Corso di Tecnologie e linguaggi per il web
A.A. 2019/2020

Valerio Cislaghi

17 giugno 2020

Indice

Capitolo 1

Introduzione

Il progetto realizzato consiste in una web live chat: Un utente, dopo essersi registrato, può iniziare a chattare con i propri amici/conoscenti solamente conoscendo il loro username (scelto appositamente nella fase di registrazione).
Per ottimizzare la velocità di esecuzione ho deciso di non affidarmi a framework lato client (es: jquery, angular ecc..), ma ho realizzato tutto il codice con javascript nativo, attraverso il pattern Model View Controller con la specifica standard ECMAScript 6.
Per il server ho utlizzato Node.js con il framework Express.
come DBMS ho utilizzato Postgresql, con la libreria 'pg-promise' per interfacciarmi ad esso.

Figura 1.1: Logo dell'applicazione

1.1 Destinatari

L'applicazione è rivolta al 'grande pubblico', essendo una chat molto semplice da usare; infatti basta registrarsi ed iniziare a chattare. Per chattare, essendo una web application, si potrà usare qualsiasi dispositivo che si possa connettere ad internet (è preribile utilizzare una versione desktop dell'applicazione)

1.2 Flusso dei dati

I dati richiesti per la registrazione sono:

- Username
- Password
- URI di un immagine per il profilo

1.3 Aspetti tecnologici

1.3.1 Struttura del codice

Model View Controller

Tutto il progetto si basa sulla realizzazione del pattern di sviluppo: Model View Controller:

- View = componenti che si occupano di gestire una parte ben specifica dell'interfaccia grafica, come per esempio le anteprime delle chat (side-panel.js) e la chat vera e propria (chat-frame.js).
- Controller = componenti che reagiscono ad un evento e passano il controllo al model adatto a gestire una specifica situazione. Un

evento può essere generato dalla view (es: click sul bottone di invio del messaggio), ma anche un evento esterno dalla GUI (es: sono arrivati dei nuovi messaggi dal server).

- Model = gestisce le iterazioni con il REQUSTER (unico oggetto che comunica con il server) e aggiorna di conseguenza la view associata a lui.

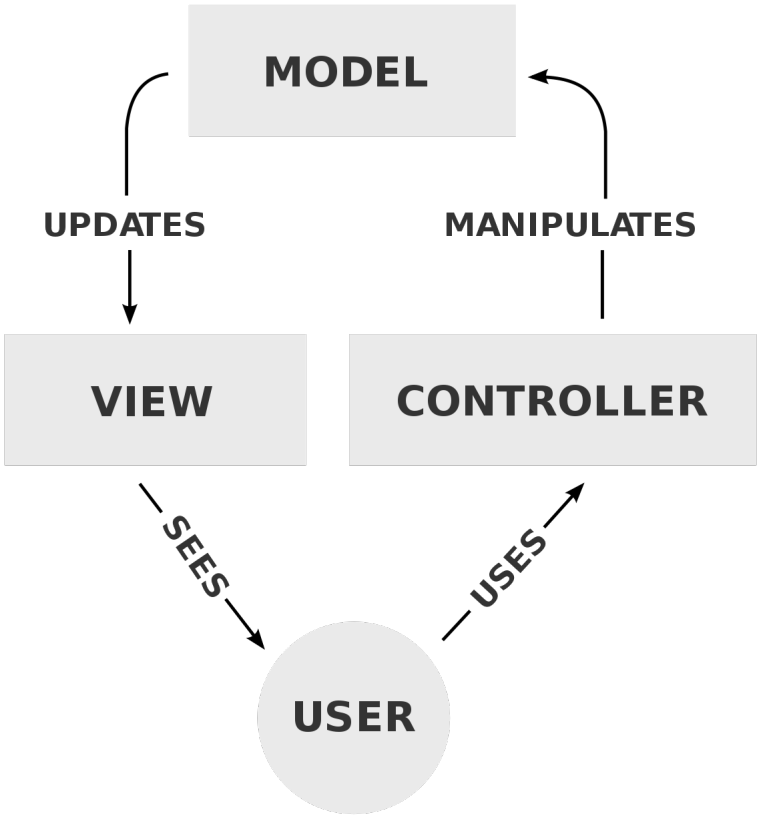

Figura 1.2: Model View Controller

Prototype Chain

Ho deciso di utilizzare le potenzialità dei prototipi con cui creare classi e oggetti per modellare tutto il progetto. Infatti, Per la gestione di Chat, Utenti, Messaggi ecc... **javascript client e Node.js utilizzano gli stessi oggetti, per una maggiore compatibilità**.

