Building a chat application with Spring Boot and WebSocket

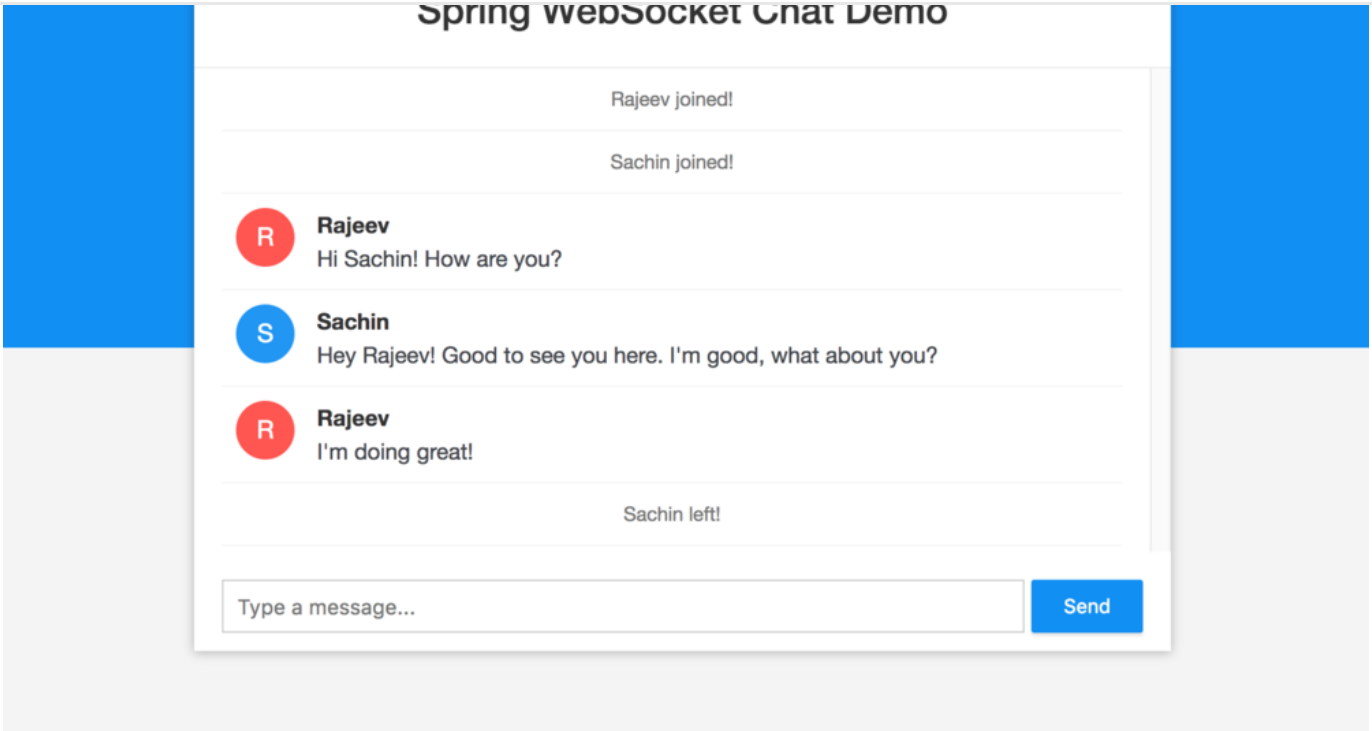
Rajeev Kumar Singh • Spring Boot • Jul 27, 2017 • 13 mins read

In this article, you'll learn how to use WebSocket API with Spring Boot and build a simple group chat application at the end.

> **You can explore the live demo of the application by clicking this link - https://spring-ws-chat.herokuapp.com/.**
>
> You can just type in your name and start chatting with others. If no one is available in the chat room, then you can open the app in two tabs, login with different usernames and start sending messages.

Following is a screen shot of the chat application that we'll be building in this tutorial -

CALLICODER



WebSocket is a communication protocol that makes it possible to establish a two-way communication channel between a server and a client.

WebSocket works by first establishing a regular HTTP connection with the server and then upgrading it to a bidirectional websocket connection by sending an `Upgrade` header.

WebSocket is supported in most modern web browsers and for browsers that don't support it, we have libraries that provide fallbacks to other techniques like comet and long-polling.

Well, now that we know what websocket is and how it works, let's jump into the implementation of our chat application.

