Websockets-02 Simple Examples

1 Recap of Websockets-1

Websocket is a connection protocol that is compatible with HTTP. Websockets allow full-duplex and persistent communications. This allows the server to send messages to clients and is useful for broadcasts and games etc.

On the server side, the websocket URL needs to be registered. After that, server can respond to open, message, close events. Server can send messages to clients. On the client side, client needs to connect to server. After that, client can respond to open, message, close events. Client can send messages to server.

ws://echo.websocket.org is a websocket server (that you can use to test your client codes).

websocket.org/echo.html is a websocket client that you can use to test your websocket server codes)

2 About this document

Here we will present and discuss three examples.

- HTML/JavaScript Client code
- Springboot Server code
- Android Client code

These examples can be found at https://git.linux.iastate.edu/cs309/tutorials.git

3 HTML/Javascript Client Code

3.1 HTML5 Websocket specification

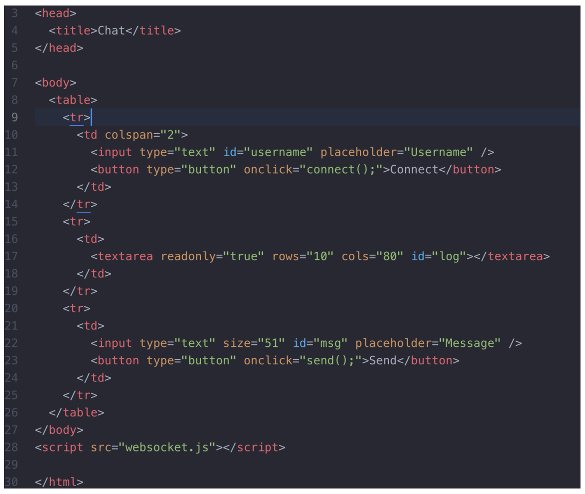
The latest specs for HTML is at https://html.spec.whatwg.org/multipage/

The latest specs for WebSockets is https://html.spec.whatwg.org/multipage/web-sockets.html#network. It is included directly in HTML5 (i.e. no need to include special javascript libraries). There is basically a **constructor** that returns a new WebSocket object. This object has a send method. It also calls handlers for **open**, **error**, **close**, on receiving a **message**.

3.2 Example HTML/JS code

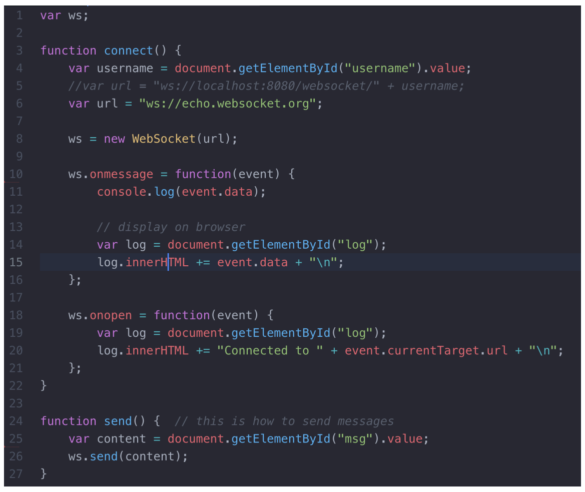
These examples can be found at https://git.linux.iastate.edu/cs309/tutorials.git

HTML



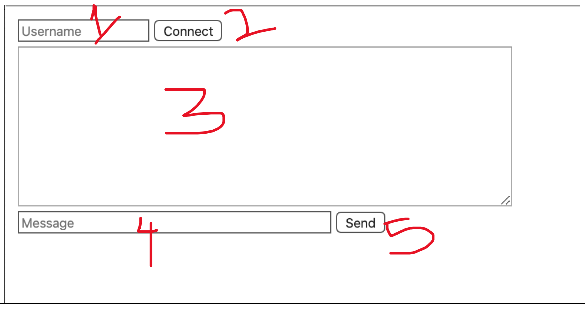
```html
<head>
  <title>Chat</title>
</head>

<body>
  <table>
    <tr>
      <td colspan="2">
        <input type="text" id="username" placeholder="Username" />
        <button type="button" onclick="connect();">Connect</button>
      </td>
    </tr>
    <tr>
      <td>
        <textarea readonly="true" rows="10" cols="80" id="log"></textarea>
      </td>
    </tr>
    <tr>
      <td>
        <input type="text" size="51" id="msg" placeholder="Message" />
        <button type="button" onclick="send();">Send</button>
      </td>
    </tr>
  </table>
</body>
<script src="websocket.js"></script>

</html>
```