1.3.2 Tecnologie utilizzate

Inizialmente ho strutturato il progetto con l'utilizzo di jQuery, ma dopo aver fatto qualche analisi sulle performance (con gli strumentri di chrome), mi sono convinto che utilizzare le funzionalità native del linguaggio Javascript, come i prototipi, lo scope, il binding delle funzioni ecc... avrebbe migliorato le performance e la stabilità del progetto.
per il server, mi sono invece affidato a Node.js e Express con il quale ho realizzato le routes opportune per le viste e per le API. Come moduli ho utiilzzato

- ejs, per il rendering dell'html.
- pg-promise, per l'interrogazione del DataBase
- nodemon, per il refresh automatico del server dopo qualche cambiamento del codice

Capitolo 2

Interfacce

2.1 Login

L'utente, se già possessore di un account, potrà accedere all'applicazione tramite username e password inseriti durante l'iscrizione, altrimenti dovrà iscriversi. Durante la fase di registrazione il client, tramite chiamate REST, controllerà se lo username iserito è già presente nel database, in caso di successo mostrerà un messaggio di errore all'utente chiedendogli di inserire un altro nome utente.
Quindi prima di inviare la richiesta di inserimento dell'utente il client controllerà se lo username non è occupato e se le password inserite corrispondono.

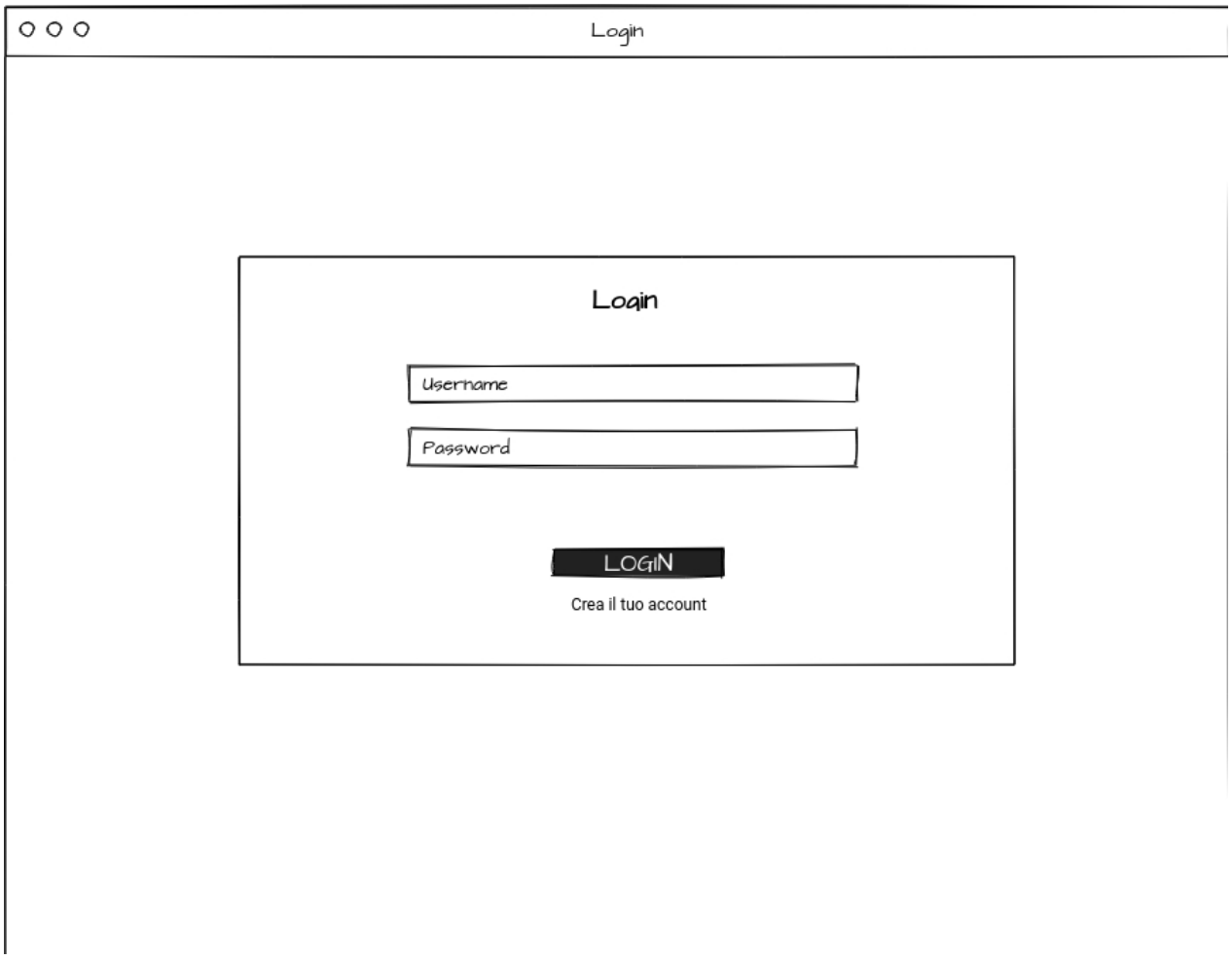

Figura 2.1: Mock Login

2.2 Chat

La pagina principale della web application, a sinistra troviamo le nostre chat già attive, mentre attraverso la barra di ricerca possiamo trovare gli utenti, con cui non abbiamo chat attive, per poter iniziare a messaggiare.

Le chat verranno aggiornate automaticamente, dunque non c'è bisogno di aggiornarle manualmente. Quando inviamo un messaggio e lo vediamo apparire sulla chat, vuol dire che il messaggio è stato inserito correttamente nel DataBase, e di conseguenza ricuvuto dal destinatario.

