Web Portfolio Project

Aquilante Stanislao - 124002538
Cuomo Luigi - 124002553
Maglione Federico - 124002605
Virano Lorenzo - 124002536

Reti di Calcolatori

Uniparthenope

Indice

1 Introduzione

Il progetto del sito di social network per aspiranti fotografi nasce con l'obiettivo di fornire una piattaforma dedicata agli appassionati di fotografia che desiderano condividere e mettere in mostra i propri lavori. L'idea principale è quella di creare uno spazio online dove i fotografi amatoriali possano esporre le proprie foto e ricevere feedback dal pubblico interessato.

Il sito si propone di essere una comunità inclusiva, aperta a tutti coloro che hanno una passione per la fotografia, indipendentemente dal livello di esperienza. Gli utenti avranno la possibilità di caricare le proprie immagini, organizzarle in gallerie e interagire con altri membri della community attraverso commenti e valutazioni.

Uno degli obiettivi principali del sito è quello di creare un ambiente positivo e costruttivo dove gli aspiranti fotografi possano migliorare le proprie abilità attraverso l'interazione con gli altri membri e il confronto con le opere altrui. Il feedback e il supporto della community possono essere preziosi strumenti di crescita per chiunque desideri sviluppare le proprie competenze nel campo della fotografia.

Attraverso il sito, gli utenti avranno l'opportunità di esplorare una vasta gamma di stili fotografici, scoprire nuove tecniche e ispirarsi al lavoro degli altri membri della community. Inoltre, il sito potrà fungere da vetrina per gli utenti che desiderano promuovere il proprio talento e ottenere visibilità nel mondo della fotografia.

In sintesi, il sito di social network per aspiranti fotografi si propone di essere un luogo accogliente e stimolante dove gli amanti della fotografia possono condividere la loro passione, migliorare le proprie abilità e costruire relazioni significative con altri membri della community.

2 Descrizione del progetto

Il seguente progetto interdisciplinare è suddiviso in due parti:

- Portfolio Web App;
- Chat locale in C;

La prima sezione di questo progetto, in sintonia con le altre prove d'esame, si propone di sviluppare un'applicazione web che sfrutti i protocolli del modello TCP/IP per integrare una **chat istantanea remota 1 a 1** tra i clienti e i fotografi, utenti della nostra web app. In particolare, sono adottati due protocolli operanti al livello di trasporto e basati su TCP: WebSocket e STOMP.
L'architettura della web app si compone di un frontend, realizzato mediante l'utilizzo della libreria **React** e programmato in **Javascript**, e di un backend sviluppato con il framework **Spring Boot** utilizzando il linguaggio di programmazione **Java**.
La seconda sezione di questo progetto, più incentrata sulla disciplina in questione e caratterizzata dall'adozione di un approccio di più basso livello attraverso l'utilizzo diretto dei socket, prevede l'implementazione di una **chat tra due processi attivi in locale**. Questi processi si scambiano messaggi reciprocamente attraverso l'intermediazione di un server dedicato, che funge da gestore di messaggi. Questa sezione è implementata in **linguaggio C** e fa uso delle Unix Socket e delle relative chiamate di sistema per gestire la comunicazione tra i processi locali.