Javascript code



```javascript
1  var ws;
2
3  function connect() {
4      var username = document.getElementById("username").value;
5      //var url = "ws://localhost:8080/websocket/" + username;
6      var url = "ws://echo.websocket.org";
7
8      ws = new WebSocket(url);
9
10     ws.onmessage = function(event) {
11         console.log(event.data);
12
13         // display on browser
14         var log = document.getElementById("log");
15         log.innerHTML += event.data + "\n";
16     };
17
18     ws.onopen = function(event) {
19         var log = document.getElementById("log");
20         log.innerHTML += "Connected to " + event.currentTarget.url + "\n";
21     };
22 }
23
24 function send() {  // this is how to send messages
25     var content = document.getElementById("msg").value;
26     ws.send(content);
27 }
```

HTML PAGE when displayed



1 is username text entry field
2 is connect button
3 is log text area display field
4 is msg text entry field
5 is send button

Here, we have very simple websocket code.

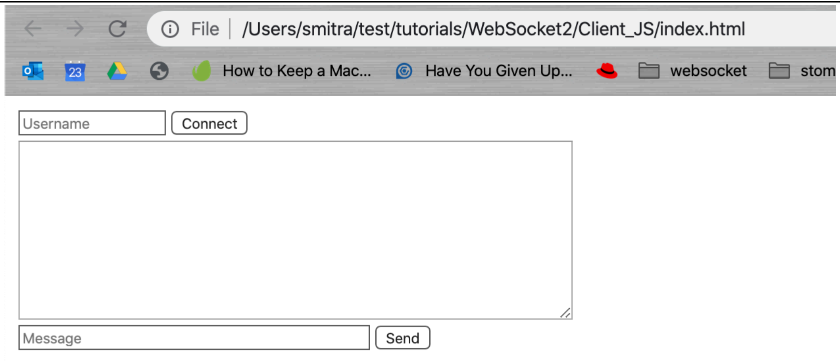
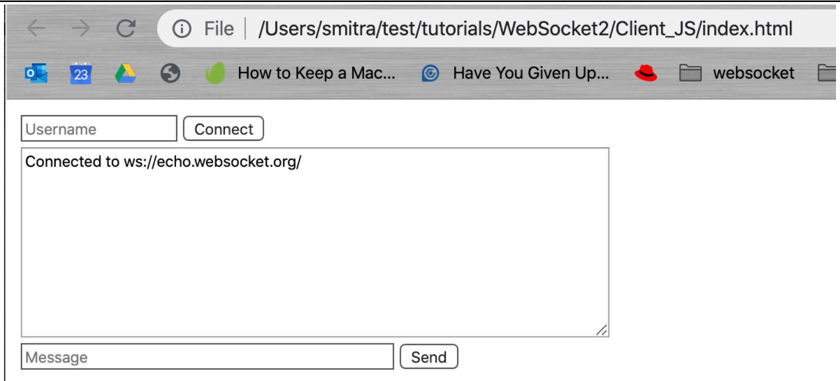
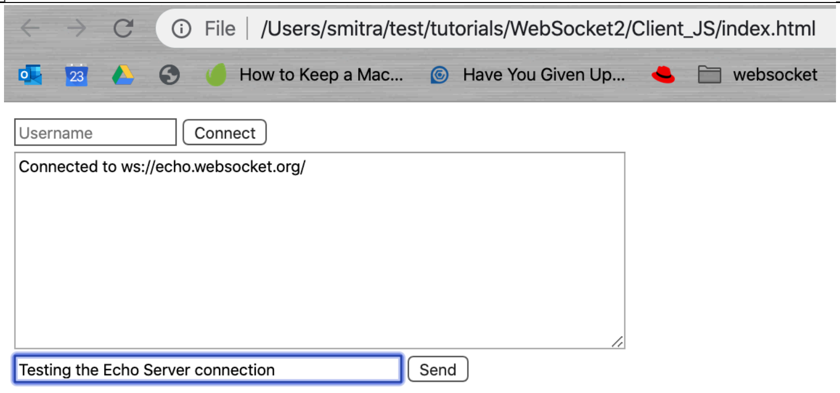
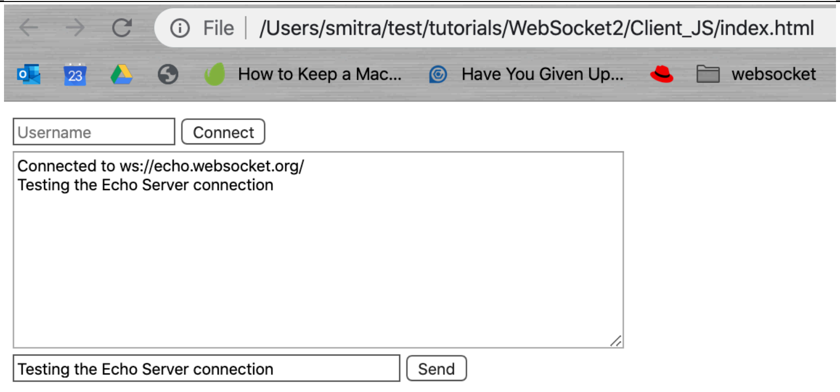
On line 8, we CREATE a new websocket object. This connects using websocket protocol to the given URL. This is called when the connect button (i.e. area 2) is clicked.

