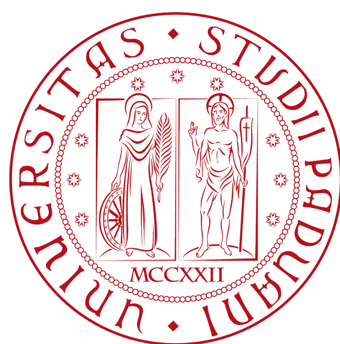
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Progettazione e sviluppo di un'applicazione web per la gestione di profili anagrafici e competenziali per l'organizzazione di team di lavoro

Tesi di laurea triennale

Relatore

Prof.Massimiliano de Leoni

Laureando

Nicola Pavin

Anno Accademico 2021-2022

Nicola Pavin: *Progettazione e sviluppo di un'applicazione web per la gestione di profili anagrafici e competenziali per l'organizzazione di team di lavoro,* Tesi di laurea triennale, © Dicembre 2022.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Nicola Pavin presso l'azienda Omicron Consulting S.r.l. supervisionato dal tutor aziendale, Luca Rossato, e dal relatore, Prof. Massimiliano de Leoni.
L'obbiettivo dello stage era la progettazione e realizzazione del back-end di un'applicazione web che permetta di creare e visualizzare profili customizzabili dei dipendenti dell'azienda, poi organizzabili in team di lavoro. In primo luogo era richiesto dunque lo studio e l'individuazione delle tecnologie da utilizzare, ed in secondo luogo la progettazione e la realizzazione dell'applicazione.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Massimiliano de Leoni, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero poi ringraziare le mie figure di riferimento aziendali, Luca Rossato e Luca Scatigno, i quali mi hanno aiutato e sostenuto durante la mia esperienza di tirocinio.

Padova, Dicembre 2022 Nicola Pavin

Indice

1 Introduzione **1**
1.1 Presentazione dell'azienda 1
1.2 L'obbiettivo dello stage 2
1.3 Organizzazione del testo 3

2 Pianificazione dello stage **5**
2.1 Analisi preventiva dei rischi 5
2.2 Pianificazione iniziale 6
2.3 Modalità di lavoro 6
2.4 Obbiettivi e requisiti 6
2.4.1 Notazione 6
2.4.2 Requisiti accordati 7

3 Descrizione dello tecnologie utilizzate **9**
3.1 Introduzione al progetto 9
3.2 Strumenti e tecnologie 9
3.2.1 Repository 9
3.2.2 Database 10
3.2.3 Back-end 10
3.2.4 Typescript 10
3.2.5 Docker 11
3.2.6 LDAP 12
3.2.7 Ready Player Me 13
3.2.8 Visual Studio Code 13
3.2.9 Insomnia 13

4 Analisi dei requisiti **15**
4.1 Casi d'uso 15
4.1.1 Attori 15
4.1.2 Diagrammi dei casi d'uso 17
4.2 Tracciamento dei requisiti 21

5 Progettazione e codifica **23**
5.1 Progettazione 23
5.1.1 Il modello 23
5.1.2 Il database 27
5.1.3 Divisione delle responsabilità 27
5.1.4 Profili 28
5.1.5 Endpoints 28

5.2 Codifica . . . 30
5.2.1 Prodotto ottenuto . . . 33

6 Verifica e validazione **37**
6.1 Test di unità . . . 37
6.1.1 Tecnologie utilizzate . . . 37
6.1.2 Test sul modello . . . 37
6.1.3 Test dei middleware . . . 38
6.2 Validazione . . . 39

7 Conclusioni **41**
7.1 Raggiungimento degli obiettivi . . . 41
7.2 Conoscenze acquisite . . . 41
7.3 Valutazione personale . . . 41

Glossario **43**

Bibliografia **45**

Elenco delle figure

2.1 Pianificazione iniziale del lavoro 6

3.1 Esempio di gitflow, **https://nvie.com/posts/a-successful-git-branching-model/** 11
3.2 Parte del file *docker-compose.yml* prodotto, che fornisce istruzioni sulla costruzione del *container* per il back-end e che permette di definire delle variabili d'ambiente 12
3.3 Parte del file *docker-compose.yml* prodotto, che fornisce istruzioni sulla costruzione del *container* per il database, e che definisce un volume chiamato mongodb 12

4.1 Attori individuati 16
4.2 Scenario principale 17
4.3 Scenario principale 2 17
4.4 UC03 - Inserimento e verifica dati 19

5.1 Diagramma delle classi pertinenti al modello 24
5.2 funzione che tenta la connessione al database aziendale 31
5.3 Parte del codice sviluppato che verifica l'autorizzazione tramite JWT . 32
5.4 Funzione che controlla che il tipo dell'utente si *Admin* e, in caso contrario, lancia uno *StatusError* 33
5.5 Esempio di richiesta a **POST /signup** 34
5.6 Esempio di richiesta a **POST /sections** 35
5.7 Esempio di richiesta a **PUT /teams/:id/sid** 36

6.1 Definizione ed implementazione dei test per i metodi *getById* e *queryById* della classe **User**, che corrispondono ai test **TU7**, **TU8**, **TU9** e **TU10** 39

Capitolo 1

Introduzione

In questo capitolo si presenta l'azieda promotrice del progetto di stage, l'obbiettivo del progetto, e viene brevemente spiegata l'organizzazione del testo.

1.1 Presentazione dell'azienda

Omicron Consulting è presente sul mercato ICT fin dal 1980. E' un'azienda specializzata nello sviluppo di software gestionale e di revisione dei processi aziendali, con particolare riferimento ai settori Manufacturing – Automotive – Aerospace - Logistics , ecc., su cui abbiamo effettuato importanti implementazioni in area ICT. Nel settore Manufacturing ci siamo specializzati nello sviluppo di progetti complessi di trasformazione ERP (acquisendo la certificazione VAR di SAP), con particolare focus sulle tematiche supply chain di pianificazione ed ottimizzazione della produzione sia in ambienti di tipo "job shop" che in ambiente a flusso/ripetitivo; inoltre, in relazione alla specificità del "mondo Manufacturing", abbiamo stretto alleanze strategiche con realtà ICT nazionali ed internazionali allo scopo di poter formulare proposte e fornire soluzioni adeguate a tale specifico settore. I servizi offerti ai Ns Clienti includono, oltre alle attività di consulenza tecnica e di processo, la gestione AMS di sistemi ERP, la realizzazione di progetti "turn-key" sia in ambito di sistemi ERP di Business Intelligence (Business Objects, SAP BW, QlikView, ecc), che di sviluppi su sistemi custom con tecnologie Java, .Net, Sharepoint, ecc. In riferimento al mondo della BI, abbiamo acquisito un'esperienza ventennale nello sviluppo di DW ed ETL a cui affianchiamo i migliori specialisti di BI, in particolare sulle tecnologie Business Objects e Qlikview, creando un team che ha la capacità e la competenza per affrontare i progetti più ambiziosi. A completare la nostra offerta, va sottolineato che abbiamo competenze di altissimo livello negli ambiti BANKING, FINANCE, INSURANCE, lavorando con tutti i più importanti istituti bancari ed assicurativi italiani. Ciò rende la nostra azienda il partner ideale per tutte quelle realtà che oggigiorno guardano al futuro con la consapevolezza che il loro successo è strettamente collegato alla scelta del loro partner nell'area ICT. Oltre all'offerta dedicata ai clienti, Omicron punta molto sulla formazione delle proprie risorse, investendo su alcuni progetti interni di ricerca e sviluppo che permettono sia di portare avanti l'attività formativa dei tecnici, sia di aumentare le competenze su nuove tecnologie e linguaggi di programmazione attuali. Proprio in questo ambito si colloca il progetto che lo stagista dovrà seguire, di seguito denominato "Logistica Avanzata". Tale progetto si occupa di migliorare e personalizzare l'accoglienza dei visitatori delle

nostre sedi aziendali, offrendo servizi di riconoscimento individuale e percorsi guidati verso la persona con la quale si ha un appuntamento.

1.2 L'obbiettivo dello stage

L'obiettivo dello stage è la produzione e lo sviluppo di un'applicazione web che permetta di visualizzare la scheda anagrafica di uno o più dipendenti, in modo da offrire una visione completa sulle competenze e sui relativi livelli di preparazione, così come le tecnologie conosciute, soft skills e peculiarità principali della persona. L'applicativo deve prevedere la possibilità di costruire team composti da più profili selezionati dall'utente amministratore, in modo da poter presentare squadre di lavoro complete delle informazioni relative ai componenti del team. La peculiarità principale del progetto è legata soprattutto a due elementi: la presentazione dei dati e l'organizzazione degli stessi. I dati infatti verranno presentati in forma accattivante cercando di applicare un processo di gamification (si pensi ai videogiochi che spesso presentano le schede dei personaggi con le relative caratteristiche) alla pagina che viene presentata all'utilizzatore. L'idea è infatti quella di mostrare un avatar tridimensionale della persona interessata e di costruire in modo dinamico delle schede tematiche contenenti i dati delle competenze professionali, come ad esempio: linguaggi e tecnologie conosciute e relativi livelli di esperienza, soft skills, lingue parlate, certificazioni e altri attributi che verranno di volta in volta caricati dinamicamente a seconda del ruolo e del profilo selezionato. L'altro punto di attenzione dell'applicativo è infatti legato all'eterogeneità dei dati trattati, proprio per non doversi vincolare ad uno schema prefissato, ma rendendo possibile l'inserimento e la fruizione di informazioni suddivise in schede "dinamiche", per cui una persona potrebbe avere un profilo con un formato differente rispetto ad altri. Particolare attenzione, dunque, verrà riposta sulla creazione dell'avatar, sulla definizione di una base di dati non relazionale e sulla UI/UX dell'applicativo. Si prevedono tre profili utilizzatore, all'interno della stessa applicazione, accedibili in base a ruoli predefiniti. I tre profili permettono principalmente le seguenti funzionalità:

* Profilo base: accessibile solo da utenti dell'azienda abilitati, grazie al quale sarà possibile visualizzare e gestire solamente la propria scheda anagrafica. Nello specifico sarà possibile:
    - Creare una nuova scheda anagrafica, in caso di primo accesso, e compilarne le relative sezioni.
    - Modificare una scheda anagrafica, se già presente.
    - Creare un nuovo avatar per poterlo associare all'anagrafica.
    - Creare nuove sezioni, non presenti nel template standard.
* Profilo amministratore: accessibile solo da utenti dell'azienda abilitati, grazie al quale sarà possibile visualizzare e gestire le schede anagrafiche di tutti i profili disponibili. Nello specifico sarà possibile:
    - Creare una nuova scheda anagrafica e compilarne le relative sezioni.
    - Modificare una scheda anagrafica, se già presente.
    - Creare nuove sezioni, non presenti nel template standard
    - Creare un nuovo avatar per poterlo associare all'anagrafica.
    - Ricercare e filtrare le anagrafiche presenti nel sistema.

- Creare nuovi team tramite la selezione di più anagrafiche.
- Gestire (modifica/eliminazione) team esistenti.

* Profilo visualizzatore: accessibile solo da utenti dell'azienda abilitati, grazie al quale sarà possibile visualizzare le schede anagrafiche di tutti i profili disponibili e organizzarle in team dedicati. Nello specifico sarà possibile:
  - Ricercare e filtrare le anagrafiche presenti nel sistema.
  - Creare nuovi team tramite la selezione di più anagrafiche.
  - Gestire (modifica/eliminazione) team esistenti, solo se creati dall'utente in questione.

1.3 Organizzazione del testo

Il secondo capitolo presenta un'analisi preventiva dei rischi, la modalità di lavoro utilizzata e la lista degli obiettivi da raggiungere.

Il terzo capitolo descrive le tecnologie di cui si è fatto utilizzo, e come esse siano state configurate.

Il quarto capitolo descrive i casi d'uso individuati, facendo utilizzo anche dei diagrammi dei casi d'uso, e il tracciamento dei requisiti.

Il quinto capitolo approfondisce le varie scelte progettuali e architetturali prese, e la codifica di alcune importanti funzionalità.

Il sesto capitolo approfondisce il processo di verifica e validazione, con particolare attenzion ai test di unità.

Il settimo capitolo esprime le conclusioni finali.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Pianificazione dello stage

In questo capitolo è presente un'analisi preventiva dei rischi, la modalità di lavoro utilizzata e la lista degli obiettivi da raggiungere.

2.1 Analisi preventiva dei rischi

Ad inizio stage si è ragionato sui possibili rischi che si possono riscontrare, e sulle possibili soluzioni preventive o correttive. Ciò che è stato individuato è quanto segue:

* **Perdita dati**
  Descrizione: Il lavoro svolto potrebbe andare perduto a causa di guasti o malfunzionamenti dell'hardware.
  Soluzione: L'azienda ha fornito un repository dove caricare il lavoro prodotto.

* **Problematiche nello svolgimento**
  Descrizione: Lo stagista potrebbe non riuscire a capire o ad applicare alcuni concetti, specialmente in abito delle teconlogie non precedentemente conosciute ed utilizzate.
  Soluzione: Il tutor aziendale si è reso disponibile ad aiutare lo stagista, sia personalmente che tramite altri colleghi specializzati nelle varie tecnologie.

* **Contrazione virus COVID-19**
  Descrizione: Lo stagista o il tutor aziendale potrebbero contrarre infezione COVID-19.
  Soluzione: Allo stagista è stato fornito un computer portatile aziendale con tutto gli strumenti necessari per lo svolgimento dello stage. Sono stati individuati i canali di comunicazioni da utilizzare in remoto (Microsoft Teams e e-mail). Inoltre si è individuata una seconda persona di riferimento per lo stagista in casi di indisponibilità del tutor aziendale.
  Attualizzazione: Alla fine della prima settimana di lavoro lo stagista ha riscontrato di essere positivo al test per il COVID-19. Di conseguenza lo stage è proseguito da remoto, utilizzando il pc aziendale. Nonostante i lavori siano parzialmente rallentati, non si è riscontrato un discostamento significativo rispetto alle previsioni fatte.

2.2 Pianificazione iniziale

Inizialmente la pianificazione effettuata prevedeva di lavorare principalmente sul front-end, e solo parzialmente sul back-end, come si può vedere dalla pianificazione iniziale illustrata in figura 2.1. La fase di studio iniziale prevedeva infatti di studiare *React.js* per lo sviluppo del front-end, mentre ciò che si è effettivamente studiato è stato principalmente *Node.js* ed *Express* per lo sviluppo del back-end.

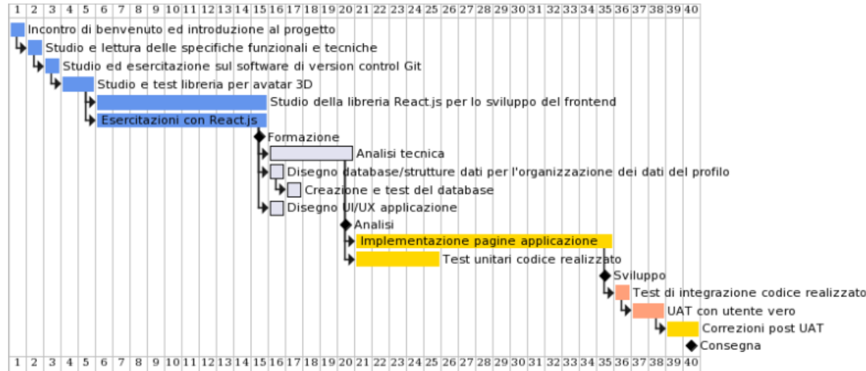


Figura 2.1: Pianificazione iniziale del lavoro

Dopo però aver discusso con lo stagista riguardo i suoi interessi e le sue aspettative, vista anche già l'esperienza passata avuta con il front-end ed in particolare il framework *React*, si è deciso di riorganizzare il lavoro in modo da fornire la possibilità di esplorare concetti e dinamiche completamente nuove. Si è quindi concordato di dare molta più rilevanza al back-end, introducendo anche alcuni concetti pertinenti al *DevOps*, ed in particolare l'utilizzo e la configurazione di Docker. Si sono dunque definiti i macro obbiettivi, riportati in sezione 2.4 e, tenendo come riferimento la pianificazione iniziale del lavoro, si è passati allo studio delle varie tecnologie.

2.3 Modalità di lavoro

La modalità di lavoro che si è pianificato di utilizzare è basata sulla divisione del lavoro in brevi periodi, a cicli incrementali, dove in ogni periodo si definiscono chiaramente gli obbiettivi da raggiungere, si pianifica e si progetta il lavoro da svolgere, si implementa la soluzione trovata e la si verifica tramite una piccola dimostrazione al tutor aziendale. Questa strategia permette di concentrarsi sul periodo immediato, minimizzando il rischio di dispersione del lavoro, e di raffinare sempre di più la pianificazione e la progettazione iniziale, man mano che si acquisisce più padronanza delle tecnologie e del problema.

2.4 Obbiettivi e requisiti

Di seguito si elencano i requisiti individuati.

2.4.1 Notazione

Si sono individuati due tipi di requisiti:

* Requisiti funzionali, contraddistinti dal carattere **F**, che definiscono gli obbiettivi da raggiungere.
* Requisiti di stage, contraddistinti dal carattere **S**, che definiscono i vincoli da rispettare.

A sua volta ogni requisito viene catalogato in due categorie:

* Requisito obbligatorio, contraddistinto dal carattere **O**, che deve necessariamente essere rispettato e/o raggiunto.
* Requisito desiderevole, contraddistinto dal carattere **D**, che porta valore aggiunto riconosciuto, ma non vincolante, rispetto alla buona riuscita dello stage.

Un requisito viene quindi definito come una stringa di due caratteri, dove il primo carattere identifica la natura del requisito (funzionale o di stage) e il secondo identifica la caratteristicità del requisito (obbligatorio o facoltativo). Questi caratteri sono poi seguiti da una coppia di numeri sequenziali. Eventuali sotto-requisiti vengoni identificati tramite l'aggiunta di un ulteriore carattere numerico preceduto da un punto.