3 Architettura del sistema

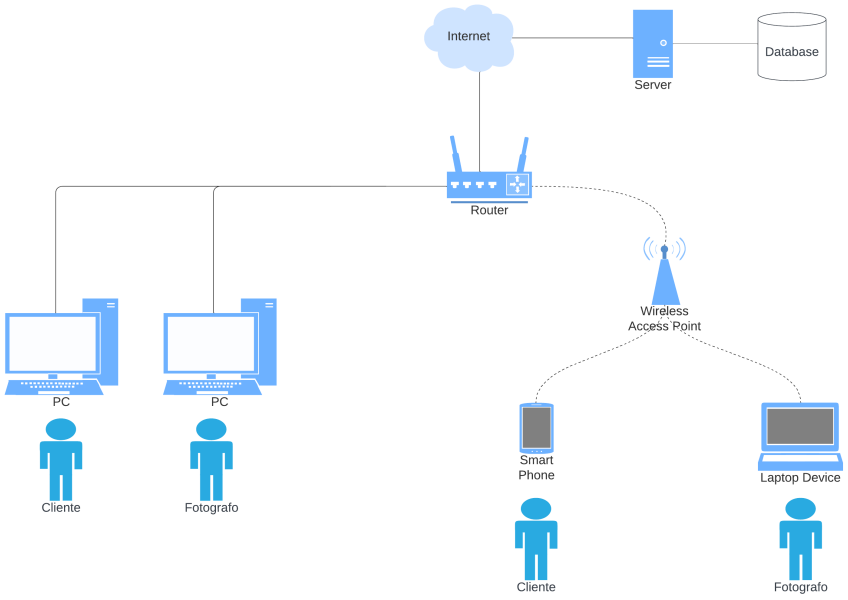

Figura 1: Architettura Client-Server

L'architettura adottata per il sistema seguente aderisce alla definizione di modello Client-Server, dove il client è l'applicazione realizzata in React mentre il server è il backend realizzato usufruendo del framework Spring Boot. Ciò ci consente di applicare delle responsabilità specifiche agli attori del sistema, che non risultano allo stesso livello. Il frontend gestisce l'interfaccia utente e l'interazione con l'utente, il backend è costituito della logica di business e delle operazioni di elaborazione dei dati.
Essi comunicano mediante protocollo HTTP: il client effettua delle richieste al server, facendo uso nella fattispecie di RESTful APIs, il quale ritorna una risposta fornendo una risorsa al frontend. Tramite il frontend gli utenti hanno la possibilità di scambiarsi dei messaggi in una chat in tempo reale mediante l'utilizzo di un protocollo che opera su TCP: Web Socket. A differenza delle richieste HTTP tradizionali, WebSocket consente una comunicazione persistente e full-duplex tra il client e il server.

3.1 WebSocket

I WebSocket sono un protocollo di comunicazione bidirezionale (full-duplex) che opera sopra il protocollo TCP, fornendo una connessione persistente e in tempo reale tra un client e un server. A differenza dei protocolli di comunicazione

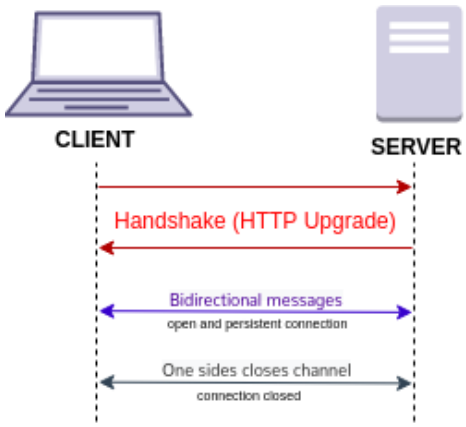

Figura 2: Comunicazione tramite Web Socket

tradizionali basati su HTTP, che seguono il modello di richiesta-risposta, i WebSocket, che aggiungono un livello di astrazione sopra il TCP, consentono una comunicazione bidirezionale efficiente e istantanea.

Ecco come funziona il WebSocket:

Handshake iniziale: Il processo inizia con un handshake iniziale, simile a quello utilizzato in HTTP. Il client invia una richiesta WebSocket al server, che risponde accettando la connessione. Questo avviene attraverso un'upgrade della connessione HTTP a WebSocket.
Connessione persistente: Una volta stabilita la connessione WebSocket, questa rimane aperta e persistente, consentendo la comunicazione bidirezionale senza la necessità di aprire una nuova connessione per ogni messaggio.

Frame di dati: Dopo l'handshake, i dati possono essere scambiati in entrambe le direzioni attraverso la connessione aperta. I dati sono inviati sotto forma di "frame", che possono contenere messaggi di testo o binari.

