Implementing a Spring Websocket Server and Client

Table of Contents

This article guides you through the implementation of a WebSocket server and client based on the Spring Framework. It demonstrates full duplex communication and how the server can push messages to the client. You will learn a little about how WebSockets work, as well as the STOMP messaging format used for communication between the server and the client.

To get the most out of his article, you need good knowledge of Java, along with some exposure to the Spring Framework, especially Spring MVC. The source code for the server that we are oing to implement is available on GitHub (https://github.com/jaysridhar/spring-websocket-server).

Cliff's Notes On Websockets, STOMP, and Spring

To get you up and running with Websocket on the JVM, here's a quick intro to what we'll need later in the article.

Introduction to WebSockets

WebSocket (https://en.wikipedia.org/wiki/WebSocket) is a full-duplex communications protocol layered over TCP. It is typically used for interactive communication between a user's browser and a back-end server. An example would be a chat server with real-time communications between the server and the connected clients. Another example would be a Stock Trading application where the server sends stock price variations to subscribed clients without an explicit client request.

A WebSocket is a communication channel which uses TCP as the underlying protocol. It is initiated by the client sending a HTTP request to the server requesting a connection upgrade to WebSocket. If the server supports WebSockets, the client request is granted and a WebSocket connection is established between the two parties. After this connection is established, all communication happens over the WebSocket and the HTTP protocol is no longer used.

This article (https://www.sitepoint.com/websockets-stable-and-ready-for-developers/) does a very good job of explaining the details of how WebSockets work.

STOMP over WebSocket

The WebSocket communication protocol itself does not mandate any particular messaging format. It is up to the applications to agree upon the format of the messages exchanged. This format is referred to as the subprotocol. (Kinda similar to how the web browser and web server have agreed to using the HTTP protocol over TCP sockets.)

One commonly used format is the STOMP protocol (https://stomp.github.io/stomp-specification-1.2.html) (Streaming Text Oriented Message Protocol) used for general purpose messaging. Various message oriented middle-ware (MOM) (https://en.wikipedia.org/wiki/Message-oriented_middleware) systems such as Apache ActiveMQ (http://activemq.apache.org/), HornetQ (http://hornetq.jboss.org/) and RabbitMQ (https://www.rabbitmq.com/) support STOMP.

The Spring Framework (https://spring.io/) implements WebSocket communication with STOMP as the messaging protocol.

Spring Support for WebSockets

Spring Framework provides support for WebSocket messaging in web applications. Spring version 4 (http://docs.spring.io/spring/docs/4.3.6.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/) includes a new module `spring-websocket` (https://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html), which is used to add WebSocket support to the server and the client.

The Spring Framework Guide includes a detailed HOWTO demonstrating a sample server implementation (https://spring.io/guides/gs/messaging-stomp-websocket/). However, it is not necessary to have read that guide; we cover all the details required for understanding the implementation in this article.

One shortcoming of the Spring Framework Guide is that no information is presented on implementing a Java/Spring Client to communicate with the server. We address that issue in this article. As you will see, the implementation requires correct incantations to get it working just right.

Spring Chat Server Implementation

With the prerequisites out of the way, it is time to implement the chat server. Here's a rundown of how it works:

- The Spring chat server creates an HTTP endpoint (`/chat`), which is used for establishing the WebSocket channel.
- Once the channel is established, the server and client exchange messages over the channel.
- While the server listens to and responds to client messages, the client can also register a callback for "push" messages from the server. This allows the server to send notifications to the client as and when required without an explicit client request.

The message "push" from the server to the client is what makes the WebSocket communication different from normal HTTP interaction.

Build Configuration

The Maven dependencies required for the server are shown below. The `spring-boot-starter-websocket` dependency includes the required libraries for server side implementation of WebSockets.

