Real-time collaborative text editor - Final report

Maria Mengozzi
maria.mengozzi3@studio.unibo.it

September 2022

Sempre più spesso capita di dover scrivere un documento collaborando con più persone, il che può portare a diversi problemi qualora il documento sia accessibile da un solo utente per volta. Il seguente progetto verte alla realizzazione di un'applicazione web in grado di fornire uno strumento per la creazione e scrittura collaborativa di documenti testuali in real-time, sfruttando tecnologie come Express, Node.js e altri servizi esterni. L'obiettivo principale di questa applicazione è quindi quello di permettere ad un gruppo di persone di avere uno spazio unico e condiviso su cui poter prendere appunti contemporaneamente e visionare chi attualmente sta modificando il documento.

Indice

1 Obiettivi/Requisiti

L'obbiettivo del progetto è quello di realizzare un sistema distribuito basato su un'architettura client-server che permetta agli utenti di connettersi da remoto e creare ed editare in modo collaborativo documenti testuali.

Il sistema dovrà essere in grado di fornire le seguenti funzionalità:

- creare, modificare e salvare un nuovo documento, sia individuale che condiviso con gli altri utenti della piattaforma;
- creare dei gruppi di lavoro tra gli utenti registrati alla piattaforma e limitare l'accesso al documento ai soli membri del gruppo creato;
- accedere e scrivere in modo collaborativo su un documento prescelto;
- visionare la lista degli utenti che attualmente stanno modificando il documento;
- vedere in real-time le modifiche apportate dagli altri utenti al documento;
- vedere il proprio cursore e quello degli altri utenti mentre il documento viene modificato;
- consentire agli utenti del gruppo di lavoro di compilare una tabella su cui poter indicare i giorni della settimana in cui sono disponibili per i meeting;
- proporre il giorno migliore in cui programmare il meeting sulla base dei dati contenuti nella tabella precedentemente compilata.

1.1 Scenari

Supponiamo vi sia un utente che decide di utilizzare l'applicazione, per poter creare documenti testuali e collaborare con altre persone alla stesura di essi. Una volta aperta l'applicazione, all'utente finale verrà richiesto di registrarsi oppure di effettuare il login se già in possesso di un account.

Una volta effettuato l'accesso, l'utilizzatore potrà decidere se creare un nuovo documento oppure iniziare o continuare ad editare uno già precedentemente creato. Qualora decidesse di creare un nuovo documento, oltre a specificare il titolo potrà scegliere anche la lista di altri utenti, tra quelli già registrati alla piattaforma, che potrà collaborare e quindi accedere al documento. Questo quindi implica che fra tutti gli iscritti alla piattaforma, ve ne deve essere uno che crea il nuovo documento e che gli altri, aggiunti da quest'ultimo vi possano accedere.

Dopo aver creato il documento, con il rispettivo titolo, l'utente e gli eventuali collaboratori prescelti, lo vedranno aggiunto alla lista di quelli a cui possono accedere.

Per poter editare un documento l'utente non dovrà fare altro che selezionarlo dalla suddetta lista, dopo di che verrà reinderizzato alla pagina in cui potrà appunto visionare e modificare il contenuto del file.

Nella stessa schermata l'utente potrà inoltre visualizzare l'elenco degli utenti attivi che stanno modificando il documento e le modifiche che questi stanno apportando in tempo reale. Tali modifiche verranno salvate periodicamente affinchè ad ogni nuovo accesso al documento, questo sia aggiornato.

L'utente, sempre dalla pagina di modifica del documento, può accedere e compilare un planner settimanale utile per programmare gli incontri con gli altri membri del gruppo di lavoro. Sulla base di questa verrà proposto il giorno ideale in cui poter effettuare il meeting relativo alla discussione del contenuto del documento.

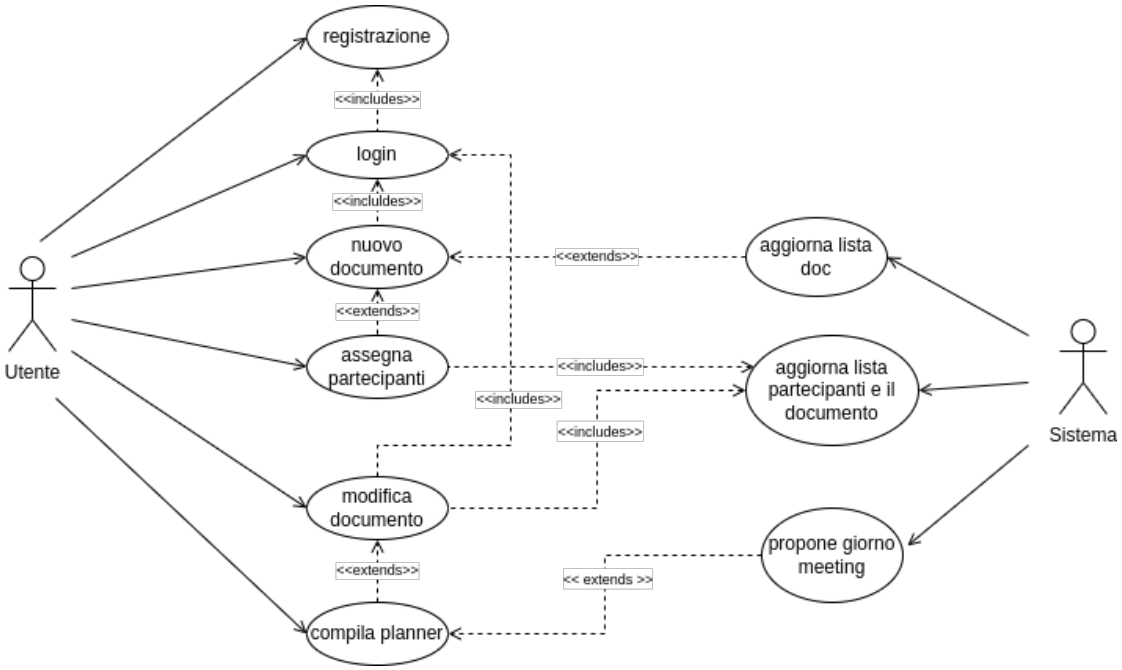


Figura 1: Diagramma dei casi d'uso dell'applicazione

1.2 Metodi di autovalutazione

Per valutare l'efficacia del software prodotto, si pensa di utilizzare sia test automatici che manuali. In particolare si pensa di testare alcune delle funzionalità di *back-end* tramite test automatici, mentre la parte di interfaccia grafica mediante test manuali. I dettagli riguardanti i test verranno approfonditi nella sezione 5.

Per testare invece la qualità del servizio offerto, si guarderà se l'applicativo realizzato è in grado di soddisfare tutti i requisiti emersi in fase di analisi e come tali requisiti vengono soddisfatti, cercando di rispettare i diversi obiettivi stabiliti e realizzare un sistema efficiente.

2 Analisi dei requisiti

L'applicazione ha come obiettivo principale quello di permettere agli utenti di connettersi da remoto e creare ed editare in modo collaborativo documenti testuali, a tal fine sono stati individuati i seguenti requisiti funzionali e non funzionali.

2.1 Requisiti funzionali

- All'avvio dell'applicazione, verrà mostrata l'interfaccia di login, attraverso cui sarà possibile: registrare un nuovo utente oppure accedere, mediante la propria e-mail e password, alla piattaforma.
- Dopo aver effettuato l'accesso un utente potrà:
  - creare un nuovo documento, specificando titolo e lista di utenti che possono accedervi, scelti tra quelli già registrati all'applicazione;
  - visualizzare l'elenco dei documenti, aggiornato in real-time, di cui gli è garantito l'accesso, ossia di cui è stato indicato come partecipante;
  - accedere ad uno dei suddetti documenti.
- Una volta selezionato un documento tra quelli disponibili, un'utente avrà la possibilità di:
  - visionare la lista degli utenti che stanno attualmente lavorando su quel documento, aggiornata in real-time;
  - visionare ed editare il contenuto del documento;
  - visionare il cursore degli altri partecipanti e le modifiche apportate da questi in real-time;
  - visionare e compilare il planner per gli incontri settimanali;
  - avere un'indicazione del giorno migliore per il meeting settimanale.

2.2 Requisiti non funzionali

- Il servizio deve essere in grado di gestire le richieste da più utenti contemporaneamente;
- l'applicazione deve poter essere eseguita e funzionare correttamente sui diversi tipi di browser;
- l'interfaccia deve essere reattiva all'interazione con l'utente;
- il design deve permettere una facile estensibilità in caso di modifiche future o eventuali funzionalità aggiuntive;
- le operazioni di manutenzione dell'applicazione devono essere agevoli da eseguire.

2.3 Analisi e modello del dominio

Dato che l'applicazione consiste in un servizio web messo a disposizione dei diversi utenti, si vuole realizzare il progetto aderendo al modello dei servizi *RESTful*, realizzando un'interfaccia del servizio offerto che possa essere riutilizzata.

Per questo le entità di base del dominio (Fig. 2) che si prevede di modellare sono:

- il **client** web, che invierà le richieste al server mediante opportune rotte, le quali consentiranno di eseguire le diverse operazioni;
- il **server**, il quale si occuperà di esporre il servizio ai clients, di definire le rotte e di implementare le diverse operazioni da effettuare interagendo con un database.
- un **database relazionale**, il quale sarà incaricato di mantenere le informazioni relative agli utenti, ai documenti, ai gruppi di lavoro e al planner per ogni documento e di fornirle al server su richiesta, per l'esecuzione delle operazioni.

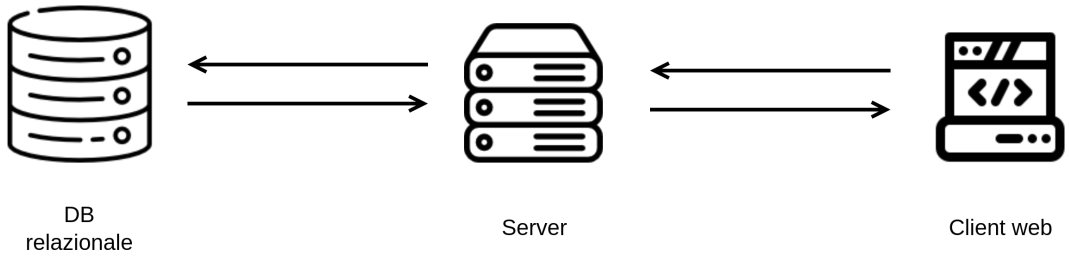


Figura 2: Rappresentazione delle entità del dominio e le loro interazioni

3 Design

In base alle informazioni raccolte durante la fase di analisi, si è proceduto alla fase di progettazione definendo la struttura delle diverse entità coinvolte e le interazioni tra di esse, i cui dettagli verranno trattati nelle prossime sezioni.

3.1 Struttura

Per la realizzazione del seguente progetto si è deciso di realizzare un'architettura con un server centrale (Fig. 3). Tale tipologia di architettura rappresenta la tipica modalità client-server, in cui: un server offre un servizio e i client effettuano richieste o si connettono ad esso. Il singolo client quindi non vede e non conosce gli altri.

Il server rappresenta il punto critico di tutta l'architettura, infatti in caso di problemi o guasti non è possibile offrire il servizio a nessun client. D'altra parte, mediante questo approccio, i messaggi di comunicazione risultano molto più semplici da gestire e da progettare.

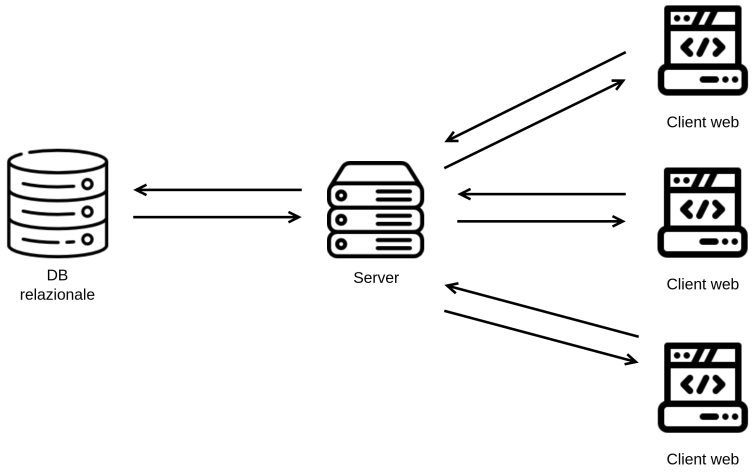


Figura 3: Architettura generale dell'applicazione

3.1.1 Struttura del progetto principale

Il progetto è stato realizzato effettuando una suddivisione in più moduli, i quali risultano essere:

- **pre-start**, racchiude tutto ciò che serve per la configurazione iniziale, ossia la scelta delle variabili d'ambiente da utilizzare in base alla tipologia di esecuzione scelta: development, test e production;
- **server**, racchiude al suo interno tutto ciò che riguarda il server, quindi gli script per identificare le rotte e i metodi per reperire le risorse dal database;
- **db**, tale modulo viene utilizzato per interagire con il database, quindi contiene le funzioni necessarie al collegamento con il database ed i componenti che definiscono le query per operare sui dati;
- **client**, racchiude il client web dell'applicazione.