Creating the Application

Let's use Spring Boot CLI to bootstrap our application. Checkout the Official Spring Boot documentation for instructions on how to install Spring Boot CLI.

CALLICODER

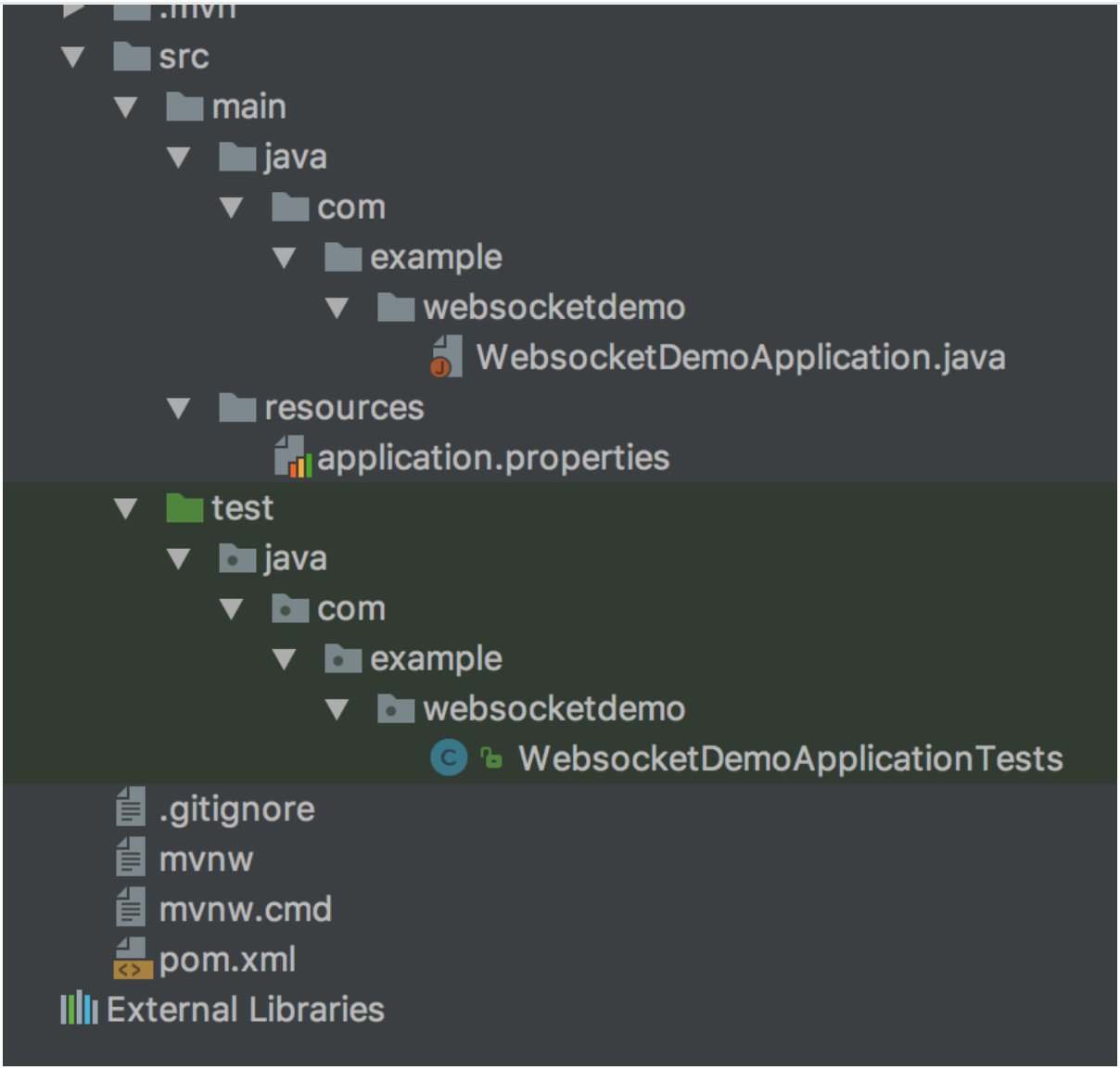
```
$ spring init --name=websocket-demo -dependencies=websocket websocket
```

If you don't want to install Spring Boot CLI, No worries, You can use Spring Initializer web tool to generate the project. Follow the steps below to generate the project using Spring Initializer -

1. Go to http://start.spring.io/.
2. Enter Artifact's value as **websocket-demo**.
3. Add **Websocket** in the dependencies section.
4. Click **Generate Project** to download the project.
5. Extract the downloaded zip file.

