device.id + ")>Turn off</a>)";
    } else if (device.status === "Off") {
        deviceStatus.innerHTML = "<b>Status:</b> " + device.status + " (<a href=\"#\"
OnClick=toggleDevice(" + device.id + ")>Turn on</a>)";
        //deviceDiv.setAttribute("class", "device off");
    }
    deviceDiv.appendChild(deviceStatus);

    var deviceDescription = document.createElement("span");
    deviceDescription.innerHTML = "<b>Comments:</b> " + device.description;
    deviceDiv.appendChild(deviceDescription);
```

```
   var removeDevice = document.createElement("span");
   removeDevice.setAttribute("class", "removeDevice");
   removeDevice.innerHTML = "<a href=\"#\" OnClick=removeDevice(" + device.id +
")>Remove device</a>";
   deviceDiv.appendChild(removeDevice);
}

function showForm() {
   document.getElementById("addDeviceForm").style.display = '';
}

function hideForm() {
   document.getElementById("addDeviceForm").style.display = "none";
}

function formSubmit() {
   var form = document.getElementById("addDeviceForm");
   var name = form.elements["device_name"].value;
   var type = form.elements["device_type"].value;
   var description = form.elements["device_description"].value;
   hideForm();
   document.getElementById("addDeviceForm").reset();
   addDevice(name, type, description);
}

function init() {
   hideForm();
}
```

El archivo realiza las siguientes acciones:

- Asigna el punto final de servidor WebSocket a la URI definida en " Creando el WebSocket Server Endpoint".
- Captura los eventos de JavaScript para añadir, eliminar y cambiar el estado de un dispositivo y envía estos eventos al servidor WebSocket. Estos métodos son AddDevice(), removeDevice(), y toggleDevice(). Las acciones son enviados en mensajes JSON al servidor WebSocket.
- Define un método de devolución de llamada para el evento WebSocket onmessage. El evento onmessage captura los eventos enviados desde el servidor WebSocket (en JSON) y procesa esas acciones. En esta aplicación, estas acciones generalmente provocan los cambios en la interfaz de usuario cliente.
- Alterna la visibilidad de un formulario HTML para añadir un nuevo dispositivo.

La estructura del proyecto quedará de la siguiente manera:



Procesando los eventos WebSocket en el servidor

Abrimos la clase DeviceWebSocketServer e inyectamos un objeto del tipo DeviceSessionHandler para procesar los eventos del ciclo de vida del WebSocket en cada sesiós:

```
...
import javax.websocket.server.ServerEndpoint;
import javax.inject.Inject;

@ApplicationScoped
@ServerEndpoint("/actions")
public class DeviceWebSocketServer {

   @Inject
   private DeviceSessionHandler sessionHandler;

   @OnOpen
   public void open(Session session) {
   }

   ...
}
```

Procesamos el evento OnMessage del WebSocket añadiendo el siguiente código al método open:

```
@OnOpen
   public void open(Session session) {
      sessionHandler.addSession(session);
   }
```

Procemos el evento OnOpen WebSocket:

```
...
import java.io.StringReader;
```

```
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
import org.example.model.Device;

...

   @OnMessage
   public void handleMessage(String message, Session session) {

      try (JsonReader reader = Json.createReader(new StringReader(message))) {
         JsonObject jsonMessage = reader.readObject();

         if ("add".equals(jsonMessage.getString("action"))) {
            Device device = new Device();
            device.setName(jsonMessage.getString("name"));
            device.setDescription(jsonMessage.getString("description"));
            device.setType(jsonMessage.getString("type"));
            device.setStatus("Off");
            sessionHandler.addDevice(device);
         }

         if ("remove".equals(jsonMessage.getString("action"))) {
            int id = (int) jsonMessage.getInt("id");
            sessionHandler.removeDevice(id);
         }

         if ("toggle".equals(jsonMessage.getString("action"))) {
            int id = (int) jsonMessage.getInt("id");
            sessionHandler.toggleDevice(id);
         }
      }
   }
```

Implementamos los eventos OnClose y OnError:

```
...
import java.util.logging.Level;
import java.util.logging.Logger;

...

   @OnClose
   public void close(Session session) {
      sessionHandler.removeSession(session);
   }

   @OnError
   public void onError(Throwable error) {
      Logger.getLogger(DeviceWebSocketServer.class.getName()).log(Level.SEVERE,        null,
error);
   }
```

Implementando las acciones WebSocket en el Session Handler

Abrimos la clase DeviceSessionHandler y definimos variables para guardar los devices encontrados en el servidor:

```
...

public class DeviceSessionHandler {

   private int deviceId = 0;
   private final Set sessions = new HashSet<>();
   private final Set devices = new HashSet<>();

   ...

