CREstionale

Luca Assolari Sergio Piatti

Gennaio 2022

Indice

Parte I

Iterazione 0

Capitolo 1

Introduzione

Il progetto in questione si prepone di realizzare un software per la gestione interna di un centro estivo per bambini e ragazzi. L'esigenza è nata dall'esperienza di uno dei due autori, che da anni svolge il ruolo di coordinatore all'interno di uno di questi centri. Vedendo in esso molte potenzialità di informatizzazione, questo lavoro vuole sviluppare diverse funzioni gestionali e burocratiche, per aiutare a rendere più snella e immediata la raccolta e lo scambio di informazioni che quotidianamente devono essere elaborate in un'attività di questo tipo.

L'idea iniziale è quella di implementare l'utilizzo del software dal punto di vista di uno solo dei possibili attori, come verrà spiegato in seguito. Tuttavia, uno degli obiettivi del progetto rimane quello di garantirne l'espandibilità, in modo tale da poter essere completato ed effettivamente *deployato* in un secondo momento.

Capitolo 2

Analisi del contesto

2.1 Contesto reale

I centri estivi, noti anche con il nome di CRE (centro ricreativo estivo) o GREST (gruppo ricreativo estivo), rappresentano una realtà molto diffusa in tutto il nord Italia, dove storicamente uno dei luoghi di aggregazione giovanile più frequentati è l'oratorio (a differenza del centro-sud Italia, dove è sempre stato più facile osservare ritrovi di bambini e ragazzi semplicemente in piazze o spazi aperti comunali). Negli ultimi anni si sono sempre più diffuse alternative molto simili, proposte da associazioni sportive, territoriali e giovanili; le motivazioni di fondo e le *mission* delle singole realtà esulano dalla trattazione di questo progetto, ma è possibile ritrovare in tutte la necessità pratica delle famiglie di trovare uno spazio sicuro ed educativo a cui affidare i propri figli in quel periodo estivo di inizio stagione.

Un centro estivo è solitamente strutturato per coprire le fasce orarie mattutine (dalle 8/9 fino alle 12 circa), pomeridiane (dalle 14 alle 18/19) e l'orario del pranzo. Diversi centri offrono diverse opzioni di iscrizione; in ogni caso il software pensato verrà proposto per essere modulare e potersi adattare alle singole esigenze. Gli utenti del centro sono bambini e ragazzi, solitamente dai 6 ai 14 anni, assistiti da animatori dai 14 ai 18 anni, e coordinatori maggiorenni che si occupano dell'organizzazione e della gestione delle attività, oltre che delle pratiche burocratiche e dei rapporti con le famiglie. I bambini e ragazzi vengono suddivisi in squadre per fasce d'età, che vengono poi affidate a gruppi di animatori, con un referente per ogni squadra.

Qui di seguito viene riportato lo schema di una giornata tipo nel CRE gestito da uno degli autori.

> All'arrivo al mattino, dopo l'appello, si svolgono delle attività a gruppi, oppure, a seconda del giorno, si effettua un'uscita sul territorio. A pranzo si ferma chi è iscritto e confermato, e ogni bambino può scegliere le preferenze per il pasto. Nel primo pomeriggio, all'arrivo di chi è iscritto solo al pomeriggio e al rientro di chi è tornato a casa a pranzo, viene rifatto l'appello. Successivamente il pomeriggio è diviso in due macro-momenti, intervallati dalla pausa merenda; in questi momenti si svolgono solitamente giochi, arbitrati dagli animatori, e laboratori, con il supporto di volontari e organizzatori. La giornata finisce con un momento di preghiera, la consegna degli avvisi e il saluto. Un giorno diverso dagli altri è quello della gita, quando per tutta la giornata ci si sposta verso un parco acquatico, un rifugio in montagna o il lago; queste giornate vengono gestite di volta in volta, a seconda delle presenze e della meta.

La gestione di un centro estivo è assimilabile, seppur in maniera ridotta, a quella di una piccola azienda. Suddivisione del lavoro, orari prestabiliti, gerarchie di riferimento sono tutti elementi organizzativi che accomunano una realtà ricreativa di questo tipo ad una vera e propria attività lavorativa.

In questo senso, così come da anni le imprese tendono verso una sempre maggiore informatizzazione dei processi e delle informazioni, nello stesso modo gli autori hanno ipotizzato di adottare una soluzione di questo tipo, al fine di semplificare drasticamente molti processi. Ad oggi, infatti, la maggior parte della gestione dell'informazione, dal più banale appello fino all'attività di *time management* delle giornate, è ancorata a metodi lenti e ormai superati, come l'utilizzo esclusivo di carta e penna, o la trasmissione orale e 'in presenza' delle informazioni organizzative.

A titolo d'esempio, come da esperienza diretta di uno degli autori, il procedimento per effettuare il conteggio delle presenze alla mensa, con le relative preferenze alimentari di ogni iscritto, è lento e poco efficiente: ogni referente, nel momento dell'appello mattutino, riporta su un foglio l'eventuale presenza al pasto di tutti i componenti della propria squadra e le relative preferenze; in seguito alla consegna dei fogli di tutte le squadre ad un coordinatore, questi provvede al conteggio manuale e aggregato dei dati; i totali vengono poi fisicamente comunicati ai responsabili della mensa, che provvederanno alla

preparazione dei pasti. Come si può intuire, questo procedimento ha diversi punti di fragilità, come l'eventuale perdita di informazione dovuta a richiami o ritardi nelle consegne.

2.2 Confronto con altri software

Da una rapida analisi delle alternative già presenti sul mercato, si è concluso che questi prodotti considerano come loro utente finale le famiglie degli iscritti, semplificando i processi di iscrizione e controllo, anche a livello economico, delle attività dei loro figli all'interno dei centri. Queste soluzioni, seppure rappresentino delle valide proposte per migliorare la gestione del rapporto con le famiglie, non offrono però particolari strumenti alle figure direttamente coinvolte nelle attività organizzative interne. A loro si rivolge, invece, il focus di questo progetto, che si delinea quindi come una proposta affiancabile alle suddette complementari, e/o integrabile in prodotti di più ampio respiro.

2.3 Analisi dei requisiti

Prima dell'effettivo sviluppo del progetto si è resa necessaria una fase iniziale di raccolta, analisi e revisione di requisiti e specifiche, ovvero di tutte le proprietà richieste, o auspicabili, del prodotto. La stesura del documento dei requisiti aiuta ad avere sempre sotto controllo le funzionalità implementate ad ogni iterazione dello sviluppo e funge come una sorta di contratto tra il committente e lo sviluppatore, richiedendo dei comportamenti ben delineati al prodotto finale.

Il processo di analisi dei requisiti si apre con una fase di **esplorazione**, nella quale vengono effettivamente ricercati e raccolti, con modalità diverse, i requisiti necessari e desiderabili del prodotto in sviluppo. Per il lavoro in questione si è provveduto a sottoporre una serie di domande ai responsabili dei centri estivi nei paesi limitrofi, evidenziando come, per la maggior parte, le necessità e le richieste siano accomunabili fra di loro e pressoché identiche a quelle derivanti dall'esperienza personale di uno dei due autori.

2.3.1 Requisiti non funzionali

Sono requisiti che non rappresentano vincoli funzionali da rispettare ma definiscono piuttosto le prestazioni di qualità del prodotto.

Usabilità Il sistema deve essere quanto più possibile di immediato utilizzo, essendo pensato per semplificare una serie di processi interni al centro estivo. La facilità e l'intuitività d'uso devono essere caratteristiche fondamentali, in modo tale che i vari attori finiscano per preferire una soluzione informatizzata di questo tipo piuttosto che i passati strumenti e metodi di lavoro.

Affidabilità Le prestazioni di affidabilità del sistema devono essere discretamente elevate, poiché un rischio di guasti e conseguente perdita di dati potrebbe causare grandi problemi logistici all'intero centro.

Manutenibilità Il sistema deve essere progettato in modo tale da essere altamente modificabile e adattabile alla specifiche esigenze di ciascun centro nel quale possa essere utilizzato. A tale scopo, l'utilizzo di Angular per la gestione del *front end* garantisce un'elevata modularità del software, essendo questa una proprietà intrinseca nel linguaggio stesso (come verrà approfondito in seguito). I cosiddetti **NgModule** permettono di aggregare tra loro le funzionalità di un singolo componente, in modo tale da poterle modificare facilmente in caso di necessità senza intaccare il resto dell'applicazione.

Portabilità Il sistema deve avere un'elevata portabilità, in modo tale da poter essere utilizzato da un vasto numero di tipologie di dispositivi differenti. La soluzione proposta è quella della *web app*, un'applicazione fruibile via web tramite un collegamento a internet. Questo garantisce un facile accesso al servizio da parte di tutti i possibili attori, evitando scomode problematiche come l'installazione di file in locale, con conseguenti problemi di compatibilità di dispositivo o di versione del sistema, o la necessità di aggiornamenti del client dell'app, che possono avvenire in momenti in cui è invece richiesto un rapido e immediato utilizzo del software.

Requisiti legislativi Con questo termine si indicano i requisiti **di sicurezza** e **di riservatezza**, molto importanti soprattutto in questo contesto, dove vengono trattati dati sensibili di persone perlopiù minorenni. È quindi necessario conservare i dati in database sicuri con accesso protetto.

2.3.2 Casi d'uso

Nello studio dei casi d'uso, ci si è concentrati unicamente sull'analisi di quelli relativi ad uno solo dei possibili attori: l'animatore/coordinatore. Questo perché, come già detto, il focus del progetto è quello di semplificare lo svolgimento di quei processi che sono a carico di queste figure, soggetti che più possono beneficiare di una informatizzazione delle procedure.

UC1: Registrazione

1. **Attori**
   Animatore
2. **Descrizione**
   L'attore provvede all'inserimento dei dati personali e alla sua registrazione.
3. **Precondizioni**
   L'attore non deve essere già registrato nel sistema.
4. **Passaggi principali**
   (a) L'attore preme il pulsante di registrazione;
   (b) Il sistema propone la scelta all'attore, chiedendogli di identificarsi come animatore, genitore o bambino;
   (c) L'attore, premendo l'apposito pulsante, indica al sistema di volersi registrare in qualità di animatore;
   (d) L'attore inserisce le proprie generalità nel form di registrazione;
   (e) Il sistema mostra un calendario con le settimane di svolgimento del centro estivo;
   (f) L'attore specifica la propria disponibilità selezionando le settimane scelte;
   (g) L'attore conferma i dati premendo sul pulsante di inserimento registrazione;
   (h) Il sistema aggiunge i dati al database e mostra un messaggio di avvenuta registrazione.
5. **Situazioni eccezionali**

(f) Il sistema mostra un messaggio d'errore se l'attore seleziona un numero di settimane inferiore ad un minimo prestabilito;

(h) L'attore risulta già inserito nel sistema.

6. **Postcondizioni**
   L'attore viene registrato con successo nel sistema.

7. **Casi d'uso correlati**
   /

UC2: Login

1. **Attori**
   Animatore

2. **Descrizione**
   L'attore effettua l'accesso al sistema.

3. **Precondizioni**
   L'attore deve essere già registrato nel sistema.

4. **Passaggi principali**

   (a) L'attore preme il pulsante di *login*;

   (b) L'attore inserisce le proprie credenziali nella pagina di login mostrata dal sistema e preme il pulsante di conferma;

   (c) Il sistema verifica la presenza dell'attore nel database e acconsente il suo accesso.

5. **Situazioni eccezionali**

   (c) Il sistema mostra un messaggio d'errore se l'attore non è presente nel database o se inserisce le credenziali sbagliate.

6. **Postcondizioni**
   L'attore accede al sistema.

7. **Casi d'uso correlati**
   /

UC3: Appello

1. **Attori**
   Animatore referente di squadra
2. **Descrizione**
   L'attore effettua l'appello in base al momento della giornata.
3. **Precondizioni**
   /
4. **Passaggi principali**
   (a) L'attore preme il pulsante di compilazione appello;
   (b) Il sistema mostra all'attore l'elenco dei bambini/ragazzi della sua squadra che dovrebbero essere presenti in quella determinata fase della giornata;
   (c) L'attore *flagga* i nomi di chi è effettivamente presente;
   (d) L'attore preme il pulsante di conferma;
   (e) Il sistema inserisce i dati nel database affinché siano utilizzabili in altre funzioni.
5. **Situazioni eccezionali**
   /
6. **Postcondizioni**
   /
7. **Casi d'uso correlati**
   Incluso in UC4, UC5

UC4: AppelloMattutino

1. **Attori**
   Animatore referente di squadra
2. **Descrizione**
   L'attore effettua l'appello mattutino, il quale include l'elenco per la mensa.

3. **Precondizioni**
   /

4. **Passaggi principali**

   (a) UC3: Appello

   (b) Prima di premere il pulsante di conferma, l'attore deve *flaggare* anche i nomi dei bambini/ragazzi che si fermano in mensa, specificando la preferenza di ognuno sulle variazioni del pasto;

   (c) Il sistema provvede all'aggregazione continua dei dati di tutte le squadre, per poter poi comunicare i numeri totali ai responsabili della mensa.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   /

7. **Casi d'uso correlati**
   Include UC3

UC5: AppelloPomeridiano

1. **Attori**
   Animatore referente di squadra

2. **Descrizione**
   L'attore effettua l'appello pomeridiano.

3. **Precondizioni**
   /

4. **Passaggi principali**

   (a) UC3: Appello

   (b) Prima di premere il pulsante di conferma, l'attore deve *flaggare* nuovamente i nomi dei bambini/ragazzi che erano già presenti al mattino, confermandone la presenza anche al pomeriggio;

(c) Il sistema provvede all'aggregazione continua dei dati di tutte le squadre, per poter poi comunicare i numeri totali ai responsabili della merenda.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   /

7. **Casi d'uso correlati**
   Include UC3

UC6: ModificaDisponibilità

1. **Attori**
   Animatore

2. **Descrizione**
   L'attore modifica le proprie disponibilità di presenze al centro estivo.

3. **Precondizioni**
   /

4. **Passaggi principali**

   (a) L'attore preme il pulsante di modifica disponibilità

   (b) L'attore specifica le nuove disponibilità oppure modifica quelle già indicate, sia settimanali che giornaliere;

   (c) Il sistema provvede all'accettazione delle nuove date e a notificare i coordinatori.

5. **Situazioni eccezionali**

   (b) Il sistema mostra un avvertimento, indicante la necessità di avere un confronto diretto con un coordinatore se la richiesta di variazione risultasse troppo eccessiva rispetto a quanto specificato in fase di iscrizione.

6. **Postcondizioni**
   Le disponibilità dell'attore vengono modificate di conseguenza, in base alla richiesta inoltrata.