After generating the project, import it into your favorite IDE. The project's directory structure should look like this -

CALLICODER



WebSocket Configuration

The first step is to configure the websocket endpoint and message broker. Create a new package `config` inside `com.example.websocketdemo` package, then create a new class `WebSocketConfig` inside `config` package with the following contents -

```
package com.example.websocketdemo.config;
```

CALLICODER

```
import org.springframework.web.socket.config.annotation.*;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerCo

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry
        registry.addEndpoint("/ws").withSockJS();
    }

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

The `@EnableWebSocketMessageBroker` is used to enable our WebSocket server. We extend `AbstractWebSocketMessageBrokerConfigurer` and override some of its methods to configure the websocket connection.

In the first method, we register a websocket endpoint that the clients will use to connect to our websocket server.

Notice the use of `withSockJS()` with the endpoint configuration. **SockJS** is used to enable fallback options for browsers that don't support websocket.

You might have noticed the word **STOMP** in the method name. These methods come from Spring frameworks STOMP implementation. STOMP stands for Simple Text Oriented Messaging Protocol. It is a messaging protocol that defines the format and rules for data exchange.

subscribed to a particular topic, or how to send a message to a particular user. We need STOMP for these functionalities.

In the second method, we're configuring a message broker that will be used to route messages from one client to another.

The first line defines that the messages whose destination starts with "/app" should be routed to message-handling methods (we'll define these methods shortly).

And, the second line defines that the messages whose destination starts with "/topic" should be routed to the message broker. Message broker broadcasts messages to all the connected clients who are subscribed to a particular topic.

In the above example, We have enabled a simple in-memory message broker. But you're free to use any other full-featured message broker like RabbitMQ or ActiveMQ.

Creating the ChatMessage model

`ChatMessage` model is the message payload that will be exchanged between the clients and the server. Create a new package `model` inside `com.example.websocketdemo` package, and then create the `ChatMessage` class inside `model` package with the following contents -

```
package com.example.websocketdemo.model;

public class ChatMessage {
    private MessageType type;
    private String content;
    private String sender;
```

```
        JOIN,
        LEAVE
    }

    public MessageType getType() {
        return type;
    }

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

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getSender() {
        return sender;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }
}
```

Creating the Controller for sending and receiving messages

CALLICODER

to others.

Create a new package `controller` inside the base package and then create the ChatController class with the following contents -

```
package com.example.websocketdemo.controller;

import com.example.websocketdemo.model.ChatMessage;
import org.springframework.messaging.handler.annotation.MessageMappin
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.messaging.simp.SimpMessageHeaderAccessor;
import org.springframework.stereotype.Controller;

@Controller
public class ChatController {

    @MessageMapping("/chat.sendMessage")
    @SendTo("/topic/public")
    public ChatMessage sendMessage(@Payload ChatMessage chatMessage)
        return chatMessage;
    }

    @MessageMapping("/chat.addUser")
    @SendTo("/topic/public")
    public ChatMessage addUser(@Payload ChatMessage chatMessage,
                               SimpMessageHeaderAccessor headerAccess
        // Add username in web socket session
        headerAccessor.getSessionAttributes().put("username", chatMes
        return chatMessage;
    }
```

If you recall from the websocket configuration, all the messages sent from clients with a destination starting with `/app` will be routed to these message handling methods annotated with `@MessageMapping`.

For example, a message with destination `/app/chat.sendMessage` will be routed to the `sendMessage()` method, and a message with destination `/app/chat.addUser` will be routed to the `addUser()` method.

Adding WebSocket Event listeners

We'll use event listeners to listen for socket connect and disconnect events so that we can log these events and also broadcast them when a user joins or leaves the chat room -

```
package com.example.websocketdemo.controller;

import com.example.websocketdemo.model.ChatMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.event.EventListener;
import org.springframework.messaging.simp.SimpMessageSendingOperation
import org.springframework.messaging.simp.stomp.StompHeaderAccessor;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.messaging.SessionConnectedEvent
import org.springframework.web.socket.messaging.SessionDisconnectEven

@Component
public class WebSocketEventListener {
```

CALLICODER

```
    @Autowired
    private SimpMessageSendingOperations messagingTemplate;

    @EventListener
    public void handleWebSocketConnectListener(SessionConnectedEvent
        logger.info("Received a new web socket connection");
    }