On line 10, we attach a handler for what to do when a message is received from the server. Here we simply print the message to the browser console and also append it to the log area (i.e. area 3) of the displayed page.

On line 18, we attach a handler for what to do when the connection is OPENED (i.e. when the connection occurs). Here we simply append the Connection message to the log area (i.e. area 3) of the displayed page.

On line 24, we have code for the send method. This method sends the text from the msg area (i.e. area 4) to the server. This is called when the send button (i.e. area 5) is clicked.

3.3 Testing the code

What you do	What happens on the browser display
1. Open the index.html in your browser	 File How to Keep a Mac... Have You Given Up... websocket stom Username Connect Message Send /Users/smitra/test/tutorials/WebSocket2/Client_JS/index.html
2. Click on connect Note the message in the log area.	 File How to Keep a Mac... Have You Given Up... websocket Username Connect Message Send /Users/smitra/test/tutorials/WebSocket2/Client_JS/index.html Connected to ws://echo.websocket.org/
3. Type "Testing the Echo Server connection" in the msg text area	 File How to Keep a Mac... Have You Given Up... websocket Username Connect Send /Users/smitra/test/tutorials/WebSocket2/Client_JS/index.html Connected to ws://echo.websocket.org/ Testing the Echo Server connection
4. Click on send button. Note the message in the log area. You can repeat this last step if you want.	 File How to Keep a Mac... Have You Given Up... websocket Username Connect Send /Users/smitra/test/tutorials/WebSocket2/Client_JS/index.html Connected to ws://echo.websocket.org/ Testing the Echo Server connection Testing the Echo Server connection

4 Java Websocket Specification

Here is a good reference (https://www.baeldung.com/java-websockets)

The WebSocket protocol standard is RFC 6455.

JSR 356 is the Java API for Websockets protocol! Download here (https://download.oracle.com/otndocs/jcp/websocket-1_0-fr-eval-spec/index.html)

This JSR 356 API is a SPECIFICATION and describes **both** the Server side AND the Client side. Note that there could be many different implementations for the same specs.

The javax.websocket library provides one IMPLEMENTATION for this api. Underneath the covers, the API implements the RFC 6455 websocket protocol.

In Springboot, we will focus on using ANNOTATIONS to use websockets. Another way is to use programmatic means to create and use websockets (which the Android implementation will use).

Note that this specs do not have anything to do with either springboot or android.

On Server Side:

- Register the URL. Use class-level annotation @ServerEndpoint to indicate that the class is a websocket target endpoint. The URL is specified here.
  Example: @ServerEndpoint("/websocket/{username})

- Handle Events
  - Use method-level annotations @OnOpen, @OnClose, @OnMessage, @OnError

- Sending: <socket-object>.getBasicRemote().sendText(message); The OnOpen gives the client socket information that we can save to send to the client later on in the code.