7. **Casi d'uso correlati**
   /

UC7: IscrizioneGita

1. **Attori**
   Animatore

2. **Descrizione**
   L'attore provvede ad iscriversi nell'elenco degli animatori disponibili come accompagnatori per la gita settimanale.

3. **Precondizioni**
   /

4. **Passaggi principali**

   (a) L'attore preme il pulsante di iscrizione alla gita;

   (b) Il sistema mostra un riepilogo delle informazioni riguardanti la gita: destinazione, orari, attività; chiede poi all'attore di confermare la disponibilità;

   (c) L'attore conferma la propria disponibilità premendo l'apposito pulsante.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   L'attore risulta inserito nella lista degli accompagnatori per la gita settimanale.

7. **Casi d'uso correlati**
   /

UC8: ConfermaPresenza

1. **Attori**
   Animatore
2. **Descrizione**
   L'attore conferma la propria presenza.
3. **Precondizioni**
   L'attore è già presente nell'elenco in questione.
4. **Passaggi principali**
   (a) Il sistema, entro determinati limiti di tempo, propone all'attore di confermare la propria presenza;
   (b) Il sistema mostra un avviso che recita "Confermi la tua presenza per ..?" e propone un tasto di conferma e un tasto di rifiuto;
   (c) L'attore preme il pulsante concorde alle proprie intenzioni;
   (d) Il sistema reagisce di conseguenza (*cfr. UC9, UC10*).
5. **Situazioni eccezionali**
   /
6. **Postcondizioni**
   Se l'attore conferma la presenza, il suo nome viene lasciato nel relativo elenco. In caso di rifiuto, il suo nome viene rimosso dall'elenco e viene emessa una notifica ai coordinatori.
7. **Casi d'uso correlati**
   Generalizzazione di UC9, UC10.

UC9: ConfermaPresenzaGiornoSuccessivo

1. **Attori**
   Animatore
2. **Descrizione**
   L'attore conferma la propria presenza per la giornata successiva.
3. **Precondizioni**
   L'attore è già presente nell'elenco della giornata successiva.

4. **Passaggi principali**

   (a) Il sistema, il giorno precedente, propone all'attore di confermare la propria presenza per il giorno successivo;

   (b) Il sistema mostra un avviso che recita "Confermi la tua presenza per domani?" e propone un tasto di conferma e un tasto di rifiuto;

   (c) L'attore preme il pulsante concorde alle proprie intenzioni;

   (d) Il sistema reagisce di conseguenza, lasciando il nome dell'attore in elenco oppure rimuovendolo se rifiuta.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   L'elenco del giorno successivo viene aggiornato in base alla scelta dell'attore.

7. **Casi d'uso correlati**
   Specializzazione di UC8.

UC10: ConfermaPresenzaSettimanale

1. **Attori**
   Animatore

2. **Descrizione**
   L'attore conferma la propria presenza per la settimana successiva.

3. **Precondizioni**
   L'attore è già presente nell'elenco della settimana successiva.

4. **Passaggi principali**

   (a) Il sistema, alla fine di una settimana, propone all'attore di confermare la propria presenza per quella successiva;

   (b) Il sistema mostra un avviso che recita "Confermi la tua presenza per settimana prossima?" e propone un tasto di conferma e un tasto di rifiuto;

   (c) L'attore preme il pulsante concorde alle proprie intenzioni;

(d) Il sistema reagisce di conseguenza, lasciando il nome dell'attore in elenco oppure rimuovendolo se rifiuta.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   L'elenco della settimana successiva viene aggiornato in base alla scelta dell'attore.

7. **Casi d'uso correlati**
   Specializzazione di UC8.

UC11: Logout

1. **Attori**
   Animatore

2. **Descrizione**
   L'attore si disconnette dal sistema.

3. **Precondizioni**
   L'attore è attualmente connesso nel sistema.

4. **Passaggi principali**

   (a) L'attore preme il pulsante di *logout*;

   (b) Il sistema mostra un avviso in cui chiede conferma dell'effettiva volontà di disconnettersi dal sistema;

   (c) L'attore preme il pulsante di conferma;

   (d) Il sistema rimuove le credenziali dell'attore e mostra nuovamente la pagina iniziale in cui potersi registrare o fare login.

5. **Situazioni eccezionali**
   /

6. **Postcondizioni**
   L'attore è disconnesso dal sistema.

7. **Casi d'uso correlati**
   /

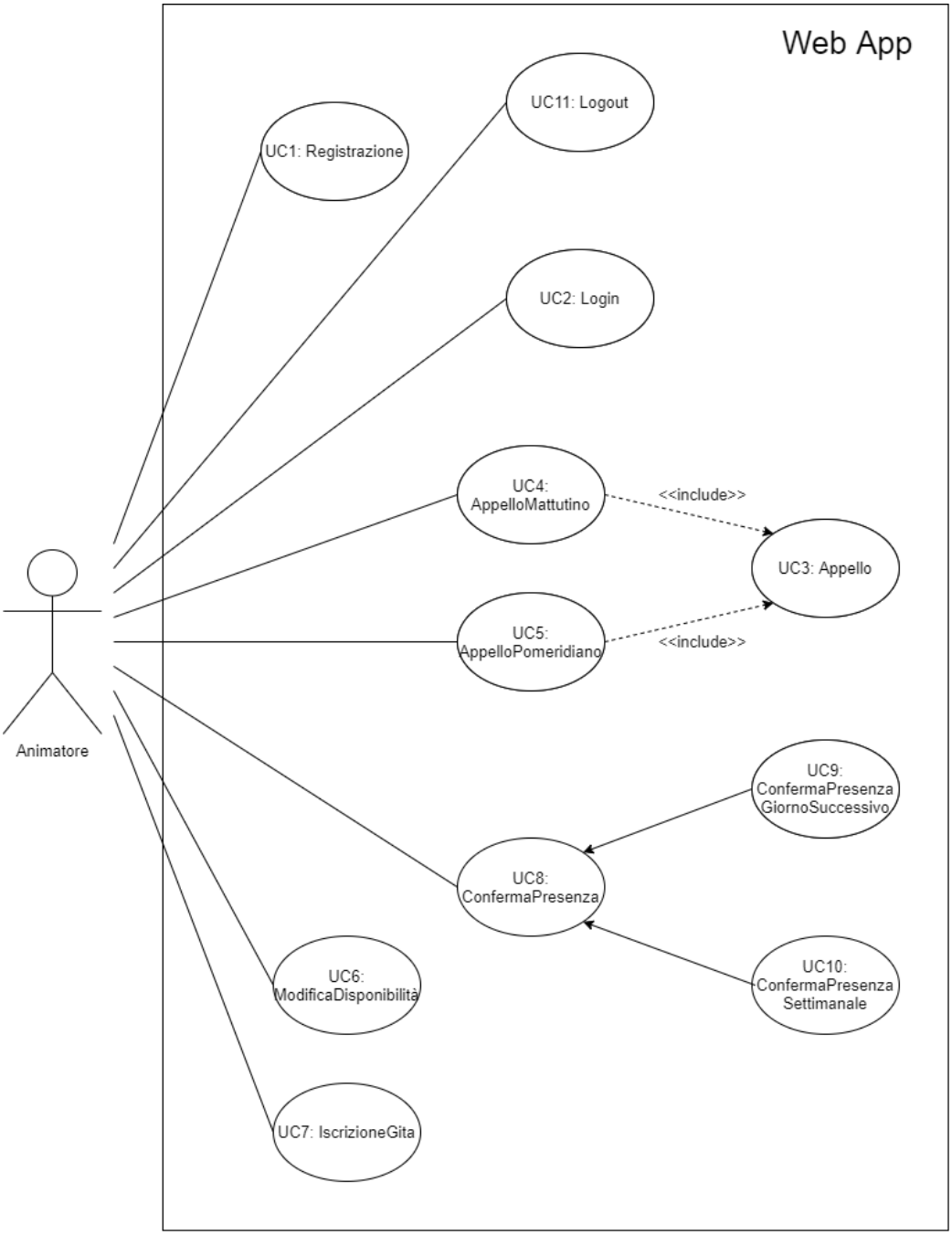


Figura 2.1: Diagramma dei casi d'uso.

2.3.3 Requisiti funzionali

I requisiti, o specifiche, funzionali sono funzioni e servizi che il software deve poter offrire agli utenti. Nel caso in questione, visti anche i sopraelencati casi d'uso, i requisiti funzionali che verranno presi in considerazione sono i seguenti:

- **Registrazione, login e logout**: il sistema deve contemplare la necessità di potersi registrare ed avere un proprio profilo utente, al quale verranno associati tutti i dati personali, le proprie disponibilità e le presenze.
- **Visualizzazione attività quotidiane**: il software deve fornire una panoramica sulle attività schedulate per la giornata corrente.
- **Compilazione appello**: il sistema mette a disposizione una versione informatizzata del processo di appello che l'animatore deve compilare quotidianamente, anche più volte al giorno a seconda del momento della giornata.
- **Iscrizione alle gite**: il programma offre la possibilità agli animatori di iscriversi e/o confermare la propria presenza alla gita settimanale.
- **Conferma presenza**: il software deve proporre un avviso di notifica, alla fine di una giornata e alla fine di ogni settimana, tramite il quale confermare la propria presenza al periodo in questione, se si è già iscritti al suddetto periodo.

ID	Nome	Priorità	Implementazione
01	Registrazione	Alta	Sì
02	Login	Alta	Sì
03	Logout	Alta	Sì
04	Visualizzazione attività	Media	Sì
05	Compilazione appello	Media	Sì
06	Iscrizione gite	Bassa	No
07	Conferma presenza	Bassa	No

Tabella 2.1: Stack dei requisiti funzionali

2.4 Studio di fattibilità

Per la specifica tipologia del prodotto in questione, si è pensato ad una web application che sia accessibile via browser, sia da dispositivi mobili che da ambienti desktop. In questo modo si evita di dover sviluppare un'applicazione nativa per i diversi SO mobile, oltre ad un software apposito per i sistemi fissi.
Per questo motivo, il modello applicato è quello della **SPA**, *Single Page Application*, una pagina web che non viene ricaricata o alternata ad altre pagine, ad ogni richiesta dell'utente, bensì si adatta e aggiorna la propria visualizzazione in base al contenuto da mostrare che man mano viene recuperato dal server. Questo permette di fornire un'esperienza utente più fluida e simile alle applicazioni tradizionali.

Dal punto di vista dell'utente, una soluzione di questo tipo è sicuramente più accessibile e immediata, in quanto non necessita di dover essere scaricata e installata. Inoltre, con la sempre maggiore diffusione delle **PWA**, *Progressive Web Applications*, è ipotizzabile convertire il servizio in una di esse tramite pochi passaggi, usufruendo delle funzionalità delle app native (icona individuale, notifiche push, ecc.) senza soffrire dei suddetti vincoli di installazione.

Capitolo 3

Envisioning architetturale

3.1 Pattern architetturale

Il pattern architetturale fondamentale selezionato per lo sviluppo dell'applicazione è di tipo **client-server**, variante **three-tier**, caratterizzata dalla presenza di 3 livelli fisici: una macchina client, sulla quale viene eseguito il layer GUI e parte dell'Application logic, una macchina server sulla quale viene eseguita la restante parte dell'Application logic, e un database gestito da MongoDB Atlas, uno dei più noti servizi di cloud database. Per l'implementazione di una simile architettura è stato adoperato lo **stack MEAN**, illustrato di seguito.

3.2 Stack MEAN

MEAN è uno stack JavaScript end-to-end largamente utilizzato per sviluppare applicazioni in hosting sul cloud dinamiche e responsive.
I vantaggi di questo stack open source sono la scalabilità e l'utilizzo dello stesso linguaggio per tutte le sue componenti, oltre che ad una buona gestione degli accessi contemporanei. Inoltre, il framework frontend Angular lo rende ideale per sviluppare le suddette applicazioni SPA, che forniscono tutte le informazioni e le funzionalità su un'unica pagina.
Lo stack MEAN supporta il pattern architetturale MVC, Model View Controller ???

3.2.1 Componenti

- **MongoDB**: MongoDB è un database NoSQL open source, progettato per le applicazioni cloud. Utilizza un'organizzazione orientata agli oggetti invece di un modello relazionale. Nel caso di CREstionale, conterrà i dati come file JSON, racchiusi in collezioni.

- **Express**: Express è un framework di applicazioni web per Node.js. Formando il backend dello stack MEAN, Express gestisce tutte le interazioni tra il frontend e il database, garantendo un regolare trasferimento dei dati all'utente finale. È progettato per essere usato con Node.js e quindi garantisce continuità d'uso coerente di JavaScript in tutto lo stack.

- **Angular**: Angular è il framework JavaScript per il frontend di applicazioni web sviluppato principalmente da Google e reso disponibile in modo open source. Basato su TypeScript, un super-set di JavaScript, il codice generato da Angular può essere eseguito su tutti i principali web browser moderni, rendendolo di fatto compatibile con qualunque dispositivo connesso al web. Angular permette molto facilmente l'adattabilità della visualizzazione in base al dispositivo utilizzato, rendendo l'applicazione responsive.

- **Node.js**: Node.js è un Runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Si tratta di un framework ideale per un'applicazione basata sul cloud, in quanto può scalare facilmente su richiesta. Node.js utilizza un modello I/O event-driven e non bloccante, che lo rende leggero ed efficiente, particolarmente adatto per applicazioni real-time che possono essere eseguite su molti dispositivi contemporaneamente.

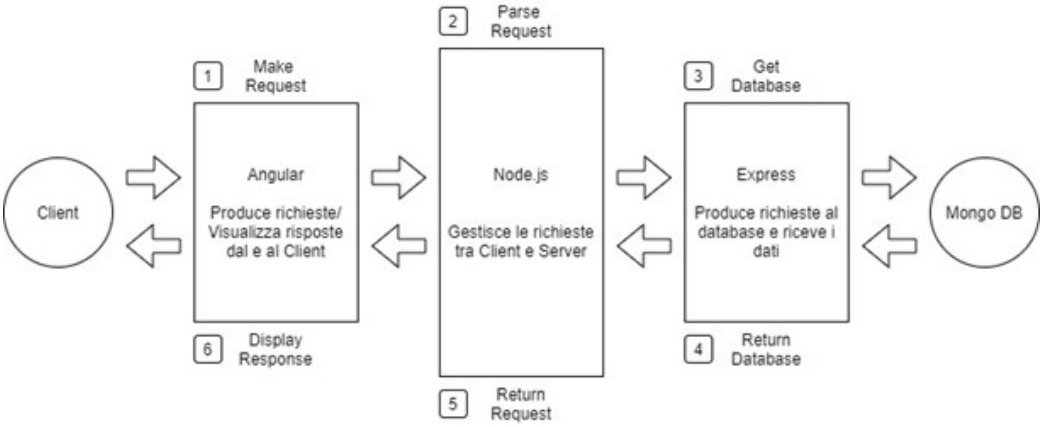


Figura 3.1: Architettura dello stack

3.3 Deployment diagram

Il deployment diagram mostra le componenti del sistema per come sono allocate e istanziate sui nodi computazionali, indicando, in questo modo, le risorse fisiche su cui il sistema è deployato. Nel caso di CREstionale, il deployment diagram mostra come il database sia allocato su Atlas, all'interno di uno spazio server dedicato, mentre la parte frontend di Angular e il server di Express/Node.js siano inizialmente in esecuzione su localHost. Quando il progetto verrà deployato ufficialmente, il deployment diagram sarà diverso (Angular e Node.js/Express su server diversi).

