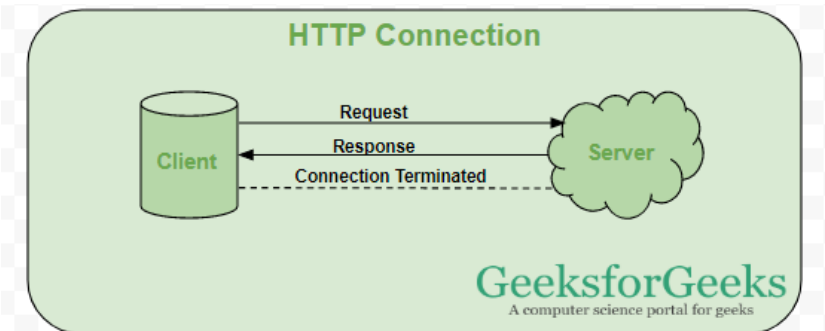
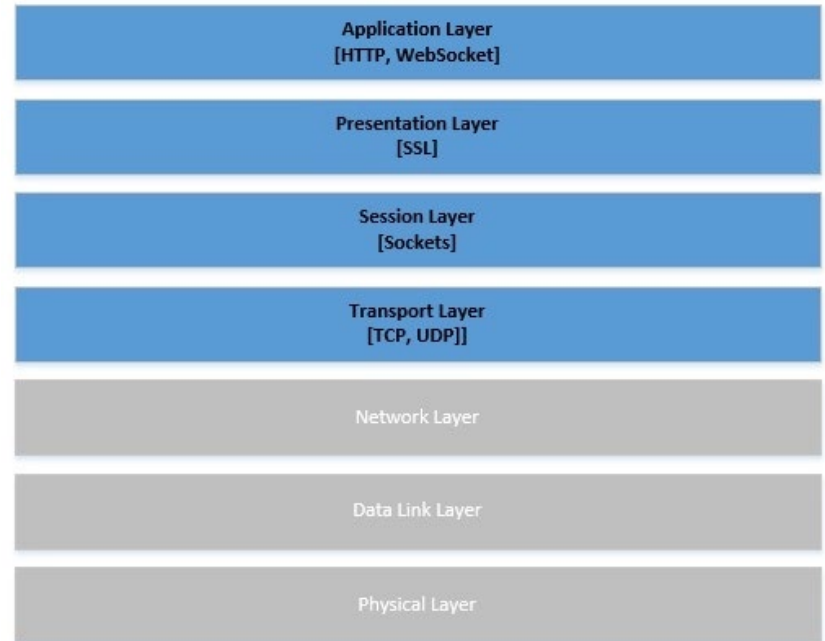


CT5106

WebSocket

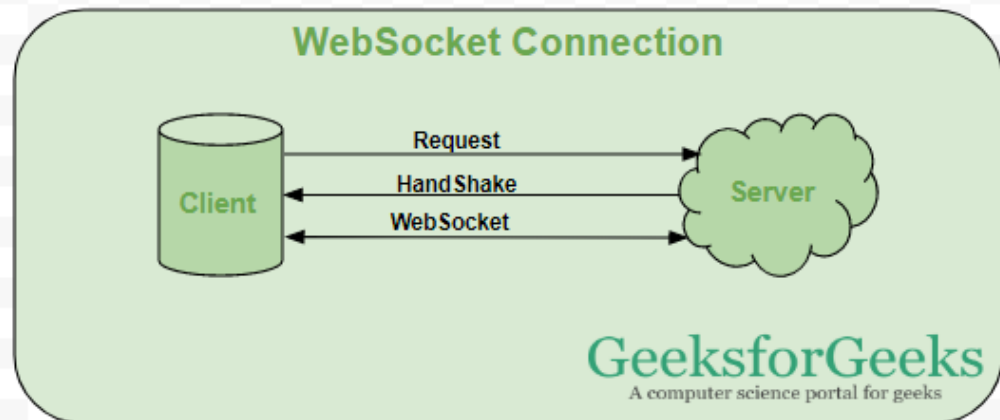
Introduction to WebSocket

- Protocols used in the application layer include HTTP and WebSocket
- HTTP is called *unidirectional* in that the client sends the request and the server sends a response
 - After which the connection is closed
- HTTP is stateless, runs on top of TCP
 - Each HTTP request opens a separate TCP connection to the server