Il modulo **server** dipende dal modulo **db** e **pre-start** per il suo funzionamento, mentre **client** risulta essere completamente indipendente dai precedenti. Nelle prossime sezioni verranno approfondi tali moduli indicando la loro struttura e composizione.

3.1.2 Struttura del database

Per prima cosa è stata definita la struttura del database mediante lo schema *Entity/Relationship*, mettendo in evidenza quali sono le diverse entità che dovranno essere modellate e le relazioni che sussistono tra di esse.

Come si può vedere dalla figura 4 lo schema è costituito dalle seguenti entità:

- **Utente**, rappresenta gli utenti iscritti alla piattaforma;

- **Documento**, rappresenta il documento testuale su cui gli utenti possono lavorare;
- **Planner**, rappresenta lo storico relativo alle preferenze dei giorni in cui un utente, per un determinato documento, è disponibile.

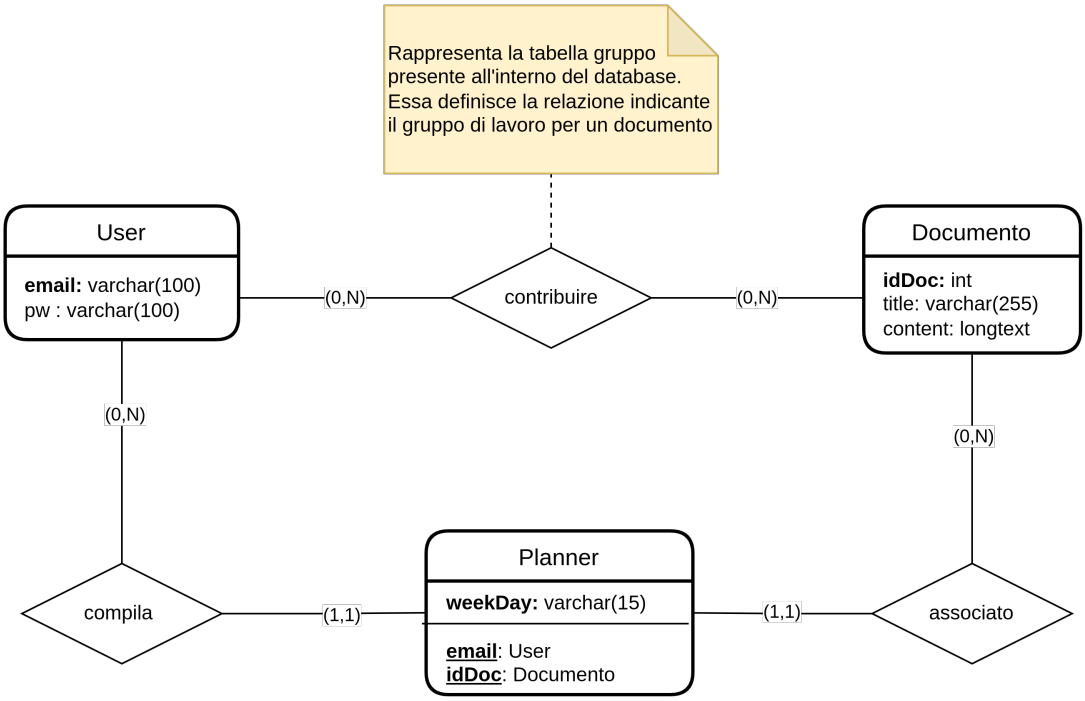


Figura 4: Schema Entity/Relationship

A partire dallo schema ER è stato poi definito lo schema logico da cui è scaturito il seguente risultato, in cui il testo in grassetto rappresenta le primary key e quello sottolineato le foreign key:

- UTENTE: **email**, pw
- DOCUMENTO: **idDoc**, title, content
- GRUPPO: **idDoc, email**. Tale tabella rappresenta il gruppo di lavoro relativo ad ogni documento, ossia definisce quali utenti possono accedere o meno ad un determinato documento);
- PLANNER: **weekDay, idDoc, email**

Il modulo **db** contiene due directory:

- **repo**, che contiene la definizione delle query suddivise in base all'entità a cui fanno riferimento;
- **util**, che contiene le funzioni per connettersi al database, resettarlo/inizializzarlo ed effettuare le query.

3.1.3 Struttura del modulo server - backend

Il modulo **server** rappresenta come detto il backend dell'applicazione, il cui compito è oltre che gestire le richieste, quello di interagire con il database per effettuare le operazioni richieste.

La struttura del backend, come si può vedere nella figura 5, è stata organizzata in quattro directories principali:

- **routes**
- **services**
- **models**
- **shared**

La directory `routes`, contiene le rotte che possono essere utilizzate dai clients per poter manipolare le relative risorse. Quindi effettuare le operazioni *CURD*: Create, Read, Update e Delete.

Le rotte sono assegnate al rispettivo endpoint a seconda della risorsa che manipolano. Gli endpoint esposti dal servizio sono:

- `/`, la root del servizio, la quale si occuperà di gestire l'accesso e la registrazione alla piattaforma;
- `/documents`, la quale si occupa della creazione dei nuovi documenti e la visualizzazione di quelli a cui l'utente può accedere;
- `/document`, la quale ha il compito di gestire la modifica di uno specifico documento, tenere traccia degli utenti che attualmente lavorano al documento e gestire il planner.

La directory `services`, si occupa dei servizi messi a disposizione dal server e utilizzati dalle rotte, come ad esempio quelli per effettuare le query al database. La directory `models` mantiene i modelli che possono essere utilizzati dai diversi controller, consentendo loro di avere un'interfaccia relativa all'output delle query. Infine la directory `shared` contiene alcune funzioni di utility, come ad esempio la definizione degli errori e le funzioni di log.

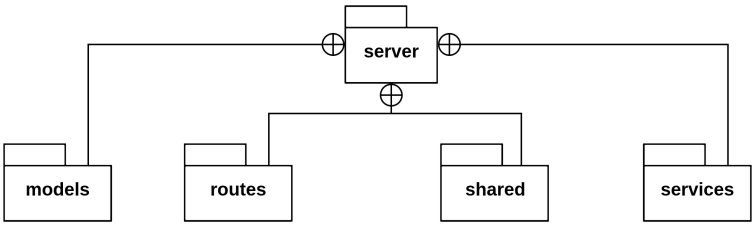


Figura 5: Diagramma dei packages del modulo server

Dalla figura 6 è possibile osservare più in dettaglio le dipendenze presenti tra i componenti del backend: le richieste effettuate dai client tramite le rotte vengono elaborate da queste e completate mediante i servizi, i quali si occuperanno di interagire con il database per mezzo delle utility contenute nel modulo **db**. A supporto dell'output ottenuto dalle query si sfruttano i modelli definiti sotto forma di interfacce.

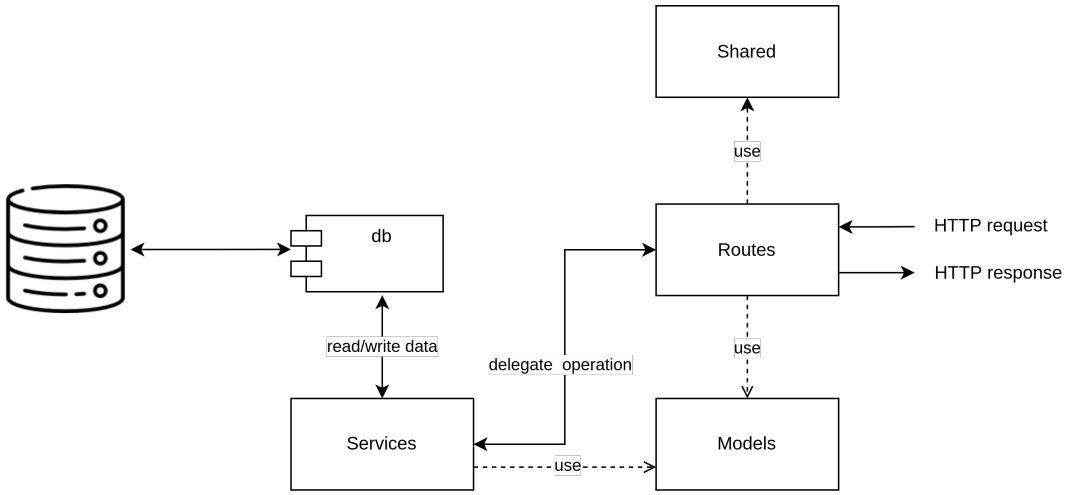


Figura 6: Relazioni tra i componenti del server

Struttura per l'editing di un documento Il documento è l'elemento centrale del sistema. Ogni documento, come specificato nella sezione 3.1.2 è rappresentato da un id, che lo identifica, un titolo ed un contenuto. La stessa rappresentazione è possibile constatarla nell'interfaccia **Document** presente nella figura 7, la quale mostra le relazioni che sussistono tra i vari componenti del backend al fine di gestire l'editing di un documento.

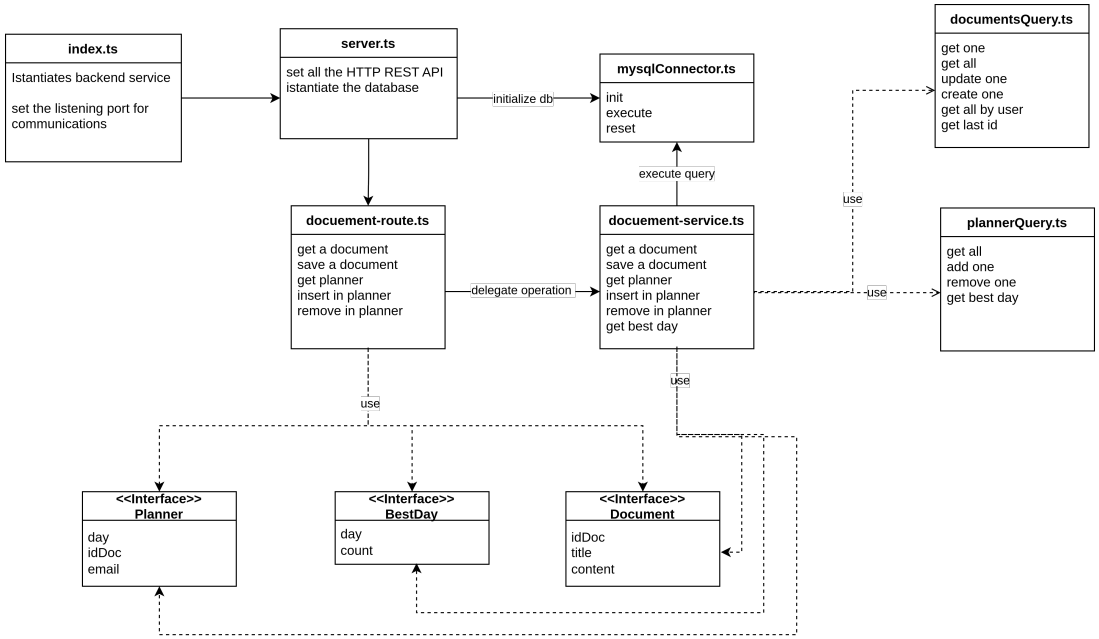


Figura 7: Relazioni tra i componenti del server per gestire l'editing di un documento

La soluzione adottata prevede un file iniziale *index.ts* il quale si occupa di configurare la porta per la comunicazione HTTP REST e di istanziare il file *server.ts*. In quest'ultimo file sono presenti e configurate tutte le API che il frontend chiama per comunicare con il backend e sfruttare le funzionalità implementate. La scelta di far ricadere su questo componente del sistema la maggior parte delle funzionalità di calcolo è dovuta al fatto che si vuole mantenere un frontend il più leggero ed indipendente possibile.

Nel caso della gestione dell'editing del documento, le API relative ai servizi esposti dal server sono contenute nel file *document-router.ts*, il quale sfrutta i servizi contenuti nel file *document-service.ts* per riuscire a soddisfare le richieste relative alle rotte per la gestione del documento e del planner associato ad esso. Sia i servizi di *document-service.ts*, che le stesse route fanno uso dell'interfaccia **Planner, Document** e **BestDay** al fine di rappresentare correttamente il dato ottenuto dall'interrogazione del database mediante le query definite in *PlannerQuery.ts* e *DocumentQuery.ts*.

3.1.4 Struttura del modulo client

Il modulo client contiene tutto ciò che rappresenta il frontend dell'applicazione. In esso è possibile individuare due directory (Fig. 8):

- **public**, la quale contiene gli script javascript necessari a gestire la logica lato client delle diverse pagine dell'applicazione e l'invio delle richieste HTTP al server; per svolgere tali operazioni si è fatto uso della libreria `jQuery` poichè rende più agevole la programmazione lato client e permette di realizzare un sito web dinamico e reattivo alle operazioni dell'utente;

- **view**, la quale contiene la definizione HTML delle pagine dell'applicazione.