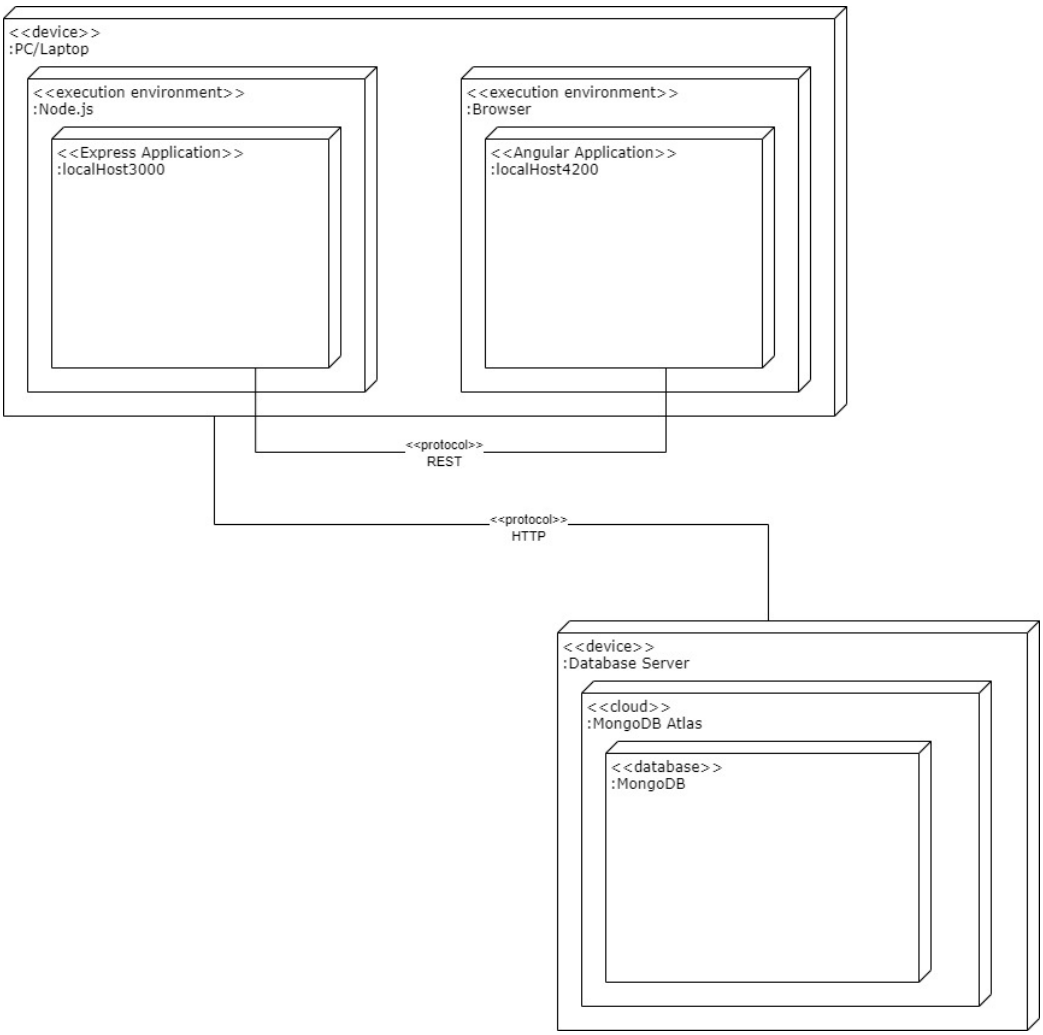


Figura 3.2: Deployment diagram

3.4 Toolchain

Per la realizzazione del progetto verranno utilizzate i seguenti strumenti e tecnologie:

- Visual Studio Code, IDE di Microsoft;
- MongoDB Atlas, servizio di gestione per il database MongoDB;

- AWS, servizio di Amazon di affitto spazio cloud;
- Angular CLI, interfaccia a linea di comando per la gestione dei componenti Angular;
- NPM, manager dei pacchetti per la condivisione e l'importazione di moduli JavaScript;
- Express, web framework per Node.js
- Bulma, framework CSS che fornisce gli stili per le pagine HTML;
- Mongoose, strumento di modellazione per oggetti su MongoDB;
- Bcrypt, libreria per generare l'hash di stringhe (password);
- nodemon, tool per il riavvio automatico del server quando vengono rilevate modifiche;
- Postman, piattaforma per lo sviluppo e l'utilizzo di API;
- GitHub;
- WhatsApp, Telegram;
- Trello

3.5 Best practices agili

Per la realizzazione del progetto sono state utilizzate le seguenti best practices agili:

- Pair programming - si è sempre lavorato in coppia sul codice, alternando le figure del **driver** e del **navigator**;
- Time boxing - quando possibile, si è cercato di rispettare una certa regolarità tra il tempo di lavoro e le pause;
- Integrazione continua (git) - è stato utilizzato *git* come sistema di versioning del codice, in modo tale da poter integrare in maniera continuativa il codice ad ogni piccolo incremento;

- Collaborazione col cliente - in questo caso, il cliente è uno dei due autori, quindi è stato coinvolto pienamente nell'intera realizzazione del progetto;
- Modifica e affinamento dei requisiti in corso d'opera;
- Stack dei requisiti - vedasi paragrafo 2.3.3;

Parte II

Iterazione 1

Capitolo 4

Introduzione

In questa iterazione vengono poste le basi per lo sviluppo effettivo del servizio, e sono implementate alcune delle funzionalità fondamentali dell'applicazione CREstionale. Nello specifico, si costruirà la landing page del servizio, la visualizzazione che dà il benvenuto all'utente e spiega le potenzialità e la struttura dell'applicazione. All'interno di essa si sviluppano la navbar, il footer, e le funzioni di registrazione e login/logout.
I casi d'uso formalizzati sono UC1 e UC2 (+UC11), racchiusi dai requisiti funzionali 01, 02, 03, di priorità alta.

4.1 Component diagram

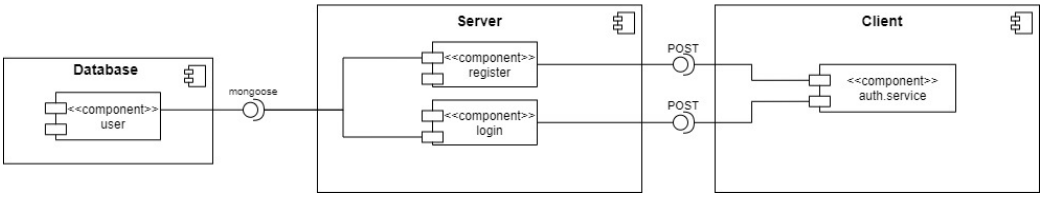


Figura 4.1: Component diagram

Nel *component diagram* viene data enfasi ai componenti del sistema e alle loro interdipendenze. Non viene più mostrata l'allocazione fisica delle singole parti del progetto, bensì si preferisce un'aggregazione concettuale dei componenti. Le dipendenze tra di essi vengono specificate con la notazione

ball-and-socket (detta anche *lollipop*), dove il pallino pieno rappresenta l'interfaccia esposta da una determinata componente, mentre il mezzo pallino indica un'interfaccia richiesta da una componente. Nel caso di CREstionale, è possibile osservare come la componente per l'autenticazione offra, tramite l'insieme di vincoli architetturali REST, un'interfaccia al client per poter sfruttare le funzioni di login e registrazione. Inoltre, il server espone un'interfaccia, tramite la libreria Mongoose, per la schematizzazione e la validazione dei dati del database, che per loro natura, proveniendo da un DB NoSQL, non godono di queste caratteristiche di default.

4.2 Deployment diagram

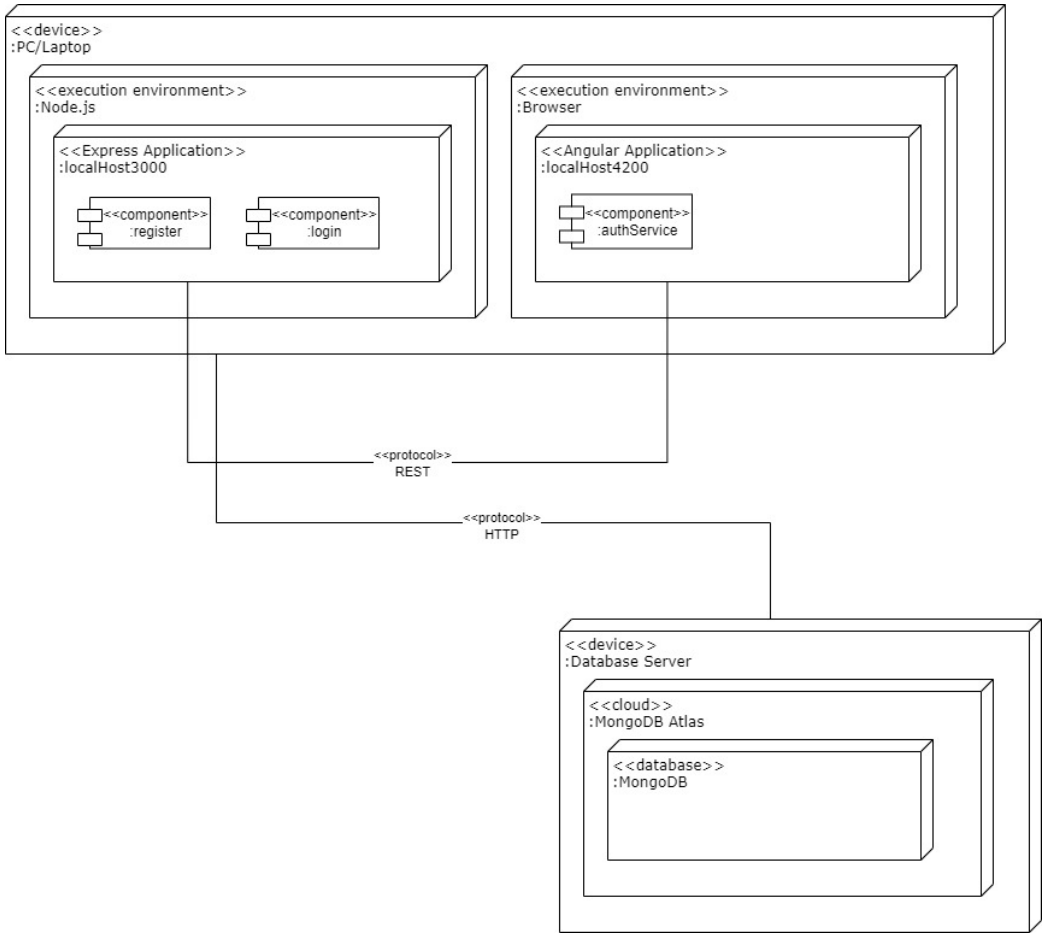


Figura 4.2: Deployment diagram

L'aggiornamento del *deployment diagram* per l'iterazione 1 presenta i nuovi componenti che verranno introdotti, i quali hanno il compito di gestire tutte le funzionalità legate all'autenticazione.

4.3 Impostazioni preliminari

4.3.1 Bulma CSS

Bulma è un framework CSS gratuito e open-source che fornisce componenti facilmente combinabili per la realizzazione di interfacce web *responsive*. Una delle caratteristiche più interessanti offerte da Bulma è la suddivisione della visualizzazione della pagina in colonne, e la conseguente responsività del layout in base al dispositivo utilizzato. Le colonne si auto-adattano per sfruttare al meglio lo spazio disponibile, diventando così fruibili anche da browser mobile, principale strumento utilizzato dall'utente tipo di CREstionale per i motivi precedentemente indicati.

4.3.2 Navbar e Footer

La barra di navigazione e il pié di pagina sono due componenti importanti nella visualizzazione di un servizio web. La navbar include alcuni dei comandi principali, come i pulsanti di login e registrazione, e attraverso un menu a cascata è possibile utilizzare alcune funzioni aggiuntive. Questo componente è responsive dato che collassa nel momento in cui la larghezza della finestra di visualizzazione scende sotto un valore minimo, andando così ad abilitare una visualizzazione alternativa delle funzionalità della navbar, ovvero il menu hamburger.
Sono due componenti che vengono sempre mostrati in qualsiasi sezione dell'app, perché all'interno dell'app-component, componente principale dell'app che gestisce la visualizzazione di tutti gli altri, gli elementi della barra di navigazione e del pié di pagina sono posizionati all'esterno di <router-outlet>, il tag che racchiude la sezione routata dall'URL.

```
<!DOCTYPE html>
<body>
  <app-navbar></app-navbar>
  <router-outlet></router-outlet>
  <br>
  <app-footer></app-footer>
</body>
```

Listing 4.1: Struttura dell'applicazione

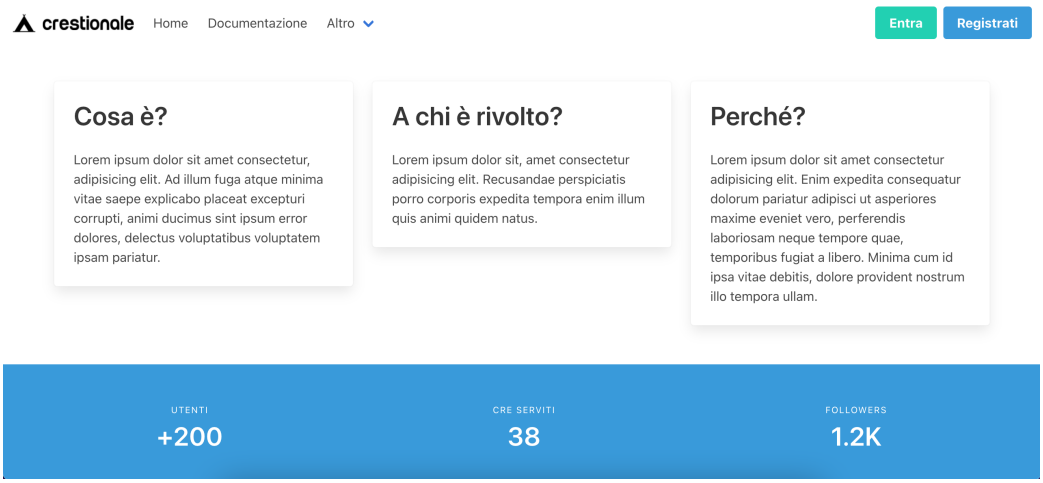


Figura 4.3: Landing page

4.3.3 Landing page

La landing page è la pagina che accoglie l'utente nel momento in cui raggiunge il servizio per la prima volta, prima ancora di aver effettuato alcuna operazione di registrazione. Questa visualizzazione ha il compito di informare chi legge riguardo le funzionalità e le potenzialità del servizio offerto da CREstionale, mostrando anche, per esempio, alcune recensioni ed esperienze lasciate da precedenti utenti. Nel momento in cui l'utente si autentica tramite registrazione o login, come si vedrà meglio nella sezione relativa all'autenticazione, viene rilasciato un token che ha una durata prestabilita. Se l'utente abbandona il servizio e rientra entro la scadenza del token, verrà reindirizzato automaticamente dalla landing page alla home page, permettendogli di interagire direttamente con le funzioni del servizio e i dati a lui collegati.

