

Realtime Collaborative Whiteboard

Valerio Di Zio

`valerio.dizio@studio.unibo.it`

Francesco Magnani

`francesco.magnani14@studio.unibo.it`

Thomas Capelli

`thomas.capelli@studio.unibo.it`

October 2023

Il progetto da noi presentato consiste in una lavagna collaborativa real-time dove più utenti possono disegnare e vedere le modifiche apportate dagli altri in tempo reale. Vi è un interfaccia web dove gli utenti, tramite sistema di autenticazione, possono accedere e creare, modificare, eliminare e condividere la propria lavagna virtuale. L'architettura è di tipo client-server, ogni client sfrutta delle API esposte dal server per interagire con il sistema. L'interazione degli utenti con la lavagna è stata realizzata tramite tecnologie che sfruttano protocolli real-time (Socket.IO). Per ridurre la quantità di dati che vengono inviati durante l'interazione real-time, sono state utilizzate tecniche di interpolazione in modo da mostrare l'interazione come più fluida possibile.

1 Obiettivi/Requisiti

Durante la fase di definizione dei requisiti, abbiamo definito diversi obiettivi che possono essere riassunti nei seguenti punti:

- Un utente deve poter creare un account ed effettuare il login nell'applicativo;
- Un utente deve poter modificare impostazioni utente, come: nome, password, ecc;
- Un utente deve poter creare, rinominare ed eliminare lavagne;
- Un utente deve poter condividere la propria lavagna con vari collaboratori;
- Più utenti devono poter disegnare in modo collaborativo sulla stessa lavagna;

- Un utente deve poter visualizzare i collaboratori attualmente connessi alla sessione di disegno;
- Dev'essere possibile personalizzare il tratto sulla lavagna (scegliendone il colore e lo spessore)
- Deve esserci un sistema di notifiche che permetta l'invio e la ricezione di notifiche in tempo reale per informare l'utente di avvenimenti come la condivisione;

1.1 Scenarios

1.1.1 Autenticazione e impostazioni utente

L'applicativo se non si è registrati non offre nessuna funzionalità, se non accedere alla pagina principale in cui vengono mostrati alcuni esempi di utilizzo. È possibile quindi creare un account o effettuare l'accesso nel caso in cui se ne posseda già uno.

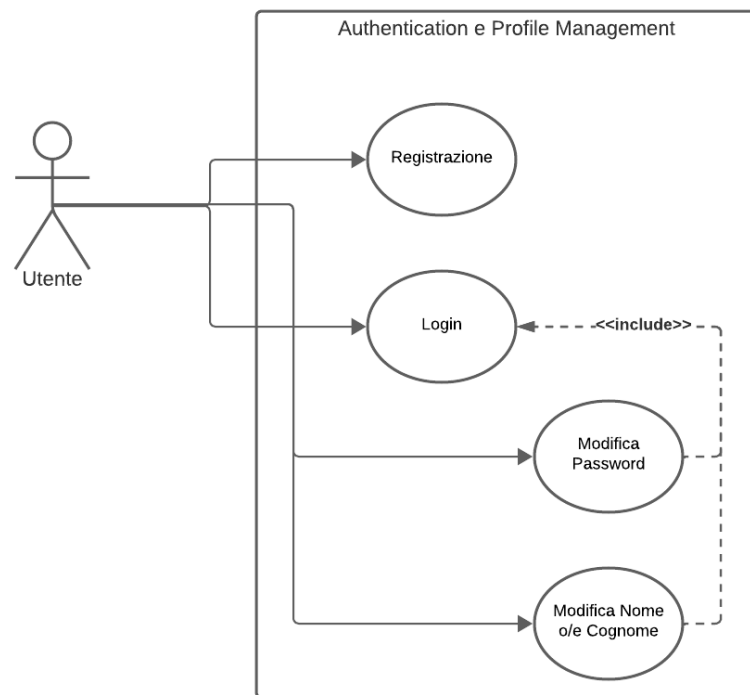


Figure 1: Caso d'uso autenticazione e gestione profilo

1.1.2 Lavagna

Per poter gestire e utilizzare le lavagne è necessario che l'utente abbia effettuato l'accesso. La gestione delle lavagne permette ad un utente di creare una lavagna, specificando un

nome, e successivamente di accedervi per disegnare. Le altre funzionalità permettono di modificare il nome della lavagna (assegnato in fase di creazione) o di eliminarla, se non più necessaria. In più vi è l'opzione di condivisione che permette di specificare l'utente con il quale si vuole collaborare durante le sessioni di disegno.