WebSocket

- WebSocket is bidirectional
 - `ws://` or `wss://`
 - Stateful – connection is kept alive until client or server terminates it
- Once connected through an HTTP request/response pair
 - the client can use a mechanism called an upgrade header to switch their connection from HTTP over to WebSockets
 - A WebSocket connection is established through a websocket handshake over the TCP

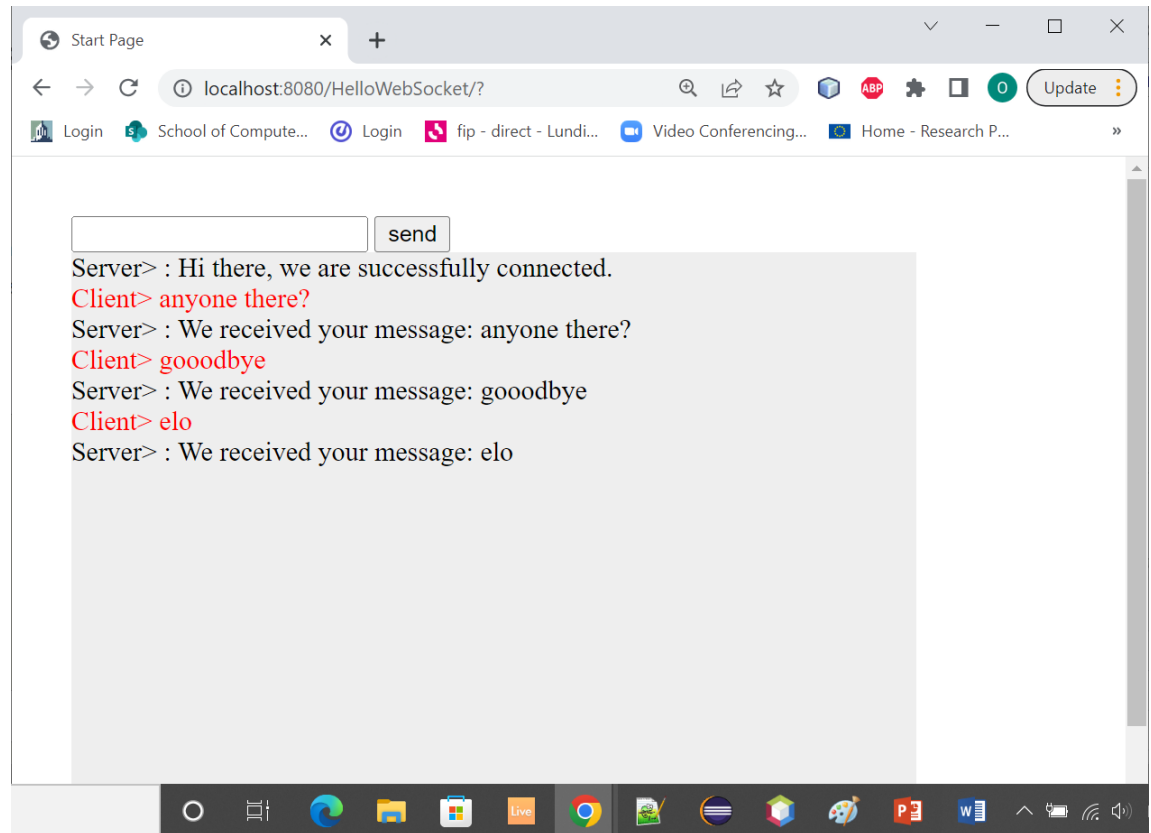


Use for WebSocket

- When you want continuous updates between client and server, e.g.
 - ▣ Real-time trading
 - ▣ Gaming
 - ▣ Chat
- When not to use WebSocket
 - ▣ When accessing persisted (old) data on a once-off / infrequent basis
 - RESTful web services are sufficient in such cases

Simple example app

- A simple app which allows client to send message to server
- Server in this case just sends acknowledgement back



Creating a connection

- The connection is initialised by the client, using Javascript

```
var websocket = new WebSocket("ws://localhost:8080/HelloWebSocket/hello");
```