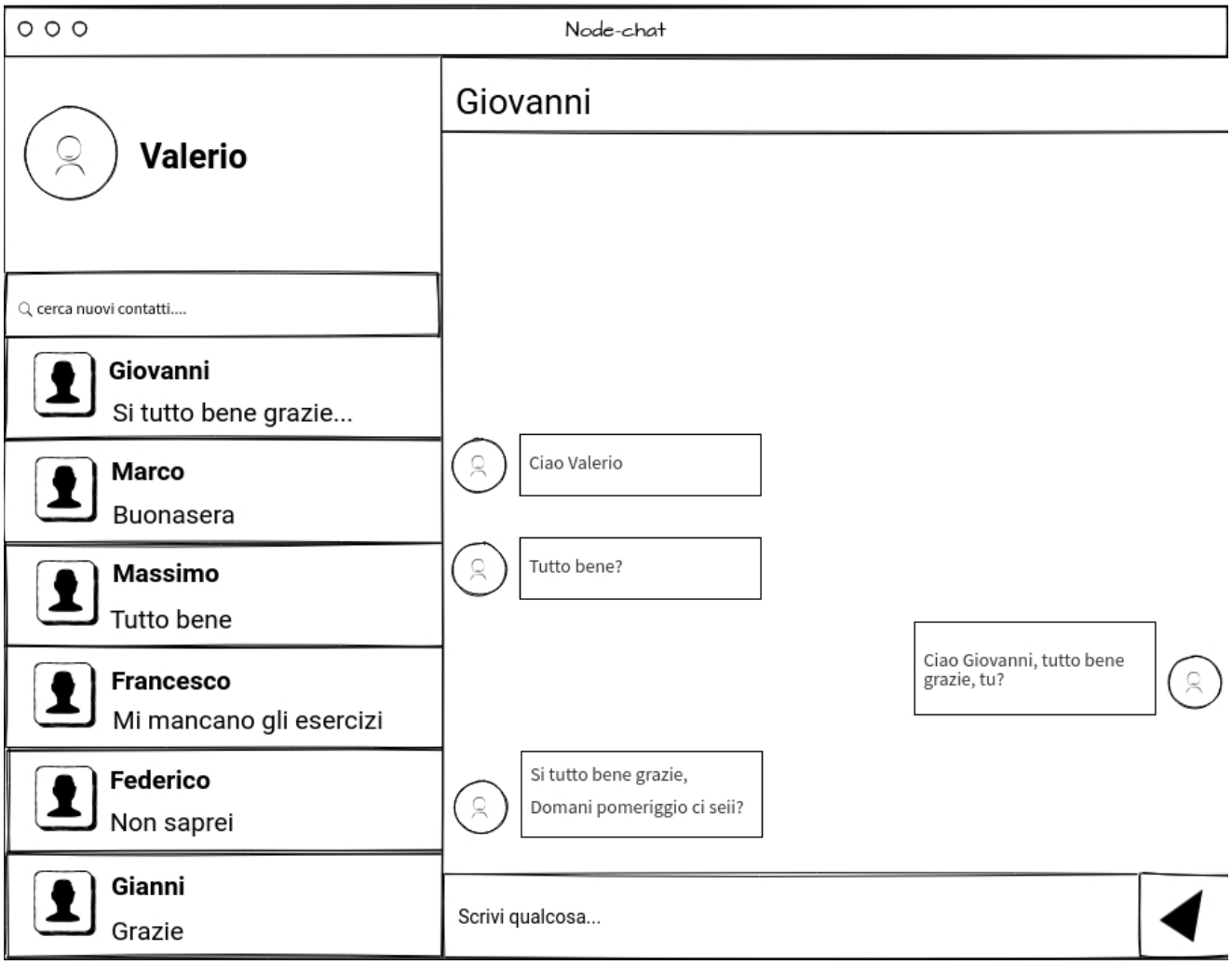

Figura 2.2: Mock Chat

Capitolo 3

Architettura

3.1 Diagramma dell'ordine gerarchico delle risorse

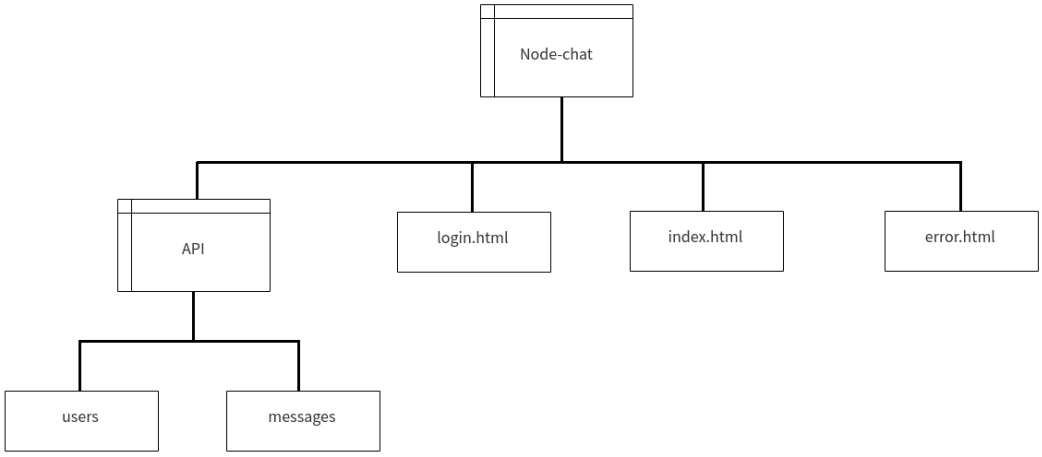

Figura 3.1: Struttura del sito

3.2 Diagramma delle richieste REST

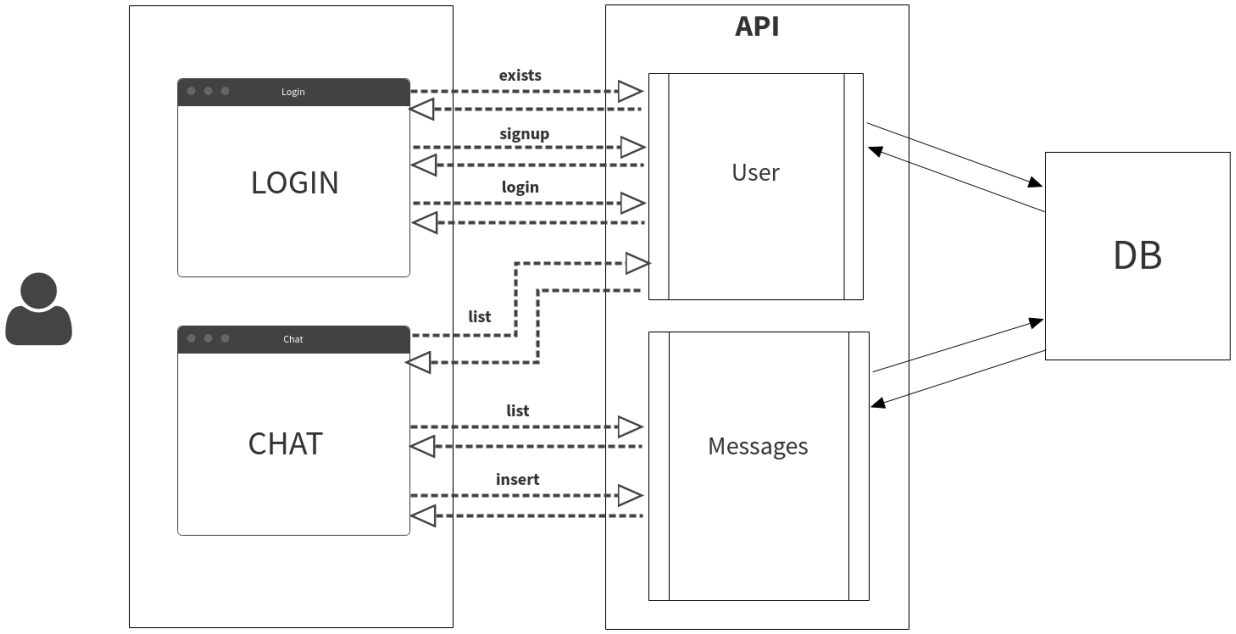

Figura 3.2: Richieste REST

Capitolo 4

Codice

4.1 Struttura del codice

Il codice è strutturato affinchè sia server che client utilizzino le stesse classi per la gestione dei dati.

```
if (typeof module !== 'undefined') {
    module.exports = {
        User: User,
        Message: Message,
        Chat: Chat
    }
}
```