4.3.4 Creazione del server

```
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')

const PORT = 3000
const api = require('./routes/api')
```

```
const app = express()
app.use(cors())
app.use(bodyParser.json())

app.use('/api', api)

app.get('/', function(req, res){
    res.send('Hello from server')
})

app.listen(PORT, function() {
    console.log('Server running on localhost:' + PORT)
})

module.exports = app
```

Listing 4.2: Creazione del server

Il codice qui sopra costituisce la creazione del server Express e il collegamento di alcuni middleware. Innanzitutto, vengono importati i moduli di Express, Body-parser e CORS, rispettivamente il server framework e due middleware utilizzati per la decodifica delle richieste HTTP e per la condivisione di risorse cross-origin. Viene specificata la porta sulla quale sarà in esecuzione il server, poi viene creato un collegamento con le API (ulteriori informazioni nella prossima sezione). La costante 'app' viene creata come istanza di Express, e costituisce il server vero e proprio. Dopo aver specificato l'utilizzo dei middleware e delle API sopra descritte, si indica su quale porta il server dovrà essere eseguito con il metodo *listen(PORT)*.

4.4 Sviluppo dell'autenticazione - UC1, UC2, UC11

In questa sezione si entra nel clue dello sviluppo delle prime funzionalità vere e proprie di CREstionale. Nello specifico, di seguito verranno descritti i passaggi svolti per l'implementazione della componente di autenticazione, ovvero la possibilità degli utenti di registrarsi all'applicazione tramite alcuni dati personali e i campi che verranno utilizzati per interfacciarsi con le funzionalità più avanzate del sistema, sviluppate nell'iterazione successiva.

4.4.1 UC1: registrazione

Creazione modello di dato 'user' nel database

Per poter permettere la registrazione dell'utente, quindi un animatore/coordinatore, è necessario predisporre un form di registrazione dove inserire le proprie generalità. Prima di ciò, il sistema necessita di una schematizzazione dei dati dell'utente, di modo da poter essere meglio parametrizzati all'interno del database. Si sfrutta, in questo senso, la modellizzazione fornita dalla libreria **Mongoose**, per creare uno schema, un modello di dato 'user'.

```
const mongoose = require('mongoose');
const uniqueValidator = require('mongoose-unique-validator');

const Schema = mongoose.Schema
const userSchema = new Schema({
    name: {type: String, required: true},
    surname: {type: String, required: true},
    email: {type: String, required: true, unique: true},
    password: {type: String, required: true},
    role: {type: String, required: true}
})

userSchema.plugin(uniqueValidator);

*/
module.exports = mongoose.model('user', userSchema, 'users')
```

Listing 4.3: Creazione del modello 'user'

In questo schema si nota come i dati associati all'utente sono il nome, il cognome, la mail, una password e il ruolo (animatore/coordinatore). Il *plugin(uniqueValidator)* permette la validazione dei dati, ovvero fornirà errori di validazione specifici nel caso in cui si dovesse provare a creare un nuovo utente con le stesse informazioni di un altro già presente.

Creazione del componente Registration page

Nella parte HTML del componente viene predisposta la visualizzazione del form di registrazione, che come si può notare dallo screenshot qua sotto, richiede gli stessi dati elencati nel modello di dato 'user' appena definito.

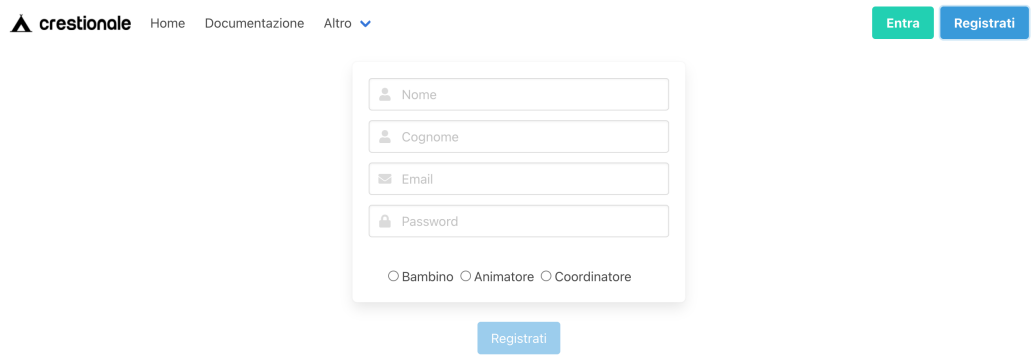


Figura 4.4: Registration page

Nello script del componente avviene la validazione dei dati inseriti, tramite la funzione *validateEmail()*, che verifica che la stringa 'email' rappresenti un indirizzo valido. Per fare ciò viene viene utilizzata un'**espressione regolare**, composta da alcuni vincoli precisi che identificano il formato corretto di un indirizzo mail.

```
validateEmail(email) {
    var mailformat = /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA
   -Z0-9-]+\.[a-zA-Z0-9-]+$/

    if(email.match(mailformat))
      return true
    else
      return false
  }
```

Listing 4.4: Script per la validazione della mail

Successivamente, il metodo *registerUser()* del servizio *authService*, invocato quando si preme il pulsante di conferma dopo aver inserito i dati nel form, effettua una richiesta HTTP di POST al database, definita e dichiarata nelle *api.js*.

```
registerUser() {

    this.emailIsInvalid = false
    this.userAlreadyExists = false

    if(!this.validateEmail(this.registerUserData.email)) {
      this.emailIsInvalid = true
    }
```

```
    else {
      this.emailIsInvalid = false
      this.registrationStatusListenerSubs = this._auth.
   registerUser(this.registerUserData)
        .subscribe(res => {
          if(!res) {
            this.userAlreadyExists = true
            this.registrationStatusListenerSubs.unsubscribe()
          }

          else {
            this.userAlreadyExists = false
            this.registrationStatusListenerSubs.unsubscribe()
          }
        })
    }
  }
```

Listing 4.5: Metodo registerUser() della Registration page

```
registerUser(user) {
    this.http.post<any>(this._registerUrl, user)
      .subscribe(response => {
        if(response.status) {
          this.registrationStatusListener.next(true)
          this.loginUser(user)
        }
        else
          this.registrationStatusListener.next(false)
      })

    return this.registrationStatusListener.asObservable()
  }
```

Listing 4.6: Metodo registerUser() dell'authService

L'authService si occupa anche di segnalare l'avvenuta registrazione, tramite l'observable *registrationStatusListener*, e di loggare automaticamente l'utente appena registrato tramite la funzione *loginUser()*.

API per la registrazione

La richiesta definita nelle API controlla che l'email specificata nella registrazione del nuovo utente non sia già presente nel database, e procede con la

creazione dell'utente tramite il servizio di *authService.ts*. Se la richiesta di registrazione non va a buon fine, perché l'utente con quella mail specifica risulta già presente nel database, viene messa a true la booleana *userAlreadyExists*, che permette poi di mostrare, nel form HTML, il corretto messaggio di errore, oltre a quello restituito dall'utilizzo di una mail non valida, controllata da *validateEmail()*.

```
router.post('/register', (req, res) => {

    bcrypt.hash(req.body.password, 10)
        .then(hash => {
            const user = new User({
                name: req.body.name,
                surname: req.body.surname,
                email: req.body.email,
                password: hash,
                role: req.body.role
            })

            user.save((error) => {
                if(error) {
                    res.json({
                        status: false
                    })
                }
                else
                    res.json({
                        status: true
                    })
            })
        })
})
```

Listing 4.7: Metodo per la registrazione lato server

Confronto con il caso d'uso ipotizzato

Nello stack dei casi d'uso ipotizzati nell'*envisioning* durante l'iterazione 0, era stato ipotizzato di poter permettere la registrazione di bambini, coordinatori e animatori nello stesso form, sfruttandone poi le distinzioni in altri moduli. Questa idea è rimasta valida, per ora, solo nel modello di dato 'utente', dove effettivamente viene chiesto, nel form di registrazione, la tipologia di utenza; quest'informazione infatti non viene poi più (o ancora) utilizzata in altri

moduli.

Anche la possibilità di specificare le proprie settimane di presenza in sede di registrazione, proposta e ipotizzata nei casi d'uso durante l'*envisioning*, non è effettivamente ancora implementata.

4.4.2 UC2: login

Creazione del componente Login page

Il componente LoginPage dispone di un HTML che indica la struttura della visualizzazione del form di login.

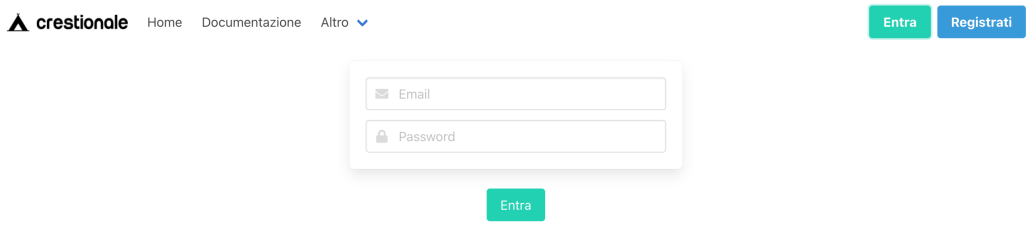


Figura 4.5: Login page

Quando si inseriscono i dati e si preme il pulsante di conferma, interviene lo script che controlla prima di tutto la validità dell'email inserita, tramite la funzione *validateEmail()*; successivamente, se questa è valida, tramite il servizio di *authService*, viene formulata una richiesta HTTP di POST, definita e dichiarata nelle API.

```
loginUser() {

    this.emailIsInvalid = false
    this.userNotExists = false
    this.passwordWrong = false

    if(!this.validateEmail(this.loginUserData.email)) {
      this.emailIsInvalid = true
    }
    else {
      this.emailIsInvalid = false
      this.loginStatusListenerSubs = this._auth.loginUser(
   this.loginUserData)
      .subscribe(res => {
```

```
        if(res) {
          if(res == 'User not exists')
            this.userNotExists = true
          else if(res == 'Password wrong')
            this.passwordWrong = true
          else
            window.alert('Ooops! Qualcosa e andato storto')
        }

        this.loginStatusListenerSubs.unsubscribe()
      })
    }

  }
```

Listing 4.8: Metodo loginUser() della Login page

```
loginUser(user) {
    this.http.post<{token: string, email: string, expiresIn:
   number}>(this._loginUrl, user)
      .subscribe(response => {
        const token = response.token
        const email = user.email
        this.token = token
        this.email = email

        if(token)
          const expiresInDuration = response.expiresIn
          this.setAuthTimer(expiresInDuration)
          this.isAuthenticated = true
          this.authStatusListener.next(true)
          const now = new Date()
          const expirationDate = new Date(now.getTime() +
   expiresInDuration * 1000)
          this.saveAuthData(token, expirationDate, email)
          this.router.navigate(['/home'])
        }
        this.loginStatusListener.next('')
      },
      error => {
        this.loginStatusListener.next(error.error.message)
      })

      return this.loginStatusListener.asObservable()
```

```
  }
```

Listing 4.9: Metodo loginUser() dell'authService

API per il login

Questa richiesta verifica se nel database esiste un utente con quella mail, controlla la password, e in caso positivo autentica l'utente, restituendo un token di autenticazione che ha una durata prestabilita. Questi dati vengono restituiti in risposta all'authService, che si occupa di informare, tramite observable, dell'avvenuta autenticazione, settando i campi *authStatusListener* e *isAuthenticated*. Se la richiesta di post('/login') non trova riscontro sia per mail che per password, solleva un errore che viene restituito all'authService, il quale, in mancanza di token, comunica solo il messaggio di errore, tramite observable. In questo modo, il login-page.component è a conoscenza dello specifico errore di login, e può comunicare all'utente in maniera più efficace, mostrando un messaggio specifico e modificando alcune parti della visualizzazione (casella del form con contorno rosso).

```
router.post('/login', (req, res) => {
    let userData = req.body

    User.findOne({ email: userData.email }).then(user => {

        if(!user)
            throw new Error('User not exists')
        return bcrypt.compare(userData.password, user.
   password)
    })
    .then(result => {
        if(!result)
            throw new Error('Password wrong')

        const token = jwt.sign(
            {email: userData.email},
            'thats_the_secret_key',
            {expiresIn: '1h'})

        res.status(200).json({
            token: token,
            email: userData.email,
            expiresIn: 3600 // in secondi, tempo in cui il
   token scade
```

```
        })
    })
    .catch(err => {
        return res.status(401).json({message: err.message})
    })

})
```

Listing 4.10: Metodo per il login lato server

In ultimo, se l'autenticazione va a buon fine, si reindirizza l'utente sulla home.