- In this case the Javascript is simply embedded in the index.html, but usually it would be in a separate file
- When we have established the socket connection, then we can send messages to the server very simply

```
websocket.send(msgToSend);
```

Web Sockets Events

- There are 4 main events
 - Open
 - Message
 - Close
 - Error

Open

- Once the connection has been established between the client and the server, the open event is fired from Web Socket instance. It is called as the initial handshake between client and server. The event, which is raised once the connection is established, is called onopen.
- Useful for debugging to write messages to the console

```
websocket.onopen = function ()  
{  
    console.log("connection opened");  
};
```


Message

- Message event happens usually when the server sends some data. Messages sent by the server to the client can include plain text messages, binary data or images. Whenever the data is sent, the **onmessage** function is fired

```
websocket.onmessage = function (message)
{
    divMsg.innerHTML += "Server> : " + message.data;
};
```

Close

- Close event marks the end of the communication between server and the client. Closing the connection is possible with the help of **onclose** event. After marking the end of communication with the help of **onclose** event, no messages can be further transferred between the server and the client. Closing the event can happen due to poor connectivity as well.
- Closing the browser tab will trigger the close message being sent to the server

```
websocket.onclose = function ()  
{  
    console.log("connection closed");  
};
```

Error

- Error marks for some mistake, which happens during the communication. It is marked with the help of **onerror** event. **Onerror** is always followed by termination of connection

```
websocket.onerror = function werror(message)
{
    console.log("error: " + message);
};
```

Web Sockets Actions

- Events are usually triggered when something happens. On the other hand, actions are taken when a user wants something to happen. Actions are made by explicit calls using functions by users.
- The Web Socket protocol supports two main actions, namely –
 - ▣ `send()`
 - ▣ `close()`

send()

- This action is usually preferred for some communication with the server, which includes sending messages, which includes text files, binary data or images

```
function sendMsg()
{
    websocket.send(msgToSend);
    divMsg.innerHTML += "<div style='color:red'>Client> " + msgToSend + "</div>";
    msgField.value = "";
};
```

close()

- This method stands for goodbye handshake. It terminates the connection completely and no data can be transferred until the connection is re-established.

```
socket.close( );
```

Handling String messages from server

- WebSocket supports text and binary data
- Strings in the form of JSON are very useful for communication and transferring information
- Can check the type of data received if necessary

```
socket.onmessage = function(event)
{
  if ( typeof event.data === String )
  {
    console.log("Received data string");
  }
}
```

JSON (JavaScript Object Notation)

- lightweight format for transferring human-readable data between the computers. The structure of JSON consists of key-value pairs
- Accessing the data is fairly easy using js

```
{  
  sender: "Ari",  
  message: "Time for Tea"  
}
```

```
socket.onmessage = function(event)  
{  
  if(typeof event.data === String )  
  {  
    //create a JSON object  
    var jsonObject = JSON.parse(event.data);  
    var sender = jsonObject.sender;  
    var message = jsonObject.message;  
  
    console.log("Received data string");  
  }  
}
```


Server end

HelloWorldEndpoint.java

- At it's most basic, we simply need to annotate a class as a server endpoint, e.g.

```
@ServerEndpoint("/hello")  
public class HelloWorldEndpoint
```

- The endpoint is annotated to receive messages coming in with a particular path
- The endpoint must also handle the 4 events types
 - ▣ Open
 - ▣ Error
 - ▣ Message
 - ▣ Close

Open

HelloWorldEndpoint.java

- Normally you would want to keep track of the different clients, but in this basic example, we just echo back the clients message directly to them
- To send messages to the other end of the 'conversation' you get the remote endpoint from the session
- Each client-server connection creates a new session – a `MessageHandler` can be registered for this session to handle incoming messages (in another example)

```
@OnOpen
public void onOpen(Session session)
{
    System.out.printf("Session opened, id: %s%n", session.getId());
    try
    {
        session.getBasicRemote().sendText("Hi there, we are successfully connected.");
    } catch (IOException ex) { }
}
```

@OnClose

HelloWorldEndpoint.java

- onClose trigger when client closes the session

```
@OnClose
public void onClose(Session session)
{
    System.out.printf("Session closed with id: %s%n", session.getId());
}
```