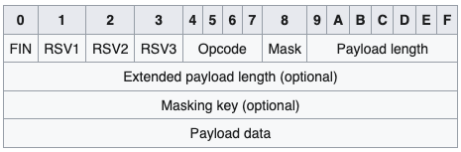

Figura 3: Web Socket "message"

Campi del messaggio WebSocket:

- **FIN(Final Fragment)**: 1 bit, indica se questo è l'ultimo frame;

- **RSV(Reserved)**: bit riservati impostati a 0, a meno che non sia definito da un'estensione di WebSocket;
- **Opcode**: specifica di che natura siano i dati presenti nel payload (ad esempio frame di tipo testuale, binario ecc.);
- **Mask**: bit per specificare se i dati sono mascherati;
- **Payload length**: lunghezza del payload espressa in byte, normalmente ha dimensione di 7 bit, ma se eccede questa dimensione, gli vengono riservati più bit;
- **Masking key**: contiene la maschera a 32 bit se il mask bit è impostato a 1;
- **Payload Data**: i dati che vengono effettivamente inviati nel messaggio;

Chiusura della connessione: La connessione WebSocket può essere chiusa da entrambe le parti, quando necessario, attraverso la trasmissione di frame di chiusura.

3.2 STOMP

Il protocollo STOMP (Simple Text Oriented Messaging Protocol) è un sistema di messaggistica testuale progettato per facilitare la comunicazione tra client e server attraverso diversi protocolli di trasporto, tra cui TCP, WebSocket e altri.

Il processo di comunicazione inizia con l'apertura di una connessione da parte del client verso una porta specifica, come la porta 61613 per connessioni non sicure. Questa connessione può essere instaurata tramite protocolli di trasporto come TCP o WebSocket, fornendo una base flessibile per le comunicazioni.

La trasmissione di messaggi tra client e server avviene mediante l'invio e la ricezione di "frame". Un frame STOMP è un messaggio di testo con una struttura predefinita, e il protocollo definisce vari tipi di frame, tra cui: CONNECT, SUBSCRIBE, UNSUBSCRIBE, SEND, BEGIN, COMMIT, ABORT, ACK, NACK, CONNECTED, ERROR e DISCONNECT.

I client interagiscono inviando comandi specifici al server attraverso frame dedicati; per esempio, utilizzano il comando CONNECT per stabilire una connessione, SUBSCRIBE per sottoscriversi a un canale e SEND per inviare un messaggio. Il server risponde con frame appropriati, come CONNECTED per confermare una connessione riuscita o ERROR per segnalare eventuali problematiche.

Ciascun frame può contenere intestazioni (headers) e un corpo che rappresenta il payload del messaggio. Il protocollo STOMP introduce il concetto di "destinazioni" per identificare canali o code di messaggi, permettendo ai client

di sottoscriversi a una destinazione specifica per ricevere i messaggi destinati a quella locazione.

Per garantire una consegna affidabile dei messaggi, STOMP supporta la gestione dell'acknowledgment. I client possono inviare messaggi di ACK (acknowledge) o NACK (negative acknowledge) per confermare o negare la ricezione di un messaggio, contribuendo a mantenere l'integrità della comunicazione.

3.3 Unix Socket

Per implementare una chat tra due terminali sullo stesso personal computer, è necessario adottare il meccanismo dei socket fornito dal sistema operativo. Sebbene i socket siano comunemente utilizzati per la comunicazione tra host remoti attraverso una rete, in un ambiente Unix-like, essi offrono anche la possibilità di consentire la comunicazione tra processi locali, implementando il cosiddetto "Inter-Process Communication" (IPC).

In questo contesto, useremo il linguaggio C per sviluppare due client che si scambieranno messaggi tramite socket, facendo uso delle chiamate API del sistema operativo offerte da alcune librerie C. Inoltre, sarà presente un server, un processo attivo che fungerà da gestore dei messaggi. Questo approccio riflette il workflow precedentemente realizzato per la Web App.

L'utilizzo dei socket e delle API di sistema consentirà ai due client di inviarsi reciprocamente messaggi, replicando in un contesto locale la stessa logica di comunicazione implementata nella Web App.