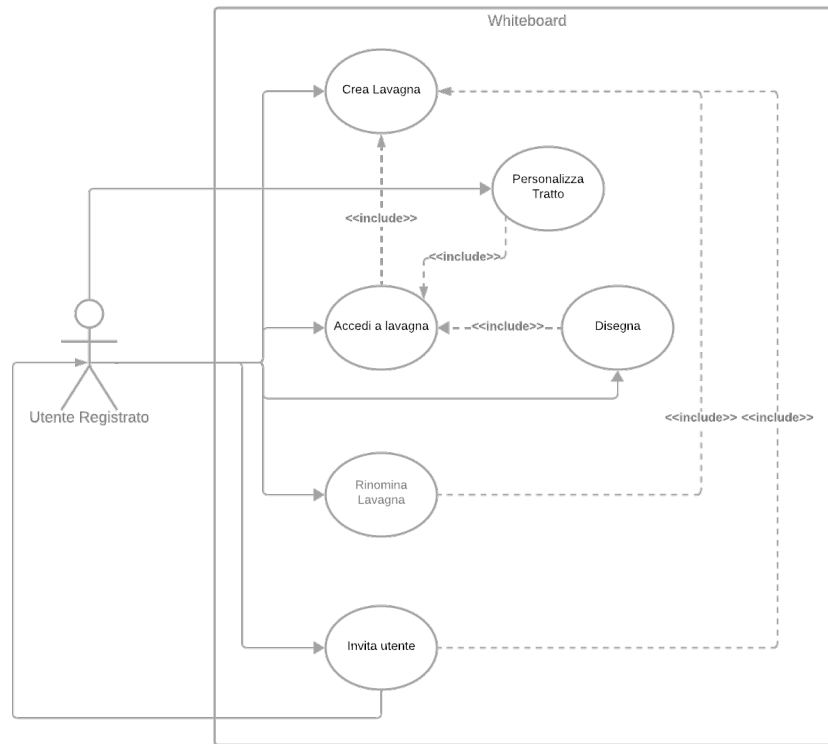


Figure 2: Caso d'uso per la gestione delle lavagne

Il disegno effettuato sulla lavagna avviene in tempo reale e in maniera collaborativa. Gli altri utenti possono vedere la modifica in modo fluido mentre avviene.

1.1.3 Notifiche

Per poter mandare e ricevere notifiche è necessario che l'utente abbia effettuato l'accesso. La creazione di un sistema di notifiche permette di mantenere l'utente sempre aggiornato su quello che succede durante l'utilizzo dell'applicativo. Le informazioni che abbiamo identificato come importanti da inviare ad un utente sotto forma di notifica sono:

- Un utente si collega ad una lavagna
- Un utente condivide una lavagna

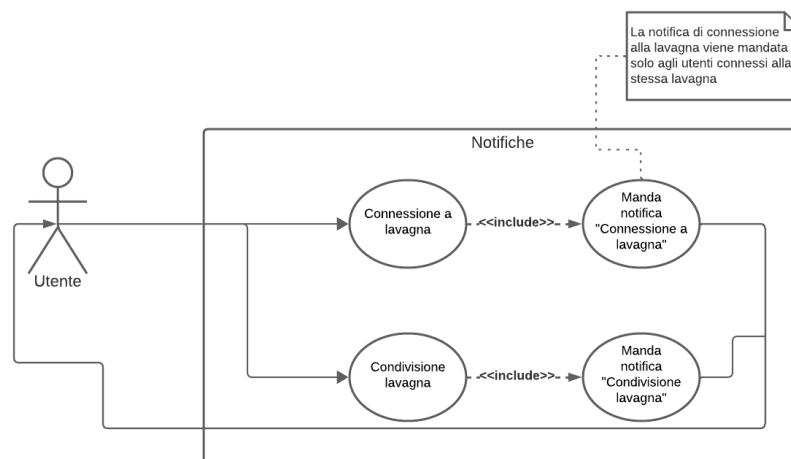


Figure 3: Caso d'uso invio notifiche

Le notifiche inviate all'utente devono essere facilmente recuperabili e deve essere possibile la loro visualizzazione ed eliminazione.

2 Analisi dei requisiti

I requisiti proposti si articolano principalmente attorno a 3 punti:

- Autenticazione e Autorizzazione degli utenti nell'applicazione
- Operazioni CRUD sulle lavagne
- Comunicazioni Real-Time per il disegno e per l'invio di notifiche

2.1 Autenticazione e Autorizzazione

Dev'essere possibile autenticare gli utenti attraverso un semplice *login system*, e garantire **solo** agli utenti che sono stati autenticati l'accesso a certe operazioni, come la creazione di lavagne e la modifica delle stesse.

Questo problema può essere risolto tramite l'utilizzo di **token**, forniti in seguito all'autenticazione con nome utente e password, e, attraverso opportune librerie, controllati prima di concedere qualsivoglia operazione che richiede l'autorizzazione.

2.2 Operazioni CRUD sulle lavagne

Dev'essere possibile creare, modificare, ottenere ed eliminare lavagne. Questo viene facilmente gestito tramite opportune rotte all'interno dell'API dell'applicazione: esistono numerose librerie che facilitano queste operazioni.

Aspetto sottile e implicito nella definizione dei requisiti riguarda l'invito di altri utenti all'interno della propria lavagna. E' sottinteso, infatti, un concetto di *owner* della lavagna e di *participant* (utente invitato). E' necessario integrare il sistema di autorizzazione nella gestione delle lavagne e permettere certe operazioni sulle stesse solo a chi ne è proprietario e altre anche ai semplici partecipanti. Questo complicherà il sistema di autorizzazione.

2.3 Comunicazioni Real-Time

Si tratta di uno degli aspetti più critici di questo progetto. Una soluzione può essere appoggiarsi a protocolli real-time o librerie che ne astraggono l'utilizzo. Inoltre, dato che gruppi di utenti possono accedere alla stessa lavagna per operare in maniera collaborativa, è necessario gestire gruppi di utenti a cui inoltrare gli stessi dati, ovvero un sistema di *room* in cui solo un determinato insieme di utenti riceve aggiornamenti sullo stato della lavagna in cui essi stanno operando.

Oltre a questo, l'interazione deve avvenire in maniera fluida, dunque, per evitare di saturare la rete con troppe comunicazioni, si può utilizzare un sistema di **interpolazione**, dove vengono inviati meno aggiornamenti per unità di tempo "riempiendo i buchi" con dati approssimati interpolando quelli ricevuti. In pratica, quello che viene effettuato è il *sottocampionamento* (o "downsampling") dei dati di disegno real-time.

