

Report Outline and Marking Schema

Acquadro Federico, Rastello Umberto, Tricarico Riccardo

Sviluppo Front-end

Soluzioni:

Per l'implementazione del nostro progetto, abbiamo scelto di utilizzare React in combinazione con Vite per diversi motivi chiave. React, con il suo approccio basato su componenti, ci consente di creare interfacce utente modulari e riutilizzabili. Grazie al Virtual DOM di React, possiamo ottimizzare il rendering delle nostre applicazioni, assicurando performance elevate anche con interfacce complesse come la nostra.

Un altro motivo per cui abbiamo scelto di usare React è che quest'ultimo è ampiamente richiesto dalle aziende informatiche specializzate nello sviluppo front-end. La decisione di utilizzare React nel nostro progetto non solo ci permette di affrontare le sfide tecniche con una tecnologia consolidata e ben supportata, ma ci offre anche l'opportunità di acquisire competenze che saranno fondamentali per il nostro sviluppo professionale futuro.

Vite, d'altra parte, si distingue per la sua velocità di sviluppo. Il suo sistema di build veloce e snello ci consente di avere tempi di caricamento rapidi durante lo sviluppo, migliorando la nostra produttività. Vite supporta il caricamento veloce dei moduli e l'aggiornamento istantaneo dei cambiamenti nel codice, rendendo il processo di sviluppo più efficiente.

Inoltre, React è compatibile con i file con estensione .jsx, particolari file che permettono la perfetta collaborazione tra codice JavaScript e codice HTML. Questo offre ai programmatori uno sviluppo molto più rapido e ordinato rispetto all'utilizzo di file JS e HTML separati. Un'altra cosa da dire è che JSX offre un sistema di gestione dello stato molto semplice. L'uso di 'useState' e 'useEffect' permette di gestire lo stato dell'applicazione in modo predittivo e senza effetti collaterali inaspettati.

Problemi e limitazioni:

L'utilizzo di React è relativamente facile, l'unica difficoltà che abbiamo rilevato è l'apprendimento della sintassi .jsx che ha richiesto circa 30 ore di tutorial

Conclusione:

utilizzare react è stata a nostro parere un'ottima idea sia dal punto di vista di crescita personale, che dal punto di vista funzionale per il nostro progetto.

Divisione del lavoro:

30% Tricarico, 30% Acquadro, 40% Rastello

Bibliografia:

 Introduzione a React - React Tutorial Italiano 01

Sessione utente

Soluzioni:

Inizialmente l'idea era quella di gestire completamente le informazioni prodotte dall'utente attraverso il localStorage, simulando così il salvataggio delle proprie informazioni. Abbiamo poi optato per un salvataggio delle informazioni nei database, così da rendere più verosimile questa funzionalità.

Abbiamo implementato la sessione utente dando la possibilità di registrarsi tramite password e username, salvandoli come record su MongoDB. Da quel momento in avanti, l'utente potrà fare login e logout utilizzando le credenziali appena salvate. Oltre a questi dati, abbiamo dato la possibilità di poter salvare giocatori, squadre e formazioni preferite. Le informazioni sono salvate come array in MongoDB nel record dell'utente.

Per quanto riguarda invece le formazioni preferite, queste vengono salvate nel database come un array di oggetti contenente i giocatori nelle varie posizioni.

Non vi è un massimo di club e giocatori salvabili, a differenza del numero di formazioni: esso è limitato al numero di moduli di gioco predefiniti durante la creazione della formazione. Per esempio, è possibile salvare una sola formazione con modulo di gioco '4-4-2'.

Il localStorage è comunque utilizzato in questa implementazione. Infatti, una volta fatto login, il back-end manda tutte le informazioni relative all'utente: queste vengono sia utilizzate dal front-end che salvate in localStorage. Ciò per minimizzare il numero di richieste al server, anche in caso di alterazione dei dati. Per esempio, se si salva un nuovo club preferito l'informazione sarà immediatamente disponibile al front-end senza dover aspettare una risposta dal server.

Problemi e limitazioni:

La comunicazione tra back-end e front-end, anche se porta informazioni sensibili, è in chiaro: non sono state implementate funzioni di criptazione o decriptazione.

Per salvare nel database le formazioni viene mandata una matrice contenente l'intero oggetto giocatore invece che solo l'id. Forse, questo è uno spreco di spazio che non scala bene con il crescere degli utenti.

Conclusione:

I passaggi finora elencati sono quindi stati utili a raggiungere un obiettivo semplice ma utile per il sito: salvare le informazioni dell'utente. Sarebbe stato interessante affrontare i problemi relativi alla sicurezza delle comunicazioni, ma abbiamo preferito utilizzare il tempo su altri aspetti del progetto.