Sequence diagram del login

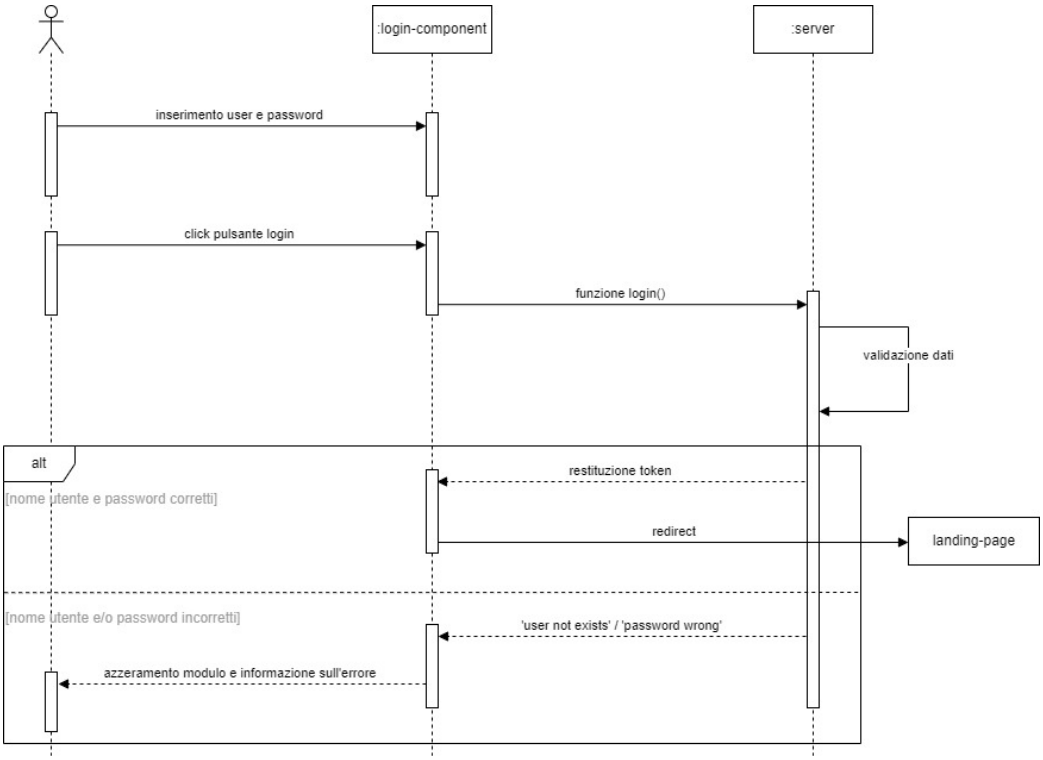


Figura 4.6: Sequence diagram per la procedura di login

4.4.3 UC11: logout

Alla pressione del pulsante di logout, presente nella navbar persistente e disponibile solamente dopo essersi autenticati, viene invocato il metodo *lo-*

goutUser() di *authService*, il quale cancella tutte le informazioni salvate della sessione dell'utente autenticato, ne rimuove il token e condivide l'informazione di logout. Infine, si reindirizza la visualizzazione sulla landing page '/', la quale ripresenterà la possibilità di registrarsi o di fare login.

```
logoutUser() {
    this.token = null
    this.isAuthenticated = false
    this.authStatusListener.next(false)
    clearTimeout(this.tokenTimer)
    this.clearAuthData()
    this.router.navigate(['/'])
  }
```

Listing 4.11: Metodo per il logout

Confronto con il caso d'uso ipotizzato

Durante l'*envisioning* dell'iterazione 0 era stato ipotizzato di chiedere conferma all'utente prima di effettuare il logout. Questa conferma non è implementata, e nel momento in cui si preme sul pulsante, il logout avviene istantaneamente.

4.5 Testing

4.5.1 Introduzione al testing

La fase di testing del software sta assumendo molta importanza nel corso del tempo in quanto diversi studi hanno dimostrato come lo sforzo impiegato da un programmatore nel realizzare dei test viene ampiamente ripagato con del codice di miglior qualità e più manutenibile nel corso del tempo.
Per questo motivo, Angular viene fornito con due componenti built-in per il testing come **Jasmine**, un test framework contenente tutto l'essenziale per scrivere i test, e **Karma**, un test runner che esegue un webserver sul quale vengono effettuati in maniera automatica tutti i test definiti dall'utente ogniqualvolta un file viene modificato.

Per questo progetto si è deciso, prima di passare all'iterazione successiva, di implementare dei **test unitari** dei principali componenti, ovvero dei test riferiti a singole unità di software, completamente slegate dal resto del sistema. Questa fase viene quindi svolta sia per l'iterazione 1, sia per tutte

le successive iterazioni, qualora si andasse ad implementare un incremento software corposo tale da necessitare degli specifici test.

4.5.2 Elementi comuni del testing

Ogni file di test definito per un componente Angular deve avere lo stesso nome del componente stesso, con in aggiunta l'estensione *.spec.ts*. Ad esempio, se si vuole testare il componente *navbar.component.ts*, il rispettivo file di test avrà il nome *navbar.component.spec.ts* e, per convenzione, verrà inserito nella stessa cartella.
La prima parte di ogni test consiste nell'andare a descrivere la suit di test e in Jasmine è possibile fare ciò utilizzando la funzione *describe*, che riceve come primo parametro il nome della suit, che tipicamente è lo stesso del componente, e come secondo parametro una funzione contenente i vari test da effettuare. Questi test, più nel dettaglio chiamati *specifiche*, vanno definiti usando la parola chiave *it*, seguita dal nome del singolo test e da un'altra funzione nella quale si andrà a definire nel dettaglio tutte le istruzioni che devono essere eseguite.
Tipicamente, ciascuno dei test eseguiti in una specifica suit condividono delle operazioni preliminari di inizializzazione, e per evitare ridondanza nel codice andando a definirle in ogni singola specifica è possibile inserirle in un metodo *beforeEach*, che viene eseguito automaticamente prima di ciascun test.

4.5.3 Testing della navbar

Il primo componente di cui è stato fatto il test unitario è la navbar, andando a definire le seguenti specifiche.

```
it('should have "Entra" button if user is not authenticated',
    () => {
    expect(de.query(By.css('.button.is-primary'))
        .nativeElement.innerText).toBe('Entra')
})

it('should have "Registrati" button if user is not
   authenticated',
    () => {
    expect(de.query(By.css('.button.is-info'))
        .nativeElement.innerText).toBe('Registrati')
})
```

```
it('should have "Esci" button if user is authenticated',
    () => {
    component.userIsAuthenticated = true
    fixture.detectChanges()
    expect(de.query(By.css('.button.is-danger'))
        .nativeElement.innerText).toBe('Esci')
})

it('should have dropdown if "Altro" button is hovered',
    () => {
    de.query(By.css('#other-button'))
        .triggerEventHandler('mouseover', {})
    fixture.detectChanges()
    const e = de.query(By.css('#other-dropdown'))
        .nativeElement
    expect(getComputedStyle(e).opacity).toEqual('1')
})
```

Listing 4.12: Testing della navbar

La prime due specifiche vanno a verificare che, in caso l'utente non sia autenticato (quindi, il caso di default), siano presenti nella navbar i due pulsanti *Entra* e *Registrati*. Questo è possibile andando a fare una query con il selettore CSS opportuno, andando a verificare l'esistenza dell'elemento renderizzato e la correttezza del testo contenuto al suo interno.
La terza, al contrario, verifica che, se l'utente è autenticato, sulla navbar è presente il pulsante di *Logout*. Per fare ciò, come si può notare nel codice sopra illustrato, si va ad assegnare alla variabile *userIsAuthenticated* il valore *true*, in modo tale da simulare l'avvenuta autenticazione da parte dell'utente.
Infine, con l'ultima specifica è possibile verificare che, se il mouse passa sopra l'elemento *Altro* nella navbar, appaia correttamente un menu dropdown.

4.5.4 Testing del form di registrazione

Il secondo componente che si va a testare è quello relativo al form con il quale è possibile effettuare la registrazione all'applicazione.

```
it('should not display error message if email is valid',
   async(() => {
    fixture.whenStable().then(() => {
        let name = de.query(By.css('#name-field'))
            name.nativeElement.value = 'Luca'
            name.nativeElement
                .dispatchEvent(new Event('input'))
```

```
        let surname = de.query(By.css('#surname-field'))
            surname.nativeElement.value = 'Assolari'
            surname.nativeElement
                .dispatchEvent(new Event('input'))
        let email = de.query(By.css('#email-field'))
            email.nativeElement
                .value = 'luca.assolari405@gmail.com'
            email.nativeElement
                .dispatchEvent(new Event('input'))
        let password = de.query(By.css('#password-field'))
            password.nativeElement.value = 'prova'
            password.nativeElement
                .dispatchEvent(new Event('input'))
        de.query(By.css('#checkbox')).nativeElement.click()
        de.query(By.css('button')).nativeElement.click()
        fixture.detectChanges()
        expect(de.query(By.css('#invalid-email'))).toBeNull()
    })
}))

it('should display error message if email is not valid',
    async(() => {
     fixture.whenStable().then(() => {
        let name = de.query(By.css('#name-field'))
            name.nativeElement.value = 'Luca'
            name.nativeElement
                .dispatchEvent(new Event('input'))
        let surname = de.query(By.css('#surname-field'))
            surname.nativeElement.value = 'Assolari'
            surname.nativeElement
                .dispatchEvent(new Event('input'))
        let email = de.query(By.css('#email-field'))
            email.nativeElement
                .value = 'luca.assolari405@gmail'
            email.nativeElement
                .dispatchEvent(new Event('input'))
        let password = de.query(By.css('#password-field'))
            password.nativeElement.value = 'prova'
            password.nativeElement
                .dispatchEvent(new Event('input'))
        de.query(By.css('#checkbox')).nativeElement.click()
        de.query(By.css('button')).nativeElement.click()
        fixture.detectChanges()
        expect(de.query(By.css('#invalid-email')))
            .toBeDefined()
```

```
    })
}))
```

Listing 4.13: Testing del form di registrazione

In questo caso, l'unica funzionalità che è possibile testare lato frontend è quella relativa alla validazione della email. Infatti, se il form viene riempito e si preme il pulsante per registrarsi, nel caso in cui la mail inserita non sia una mail valida appare un messaggio d'errore, che informa l'utente in modo tale che possa correggere; nel caso in cui, al contrario, la mail inserita è corretta, non deve apparire nulla.

4.5.5 Testing del form di login

Il form di login si comporta esattamente come quello relativo alla registrazione, ovvero con la validazione della mail nel caso in cui non venisse inserita correttamente. Per questo motivo, si è scelto di non spiegare il test effettuato in modo tale da evitare ripetizioni.

4.5.6 Risultato dei test

Una volta implementata la logica di test, è possibile eseguirli tramite il comando **ng test**; verrà aperta una pagina web con Karma, sulla quale verrà eseguito il testing e sarà possibile monitorarne il risultato. La seguente immagine mostra il risultato dei test descritti in precedenza, inclusi quelli della componente per il login.

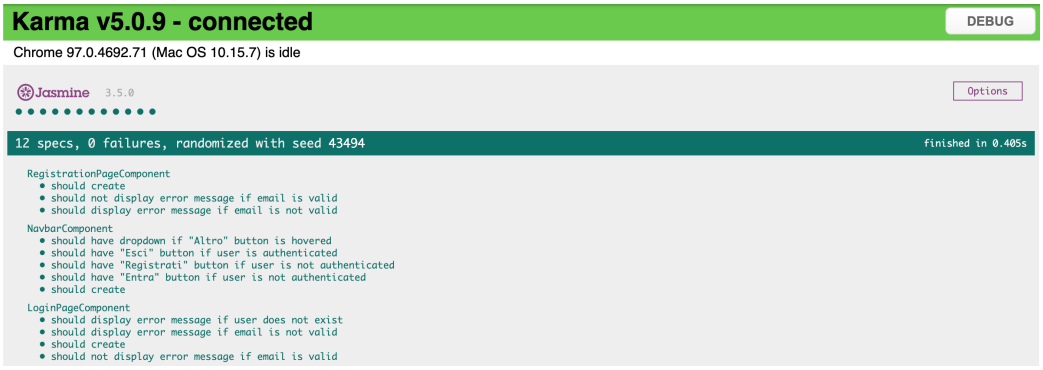


Figura 4.7: Risultato dei test della prima iterazione

4.5.7 Testing API

Lato backend, è possibile testare il corretto funzionamento delle API tramite Postman, un tool utile per simulare richieste HTTP fornendo direttamente il corpo della richiesta ad uno specifico endpoint, senza il bisogno di effettuarle realmente all'interno dell'app. Le due API finora presenti sono quelle relative alla registrazione e al login, che fondamentalmente effettuano due azioni molto simili: vengono inviati, tramite una richiesta POST, i dati dell'utente, i quali vengono processati lato backend e, sulla base dell'esito di questo processamento viene ritornato un risultato positivo o negativo. Per via di questa similarità, si è scelto di mostrare il test di una sola di queste, in particolare l'API relativa al login. Supponendo di avere un utente registrato con email *luca.assolari405@gmail.com* e password *password*, nelle due seguenti immagini è possibile notare l'esito della richiesta fornendo, rispettivamente, una password corretta, una password errata oppure una mail inesistente all'interno del database.