2.4.2 Requisiti accordati

Si sono fissati i seguenti requisiti:

* Requisiti di stage
  - **SO01:** Il codice deve essere versionato con git.
  - **SO02:** Utilizzo di MongoDB per l'implementazione del database
  - **SO03:** Utilizzo di Node.js + Express per l'implementazione del back-end
  - **SO04:** Utilizzo di Typescript per l'implementazione del back-end
  - **SO05:** Utilizzo di Docker
  - **SO05.1:** Predisposizione di un container Docker per il back-end
  - **SO05.2:** Predisposizione di un container Docker per il database
* Requisiti funzionali
  - **FO01:** Il back-end deve poter comunicare con il database aziendale tramite LDAP per gestire gli accessi.
  - **FO01.1:** Deve essere presente un file di configurazione per poter facilmente impostare tutti i dati necessari alla connessione tramite LDAP.
  - **FO02:** Devono essere presenti 3 tipi di profili distinti: *Base*, *Admin*, *Viewer*.
  - **FO02.1:** I profili *Base* devono poter personalizzare il proprio profilo tramite la compilaizone di sezioni.
  - **FO02.2:** I profili *Viewer* devono poter personalizzare il proprio profilo tramite la compilaizone di sezioni, e possono creare team di lavoro.
  - **FO02.3:** I profili *Admin* devono poter creare e modificare qualsiasi tipo di profilo e team di lavoro.
  - **FO03:** Utenti di tipo *Admin* devono poter caricare dei *template* di sezioni, compilabili poi dai vari utenti.

- **FO04:** Devono essere presenti due tipi di sezioni: *Standard* e *Custom*.
- **FO05:** Deve essere possibile organizzare profili in team di lavoro.
- **FO06:** Deve essere possibile condividere un team di lavoro con utenti non riconosciuti, in particolare tramite l'utilizzo di un *URL*.
- **FD01:** Implementazione di test di unità.
- **FD02:** Possibilità di ricerca profili e team tramite semplici filtri.
- **FO03:** Le sezioni *Standard* e *Custom* devono avere comportamenti diversi.
- **FO03.1:** Le sezioni *Standard* non devono permettere agli utenti di definire campi non presenti nel *tempalate* di sezione.
- **FO03.2:** Le sezioni *Custom* devono permettere agli utenti di definire e di compilare campi non definiti nel *tempalate* di sezione.
- **FD04:** Sviluppo scheletro front-end.
- **FD05:** Implementazione codice per la visualizzazione degli avatar 3D con Threejs.
- **FD06:** Organizzazione dei dockerfiles tramite l'utilizzo di Makefiles.

Capitolo 3

Descrizione dello tecnologie utilizzate

In questo capitolo vengono descritte le tecnologie di cui si è fatto utilizzo, e come esse siano state configurate.

3.1 Introduzione al progetto

Lo scopo del progetto è la realizzazione di un back-end che espogna diversi endpoint in modo da poter gestire l'accesso, la creazione e la personalizzazione di profili, e la organizzazione di diversi profili in team di lavoro.
Ogni profilo creato presenterà dei campi standard (nome, cognome, data di nascita...) che saranno reperibili dal database aziendale tramite LDAP, e dei campi non standard, dove viene data la possibilità di descrivere le proprie abilità e conoscenze. Sarà inoltre necessario poter salvare e associare un avatar 3D ad ogni profilo . I dati persistenti saranno quindi salvati su un database non relazionale, isolato dal back-end tramite l'utilizzo di due container Docker.

3.2 Strumenti e tecnologie

Nella prima fase di stage ci si è concentrati sullo studio delle tecnologie e sulla configurazione degli strumenti che poi sono stati necessari per lo svolgimento di tutto lavoro.
Di seguito vengono trattati gli argomenti che sono stati studiati, e quali eventuali scelte di metodologia o configurazione sono state prese.

3.2.1 Repository

Come primo argomento si è trattato *git* ed in particolare il *gitflow*, e si è quindi predisposta la *repository* dove poi è stato mantenuto il codice.
git è un sistema di versionamento distribuito che permette di tracciare i cambiamenti su di un insieme di file.
gitflow è una metodologia di lavoro che detta come organizzare ed effettuare i *merge* tra i vari *branches*.

Un *branch* in *git* è essenzialmente un puntatore ad una versione dei cambiamenti. Secondo il *gitflow* sono sempre presenti 2 *branch*, uno denominato *main* (o *master*) e uno denominato *develop*.
Quando bisogna sviluppare una nuova funzionalità si crea un *branch* a partire dal *branch develop* e, una volta implementata, si effettua il merge su *develop*.
Quando più funzionalità su *develop* sono pronte al rilascio, si crea un nuovo *branch*, sempre basato su *develop*, chiamato *release* dove, dopo aver risolto le ultime eventuali problematiche, si effettua un *merge* con *main*.
Un esempio di questo può essere osservato nella fiugura figura 3.1, dove si vede che ogni nuova funzionalità viene implementate in un *branch* specializzato a partire dal *branch develop*, e quando si è vicini al rilascio si prepara un *branch release*, poi portato in *master*.

Il progetto tuttavia è stato interamente sviluppato da me, quindi ogni nuova funzionalità è stata implementata sequenzialmente una volta terminata la precedente, e aggiunta su *develop* con un *merge*, mentre il *main* è rimasto inalterato.

3.2.2 Database

L'utilizzo di MongoDB, olre ad essere richiesto dai requisiti, presenta queste caratteristiche:

* Non relazionale, offre quindi la flessibilità richiesta per la gestione dei dati, in particolare delle sezioni.
* Presenta un'ottima documentazione, di facile utilizzo e con molti tutorial disponibili online.
* Viene utilizzato in azienda, e viene quindi offerto aiuto in caso di necessità.

3.2.3 Back-end

Per lo sviluppo del back-end si è deciso di utilizzare Node.js ed Express. Node.js è un *runtime environment* per *javascript*, mentre Express è un framework che semplifica significativamente l'utilizzo di Node.js.
Si è inoltre deciso di utilizzare JSON Web Token per gestire l'autenticazione. Utilizzando la libreria jsonwebtoken per Node.js si può generare un token univoco per utente che contiene delle informazioni riguardanti l'utente stesso.
La procedura di utilizzo standard, che corrisponde a quella utilizzata nel progetto, prevede la creazione del token in seguito ad il *log in* di un utente, e la sua aggiunta alla risposta inviata. Ad ogni successiva richiesta sarà quindi aggiunto il token stesso che, una volta validato, confermerà l'autenticità della richiesta.

3.2.4 Typescript

Tutto il codice prodotto è stato scritto utilizzando Typescript, un'estensione del linguaggio javascript che permette, tra le altre funzionalità, di utilizzare tipi, interfacce e classi.
Con l'utilizzo di Typescript il codice prodotto risulta più manutenibile, e risulta inoltre possibile rilevare vari problemi già a *compile time*, evitando quindi che si verifichino a *runtime*.

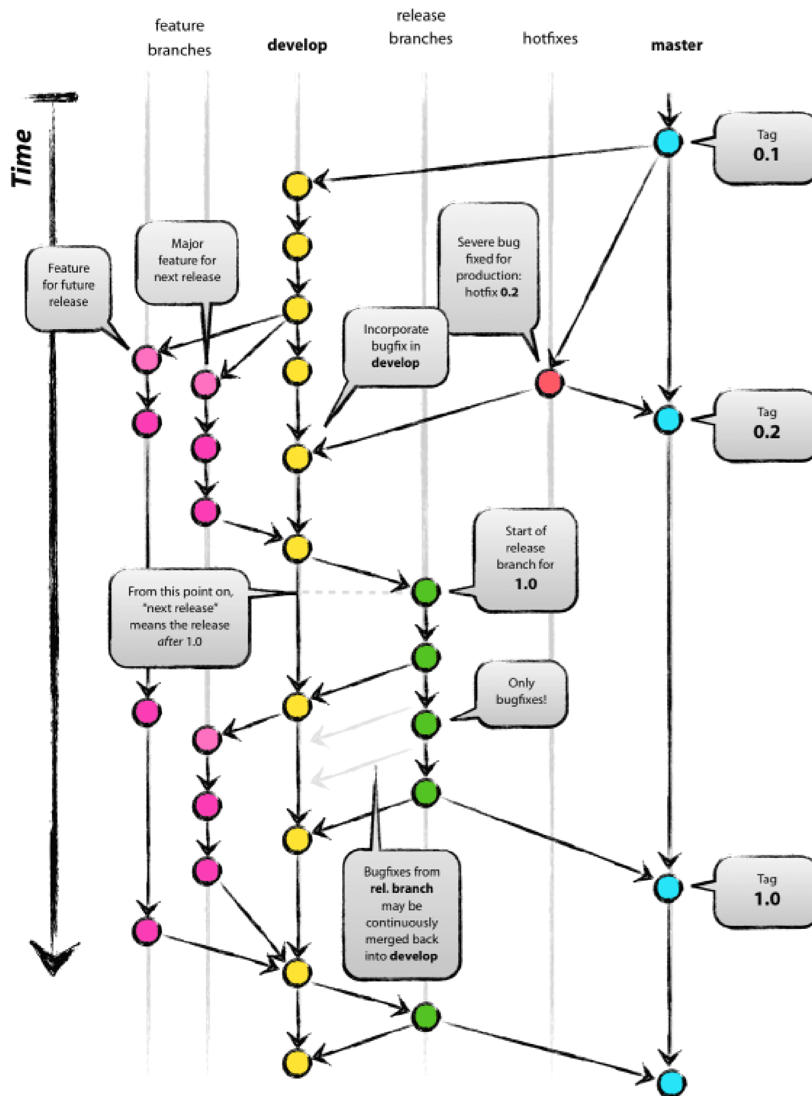


Figura 3.1: Esempio di gitflow, **https://nvie.com/posts/a-successful-git-branching-model/**