3 Design

3.1 Struttura

La struttura della parte server (API) è stata suddivisa in cinque "sfere" a seconda del tipo di contesto in cui le sue funzioni ricadono:

- **Modello:** astrae tutte le comunicazioni necessarie per la memorizzazione in possibili database
- **Auth:** gestisce tutte le funzioni di autenticazione e autorizzazione, quindi gestione dei token e della registrazione di utenti
- **Controller:** risponde alle richieste degli utenti. E' stato creato un controller diverso per ogni "tipo" di richiesta (o meglio per ogni "sezione" dell'applicativo)
- **Real-Time:** responsabile dell'interazioni in tempo reale. Necessario per dati di disegno e notifiche ai client.
- **Rotte:** gestisce unicamente l'albero delle rotte in cui è possibile trovare le risorse dell'applicativo, collegandole agli opportuni controller

Il modo in cui queste interagiscono è riassunto nel diagramma in Figura 4.

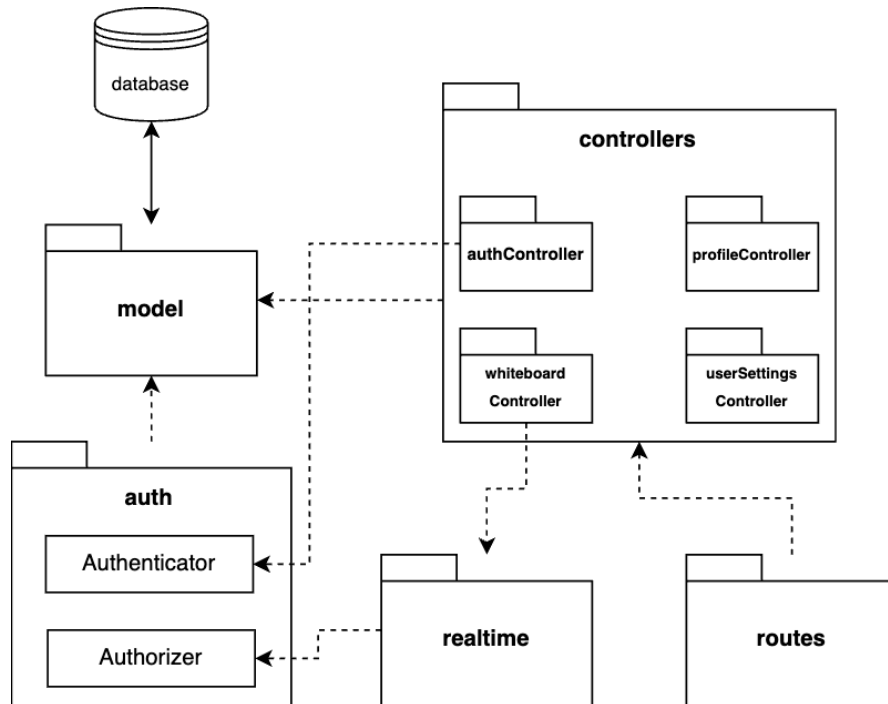


Figure 4: Struttura dei package del progetto

All'interno del package *auth*, due classi *Authenticator* e *Authorizer* si occupano rispettivamente di tutte le funzioni per l'autenticazione e l'autorizzazione degli utenti.

3.2 Comportamento

Il diagramma in Figura 5 rappresenta il susseguirsi di eventi che avvengono quando un utente cerca di accedere ad una particolare lavagna. Selezionando una lavagna, l'utente invia al server una richiesta di accesso e, se autorizzata, il server ne recupera i dati per restituirli al client. Una volta ricevuti i dati, il client non fa altro che disegnare su schermo i tratti (se presenti). Una volta fatto ciò è possibile procedere con la connessione al server responsabile dei dati in tempo reale (qui soprannominato *Socket*) che informerà tutti gli altri client che sono connessi alla stessa lavagna di un nuovo accesso.

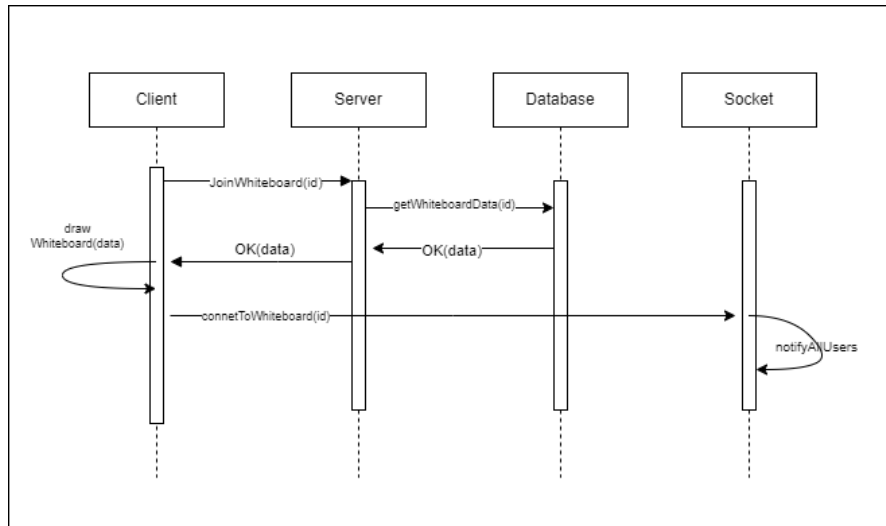


Figure 5: Schema di connessione alla lavagna

Una volta connesso ad una lavagna un utente è pronto per disegnare nuovi dati e questa procedura viene descritta nel diagramma di sequenza riportato in figura 6.