4 WebSocket Chat

4.1 Spring Boot Server

Per l'implementazione delle connessioni WebSocket lato backend è stato utilizzato il modulo WebSocket di Spring Boot. Questa ha costituito una nuova dipendenza, tra le altre, all'interno del nostro file di configurazione del progetto `pom.xml`.

4.1.1 Configurazione di WebSocket

```
1  public class WebSocketConfig implements
       WebSocketMessageBrokerConfigurer {
2
3      @Override
4      public void
           configureMessageBroker(MessageBrokerRegistry
           registry) {
5          registry.enableSimpleBroker("/user");
6          registry.setApplicationDestinationPrefixes("/app");
7          registry.setUserDestinationPrefix("/user");
8      }
9
10     @Override
11     public void
           registerStompEndpoints(StompEndpointRegistry
           registry) {
12         registry.addEndpoint("/ws");
13     }
14
15     @Override
16     public boolean
           configureMessageConverters(List<MessageConverter>
           messageConverters) {
17         DefaultContentTypeResolver resolver = new
               DefaultContentTypeResolver();
18         resolver.setDefaultMimeType(APPLICATION_JSON);
19         MappingJackson2MessageConverter converter = new
               MappingJackson2MessageConverter();
20         converter.setObjectMapper(new ObjectMapper());
21         converter.setContentTypeResolver(resolver);
22         messageConverters.add(converter);
23         return false;
24     }
25 }
```

WebSocketConfig

Questa classe implementa un'interfaccia messa a disposizione dalla libreria che ci consente di riscrivere i suoi metodi. Di seguito vengono illustrate le loro

implementazioni.

Il metodo `configureMessageBroker(MessageBrokerRegistry registry)` è stato impiegato per configurare il message broker dell'applicazione, il quale si occupa di instradare i messaggi tra i vari client. Nello specifico:

`registry.enableSimpleBroker("/user")` abilita un broker semplice con il prefisso "/user". Ciò consente ai client di inviare messaggi a indirizzi di destinazione che iniziano con "/user" tramite connessione web socket.

`registry.setApplicationDestinationPrefixes("/app")` specifica il prefisso per i messaggi inviati dal client all'applicazione. Ad esempio, se un client invia un messaggio a "/app/messaggio", questo verrà instradato all'interno dell'applicazione.

`registry.setUserDestinationPrefix("/user")` configura il prefisso per le destinazioni degli utenti. Questo viene utilizzato per inviare messaggi diretti a utenti specifici.

`registerStompEndpoints(StompEndpointRegistry registry)`: Questo metodo registra gli endpoint Stomp, che sono i punti finali a cui i client possono connettersi usando WebSocket. Nella fattispecie, viene registrato un endpoint "/ws".

`configureMessageConverters(List<MessageConverter> messageConverters)`:
Questo metodo configura i convertitori dei messaggi. Viene creato un DefaultContentTypeResolver con il tipo MIME predefinito impostato su APPLICATION_JSON. Successivamente viene istanziato un MappingJackson2MessageConverter che utilizza un ObjectMapper per convertire gli oggetti in formato `JSON`. Il convertitore viene configurato con il resolver del tipo di contenuto e aggiunto alla lista di convertitori dei messaggi.

Questa configurazione ci consente di supportare la comunicazione in tempo reale attraverso WebSocket, con un message broker configurato per gestire la messaggistica tra i client; la configurazione dei convertitori dei messaggi, nello specifico, assicura invece che gli oggetti Java vengano convertiti in formato JSON prima dell'invio e della ricezione attraverso WebSocket.

4.1.2 Endpoint per la chat

I controller sono componenti fondamentali all'interno di un'applicazione Spring Boot, incaricati di gestire le richieste degli utenti provenienti dal client. Essi fungono da tramite tra le richieste HTTP inviate dai client e la logica di business dell'applicazione. In particolare, i controller sono responsabili di esporre e mettere a disposizione all'esterno le funzionalità offerte dall'applicazione.