3.2.5 Docker

Docker è un software che permette di sviluppare, eseguire e spedire applicazioni in unità standardizzate, chiamate *container*, che forniscono tutto il necessario al corretto funzionamento dell'applicazione.
A differenza di una *virtual machine*, che virtualizza l'hardware sottostante, un *container* virtualizza il sistema operativo, e risulta quindi più leggero e facilmente gestibile.
Grazie all'utilizzo di Docker è inoltre possibile definire delle variabili di ambiente, utilizzabili poi da Node.js, all'interno del container.
Tutto questo può essere visto in figura 3.2, ed in particolare si può notare la definizione

delle variabili d'ambiente utilizzate per definire la connessione tramite LDAP con il database aziendale (censurate per motivi di sicurezza).

```
16    back-end:
17      image: back-end
18      build:
19        context: ./server/
20        target: dev
21      user: "node"
22      environment:
23        - CONNECTIONSTRING=mongodb://mongo:27017/mydb
24        - LDAP_URL=
25        - LDAP_BASE_DN=
26        - LDAP_USERNAME=
27        - LDAP_PASSWORD=
28        - GROUP_BASE_PROFILE=OM_DELIVERYONE_SVILUPPATORE
29        - GROUP_VIEWER_PROFILE=OM_DELIVERYONE_PM
30        - GROUP_ADMIN_PROFILE=OM_DELIVERYONE_DELIVERY_MANAGER
31        - JWT_KEY=
32      ports:
33        - 5000:5000
34        # - 9229:9229
35      volumes:
36        - ./server:/usr/back-end
37        - /usr/back-end/node_modules
38      command: "npm run dev"
39      # command: "npm run debug" -> package.json: "debug": "nodemon -L --exec \"node --inspect-brk=0.0.0.0:9229 --require ts-node/register src/app.ts\""
40      depends_on:
41        - mongo
```

Figura 3.2: Parte del file *docker-compose.yml* prodotto, che fornisce istruzioni sulla costruzione del *container* per il back-end e che permette di definire delle variabili d'ambiente

Per lo sviluppo del back-end si è fatto utilizzo del *bind mount*, che permette di montare dei file presenti nella macchina ospitante direttamente nel *container*, permettendo quindi la condivisione degli stessi. In particolare tutto il codice sorgente del back-end è stato montato direttamente sul *container* del back-end, in modo da sviluppare e testare il codice prodotto direttamente sul *container*, isolandosi quindi dalla macchina ospitante utilizzata per scrivere il codice.

Inoltre, per comodità, nel *container* dedicato al database è stato fatto utilizzo di un volume, come si può vedere nella figura 3.3, in modo da poter salvare i dati persistenti sulla macchina ospitante ed evitando di perdere gli stessi ad ogni riavvio del container.

```
  mongo:
    image: mongo:5
    volumes:
    - mongodb:/data/db
    # - mongodb_config:/data/configdb
    command: "mongod --quiet --logpath /dev/null" # disable mongo logging

volumes:
  mongodb:
# mongodb_config:
```

Figura 3.3: Parte del file *docker-compose.yml* prodotto, che fornisce istruzioni sulla costruzione del *container* per il database, e che definisce un volume chiamato mongodb

Per facilitare la gestione e l'utilizzo dei container si è utilizzato l'applicazione Docker Desktop.

3.2.6 LDAP

LDAP (Lightweight Directory Access Protocol) è un protocollo standard per l'interrogazione e la modifica dei servizi di directory. Tramite esso si può interagire in modo

standard per reperire un qualsiasi raggruppamento di informazioni, espresso come record di dati, organizzato in modo gerarchico.
In particolare nel progetto tramite l'utilizzo di *activedirectory2*, un *client* di ldapjs, a sua volta un framework per implementare LDAP *clients* e *servers* in Node.js, si può interrogare l'Active Directory aziendale, che mantiene tutti i dati dei dipendenti, in particolare nome, cognome e gruppi di appartenenza.

3.2.7 Ready Player Me

Ready Player Me è una piattaforma per la creazione di *avatar* digitali personalizzabili tramite un'interfaccia web integrabile da altre applicazioni. L'avatar viene poi memorizzato direttamente nei loro server e viene reso disponibile per essere scaricato tramite un *URL* persistente e unica.

3.2.8 Visual Studio Code

Tutto il codice prodotto è stato sviluppato utilizzando Visual Studio Code, l'editor di codice sorgente più diffuso ed utilizzato in ambito Node.js.

3.2.9 Insomnia

Per semplificare il testing dei vari endpoint esposti dal back-end si è fatto utilizzo dell'applicazione *Insomnia*.
Questa permette di creare ed inviare richieste facilmente personalizzabili ai vari endpoint di interesse in modo da vedere se queste sono ricevute e processate correttamente.

Capitolo 4

Analisi dei requisiti

Il seguente capitolo descrive i casi d'uso individuati, facendo utilizzo anche dei diagrammi dei casi d'uso, e il tracciamento dei requisiti.

4.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) sono diagrammi UML dedicati alla descrizione delle funzionalità e dei servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.

4.1.1 Attori

Nella stesura dei diagrammi si è deciso di mantenere un alto livello di astrazione, in modo da descrivere le funzionalità dal punto di vista degli utenti utilizzatori, e non di individuare un possibile front-end come attore principale che interagisce con il sistema sotto analisi, il back-end.
Questo permette di meglio individuare e descrivere le varie funzionalità, oltre che a meglio rappresentare come effettivamente poi il prodotto andrebbe utilizzato.
Tutti gli attori individuati, presentati in figura 4.1, sono divisi in due grupppi, attori principali e attori secondari.

Gli attori principali individuati sono dunque i seguenti:

* **Utente non registrato**: dipendente che non ha ancora effettuato la registrazione al sistema.
* **Utente non autenticato**: utente che ha eseguito la registrazione al sistema, ma che non risulta in possesso di un token di autenticazione.
* **Utente Base**: utente che ha effettuato l'accesso, appartenente al gruppo di utenti con funzionalità di base.
* **Utente Viewer**: utente che ha effettuato l'accesso, appartenente al gruppo di utenti con funzionalità avanzate; generalizzazione dell'utente **Base**.

* **Utente Admin**: utente che ha effettuato l'accesso, appartenente al gruppo di utenti con diritti da amministratore; generalizzazione dell'utente **Viewer**.

Per quanto riguarda invece gli attori secondari è stato individuato solamente un attore:

* **DB aziendale**: database aziendale, accessibile tramite LDAP, conenente i dati anagrafici dei vari dipendenti, oltre che al gruppo che determina la tipologia dell'utente.

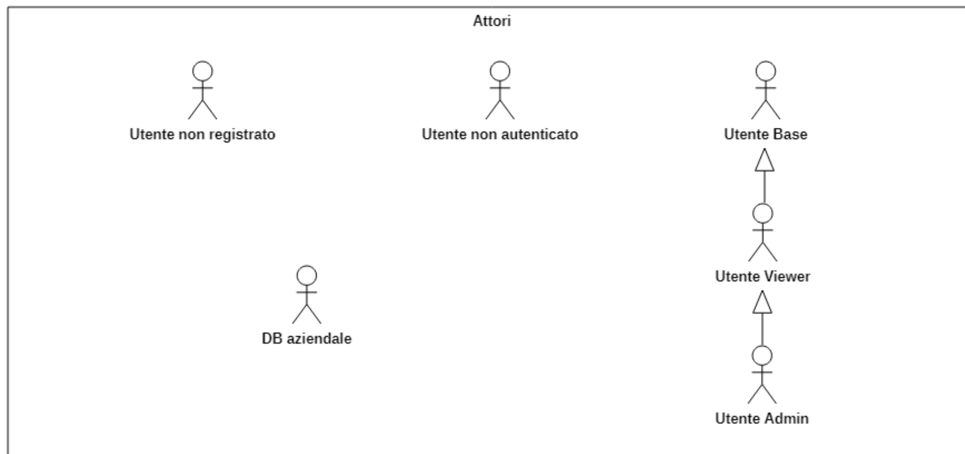


Figura 4.1: Attori individuati

4.1.2 Diagrammi dei casi d'uso

Di seguito, in figura 4.2 e figura 4.3, vengono presentati i diagrammi dei casi d'uso, con successiva spiegazione testuale.