- If the server endpoint is keeping track of clients it would remove this one from the list

@OnMessage

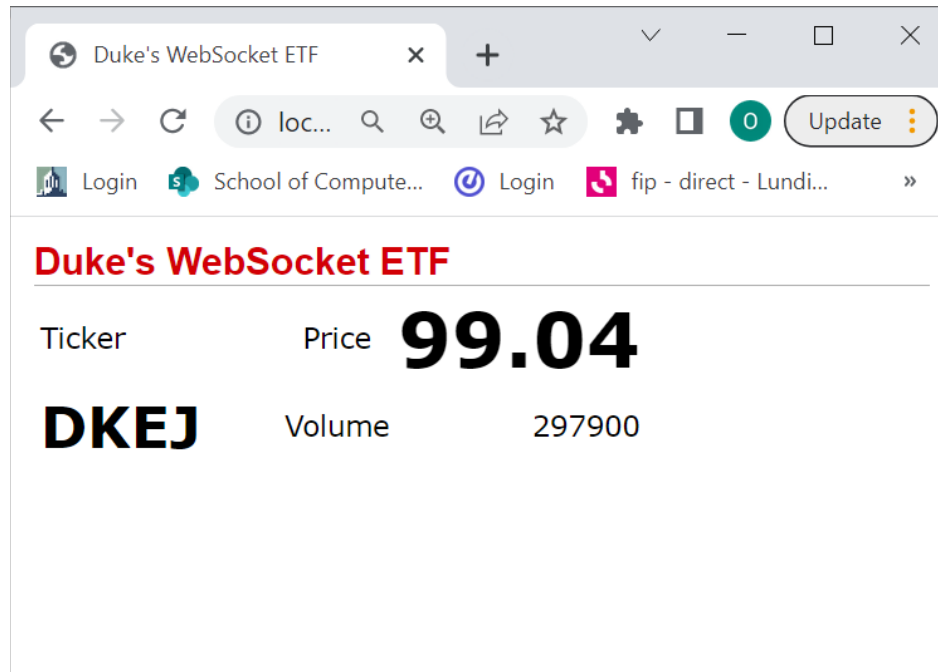
HelloWorldEndpoint.java

- The **OnMessage** event is raised when a client sends data to the server – the session (to identify the client) is included
- Inside this event handler, the incoming message can be transmitted to all clients, or perhaps just selected ones

```
@OnMessage
public void onMessage(String message, Session session) {
    System.out.printf("Message received. Session id: %s Message: %s%n", session.getId(),
message);
    try {
        session.getBasicRemote().sendText(String.format("We received your message: %s%n",
message));
    } catch (IOException ex) {
    }
}
```

Second example

- dukeetf – a version of part of the standard Java EE tutorial
- This app sends updated stock price and trade volume for a fictitious stock every second – to every client



The screenshot shows a web browser window titled "Duke's WebSocket ETF". The address bar contains "loc..." and a search icon. The browser's taskbar shows several open tabs: "Login", "School of Compute...", "Login", and "fip - direct - Lundi...". The main content area displays the title "Duke's WebSocket ETF" in red. Below the title, there is a table with the following data:

Ticker	Price	99.04
DKEJ	Volume	297900

PriceVolumeBean

PriceVolumeBean.java

- This is a single session bean which is instantiated immediately on application startup

```
@Startup  
@Singleton  
public class PriceVolumeBean
```

- It uses the application container's timer service

```
@Resource TimerService tservice;
```

This creates a timer interval, set to elapse after 1000ms and every 1000ms thereafter

```
tservice.createIntervalTimer(1000, 1000, new TimerConfig());
```

- This method is called by the container every time the timer elapses

```
@Timeout  
public void timeout() { ... }
```

The endpoint class

ETFEndpoint.java

- This class is annotated as the endpoint for websocket calls coming in on this path

```
@ServerEndpoint("/dukeetf")  
public class ETFEndpoint {
```

- To keep track of the client sessions, their Session's are stored in a queue

```
static Queue<Session> queue = new ConcurrentLinkedQueue<>();
```

```
@OnOpen  
public void openConnection(Session session) {  
    /* Register this connection in the queue */  
    queue.add(session);  
}
```