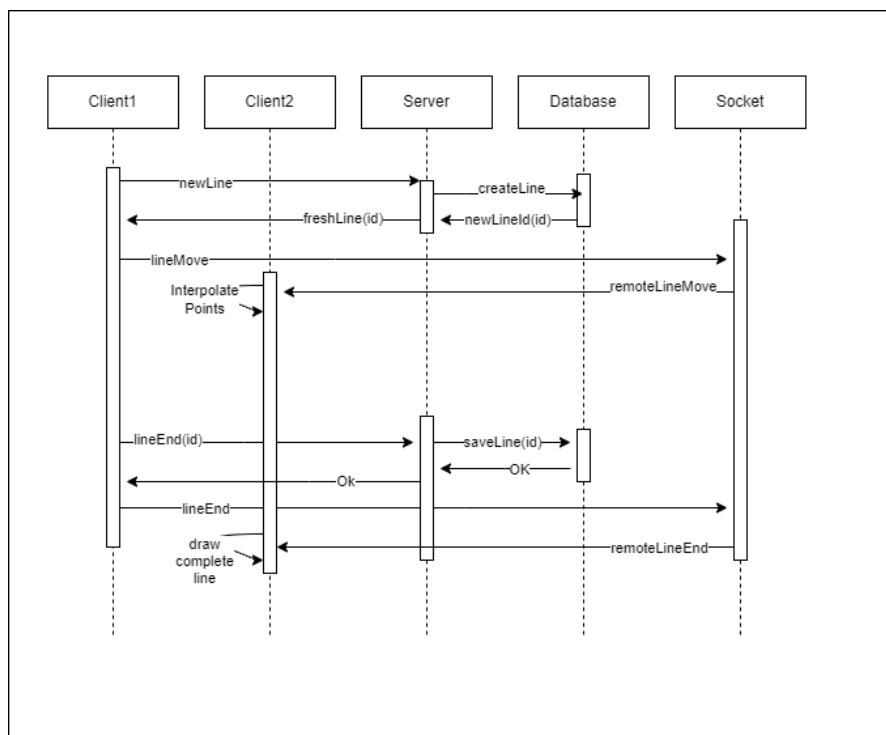


Figure 6: Schema di disegno sulla lavagna

Quando l'utente clicca sulla lavagna per disegnare una nuova linea comunica con il server l'avvenimento, e se autorizzato, richiede al database di generare un nuovo ID per la nuova linea da restituire al client. Una volta fatto ciò, continuando a disegnare, il client comunica man mano al Socket solo una piccola porzione dei punti del tratto attuale. Questi punti vengono inviati dal Socket ad ogni utente connesso in quel momento. I vari client si occuperanno dunque di approssimare, tramite tecniche di interpolazione, i dati ricevuti. Una volta che la linea viene terminata, viene prima memorizzata nel database e poi viene comunicato l'avvenimento al Socket che, come avvenuto precedentemente, lo comunica a sua volta ai client connessi i quali non faranno altro che eliminare il dato interpolato per disegnare quello completo.

Per poter collaborare lavorando con più dispositivi sulla stessa lavagna è necessario invitare gli utenti a collaborare, questa procedura è descritta nella Figura 7, notiamo come un utente, tramite apposito *form*, effettua una chiamata all'API del server specificando la lavagna che vuole condividere e con quale utente vuole condividerla, il server si occupa di fare 2 cose: salvare la notifica nel database, per permettere all'utente di vederla nel caso non fosse online e inviare una notifica all'utente destinato a ricevere la lavagna condivisa.

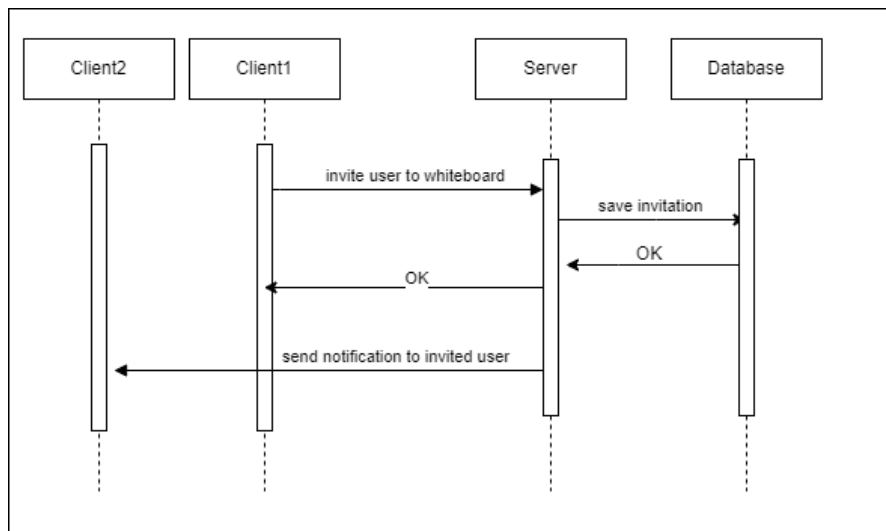


Figure 7: Schema per l'invito di nuovi utenti sulla lavagna

3.3 Interazioni

Nella Figura 8 viene riportato un diagramma riassuntivo che descrive la logica del sistema e della navigazione al suo interno, dal momento in cui l'utente avvia l'applicazione. Il diagramma in questione presuppone che un utente sia già in possesso di credenziali e inizia dunque dal login dove sono richiesti username e password. Se ci sono errori si tornerà alla pagina di login per reinserire le credenziali, altrimenti, una volta effettuato l'accesso verrà subito mostrata la pagina contenente la lista delle lavagne create o condi-

visive con l'utente. In caso di lista vuota sarà possibile procedere con la creazione di una nuova tramite l'apposito modale.

Selezionando invece una lavagna creata precedentemente l'utente entrerà in essa, dove vedrà tutte le modifiche e potrà effettuarne di nuove. In qualsiasi momento è possibile spostarsi anche nelle altre pagine disponibili tramite il menu. Allo stesso modo è possibile terminare la "sessione" eseguendo il logout tramite l'apposito bottone.