The remaining `org.webjars` dependencies are jQuery, Bootstrap and client-side libraries for WebSocket and STOMP messaging required by the HTML/Javascript chat client. These Javascript dependencies are not used by the Java WebSocket server implementation. They are required for the front-end client which is completely separate from the back-end application. These dependencies can safely be removed from the POM if the HTML/Javascript client is not used or packaged separately from the Spring Application.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator</artifactId>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>sockjs-client</artifactId>
  <version>1.0.2</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>stomp-websocket</artifactId>
  <version>2.3.3</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.7</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.1.0</version>
</dependency>
```

Message Controller

In Spring MVC, decorating a class with the `@Controller` annotation marks the class as a web controller (https://spring.io/guides/gs/serving-web-content/). This works in conjuntion with the `@RequestMapping` decoration to mark a method as an HTTP endpoint. In a similar way, the annotation `@MessageMapping` is available with Spring Framework Messaging to register the method as a message listener.

For our simple chat server, the response echoes the message received from client and adds a couple of fields. For a real-world application, this method is where the business logic is implemented. Here is the implementation of the method:

```
@MessageMapping("/chat/{topic}")
@SendTo("/topic/messages")
public OutputMessage send(
        @DestinationVariable("topic") String topic, Message message)
        throws Exception {
    return new OutputMessage(message.getFrom(), message.getText(), topic);
}
```

The Message Controller method accepts a POJO (a Plain Old Java Object) into which a message is de-serialized from JSON. The `@DestinationVariable` annotation is used to capture the template variable `topic` from the destination. The following `Message` class is defined by the server to read messages from the client.

```
(/)
public class Message {

    private String from;
    private String text;

    // adding getters and setters here
}
```

To send responses back to the client, the message controller method uses a `@SendTo` annotation specifying the client-side queue to which the response is to be sent. The value returned from the method is serialized to JSON and sent to the specified destination. In our case, we define the response message as follows. Note that the input message field `text` is just copied to the response message field `message`. In your application, you can do whatever additional processing required and also include more fields in the request or response classes.

```
public class OutputMessage {

    private String from;
    private String message;
    private String topic;
    private Date time = new Date();

    // add getters and setters here
}
```

Configuring Spring

The application is configured for WebSockets using a `@Configuration` class which extends the `AbstractWebSocketMessageBrokerConfigurer` and provides implementation for the following methods.

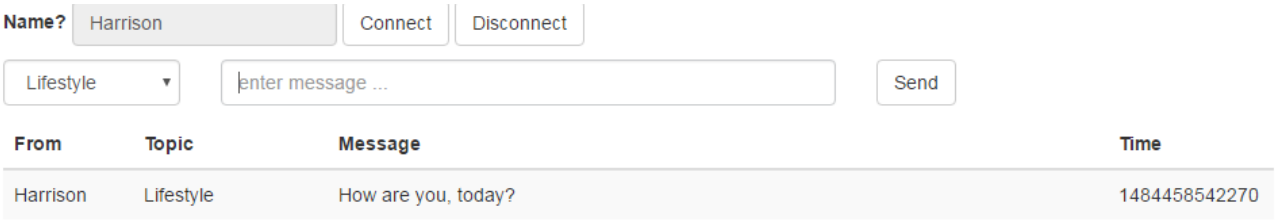
```
@Override
public void configureMessageBroker(MessageBrokerRegistry config) {
    config.enableSimpleBroker("/topic");
    config.setApplicationDestinationPrefixes("/app");
}

@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/chat").setAllowedOrigins("*").withSockJS();
}
```

Note the following points:

- The application destination prefix is set to "`/app`". This means WebSocket mappings are prefixed with "`/app`" to obtain the real value.
- The endpoint "`/chat`" is registered for starting the WebSocket protocol handshake.
- The method `setAllowedOrigins("*")` is used to CORS-enable the server so cross-origin (https://www.w3.org/TR/cors/) clients can connect to the server. If your requirements do not need cross-origin client connection, you can remove this invocation.

Web Browser Client

A sample web application written in HTML and Javascript is included [in the code (https://github.com/jaysridhar/spring-websocket-server)](https://github.com/jaysridhar/spring-websocket-server) – it implements a simple front-end for the chat application. A snapshot of the application is shown below.



Connecting and Registering Push Callback

The application uses the [SocksJS (https://github.com/sockjs/sockjs-client)](https://github.com/sockjs/sockjs-client) library for dealing with the WebSocket connection, and the [stomp-websocket (http://jmesnil.net/stomp-websocket/doc/)](http://jmesnil.net/stomp-websocket/doc/) library for the STOMP support. The connection code below creates a WebSocket channel and uses it to create the STOMP client. Once the WebSocket is connected, a callback is registered for push messages from the server. The message is de-serialized from JSON and displayed to the user.