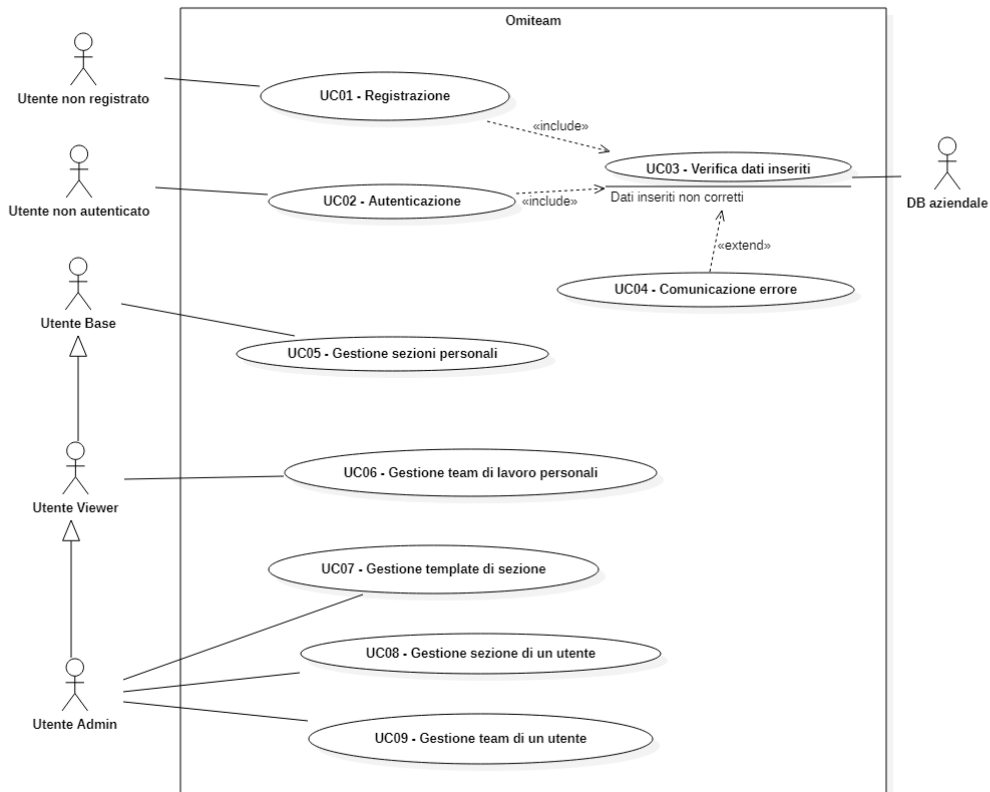


Figura 4.2: Scenario principale

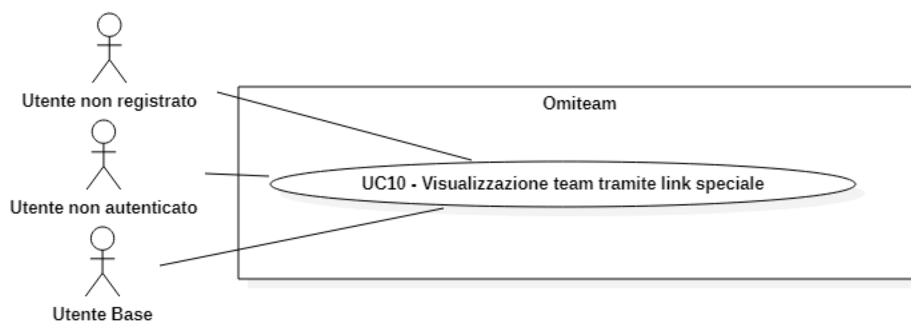


Figura 4.3: Scenario principale 2

UC01: Registrazione

Attori Principali: Utente non registrato.

Precondizioni: L'utente non ha ancora effettauta la registrazione presso il sistema.

Descrizione: L'utente vuole registrarsi presso il sistema.

Postcondizioni: L'utente risulta registrato ed autenticato.

Descrizione:

1. Inserimento email aziendale (UC03.1).
2. Inserimento password aziendale (UC03.2).

UC02: Autenticazione

Attori Principali: Utente non autenticato.

Precondizioni: L'utente non ha ancora effettauta l'autenticazoine presso il sistema, ma ha già effettuato la registrazione in un momento precedente.

Descrizione: L'utente vuole autenticarsi presso il sistema.

Postcondizioni: L'utente risulta autenticato.

Descrizione:

1. Inserimento email aziendale (UC03.1).
2. Inserimento password aziendale (UC03.2).

UC03: Inserimento e verifica dati (figura 4.4)

Attori Principali: Utente non registrato / Utente non autenticato.

Attori Secondari: DB aziendale.

Precondizioni: L'utente sta cercando di **registrarsi / autenticarsi**.

Descrizione: L'utente vuole **registrarsi / autenticarsi** presso il sistema.

Postcondizioni: L'utente risulta **registrato ed autenticato / autenticato**.

Descrizione:

1. L'utente inserisce l'email aziendale (UC03.1).
2. L'utente inserisce la password aziendale (UC03.2).

Estensioni:

1. **Nel caso in cui l'untente non inserisca i dati aziendali corretti:**
   (a) L'utente non viene **registrato / autenticato** presso il sisitema.
   (b) Viene inviato un messaggio di errore.

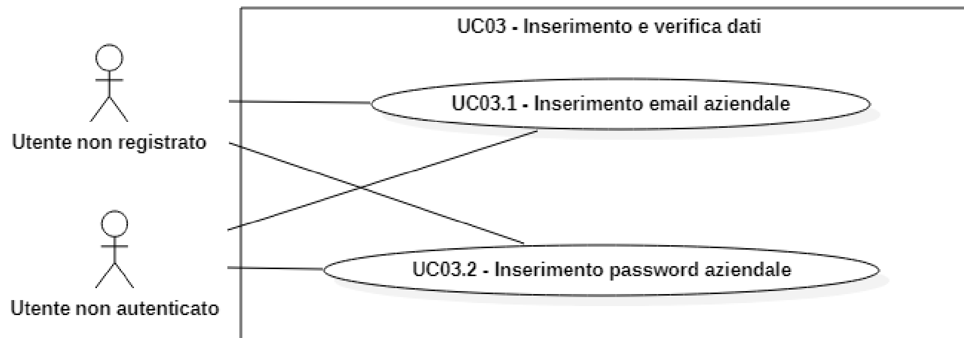


Figura 4.4: UC03 - Inserimento e verifica dati

UC03.1: Inserimento email aziendale

Attori Principali: Utente non registrato / Utente non autenticato.

Precondizioni: L'utente sta cercando di **registrarsi / autenticarsi**.

Descrizione: L'utente deve inserire la propria email aziendale per **registrarsi / autenticarsi** presso il sistema.

Postcondizioni: L'utente ha inserito la propria email aziendale.

Descrizione:

1. L'utente inserisce la propria email aziendale.

UC03.2: Inserimento password aziendale

Attori Principali: Utente non registrato / Utente non autenticato.

Precondizioni: L'utente sta cercando di **registrarsi / autenticarsi**.

Descrizione: L'utente deve inserire la propria password aziendale per **registrarsi / autenticarsi** presso il sistema.

Postcondizioni: L'utente ha inserito la propria password aziendale.

Descrizione:

1. L'utente inserisce la propria password aziendale.

UC04: Comunicazione errore

Attori Principali: L'utente sta cercando di **registrarsi / autenticarsi**.

Precondizioni: L'utente non ha ancora effettauta la registrazione presso il sistema.

Descrizione: L'utente vuole **registrarsi / autenticarsi** presso il sistema ma inserisce dati errati.

Postcondizioni: L'utente riceve comunicazione che i dati aziendali inseriti sono scorretti.

Descrizione:

1. L'utente cerca di **registrarsi** / **autenticarsi** ma inserisce dati dati errati.
2. L'utente riceve comunicazione che i dati inseriti sono scorretti.

UC05: Gestione sezioni personali

Attori Principali: Utente Base.

Precondizioni: L'utente ha effettuato l'accesso.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** una sezione.

Postcondizioni: L'utente **aggiunge/modifica/elimina** una sezione correttamente.

Descrizione:

1. L'utente invia una richiesta di gestione di una sezione, in riferimento ad un template di sezione.

UC06: Gestione team di lavoro personali

Attori Principali: Utente Viewer.

Precondizioni: L'utente ha effettuato l'accesso.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** una team di lavoro da lui creato.

Postcondizioni: L'utente **aggiunge/modifica/elimina** un team di lavoro da lui creato.

Descrizione:

1. L'utente invia una richiesta di gestione di un team di lavoro da lui creato.

UC07: Gestione template di sezione

Attori Principali: Utente Admin.

Precondizioni: L'utente ha effettuato l'accesso.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** un singolo tempalte di sezione.

Postcondizioni: L'utente **aggiunge/modifica/elimina** un singolo tempalte di sezione.

Descrizione:

1. L'utente invia una richiesta di gestione di un singolo tempalte di sezione, specificando la sua tipologia (*Standard* o *Custom*).

UC08: Gestione sezione di un utente

Attori Principali: Utente Admin.

Precondizioni: L'utente ha effettuato l'accesso.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** una sezione di un utente.

Postcondizioni: L'utente **aggiunge/modifica/elimina** una sezione di un utente.

Descrizione:

1. L'utente invia una richiesta di gestione di una sezione di un utente.

UC09: Gestione team di un utente

Attori Principali: Utente Admin.

Precondizioni: L'utente ha effettuato l'accesso.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** un team di un utente.

Postcondizioni: L'utente **aggiunge/modifica/elimina** un team di un utente.

Descrizione:

1. L'utente invia una richiesta di gestione di un team di un utente.

UC10: Visualizzazione team tramite link speciale

Attori Principali: Utente non registrato / Utente non autenticato / Utente Base.

Precondizioni: L'utente è in possesso di un link speciale per la visualizzazione di un team.

Descrizione: L'utente vuole **aggiungere/modificare/eliminare** un team di un utente.

Postcondizioni: L'utente **aggiunge/modifica/elimina** un team di un utente.

Descrizione:

1. L'utente invia una richiesta di gestione di un team di un utente.

4.2 Tracciamento dei requisiti

Dall'analisi dei requisiti e dai casi d'uso individuati è stata stilata la tabella che traccia i requisiti funzionali obbligatori in rapporto ai casi d'uso.
Nella tabella 4.1 sono riassunti i requisiti e il loro tracciamento con i casi d'uso.