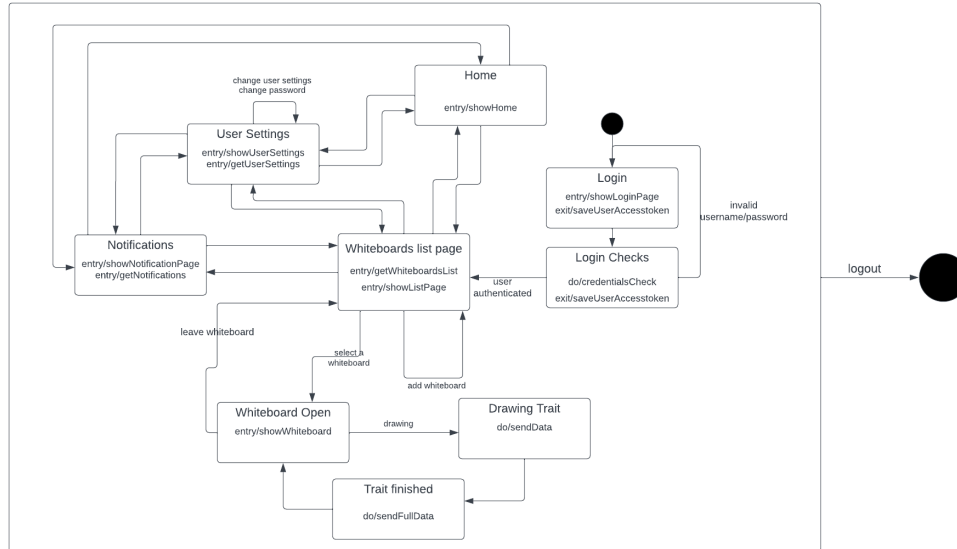


Figure 8: Navigazione nell'applicativo

4 Dettagli implementativi

Verranno riportati di seguito alcuni dettagli implementativi, ovvero porzioni del codice scritto, ritenuti di particolare interesse.

4.1 Login e Registrazione

In fase di registrazione, la password inserita dall'utente viene convertita in hash usando la libreria *bcrypt*. In questo modo, nel database le password non sono salvate in chiaro a fini di sicurezza. Per la creazione di token viene utilizzata la libreria *JWT* di npm. Le chiavi dei due tipi di token (access e refresh) sono gestite dalla classe *Authenticator*, che si occupa di tutte le funzioni di autenticazione. E' possibile regolare al suo interno le durate dei token, nel nostro caso sono state scelte 10 minuti e 7 giorni rispettivamente per access e refresh token.

```

if (user && (await bcrypt.compare(password, user.password))) {
  // Create refresh token
  user.refreshToken = jwt.sign(

```

```

        {user_id: user._id, username},
        this.refreshTokenKey,
        {
            expiresIn: this.refreshTokenExpiration,
        }
    );

    user.accessToken = jwt.sign(
        {user_id: user._id, username},
        this.accessTokenKey,
        {
            expiresIn: this.accessTokenExpiration
        }
    )
    ...
    // ritorno dell'utente e gestione degli errori
}

```

Listing 1: Sezione della funzione di login

4.2 Disegno nella lavagna

Lato client, il disegno della lavagna è stato gestito separando i tre eventi di disegno locali (*lineStart*, *lineMove* e *lineEnd*) dai tre remoti (*remoteLineStart*, *remoteLineMove* e *remoteLineEnd*). In corrispondenza di questi eventi, vengono assegnati degli ID ai tratti che vengono disegnati, in questo modo è possibile distinguere i tratti di più utenti disegnati contemporaneamente e usare l'interpolazione separatamente.

```

lineStart: function (e) {
    if ((!this.whiteboardJoined) || (this.panSelected === true)) {return}

    this.undo = true;

    let pointToAdd = this.getViewBoxCoordinates(e);
    this.$refs.socket.drawStart(pointToAdd.x, pointToAdd.y, this.lineColor)
    this.addPointToLine(pointToAdd);

    ...
},

```

Listing 2: Porzione della funzione lineStart

In corrispondenza dell'inizio di un tratto locale, questa funzione svolge 4 compiti:

- Abilita l'*undo* dei tratti
- Converte le coordinate del click dell'utente da spazio della finestra a spazio interno alla lavagna
- Avvisa un sotto-componente adibito alla comunicazione con il server dell'avvenuto evento di "drawStart", inviandogli il primo punto del tratto
- Aggiunge il primo punto alla linea che l'utente locale sta disegnando

```

lineMove: function (e) {

  if (!this.whiteboardJoined || this.panSelected) {return}

  let pointToAdd = this.getViewBoxCoordinates(e);

  if (this.gesture === true) {
    this.addPointToLine(pointToAdd);
    this.$refs.socket.drawing(pointToAdd.x, pointToAdd.y);
    this.traceCursorMovement(e)
  }

  ...
},

```

Listing 3: Porzione della funzione lineMove

Questa seconda parte gestisce il movimento del cursore dell'utente. Come prima, convertiamo le coordinate del cursore. Dopodiché, effettuiamo le stesse operazioni di prima, stavolta per l'evento drawing. La funzione traceCursorMovement serve per mostrare piccoli cerchi che indicano il tratto che sta venendo disegnato.

```

lineEnd: function (e) {
  ...
  const pointToAdd = this.getViewBoxCoordinates(e);
  this.addPointToLine(pointToAdd);
  this.$refs.socket.drawEnd(this.lineToSend, this.lineColor);

  // taken from the socket component, just updated with the new id
  const id = this.$refs.socket.drawingId;
  this.createPath(id, this.line, this.lineColor, this.width);
  ...
}