- Take a look at the springboot server example to see details of how these are used.

On Client Side:

- Connect to Server (new WebSocket)
- Handle Events
  - Use method-level annotations @OnOpen, @OnClose, @OnMessage, @OnError
- Similar sending mechanism

5 Springboot Server Code

5.1 The big picture

There are several things that need to get done.

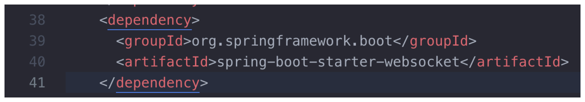
(1) Dependencies must be setup in pom.xml (sprint-boot-starter-websocket)
(2) Server websocket endpoint must be registered @serverendpoint
(3) Server side event handling must be setup @onopen etc
(4) Code to send message to clients must be setup sendText etc

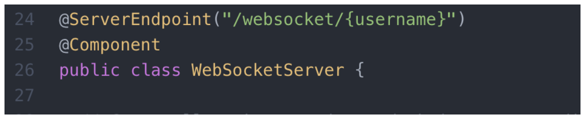
Note that these annotations have nothing to do with Springboot. These are javax.websocket annotations. In order to allow springboot to recognize our websockets, we must ALSO register it to springboot.

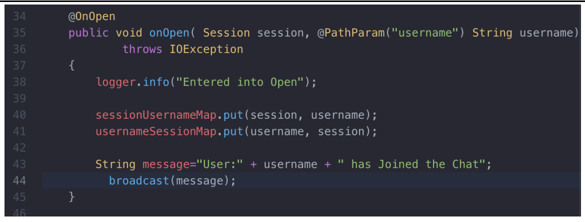
(5) Register websocket endpoint and make it known to springboot

These examples can be found at https://git.linux.iastate.edu/cs309/tutorials.git

5.2 Example Code Segments for each of the five parts.

1. Add dependency for javax.websocket library to pom.xml	 `<dependency>`
	`  <groupId>org.springframework.boot</groupId>`
	`  <artifactId>spring-boot-starter-websocket</artifactId>`
	`</dependency>`

2. Server websocket endpoint must be registered @ServerEndpoint
(see WebSocketServer.java in example)
 	
The @Component makes sure that spring will scan this class.
	 `@ServerEndpoint("/websocket/{username}")`
	`@Component`
	`public class WebSocketServer {`
	
	Users will connect to websocket at ws://host:port/websocket/{**username**}
	example:
	ws://localhost:8080/websocket/smitra
	The **username** parameter can be accessed in the onOpen handler.

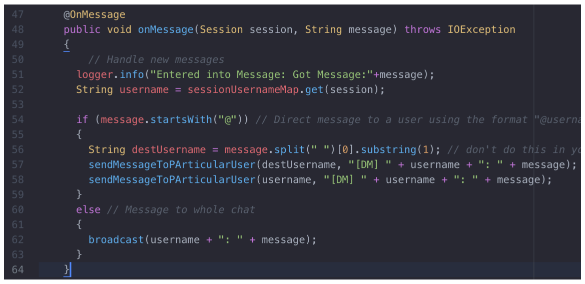
3a) Server side handling of onOpen
 	
note: see use of @PathParam to extract username in the @onOpen annotated method.
 	
note: see how session is passed to the websocket – this is important. The session consists of the websocket handle to be able to send message back to client.
	 `@OnOpen`
	`public void onOpen( Session session, @PathParam("username") String username)`
	`        throws IOException`
	`{`
	`    logger.info("Entered into Open");`
	
	`    sessionUsernameMap.put(session, username);`
	`    usernameSessionMap.put(username, session);`
	
	`    String message="User:" + username + " has Joined the Chat";`
	`      broadcast(message);`
	`}`
	
	There is a map where key is session and value is username.
	
	There is another map where key is username and value is session. So given one, we can easily get the other

3b) Server side handling of onMessage

This is called when a message is received from the client.

It knows who sent it (given session, can find the username).

It knows whom to send the message to (either to @user or broadcast to everyone).