    @EventListener
    public void handleWebSocketDisconnectListener(SessionDisconnectEv
        StompHeaderAccessor headerAccessor = StompHeaderAccessor.wrap

        String username = (String) headerAccessor.getSessionAttribute
        if(username != null) {
            logger.info("User Disconnected : " + username);

            ChatMessage chatMessage = new ChatMessage();
            chatMessage.setType(ChatMessage.MessageType.LEAVE);
            chatMessage.setSender(username);

            messagingTemplate.convertAndSend("/topic/public", chatMes
        }
    }
}
```

We're already broadcasting user join event in the `addUser()` method defined inside `ChatController`. So, we don't need to do anything in the SessionConnected event.

In the SessionDisconnect event, we've written code to extract the user's name from the websocket session and broadcast a user leave event to all the connected clients.

CALLICODER

Create the following folders and files inside `src/main/resources` directory -

```
static
   └── css
        └── main.css
   └── js
        └── main.js
        └── sockjs.min.js
        └── stomp.min.js
   └── index.html
```

The `src/main/resources/static` folder is the default location for static files in Spring Boot.

1. Adding SockJS and STOMP JS libraries.

SockJS is a WebSocket client that tries to use native WebSockets and provides intelligent fallback options for older browsers that don't support WebSocket. STOMP JS is the stomp client for javascript.

You can download `sockjs.min.js` from sockjs-client github page, and `stomp.min.js` from stomp-websocket github page.

Once downloaded, add them to `src/main/resources/static/js` folder.

2. Creating the HTML - `index.html`

I won't go into details of the html file. It is short and simple. Just add the following code to the `index.html` file -

CALLICODER

```
<html>
  <head>
      <meta name="viewport" content="width=device-width, initial-scal
      <title>Spring Boot WebSocket Chat Application</title>
      <link rel="stylesheet" href="/css/main.css" />
  </head>
  <body>
    <noscript>
      <h2>Sorry! Your browser doesn't support Javascript</h2>
    </noscript>

    <div id="username-page">
        <div class="username-page-container">
            <h1 class="title">Type your username</h1>
            <form id="usernameForm" name="usernameForm">
                <div class="form-group">
                    <input type="text" id="name" placeholder="Usernam
                </div>
                <div class="form-group">
                    <button type="submit" class="accent username-subm
                </div>
            </form>
        </div>
    </div>

    <div id="chat-page" class="hidden">
        <div class="chat-container">
            <div class="chat-header">
                <h2>Spring WebSocket Chat Demo</h2>
            </div>
            <div class="connecting">
                Connecting...
            </div>
```

```
                </ul>
                <form id="messageForm" name="messageForm" nameForm="messa
                    <div class="form-group">
                        <div class="input-group clearfix">
                            <input type="text" id="message" placeholder="
                            <button type="submit" class="primary">Send</b
                        </div>
                    </div>
                </form>
            </div>
        </div>

        <script src="/js/sockjs.min.js"></script>
        <script src="/js/stomp.min.js"></script>
        <script src="/js/main.js"></script>
    </body>
</html>
```

3. JavaScript - `main.js`

Let's now add the javascript required for connecting to the websocket endpoint and sending & receiving messages. First, add the following code to the `main.js` file, and then we'll explore some of the important methods in this file -

```
'use strict';

var usernamePage = document.querySelector('#username-page');
var chatPage = document.querySelector('#chat-page');
var usernameForm = document.querySelector('#usernameForm');
var messageForm = document.querySelector('#messageForm');
var messageInput = document.querySelector('#message');
```

CALLICODER

```
var stompClient = null;
var username = null;

var colors = [
    '#2196F3', '#32c787', '#00BCD4', '#ff5652',
    '#ffc107', '#ff85af', '#FF9800', '#39bbb0'
];

function connect(event) {
    username = document.querySelector('#name').value.trim();

    if(username) {
        usernamePage.classList.add('hidden');
        chatPage.classList.remove('hidden');

        var socket = new SockJS('/ws');
        stompClient = Stomp.over(socket);

        stompClient.connect({}, onConnected, onError);
    }
    event.preventDefault();
}


function onConnected() {
    // Subscribe to the Public Topic
    stompClient.subscribe('/topic/public', onMessageReceived);

    // Tell your username to the server
    stompClient.send("/app/chat.addUser",
        {},
        JSON.stringify({sender: username, type: 'JOIN'})
```

CALLICODER