```

Listing 4: Porzione della funzione lineEnd

Infine, seguendo l'esempio dei precedenti due metodi, anche lineEnd effettua operazioni simili. Questo metodo, tuttavia, effettua anche un'altra importante operazione, rappresentata dalle righe 8 e 9. In questo punto, prima otteniamo l'ID del tratto che abbiamo appena finito di disegnare: questo ID viene settato in automatico dal sotto-componente *Socket* quando il tratto è cominciato, comunicando con il server. Dopodiché, creiamo un *path* all'interno della lavagna (che è essenzialmente un insieme di punti all'interno di un immagine in formato SVG), che verrà creata in modo da essere identica a quella inviata agli altri utenti. Lato backend, una volta ricevuto questo evento, la linea viene anche **salvata nel database**.

```

remoteLineStart: function (data) {
  if (data.id !== undefined) {
    this.$refs.interpolation.createInterpolatingPath(data.id, data.color);
  }
},
remoteLineMove: function (data) {
  if (data.id && data.point) {

```

```

    const point = data.point
    this.$refs.interpolation.interpolate(point.x, point.y, data.id)
  }
},
remoteLineEnd: function (data) {
  if (data.id && data.points) {
    this.$refs.interpolation.deleteInterpolatingPath(data.id);

    let remoteLine = "";
    let point = data.points[0];

    // creazione del path disegnato da remoto prendendo i punti ricevuti
    ...

    const id = data.id;
    this.createPath(id, remoteLine, data.color, this.width);
  }
},

```

Listing 5: Funzioni per gestione di eventi di disegno remoti

Infine, queste tre funzioni si occupano degli stessi eventi delle tre funzioni precedenti, stavolta però quando avvenute da remoto (altri utenti). Le prime due operano sullo stesso oggetto, ovvero **interpolation**. Quando vengono ricevuti punti da altri utenti, questo sotto-componente crea un path "temporaneo" creato usando tecniche di interpolazione su quei punti: in questo modo, finché l'utente remoto sta ancora disegnando, abbiamo bisogno di ricevere molti meno punti per rendere il tratto credibile e fluido.

Una volta che la linea remota è conclusa, eliminiamo il path temporaneo creato interpolando i punti. Riceviamo tutti i punti della linea in questione, questo ci permette di creare il path definitivo, presumibilmente identico a quello locale all'utente autore del tratto.

5 Deployment

È stato creato un file `docker-compose.yml` che permette di lanciare tutto l'ambiente digitando il comando:

```
$ docker compose up -d
```

Per interrompere l'esecuzione invece è possibile lanciare il comando:

```
$ docker compose down
```

Dato il comando verranno istanziati 4 container:

5.1 Database

Per il container del DB è stata utilizzata l'immagine predefinita di MongoDB dal registro Docker Hub, andando a definire alcune proprietà come la porta e il volume da utilizzare per il data persistence.

```

1 db:
2   image: mongo:latest
3   networks:
4     main:
5       aliases:
6         - db
7   ports:
8     - '27017:27017'
9   expose:
10    - 27017
11   volumes:
12    - dbdata6:/data/db

```

Listing 6: Sezione del docker-compose del DB

5.2 Ngnix

Per istanziare il container di Ngnix è stato creato un Dockerfile che, a partire dall'immagine di default di “ngnix”, va a modificare i file di configurazione con quelli specifici per il progetto, presenti localmente.

```

1 FROM nginx
2 RUN rm /etc/nginx/conf.d/default.conf
3 COPY conf /etc/nginx

```

Listing 7: Dockerfile Ngnix

Il container è poi dichiarato all'interno del docker compose che si occuperà di istanziarlo:

```

1 nginx:
2   image: whiteboard/nginx:latest
3   build: ./nginx
4   ports:
5     - "80:80"
6   networks:
7     main:
8       aliases:
9         - nginx

```

Listing 8: Sezione del docker-compose di Ngnix

5.3 Backend - Api

Per le api è stata creata un'immagine custom, a partire dall'immagine di "node", definendo all'interno del Dockerfile le dipendenze, le variabili d'ambiente e copiando i file che compongono il nostro applicativo:

```
1 FROM node:14
2
3 WORKDIR /collaborative-whiteboard-app
4 COPY package.json .
5 RUN npm install
6 COPY . .
7 ENV DB_IP=db
8 ENV FRONTEND_IP=localhost
9 ENV NODE_ENV=dev
10 CMD npm run dev
```

Listing 9: Dockerfile Api

Questa immagine viene poi utilizzata all'interno del docker-compose per istanziare il container:

```
1 backend:
2   hostname: collaborative_whiteboard_api
3   image: whiteboard/backend:latest
4   build: ./backend
5   networks:
6     main:
7       aliases:
8         - api
9   ports:
10    - "4000:4000"
11   expose:
12    - 4000
```

Listing 10: Sezione del docker-compose del backend

5.4 Frontend - Http Server

Per il frontend è stata creata un'immagine custom, a partire dall'immagine di "node", definendo all'interno del Dockerfile le dipendenze, le variabili d'ambiente e copiando i file che compongono il nostro applicativo:

```
1 FROM node:16-alpine
2
3 RUN npm install -g http-server
```

```

4 WORKDIR /app
5 COPY package*.json ./
6 ENV VUE_APP_BACKEND_IP="localhost:80"
7 RUN npm install
8 COPY . .
9
10 RUN npm run build
11 EXPOSE 8080
12 CMD [ "http-server", "dist" ]

```

Listing 11: Dockerfile Frontend

Questa immagine viene poi utilizzata all'interno del docker-compose per istanziare il container:

```

1 frontend:
2     image: whiteboard/frontend:latest
3     build: ./frontend/whiteboard-frontend
4     networks:
5         main:
6             aliases:
7                 - web
8     ports:
9         - "8080:8080"
10    volumes:
11        - ../usr/src/app/app
12        - /usr/src/app/app/node_modules

```

Listing 12: Sezione del docker-compose del backend

6 Test

Sono stati realizzati dei test automatici all'interno del progetto sfruttando le librerie *Jest* e *Mongo DB In-Memory-Server*. In particolare, è stata creata una suite di test per ogni sezione dell'API (autenticazione, profilo, real-time, impostazioni utente) più una suite aggiuntiva per testare casi limite o comunque richieste incomplete e/o mal formate.