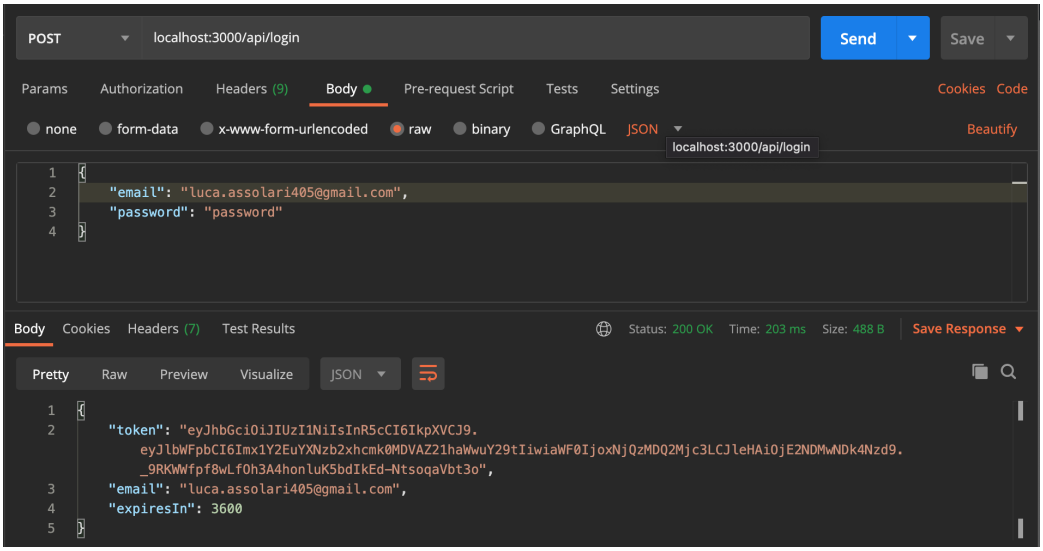


Figura 4.8: Test API di login: login corretto

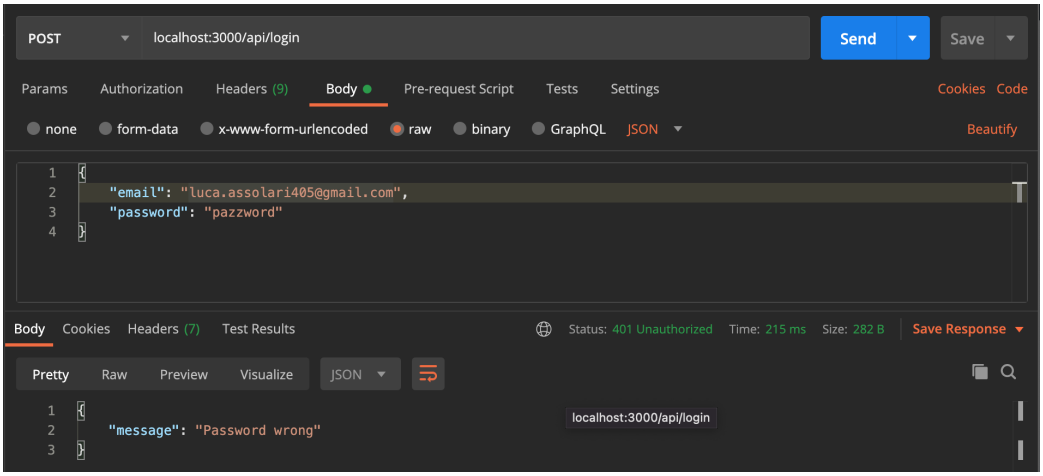


Figura 4.9: Test API di login: password errata

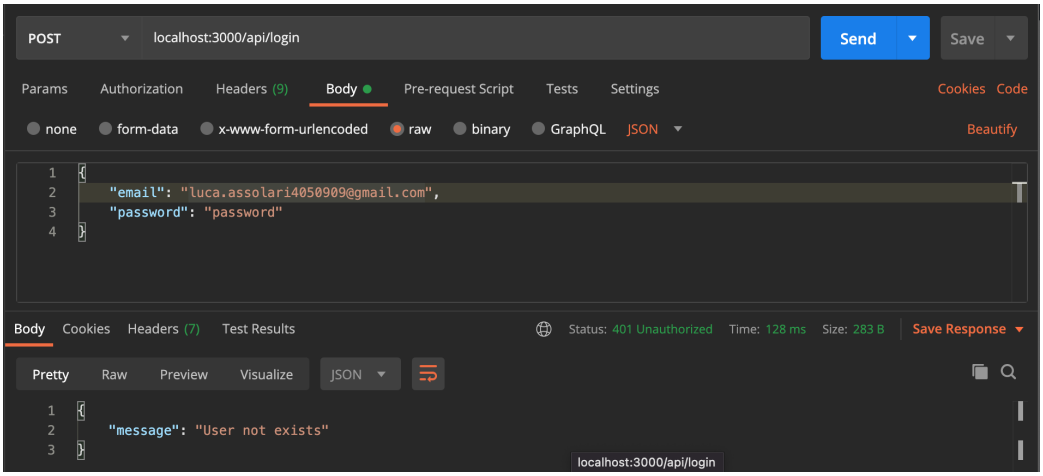


Figura 4.10: Test API di login: utente inesistente

Come si può notare, nel primo caso i dati sono stati inseriti correttamente, quindi la risposta da parte del server è positiva (200) e vengono ritornati, oltre alla mail dell'utente, il token per la sessione corrente e il tempo di durata della sessione, scaduto il quale verrà richiesto nuovamente di effettuare il login. Nel secondo caso, invece, la mail inserita è corrente, ma non lo è la password: per questo motivo viene ritornato un errore da parte del backend, con in allegato un messaggio che verrà poi mostrato a schermo. Nel terzo

e ultimo caso viene inserita una mail non presente nel database, ricevendo quindi in risposta nuovamente un codice di errore con relativo messaggio.

Parte III

Iterazione 2

Capitolo 5

Introduzione

In questa iterazione vengono implementate le funzioni operative che l'utente può svolgere dalla home page, a partire dalla visualizzazione della **home page** stessa. Da questo punto l'utente potrà accedere alla **visualizzazione delle attività quotidiane** (denominate 'eventi'), e alla **compilazione degli appelli giornalieri**. La struttura gerarchica della pagina è la seguente: il componente padre è la home page, che racchiude al suo interno la visualizzazione degli eventi; a sua volta, questo risulta il componente padre della compilazione degli appelli, che possono essere mostrati all'inizio della lista degli eventi.

Formalmente, i requisiti funzionali sviluppati in questa iterazione sono lo 04 e lo 05, entrambi di priorità media. Parallelamente a ciò, in questa iterazione vengono predisposti anche due moduli per la creazione manuale di eventi e schede bambino/ragazzo, dati che potrebbero essere creati e manipolati diversamente, in future espansioni del progetto.

5.1 Component diagram

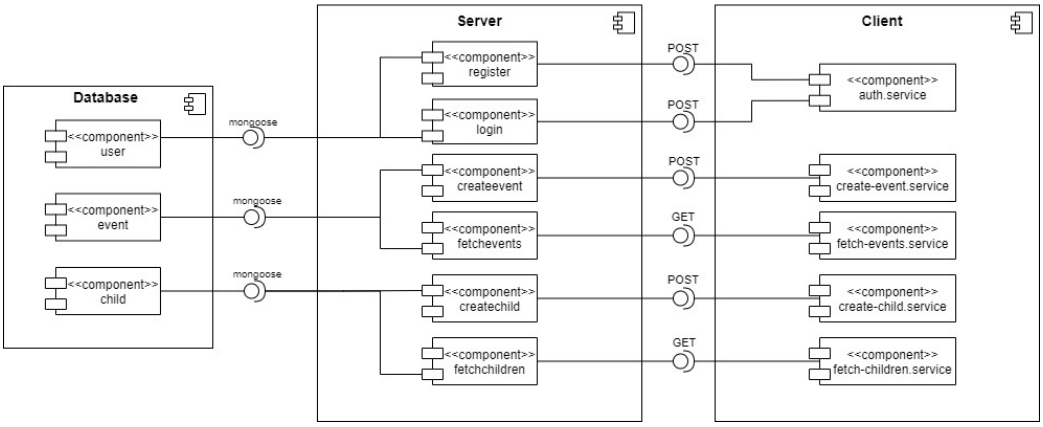


Figura 5.1: Component diagram

Negli aggiornamenti dei due diagrammi, *component* e *deployment* (mostrato nel paragrafo successivo), relativi all'iterazione 2, è possibile notare l'aggiunta dei componenti legati ai requisiti 04 e 05, ovvero la gestione degli eventi e degli appelli.

5.2 Deployment diagram

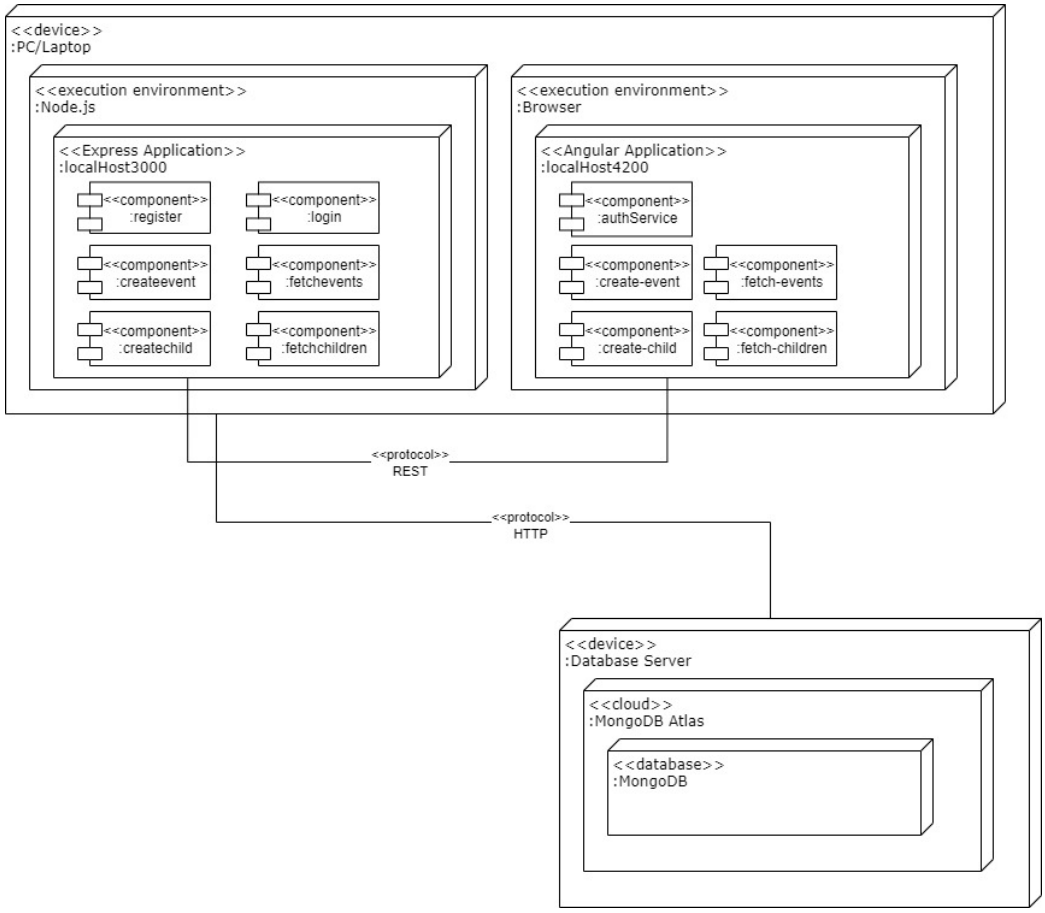


Figura 5.2: Deployment diagram

5.3 Sviluppo della gestione eventi - requisito 04

Creazione del modello di dato 'events' nel database

Prima di poter essere visualizzati, gli eventi devono poter essere creati dagli utenti (idealmente, i coordinatori o animatori con particolari privilegi). Inizialmente non è prevista una specializzazione dell'evento, ma solo le squadre coinvolte, identificate nel componente dalle mail dei coordinatori di riferimento, e la data&ora dello stesso. Gli eventi possono quindi rappresentare

attività/giochi/laboratori/gite e tutto ciò che ha un'estensione programmata nel tempo.

```
const mongoose = require('mongoose');
const uniqueValidator = require('mongoose-unique-validator');

const Schema = mongoose.Schema
const eventSchema = new Schema({
    event_name: {type: String, required: true},
    team1: {type: String, required: true},
    team2: {type: String, required: false},
    team3: {type: String, required: false},
    team4: {type: String, required: false},
    event_hour: {type: Number, required: true}
})

eventSchema.plugin(uniqueValidator);
module.exports = mongoose.model('event', eventSchema, 'events
    ')
```

Listing 5.1: Modello di dato 'event'

Creazione del componente CreateEvent e del relativo service

Viene predisposto un form HTML che richiede il nome dell'evento, la mail del coordinatore di almeno una squadra coinvolta, e l'orario di svolgimento.

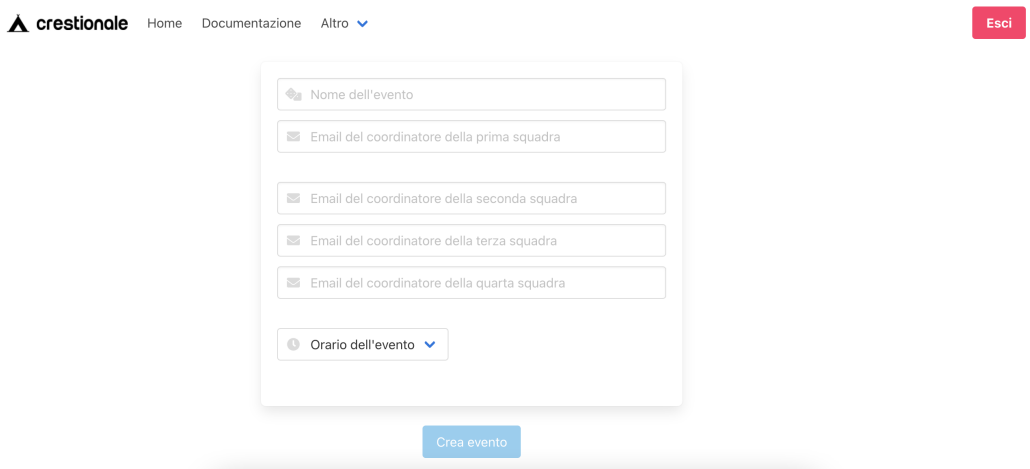


Figura 5.3: Form per la creazione di un evento

Lo script del componente non fa altro che mettere a disposizione il metodo *createEvent()*, definito nel rispettivo service, richiamato al click del pulsante di conferma, dopo aver inserito i dati nel modulo. Questo metodo effettua, nel service, una richiesta HTTP di post, che verrà definita nelle API, per inserire l'evento nel database.

```
createEvent(event) {
    this.http.post<any>(this._createEventUrl, event)
      .subscribe(response => {
        window.alert(response.message)
      })
  }
```

Listing 5.2: Metodo *createEvent()* del service

Aggiunta del routing per CreateEvent

Prima di definire le API per la creazione degli eventi, viene modificato l'app-routing.module per inserire il percorso 'createevent' che rimanda al componente CreateEventComponent. Inoltre, viene aggiunto nella navbar il pulsante che richiama il suddetto componente, per rimandare l'utente al form di creazione eventi.

API per la creazione di eventi

Viene aggiunta, nelle api.js, la funzione di post('createevent'). Nello specifico, questa si comporta come di seguito elencato:

- Recupera i dati dell'evento dalla richiesta;
- Controlla se nell'evento sono specificate più squadre o solo una; nel dettaglio:
  - Controlla se i campi relativi all'email dei coordinatori della squadra 2, 3 e 4 sono compilati;
  - Se non lo sono, setta a true i campi found2, found3 e found4, per 'bypassare' il controllo;
  - Se sono compilati, controlla che le email di quei coordinatori esistano nel database;
  - Se lo sono, setta a true i campi found2, found3 e found4, false altrimenti;

- Controlla la mail del coordinatore della squadra principale, se esiste nel database;
- Crea l'evento e segnala a tutti i subscribers che l'evento sia stato creato

```
router.post('/createevent', (req, res) => {

    let eventData = req.body
    let found2 = false
    let found3 = false
    let found4 = false

    if(!eventData.team2)
        found2 = true
    else {
        User.findOne({email: eventData.team2}).then(user => {
            if(!user)
                found2 = false
            else
                found2 = true
        })
    }

    if(!eventData.team3)
        found3 = true
    else {
        User.findOne({email: eventData.team3}).then(user => {
            if(!user)
                found3 = false
            else
                found3 = true
        })
    }

    if(!eventData.team4)
        found4 = true
    else {
        User.findOne({email: eventData.team4}).then(user => {
            if(!user)
                found4 = false
            else
                found4 = true
        })
    }
```

```
 User.findOne({email: eventData.team1}).then(user => {
     if(!user)
         res.json({
             message: 'Errore: non esiste nessun animatore
 con la prima mail specificata'
         })
     else {
         if(found2 && found3 && found4) {
             const event = new Event({
                 event_name: eventData.event_name,
                 team1: eventData.team1,
                 team2: eventData.team2,
                 team3: eventData.team3,
                 team4: eventData.team4,
                 event_hour: eventData.event_hour
             })

             event.save((error, registeredEvent) => {
                 if(error)
                     res.json({
                         message: 'Errore nella creazione
evento'
                     })
             else
                 res.json({
                     message: 'Evento creato',
                     result: registeredEvent
                 })
             })
         }
         else {
             if(!found2)
                 res.json({ message: 'Errore: non esiste
nessun animatore con la seconda mail specificata' })
             else if(!found3)
                 res.json({ message: 'Errore: non esiste
nessun animatore con la terza mail specificata' })
             else
                 res.json({ message: 'Errore: non esiste
nessun animatore con la quarta mail specificata' })
         }
     }
 })
```

```
})
```

Listing 5.3: API di POST per la creazione degli eventi nel DB

Home page e visualizzazione eventi

L'HTML della home page mostra il componente EventList, quando gli eventi vengono correttamente recuperati dal database a seconda dell'utente che risulta autenticato e loggato.