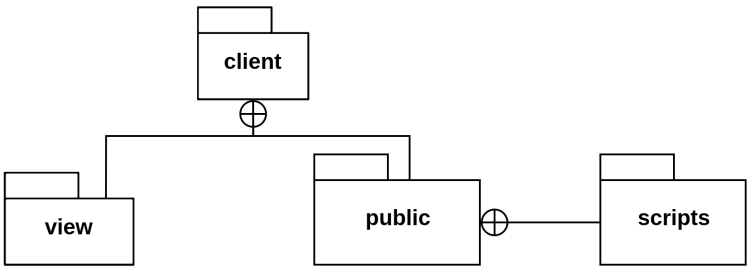


Figura 8: Diagramma dei packages del modulo client

Il client web realizzato aderisce al modello a quattro livelli, dove sono previsti: un livello HTML, un livello di presentation, un livello di application logic e infine un livello di storage. Nel progetto il client detiene i livelli HTML e presentation, mentre il server, i livelli di application logic e storage, dove per application logic intendiamo la gestione e l'esecuzione delle operazioni richieste dall'utente. Il client, infatti, non esegue direttamente queste operazioni, ma invia al server tutti i dati necessari per poterle eseguire.

3.2 Comportamento

3.2.1 Comportamento del server

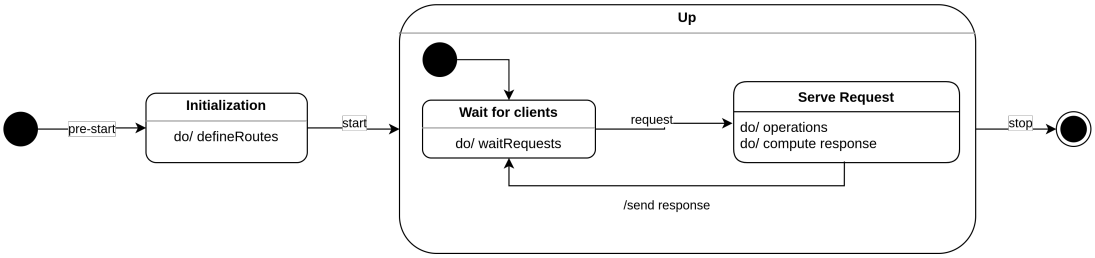


Figura 9: Diagramma degli stati del server

Il comportamento generale del server può essere descritto mediante il diagramma degli stati visibile in figura 9: il server dopo aver completato la fase di pre-start in cui vengono inizializzate le variabili d'ambiente entra nello stato di `inizializzazione`. Durante questo stato definisce le route che saranno messe a disposizione dei client; dopo di che passa nello stato `Up`. All'interno di questo stato il server rimane in attesa (stato `wait for clients`) della richiesta da parte di uno dei client che si sono connessi. All'arrivo della richiesta per servirla, entra in `Serve Request`. Qui il server effettua delle operazioni per elaborare la richiesta al fine di generare la risposta, la quale una volta pronta verrà

inviata al client. Inviata la risposta torna nello stato di attesa. Il server termina una volta che, dallo stato `Up` riceve un evento di `stop`. É opportuno far notare che dopo essere entrati nello stato `Up`, il server è in grado di gestire più richieste contemporaneamente, provenienti anche da clients diversi; per ogni richiesta il comportamento mantenuto dal server è lo stesso, descritto precedentemente.

Sebbene questo sia il comportamento generale del server, si preferisce effettuare una nota riguardante il servizio offerto da quest'ultimo. Un server può essere di due tipologie: *stateful* o *stateless*. Nel primo caso il server è con memoria di stato, per cui risponde alle richieste dei client in modo coerente dipendentemente dallo stato attuale in cui si trova, mentre nel secondo caso il server è senza memoria di stato, ossia tratta ogni richiesta come una transazione indipendente rispetto ogni richiesta precedente, per cui il client dovrà inviare all'interno della richiesta tutte le informazioni necessarie al server per poter elaborare la risposta.

Sulla base di queste definizioni, si è deciso di realizzare un server il quale offre un servizio di tipo stateful, salvando lo stato all'interno di un database, al fine di ricercare un'implementazione senza memoria di stato, ma il quale, per come implementato risulterà stateless. Per fare questo ogni richiesta effettuata al server genererà un output che non sarà più solo in funzione delle informazioni contenute nel client, caso normale di server stateless, ma sarà anche in funzione dello stato del database con il quale il server comunica.

3.2.2 Comportamento del client

Le principali funzioni compiute lato client partono da un input dell'utente. Infatti è quest'ultimo che mediante lo svolgimento di operazioni sulla piattaforma, determinerà lo stato del client. Le stesse operazioni sono quelle che scatenano gli eventi per cui il client effettua le richieste HTTP al server, come per esempio il submit delle credenziali fa si che il client contatti il server per l'autenticazione e sulla base di questa ottenga l'accesso all'applicazione.

Poichè, come detto in precedenza, il server implementato è stateless, il client, durante tutta l'esecuzione dell'applicazione, mantiene salvate le informazioni necessarie per lo svolgimento delle diverse operazioni, come, l'email dell'utente autenticato ed autorizzato e l'identificativo del documento scelto per la modifica.

Il comportamento del client, descritto in figura 10, risulta il seguente: una volta avviata l'applicazione questa entra nello stato di `Start`, da qui non appena l'utente effettua il login o la registrazione si passa nello stato `All documents` durante il quale verranno mostrati i documenti relativi all'utente loggato e si rimarrà in ascolto degli aggiornamenti alla lista dei documenti ai quali l'utente può accedere. Tale lista verrà aggiornata a seguito dell'evento `add` che viene scatenato una volta che un utente crea un nuovo documento ed indica l'utente loggato come partecipante ad esso. Dallo stato `All documents` se l'utente clicca sul `new document` si passa nello stato in cui il client mantiene tutte le funzionalità del precedente, in più mostrando anche la possibilità di creare il nuovo documento. Da qui si può tornare allo stato precedente, sia a seguito dell'azione di `add` che di click di `new document`. Sia da `All documents` che `New Doc` è possibile, a segui-

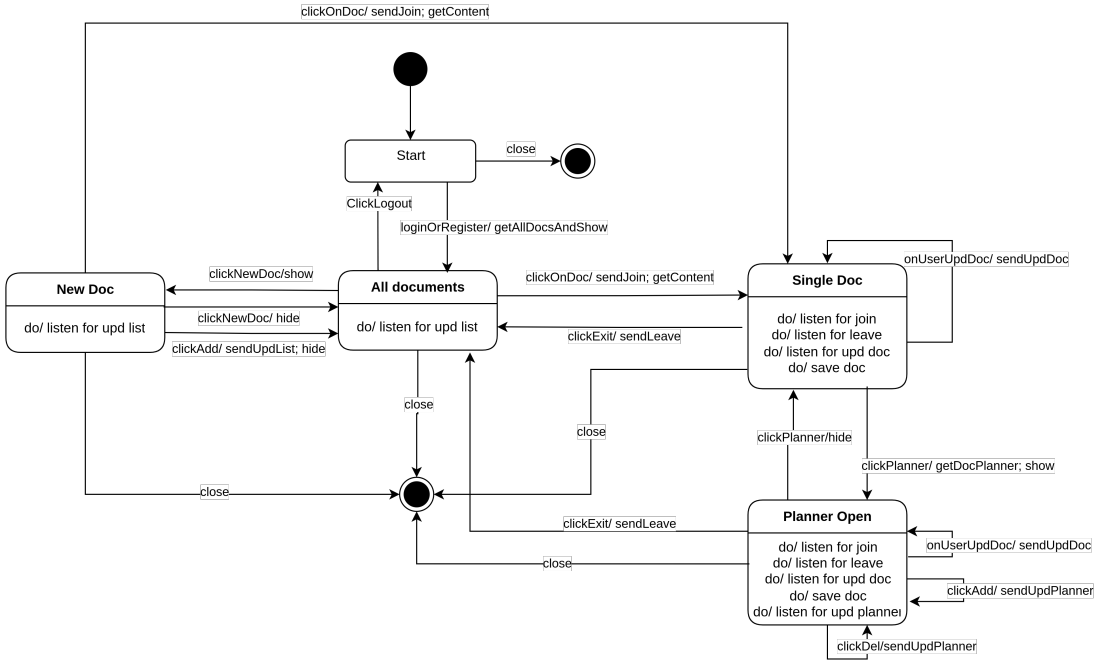


Figura 10: Diagramma degli stati del client

to della selezione da parte dell'utente di uno dei documenti, passare allo stato `Single Doc`. Qui il client rimane in ascolto degli eventi di join e leave, scatenati rispettivamente quando si entra ed esce dallo stato `Single Doc` per tornare in `All documents`. Inoltre rimane anche in attesa degli eventi di aggiornamento del documento, scatenati dalla modifica del documento da parte degli utenti. Al click dell'utente su `planner` si passa allo stato in cui questo è visibile. In tale stato oltre alle operazioni precedenti si rimane in attesa degli aggiornamenti al planner. Al nuovo click `planner` si torna allo stato precedente. Da tutti gli stati è possibile attraverso la chiusura dell'applicazione, ossia della pagina, terminarla, mentre l'evento di `logout` è permesso solo quando ci si trova in `All documents` e fa si che si torni allo stato iniziale di `Start`.

3.3 Interazioni

In questa sezione verranno spiegati come i vari componenti del sistema, client, server e database, interagiscono gli uni con gli altri.

Le interazioni, come si può vedere nelle figure 11, 12 e 13 partono, sempre dal client a seguito di una richiesta HTTP, questa verrà presa in carico dal server, il quale, dopo aver controllato gli eventuali dati passati, comunica con il database per effettuare le operazioni di inserimento, selezione ed aggiornamento sulla base dei dati passati dal client. Una volta completata l'operazione il server comunica al client la risposta, che può essere: di errore qualora qualcosa durante una delle operazioni svolte non sia andata a buon

fine oppure di successo con, come nel caso della richiesta dell'elenco degli iscritti alla piattaforma, o senza, come nel caso del salvataggio del documento, restituire dei dati.

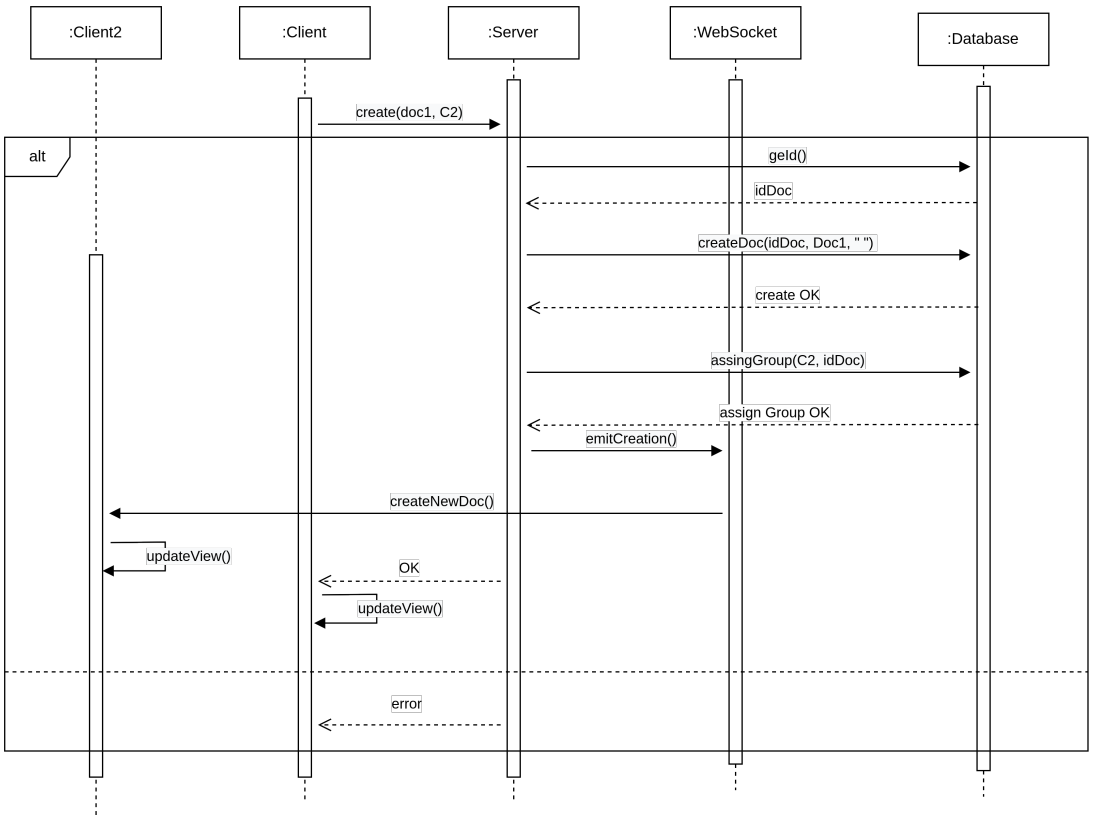


Figura 11: Diagramma di sequenza per la creazione di un nuovo documento

Di seguito (Fig.11) viene mostrato un esempio di interazione senza invio di dati nella risposta, è il caso della creazione di un nuovo documento. Il processo parte dal client il quale attraverso la richiesta `HTTP POST` richiede la creazione del documento, inviando al server le informazioni relative al titolo di quest'ultimo e alla lista di utenti che potranno accedere al documento, nel diagramma rappresentato da client2. Il server, dopo aver ottenuto l'identificativo da assegnare al documento, effettua le operazioni necessarie a creare il documento e, se queste vanno a buon fine, procede con l'assegnare ad ogni utente elencato (client2) la possibilità di accedere al documento. Fatto ciò procede, mediante la webSocket ad informare gli utenti abilitati all'accesso (client2) che un nuovo documento è stato creato. Tali client una volta ricevuta la notifica procederanno all'aggiornamento dell'interfaccia al fine di aggiungere il nuovo documento alla lista di quelli che possono modificare. Se anche quest'ultima operazione di notifica di creazione va a buon fine il server restituisce al client, creatore del documento, una risposta positiva, altrimenti, qualora si verifichi un errore durante una delle fasi del processo di creazione, restituisce al client un errore.