In modo più dettagliato, i controller svolgono le seguenti funzioni:

- **Gestione delle Richieste HTTP**: I controller definiscono metodi che rispondono a specifici endpoint HTTP. Ad esempio, un metodo può essere associato all'endpoint "/registrazione" per gestire le richieste di registrazione degli utenti.
- **Elaborazione delle Richieste**: Quando una richiesta arriva a un endpoint gestito da un controller, il relativo metodo del controller viene eseguito. Questo metodo contiene la logica necessaria per elaborare la richiesta, accedere ai servizi di business e ottenere i dati necessari.
- **Risposta al Cliente**: Dopo l'elaborazione della richiesta, il controller è responsabile di restituire una risposta appropriata al client. Questa risposta può essere un oggetto JSON, una pagina HTML o qualsiasi altro formato richiesto dalla richiesta del client.
- **Iniezione di Dipendenze**: I controller possono interagire con altri componenti dell'applicazione, come servizi, repository e altre classi di supporto. Queste dipendenze vengono iniettate nel controller attraverso l'inversione di controllo di Spring.

Di seguito, un controller che gestisce la logica di comunicazione in tempo reale attraverso WebSocket, inclusa la ricezione e l'invio di messaggi tramite WebSocket, nonché la gestione delle richieste HTTP per recuperare la cronologia dei messaggi salvati nel database.

```
1  @Controller
2  @RequiredArgsConstructor
3  public class ChatController {
4
5      private final SimpMessagingTemplate messagingTemplate;
6      private final ChatMessageService chatMessageService;
7
8      @MessageMapping("/chat")
9      public void processMessage(@Payload ChatMessage
           chatMessage) {
10         ChatMessage savedMsg =
               chatMessageService.save(chatMessage);
11         messagingTemplate.convertAndSendToUser(
12                 chatMessage.getRecipientId(),
                       "/queue/messages",
13                 new ChatNotification(
14                         savedMsg.getId(),
15                         savedMsg.getSenderId(),
16                         savedMsg.getRecipientId(),
17                         savedMsg.getContent()
18                 )
19         );
20     }
21
22     @GetMapping("/messages/{senderId}/{recipientId}")
```

```
23      public ResponseEntity<List<ChatMessage>>
            findChatMessages(@PathVariable String senderId,
24                                                          @PathVariable
                                                                String
                                                                recipientId)
                                                                {
25          return ResponseEntity
26                  .ok(chatMessageService.findChatMessages(senderId,
                        recipientId));
27      }
28  }
```

ChatController

Di seguito un'analisi del codice mostrato:

@Controller:
Indica che questa classe è un componente di **Spring MVC** che gestisce le richieste HTTP e WebSocket.

`private final SimpMessagingTemplate messagingTemplate`:
Questo campo rappresenta il SimpMessagingTemplate, che è un componente di Spring WebSocket utilizzato per inviare messaggi a destinazioni specifiche. In questo caso, verrà utilizzato per inviare messaggi attraverso WebSocket.

`private final ChatMessageService chatMessageService`:
Questo campo rappresenta un servizio (ChatMessageService) che sembra essere utilizzato per gestire la persistenza e il recupero dei messaggi di chat.

`@MessageMapping("/chat")`
`processMessage(@Payload ChatMessage chatMessage)`:
Questo metodo è mappato all'endpoint WebSocket "/chat". Riceve i messaggi inviati dai client attraverso WebSocket. Il parametro @Payload indica che l'oggetto ChatMessage ricevuto nel corpo del messaggio WebSocket deve essere deserializzato e passato come argomento al metodo. Salva il messaggio utilizzando il servizio chatMessageService e quindi utilizza messagingTemplate per inviare una notifica al destinatario attraverso WebSocket, utilizzando il metodo convertAndSendToUser.

`@GetMapping("/messages/senderId/recipientId")`
`public ResponseEntity<List<ChatMessage>> findChatMessages(...)`:
Questa metodo gestisce le richieste HTTP GET per ottenere la cronologia dei messaggi tra un mittente (senderId) e un destinatario (recipientId). Utilizza il servizio chatMessageService per recuperare i messaggi e li restituisce come una lista nella risposta HTTP.

4.2 React Client

Per l'implementazione delle connessioni WebSocket lato frontend, la libreria `@stomp/stompjs` ha messo a disposizione delle API accessibili per stabilire connessioni di tipo WebSocket nei confronti del server in Spring Boot.