Lo script del componente home-page contiene l'informazione sulla mail dell'utente loggato, e predispone un vettore per contenere la lista degli eventi, oltre che un booleano per tenere traccia dell'avvenuto recupero della lista degli eventi.

```
ngOnInit(): void {
    this.userEmail = this._authService.getEmail()
    this.updateEvents()
  }

  updateEvents() {
    this._fetchEvents.fetchEvents(this.userEmail)
      .subscribe(res => {
        this.events = res.result
        this.eventsFetched = true
      })
  }
```

Listing 5.4: Snippet dello script della Home page

```
fetchEvents(email: string): any {
    return this.http.get(this._fetchEventsUrl, {params: {
   email: email}})
  }
}
```

Listing 5.5: Metodo fetchEvents() del service della Home page

All'avvio, il componente recupera la mail dell'utente loggato e aggiorna la lista degli eventi tramite il metodo updateEvents(), il quale sfrutta il servizio fetch-events che si appoggia alla API descritta qui di seguito, tramite una richiesta di post.

API per il recupero degli eventi

Il funzionamento è semplice: si estrapola dalla richiesta HTTP l'utente loggato, e si va a cercare tra gli eventi quelli che hanno come coordinatore di una delle quattro possibili squadre coinvolte l'utente loggato. Se l'utente non risulta coinvolto in nessun evento, viene mostrato un messaggio, altrimenti si crea una lista con il nome e l'orario dei singoli eventi, che viene poi restituita come risultato della richiesta.

```
router.get('/fetchevents', (req, res) => {

    let userEmail = req.query.email

    Event.find({$or:[{team1: userEmail}, {team2: userEmail},
   {team3: userEmail}, {team4: userEmail}]}).then(events => {
        if(!events)
            console.log('Oggi non hai nulla da fare, strano')
        else {
            let lista = []

            events.forEach(element => {

                let item = {
                    'event_name': element.event_name,
                    'event_hour': element.event_hour
                }

                lista.push(item)
            })

            res.status(200).json({result: lista})
        }
    })
})
```

Listing 5.6: API *fetchevents* per il recupero degli eventi

Event List Component

Il componente viene mostrato nella home page, a seconda dell'utente loggato. La lista degli eventi viene suddivisa in due sezioni, per migliorarne la leggibilità: una prima parte, denominata 'Tra poco', mostra il prossimo evento a partire dall'orario attuale, mentre la sezione 'In seguito' riporta i restanti eventi successivi.

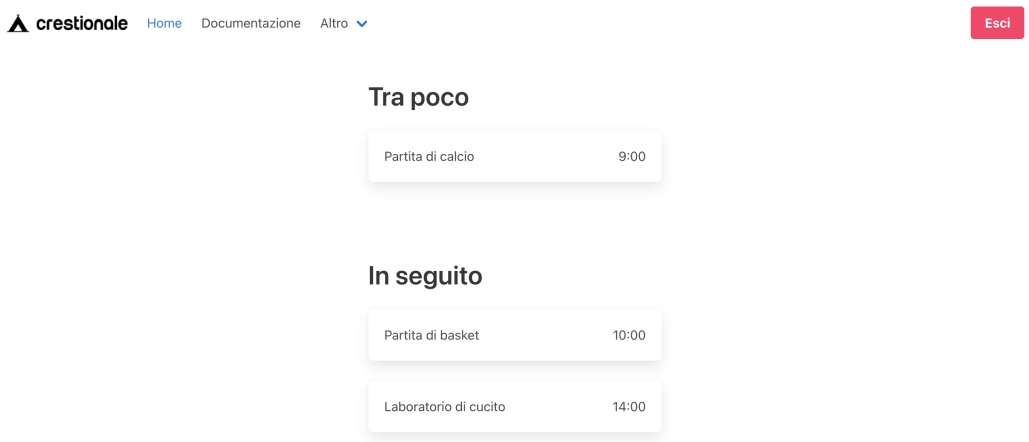


Figura 5.4: Visualizzazione della lista di eventi

Per ottenere questa suddivisione, lo script si comporta nel seguente modo:

- All'avvio del componente, questo riordina la lista degli eventi in base all'orario, tramite un algoritmo **Insertion Sort**, illustrato nel dettaglio nella sezione successiva;
- Una volta riordinati, due funzioni si occupano di recuperare il prossimo evento in ordine temporale e la lista dei successivi eventi; nel dettaglio:
  - getNextEvent() prende il primo evento della lista degli eventi riordinati;
  - getFutureEvents() prende tutti gli eventi successivo il primo;
- Se la lista degli eventi riordinati contiene un solo elemento, viene settato a true il booleano noFutureEvents, in modo tale da non mostrare la sezione 'In seguito'; se invece la lista è completamente vuota, anche il booleano noNextEvent viene settato a true, di modo da mostrare all'utente un messaggio di avviso sull'assenza di attività da svolgere.

```
getNextEvent() {

    if(this.orderedEvents.length > 0)
      this.nextEvent = this.orderedEvents[0]
    else
      this.noNextEvent = true
```

```
}

getFutureEvents() {

  if(this.orderedEvents.length > 1) {
    for(let i = 1; i < this.orderedEvents.length; i++) {
      this.futureEvents.push(this.orderedEvents[i])
    }
  }
  else
    this.noFutureEvents = true
}
```

Listing 5.7: Metodi per la suddivisione delle liste eventi

In aggiunta a questa funzione di visualizzazione degli eventi, l'Event List Component propone anche, mostrandolo all'interno della home page, la compilazione dell'appello, che l'utente può svolgere differenziato a seconda del momento della giornata in cui si trova. Maggiori dettagli verranno forniti nello sviluppo del paragrafo incentrato sul requisito 05: compilazione appello.

5.3.1 Algoritmo per l'ordinamento degli eventi

Gli eventi mostrati nella home page, come detto, sono recuperati dal server tramite una chiamata API. Tuttavia, vengono ritornati nell'ordine in cui sono stati inseriti, che non sempre corrisponde a quello temporale. Per questo motivo, è necessario effettuare un ordinamento di questi eventi prima di mostrarli all'utente finale, in modo tale da rendere la visualizzazione a schermo molto più intuitiva, oltre a permettere la corretta distinzione tra *nextEvent* (quindi l'evento più prossimo in termini di orario) e *futureEvents*.
Per effettuare l'ordinamento, si è scelto di utilizzare il noto algoritmo **Insertion Sort**, in quanto molto efficiente su un input di dimensioni ridotte, come in questo caso; infatti, gli eventi ritornati dall'API sono intesi come gli eventi di una singola giornata, che logicamente saranno un numero molto piccolo.

Nel dettaglio, l'implementazione è la seguente.

```
for(let j = 1; j <= (upcomingEvents.length - 1); j++) {
    let key = upcomingEvents[j]
    let i = j - 1
    while(i >= 0 && upcomingEvents[i].event_hour >
      key.event_hour) {
```

```
        upcomingEvents[i + 1] = upcomingEvents[i]
        i = i - 1
    }
    upcomingEvents[i + 1] = key
}
```

Listing 5.8: Implementazione dell'algoritmo *insertion sort*

Come si può notare dal codice riportato, si tratta di un algoritmo *in loco*, ovvero ordina l'array senza la necessità di doverne creare una copia, risparmiando spazio in memoria.
Dal punto di vista dell'analisi prestazionale, la *complessità temporale* di questo algoritmo dipende dalla particolare distribuzione dei dati in input.
Infatti, se l'array è già ordinato in senso crescente, ogni nuovo elemento *upcomingEvents[j]* che si considera nel ciclo *for* avrà un orario sempre maggiore del precedente, quindi il ciclo *while* si interrompe immediatamente, riducendo così l'algoritmo ad una semplice scansione dell'array: la complessità è quindi $\Theta(n)$.
Viceversa, il caso peggiore si verifica quando l'array è ordinato in senso decrescente, quindi con l'evento più futuro come primo elemento. In questo caso, il ciclo *while* richiede *j - 1* confronti, rendendo così la complessità dell'algoritmo pari a $\Theta(n^2)$.

L'analisi prestazionale aiuta a capire il motivo della scelta di questo specifico algoritmo: gli eventi nel database vengono inseriti, idealmente, da un attore umano, il quale andrà intuitivamente ad inserirli in ordine. Gli eventi che non si trovano nell'ordine corretto saranno solo quelli inseriti successivamente alla decisione della schedulazione per una giornata, come nel caso di cambi imprevisti del programma o novità dell'ultimo minuto. Questa assunzione permette di supporre che, nella maggior parte dei casi, l'array di eventi venga ritornato dal backend già ordinato o, al più, con un numero ridotto di elementi fuori posto: l'Insertion Sort è quindi l'algoritmo ideale per trattare degli input di questo tipo.

5.4 Sviluppo della gestione appelli - requisito 05

Creazione del modello di dato 'child' nel database

Come nel caso degli eventi, è necessario predisporre un modulo di creazione delle schede per ogni bambino/ragazzo. Queste prevedono le generalità base, un collegamento al coordinatore della squadra tramite la sua mail, ed alcune variabili per distinguere le diverse possibilità di presenza (mattino/pomeriggio e mensa).

```
const mongoose = require('mongoose');
const uniqueValidator = require('mongoose-unique-validator');

const Schema = mongoose.Schema
const userSchema = new Schema({
    name: {type: String, required: true},
    surname: {type: String, required: true},
    coordinatedBy: {type: String, required: true},
    presenza: {type: Number, required: true},
    presenza_in_mensa: {type: Boolean, required: true}
})

userSchema.plugin(uniqueValidator);
module.exports = mongoose.model('child', userSchema, '
   children')
```

Listing 5.9: Modello di dato 'child'

Creazione del componente CreateChild e del relativo service

In maniera molto simile a quanto visto per il componente CreateEvent, viene predisposto un form HTML che richiede i dati appena descritti nel modello 'child'.

Figura 5.5: Form per la creazione di una scheda bambino/ragazzo

Lo script del componente non fa altro che mettere a disposizione il metodo *createChild()*, definito nel rispettivo service, richiamato al click del pulsante di conferma, dopo aver inserito i dati nel modulo. Questo metodo effettua, nel service, una richiesta HTTP di post, che verrà definita nelle API, per inserire la scheda del bambino/ragazzo nel database.

```
createChild(child) {
    this.http.post<any>(this._createChildUrl, child)
      .subscribe(response => {
        window.alert(response.message)
      })
  }
```

Listing 5.10: Metodo *createChild()* del service

Aggiunta del routing per CreateChild

Prima di definire le API per la creazione delle schede di bambini/ragazzi, viene modificato l'app-routing.module per inserire il percorso 'createchild' che rimanda al componente CreateChildComponent. Inoltre, viene aggiunto nella navbar il pulsante che richiama il suddetto componente, per rimandare l'utente al form di registrazione per l'inserimento di una nuova scheda bambino/ragazzo.

API per la creazione di schede bambino/ragazzo

Viene aggiunta, nelle api.js, la funzione di post('createchild'). Nello specifico, questa si comporta come di seguito elencato:

- Recupera le informazioni sul bambino/ragazzo dal corpo della richiesta;
- Controlla che nel database esista un utente che abbia la stessa mail specificata nella scheda bambino/ragazzo, indicante il coordinatore della squadra;
- Se c'è, viene creata una nuova scheda con le informazioni estrapolate dalla richiesta;
- Se non c'è, viene visualizzato un messaggio di errore;
- Una volta creata la scheda, ciò viene segnalato a tutti i subscribers.

```
router.post('/createchild', (req, res) => {

    let childData = req.body

    User.findOne({email: childData.coordinatedBy}).then(user
   => {
        if(!user)
            res.json({
                message: 'Errore: non esiste un animatore con
    questa mail',
            })
        else {
            const child = new Child({
                name: childData.name,
                surname: childData.surname,
                coordinatedBy: childData.coordinatedBy,
                presenza: childData.presenza,
                presenza_in_mensa: childData.
   presenza_in_mensa
            })

            child.save((error, registeredChild) => {
                if(error)
                    res.json({
                        message: 'Errore nella creazione
   della scheda',
```

```
                    })
                else
                    res.json({
                        message: 'Scheda bambino creata',
                        result: registeredChild
                    })
            })
        }
    })

})
```

Listing 5.11: API di POST per la creazione delle schede bambino/ragazzo

Creazione del componente RollCall per l'appello

Innanzitutto, la visualizzazione dell'appello avviene all'interno del componente EventList, dove vengono predisposte alcune variabili che contengono l'informazione sullo stato dell'appello. Queste variabili sono 3, *morningDone*, *lunchDone* e *afternoonDone*, e valgono 0 nel momento in cui il relativo appello non è stato ancora svolto, 1 quando deve essere svolto (a seconda quindi dell'orario del giorno) e 2 quando è stato correttamente compilato.
All'avvio del componente EventList, viene eseguito il metodo *getRollCallStatus()* che va a cercare nel localStorage se l'appello è già stato fatto (e quindi il suo stato attuale), oppure setta le tre variabili appena descritte a 0.

```
  morningDone : number
  lunchDone : number
  afternoonDone : number

  rollCallToggle : boolean = false
  rollCallType : number

  constructor() { }

  ngOnInit(): void {
    this.getRollCallStatus()
    this.orderedEvents = this.orderEvents()
    this.getNextEvent()
    this.getFutureEvents()
  }

  getRollCallStatus() {
```

```
    this.morningDone = parseInt(localStorage.getItem('
  morningDone')) || 0
    this.lunchDone = parseInt(localStorage.getItem('lunchDone
  ')) || 0
    this.afternoonDone = parseInt(localStorage.getItem('
  afternoonDone')) || 0
}
```

Listing 5.12: Snippet da EventList per la gestione degli appelli

Un altro metodo, *checkRollCall()*, imposta, in base all'orario attuale, la variabile dell'appello a 1, cioè lo imposta come 'da svolgere', nel caso in cui all'orario limite della fase della giornata (quindi 9 per il mattino, 12 per il pranzo e 14 per il pomeriggio) esso non sia stato ancora compilato.