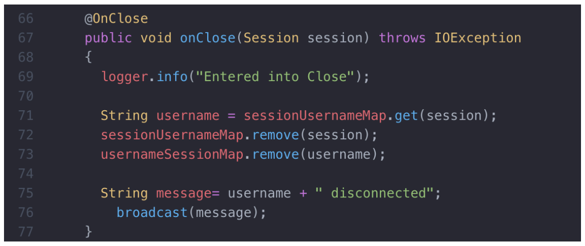


```java
    @OnMessage
    public void onMessage(Session session, String message) throws IOException
    {
        // Handle new messages
      logger.info("Entered into Message: Got Message:"+message);
      String username = sessionUsernameMap.get(session);

      if (message.startsWith("@")) // Direct message to a user using the format "@userna
      {
        String destUsername = message.split(" ")[0].substring(1); // don't do this in yo
        sendMessageToPArticularUser(destUsername, "[DM] " + username + ": " + message);
        sendMessageToPArticularUser(username, "[DM] " + username + ": " + message);
      }
      else // Message to whole chat
      {
        broadcast(username + ": " + message);
      }
    }
```

3c) Server side handling of onClose

When either the client or the server shuts down the websocket, this method is called.

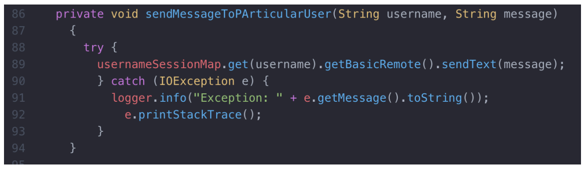
Here we simply remove the appropriate entries from the two maps and then broadcast a closing message to all the other clients.



```java
    @OnClose
    public void onClose(Session session) throws IOException
    {
      logger.info("Entered into Close");

      String username = sessionUsernameMap.get(session);
      sessionUsernameMap.remove(session);
      usernameSessionMap.remove(username);

      String message= username + " disconnected";
        broadcast(message);
    }
```

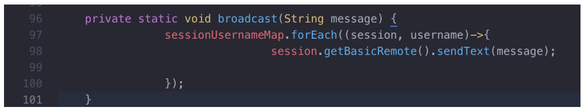
4a) Server side code to send message to client

Note how getBasicRemote() is used to send text message back to the client.



```java
    private void sendMessageToPArticularUser(String username, String message)
      {
        try {
          usernameSessionMap.get(username).getBasicRemote().sendText(message);
          } catch (IOException e) {
            logger.info("Exception: " + e.getMessage().toString());
              e.printStackTrace();
          }
      }
```

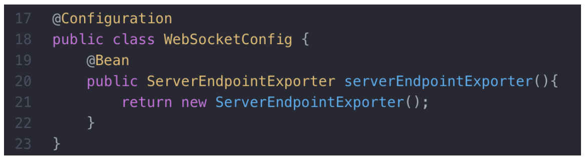
4b) Server side code to broadcast message to all clients
Loop thru all the users and send them the message.



```java
    private static void broadcast(String message) {
                sessionUsernameMap.forEach((session, username)->{
                                session.getBasicRemote().sendText(message);

                });
    }
```

5 Server side code to register websocket to Springboot
(see WebSocketConfig.java)

Basically, when springboot starts, it will see that serverendpoint needs to be considered. Then, it will accept ws:// requests in addition to http:// requests

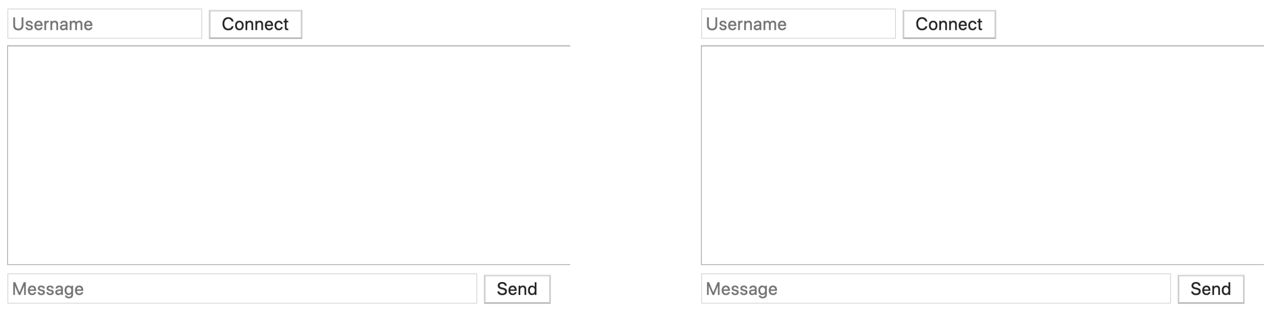


```java
@Configuration
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter(){
        return new ServerEndpointExporter();
    }
}
```