Figura 4.1: Export delle classi per Node

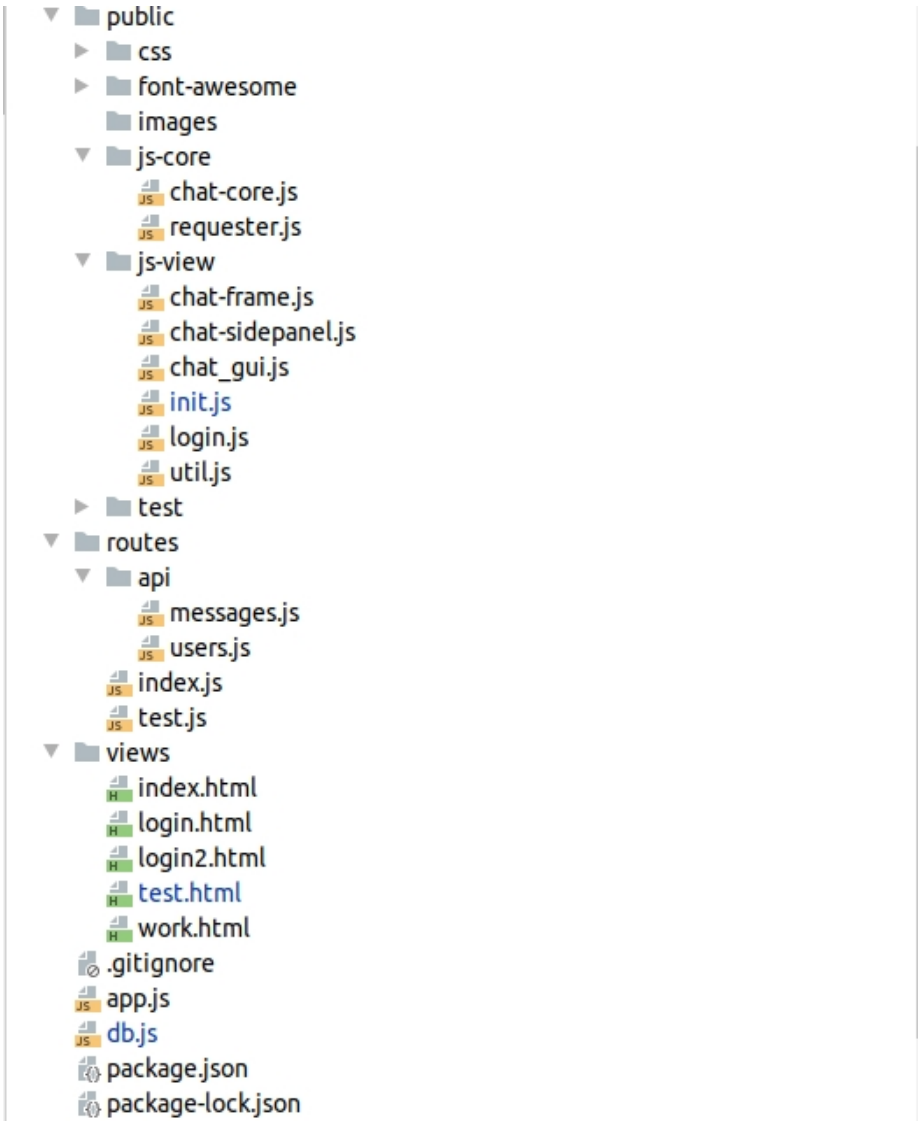

Figura 4.2: Struttura del codice

4.2 HTML

4.2.1 Login

Tramite l'attributo display, scelgo se far vedere il frame per il login o per la registrazione, stessa cosa per i messaggi di errore/successo.

```
<div id="sign_up_container" class="login100-form validate-form" style="display: none">
    <span id="passwordErrorMessage" class="login100-form-title" style="color: red; display: none">
            Le due password non corrispondono
    </span>
    <span id="usernameErrorMessage" class="login100-form-title" style="color: red; display: none">
            Username già esistente
    </span>
    <span id="userErrorMessage" class="login100-form-title" style="color: red; display: none">
            Errore nella creazione dell'utente
    </span>
    <span id="userSuccessMessage" class="login100-form-title" style="color: green; display: none">
            Utente creato con successo
    </span>
    <span class="login100-form-title">
            Registrati
        </span>

    <div class="wrap-input100 validate-input">
        <input id="user_sign_up" class="input100" type="text" name="username" placeholder="username">
        <span class="focus-input100"></span>
        <span class="symbol-input100">
                <i class="fa fa-envelope" aria-hidden="true"></i>
            </span>
    </div>
```

Figura 4.3: Messaggi di errore per il login

4.2.2 Chat

Come mostrato dalla figura 4.4 il codice HTML statico sono solamente i contenitori per il pannello a sinistra e per la chat vera e propria.
Mentre tutti i componenti saranno generati dalle opportune view (chat-frame.js, chat-sidepanel.js)

```
<div id="frame">
    <div id="sidepanel">
    </div>
    <div class="content" id="active-chat">
    </div>
</div>
<a id="logout_button" href="#" class="float">
