**Project Report**

# Group Members:

* Shobhit Kumar BT20CSE032
* Kush Waikar BT20CSE069
* Garvit Verma BT20CSE078
* Shubham Vishwakarma BT20CSE088

# Contribution:

Garvit Verma (BT20CSE078):

* JAVA Backend server code,
* Documentation for JAVA code

Shobhit Kumar (BT20CSE032):

* Diagram Designing
* Document Design and Layout

Shubham Vishwakarma (BT20CSE088):

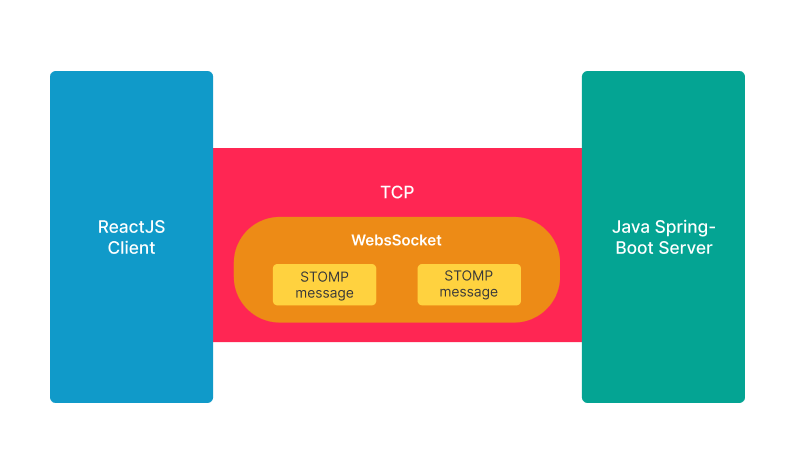
* ReactJS frontend client code
* Documentation for ReactJS code

Kush Waikar (BT20CSE069):

* ReactJS frontend client code
* Documentation for ReactJS code.

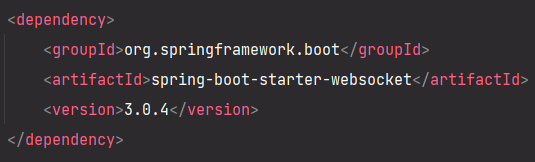
# Code Summary:

The project works on the concept of STOMP messaging protocols on WebSocket. This project relies heavily on the implementation of WebSocket by Spring and the npm packages.



## Server Side:

The WebSocket server-side is made by utilizing the Spring Boot framework. Firstly, the WebSocket library dependency is added into the pom.xml file.



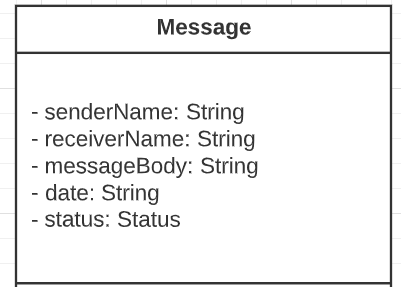
The configuration to enable Spring handle WebSocket and STOMP messaging is done in the file ***WebSocketConfig.java***. The annotations **@Configuration** tells us that it is a Spring configuration class while **@EnableWebSocketMessageBroker** enables WebSocket message handling, backed by a message broker.

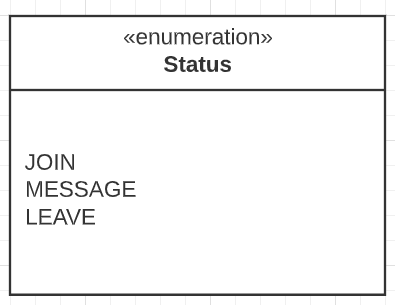


The method ***configureMessageBroker*** does two things:

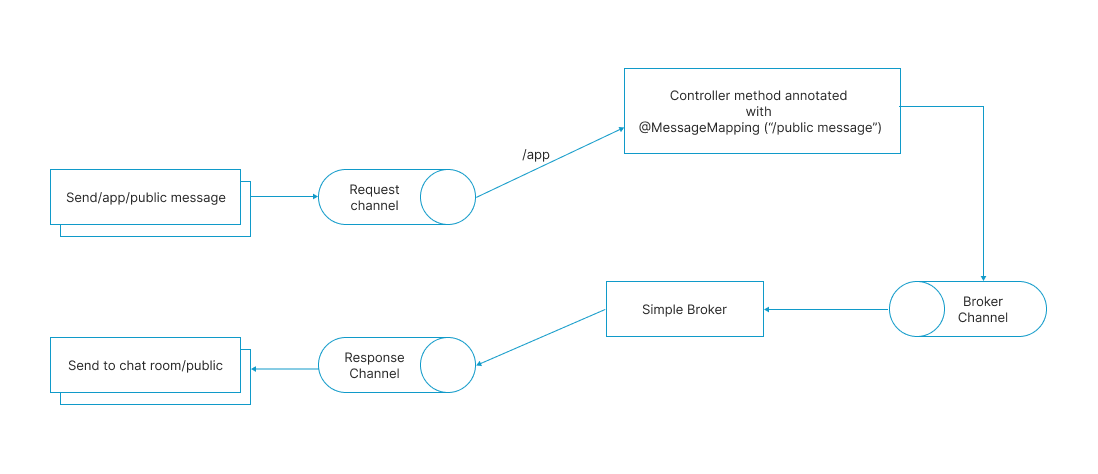
1. Creates a message broker with one or more destinations for sending and receiving messages. In the image above, two destination prefixes are defined: ***chatroom*** and ***user***. Messages that are intended to be carried on to all subscribed clients are prefixed with **chatroom**, while destinations for private messages are prefixed by **user**.
2. Defines the prefix ***app***. Methods annotated by **@MessageMapping** will be triggered only if the message has the “**app”** prefix. The controller, after processing the message, will send it to the broker.

To model the message that carries the data, a POJO is created. The name of this class is ***Message***.



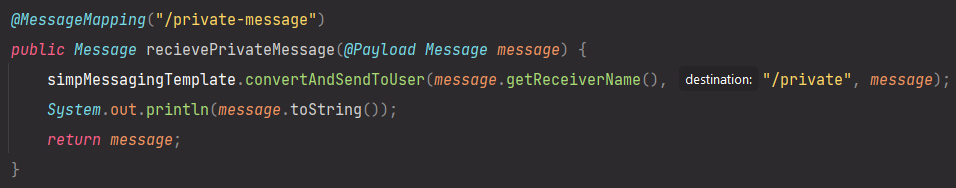


The class **ChatController** is responsible for handling the user requests coming in from the client.



The incoming messages from ***/app/public-message*** are handled by the method ***recievePublicMessage***. After printing the message contents to the console, it is then sent to the destination: ***/chatroom/public*** mentioned in the **@SendTo** annotation, where it is displayed to all the clients by the ReactJS clients listening at the destination ***/chatroom/public***.

The messages coming from the source ***/app/private-message*** are handled by the method ***recievePrivateMessage.***



From the received **Message** object, the receiver’s name is extracted and passed into the method ***convertAndSendToUser*** of the Spring class ***SimpMessagingTemplate***. This method takes the following parameters:

* user - the user that should receive the message. This information is extracted from the incoming message object.
* destination - the destination to send the message to. In this case it is ***“/app/private”***.
* payload - the payload to send. In this case it is the message.

### How to run the Server?

Open the directory ***chat-server*** in a terminal and enter the following command: ***mvn spring-boot:run***(This method requires Maven to be installed in the system).

OR

Open the ***chat-server*** folder in an IDE for JAVA like IntelliJ Idea, VS Code or Eclipse. Now run the ***ChatApplication.java*** file.