Un esempio di richiesta con la presenza dei dati all'interno della risposta è invece rappresentata dalla richiesta `HTTP GET` per richiedere la lista degli utenti iscritti all'applicazione (Fig.12). L'unica differenza rispetto al caso precedente è relativa al fatto che dall'interazione con il database il server ottiene una risposta contenente i dati. Una volta ricevuti li elabora in modo da formattare la risposta che poi invierà al client. Il client, ricevuta la risposta, effettuerà alcune operazioni come ad esempio l´aggiornamento dell'interfaccia per mostrare i dati ricevuti.

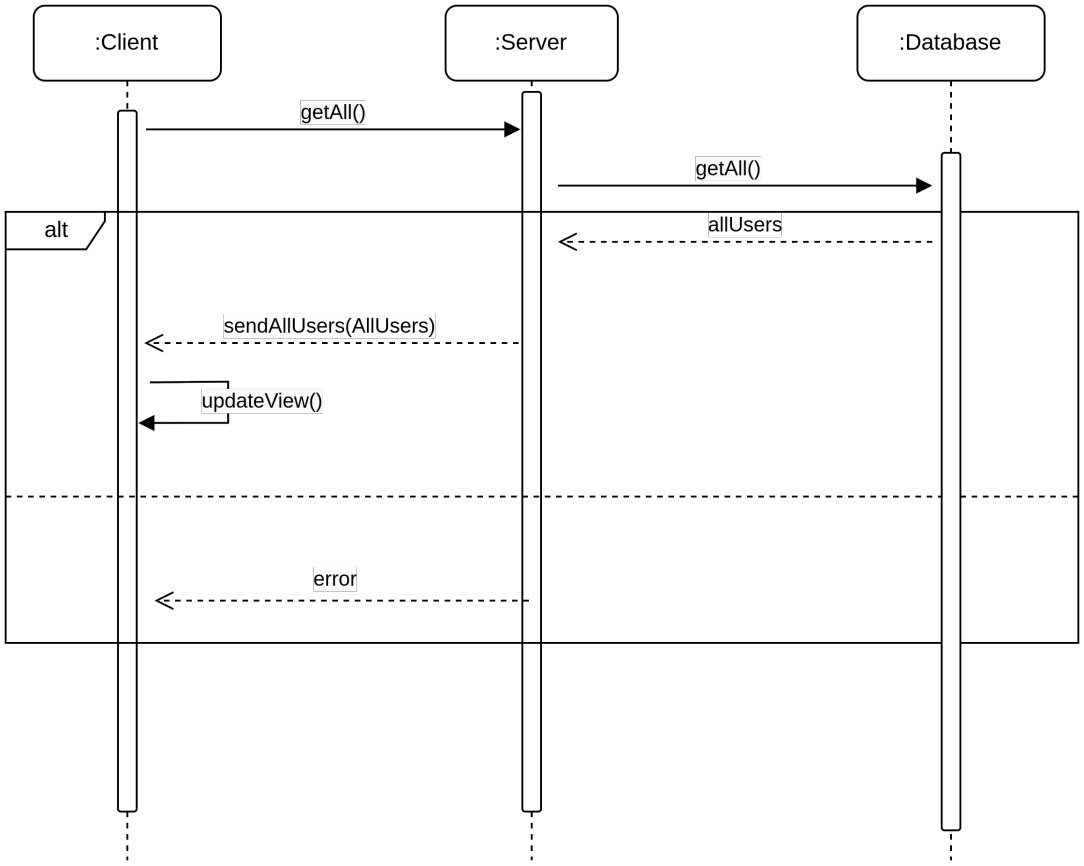


Figura 12: Diagramma di sequenza per ottenere tutti gli utenti iscritti

Una combinazione delle due tipologie di richieste è possibile visualizzarla nella figura 13, in cui viene mostrata l'interazione tra le diverse componenti del sistema durante una sessione di editing di un documento.

L'operazione, come già detto, viene inizializzata dal client una volta che seleziona il documento da iniziare ad editare. A seguito di questa azione il client comunica con il server richiedendo mediante la `get(idDoc)` le informazioni relative al documento selezionato. Tali informazioni verranno inviate dal server al client all'interno della risposta, solo dopo averle ottenute dall'interrogazione del database. Una volta ottenute le informazioni, il client è pronto a modificare il documento e a ricevere da gli altri client, mediante la

webSocket, gli aggiornamenti relativi al contenuto dello stesso documento. Ogni aggiornamento ricevuto scatena sul client l'operazione di aggiornamento del contenuto e della posizione del cursore dell'utente che ha effettuato la modifica.

Periodicamente il client invierà una richiesta HTTP al server, `save(idDoc, content)`, al fine di salvare il contenuto del documento. Tale operazione ha solo l'effetto di aggiornare il database con il nuovo contenuto del documento, per cui a seguito di tale richiesta, il client non aspetta alcun contenuto all'interno della risposta, se non l'OK di relativo al successo dell'operazione.

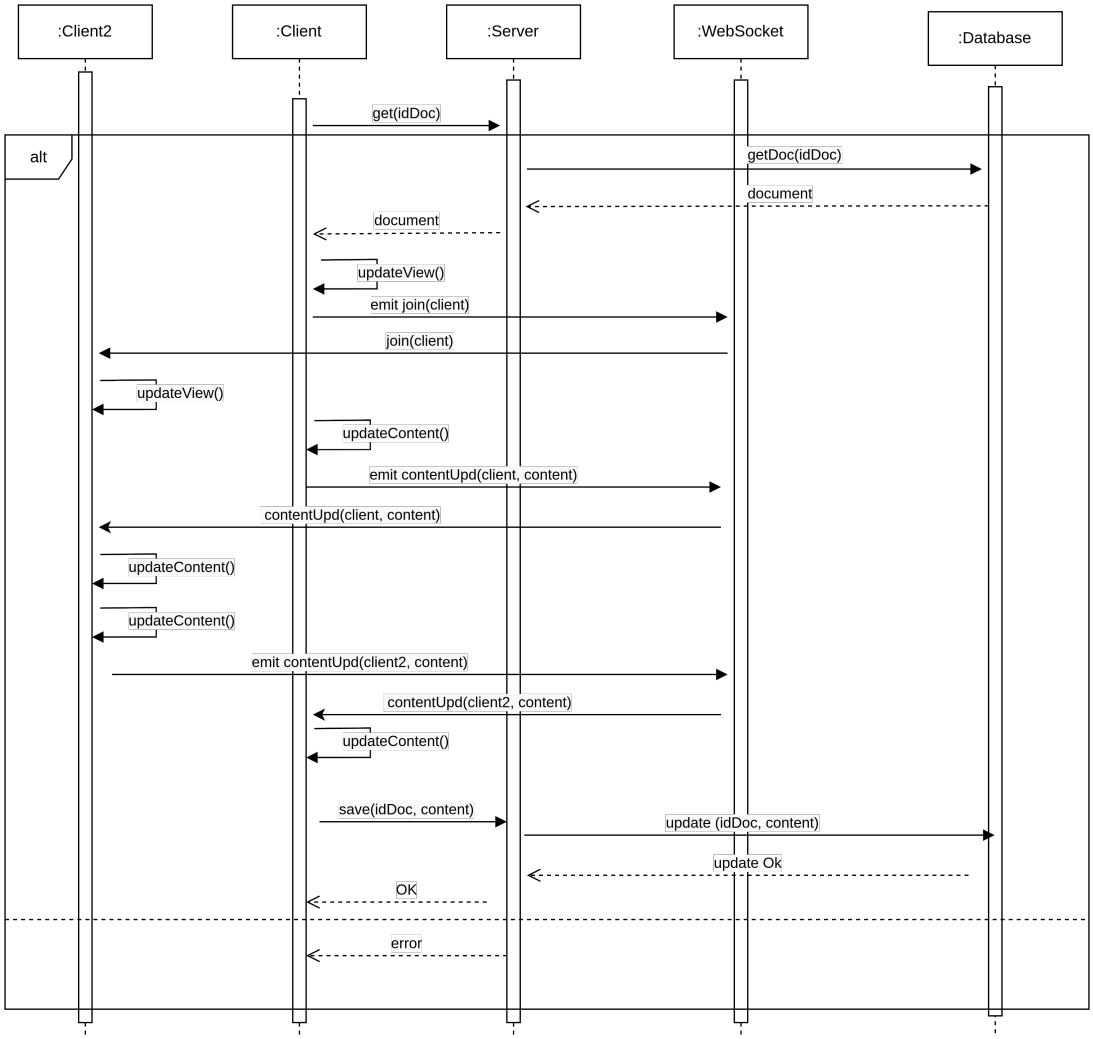


Figura 13: Diagramma di sequenza per l'editing di un documento

4 Dettagli implementativi

Nel seguente capitolo, verranno discusse le tecnologie utilizzate e alcuni dettagli implementativi delle diverse entità che sono state realizzate, cercando di spiegare, dove necessario, come mai sono state effettuate determinate scelte.

4.1 Tecnologie utilizzate

Linguaggi

- **TypeScript** è stato scelto per lo sviluppo della parte backend dell'applicazione.

  TypeScript consiste in un superset di JavaScript, che aggiunge tipi, classi, interfacce e moduli opzionali al JavaScript tradizionale. Si tratta quindi di una estensione di JavaScript. La sua scelta è dovuta a quello che rappresenta il suo principale vantaggio, ossia il supporto alla tipizzazione delle variabili insieme a una migliore gestione del paradigma object oriented. Inoltre, essendo TypeScript compilato in JavaScript, è possibile configurare il compilatore in modo che generi codice compatibile con qualsiasi browser, o conforme ad un particolare standard.

- **JavaScript** è stato scelto per lo sviluppo della parte logica di gestione del client.

  JavaScript consiste in un linguaggio di programmazione multi paradigma orientato agli eventi, comunemente utilizzato per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

- **HTML**: utilizzato per la creazione dell'interfaccia grafica.

  HTML rappresenta un linguaggio di markup usato per il disaccoppiamento della struttura logica di una pagina web dalla sua formattazione. Quest'ultima infatti viene gestita nel progetto mediante il framework *Bootstrap*.

Frameworks

- **Express**: come riporta la documentazione[2] è un framework per applicazioni web Node.js flessibile e leggero che fornisce una serie di funzioni avanzate per le applicazioni web e per dispositivi mobili.

  La scelta del suo utilizzo è dovuta al fatto che consente di creare potenti API di routing e di impostare middleware per rispondere alle richieste HTTP, fornendo semplici meccanismi di debugging e una rapida integrazione con vari motori di templating.

- **Bootstrap** è un framework CSS open-source utilizzato per lo sviluppo di pagine web responsive e di tipo mobile-first. Bootstrap è una raccolta di strumenti grafici (HTML), stilistici (CSS) e di impaginazione (JavaScript) che permettono quindi di avere a disposizione una gran quantità di funzionalità e di stili modificabili.

Nel progetto è stato impiegato nella versione 5 per la gestione degli aspetti presentazionali del client web.

- **Jest**: è un framework di test JavaScript. Come si può leggere nella documentazione[4] è progettato per garantire la correttezza di qualsiasi codebase JavaScript. Consente infatti di scrivere test con un'API accessibile, familiare e ricca di funzionalità che ti fornisce in modo rapido i risultati. Jest permette inoltre di ottenere la *coverage* del progetto, ossia un'indicazione di quanto del codice del progetto è coperto dai test.

  Tale framework, come si può intuire, è stato utilizzato per testare alcune funzionalità dell'applicazione, le quali verranno approfondite nella sezione 5.

Tools

- **Docker** come riporta la documentazione[1], è una piattaforma open-source per lo sviluppo, il rilascio e l'esecuzione di applicazioni. Esso consente di separare le applicazioni dall'infrastruttura dell'host, in modo da poter fornire il software rapidamente. Nel progetto è stato utilizzato sia per dockerizzare l'intera applicazione che solo per il database, come descritto nella sezione 4.4

- **Node.js** è un runtime JavaScript costruito sul motore JavaScript V8 di Chrome. Questo permette l'esecuzione di codice JavaScript lato server, in modo tale da poter usare lo stesso linguaggio di programmazione sia lato client che lato server.