    <i class="fa fa-sign-out my-float"></i>
</a>
</body>
</html>

<script>

    document.getElementById("logout_button").onclick = function () {
        localStorage.removeItem("current_user");
        redirectToLogin();
    };
</script>
```

Figura 4.4: Contenitori per i componenti

4.3 CSS

Porzione di CSS che descrive il chat frame e il side-panel.

```css
#frame #sidepanel {
    float: left;
    min-width: 280px;
    max-width: 340px;
    width: 40%;
    height: 100%;
    background: #2c3e50;
    color: #f5f5f5;
    overflow: hidden;
    position: relative;
}

@media screen and (max-width: 735px) {
    #frame #sidepanel {
        width: 58px;
        min-width: 58px;
    }
}

#frame #sidepanel #profile {
    width: 80%;
    margin: 25px auto;
}

@media screen and (max-width: 735px) {
    #frame #sidepanel #profile {
        width: 100%;
        margin: 0 auto;
        padding: 5px 0 0 0;
        background: #32465a;
    }
}
```

Figura 4.5: Stile per il frame/sidepanel

4.4 Javascript Client

4.4.1 Init

Una volta loggati, saremo reindirizzati alla pagina della chat vera e propria, la prima cosa che verrà eseguita sarà il file init.js (figura 4.6).
Questo script ha il compito di inizializzare l'interfaccia grafica, creando le struttura MVC per il sidepanel e per il frame della chat.
Finita la creazione della GUI partirà il "demone" REQUESTER, che si occuperà delle richieste REST.

```
const sidePanelView = new SidePanelView("sidepanel");
const sidePanelModel = new SidePanelModel();
const sidePanelController = new SidePanelController(sidePanelView, sidePanelModel);

const chatFrameView = new ChatFrameView( chatElementId: "active-chat");
const chatFrameModel = new ChatFrameModel();
const chatFrameController = new ChatFrameController(sidePanelView, chatFrameView, chatFrameModel);

REQUESTER.start();
```