4.2.1 Connessione al server

Di seguito una funzione Javascript che configura e attiva un client STOMP (con l'apposita classe fornita dalla libreria) specificando determinati argomenti nel costruttore, tra cui l'URL con il protocollo, l'host, la porta e il path dov'è stato configurato WebSocket sul server. Quando la connessione è stabilita con successo dopo la chiamata a `stompClient.onConnect`, l'utente corrente si iscrive alla propria coda di messaggi, specificando poi una callback come argomento del `stompClient.subscribe()` che verrà invocata alla ricezione di un nuovo messaggio da tale endpoint.

```
const setupStompClient = (username) => {
        const stompClient = new Client({
            brokerURL: "ws://localhost:8000/ws",
            reconnectDelay: 5000,
            heartbeatIncoming: 4000,
            heartbeatOutgoing: 4000,
            connectHeaders: config.headers,
        });

        stompClient.onConnect = () => {
            stompClient.subscribe(`/user/${username}/queue/messages`,
                (data) => {
                 onMessageReceived(data);
            });
        };

        stompClient.activate();
        setStompClient(stompClient);
    };
```

Configurazione STOMP client

Di seguito, viene implementata la procedura di invio di un messaggio. Nel dettaglio, vengono specificati il mittente, il destinatario (in questo caso, il fotografo), il testo del messaggio e la data di invio. Il metodo `publish` fornito dalla libreria Stomp.js viene utilizzato per inviare il corpo del messaggio, che contiene il payload del messaggio trasformato in una stringa JSON. Il messaggio è destinato

al percorso "/app/chat". Infine, la lista dei messaggi della chat viene aggiornata per includere il messaggio appena inviato. Questo processo garantisce la sincronizzazione della visualizzazione dei messaggi tra i partecipanti alla chat.

```
const sendMessage = (e) => {
        e.preventDefault();
        if (value && stompClient) {
            const payload = {
                senderId: username,
                recipientId: photographer,
                content: value,
                timestamp: new Date(),
            };
            stompClient.publish({ destination: '/app/chat',
                body: JSON.stringify(payload) });
            setValue("");
            setMessages(prevMessages => [...prevMessages,
                payload]);
        }
    };
```

Invio di un messaggio

La funzione onMessageReceived è responsabile della gestione dei messaggi ricevuti. In particolare, il parametro data rappresenta il messaggio ricevuto, il quale viene analizzato attraverso JSON.parse per ottenere un oggetto JavaScript.

Successivamente, viene verificato se il mittente del messaggio è diverso dall'attuale utente (rappresentato dalla variabile username). Questo controllo è effettuato per evitare di visualizzare i propri messaggi nella chat.

Inoltre, si verifica se il messaggio ricevuto non è già presente nella lista dei messaggi. In caso contrario, il messaggio viene aggiunto alla lista tramite la funzione setMessages, garantendo che solo i nuovi messaggi siano inclusi nella visualizzazione della chat.

```
const onMessageReceived = (data) => {
        const message = JSON.parse(data.body);
        if (message.senderId !== username) {
            if (!messages.some(msg => msg.id ===
                message.id)) {
                setMessages(prevMessages =>
                    [...prevMessages, message]);
            }
        }
    };
```

Messaggio in ricezione

La funzione loadHistoryMessages ha lo scopo di recuperare la cronologia dei messaggi tra l'utente (identificato dalla variabile username) e un fotografo (identificato dalla variabile photographer). Per fare ciò, viene effettuata una richiesta HTTP di tipo GET tramite Axios all'URL specificato, il quale è composto dalla base URL "http://localhost:8000" e dalla path /messages/$username/$photographer.

La richiesta include anche gli header necessari, come l'autorizzazione attraverso il token di accesso.