```
    connectingElement.classList.add('hidden');
}


function onError(error) {
    connectingElement.textContent = 'Could not connect to WebSocket s
    connectingElement.style.color = 'red';
}


function sendMessage(event) {
    var messageContent = messageInput.value.trim();
    if(messageContent && stompClient) {
        var chatMessage = {
            sender: username,
            content: messageInput.value,
            type: 'CHAT'
        };
        stompClient.send("/app/chat.sendMessage", {}, JSON.stringify(
        messageInput.value = '';
    }
    event.preventDefault();
}


function onMessageReceived(payload) {
    var message = JSON.parse(payload.body);

    var messageElement = document.createElement('li');

    if(message.type === 'JOIN') {
        messageElement.classList.add('event-message');
        message.content = message.sender + ' joined!';
```

CALLICODER

```
            message.content = message.sender + ' left!';
        } else {
            messageElement.classList.add('chat-message');

            var avatarElement = document.createElement('i');
            var avatarText = document.createTextNode(message.sender[0]);
            avatarElement.appendChild(avatarText);
            avatarElement.style['background-color'] = getAvatarColor(mess

            messageElement.appendChild(avatarElement);

            var usernameElement = document.createElement('span');
            var usernameText = document.createTextNode(message.sender);
            usernameElement.appendChild(usernameText);
            messageElement.appendChild(usernameElement);
        }

        var textElement = document.createElement('p');
        var messageText = document.createTextNode(message.content);
        textElement.appendChild(messageText);

        messageElement.appendChild(textElement);

        messageArea.appendChild(messageElement);
        messageArea.scrollTop = messageArea.scrollHeight;
    }


    function getAvatarColor(messageSender) {
        var hash = 0;
        for (var i = 0; i < messageSender.length; i++) {
            hash = 31 * hash + messageSender.charCodeAt(i);
        }
```

```
}

usernameForm.addEventListener('submit', connect, true)
messageForm.addEventListener('submit', sendMessage, true)
```

The `connect()` function uses `SockJS` and `stomp` client to connect to the `/ws` endpoint that we configured in Spring Boot.

Upon successful connection, the client subscribes to `/topic/public` destination and tells the user's name to the server by sending a message to the `/app/chat.addUser` destination.

The `stompClient.subscribe()` function takes a callback method which is called whenever a message arrives on the subscribed topic.

Rest of the code is used to display and format the messages on the screen.

4. Adding CSS - `main.css`

Finally, Add the following styles to the `main.css` file -

```
* {
    -webkit-box-sizing: border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
}

html,body {
    height: 100%;
    overflow: hidden;
}
```

CALLICODER

```
    padding: 0;
    font-weight: 400;
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
    font-size: 1rem;
    line-height: 1.58;
    color: #333;
    background-color: #f4f4f4;
    height: 100%;
}

body:before {
    height: 50%;
    width: 100%;
    position: absolute;
    top: 0;
    left: 0;
    background: #128ff2;
    content: "";
    z-index: 0;
}

.clearfix:after {
    display: block;
    content: "";
    clear: both;
}

.hidden {
    display: none;
}

.form-control {
    width: 100%;
```

```
    border: 1px solid #c8c8c8;
}

.form-group {
    margin-bottom: 15px;
}

input {
    padding-left: 10px;
    outline: none;
}

h1, h2, h3, h4, h5, h6 {
    margin-top: 20px;
    margin-bottom: 20px;
}

h1 {
    font-size: 1.7em;
}

a {
    color: #128ff2;
}

button {
    box-shadow: none;
    border: 1px solid transparent;
    font-size: 14px;
    outline: none;
    line-height: 100%;
    white-space: nowrap;
    vertical-align: middle;
```

CALLICODER

```
    transition: all 0.2s ease-in-out;
    cursor: pointer;
    min-height: 38px;
}

button.default {
    background-color: #e8e8e8;
    color: #333;
    box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
}

button.primary {
    background-color: #128ff2;
    box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
    color: #fff;
}

button.accent {
    background-color: #ff4743;
    box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
    color: #fff;
}

#username-page {
    text-align: center;
}

.username-page-container {
    background: #fff;
    box-shadow: 0 1px 11px rgba(0, 0, 0, 0.27);
    border-radius: 2px;
    width: 100%;
    max-width: 500px;
```

CALLICODER