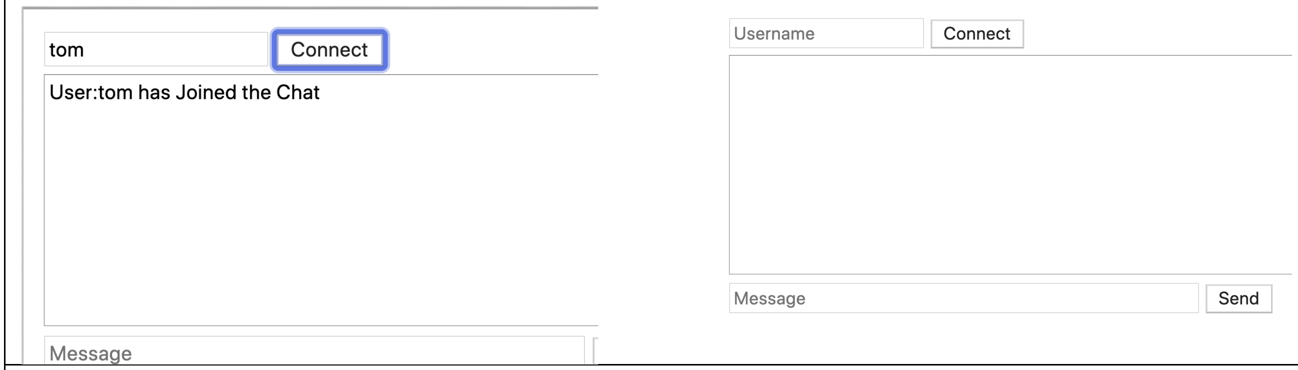
5.3 Testing the code

First, build the springboot project by typing "mvn package" in the directory where the pom.xml is.
Next, run the server by typing
java -jar WebSocketServer-0.0.1-SNAPSHOT.jar in the target folder
The server will start up!

Go to WebSockets/Client_JS folder and then open index.html in a browser.
You have to go to websocket.js and change the URL to localhost.
Open another one in another tab (or window) of your browser. You will get the below.

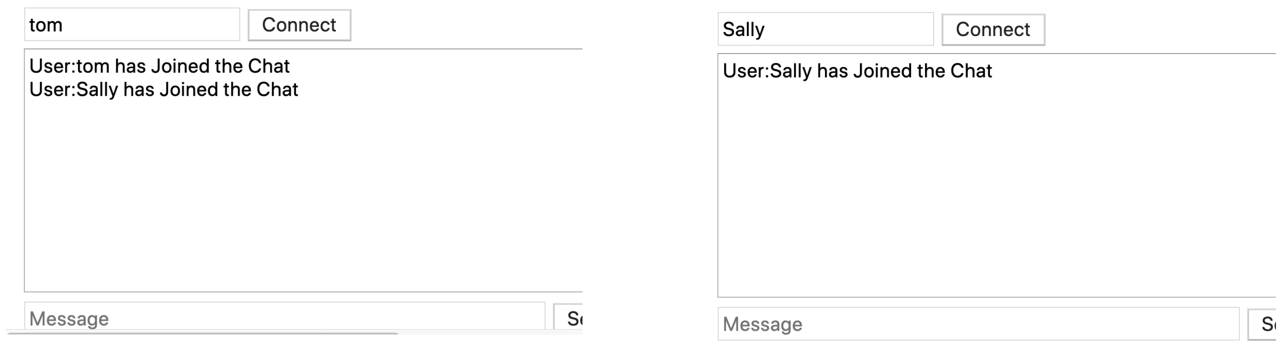


In the left hand type a username (say Tom) and connect. The right hand side will remain unconnected.



```
User:tom has Joined the Chat
```

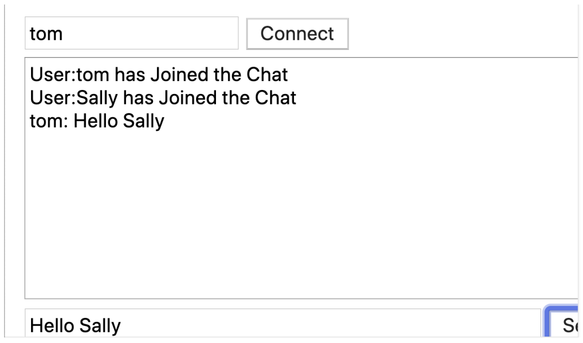
In the right hand type a username (say Sally) and connect. Notice that the message was broadcast to both the users.



