Example of WebSocket/STOMP using Java

WebSockets enable 2-way, duplex communication between client and server. All major browsers and all major Java servers like Jetty 9, Tomcat 7, etc. support websockets. The present example has been done with Tomcat 8 as a server and a Tyrus based simple Java client.

# Simple WebSocket Server

We will use the WebSocket API to create a simple Server.

### Configuration: pom.xml

The pom would be the same as a typical JEE pom. Just put the following dependency:

<dependency>

<groupId>javax.websocket</groupId>

<artifactId>javax.websocket-api</artifactId>

<version>1.1</version>

<scope>provided</scope>

</dependency>

### Server Code

@ServerEndpoint(value = "/hello-stomp")

**public** **class** SimpleServerEndpoint {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(SimpleServerEndpoint.**class**);

**private** **static** **final** String ***DATE\_FORMAT*** = "dd-MM-yyyy HH:mm:ss";

@OnMessage

**public** String onMessage(String message, Session session) {

***LOGGER***.info("Message from client: `{}`", message);

String serverMessage = **new** StringBuilder(100).append("Message processed by SimpleServer at [").append(**new** SimpleDateFormat(***DATE\_FORMAT***).format(**new** Date())).append("]").append(message)

.toString();

**return** serverMessage;

}

}

# Simple WebSocket Java Client

### Configuration: pom.xml

The pom would be a simple one, just add these following dependencies:

<dependency>

<groupId>javax.websocket</groupId>

<artifactId>javax.websocket-api</artifactId>

<version>1.1</version>

</dependency>

<dependency>

<groupId>org.glassfish.tyrus.bundles</groupId>

<artifactId>tyrus-standalone-client</artifactId>

<version>1.12</version>

</dependency>

### Client Code

Define an Endpoint:

**public** **class** SimpleClientEndpoint **extends** Endpoint {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(SimpleClientEndpoint.**class**);

@Override

**public** **void** onOpen(Session session, EndpointConfig config) {

***LOGGER***.info("session opened");

session.addMessageHandler(String.**class**, **new** Whole<String>() {

@Override

**public** **void** onMessage(String text) {

***LOGGER***.info("recieved message from server: `{}`", text);

}

});

String message = "Hello from client";

***LOGGER***.info("trying to send message `{}` to server...", message);

**try** {

session.getBasicRemote().sendText(message);

***LOGGER***.info("message sent successfully");

} **catch** (IOException e) {

***LOGGER***.error("error sending message to server", e);

}

}

}

Then invoke this:

**public** **class** SimpleClient {

**public** **static** **void** main(String[] args) **throws** InterruptedException,

URISyntaxException, DeploymentException, IOException {

ClientEndpointConfig cec = ClientEndpointConfig.Builder.*create*()

.build();

ClientManager client = ClientManager.*createClient*();

client.connectToServer(SimpleClientEndpoint.**class**, cec, **new** URI(

"ws://localhost:8080/stomp-server/hello-stomp"));

Thread.*sleep*(10\_000);

}

}

Note that there is a 2-way communication from server/client. This is handled in the MessageHandler defined in SimpleClientEndpoint:

session.addMessageHandler(String.**class**, **new** Whole<String>() {

@Override

**public** **void** onMessage(String text) {

***LOGGER***.info("recieved message from server: `{}`", text);

}

});

# Spring WebSocket Handler Server

Though the above example is simple and good for starters, this is not practical. For one, it does not integrate with Spring. Lets build something more closer to the real world.

### Overview

The client attempts to invoke the BankDetailsService asynchronously, through websockets. For this, it first initiates the request, sending the server the sort-order. The server then triggers a dao-call. Each time when a new row is read off the ResultSet, it is immediately sent to the client. After all the records have been read, the session is closed. Note that the messages are binary. Json strings are gzipped and then sent across the wire to save bandwidth.

We will use the Spring websocket module for integrating WebSockets with Spring.

### Configuration: pom.xml

We would need the spring websocket module:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-websocket</artifactId>

</dependency>

### Spring Configuration

@Configuration

@EnableWebSocket

**public** **class** WebSocketHandlerConfig **implements** WebSocketConfigurer {

@Autowired

**private** BankDetailService bankDetailService;

@Override

**public** **void** registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(**new** ExceptionWebSocketHandlerDecorator(**new** BankDetailsWebSocketHandler(bankDetailService)), "/streaming-bank-details").setAllowedOrigins("\*");

}

@Bean

**public** ServletServerContainerFactoryBean createWebSocketContainer() {

ServletServerContainerFactoryBean container = **new** ServletServerContainerFactoryBean();

container.setMaxTextMessageBufferSize(100\_000);

container.setMaxBinaryMessageBufferSize(100\_000);

container.setAsyncSendTimeout(100\_000\_000\_000\_000L);

container.setMaxSessionIdleTimeout(100\_000\_000\_000\_000L);

**return** container;

}

}

The ExceptionWebSocketHandlerDecorator is a decorator for handling any runtime exception gracefully by closing the connection.