```
    vertical-align: middle;
    position: relative;
    padding: 35px 55px 35px;
    min-height: 250px;
    position: absolute;
    top: 50%;
    left: 0;
    right: 0;
    margin: 0 auto;
    margin-top: -160px;
}

.username-page-container .username-submit {
    margin-top: 10px;
}


#chat-page {
    position: relative;
    height: 100%;
}

.chat-container {
    max-width: 700px;
    margin-left: auto;
    margin-right: auto;
    background-color: #fff;
    box-shadow: 0 1px 11px rgba(0, 0, 0, 0.27);
    margin-top: 30px;
    height: calc(100% - 60px);
    max-height: 600px;
    position: relative;
}
```

CALLICODER

```
    list-style-type: none;
    background-color: #FFF;
    margin: 0;
    overflow: auto;
    overflow-y: scroll;
    padding: 0 20px 0px 20px;
    height: calc(100% - 150px);
}

#chat-page #messageForm {
    padding: 20px;
}

#chat-page ul li {
    line-height: 1.5rem;
    padding: 10px 20px;
    margin: 0;
    border-bottom: 1px solid #f4f4f4;
}

#chat-page ul li p {
    margin: 0;
}

#chat-page .event-message {
    width: 100%;
    text-align: center;
    clear: both;
}

#chat-page .event-message p {
    color: #777;
    font-size: 14px;
```

CALLICODER

```
#chat-page .chat-message {
    padding-left: 68px;
    position: relative;
}

#chat-page .chat-message i {
    position: absolute;
    width: 42px;
    height: 42px;
    overflow: hidden;
    left: 10px;
    display: inline-block;
    vertical-align: middle;
    font-size: 18px;
    line-height: 42px;
    color: #fff;
    text-align: center;
    border-radius: 50%;
    font-style: normal;
    text-transform: uppercase;
}

#chat-page .chat-message span {
    color: #333;
    font-weight: 600;
}

#chat-page .chat-message p {
    color: #43464b;
}

#messageForm .input-group input {
```

CALLICODER

```
}

#messageForm .input-group button {
    float: left;
    width: 80px;
    height: 38px;
    margin-left: 5px;
}

.chat-header {
    text-align: center;
    padding: 15px;
    border-bottom: 1px solid #ececec;
}

.chat-header h2 {
    margin: 0;
    font-weight: 500;
}

.connecting {
    padding-top: 5px;
    text-align: center;
    color: #777;
    position: absolute;
    top: 65px;
    width: 100%;
}


@media screen and (max-width: 730px) {

    .chat-container {
```

CALLICODER

```
            margin-top: 10px;
        }
    }

    @media screen and (max-width: 480px) {
        .chat-container {
            height: calc(100% - 30px);
        }

        .username-page-container {
            width: auto;
            margin-left: 15px;
            margin-right: 15px;
            padding: 25px;
        }

        #chat-page ul {
            height: calc(100% - 120px);
        }

        #messageForm .input-group button {
            width: 65px;
        }

        #messageForm .input-group input {
            width: calc(100% - 70px);
        }

        .chat-header {
            padding: 10px;
        }

        .connecting {
```

```
    .chat-header h2 {
        font-size: 1.1em;
    }
}
```

Running the application

You can run the Spring Boot application by typing the following command in your terminal -

```
$ mvn spring-boot:run
```

The application starts on Spring Boot's default port 8080. You can browse the application at http://localhost:8080.

Using RabbitMQ as the message broker

If you want to use a full featured message broker like RabbitMQ instead of the simple in-memory message broker then just add the following dependencies in your `pom.xml` file -

```
<!-- RabbitMQ Starter Dependency -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>

<!-- Following additional dependencies are required for RabbitMQ STOM
```

```
        <artifactId>reactor-core</artifactId>
    </dependency>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-net</artifactId>
    </dependency>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.19.Final</version>
    </dependency>
```

Once you've added the above dependencies, you can enable RabbitMQ message broker in the `WebSocketConfig.java` file like this -

```
public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.setApplicationDestinationPrefixes("/app");

    // Use this for enabling a Full featured broker like RabbitMQ
    registry.enableStompBrokerRelay("/topic")
            .setRelayHost("localhost")
            .setRelayPort(61613)
            .setClientLogin("guest")
            .setClientPasscode("guest");
}
```

Conclusion

Congratulations folks! In this tutorial, we built a fully-fledged chat application from scratch using Spring Boot and WebSocket.