Una volta eseguita la richiesta, la funzione gestisce la risposta tramite una promessa. Nel caso di successo (then), il corpo della risposta (contenuto nel campo response.data.data) viene stampato sulla console. In caso di errore (catch), viene stampato un messaggio di errore nella console.

```
const loadHistoryMessages = () => {
        axios.request({
            headers: {
                'Authorization': `Bearer
                    ${Cookies.get('token')}`,
                'Content-Type': 'application/json',
            },
            method: "GET",
            url:
                `http://localhost:8000/messages/${username}/${photographer}`
        }).then(response => {
            console.log(response.data.data)
        }).catch(error => {
            console.log('Si e\' verificato un errore
                durante il caricamento dei messaggi:',
                error);
        });
    }
```

Caricamento dei messaggi antecedenti

La costante config è un oggetto che contiene un campo headers, a sua volta un oggetto. Quest'ultimo è utilizzato per specificare gli header delle richieste HTTP effettuate tramite Axios.

In particolare, è definito un solo header: 'Authorization', il quale viene valorizzato con una stringa che include il termine "Bearer" seguito dal token di accesso recuperato dal cookie. Questo tipo di autenticazione, noto come "Bearer Token Authentication", è spesso utilizzato per autenticare richieste HTTP nell'ambito delle API REST. Il token di accesso viene aggiunto all'header per dimostrare che l'utente autenticato ha il diritto di effettuare la richiesta.

In questo caso specifico, il token viene recuperato dal cookie denominato 'token' tramite Cookies.get('token'). Quindi, la configurazione config può essere

utilizzata per aggiungere l'header di autorizzazione alle richieste Axios.

```
const config = {
        headers: {
            'Authorization': `Bearer
                ${Cookies.get('token')}`,
        }
    };
```

JWT nell'header delle richieste

5 Unix Socket Chat

5.1 Unix Socket Server

Per la gestione della chat tra due terminali, il server intermedio consente di instradare correttamente i messaggi verso le porte corrette.

Affinchè i client possano connettersi, il server deve essere associato ad un endpoint, ovvero l'accoppiata indirizzo e porta, inizializzando il proprio socket. Vengono definiti famiglia di protocollo, porta e indirizzo. Dopodichè viene utilizzata la funzione `memset` per impostare tutti i byte dell'array `sin_zero` all/'interno della struttura dati `serverAddr`. Ciò è fatto per garantire che la struttura sia completamente inizializzata e non contenga valori indesiderati o dati residui. Il socket viene associato tramite `bind` alla struttura dati inizializzata.

```
unix_server_socket = socket(PF_INET, SOCK_STREAM, 0);
    serverAddr.sin_family = PF_INET;
    serverAddr.sin_port = htons(7891);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.01");
    memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
    bind(unix_server_socket, (struct sockaddr *) &serverAddr,
        sizeof(serverAddr));
```

Creazione socket server

```
    // Inizio dell'ascolto delle connessioni in entrata
    if (listen(unix_server_socket,5)==0)
        printf("Listening\n");
    else
        printf("Error\n");
```

Server in ascolto

Una volta inizializzato il socket e configurato in ascolto di 5 possibili connessioni differenti, viene impostato un loop in cui è il primo client a poter inviare un messaggio al server e una volta ricevuto, questo viene indirizzato verso il secondo client.

```
    // Ricezione del messaggio dal Client1 e invio al Client2
        recv(first_client, buffer, 1024, 0);
        printf ("%s\nInvia al Client2\n", buffer);
        send(second_client,buffer, 1024, 0);
        if (compare_strings(buffer, "exit")==0)
        {
            cmdEXIT = 1;
        }
```

Primo client invia primo messaggio

Successivamente, se la connessione non è stata chiusa dal primo client scrivendo la parola "exit" (condizione affinchè il loop possa terminare), allora il

server si prepara a ricevere la risposta del secondo client che verrà stampata sul terminale e inviata al primo client, controllando sempre la condizione di "exit" del loop.

```
    // Azzera il buffer e ricevi il messaggio dal Client2
            memset(&buffer[0], 0, sizeof(buffer));
            recv(second_client, buffer, 1024, 0);
            printf ("%s\nInvia al Client1\n", buffer);
            // Invia il messaggio ricevuto dal Client2 al Client1
            send(first_client,buffer,1024,0);
            if (compare_strings(buffer, "exit")==0)
            {
                cmdEXIT = 1;
            }
```