### Server Code

This is how the BankDetailsWebSocketHandler looks like:

**public** **class** BankDetailsWebSocketHandler **extends** TextWebSocketHandler {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(BankDetailsWebSocketHandler.**class**);

**private** **final** BankDetailService bankDetailService;

**public** BankDetailsWebSocketHandler(BankDetailService bankDetailService) {

**this**.bankDetailService = bankDetailService;

}

@Override

**protected** **void** handleTextMessage(WebSocketSession session, TextMessage message) {

***LOGGER***.info("recieved message from client");

BankDetailSortOrder group = BankDetailSortOrder.*valueOf*(message.getPayload().toUpperCase());

bankDetailService.getBankDetailsAsync(group, **new** DataListenerForStompHandlerImpl(session));

}

}

Note that the actual action happens in DataListenerForStompHandlerImpl:

**public** **class** DataListenerForStompHandlerImpl **implements** DataListener {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(DataListenerForStompHandlerImpl.**class**);

**private** **final** ObjectMapper mapper = **new** ObjectMapper();

**private** **final** WebSocketSession session;

**public** DataListenerForStompHandlerImpl(WebSocketSession session) {

**this**.session = session;

}

@Override

**public** **void** sendMessageToClient(BankDetail bankDetail) {

**try** {

String jsonString = mapper.writeValueAsString(bankDetail);

***LOGGER***.info("trying to apply gzip on message `{}` before sending to the client...", jsonString);

ByteArrayOutputStream buffer = **new** ByteArrayOutputStream(10\_000);

GZIPOutputStream gzipOutput = **new** GZIPOutputStream(buffer);

gzipOutput.write(jsonString.getBytes());

gzipOutput.flush();

gzipOutput.close();

WebSocketMessage<?> messageToBeSent = **new** BinaryMessage(buffer.toByteArray());

session.sendMessage(messageToBeSent);

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** endOfMessages() {

**try** {

session.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

}

This class is passed to the Dao, and after each record is read from the ResultSet, it is sent to the client.

### Client Code

Let us take a look at the client. We are still using a standard WebSocket client, not the Spring stomp client.

Like the 1st example, we have defined our Endpoint as below:

**class** StompClientEndpoint **extends** Endpoint {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(StompClientEndpoint.**class**);

**private** **final** CountDownLatch waitTillConnectionClosed;

**private** **final** SeekableByteChannel outputFileChannel;

StompClientEndpoint(CountDownLatch waitTillConnectionClosed, SeekableByteChannel outputFileChannel) {

**this**.waitTillConnectionClosed = waitTillConnectionClosed;

**this**.outputFileChannel = outputFileChannel;

}

@Override

**public** **void** onOpen(Session session, EndpointConfig config) {

session.addMessageHandler(**new** Whole<InputStream>() {

@Override

**public** **void** onMessage(InputStream inputStream) {

***LOGGER***.info("recieved binary message from server");

**try** (ReadableByteChannel reportByteChannel = Channels.*newChannel*(**new** GZIPInputStream(inputStream));) {

ByteBuffer byteBuffer = ByteBuffer.*allocate*(1000);

**while** (reportByteChannel.read(byteBuffer) > 0) {

byteBuffer.flip();

outputFileChannel.write(byteBuffer);

byteBuffer.compact();

}

outputFileChannel.write(StandardCharsets.***US\_ASCII***.encode("\n\n\n"));

} **catch** (IOException e) {

***LOGGER***.error("error", e);

}

}

});

**try** {

session.getBasicRemote().sendText("MARITAL\_STATUS");

} **catch** (IOException e) {

***LOGGER***.error("error sending message to client", e);

}

}

@Override

**public** **void** onClose(Session session, CloseReason closeReason) {

***LOGGER***.warn("Connection closed with reason:{} ", closeReason);

waitTillConnectionClosed.countDown();

}

}

All subsequent messages from the server would flow into the MessageHandler defined in the above class. The above is now invoked here:

**class** SingleStompConsumer **implements** Runnable {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(SingleStompConsumer.**class**);

**private** **static** **final** String ***STOMP\_URI*** = "ws://localhost:8080/stomp-server/streaming-bank-details";

**private** **final** String id;

**public** SingleStompConsumer(String id) {

**this**.id = id;

}

@Override

**public** **void** run() {

CountDownLatch waitTillConnectionClosed = **new** CountDownLatch(1);

ClientManager client = ClientManager.*createClient*();

Path outputFilePath = Paths.*get*("d:", "temp", "output\_" + id + ".json");

**try** (SeekableByteChannel outputFileChannel = Files.*newByteChannel*(outputFilePath, StandardOpenOption.***CREATE***, StandardOpenOption.***TRUNCATE\_EXISTING***, StandardOpenOption.***WRITE***);) {

client.connectToServer(**new** StompClientEndpoint(waitTillConnectionClosed, outputFileChannel), ClientEndpointConfig.Builder.*create*().build(), **new** URI(***STOMP\_URI***));

waitTillConnectionClosed.await();

} **catch** (InterruptedException | IOException | DeploymentException | URISyntaxException e) {

***LOGGER***.error("error", e);

}

***LOGGER***.info("All output written in the file: {}", outputFilePath);

}

}

# Source Code

<https://github.com/paawak/blog/tree/master/code/stomp>

# References

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html>

<https://tyrus.java.net/documentation/1.12/index/getting-started.html>

<https://www.websocket.org/echo.html>