Figura 4.6: init.js

4.4.2 Observer

Ho utilizzato in molti contesti il pattern Observer, per far si che il cambiamento di stato di un certo oggetto generasse chiamate ad istanze interessate al cambiamento avvenuto.

```
class ObservableView {
    constructor() {
        this._callbacks = new Map();
    }

    addObserver(event, callback) {
        if (!this._callbacks.has(event))
            this._callbacks[event] = [];

        this._callbacks[event].push(callback);
    }

    removeObserver(callback) {
        this._callbacks.splice(callback);
    }

    removeAllObserver() {
        this._callbacks.forEach( callbackfn: callback => {
            this.removeObserver(callback);
        })
    }

    notifyObserver(event, state) {
        const interestedCallbacks = this._callbacks[event];
        interestedCallbacks.forEach(callback => {
            callback(this, state);
        });
    }

}
```

Figura 4.7: util.js

4.4.3 Chat Core

Dentro il file chat-core.js ci sono le classi per la gestione logica delle strutture principali (User, Chat, Message).
Questi oggetti verrano utilizzati sia dal client che dal server (con la tecnica mostrata in figura 4.1).
Come possiamo notare dalla figura 4.7 la classe Message contiene i metodi toJSON e fromJSON, utlizzati per scambiare oggetti tra server e client in modo molto intuitivo.

```
class Message {
    constructor(from_username, to_username, text, timestamp) {
        this._fromUsername = from_username;
        this._toUsername = to_username;
        this._text = text;
        this._timestamp = timestamp
    }

    get fromUsername() {
        return this._fromUsername;
    }

    get toUsername() {
        return this._toUsername;
    }

    get text() {
        return this._text;
    }

    get timestamp() {
        return this._timestamp;
    }

    toJSON() {
        return {
            "from_username": this._fromUsername,
            "to_username": this._toUsername,
            "text": this._text,
            "timestamp": this._timestamp
        };
    }

    static fromJSON(JSONMessage) {
        return new Message(JSONMessage.from_username, JSONMessage.to_username, JSONMessage.text, JSONMessage.timestamp);
    }
```

Figura 4.8: core.js

4.4.4 Requester

```
class Requester {
    constructor() {
        this._timestampRequest = 0;
        this._requestPeriodicTime = 850;
        this._observers = [];

        const url = window.location.href;
        const tokens = url.split( separator: "/");
        this.baseURL = tokens[0] + "//" + tokens[2] + "/";

    }

    start() {
        this.chatRequest();
        window.setInterval(this.chatRequest.bind(this), this._requestPeriodicTime);
    }
    chatRequest() {
        const httpRequest = new XMLHttpRequest();
        httpRequest.onreadystatechange = function () {
            if (httpRequest.readyState === 4 && httpRequest.status === 200) {
                const chatsData = JSON.parse(httpRequest.responseText);
                const chats = [];
                chatsData.forEach(chatData => {
                    chats.push(Chat.fromJSON(chatData));
                });
                if (chats.length > 0)
                    this._observers.forEach(observer => observer.onUpdateChat(chats));
            }

        }.bind(this);
        const params = 'username=' + CURRENT_USER.username + "&" + "timestamp=" + this._timestampRequest;
        httpRequest.open( method: 'GET', url: this.baseURL + "api/messages?" + params, async: true);
        httpRequest.setRequestHeader( name: "Content-type", value: "application/x-www-form-urlencoded");

        this._timestampRequest = Date.now();
        httpRequest.send(params);
```

Figura 4.9: requester.js

REQUESTER è un oggetto globalmente accessibile inizializzato dallo script init.js(vedi figura 4.6).
E' l'unico oggetto che utilizza AJAX per richieste alle API del server.
Ogni metodo riceve in input una callback che sarà invocata una volta che la richiesta sarà conclusa.
Per alcuni metodi ho utilizzato il Pattern Observer perchè più ogget-

ti dipendevano dal risultato di quella specifica richiesta (es: una nuovo messaggio deve aggiornare sia il side-panel che il frame). Il metodo chatRequest verrà eseguito periodicamente per verificare se ci sono nuovi messaggi per l'utente corrente.