Server in ascolto

5.2 Unix Socket Client

I due client presenti nel sistema sono speculari nel loro funzionamento. Di seguito viene mostrata l'inizializzazione del socket e specificate le informazioni del server a cui vuol connettersi.

```
    // Creazione del socket del client
    clientSocket = socket(PF_INET, SOCK_STREAM, 0);
    if (clientSocket == -1) {
        perror("Errore durante la creazione del socket");
        exit(EXIT_FAILURE);
    }

    // Configurazione dell'indirizzo del server
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(7891);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
    addr_size = sizeof serverAddr;
```

Server in ascolto

Si prova la connessione:

```
    if (connect(clientSocket, (struct sockaddr*) &serverAddr, addr_size)
        == -1) {
        perror("Errore durante la connessione al server");
        close(clientSocket);  // Chiudi la connessione al socket in caso
            di errore
        exit(EXIT_FAILURE);
    }
```

Server in ascolto

Se la connessione del clientSocket all'indirizzo specificato va a buon fine, verrà mostrato un prompt che attende per l'input dallo standard input stream e quando lo riceverà, verrà inviato al server mediante la funzione `send`, come visto per il server in precedenza. Nel caso del secondo client invece si entrerà direttamente in un loop aspettando la ricezione del messaggio del primo client dal server.

Il primo client quindi, una volta inviato il messaggio, entra in un loop come il secondo client e aspetta di ricevere un messaggio dal server. Se è stato ricevuto (ovvero recvValue diverso da 1) allora stampa il messaggio ricevuto dal secondo client e resetta il buffer per il prossimo messaggio. Una volta mostrato il messaggio del secondo client, viene ripetuto il loop ed arrivato alla riga di `recvValue` la lunghezza dei dati sarà 1 (siccome si è resettato) ed entrerà nella clausola `else` e aspetterò un input dall'utente che sta usando il primo client.

```
// Inizializza tutti i byte dell'array buffer a zero
memset(&buffer[0], 0, sizeof(buffer));

// Ricevi dati dal socket clientSocket e memorizzali nell'array buffer
int recvValue = recv(clientSocket, buffer, sizeof buffer - 1, 0);

// Controlla se la ricezione    avvenuta correttamente (recvValue
    rappresenta la lunghezza dei dati ricevuti)
if (recvValue != 1) {
    // Controlla se il contenuto di buffer    diverso dalla stringa "exit"
    if (compare_strings(buffer, "exit") == -1) {
        // Stampa il messaggio ricevuto da Client 2
        printf("Client 2 : ");
        printf("%s\n", buffer);

        // Resetta tutti i byte di buffer a zero
        memset(&buffer[0], 0, sizeof(buffer));
    } else {
        // Imposta la variabile cmdEXIT a 1 se la stringa ricevuta
            "exit"
        cmdEXIT = 1;
    }
} else {
    // Se la lunghezza dei dati ricevuti    1, allora si sta chiudendo la
        connessione
    // Stampa "Client 1 : " e attende l'input da parte di Client 1
    printf("Client 1 : ");
    scanf(" %[^\n]s", buffer);

    // Invia i dati inseriti da Client 1 al socket clientSocket
    send(clientSocket, buffer, sizeof buffer - 1, 0);
}
```

Server in ascolto

6 Manuale Istruzioni

Per la prima sezione del progetto, le seguenti dipendenze sono state installate:

- React;
- Java 17;
- Spring Boot;
- Maven;

Il server Spring boot viene eseguito su porta 8000. Il server per eseguire la web app in React viene collegata alla porta 3000.

Spring Boot Build

- Nella cartella del progetto: ./mvnw spring-boot:run

React Build

- npm install;
- npm start;
- PORT=3001 npm start;

Socket in C

- gcc server.c string_util.c -o server.o
- ./server.o
- gcc client1.c string_util.c -o client1.o
- ./client1
- gcc client2.c string.c -o client2.o
- ./client2