6.1 Autenticazione

Nel test dell'autenticazione vengono usate le due funzioni principali, la *login* e la *register*. Per fare ciò viene utilizzato *Mongo DB In-Memory-Server* per creare dinamicamente un database server di *Mongo DB* e su esso vengono eseguite le comunicazioni per la creazione e login di utenti.

```
describe("POST /api/login", () => {
  it("Test User Login", async () => {
    await request(app).post("/api/auth/login")
      .send({username: "user@test.it", password: "password"})
      .then((res) => {
        expect(res.statusCode).toBe(200);
        expect(res.body.username).toBe("user@test.it");
      });
  });
});
```

Listing 13: Parte del test sul login

Il procedimento seguito in questi test viene ripetuto anche da altre suite per permettere il test di rotte che richiedono token per l'autorizzazione.

6.2 Profilo

Nel test del profilo vengono testate le principali rotte *Create*, *Read*, *Update*, *Delete* (CRUD) delle lavagne. Per fare questo viene di nuovo registrato un utente e ottenuto il suo token di accesso.

In seguito vengono eseguiti vari test che operano sul profilo.

```
describe("GET /api/profile/", () => {
  it("Test User Get Profile and Whiteboard creation", async () => {
    await profileHasZeroWhiteboards()
    const createResponse = await request(app)
      .post("/api/profile/createWhiteboard")
      .send({whiteboardName: "Test"})
      .set({ Authorization: 'Bearer ${ACCESS_TOKEN}' })
    expect(createResponse.statusCode).toBe(200);
    expect(createResponse.body).toHaveProperty("message")
    expect(createResponse.body).toHaveProperty("whiteboardId")

    CREATED_WHITEBOARDS.push(createResponse.body.whiteboardId)

    await profileHasOneWhiteboardWith("Test")
  });
});
```

Listing 14: Test per la creazione di una lavagna

6.3 Impostazioni utenti

In modo simile ai test per il profilo, anche i test per aggiornamento delle impostazioni dell'utente (nome, cognome, password) utilizzano il token di sicurezza per testare tutte le operazioni.

```
describe('PUT /api/userSetting/updatePassword', () => {
  it("Test user data retrieval", async () => {
    const passRes = await request(app)
      .put("/api/userSetting/updatePassword")
      .send({
```

```

        password: "newPassword"
      })
      .set({ Authorization: 'Bearer ${ACCESS_TOKEN}' })
    expect(passRes.statusCode).toBe(200)
    expect(passRes.body.message)
      .toBe("User data updated successfully");
    const getRes = await request(app)
      .get("/api/userSetting/")
      .send()
      .set({ Authorization: 'Bearer ${ACCESS_TOKEN}' })
    expect(getRes.statusCode).toBe(200);
  })
})

```

Listing 15: Test per l'aggiornamento della password dell'utente

6.4 Real-Time

I test sulle comunicazioni real-time hanno richiesto più tempo e documentazione. Le comunicazioni avvengono tramite la libreria Socket.IO e per testarle correttamente è necessario avere **almeno 2 client connessi**. Per fare questo è stato necessario istanziare due client usando la funzione `io.connect()` e sfruttare una complessa catena di Promise per integrare la struttura **sincrona** di *Jest* con la struttura invece **asincrona** di *Socket.IO*.

```

let drawEndPromise = new Promise((resolve) => {
  it("Test Whiteboard Draw End", async () => {
    drawingPromise.then(() => {
      let lineToSend = {id: 1, point: [{x: 3, y: 2}],
        color: "Red", stroke: 2}
      clientSocket2.on("drawEndBC", (line, lineId) => {
        expect(line).toStrictEqual(lineToSend);
        expect(lineId).toBe(traitID);
        resolve();
      });
      clientSocket.emit("drawEnd", lineToSend,
        traitID, ACCESS_TOKEN_1);
    });
  });
});

```

Listing 16: Test per verificare la fine di un tratto disegnato

In questa suite di test, si testa la corretta ricezione degli eventi di disegno inviati dal primo client e ricevuti dal secondo.

```

it("Test Trait in Whiteboard", (done)=>{
  drawEndPromise.then(() =>{
    request(app).get("/api/whiteboard/" + WHITEBOARD_ID)
      .send()
      .set({Authorization: 'Bearer ${ACCESS_TOKEN_1}'})
      .then((whiteboard) => {
        const data = whiteboard.body.whiteboardData

```

```

        expect(Object.values(data.traits)[0])
            .toEqual(lineToTest);
        done();
    })
})
})

```

Listing 17: Test per verificare che il tratto è infine salvato nel database

7 Esempi di utilizzo

Insieme alle API è stato sviluppato un portale Web per accedere ai servizi tramite un'interfaccia. E' possibile effettuare il login tramite l'apposita pagina.

Login



Not a member? [Register](#)

Figure 9: Pagina di login

Alternativamente è possibile registrarsi usando il link verso la relativa sezione. Questa operazione gestisce i token di autorizzazione sfruttando lo *storage* del browser, usabili per accedere a funzioni riservate.

Effettuato il login, nuove sezioni dell'interfaccia diventano disponibili, tra cui il profilo, la pagina delle lavagne, le notifiche e il disegno.

Figure 10: Pagina del profilo

Nel profilo è possibile rivedere e modificare i dati del proprio account, tra cui nome, cognome e password. La password è mostrata come placeholder, non è effettivamente inviata al browser.