Tabella 4.1: Tabella del tracciamento dei requisti funzionali obbligatori

Requisito	Descrizione	Use Case
FO01	Il back-end deve poter comunicare con il database aziendale tramite LDAP per gestire gli accessi	UC01 e UC02
FO02.1	I profili *Base* devono poter personalizzare il proprio profilo tramite la compilaizone di sezioni	UC05
FO02.2	I profili *Viewer* devono poter personalizzare il proprio profilo tramite la compilaizone di sezioni, e possono creare team di lavoro	UC05 e UC06
FO02.3	I profili *Admin* devono poter creare e modificare qualsiasi tipo di profilo e team di lavoro	UC08 e UC09
FO03	Utenti di tipo *Admin* devono poter caricare dei *template* di sezioni, compilabili poi dai vari utenti	UC07
FO04	Devono essere presenti due tipi di sezioni: *Standard* e *Custom*	UC07
FO05	Deve essere possibile organizzare profili in team di lavoro	UC06
FO06	Deve essere possibile condividere un team di lavoro con utenti non riconosciuti, in particolare tramite l'utilizzo di un *URL*	UC10

Capitolo 5

Progettazione e codifica

In questo capitolo si descrivono in dettaglio le varie scelte progettuali e architetturali fatte, e la codifica di alcune importanti funzionalità.

5.1 Progettazione

Per quanto riguarda la parte di progettazione, gli argomenti che sono stati trattati principalmente sono stati:

* Come organizzare il codice
* Quali classi utilizzare
* Come strutturare il database
* Quali endpoint esporre

5.1.1 Il modello

Durante la progettazione delle classi del modello si è prodotto un diagramma delle classi, presentato in figura 5.1, cercando di suddividere al più possibile le responsabilità e di minimizzare le dipendenze.

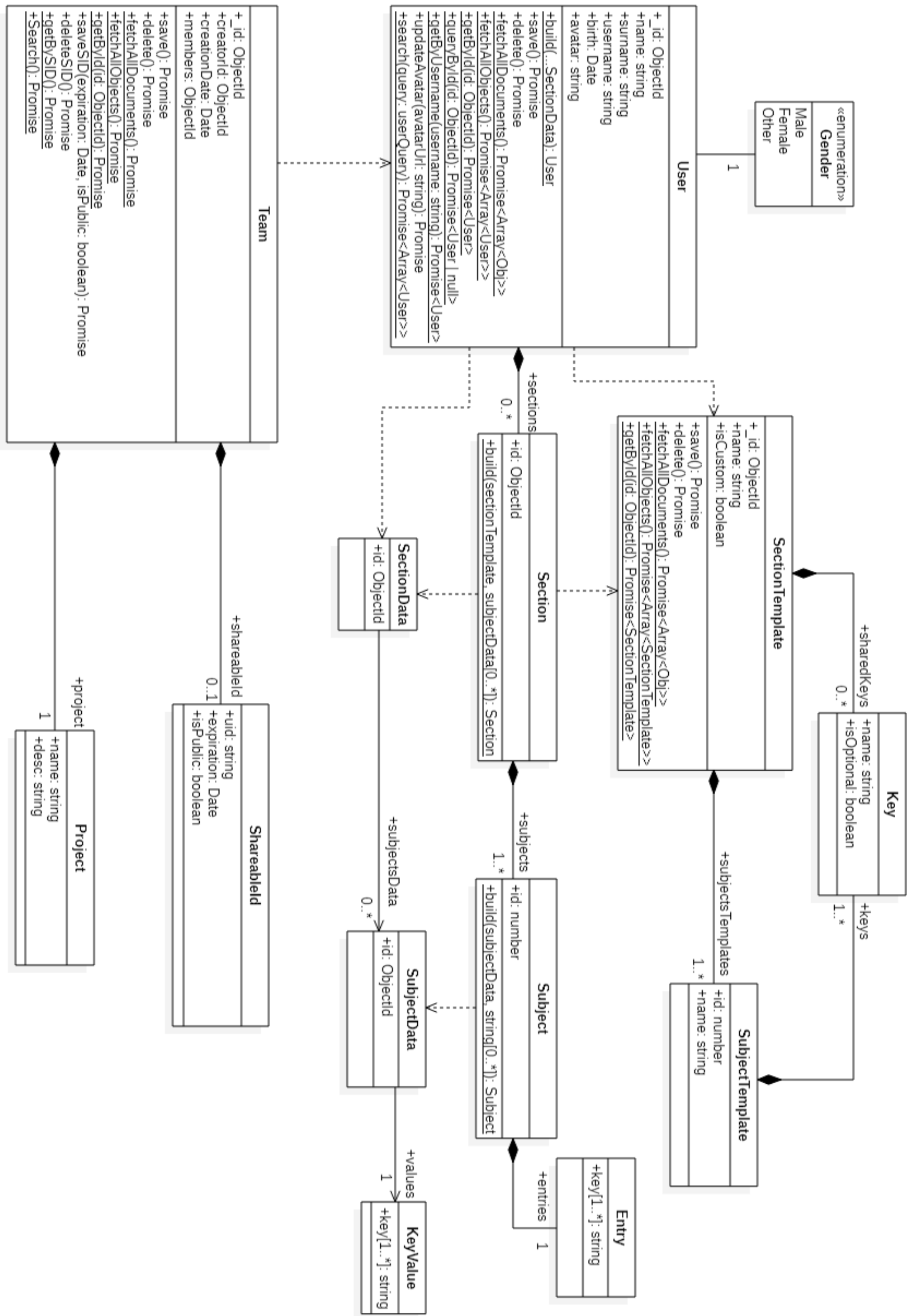


Figura 5.1: Diagramma delle classi pertinenti al modello

Sections

Particolare attenzione è stata posta su come organizzare le sezioni, e la migliore soluzione individuata distingue tre tipi diversi di sezioni:

1. **SectionTemplate**: caricate nel database da utenti di tipo *Admin*, le *Section Template* possono essere riferite dagli utenti per popolare le proprie sezioni con i propri dati.
   Sono presenti i seguenti campi:

   * **name: string**: il nome della sezione
   * **isCustom: boolean**: utilizzato per distinguere sezioni *standar* da sezioni *custom*; una sezione *custom* permette agli utenti di aggiungere i propri *subject* e le proprie *key*, mentre nelle sezioni *standard* i *subject* e le *key* delle *entry* devono essere definite da un utente *Admin* ed inserite nel database, così da poter essere utilizzate come riferimento.
   * **sharedKeys: string[]**: array di *key* condivise da tutti i *subject* di questa sezione, presente per evitare di dover ripetere la stessa *key* per ogni singolo *subject*.
   * **subjectTemplates: SubjectTemplate[]**: array di *SubjectTemplate* definite per questa *SectionTemplate*.

2. **SectionData**: rappresenta tutti i dati che un utente vuole inserire in una propria *Section*, in riferimento ad una *SectionTemplate*. Un *SectionData* contiene solamente tutti i dati di tutti i vari *subject* che un utente vuole inserire nella propria *Section*, utilizzando riferimenti a *SectionTemplate* e *SubjectTemplate*. *SectionData* presenta dunque i seguenti campi:

   * **id: ObjectId**: identificatore in riferimento ad una *SectionTemplate*
   * **subjectsData: SubjectData[]**: array di *SubjectData* che contiene tutte le coppie *Key-Value* per tutti i *Subject* della sezione a cui l'utente vorrebbe assegnare.

3. **Section**: rappresenta una istanziazione di una *SectionTemplate* per un singolo utente, con tutti i dati da lui inseriti.
   La differenza principale dunque tra *SectionData* e *Section* è che una *Section* è una istanziazione valida della *SectionTemplate* a cui si riferisce, mentre *SectionData* potrebbe contenere dei valori non validi rispetto alla *SectionTemplate* a cui si riferisce. *Section* presenta dunque i seguenti campi:

   * **id: ObjectId**: identificatore in riferimento ad una *SectionTemplate*
   * **subjects: Subject[]**: array di *Subject* che contiene tutte le coppie codificate nelle varie *Entry* di tutti i *Subject* della sezione.

Vengono ora descritti i vari *subject* appena menzionati:

* **SubjectTemplate**: una voce con un proprio nome all'interno di una *SectionTemplate*, utilizzata per definire un singolo argomento di una sezione. Può avere le proprie specifiche *key*.
  Sono presenti i seguenti campi:

- **id: number**: un numero identificatore univoco rispetto alla sezione di cui fa parte, viene definito da un utente *Admin* in modo da poter ordinare per importanza i vari *subject* all'interno di una *section.*
- **name: string**: il nome proprio del *subject*
- **keys: Key[]**: array opzionale in cui definire le varie *key* specifiche per questo *subject.*

* **SubjectData**: rappresenta i dati di uno specifico *subject* in una specifica *section* che un utente vuole inserire nel proprio profilo, in riferimento ad un *SubjectTemplate* di una *SectionTemplate.* Sono presenti dunque tutti i valori che l'utente vuole assegnare alle varie *key* del *subjectTemplate* a cui fa riferimento. *SubjectData* presenta infatti i seguenti campi:

  - **id: ObjectId**: identificatore che dovrebbe riferire un *SubjectTemplate* all'interno di una specifica *SectionTemplate.*
  - **values: KeyValue**: un oggetto contenente i valori da assegnare alle varie *key* del *subjectTemplate* in riferimento. Contiene un numero variabile di parametri, ed ogni parametro viene visto come una coppia chiave-valore, in cui il nome del parametro dovrebbe coincidere con il valore di una *key* del *subjectTemplate* e quindi dovrebbe fungere da riferimento a quella specifica *key*, mentre il valore del parametro coincide con il valore che l'utente vuole assegnare alla *key* a cui il suo nome fa riferimento.