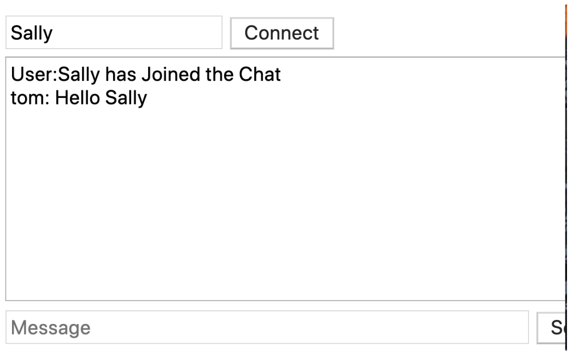
```
User:tom has Joined the Chat
User:Sally has Joined the Chat
```

```
User:Sally has Joined the Chat
```

In the left hand type a message "Hello Sally". See how that is broadcast.

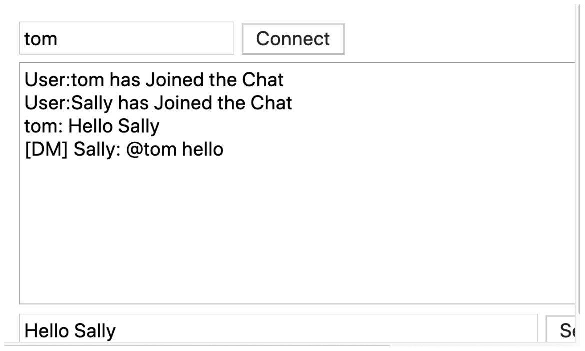


```
User:tom has Joined the Chat
User:Sally has Joined the Chat
tom: Hello Sally
```

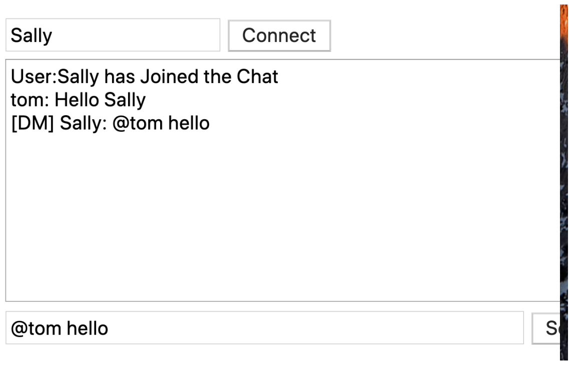


```
User:Sally has Joined the Chat
tom: Hello Sally
```

In the right hand send message "@tom Hello". See how it is sent just to tom.



```
User:tom has Joined the Chat
User:Sally has Joined the Chat
tom: Hello Sally
[DM] Sally: @tom hello
```



```
User:Sally has Joined the Chat
tom: Hello Sally
[DM] Sally: @tom hello
```

You can open more clients and try this.