```
var wsocket = new SockJS('/chat');
var client = Stomp.over(wsocket);
client.connect({}, function(frame) {
    client.subscribe('/topic/messages', function (message) {
        showMessage(JSON.parse(message.body));
    });
});
```

Sending Messages to the Server

The client sends messages to the server using `send()`. The message is in JSON and in the same format as expected by the message controller at the server end.

```
    (/)
client.send("/app/chat/" + topic, {}, JSON.stringify({
    from: $("#from").val(),
    text: $('#text').val(),
}));
```

Building and Running the Application

The server is implemented as a Spring Boot application and includes an embedded web server. Build the application using Maven.

```
mvn clean package
```

Run the server specifying the port the web server should listen on if required.

```
java -Dserver.port=9090 -jar target/chat-server-0.1.0.jar
```

Once the server is started, open the Chat application at `http://<hostname>:9090/`

Java/Spring Chat Client

While an HTML/Javascript client is useful for demonstrating WebSocket usage in the browser, a Java client is useful for interacting with the server from within an application. Maybe you have a Forex Trading application which needs to report updated prices to all connected applications. Or maybe you want to subscribe to published messages to get update notifications. Whatever the use case, it is useful to learn how to interact with a WebSocket server from within a Java application.

Set up the underlying transports and create a SockJS client. The SockJS client module within Spring implements the sockjs protocol (https://github.com/sockjs/sockjs-protocol) (here, the JS is just part of the name – no JavaScript involved) as a fallback option in case WebSocket is not supported on the (browser) client. This allows applications to use WebSockets but to fall back to other alternatives when necessary at run time – without needing to change application code. An alternative transport is Ajax/XHR streaming (https://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html#websocket-fallback-xhr-vs-iframe) used by Internet Explorer 8 and 9.

```
WebSocketClient simpleWebSocketClient = new StandardWebSocketClient();
List<Transport> transports = new ArrayList<>(1);
transports.add(new WebSocketTransport(simpleWebSocketClient));
SockJsClient sockJsClient = new SockJsClient(transports);
```

The `SockJsClient` is used to create a STOMP client which supports the messaging protocol used.

```
WebSocketStompClient stompClient = new WebSocketStompClient(sockJsClient);
```

And of course we need to map JSON to POJOs and back.

```
stompClient.setMessageConverter(new MappingJackson2MessageConverter());
```

Connect to the server and once the connection is made, a message handler is registered to listen for push messages from the server.

```
String url = "ws://localhost:9090/chat";
StompSessionHandler sessionHandler = new MyStompSessionHandler();
StompSession session = stompClient.connect(url, sessionHandler).get();
```

Subscribe to messages from the server as follows:

```
session.subscribe(topic, new StompFrameHandler() {
    @Override
    public Type getPayloadType(StompHeaders headers) {
        return ServerMessage.class;
    }

    @Override
    public void handleFrame(StompHeaders headers,Object payload) {
        System.err.println(payload.toString());
    }
});
```

Send messages to the server as follows:

```
ClientMessage msg = new ClientMessage(userId, line);
session.send("/app/chat/java", msg);
```

The complete source code for the client can be downloaded here (https://github.com/jaysridhar/spring-websocket-client).

Summary

Let's review.

We started with a brief introduction to WebSockets and STOMP. STOMP is a messaging sub-protocol running over WebSockets which provides facilities such as topic subscriptions.

The Spring Framework (used for the WebSocket server) provides modules for WebSocket both for the server as well as the client. The modules are easily configured with annotations defined by Spring. Spring also handles serialization and deserialization of messages to and from POJOs.

The Web browser client shows the interaction with the server and how push messages from the server can be received and processed. Finally the Spring Client illustrates how to communicate with the WebSocket server and subscribe to message topics.

Meet the author

Jay Sridhar (https://www.sitepoint.com/author/jsridhar/) (https://twitter.com/jay_sridhar) (https://www.reddit.com/user/jaysridhar/) (https://github.com/jaysridhar)

Software Architect and Founder of Novixys Software (http://www.novixys.com). Full Stack Developer with over 15 years experience. Experienced within Financial, Security, Publishing and Pharamaceutical industries. Currently working with Big Data.

- Forums (/community/)
- References (/html-css/css/)
- Advertise (/advertise/)

Connect

(https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (/versioning/) (https://www.sitepoint.com/feed/) (https://plus.google.com/+sitepoint)