If a client closes a connection

ETFEndpoint.java

- Then we only remove that client's session from the queue

```
@OnClose
public void closedConnection(Session session) {

    /* Remove this connection from the queue */
    queue.remove(session);

    logger.log(Level.INFO, "Connection closed.");
}
```


In case of error from client

ETFEndpoint.java

- Again handle that as an isolated client problem (it could be)

```
@OnError
public void error(Session session, Throwable t) {

    /* Remove this connection from the queue */
    queue.remove(session);

    logger.log(Level.INFO, t.toString());
    logger.log(Level.INFO, "Connection error.");
}
```

Sending to the clients

ETFEndpoint.java

- Every 1000ms the session bean calls the *send* method of the endpoint

```
ETFEndpoint.send(price, volume);
```

- In the endpoint *send* method, the string message is created and formatted, and send to all the clients in the queue, using their session objects to get the remote endpoint of the websocket connection

```
public static void send(double price, int volume)
{
    String msg = String.format("%.2f / %d", price, volume);
    try
    {
        ...
        for (Session session : queue) {
            session.getBasicRemote().sendText(msg);
            ..
        }
        ...
    }
}
```

Message received at client

- The js script on the client end splits the incoming string using the “/” delimiter and put’s the text into specific table cells

```
function onMessage(evt)
{
    var arraypv = evt.data.split("/");
    document.getElementById("price").innerHTML = arraypv[0];
    document.getElementById("volume").innerHTML = arraypv[1];
}
```

Example 3 – multiple clients sharing information

localhost:8080/WebSocketTest/#

localhost:8080/WebSocketTest/#

Login School of Compute... Login fip - direct - Lundi... Video Conferencing... Home - Research P... Blackboard NUIG Sites

Java WebSocket Home

Welcome to the Java WebSocket Home. Click the Add a device button to start adding devices.

Add a device

Add a new device

Name:

Type:

Description:

Add Cancel

Currently connected devices:

Computer
Type: Electronics
Status: Off (Turn on)
Comments: My Laptop
Remove device

Desk Lamp
Type: Lights
Status: Off (Turn on)
Comments: My desk lamp
Remove device

Web app description

- The app allows multiple clients to share information on currently used devices (e.g. electrical)
- Each new session (client) is given the current list of devices to display
- Each client can add or remove devices as well as toggle each device's status (on / off)
 - ▣ Updates are shared with all other devices

Device class

Device.java

- A simple Java Bean class

```
public class Device {  
  
    private int id;  
    private String name;  
    private String status;  
    private String type;  
    private String description;  
  
    // + constructor, getters and setters
```

The web socket server

DeviceWebSocketServer.java

- One instance per application, responding to client messages on the `/actions` path

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

- Uses an `@ApplicationScoped` bean to manage the sessions

```
@Inject  
private DeviceSessionHandler sessionHandler;
```

Delegates handling of events to handler

- The `@OnMessage` has a bit of work to it – the method parses the incoming JSON to either get the full info for a new Device, or it's id
- First get the JSON object (we assume the incoming string is formatted correctly!)

```
@OnMessage
public void handleMessage(String message, Session session) {
    try (JsonReader reader = Json.createReader(new StringReader(message))) {
        JsonObject jsonMessage = reader.readObject();
    }
}
```


If adding a device

- If the action value of the JSON string is “add” then parse out the properties and create a new Device object
- Then ask the session handler to add it to the list, and inform all the clients

```
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);  
}
```

- Code continued
- Remove and toggle only need the id of the 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);  
}
```

```
}  
}
```

Session handler

DeviceSessionHandler.java

- Application scoped bean
- Keeps track of sessions and devices

```
@ApplicationScoped
public class DeviceSessionHandler {

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

If a new client connects

DeviceSessionHandler.java

- Then add it to the list of sessions
- Also send an 'add device' message to the client for each Device in the list
- The client will need to create a new div on the HTML page holding the Device information

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

On the client side

- Too much to go through in slides here
- Check out `websocket.js` – the code the client uses to manage incoming and outgoing messages
- Worth looking at how the `printDeviceElement` method builds a new `<div>` for a new device and adds it to the document
- Each device `<div>` is given the device id, so this makes removal, and toggling of device status easier