}
```

Añadimos un bucle for en addSession para enviar la lista de devices a todos los clientes conectados:

```
public void addSession(Session session) {
      sessions.add(session);
      for (Device device : devices) {
         JsonObject addMessage = createAddMessage(device);
         sendToSession(session, addMessage);
      }

   }
```

Implementamos el método addDevice:

```
public void addDevice(Device device) {
      device.setId(deviceId);
      devices.add(device);
      deviceId++;
      JsonObject addMessage = createAddMessage(device);
      sendToAllConnectedSessions(addMessage);
   }
```

Este método crea un nuevo objeto y envía un mensaje JSON a todas las sesiones conectadas.

Implementamos el método removeDevice, que elimina un device por id y envía un mensaje en JSON a todas las sesiones conectadas:

```
public void removeDevice(int id) {
      Device device = getDeviceById(id);
      if (device != null) {
         devices.remove(device);
         JsonProvider provider = JsonProvider.provider();
         JsonObject removeMessage = provider.createObjectBuilder()
               .add("action", "remove")
               .add("id", id)
               .build();
         sendToAllConnectedSessions(removeMessage);
      }
   }
```

Implementamos toggleDevice que cambia el estado de un dispositivo y envía mensaje JSON a todas las sesiones:

```
public void toggleDevice(int id) {
    JsonProvider provider = JsonProvider.provider();
    Device device = getDeviceById(id);
    if (device != null) {
      if ("On".equals(device.getStatus())) {
        device.setStatus("Off");
      } else {
        device.setStatus("On");
      }
      JsonObject updateDevMessage = provider.createObjectBuilder()
          .add("action", "toggle")
          .add("id", device.getId())
          .add("status", device.getStatus())
          .build();
      sendToAllConnectedSessions(updateDevMessage);
    }
  }
```

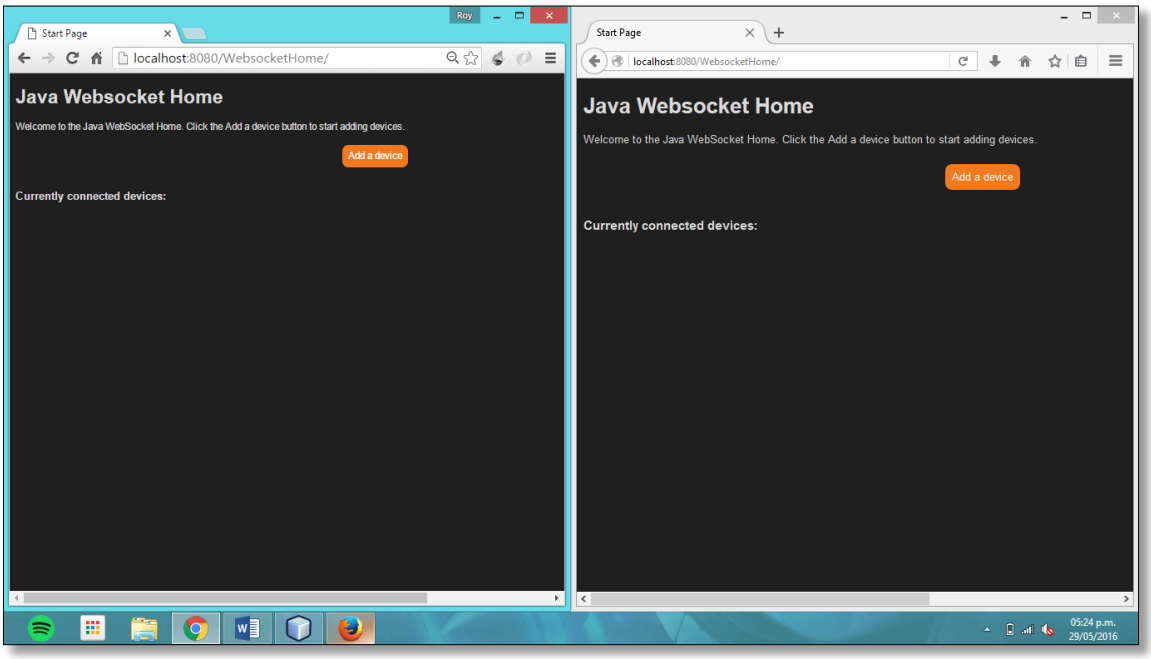
Implementamos los métodos que faltan:

```
private Device getDeviceById(int id) {
    for (Device device : devices) {
      if (device.getId() == id) {
        return device;
      }
    }
    return null;
  }

  private JsonObject createAddMessage(Device device) {
    JsonProvider provider = JsonProvider.provider();
    JsonObject addMessage = provider.createObjectBuilder()
        .add("action", "add")
        .add("id", device.getId())
        .add("name", device.getName())
        .add("type", device.getType())
        .add("status", device.getStatus())
        .add("description", device.getDescription())
        .build();
    return addMessage;
  }

  private void sendToAllConnectedSessions(JsonObject message) {
    for (Session session : sessions) {
      sendToSession(session, message);
    }
  }

  private void sendToSession(Session session, JsonObject message) {
    try {
```

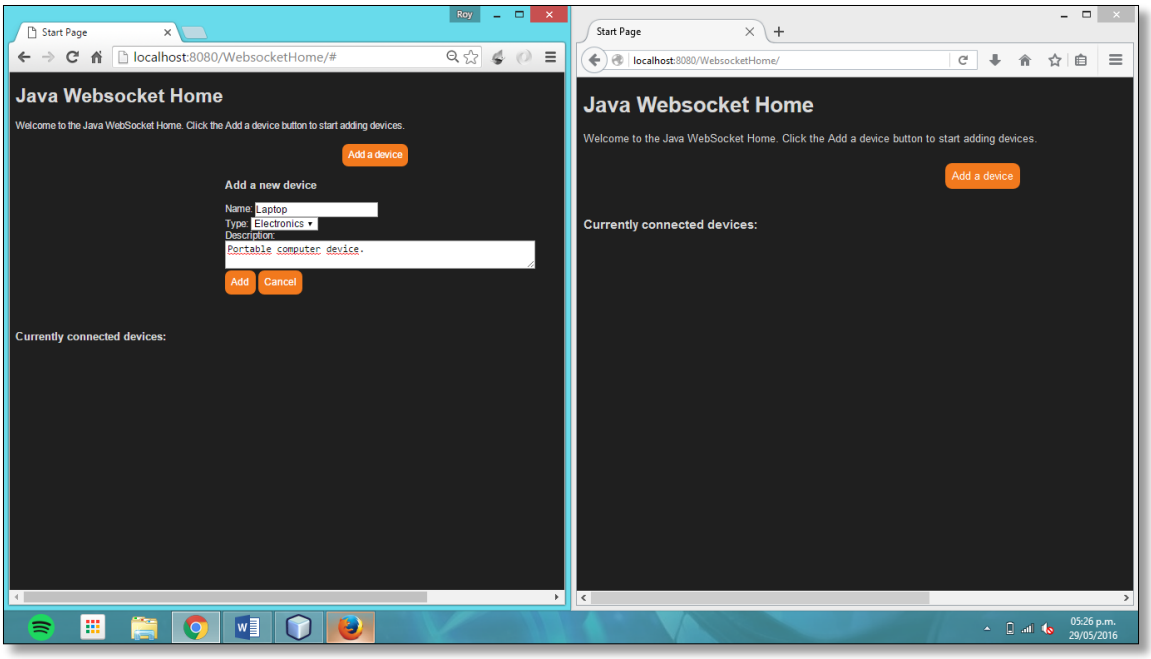
```
            session.getBasicRemote().sendText(message.toString());
        } catch (IOException ex) {
            sessions.remove(session);
            Logger.getLogger(DeviceSessionHandler.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Probando la aplicación WebSocket Home