  Node.js è orientato agli eventi asincroni, per cui esiste un *event loop single threaded*, il quale fa si che ad ogni iterazione venga verificata la presenza di nuovi eventi, se ciò si verifica i rispettivi handler vengono eseguiti, altrimenti Node.js rimane inattivo[5].

  All'interno del progetto è stato utilizzato per lo sviluppo del backend, in quanto, come detto, ha permesso l'utilizzo di codice TypeScript per la sua realizzazione.

- **npm** è un gestore di pacchetti per il linguaggio di programmazione JavaScript. Npm è il gestore di pacchetti predefinito per l'ambiente di runtime JavaScript Node.js.

  Nel progetto viene utilizzato anche come strumento per la *build automation*.

Altre tecnologie

- **jQuery**: è una libreria JavaScript open-source, che nasce con l'obiettivo di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML.

  Per il progetto si è fatto uso della libreria jQuery per la realizzazione della logica del client web.

- **MySQL**: è un sistema open-source di gestione di database relazionali SQL sviluppato e supportato da Oracle. Tramite SQL è possibile gestire le operazioni di tipo *CRUD* (Create, Read, Update, Delete) su un database. Il DBMS in questione presenta le seguenti caratteristiche: ampia capacità di integrazione con i principali linguaggi di programmazione e ambienti di sviluppo; l'elevata scalabilità, in quanto capace di integrarsi con applicazioni di diverse dimensioni; ottima efficienza nella gestione dei dati. Quest'ultima caratteristica supportata dal suo query engine ad alte prestazioni, alla capacità di inserimento dei dati estremamente veloce e al supporto delle funzioni web specializzate, ha fatto sì che MySQL sia diventato lo standard di fatto per i siti web con volumi di traffico elevati.

  Nel progetto, è stato utilizzato per la gestione del database di riferimento dell'applicazione.

- **WebSocket**: è una tecnologia che consente lo scambio bilaterale sincronizzato tra Client e Server, cosicché mediante questo protocollo anche il Server può inviare comunicazioni al Client. L'utilizzo delle WebSocket è particolarmente consigliato quando è necessaria una forte e ricorrente interazione tra client e server, come nel nostro caso. Il loro utilizzo permette di semplificare notevolmente la realizzazione di applicazioni che forniscono contenuti e giochi in tempo reale.

4.2 Dettagli implementativi Server

Per quanto riguarda il server, si vuole porre l'attenzione su due aspetti principali riscontrati durante lo sviluppo. Il primo è relativo alla gestione del meccanismo di sicurezza *CORS*, messo in campo dal client, il secondo invece è relativo alla gestione di propagazione degli eventi tra gli utenti dell'applicazione, la quale è stata realizzata mediante le librerie *Socket.IO* e *Pusher Channels*. Inoltre tutte le funzionalità sono state arrichite con apposita documentazione `JSDoc`, ed è possibile visualizzare l'API al seguente link: https://app.swaggerhub.com/apis/MariaMengozzi/TextEditor/1.0.0

4.2.1 Gestione Cross Origin Resource Sharing

Il *Cross Origin Resource Sharing* è un meccanismo basato sull'intestazione HTTP che consente a un server di indicare qualsiasi origine (dominio, schema o porta) diversa dalla propria, da cui un browser dovrebbe consentire il caricamento delle risorse. Un altro meccanismo alla base di CORS è quello mediante il quale i browser effettuano una richiesta di "preflight" al server che ospita la risorsa multi-origine, al fine di verificare che il server consenta la richiesta effettiva. Durante questa verifica preliminare, che viene fatta in particolare per i metodi che hanno side-effect sui dati del server, il browser invia le intestazioni al server, le quali indicano il metodo HTTP e le intestazioni che verranno utilizzate nella richiesta effettiva.

All'interno dell'applicazione è stato necessario gestire tale meccanismo affinchè fosse possibile accettare il caricamento delle risorse utilizzate dal client, nello specifico per consentire il caricamento delle librerie Pusher Channel, Quill, Bootstrap, jQuery e Soc-

ket.IO. Per fare ciò si è deciso di utilizzare il middleware `Helmet`[3] configurato come si può vedere nel listato1.

```
1 // Security
2 if (process.env.NODE_ENV === 'production') {
3     app.use(helmet({
4         contentSecurityPolicy: false,
5         crossOriginEmbedderPolicy: false
6     }));
7 }
```

Listing 1: Gestione del CORS

L'opzione `contentSecurityPolicy` settata a `false` serve ad indicare che viene disabilitata la *Content Security Policy*, la quale consiste in un ulteriore livello di sicurezza che aiuta a rilevare e mitigare determinati tipi di attacchi, inclusi *Cross-Site Scripting (XSS)* e attacchi di iniezione di dati. La decisione di disabilitarla è dovuta come detto alla necessità del caricamento e utilizzo da parte del client delle suddette librerie, in particolare è risultato indispensabile al fine di poter utilizzare Socket.IO e Pusher Channel per la comunicazione sincronizzata, la quale sta alla base del funzionamento di tutta l'applicazione.

L'opzione `crossOriginEmbedderPolicy`, settata anch'essa a `false`, serve ad indicare che viene disabilitata tale opzione, il cui compito di base è impedire a un documento di caricare risorse tra origini diverse che non concedono esplicitamente l'autorizzazione del documento. Questo è stato necessario al fine di rendere più semplice il caricamento delle risorse all'interno dei documenti del client.

Tutte le altre opzioni di configurazione del middleware sono lasciate invariate al fine di mantenere un livello minimo di sicurezza.

4.2.2 Gestione della propagazione degli eventi

L'applicazione che si vuole realizzare consiste, come detto, in un text editor real-time, quindi la generazione e propagazione degli eventi è di cruciale importanza al fine di riuscire ad implementarla. Infatti ogni volta che un utente connesso effettua un'operazione di creazione o modifica di un documento o inserimento e cancellazione di una disponibilità all'interno del planner, è necessario che tutti gli altri utenti coinvolti ne siano notificati, in modo tale da poter eseguire le operazioni di aggiornamento della propria pagina.

Per assolvere a questo compito si è deciso di utilizzare due meccanismi di propagazione degli eventi, entrambi basati su *WebSocket*: Socket.IO e Pusher Channel.

Socket.IO Socket.IO è una libreria per la comunicazione a bassa latenza, bidirezionale e basata su eventi tra client e server. Ciò significa che permette la comunicazione sincronizzata, in tempo reale tra client e server.

Socket.IO, come si legge nella documentazione [8], si basa sul protocollo *WebSocket* e fornisce garanzie aggiuntive come il fallback al long polling HTTP o la riconnessione automatica. Socket.IO è formato da due parti: una libreria per il lato client e una libreria

per il lato server e il funzionamento si basa, come detto, sull'invio di eventi. All'interno del progetto, lato server è stata utilizzata per notificare la creazione di nuovi documenti (listato 2) e la gestione delle stanze (*room*) di ogni documento.

Nel caso della gestione della creazione di nuovi documenti, quello che avviene è che ogni utente che accede all'applicazione si connette ad una *socket room*. Una *socket room* consiste in un canale arbitrario a cui le diverse istanze di socket possono unirsi e uscire. Questo meccanismo può essere utilizzato per trasmettere eventi a un sottoinsieme di client, come verrà sfruttato all'interno dell'applicazione.

```
export function emitMessage(
    socket: SocketIO.Socket,
): void {
    // Send a message
    const room = socket.to(socketRoomName);
    room.emit('add-new-doc');
}
```

Listing 2: Propagazione della creazione di un nuovo documento

Il server regista la connessione e la utilizzerà poi per inviare i messaggi a tutti i client partecipanti a quella stanza in base alla politica di notifica. Come si può vedere nel listato 2, il server emette a tutti i client connessi a `SocketRoomName` un messaggio il cui `topic` è `add-new-doc`; i client interessati all'evento relativo alla creazione di un nuovo documento si metteranno in ascolto su quel topic e non appena verranno notificati dell'evento effettueranno le operazioni più opportune a gestirlo.

Un meccanismo similare è stato utilizzato per la gestione delle *room* di ogni documento; in questo caso, le stanze sono univoche per ogni documento.

Come per la notifica della creazione di un nuovo documento, come prima cosa il client effettuerà la connessione (listato 3) ad una *socket room*.

```
router.get(p.connect, async (req: Request, res: Response) => {
    // check params ...
    documentService.connectSocketToRm(socket, roomName);
    documentService.emitJoinMessage(socket, roomName, user);

    var clients = await io.in(roomName).fetchSockets()
    const connectClients: Array<string> = Array()
    clients.forEach(function(data, counter){
        let ca = (data.handshake.headers.cookie as any).split(';');
        for (let i = 0; i < ca.length; i++) {
            let c = ca[i];
            while (c.charAt(0) == ' ') {
                c = c.substring(1);
            }
            if (c.indexOf("user=") == 0) {
                connectClients.push(c.substring("user=".length, c.length)
    );
            }
        }
    });
    return res.status(OK).json({
```

```
21          users: connectClients
22      });
23  });
```

Listing 3: Connessione alla room di uno specifico documento

La connessione comporta la sottoscrizione a due eventi: `join` e `leave`. Questi eventi sono fondamentali per notificare ai client quando un altro client è entrato ed uscito dalla stanza a cui sono connessi e a seguito di questa notifica effettueranno le operazioni più opportune per gestire l'evento, come ad esempio aggiornare l'elenco dei nominativi dei client connessi.

Una volta fatta la sottoscrizione il server si occupa di collezionare tutti gli utenti attualmente connessi alla stanza in modo da poterli poi includere nella risposta che invierà al client una volta che il processo di connessione sarà terminato.

La stessa connessione alla singola stanza è stata impiegata anche per notificare quando un utente aggiorna la tabella contenente le informazioni relative al planner di un documento, ossia a seguito dell'inserimento o cancellazione di una disponibilità all'interno del planner. La notifica avviene emettendo l'evento sul canale `updPlanner`, il quale verrà sottoscritto dai client interessati.

Pusher Channel Pusher Channels è una hosted API, che, come si legge nella documentazione [6], fornisce comunicazioni in tempo reale tra server, app e dispositivi. Pusher Channels utilizza *WebSocket* e HTTP e fornisce fallback per i dispositivi che non supportano *WebSocket*, questo fa si che possa funzionare ovunque. La libreria mette a disposizione degli utilizzatori un modello di tipo *publish/subscribe*, per cui chi ha bisogno di informazioni si mette in ascolto di queste effettuando la *subscribe*, mentre i produttori di informazioni le propagheranno mediante la *publish*.

All'interno dell'applicazione la libreria è stata impiegata per rendere l'editor di testo collaborativo e quindi propagare le modifiche apportate al documento dai diversi utenti. Lato server l'unica nota importante riguarda il fatto di dover, oltre che connettersi alla piattaforma con i parametri relativi all'account creato, definire la route `/pusher/auth`. Questa definizione è necessaria al fine di poter ottenere il token di autenticazione al server Pusher da inviare poi al client. Infatti quando il client web chiama il metodo di accesso su una connessione stabilita, la libreria client richiede un token di autenticazione al server, il cui endpoint di default è proprio rappresentato da `/pusher/auth`.

La scelta di utilizzare Pusher Channel invece di Socket.IO, per la gestione della propagazione dei cambiamenti che avvengono nel documento quando un utente è in modalità di editing, è dovuta alla sua facilità d'uso e scalabilità, la presenza di canali privati e pubblici e automazione basata su eventi.

4.3 Dettagli implementativi Client

Per quanto riguarda il client, si vuole porre l'attenzione sulla gestione dell'editing del documento, la quale risulta un punto cruciale per la realizzazione dell'applicazione.

Per assolvere a tale compito sono state utilizzate due librerie: Pusher Channels e Quill.

Pusher Channel, come detto nel paragrafo 4.2.2, si tratta di una hosted API che è stata impiegata per la propagazione e ricezione dell'evento di aggiornamento del documento da parte di un client agli altri che sono attualmente connessi ad un determinato canale privato che, nel caso dell'applicazione, è identificato dal topic `private-roomX`, dove `X` rappresenta l'identificativo del documento che si sta modificando. Quill, invece, è un editor WYSIWYG gratuito e open-source creato per il Web moderno[7]. L'editor è stato scelto in quanto supporta la presenza di più cursori, feature risultata indispensabile al fine di assolvere il requisito relativo al mostrare il cursore degli altri utenti mentre modificano il documento, ed inoltre grazie alla libreria **Quill-delta** offre un meccanismo semplice ed efficace per la gestione di editor collaborativi.

Quill si basa sostanzialmente sul concetto di *Delta*, il quale, come definito nella documentazione[7], è un formato semplice, ma espressivo, ossia il JSON, che può essere utilizzato per descrivere i contenuti e le modifiche di Quill. Sostanzialmente un *Delta* è costituito da un array di operazioni, che descrivono le modifiche a un documento: inserimento (`insert`), cancellazione (`delete`) e conservazione (`retain`).