4.5 Node.js

4.5.1 Configurazione rotte

Notiamo dalla figura 4.10 che sono configurate tre diverse rotte, rispettivamente due per le API, mentre l'indexRouter sarà utilizzata per le richieste di pagine HTML.

```
var logger = require('morgan');

var usersRouter = require('./routes/api/users');
var messagesRouter = require('./routes/api/messages');
var indexRouter = require('./routes/index');

var app = express();


// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.engine( ext: 'html', require('ejs').renderFile);
app.set('view engine', 'html');

app.use(logger( format: 'dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use( fn: '/', indexRouter);
// app.use('/api/', apiRouter);
app.use( fn: '/api/users', usersRouter);
app.use( fn: '/api/messages', messagesRouter);
```

Figura 4.10: app.js

4.5.2 Database

Classe che gestisce le comunicazioni con il Database, attraverso il modulo: 'pg-promise'. Ogni metodo restituirà un oggetto di tipo Promise per la gestione del risultato.

```
const pgp = require('pg-promise')(/* options */);

class DB {
    constructor(host, port, database, user, password) {
        const cn = {
            host: host,
            port: port,
            database: database,
            user: user,
            password: password,
            max: 30 // use up to 30 connections
        };
        this._db = pgp(cn)
    }

    getUser(user) {
        let query = "SELECT * FROM c_user WHERE 1=1";
        const params = {};
        if (user.username !== null && user.username !== undefined) {
            query += " AND username=${username}";
            params['username'] = user.username
        }
        if (user.password !== null && user.password !== undefined) {
            query += " AND password=${password}";
            params['password'] = user.password
        }
        return this._db.query(query, params);
    }

    getMessages(currentUser, timestamp) {
        let query =
            "SELECT cm.user_from, cm.user_to, cm.text, timestamp, c2.img as img_URL " +
            "FROM c_user c1 " +
            "INNER JOIN c_message cm on ${current_username} = cm.user_from " +
            "INNER JOIN c_user c2 on c2.username = cm.user_to " +
            "WHERE cm.timestamp > ${timestamp} " +
            "UNION " +
```

Figura 4.11: db.js

4.5.3 Esempio API Users

Il codice mostrato in figura 4.11 mostra la gestione di una richiesta REST per la registrazione di un nuovo utente. In questa procedura viene utilizzato l'oggetto DB, come possiamo vedere dai metodi della classe Promise(then e catch) il risultato della richiesta verrà trasmesso al client attraverso un HTTP status code.

```
router.post('/sign_up/', function (req, res, next) {
    const username = req.param("username");
    const imgURL = req.param("img_url");
    const password = req.param("password");
    const newUser = new User(username, imgURL, password);
    db.insertUser(newUser) Promise<any>
        .then(success => {
            res.status(201);
            res.end()
        }) Promise<any>
        .catch(_ => {
            console.log("errore", _);
            res.status(400);
            res.end()
        });
    res.setHeader( name: 'Content-Type', value: 'application/json');

});
```

Figura 4.12: users.js

4.5.4 Esempio API Messages

In figura 4.13 è mostrato un'esempio di gestione di una richiesta REST per l'inserimento di un nuovo messaggio(da notare anche il metodo, POST viene utilizzato per gli inserimenti).
Anche in questo caso il risultato sarà restituito tramite l'HTTP status code.

```
const express = require('express');
const db = require(__dirname + "/../../db");

const router = express.Router();

core = require('../../public/js-core/chat-core');
DataGenerator = require('../../public/test/generator-data');

User = core.User;
Message = core.Message;
Chat = core.Chat;


router.post('/', function (req, res, next) {
    const usernameFrom = req.param("username_from");
    const usernameTo = req.param("username_to");
    const timestamp = req.param("timestamp");
    const text = req.param("text");

    const userFrom = new User(usernameFrom);
    const userTo = new User(usernameTo);

    db.insertMessage(userFrom, userTo, timestamp, text) Promise<any>
        .then(messages => {
            res.status(201);
            res.end();
        }) Promise<any>
        .catch(_=> {
            res.status(400);
            res.end();
        });
    res.setHeader( name: 'Content-Type', value: 'application/json');
```

Figura 4.13: messages.js

[1]

Bibliografia

[1] Valerio Cislaghi. *Relazione Progetto Tecnologie e linguaggi per il web.* 2020.