Divisione del lavoro:

45% Tricarico, 45% Acquadro, 10% Rastello

Chat system:

Soluzioni:

Per implementare le chat è stata usata la libreria socket.io nel main server e socket.io-client lato client con React. Questa soluzione permette di avere un sistema di chat tramite room, in cui ogni stanza è separata dalle altre e ogni utente che vi partecipa potrà messaggiare in modo istantaneo con tutti gli altri.

Una stanza viene creata nel momento in cui il primo utente ci entra e viene chiusa quando l'ultimo esce, questo permette di avere tante stanze dinamiche e personalizzate.

Nel progetto abbiamo scelto di dare la possibilità di creare una stanza personalizzata per ogni giocatore ed ogni club nel momento in cui si entra nelle rispettive pagine in modo da poter scambiare informazione, domande o curiosità specifiche sul giocatore o club.

Inizialmente si era pensato di usare una pagina dedicata del sito per inserire i bottoni tramite cui connettersi alle diverse stanze, ma questo avrebbe generato due possibili soluzioni con problemi: avere un numero fisso e limitato di room diverse oppure avere una lista lunghissima di bottoni, uno per ogni giocatore e squadra.

Viste le inconvenienze, la soluzione migliore ci è sembrata quella di optare per un bottone in ogni pagina giocatore/squadra.

Problemi e limitazioni:

Una limitazione a questo sistema, che utilizza una pagina dedicata per scambiare messaggi, è non poter chattare mentre si naviga nel sito o si guardano le statistiche del soggetto poiché cambiando pagina automaticamente si abbandona la stanza.

Da un certo punto di vista, un'altra limitazione è l'assenza di uno storico dei messaggi di ogni chat lato server. Questo comporta che, ogni volta che si abbandona una room, tutti i messaggi vengono cancellati dal proprio client, e rientrando nella chat, questa risulterà vuota.

Conclusione:

Il chat system implementato è flessibile e dinamico. Avendo implementato anche una funzione di login, si è scelto di limitare l'accesso alle chat solamente a chi si è registrato.

Divisione del lavoro:

70% Acquadro, 15% Rastello, 15% Tricarico

Bibliografia:

Video di youtube: [Socket.io + ReactJS Tutorial | Learn Socket.io For Beginners](#)

Suddivisione dei server e gestione dei database:

Soluzioni:

Nel progetto sono stati utilizzati quattro differenti server per rispettare la richiesta sulla costellazione di server. Quello principale (Main Server) fa da tramite per la comunicazione tra il server di vite e gli altri server collegati ai database ed è sviluppato con express. Il secondo server è di nuovo sviluppato con express e serve a gestire gli accessi ad un database non relazionale: MongoDB (porta 3001). In questo sono presenti le tabelle riguardanti i dati più dinamici come le partite, gli eventi, le news, le formazioni e gli utenti registrati. Il terzo server è sviluppato con java spring boot (porta 8080) ed è collegato ad un database SQL Postgres che contiene le informazioni più statiche come le squadre, i calciatori, i campionati e i valori di mercato. l'ultimo server già citato prima è il server di vite (porta 5173) che abbiamo generato automaticamente grazie a node attraverso il comando `npm create vite@latest`.

Per gestire i dati lato server abbiamo usato il pattern MVC e le classi DTOs in java. Il lavoro di calcolo viene delegato ai due server connessi ai database: quando devono restituire dei dati la computazione per modificarli è svolta qui, il main server si occuperà solo di passare le informazioni che riceve al client.

La comunicazione tra il secondo server express e il server spring boot non è diretta ma passa per il server principale per evitare problemi di CORS.

Oltre alle tabelle create con i csv a disposizione, ne abbiamo generate di nuove per le news, per gli utenti registrati e per le classifiche delle squadre nei diversi campionati. per quanto riguarda le news, abbiamo copiato 5 notizie dal sito <https://www.gazzetta.it/>. Il codice per la visualizzazione delle notizie è stato predisposto alla renderizzazione di un numero indefinito di notizie.