La libreria Quill per effettuare gli aggiornamenti al testo, mediante la `updateContent`, sfrutta i metodi messi a disposizione da `quill-delta`, in particolare a seguito dell'evento di aggiornamento opera come segue: mantiene invariato il testo mediante la `retain` fino al punto in cui non è avvenuta la modifica, poi sovrascrive il testo contenuto nel range modificato sostituendo quello precedente con le differenze individuate comparando il vecchio testo e quello nuovo nello stesso range, ed infine lo concatena alla parte di testo rimasta invariata. È proprio l'aggiornamento che viene effettuato a seguito dell'evento `text-change`, generato ogni volta che si richiama esplicitamente l'`updateContent` o che il cursore che detiene la *authorship* del documento lo modifica, che fa sì che sia possibile lavorare in modo collaborativo all'interno del documento.

```
1 quill.on('text-change', function (delta, source) {
2             if (source === 'user') {
3                 privateChannel.trigger('client-text-edit', { delta: delta
      , source: loggedUser, selection: quill.getSelection() });
4                 cursorManager.setCursor(loggedUser, quill.getSelection().
      end);
5                 // ...
6             }
7         });
```

Listing 4: Handler per l'evento `text-change`

Per quanto riguarda la visualizzazione dei cursori, questa risulta essere un aspetto puramente grafico, ottenuto inviando all'atto della modifica (listato 4), attraverso *pusher*, quindi mediante webSocket, oltre alle informazioni relative all'aggiornamento del contenuto, anche quelle relative alla posizione e l'identificativo del cursore che detiene la *authorship*, ossia il cursore dell'utente loggato. Il client che si metterà in ascolto degli eventi sul documento (listato 5), non appena sarà notificato della modifica da parte di un altro client, aggiornerà il contenuto, come descritto precedentemente, e sposterà mediante il `moveCursor` il cursore dell'utente che ha effettuato l'aggiornamento, infatti ogni utente connesso alla pagina relativa all'editing di un documento possiede un cur-

sore identificato dal proprio nominativo, che lo rappresenterà per tutta la durata della sessione di lavoro.

```
privateChannel.bind('client-text-edit', function (res) {
    if (setUpFinished) {
        quill.updateContents(res.delta)
        cursorManager.moveCursor(res.source, res.selection.end)
    } else {
        privateChannel.trigger('client-get-content', {})
    }
});
```

Listing 5: Bind del canale privato pusher all'evento di *client-text-edit*

4.4 Docker

In questa sezione verranno descritti i dettagli implementativi relativi alla dockerizzazione del progetto nel suo complesso, al fine di poterne effettuare il deploy su qualunque tipo di dispositivo.

Per prima cosa è stato realizzato un `Dockerfile` nel quale sono definiti un elenco di comandi che Docker utilizza per configurare l'ambiente dell'applicazione Node.js. Dopodiché è stato creato un file `docker-compose` il quale si occupa di istanziare due servizi, ognuno in un container separato: il primo è quello relativo al database (`mysqldb`), il secondo è invece relativo all'applicazione Node.js creata (`app`).

Come si può vedere nel listato 6 prima di tutto viene creato il servizio riguardante il database a partire dall'immagine docker `mysqldb`. Relativamente a questo servizio viene definita la politica di riavvio del servizio (`restart: always`) in modo tale che venga riavviato ogni volta che il container viene stoppato, inoltre vengono definite le configurazioni del database a partire dal valore contenuto nelle variabili d'ambiente, precedentemente definite; infine viene definita la cartella in cui verranno memorizzati i dati cosicché una volta riavviato il container questi siano ancora disponibili.

Finita la configurazione del servizio del database si è passati a definire quello dell'applicazione realizzata, nello specifico è stata definita la dipendenza tra i due servizi mediante `depends_on: mysqldb`, la quale ha permesso di specificare che il servizio `mysqldb` deve necessariamente essere avviato prima del servizio `app`.

```
version: '3.9'
services:
  mysqldb:
    image: mysql:5.7
    restart: always
    env_file: ./.env
    environment:
      - MYSQL_ROOT_PASSWORD=$MYSQL_ROOT_PASSWORD
      - MYSQL_DATABASE=$MYSQL_DATABASE
    ports:
      - $MYSQL_LOCAL_PORT:$MYSQL_DOCKER_PORT
    volumes:
      - db:/var/lib/mysql
```

```
14    app:
15      depends_on:
16        - mysqldb
17      build:
18        context: ./texteditor
19        args:
20          - var=start
21      restart: always
22      env_file: ./.env
23      ports:
24        - $NODE_LOCAL_PORT:$NODE_DOCKER_PORT
25      environment:
26        - DB_HOST=mysqldb
27        - DB_USER=$MYSQL_USERNAME
28        - DB_PASSWORD=$MYSQL_ROOT_PASSWORD
29        - DB_NAME=$MYSQL_DATABASE
30        - DB_PORT=$MYSQL_DOCKER_PORT
31      stdin_open: true
32      tty: true
33  volumes:
34    db:
```

Listing 6: Contenuto del file docker-compose

I servizi creati hanno inoltre la possibilità di comunicare tra loro in modo automatico grazie alla rete di default messa a disposizione da Docker Compose, infatti come definito nella documentazione:

> "By default Compose sets up a single network for your app. Each container for a service joins the default network and is both reachable by other containers on that network, and discoverable by them at a hostname identical to the container name.[1]"

Il che significa che qualunque container è in grado di comunicare con gli altri sulla rete creata.

5 Autovalutazione/Validazione

Per verificare l'efficacia del progetto realizzato, come detto nella sezione 1.2, si è proceduto a testare le diverse componenti del sistema sia mediante test automatici, che tramite test manuali; nelle seguenti sezioni verranno definiti con maggiore dettaglio i test realizzati.

Il testing effettuato si poneva come ulteriore scopo quello di verificare il corretto comportamento dell'applicazione e soddisfacimento dei requisiti, indicati nella sezione 2.

5.1 Testing

Durante lo sviluppo si è deciso di procedere effettuando sia alcuni test seguendo l'approccio *TDD (Test Driven Development)*, ossia sviluppando codice che deve superare

unit-test precedentemente definiti, assicurando così che il prodotto soddisfi sempre i requisiti del progetto, sia attraverso l'utilizzo dell'applicazione, anch'esso durante tutto il ciclo di vita del progetto, per garantire il soddisfacimento dei requisiti funzionali.

Principalmente si è adottato quest'ultimo approccio in quanto si è ritenuta necessaria una maggior interazione dell'utente con l'applicazione.

Unit-test Per quanto riguarda gli unit-test, questi sono stati realizzati al fine di testare le operazioni che riguardavano un singolo client. Per la loro realizzazione, come anticipato, si è deciso di adottare il framework di test *Jest*[4], scelto in quanto uno dei maggiori framework di testing JavaScript e TypeScript.

Ogni test è stato eseguito nelle condizioni in cui il database risulti sempre essere resettato, cosicché sia possibile replicare ogni volta lo scenario di partenza.

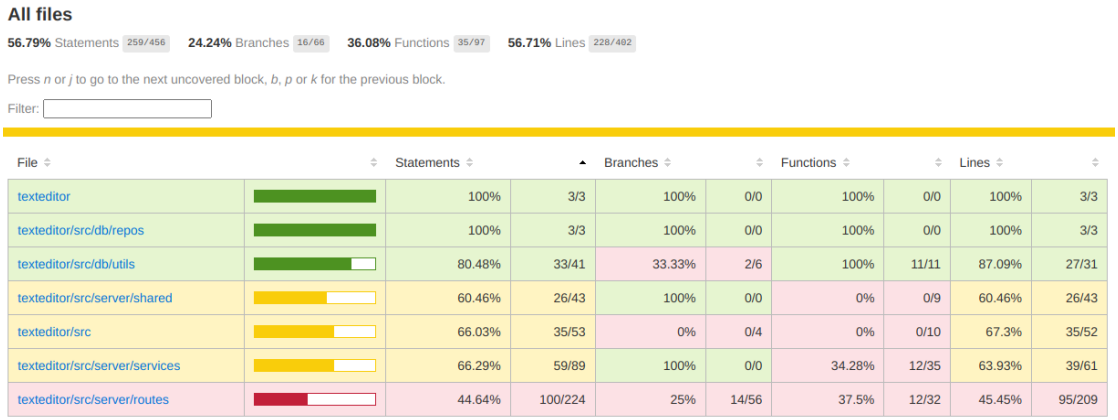
Di seguito vengono descritti i test suddividendoli in base al concetto su cui sono stati effettuati:

- **User**: questi test sono volti a verificare il corretto funzionamento delle API riguardanti la registrazione e login da parte di un utente all'interno della piattaforma. In questo caso sono stati verificati sia i casi di successo, quindi l'inserimento corretto delle credenziali e la registrazione di un nuovo utente, che quelli di fallimento, ossia l'inserimento di credenziali sbagliate o la registrazione di un utente il cui nominativo risulta già essere registrato;

- **Documents**: questi test si occupano di verificare che vengano restituiti al client tutti i documenti a cui l'utente, precedentemente autenticato, ha accesso. Quindi si va a verificare che vengano selezionati non solo i documenti creati dall'utente stesso, ma anche quelli di cui è stato designato come partecipante e che al contrario gli altri di cui non possiede l'accesso non siano presenti tra quelli restituiti;

- **Planner**: questi test riguardano la verifica dei dati relativi al planner di un documento. Nello specifico si va a testare che il corpo della risposta contenga effettivamente il numero di entry che si aspetta, che contenga quelle corrette e verifica, inoltre, che il giorno proposto, designato come *day* risulti essere quello corretto.

Come si può vedere nella figura 14, la coverage ottenuta, ossia un'indicazione di quanto del codice del progetto è coperto dai test, relativa ai test automatici non risulta eccessivamente elevata in quanto molte delle funzionalità sono state verificate mediante test manuali.



All files

56.79% Statements 259/456 **24.24%** Branches 16/66 **36.08%** Functions 35/97 **56.71%** Lines 228/402

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
texteditor		100%	3/3	100%	0/0	100%	0/0	100%	3/3
texteditor/src/db/repos		100%	3/3	100%	0/0	100%	0/0	100%	3/3
texteditor/src/db/utils		80.48%	33/41	33.33%	2/6	100%	11/11	87.09%	27/31
texteditor/src/server/shared		60.46%	26/43	100%	0/0	0%	0/9	60.46%	26/43
texteditor/src		66.03%	35/53	0%	0/4	0%	0/10	67.3%	35/52
texteditor/src/server/services		66.29%	59/89	100%	0/0	34.28%	12/35	63.93%	39/61
texteditor/src/server/routes		44.64%	100/224	25%	14/56	37.5%	12/32	45.45%	95/209

Figura 14: Coverage ottenuta sul progetto

Test manuali La maggior parte dei test sono stati svolti manualmente, sfruttando contemporaneamente più device connessi alla stessa rete e mediante l'utilizzo di browser diversi installati su di essi. In particolare per effettuare i test sono stati impiegati i seguenti device:

- PC con sistema operativo Linux
- PC con sistema operativo Windows
- PC con sistema operativo macOS
- iPad di seconda generazione
- Cellulare con sistema operativo Android

Per quanto riguarda invece i browser su cui è stata testata l'applicazione questi sono:

- Chrome
- Firefox
- Brave
- Safari

Nel caso dei browser, affinchè non venissero utilizzati i dati contenuti nella cache, all'avvio di ogni test è stata effettuata una pulizia completa di quest'ultima.

Le diverse pagine del client sono state testate una ad una, definendone prima di tutto il codice HTML e verificandone la loro corretta visualizzazione sul browser. Successivamente sono stati definiti gli elementi grafici, la cui correttezza è stata verificata sempre procedendo all'esecuzione della pagina sul browser e verificando che essa fosse corretta. Infine, si è proceduto a testare la reattività dell'interfaccia grafica e il corretto invio delle

richieste al server, nonché la corretta visualizzazione dei dati ricevuti in risposta tramite la simulazione delle operazioni dell'utente.

Durante lo svolgimento dei test manuali si è verificato quindi che le operazioni di aggiornamento della pagina in real time avvenisse nel modo corretto, ossia che le operazioni venissero propagate ai soli device che dovevano essere coinvolti nella comunicazione, ad esempio se due dispositivi sono connessi alla sessione di lavoro sul documento X e un altro sul documento Y, l'utente connesso ad Y non riceva gli aggiornamenti relativi al documento X. Si è verificato inoltre che le informazioni propagate venissero ricevute ed elaborate correttamente dai singoli dispositivi.

Per il debugging degli elementi grafici e della reattività dell'interfaccia, nonché del corretto invio delle richieste al server, si è fatto utilizzo degli strumenti di ispezione messi a disposizione dal browser stesso, che non solo offrono la possibilità di stoppare l'esecuzione del codice JavaScript mediante l'utilizzo di appositi breakpoint, ma mettono anche a disposizione uno strumento per analizzare i pacchetti inviati e ricevuti dalla rete.