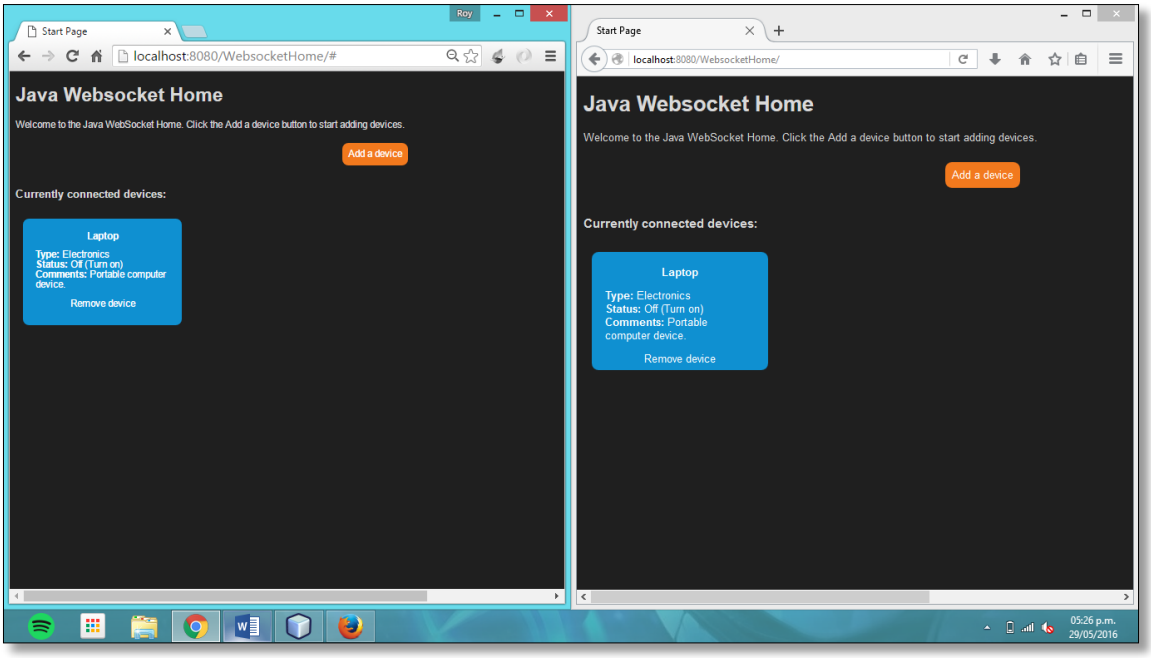
Ejecutamos la aplicación y la abrimos con Google Chrome y Firefox:



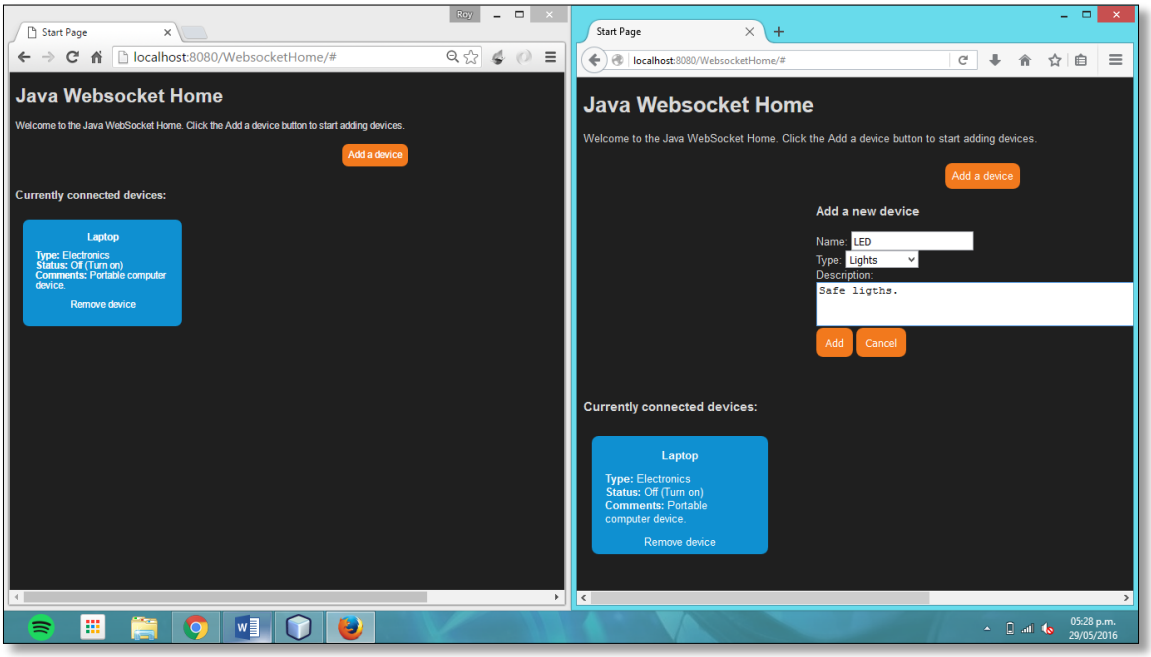
Añadimos un objeto device:

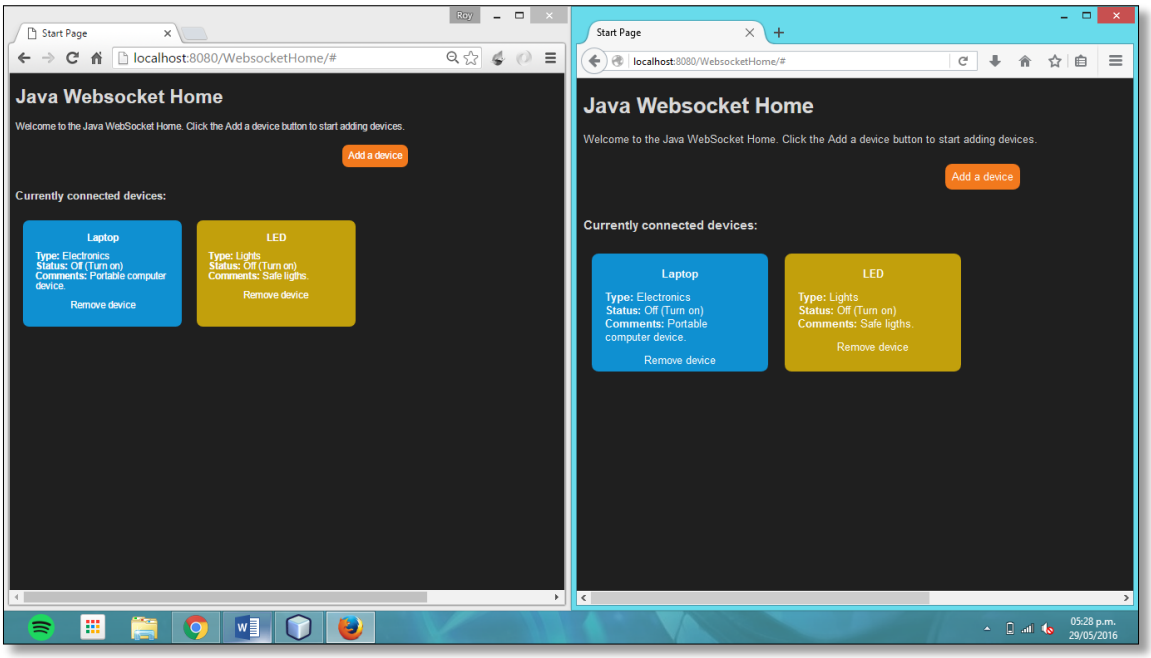


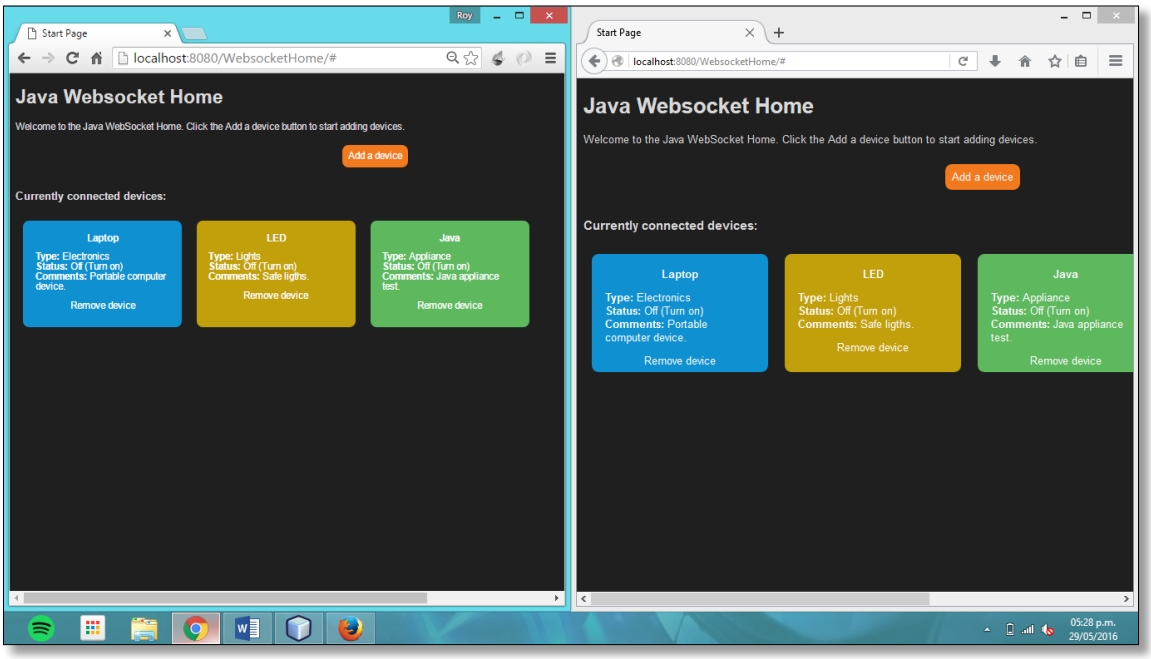
Al hacer clic en Add:



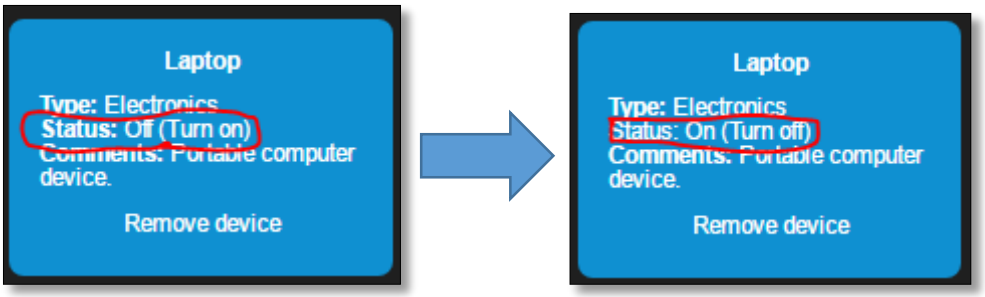
Añadimos más dispositivos:



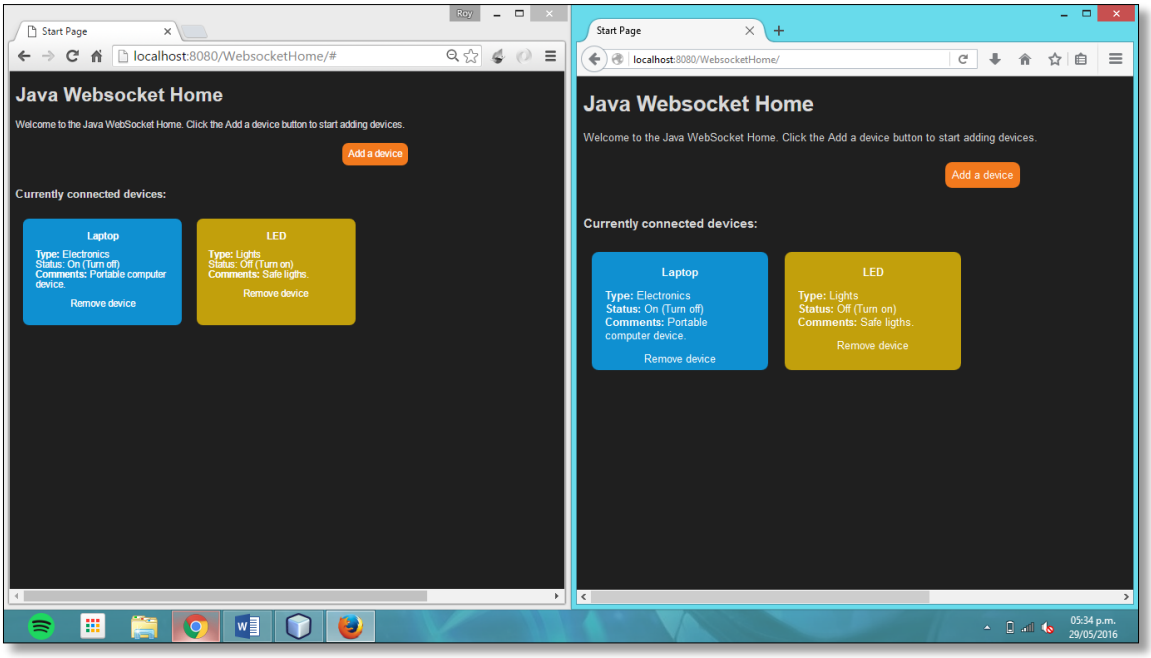




Si cambios el estado de un device en ambos clientes se verá el cambio:



Removemos un device:



Conclusiones

Se aprendió:

- Definir un endpoint de servidor WebSocket en una clase Java mediante el uso de las API de anotaciones WebSocket.
- Enviar mensajes hacia y desde el cliente al servidor WebSocket en JSON.
- Utilizar las anotaciones API WebSocket de ciclo de vida para controlar los eventos WebSocket.
- Eventos del proceso de vida WebSocket del cliente mediante el uso de HTML5 y JavaScript.