Qui sotto ci sono gli screen di due particolari route degne di nota:

la prima è http://localhost:3000/player/player_clubs/123 nel main server nel file player.js. Questa route fa richieste axios a entrambi i server. Inizialmente, vengono richiesti i dati da MongoDB, ma questi sono incompleti; ad esempio, possiamo avere solo l'ID della squadra senza il nome. Per ottenere il nome della squadra, il server principale effettua una seconda richiesta al server di PostgreSQL utilizzando l'ID della squadra.

```
umberto +2
router.get( path: '/player_clubs/:player_id', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<ResBody> ) => {
  try {
    const response = (await axios.get( url: `http://localhost:3001/events/player/${req.params.player_id}`)).data;
    const clubs_id : any[] = [];
    const data : {clubs: ...} = {
      'clubs': []
    };
    if (Symbol.iterator in Object(response.data)) {
      for (let i of response.data) {
        if (!clubs_id.includes(i.club_id)) {
          try {
            const club_name : AxiosResponse<any> = await axios.get( url: `http://localhost:8080/club?id=${i.club_id}`);
            if (club_name) {
              data.clubs.push(club_name.data.name);
              clubs_id.push(i.club_id);
            }
          } catch (err) {
            console.log(err);
          }
        }
      }
    }
    res.json(data);
  } catch (err) {
    console.log(err);
  }
}
```

```
umberto +1 *
router.get( path: '/home/:searchTerm', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<ResBody> ) => {
  const searchTerm = req.params.searchTerm;

  const urlPlayerById : string = `http://localhost:8080/player?id=${searchTerm}`;
  const urlPlayerByName : string = `http://localhost:8080/playerByName?name=${searchTerm}`;
  const urlCompetition : string = `http://localhost:8080/competition?id=${searchTerm}`;
  const urlClubByName : string = `http://localhost:8080/clubByName?name=${searchTerm}`;
  const urlCompetitionByName : string = `http://localhost:8080/competitionByName?name=${searchTerm}`;

  try {
    const [responseById : AxiosResponse<...> | {...},
      responseByName : AxiosResponse<...> | {...}, responseCompetition : AxiosResponse<...> | {...},
      responseClubByName : AxiosResponse<...> | {...}, responseCompetitionByName : AxiosResponse<...> | {...}] = await Promise.all( values: [
      axios.get(urlPlayerById).catch(err => ({error: err.response})),
      axios.get(urlPlayerByName).catch(err => ({error: err.response})),
      axios.get(urlCompetition).catch(err => ({error: err.response})),
      axios.get(urlClubByName).catch(err => ({error: err.response})),
      axios.get(urlCompetitionByName).catch(err => ({error: err.response}))
    ]);
  } catch (err) {
    console.log(err);
  }
}
```

Questa seconda route invece è la route '[http://localhost:3000/home/\\${searchterm}](http://localhost:3000/home/${searchterm})' sempre nel main server nel file index.js in cui vengono effettuate cinque richieste Axios contemporaneamente a diversi URL tramite la funzione **promise.all**. Queste richieste vengono eseguite in parallelo, e il programma attende che tutte le richieste siano complete prima di procedere. Quando tutte le richieste sono completate, i risultati vengono assegnati in un array di risposta. Questo approccio è efficiente perché permette di ridurre il tempo totale di esecuzione aspettando contemporaneamente tutte le risposte, anziché attendere ciascuna richiesta separatamente, una dopo l'altra.

Problemi e limitazioni:

Utilizzare diversi server rende più complicato trovare errori quando si eseguono le routes tra un server e l'altro. Utilizzando database differenti, in alcuni casi, per fare delle union tra i dati bisogna mandare da un server all'altro grandi quantità di dati e questo può creare dei rallentamenti nel funzionamento complessivo.


Conclusione:

La soluzione che abbiamo adottato è nel complesso scalabile anche in uno scenario in cui le richieste sono numerose, dove un server principale gestisce le richieste e delega i calcoli maggiori ad altri server.

Divisione del lavoro:

33% Acquadro, 33% Rastello, 33% Tricarico

Bibliografia:

mongoose:  Mongoose Crash Course - Beginner Through Advanced

spring boot:  How to create an REST API in Spring boot using Java

Generative AI:

Nella realizzazione di questo progetto abbiamo fatto uso dell'AI, in particolare di Chat GPT. Il motivo principale dell'utilizzo di questo strumento è stato quello di ridurre i tempi di lavoro nel scrivere codice e risolvere problemi minori: come sintassi, ecc..

Avendo imparato React per il front-end da zero per il progetto, le difficoltà che abbiamo incontrato sono state inizialmente colmate dagli esempi forniti da ChatGPT.

Anche nella documentazione abbiamo usato l'AI per la generazione dei commenti swagger e javadoc per descrivere le routes.

Questo ci ha agevolato nella velocità di sviluppo e nel tempo risparmiato che altrimenti avremmo impiegato nel cercare le soluzioni all'interno dei forum dedicati.

Sicuramente l'AI è uno strumento molto potente e versatile, che oggi è fondamentale saper utilizzare con le giuste precauzioni per stare al passo con le tecnologie che ci circondano.