5.2 Modalità di lavoro

Il progetto è stato gestito mediante un apposito repository GitLab nel quale sono presenti due branch principali: il `main` su cui è stata sviluppata la prima struttura del progetto e successivamente aggiornata con quella finale e il branch `feature/deploy` sul quale sono state apportate le modifiche necessarie al fine di completare l'applicazione e inserire tutto ciò che risulta essere necessario per il deploy dell'applicazione.

La realizzazione del progetto è stata gestita definendo prima di tutto i requisiti dell'applicazione, anche con l'aiuto di appositi *mockup* utili a visualizzare il prodotto che si intendeva realizzare. Fatto ciò si è passati a definire l'architettura generale sia dell'applicazione che del database, identificando quali componenti costituivano il sistema e come questi interagivano tra loro.

Una volta identificati tutti gli aspetti che caratterizzavano il sistema si è provveduto all'implementazione vera e propria. Questa è iniziata definendo come primo step le pagine web statiche che avrebbero costituito il sistema, successivamente si è realizzata una prima versione del server, al fine di sviluppare la versione base dell'applicazione, ossia quella che implementasse tutte le funzionalità individuate come obbligatorie. Questa prima versione è stata quindi collegata alle pagine web realizzate cosicchè fosse possibile popolarle con le informazioni ottenute dal server a seguito dell'interazione con il database. Durante questa fase si è anche provveduto alla ricerca e allo studio delle librerie di utility impiegate nello sviluppo, più precisamente *Pusher Channel* e *Quill*, e successivamente alla containerizzazione del progetto.

Terminate le funzionalità base si è poi proceduto alla realizzazione della parte opzionale, nello specifico si è prima di tutto raffinato il meccanismo di creazione di un nuovo documento dando la possibilità di creare dei gruppi di lavoro. Associato a questo si è aggiornata l'API relativa al reperimento dei documenti a cui un utente ha accesso. Successivamente si è passati alla realizzazione del meccanismo necessario alla visualizzazione dei cursori degli atri membri del gruppo di lavoro durante la sessione di modifica del documento.

Infine, è stato implementato il meccanismo relativo al planner, ossia la sua visualizzazione, compilazione e gestione degli eventi associati ad esso.

Durante tutto il progetto, come detto nella sezione precedente, sono stati effettuati i test, manuali e automatici, in modo da verificare il corretto funzionamento delle feature implementate.

6 Istruzioni per il deployment

In questa sezione verranno indicate le istruzioni per il deployment, che è necessario eseguire, per la corretta messa in funzione ed esecuzione del sistema.

6.1 Setup e installazioni di base

Per prima cosa è necessario scaricare il repository da GitLab, che è possibile trovare al seguente link: https://gitlab.com/pika-lab/courses/ds/projects/ds-project-mengozzi-ay2122/-/tree/main, e successivamente aprire il progetto in un editor di testo.

L'applicazione può essere eseguita secondo 3 modalità, le quali verranno dettagliatamente spiegate nei prossimi paragrafi.

Progetto completamente dockerizzato Per eseguire il progetto completamente dockerizzato è necessario aver installato **Docker** (per fare questo seguire le istruzioni al seguente link: https://docs.docker.com/get-docker/) sulla macchina che si sta utilizzando.

Una volta aperto il progetto nell'IDE scelto è necessario creare nella root directory un file `.env` con il seguente contenuto:

```
1  MYSQL_USERNAME=root
2  MYSQL_ROOT_PASSWORD=pass
3  MYSQL_DATABASE=sistemiDistribuiti
4  MYSQL_LOCAL_PORT=3307
5  MYSQL_DOCKER_PORT=3306
6
7  NODE_LOCAL_PORT=8571
8  NODE_DOCKER_PORT=8571
```

Listing 7: Contenuto del file .env

Fatto ciò per eseguire l'applicazione basterà posizionarsi con il terminale nella root directory del progetto ed eseguire il comando `docker compose up --build`, attendere che il container venga inizializzato e successivamente aprire un browser ed andare su:

- "http://localhost:8571/", se si vuole accedere all'applicazione dal dispositivo su cui viene lanciato il container
- "indirizzoIP:8571/", se si desidera accedere da un dispositivo diverso, ma sempre connesso alla rete locale. Dove **indirizzoIp** è l'indirizzo IP della macchina su cui gira il container, ottenibile mediante `ifconfig` o `ipconfig`. Ad esempio se l'indirizzo IP fosse 192.168.3.69 l'indirizzo da digitare sulla browser sarà "192.168.3.69:8571/"

Per il primo utilizzo o per resettare il database "http://localhost:8571/init" o se da un dispositivo diverso da quello su cui viene lanciato il container "indirizzoIP:8571/init" (es: "192.168.3.69:8571/").

Per stoppare l'esecuzione del progetto e quindi del container, fare `ctrl+c` su terminale in cui è in esecuzione docker e attendere che venga eseguito lo stop e successivamente eseguire `docker compose down --rmi all` per rimuovere il container e le immagini create. Infine per rimuovere tutti i volumi `docker volume rm $(docker volume ls -q)`.

Progetto con container mySql Per eseguire il progetto con solo il database dockerizzato è necessario aver installato **Docker** (per fare questo seguire le istruzioni al seguente link: https://docs.docker.com/get-docker/) e **Node.js** (per fare questo seguire le istruzioni al seguente link: https://nodejs.org/it/download/) sulla macchina che si sta utilizzando.

Fatto ciò per eseguire l'applicazione basterà posizionarsi con il terminale nella sottocartella **texteditor** del progetto ed eseguire il comando `npm install`, al fine di installare tutte le dipendenze necessarie per il progetto.

Successivamente sarà necessario modificare il file **texteditor/src/utils/mysqlConnector.ts** cambiando riga 5 come segue:

```
1 import { createPool, Pool, createConnection} from 'mysql';
2 import { DATA_SOURCES, DATA_SOURCES_LOCAL,
      DATA_SOURCES_WITH_MYSQL_CONTAINER } from '../../../dbConfig';
3 import * as fs from 'fs';
4 import logger from 'jet-logger';
5 const dataSource = DATA_SOURCES_WITH_MYSQL_CONTAINER.mySqlDataSource;
6 //...
```

Listing 8: Modifica al file mysqlConnector.ts

Fatto ciò sarà necessario istanziare e avviare il container contenente il database eseguendo, sempre da terminale, il comando:

```
1 docker run -p 3306:3306 --name mysqldb -e MYSQL_ROOT_PASSWORD=pass -e
      MYSQL_DATABASE=sistemiDistribuiti -d mysql:5.7
```

Listing 9: Comando per avviare il container contenente il database

Attendere che il container venga inizializzato ed eseguire, sempre da terminale, il comando `npm start` il quale si occuperà di effettuare la build del progetto ed avviare l'applicazione. Successivamente aprire un browser ed andare su:

- "http://localhost:8571/", se si vuole accedere all'applicazione dal dispositivo su cui viene lanciato il container
- "indirizzoIP:8571/", se si desidera accedere da un dispositivo diverso, ma sempre connesso alla rete locale. Dove **indirizzoIp** è l'indirizzo IP della macchina su cui gira il container, ottenibile mediante `ifconfig` o `ipconfig`. Ad esempio se l'indirizzo IP fosse 192.168.3.69 l'indirizzo da digitare sulla browser sarà "192.168.3.69:8571/"

Per il primo utilizzo o per resettare il database "http://localhost:8571/init" o se da un dispositivo diverso da quello su cui viene lanciato il container "indirizzoIP:8571/init" (es: "192.168.3.69:8571/").

Per stoppare l'esecuzione del progetto e quindi del container, fare `ctrl+c` su terminale in cui è in esecuzione l'applicazione.

Per stoppare e rimuovere il container da terminale eseguire: `docker stop container mysqldb` seguito da `docker rm container mysqldb`. Per cancellare l'immagine mysql eseguire `docker rmi mysql:5.7` ed infine per rimuovere tutti i volumi `docker volume rm $(docker volume ls -q)`.

Progetto completamente in locale Per eseguire il progetto completamente in locale è necessario aver installato **Node.js** (per fare questo seguire le istruzioni al seguente link: https://nodejs.org/it/download/) e **XAMPP** (per fare questo seguire le istruzioni al seguente link: https://www.apachefriends.org/it/index.html) sulla macchina che si sta utilizzando.

Fatto ciò per eseguire l'applicazione basterà posizionarsi con il terminale nella sottocartella **texteditor** del progetto ed eseguire il comando `npm install`, al fine di installare tutte le dipendenze necessarie per il progetto.

Successivamente sarà necessario modificare il file **texteditor/src/utils/mysqlConnector.ts** cambiando riga 5 come segue:

```
1 import { createPool, Pool, createConnection} from 'mysql';
2 import { DATA_SOURCES, DATA_SOURCES_LOCAL,
      DATA_SOURCES_WITH_MYSQL_CONTAINER } from '../../../dbConfig';
3 import * as fs from 'fs';
4 import logger from 'jet-logger';
5 const dataSource = DATA_SOURCES_LOCAL_CONTAINER.mySqlDataSource;
6 //...
```

Listing 10: Modifica al file mysqlConnector.ts

Una volta effettuata la modifica sarà necessario avviare *Apache* e *MySql*, precedentemente configurati come spiegato nella sezione successiva 6.2, attraverso l'interfaccia grafica di XAMPP.

Fatto ciò basterà eseguire da terminale il comando `npm start` il quale si occuperà di effettuare la build del progetto ed avviare l'applicazione. Successivamente aprire un browser ed andare su:

- "http://localhost:8571/", se si vuole accedere all'applicazione dal dispositivo su cui viene lanciato il container
- "indirizzoIP:8571/", se si desidera accedere da un dispositivo diverso, ma sempre connesso alla rete locale. Dove **indirizzoIp** è l'indirizzo IP della macchina su cui gira il container, ottenibile mediante `ifconfig` o `ipconfig`. Ad esempio se l'indirizzo IP fosse 192.168.3.69 l'indirizzo da digitare sulla browser sarà "192.168.3.69:8571/"

Per il primo utilizzo o per resettare il database "http://localhost:8571/init" o se da un dispositivo diverso da quello su cui viene lanciato il container "indirizzoIP:8571/init" (es: "192.168.3.69:8571/").

Per stoppare l'esecuzione del progetto e quindi del container, fare `ctrl+c` su terminale in cui è in esecuzione l'applicazione. Infine stoppare, sempre attraverso l'interfaccia messa a disposizione da XAMPP, *Apache* e *MySQL*.

6.2 Configurazione XAMPP

Qualora si decidesse di eseguire il progetto completamente in locale, come detto, è necessario installare oltre a Node.js, XAMPP, una piattaforma software multipiattaforma e open-source costituita da Apache HTTP Server, il DBMS MariaDB e tutti gli strumenti necessari per utilizzare i linguaggi di programmazione PHP e Perl.

Per farlo è necessario scaricare la piattaforma dal seguente link: https://www.apachefriends.org/it/index.html. Una volta scaricato basterà avviare l'installer e selezionare tutte le configurazioni di default e/o consigliate. Finita l'installazione avviare la piattaforma, attendere qualche istante, dopo di che si presenterà la schermata principale (Fig. 15) in cui è possibile visualizzare i servizi e il loro stato: rosso, o non colorato significa che è spento, verde che è attivo. Nota la schermata principale può cambiare da sistema operativo a sistema operativo, in ogni caso le operazioni da eseguire sono comunque le medesime.

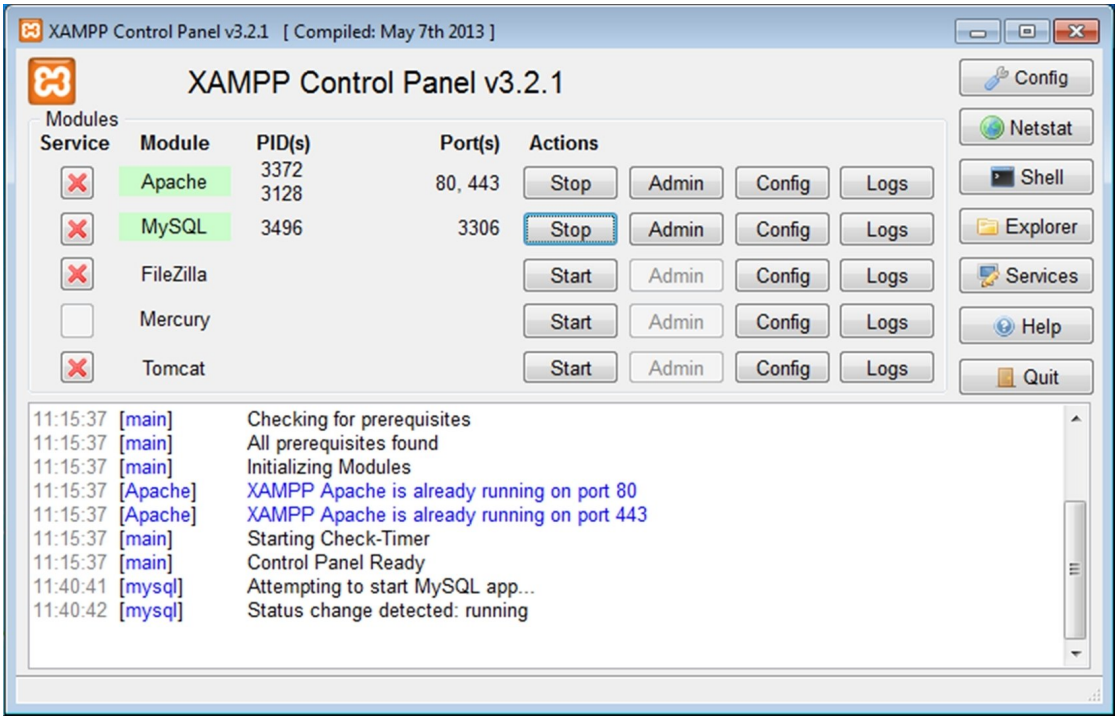


Figura 15: Schermata principale della piattaforma XAMPP in windows

La configurazione di default di **MySQL** dovrebbe essere la seguente:

- port: 3306
- user: root
- password: "" (stringa vuota)

Qualora la configurazione non fosse quella appena descritta sarà necessario modificarla. Per fare ciò bisognerà selezionare **Congif** relativo al servizio **MySQL** e configurare la porta impostandola al valore **3306**; fatto ciò sarà necessario andare a modificare i dati relativi all'utente MySQL, nello specifico utente e password. Per farlo è possibile seguire uno dei tre metodi descritti al seguente link: https://kinsta.com/it/knowledgebase/password-mysql-xampp/.