* **Subject**: rappresenta una istanziazione di una *SubjectTemplate* di una istanziazione di una *SectionTemplate* per un singolo utente, con tutti i dati da lui inseriti.
  La differenza principale dunque tra *SubjectsData* e *Subject* è, ancora una volta, che un *subject* è una istanziazione valida del *SubjectTemplate* a cui si riferisce, mentre *SubjectData* potrebbe contenere dei dati non validi rispetto al *SubjectTemplate* a cui si riferisce. *Subject* presenta dunque i seguenti campi:

  - **id: number**: identificatore in riferimento ad un *SubjectTemplate* all'interno di una specifica *SectionTemplate.*
  - **entries: Entry**: un oggetto contenente i valori assegnati alle varie *key* del *SubjectTemplate* in riferimento. Contiene un numero variabile di parametri, ed ogni parametro viene visto come una coppia chiave-valore, in cui il nome del parametro coincide con il valore di una *key* del *subjectTemplate* e funge quindi da riferimento a quella specifica *key*, mentre il valore del parametro coincide con il valore che l'utente ha assegnato alla *key* a cui il suo nome fa riferimento.

Infine le *key* sono concetti (come abilità, competenza, proprietà, risultati ottenuti...) a cui un utente può fare riferimento e dare dei valori, nel contesto di un *subject* all'interno di una *section.* Sono definite da utenti *Admin*:

* nel campo **sharedKeys** di una *SectionTemplate.*
* nel campo **keys** di un *SubjectTemplate.*

Le *key* presentano i seguenti campi:

∗ **name: string**: il nome della *key*.

∗ **isOptional: boolean**: quando un utente vuole assegna i propri valori alle varie *key* di un *SubjectTemplate*, deve necessariamente indicare un valore per tutte le *key* con campo **isOptional** === **false**.

ShareableId

I vari team di lavoro devono poter essere resi disponibili anche ad utenti non registrati tramite un link speciale.
La soluzione individuata prevede di dare la possibilità ai vari utenti proprietari dei team di creare un identificativo e di condividerlo tramite un *URL*.
Il link può essere disabilitato tramite il campo **isPublic**, e può essere impostato con una data di scadenza tramite il campo **expiration**.

5.1.2 Il database

Nel database si prevede di salvare i dati così come presenti nel modello in modo da facilitare la lettura e il salvataggio dei dati.
I *documents*, così riferiti gli oggetti salvati con MongoDB, corrispondono quindi alla traduzione in BSON degli oggetti creati dalle classi presenti nel modello.
In particolare si è pensato di utilizzare tre *collections* diverse:

∗ **Sections**: istanze della classe *SectionTemplate* caricate dagli utenti *Admin*.

∗ **Users**: istanze della classe *User*, gli utenti registrati.

∗ **Team**: istanze della classe *Team*, i vari team di lavoro creati da utenti *Admin* o *Viewer*.

Avatar 3D Gli avatar 3D verranno salvati in una cartella del back-end, mentre nel DB il campo avatar di un utente farà riferimento al *path* in cui è possibile trovare il *file* dell'avatar, in modo che possa essere servito staticamente dal back-end stesso.

5.1.3 Divisione delle responsabilità

In modo da rendere il codice più facilmente gestibile e manutenibile si è deciso di suddividerlo basandosi sul design pattern architetturale del *MVC(Model-View-Controller)*.
Secondo il *MVC* il codice si divide in 3 diverse parti in base alla loro funzione:

∗ **Model**: La logica dell'applicazione in termini del dominio del problema; gestisce direttamente i dati utili all'applicazione ed interagisce con il database.

∗ **View**: Visualizza i dati del *model* e ineragisce con l'utente.

∗ **Controller**: Riceve le richieste dell'utente dalla *view* e le converte in comandi per il *model* o la *view*.

Tuttavia, tutta la logica di visualizzazione dei dati viene gestita nel front-end, eliminando la necessità della parte di *view* nel back-end. Nel back-end bisogna inoltre gestire gli endpoint, e si può quindi individuare un'ulteriore categoria di codice, che gestisce la dichiarazione degli stessi. La divisione del codice prevista è quindi la seguente:

* **Routes**: Dichiara e gestisce le richieste dirette ai vari endpoint, verificandone la validità e risolvendole utilizzando il *controlloer*.
* **Controller**: Processa le richieste, interagendo con il *model* dove e quando necessario.
* **Model**: Gestisce e risolve le richieste del *controller*, interagendo direttamente con il database.

5.1.4 Profili

L'applicazione prevede di distinguere tre tipi di profili diversi, ognuno con i propri diritti e limitazioni.

* **Base**: Permette di creare e modificare soltanto il proprio profilo.
* **Viewer**: Permette di creare e modificare il proprio profilo, e di visualizzare e creare team di lavoro.
* **Admin**: Permette di creare e modificare sia il proprio profilo che il profilo di altri utenti; permette di creare nuovi template di sezioni; permette di creare e modificare team creati da qualsiasi utente.

5.1.5 Endpoints

In modo da esporre un Application Program Interface che sia facilmente usufruibile si è deciso di progettare un *REST API* che segua tutte le migliori convenzioni.
Un *REST API* è un Application Program Interface che segue i vincoli architetturali REST, e che permette di interagire con altri servizi che seguono le stesse regole.
In *REST* una risorsa (sia essa un documento, un'immagine, una collezione di risorse...) è la principale astrazione di informazioni, ed è caratterizzata da un nome.
Con endpoint nel contesto di un *REST API* ci si riferisce alla *route*, che corrisponde al nome della risorsa con cui si vuole interagire, e un verbo HTTP, che indica quale azione si vuole svolgere sulla risorsa.
Solitamente si utilizzano i seguenti verbi:

* **GET**: Contraddistingue una richiesta di visualizzazione della risorsa in riferimento.
* **POST**: Contraddistingue una richiesta di creazione di una nuova risorsa.
* **PUT**: Contraddistingue una richiesta di aggiornamento di una risorsa già esistente, o di creazione di una nuova risorsa (in caso non esista già).
* **DELETE**: Contraddistingue una richiesta di eliminazione della risorsa in riferimento.

Implementando questi quattro endpoint, in riferimento ad una specifica risorsa, si può dire che si supportano le quattro operazioni di base: creazione, lettura, aggiornamento e rimozione; riferite tipicamente come *CRUD (Create, Read, Update, Delete)*.

Gli endpoint che si sono previsti utilizzare sono i seguenti:
(Le risorse che contengono caratteri preceduti dai caratteri *':'* vengono trattati come parametri, e cambiano quindi da risorsa a risorsa)

* **view**: endpoint che gestiscono richieste di visualizzazione di team di lavoro senza la necessità di effettuare alcun accesso, ma disponibili soltanto se si è in possesso della chiave in riferimento al team di interesse.
  - **GET /view/:sid**: gestisce le richieste di visualizzazione di uno specifico team, riferito tramite *sid*.
* **auth**: endpoint che gestiscono richieste di utenti non riconosciti.
  - **POST /auth/signup**: gestisce le richieste di creazione di un nuovo profilo.
  - **POST /auth/login**: gestisce le richieste di accesso per utenti registrati.

  I tre endpoint appena presentati rappresentano le *Public Routes* disponibili, ovvero non necessitano di nessun tipo di autorizzazione. Tutti gli endpoint a seguire invece necessitano che l'utente abbia almeno effettuato l'accesso, con eventuali ulteriori vincoli specificati caso per caso.
* **sections**: endpoint che gestiscono richieste riguardanti i *template* di sezioni.
  - **GET /sections**: richiesta di visualizzazione di tutte le sezioni presenti nel database.
  - **POST /sections**: richiesta di aggiunta al database di una sezione. Disponibile soltanto ad utenti di tipo *Admin*.
  - **GET /sections/:id**: richiesta di visualizzazione di una specifica sezione, identificata tramite *id*.
  - **PUT /sections/:id**: richiesta di aggiornamento di una specifica sezione, identificata tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*.
  - **DELETE /sections/:id**: richiesta di rimozione di una specifica sezione, identificata tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*.
* **users**: endpoint che gestiscono richieste riguardo agli utenti registrati.
  - **GET /users**: richiesta di visualizzazione di tutti gli utenti presenti nel database.
  - **POST /users**: richiesta di aggiunta forzata nel database di un nuovo utente. Disponibile soltanto ad utenti di tipo *Admin*. Solitamente un nuovo utente viene aggiunto al database tramite una richiesta a **POST /auth/signup**.
  - **GET /users/:id**: richiesta di visualizzazione di uno specifico utente, identificato tramite *id*.
  - **PUT /users/:id**: richiesta di aggiornamento di uno specifico utente, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti che stanno richiedendo di aggiornare il proprio profilo.
  - **DELETE /users/:id**: richiesta di rimozione di uno specifico utente, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*.
  - **PUT /users/:id/avatar**: richiesta di salvataggio di un avatar 3D per uno specifico utente, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti che stanno richiedendo di aggiornare il proprio avatar 3D. L'avatar 3D deve essere presentato come un link ad una risorsa *.glb* scaricabile tramite internet.