Quando le suddette variabili sono impostate a 1, nell'HTML vengono mostrati i rispettivi pulsanti che permettono, alla pressione, di aprire e compilare lo specifico appello, come mostrato nella seguente immagine.

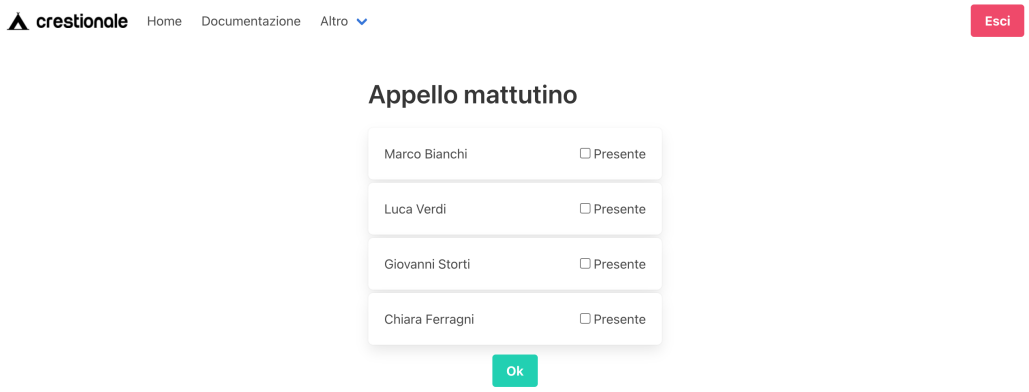


Figura 5.6: Schermata per la compilazione dell'appello

Nel dettaglio: quando uno di questi pulsanti viene premuto, si esegue il metodo *toggleRollCall()*, che fa appunto il toggle, cioè inverte lo stato, della variabile booleana *rollCallToggle*, indicante l'intenzione dell'utente di compilare un appello, e setta la variabile *rollCallType* al valore relativo all'appello desiderato (0 per quello mattutino, 1 per quello del pranzo e 2 per l'appello pomeridiano). Fatto ciò, la visualizzazione cambia, vengono nascosti i pulsanti e viene aperto il componente RollCall, al quale vengono passati la mail dell'utente e il tipo di appello che ha specificato di voler compilare, in modo tale da recuperare i dati corretti.

```
toggleRollCall(rollCallType: number) {
    this.rollCallToggle = !this.rollCallToggle
    this.rollCallToggleChange.emit(this.rollCallToggle)

    switch(rollCallType) {
      case(0): {
        this.morningDoneChange.emit(2)
        localStorage.setItem('morningDone', '2')
        break
      }
      case(1): {
        this.lunchDoneChange.emit(2)
        localStorage.setItem('lunchDone', '2')
        break
      }
      case(2): {
        this.afternoonDoneChange.emit(2)
        localStorage.setItem('afternoonDone', '2')
        break
      }
    }
  }
```

Listing 5.13: Metodo *toggleRollCall()* per l'apertura degli appelli

È importante specificare che il binding dei dati relativi alle variabili rollCallToggle, morningDone, lunchDone e afternoonDone è di tipo two-way, ovvero le modifiche che vengono fatte dal componente figlio (quindi durante la compilazione degli appelli) sono rilevate anche dal componente padre. Nel componente RollCall vero e proprio, si ha un HTML che mostra l'elenco dei bambini/ragazzi affiancati da una checkbox che è possibile selezionare per indicare la presenza del rispettivo iscritto. Lo script innanzitutto preleva, tramite il metodo fetchChildren() dell'omonimo service, la lista dei bambini/ragazzi presenti nel database associati alla mail dell'utente che ha eseguito l'accesso. In seguito, il metodo makeLists() estrapola dalla lista principale, che include tutti i bambini/ragazzi della squadra, tre sotto-liste, una per i presenti al mattino, una per i presenti al pomeriggio e una per gli iscritti a mensa.

```
  makeLists(childrenList) {

    childrenList.forEach(element => {

      switch(element.presenza) {
```

```
        case 0: {
          this.morning_list.push(element)
          break
        }
        case 1: {
          this.afternoon_list.push(element)
          break
        }
        case 2: {
          this.morning_list.push(element)
          this.afternoon_list.push(element)
          break
        }
      }

      if(element.presenza_in_mensa)
        this.lunch_list.push(element)
    })
  }

  showList() {
    switch(this.rollCallType) {
      case(0): {
        this.roll_call_list = this.morning_list
        this.roll_call_title ='Appello mattutino'
        break
      }
      case(1): {
        this.roll_call_list = this.lunch_list
        this.roll_call_title ='Appello mensa'
        break
      }
      case(2): {
        this.roll_call_list = this.afternoon_list
        this.roll_call_title ='Appello pomeridiano'
        break
      }
    }
  }
```

Listing 5.14: Metodi per la creazione e la visualizzazione degli appelli

Il metodo showList() preleva la lista da mostrare a seconda della variabile rollCallType. Questa lista è quella che poi viene effettivamente visualizzata nell'HTML del componente. Un ultimo metodo, toggleRollCall(), invocato alla pressione del pulsante 'ok' posto alla fine della lista dell'appello visualiz-

zato, si occupa di invertire nuovamente la variabile rollCallToggle, che viene trasmessa anche al componente padre, ovvero l'EventList. Così facendo, la visualizzazione cambia di nuovo, viene chiusa la sezione che mostra l'elenco dei bambini/ragazzi e si torna alla visualizzazione degli eventi. In parallelo a ciò, viene settata a 2 la variabile relativa all'appello appena compilato, indicando che esso è stato effettivamente svolto. Tutte queste modifiche vengono segnalate al componente padre, cioè all'EventList.

API per il FetchChildren

Il funzionamento è semplice: si estrapola dalla richiesta HTTP l'utente loggato, e si va a cercare nel database tutte le schede bambino/ragazzo che hanno come mail del coordinatore quella dell'utente loggato. Se l'utente non risulta coordinatore di nessun bambino/ragazzo, viene mostrato un messaggio, altrimenti si crea una lista con tutti i bambini/ragazzi, che viene poi restituita come risultato della richiesta.

```
router.get('/fetchchildren', (req, res) => {

    let userEmail = req.query.email

    Child.find({coordinatedBy: userEmail}).then(children => {
        if(!children) {
            res.status(401).json({
                message: 'Error'
            })
        }
        else {
            let lista = []

            children.forEach(element => {
                let item = {
                    'name': element.name,
                    'surname': element.surname,
                    'presenza': element.presenza,
                    'presenza_in_mensa': element
                        .presenza_in_mensa
                }
                lista.push(item)
            });

            res.status(200).json({result: lista})
        }
```

```
    })
})
```

Listing 5.15: API di GET per il recupero della lista delle schede bambino/ragazzo associate ad ogni animatore

Confronto con il caso d'uso ipotizzato

I dati ricavati tramite l'appello non vengono, per ora, utilizzati in nessun altro modulo, come invece ci si auspicava durante l'*envisioning* dell'iterazione 0.

Inoltre, l'appello della mensa risulta svincolato da quello mattutino, a differenza di quanto ipotizzato nei casi d'uso.

5.5 Testing

5.5.1 Testing della lista di eventi

Il componente che si è scelto di testare in maniera più approfondita è quello relativo alla lista degli eventi, in quanto racchiude la funzionalità principale dell'applicazione una volta che l'utente ha effettuato l'autenticazione. Al suo interno, infatti, sono mostrati gli eventi della giornata dell'utente, distinguendo tra il *prossimo evento* e gli *eventi futuri*, quindi quelli appena successivi al prossimo. Se l'evento è solo uno, tuttavia, gli eventi futuri non vengono mostrati, in quanto non presenti.
Per effettuare questi test, sono stati creati due array di eventi mock, il primo che ne contiene solo uno mentre il secondo che ne contiene due.

```
const singleEvent = [
    {
        event_name: 'Partita di calcio',
        team1: 'luca.assolari405@gmail.com',
        event_hour: 20
    }
]

const multipleEvents = [
    {
        event_name: 'Partita di calcio',
        team1: 'luca.assolari405@gmail.com',
```

```
        event_hour: 20
    },
    {
        event_name: 'Laboratorio di scacchi',
        team1: 'luca.assolari405@gmail.com',
        event_hour: 21
    }
]
```

Listing 5.16: Creazione di eventi mock

L'idea dei test è quindi quella di andare a verificare che, in caso di un solo evento, sia presente esclusivamente la sezione dedicata al *prossimo evento*, mentre nel caso in cui gli eventi siano molteplici sia visibile anche la sezione dedicata agli *eventi futuri*.

```
it('should have next event if there is one',
    () => {
    component.inputEvents = singleEvent
    component.ngOnInit()
    fixture.detectChanges()
    expect(component.nextEvent).toBeDefined()
})

it('should not have future events if there is only one',
    () => {
    component.inputEvents = singleEvent
    component.ngOnInit()
    fixture.detectChanges()
    expect(component.futureEvents.length).toBe(0)
})

it('should have future events if there are more than one',
    () => {
    component.inputEvents = multipleEvents
    component.ngOnInit()
    fixture.detectChanges()
    expect(component.futureEvents.length).toBeGreaterThan(0)
})
```

Listing 5.17: Testing della lista di eventi

Come si evince dai test sopra riportati, questi hanno tutti una struttura molto simile. Come prima cosa, si va ad assegnare alla variabile *inputEvents* del componente l'array di eventi (che normalmente verrebbe passato come parametro dal componente padre) e in seguito si richiama il metodo *ngOnInit*

Figura 5.7: Risultato dei test della seconda iterazione

che, in base ad esso, va ad inizializzare l'oggetto *nextEvent* e l'array *futureEvents*, sui quali vengono poi effettuati i controlli per determinare l'esito dei test.

5.5.2 Risultato dei test

L'immagine sottostante mostra l'esecuzione della suit di test, nella quale sono presenti, oltre ai test specificati nell'iterazione precedenti, anche quelli appena aggiunti riferiti alla lista di eventi.

5.5.3 Testing API

Per quanto riguarda il testing lato backend, sempre facendo uso di Postman è possibile andare a verificare il corretto funzionamento dell'API *fetchevents* che, tramite una richiesta GET e la mail dell'utente passata come *query parameter*, ritorna un array contenente la lista degli eventi associati a quell'utente.

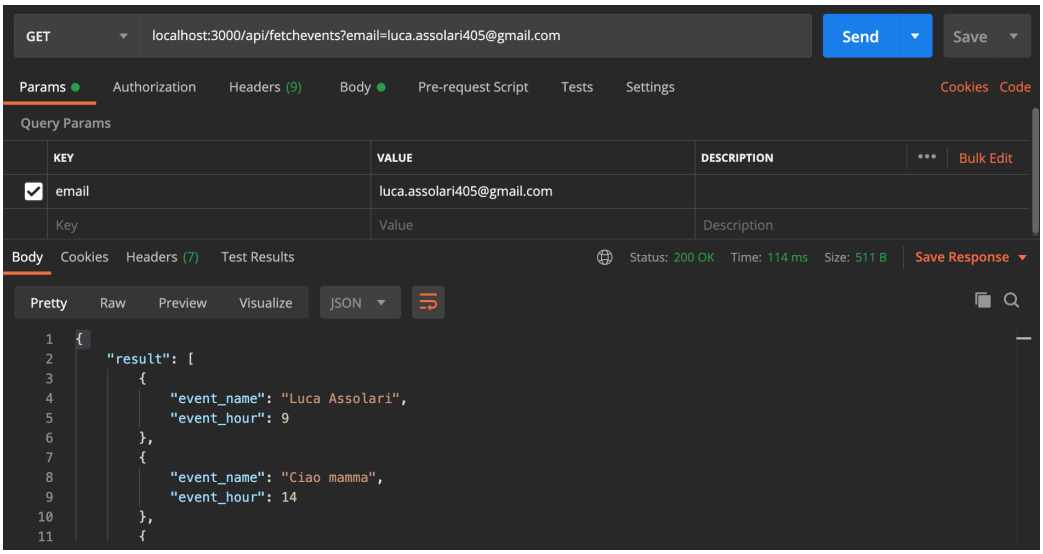


Figura 5.8: Test API per il recupero della lista eventi

Parte IV

Guida all'installazione

Capitolo 6

Installazione e avvio

6.1 Installazione di Node.js

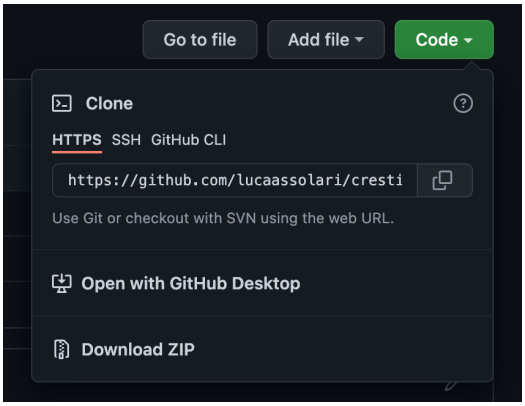
Come prima cosa è necessario installare **Node.js**, in modo tale da poter utilizzare **npm** per la gestione di tutti i pacchetti. Per farlo, visitare la seguente pagina e scaricare la versione di Node.js compatibile con il proprio sistema operativo: download Node.js
Completare l'installazione seguendo le istruzioni mostrate dall'installer.

6.2 Clone della repository

A questo punto, è possibile recuperare la repository contente il codice da questa pagina: GitHub
Una volta raggiunta la pagina, cliccare sul pulsante in verde *Code*, e ci si troverà davanti alla seguente schermata: Se si utilizza un software per il versioning del codice, tipo **git**, è possibile clonare la repository utilizzando, da terminale, il comando **git clone**, seguito dall'URL mostrato. Altrimenti, scaricare lo zip premendo sul pulsante *Download ZIP*, e scompattarlo in una directory a scelta.

6.3 Installazione delle dipendenze

Aprire un terminale nella cartella *code/crestionale* situata all'interno della repository appena scaricata e digitare il comando `npm install`, in modo tale



da installare le dipendenze di runtime e quelle di sviluppo.
Successivamente, sempre da terminale, con il comando `npm install -g @angular/cli` è possibile installare la CLI di Angular per poter compilare il codice frontend.

Attenzione se si utilizza Windows

Poiché l'applicazione è stata sviluppata su macOS, si verifica un problema con una particolare libreria, ovvero *bcrypt*, nel passaggio da un sistema operativo a un'altro. Per risolvere, entrare nella cartella *code/crestionale/server/node_modules*, eliminare la directory di bcrypt ed eseguire una nuova installazione con il comando `npm install bcrypt -g`.

6.4 Avvio dell'applicazione

Per avviare il client, entrare nella cartella *code/crestionale* ed eseguire il comando `ng serve --open`. Per quanto riguarda il server, eseguire il comando `node server` all'interno dell cartella *code/crestionale/server*.
Si è ora pronti per utilizzare l'applicazione!