Una volta terminata la configurazione, per avviare i servizi, in modo da poterli utilizzare durante l'esecuzione dell'applicazione, basterà selezionare il pulsante **start/avvia** posizionato di fianco a **Apache** e **MySQL**.

7 Esempi di utilizzo

Supponiamo di avere un gruppo di persone che decide di adottare l'applicazione come soluzione per condividere documenti testuali tra di loro, in modo da poter collaborare, tutti o solo una sottoparte di essi, alla redazione di tali documenti.

Ogni elemento del gruppo che deciderà di sfruttare l'applicativo dovrà registrarsi alla piattaforma (Fig. 16). Una volta effettuata la registrazione, o a seguito del login (Fig. 16), potrà accedere alla schermata principale (Fig. 17a) dell'applicazione, nella quale potrà vedere elencati tutti i documenti a cui ha accesso, aggiornati in tempo reale. Sempre in questa schermata cliccando il pulsante "New document" all'utente appare il form relativo alla creazione di un nuovo documento (Fig. 17b). Qui potrà specificare il titolo del documento e selezionare la lista di utenti i quali, una volta confermata la creazione mediante il pulsante "Add" o premendo "invio" una volta digitato il titolo, potranno accedere al documento. Per annullare la creazione basterà semplicemente cliccare nuovamente il pulsante "New document" il quale provvederà anche a nascondere il form di creazione.

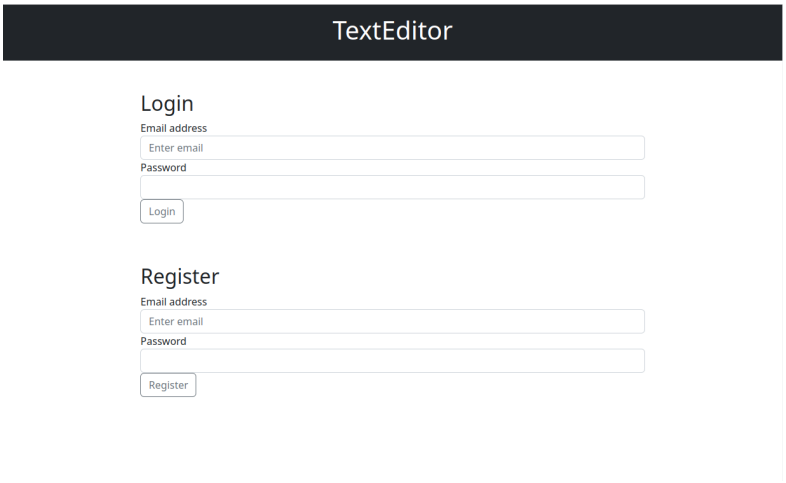
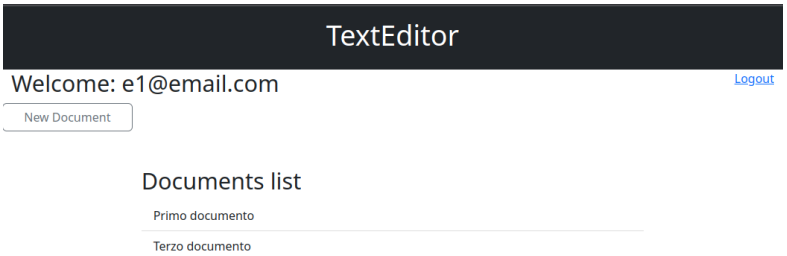
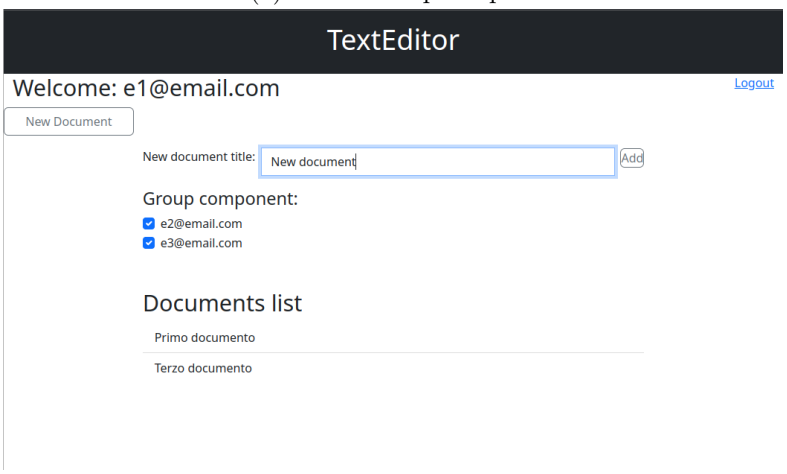


Figura 16: Schermata di login e registrazione

Per accedere al documento di cui desidera modificare il contenuto l'utente non dovrà fare altro che selezionare il titolo tra quelli elencati nella pagina principale. A seguito di questa azione verrà reinderizzato alla pagina di modifica (Fig. 18a). Qui potrà visualizzare a lato la lista, aggiornata in tempo reale, degli utenti attualmente attivi nella sessione di modifica e come elemento centrale il contenuto del documento, anch'esso aggiornato in tempo reale a seguito delle modifiche apportate dagli altri utenti. Per apportare la modifica al documento l'utente dovrà solo posizionare il cursore all'interno dell'area contente il testo del documento e iniziare a digitare, in questo modo vedrà muoversi, insieme a quello degli altri utenti, il proprio cursore segnaposto.



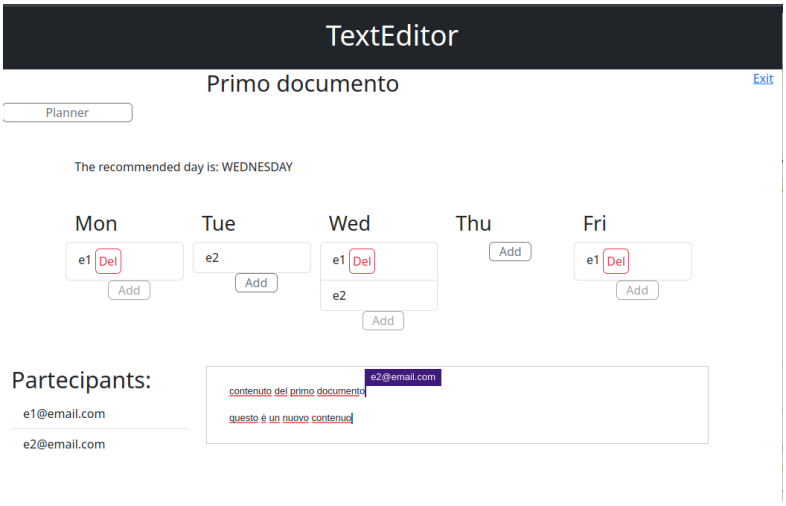
(a) Schermata principale



(b) Form per la creazione di un nuovo documento

Sempre nella stessa schermata l'utente ha la possibilità cliccando sul pulsante "Planner" di visualizzare (Fig. 18b) e nascondere il planner relativo al documento aperto. Qui avrà, per ogni giorno settimanale un'indicazione della disponibilità di ogni componente del gruppo di lavoro relativo al documento, ad un eventuale meeting settimanale. Per aggiungere la propria disponibilità per un determinato giorno all'utente basterà semplicemente cliccare il pulsante "Add" sotto il rispettivo giorno, o al contrario per rimuoverlo cliccare sul pulsante "Del" posto accanto al proprio nome, sempre nel giorno desiderato. Infine, posta sopra la tabella del planner, ha la possibilità di avere un'indicazione del giorno migliore in cui programmare il meeting con gli altri componenti del gruppo.

(a) Schermata di modifica di un documento



(b) Visualizzazione del planner relativo ad un documento

Per uscire dalla sessione di lavoro basterà cliccare semplicemente sul pulsante di "Exit", il quale rimanda alla schermata principale dell'applicazione (Fig. 17a), mentre per uscire completamente dalla piattaforma, una volta raggiunta la schermata principale, basterà cliccare sul pulsante di "Logout"e si verrà reinderizzati alla schermata di login/registrazione (Fig. 16).

8 Conclusioni

Si voleva realizzare un'applicazione che consentisse agli utenti di connettersi da remoto e creare ed editare in modo collaborativo documenti testuali.

Per realizzare questo obiettivo, per prima cosa sono stati definiti i requisiti funzionali e non funzionali che il sistema doveva soddisfare, dopodichè sono state individuate le diverse entità che costituiscono il modello del dominio, per poi passare alla loro implementazione, definendone: struttura, comportamento e interazione.

In particolare il sistema ottenuto si compone di 3 entità fondamentali: un database relazionale, un server ed un client web.

Il server realizzato è di tipo stateless ed è incaricato di soddisfare le diverse richieste provenienti dal client e interagire con il database. Il client, invece, è determinato dalle azioni svolte dall'utente attraverso l'interfaccia web, le quali possono far scaturire l'invio di richieste al server.

Il funzionamento dell'applicazione si basa sul meccanismo delle *Remote Procedure Call*, il quale prevede che il client effettui la richiesta al server e rimanga in attesa della risposta; per quanto riguarda l'interazione è possibile affermare che questa viene sempre iniziata dal client e che a seguito di una richiesta il server possa avere eventualmente più interazioni con il database, concludendo infine con la risposta al client.

Tutte le funzionalità realizzate sono state accuratamente testate mediante l'utilizzo sia di test automatici che, principalmente, con test manuali.

Analizzando quindi il risultato ottenuto posso affermare che il sistema realizzato soddisfa gli obiettivi e i requisiti stabiliti e che complessivamente posso ritenermi soddisfatta del risultato raggiunto.

8.1 Lavori futuri

Possibili funzionalità aggiuntive che possono essere realizzate in futuro, in aggiunta a quelle già presenti, possono essere le seguenti:

- consentire agli utenti creatori dei documenti di aggiungere e rimuovere dinamicamente i permessi di accesso al documento agli altri membri iscritti alla piattaforma;
- introdurre un sistema di notifica capace di ricordare agli utenti i giorni in cui è previsto il meeting;
- realizzare una versione mobile dell'applicazione.

8.2 Cosa è stato appreso

Grazie a questo progetto ho avuto la possibilità di vedere in pratica alcuni aspetti fin ora studiati solo nella teoria. Ho avuto l'occasione di utilizzare Docker per il deploy del progetto, comprendendo maggiormente le potenzialità di questo strumento. Gestire l'aspetto del real-time mi ha permesso di scontrarmi con tutti i problemi legati alla propagazione degli eventi e alla loro risoluzione, inoltre ho avuto l'occasione di studiare una libreria che offre il servizio per aggiornare un contenuto testuale in modo efficiente. Complessivamente sono abbastanza soddisfatta del lavoro svolto e dei risultati finali ottenuti, considerando anche le difficoltà e i problemi riscontrati durante lo sviluppo.

Stylistic Notes

- Il **grassetto** è stato utilizzato per indicare aspetti importanti

- Il *corsivo* è stato utilizzato per indicare aspetti tecnici, o parole inglesi non di uso comune nella lingua italiana
- Il `monospace` è stato utilizzato per indicare aspetti relativi al codice o all'implementazione

Riferimenti bibliografici

[1] Docker. https://www.docker.com/.

[2] Express. https://expressjs.com/it/.

[3] Helmet. https://www.npmjs.com/package/helmet.

[4] Jest. https://jestjs.io/.

[5] Node.js. https://www.nodejs.org/it/.

[6] Pusher. https://pusher.com/channels.

[7] Quill. https://quilljs.com/.

[8] Socket.IO. https://socket.io/docs/v4/.