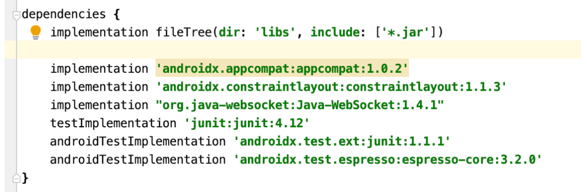
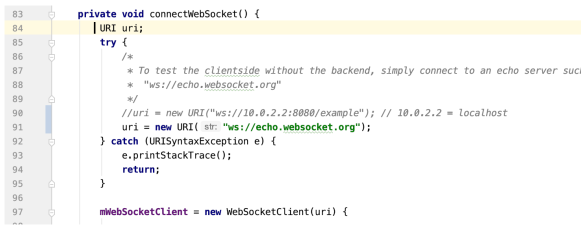
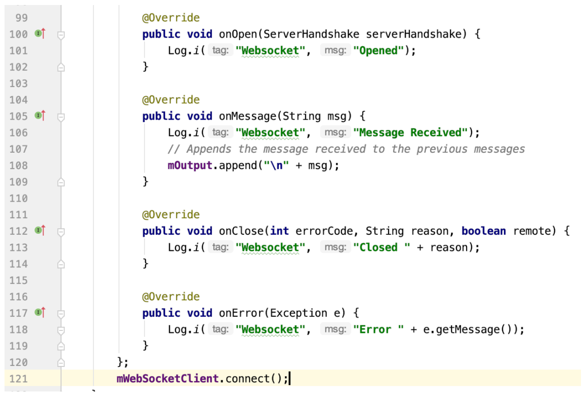
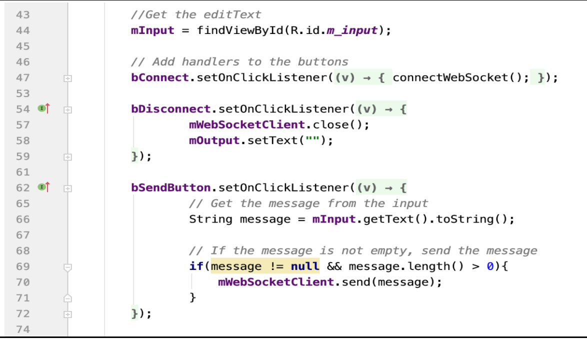
6 Android Code

6.1 About Websocket in Android

Here we use an implementation of the RFC 6455 protocol directly (See https://tools.ietf.org/html/rfc6455). The implementation used is Java-WebSocket (see https://github.com/TooTallNate/Java-WebSocket).

These examples can be found at https://git.linux.iastate.edu/cs309/tutorials.git

6.2 Example

1) Add dependencies in applications build.gradle See the websocket dependency.	
2) Add the code for connecting to the server (See MainActivity.java) See how th URI is specified and how the websocketclient object is created.	
3) handle the different operations (open, close, message, error) Methods have to be onOpen, onMessage, onClose, and onError. NOTE that on line 121 – the websocket object is connected to server.	
4) send message. the websocket object's send() method is used on line 70 to send message back to the server. You can run and test this android app!	

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation "org.java-websocket:Java-WebSocket:1.4.1"
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

```
83      private void connectWebSocket() {
84          URI uri;
85          try {
86              /*
87               * To test the clientside without the backend, simply connect to an echo server suc
88               *  "ws://echo.websocket.org"
89               */
90              //uri = new URI("ws://10.0.2.2:8080/example"); // 10.0.2.2 = localhost
91              uri = new URI("ws://echo.websocket.org");
92          } catch (URISyntaxException e) {
93              e.printStackTrace();
94              return;
95          }
96
97          mWebSocketClient = new WebSocketClient(uri) {
```

```
99              @Override
100             public void onOpen(ServerHandshake serverHandshake) {
101                 Log.i("Websocket", "Opened");
102             }
103
104             @Override
105             public void onMessage(String msg) {
106                 Log.i("Websocket", "Message Received");
107                 // Appends the message received to the previous messages
108                 mOutput.append("\n" + msg);
109             }
110
111             @Override
112             public void onClose(int errorCode, String reason, boolean remote) {
113                 Log.i("Websocket", "Closed " + reason);
114             }
115
116             @Override
117             public void onError(Exception e) {
118                 Log.i("Websocket", "Error " + e.getMessage());
119             }
120         };
121         mWebSocketClient.connect();
```

```
43          //Get the editText
44          mInput = findViewById(R.id.m_input);
45
46          // Add handlers to the buttons
47          bConnect.setOnClickListener((v) → { connectWebSocket(); });
53
54          bDisconnect.setOnClickListener((v) → {
57                  mWebSocketClient.close();
58                  mOutput.setText("");
59          });
61
62          bSendButton.setOnClickListener((v) → {
65                  // Get the message from the input
66                  String message = mInput.getText().toString();
67
68                  // If the message is not empty, send the message
69                  if(message != null && message.length() > 0){
70                      mWebSocketClient.send(message);
71                  }
72          });
74
```