Usando la pagina delle lavagne è possibile creare, rinominare, eliminare le lavagne oppure invitare altri utenti nella lavagna di cui si è proprietari. Certe operazioni vengono permesse solo agli utenti proprietari, quindi nell'interfaccia vengono omesse in caso contrario.

Cliccando su una lavagna, si accede ad essa e quindi all'interfaccia per il disegno.

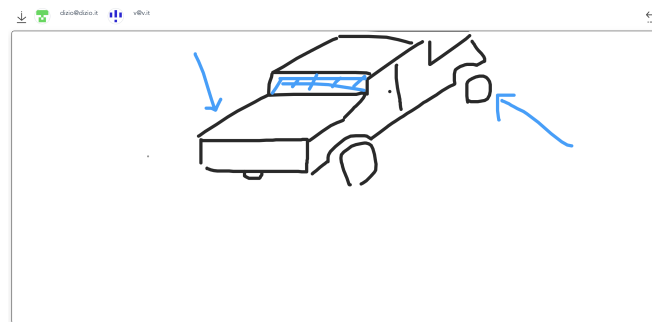


Figure 11: Pagina di disegno

Una volta entrati nella lavagna, dopo un breve caricamento, è possibile disegnare nell'apposito rettangolo con il mouse, e, usando un *widget*, modificare colore e spessore del tratto.

Sopra è mostrata la lista di utenti connessi alla lavagna sotto forma di piccoli avatar circolari.

Mentre un utente disegna, il tratto relativo viene mostrato in maniera "abbozzata" e lievemente trasparente in tempo reale mentre viene disegnato, mentre una volta finito il tratto viene mostrato interamente.

Infine, è disponibile una pagina dove accedere alle proprie notifiche, per visualizzarle e, se necessario, eliminarle definitivamente.

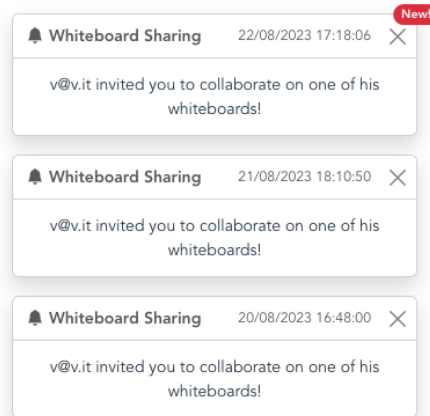


Figure 12: Pagina delle notifiche

8 Conclusioni

Il progetto realizzato ha raggiunto tutti gli obiettivi posti dai requisiti iniziali. L'applicativo è facilmente utilizzabile e può rappresentare un ottimo punto di partenza per la creazione di una più completa applicazione a livello *enterprise*. Esso, inoltre, è stato testato su più dispositivi diversi e l'interazione tra essi è risultata piacevole e senza particolari difetti.

8.1 Sviluppi futuri

Il progetto è espandibile e migliorabile sotto molti aspetti. L'interfaccia grafica lato client è ancora semplice e minimale, un miglior design potrebbe renderla più fruibile. Nell'intero applicativo, invece, molte funzionalità potrebbero essere aggiunte per renderlo più potente e competitivo. Alcune di queste vengono presentate in questa lista:

- **Lavagna più versatile:** al momento la lavagna supporta solo tratti (insiemi di punti) di diverso colore e spessore. Aggiungere oggetti multimediali (come delle immagini), blocchi di testo, forme ecc. potrebbero essere una buona aggiunta. Anche l'aggiunta di altri strumenti (gomma, strumento di rotazione o spostamento dei tratti già creati, grouping, gestione di livelli ecc.) renderebbe la lavagna molto più versatile.
- **Gestione di permessi più profonda:** al momento l'applicativo prevede due tipi di permessi alle lavagne: proprietario e ospite. All'ospite non sono permesse la cancellazione della lavagna o l'invito di altri utenti alla stessa. Migliorare questa gestione, per esempio includendo accessi di "sola lettura" alle lavagne oppure sistemi per il trasferimento di permessi tra utenti sarebbe un'ottima aggiunta.
- **Profili più completi per gli utenti:** allo stato attuale, gli utenti sono molto poveri di informazioni all'interno dell'applicativo. Aggiungere più dati agli utenti, permettere interazioni tra utenti (collaborazione, condivisione, partecipazione a

progetti in comune ecc.) e rendere il dominio dell'utente più "ricco" gioverebbe alla competitività dell'applicativo.

8.2 Concetti appresi

Sviluppare questo progetto ha permesso di approfondire le interazioni real-time in applicazioni di questo tipo, tra cui i protocolli e le librerie che è possibile utilizzare e come utilizzarle. La scala dell'applicazione è risultata sufficiente per comprendere come sistemi composti da più servizi possono interagire e, le relative difficoltà di implementazione.

Nello sviluppare la parte lato server, inoltre, sono state apprese molte delle possibili debolezze nella sicurezza e nella robustezza di un applicativo di questo tipo e possibili soluzioni per affrontarle e/o mitigarle. Infine, l'utilizzo di strumenti per il deploy automatico (come *docker* e *docker-compose*) ha fatto emergere le difficoltà per quanto riguarda l'integrazione di molti strumenti diversi usati contemporaneamente e la configurazione degli stessi.

Riferimenti

- [1] Documentazione di Bootstrap 5. <https://getbootstrap.com/docs/5.0/getting-started/introduction/>.
- [2] Documentazione di Docker. <https://docs.docker.com/>.
- [3] Documentazione di Express.js. <https://expressjs.com/en/4x/api.html>.
- [4] Documentazione di Socket.io. <https://socket.io/docs/v4/>.
- [5] Documentazione di Vuejs. <https://vuejs.org/guide/introduction.html>.