* **teams**: endpoint che gestiscono richieste riguardo ai team di lavoro.
  - **GET /teams**: richiesta di visualizzazione di tutti i team presenti nel database.
  - **POST /teams**: richiesta di aggiunta al database di un team. Disponibile soltanto ad utenti di tipo *Admin* o *Viewer*.
  - **GET /teams/:id**: richiesta di visualizzazione di uno specifico team, identificato tramite *id*.
  - **PUT /teams/:id**: richiesta di aggiornamento di uno specifico team, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti *Viewer* che richiedono di aggiornare un team da loro creato.
  - **DELETE /teams/:id**: richiesta di rimozione di uno specifico team, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti *Viewer* che richiedono di rimuovere un team da loro creato.
  - **PUT /teams/:id/sid**: richiesta di aggiornamento dello *ShareableId* del team, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti *Viewer* che richiedono di aggiornare lo *ShareableId* di un team da loro creato.
  - **DELETE /teams/:id/sid**: richiesta di rimozione dello *ShareableId* di un team, identificato tramite *id*. Disponibile soltanto ad utenti di tipo *Admin*, o ad utenti *Viewer* che richiedono di rimuovere lo *ShareableId* di un team da loro creato.

5.2 Codifica

LDAP

Per gestire gli accessi, come già menzionato, è stato necessario utilizzare *activedirectory2* per poter comunicare con il database aziendale. In particolare è stata implementata una classe **LDAP** che gestisce la comunicazione con l'*Active Directory* aziendale, esponendo dei metodi di semplice utilizzo. I metodi che esposti sono i seguenti:

* **auth**: metodo per gestire l'accesso al profilo della applicazione tramite le credenziali aziendali.

* **findUser**: metodo per effettuare la ricerca di un utente nel database aziendale.

* **getUserGroups**: metodo per la lettura dei gruppi di appartenenza dell'utente in riferimento.

* **getProfileType**: metodo che ritorna il tipo dell'utente, tra *Base, Viewr, Admin*.

La classe **LDAP** fa inoltre utilizzo di una funzione **getLDAP()** che viene chiamata non appena il server viene fatto partire, e che cerca di connettersi al database aziendale, lanciando un errore in caso non sia possibile.
Il suo comportamento si può intuire osservando la figura 5.2: innanzitutto viene controllato che le varie variabili d'ambiente siano settate correttamente, sia quelle per la connessione con il database aziendale, sia quelle per mappare i vari gruppi aziendali ai tipi di profilo dell'applicazione. Si instanzia quindi un oggetto di configurazione che viene utilizzato per inizializzare l'active directory, che viene poi ritornata.

```
function getLDAP() {

    if (!process.env.LDAP_URL || !process.env.LDAP_BASE_DN || !process.env.LDAP_USERNAME || !process.env.LDAP_PASSWORD) {
        throw new Error("Fatal Error: Missing LDAP data");
    }

    if (!process.env.GROUP_BASE_PROFILE || !process.env.GROUP_VIEWER_PROFILE || !process.env.GROUP_ADMIN_PROFILE)
        throw new Error('Fatal Error: Missing groups data')

    var config = {
        url: process.env.LDAP_URL,
        baseDN: process.env.LDAP_BASE_DN,
        username: process.env.LDAP_USERNAME,
        password: process.env.LDAP_PASSWORD
    }

    const ad = new ActiveDirectory(config);

    ad.find('CN=utreadldap', (err) => {
        if (err) {
            throw new Error('Fatal Error: Active directory binding failed.');
        }
    });

    return ad;
}
```

Figura 5.2: funzione che tenta la connessione al database aziendale

Come in precedenza accennato, per poter gestire facilmente sia i dati per connettersi al database aziendale, sia la mappatura tra gruppo aziendale e tipologia utente, viene fatto utilizzo di diverse variabili d'ambiente dichiarate direttamente nel file *docker-compose.yml*, come si può vedere in figura 3.2; in particolare si può individuare la seguente mappatura:

* Gli utenti che hanno tra i gruppi aziendali il gruppo *OM_DELIVERYONE_DELIVERY_MANAGER* saranno utenti *Admin*.
* Gli utenti che hanno tra i gruppi aziendali il gruppo *OM_DELIVERYONE_PM* saranno utenti *Viewer*.
* Gli utenti che hanno tra i gruppi aziendali il gruppo *OM_DELIVERYONE_SVILUPPATORE* saranno utenti *Base*.

Come da indicazioni ricevute, i gruppi sono mutualmente esclusivi, e non dovrebbe quindi essereci nessun dipendente che appartiene a più di un gruppo.

JSONWebToken

Quando un utente effettua l'accesso, un JSONWebToken, o JWT, viene generato ed inviato al messaggio di risposta, in modo che l'utente lo utilizzi per potersi autenticare ed interagire con i vari servizi a cui ha diritto.
Ogni richiesta indirizzata ad un endpoint diverso dalle Public Routes infatti richiede l'aggiunta di un *Authorization Header* che contanga il valore **Bearer <token>**, dove **<token>** fa riferimento ad un JWT valido. Le unice eccezioni sono tutte le richieste di tipo **OPTIONS**, dove la risposta è inviata immediatamente dopo aver impostato i *CORS headers*, in modo che le richieste di *pre-flight* vengano gestite correttamente. In sintesi, questa richiesta è necessaria per verificare che poi la vera richiesta venga permessa dal *server*. Il controllo dell'*header* appena descritto viene effettuato dalla funzione *isAuth*, la cui implementazione si può osservare in figura 5.3. Innanzitutto si controlla che effettivamente ci sia un *Authorization Header*, e che questo sia strutturato correttamente. A questo punto viene verificata la validità del token, e se risulta valido i dati dell'utente verificati vengono inseriti nella richiesta, per essere poi acceduti dalle funzioni successive.

```
interface ReqToken {
    userId: string,
    username: string,
    profile: "Base" | "Viewer" | "Admin";
}

function verifyDecodedToken(data: unknown): asserts data is ReqToken {
    if (!(data instanceof Object))
        throw new Error('Decoded token error. Token must be an object');
    if (!('userId' in data))
        throw new Error('Decoded token error. Missing required field "userId"');
    if (!('username' in data))
        throw new Error('Decoded token error. Missing required field "username"');
    if (!('profile' in data))
        throw new Error('Decoded token error. Missing required field "profile"');
}

export function isAuthN(req: express.Request, res: express.Response, next: express.NextFunction) {

    const authHeader = req.get('Authorization');

    if (!authHeader) {
        throw new StatusError('No Authorizatin header found.', 401);
    }

    const token = authHeader.split(' ')[1]; // Authorization: 'Bearer <token>'
    let decodedToken: unknown;
    try {
        decodedToken = jwt.verify(token, process.env.JWT_KEY);
    } catch (err) {
        throw new StatusError('Token validation failed', 403, { authHeader })
    }

    verifyDecodedToken(decodedToken);


    if (!decodedToken) {
        throw new StatusError('Not authenticated.', 401);
    }

    req.userData = {
        userId: decodedToken.userId,
        username: decodedToken.username,
        profile: decodedToken.profile
    };

    next();
}
```

Figura 5.3: Parte del codice sviluppato che verifica l'autorizzazione tramite JWT

Query e Filtri

Un semplice sistema di ricerca è stato implementato. Aggiungendo dei parametri di *query* alle richieste indirizzate a **GET /users** si può effettuare una ricerca degli utenti in base ai primi caratteri del nome, cognome o username.
Ad esempio, per ricercare tutti gli utenti con nome che inizi con **Jhon** e cognome che inizi con **Co** la richiesta sarebbe **GET /users?name=jhon&surname=co**.
Inoltre è possibile effettuare ricerche di team per progetto, creatore del team o per membri presenti nel team. Il nome del creatore viene espresso come un *JSON object literal* e può avere come parametri nome, cognome, username, o una combinazione di questi. Per specificare i membri del team invece bisogna utilizzare un array di *JSON object literal*.
Ad esempio la richiesta**GET teams?project=Happy&creator="name":"Jhon" &members[]="username":"Sara"&members[]="name":"Sue"** presenterebbe tutti i team con nome del progetto che inizia con *Happy*, l'utente creatore del team sia di nome *Jhon* e abbia come membri almeno due utenti, di nome *Sara* e *Sue*. Tutte le

ricerche sono *case insensitive*.

Errori

Gli errori vengono gestiti tramite una classe *StatusError* che estende la classe *Error* di javascript, aggiungendo due campi opzionali:

* **statusCode: number**: per segnalare il codice HTTP della risposta
* **data: any**: che contiene informazioni sull'errore che si è verificato.

Un esempio del suo utilizzo può essere osservato nella figura 5.4, dove in caso l'utente non sia di tipo *Admin* viene lanciato uno *StatusError*.
Si può notare che l'errore avrà come *statusCode* il valore 403, che corrisponde a *Forbidden*, e che indicherà il tipo del profilo tramite il parametro *profile* del campo *data*.

```
export function isAdmin(req: express.Request, res: express.Response, next: express.NextFunction) {

    if (req.userData.profile !== "Admin")
        throw new StatusError("You don't have the required permission", 403, { profile: req.userData.profile })

    next();
}
```

Figura 5.4: Funzione che controlla che il tipo dell'utente si *Admin* e, in caso contrario, lancia uno *StatusError*

.

5.2.1 Prodotto ottenuto

Alla fine quindi si è ottenuto un server che espone un *REST API* capace di ricevere e gestire richieste indirizzate ai vari endpoints esposti, ed inviare delle risposte. Di seguito viene riportata qualche immagine contenente delle possibili richieste che possono essere ricevute, e le relative risposte. Tutte le richieste effettuate sono state inviate tramite Insomnia.

POST signup

La figura 5.5 mostra un esempio di richiesta di registrazione presso il sistema da parte di un dipendente. In particolare la richiesta è indirizzata all'endpoint **POST /signup**, e richiede di aggiungere i propri dati aziendale (che qui fanno riferimento ad un utente per il testing).

Sulla destra si può vedere la risposta ricevuta, un oggetto con tre parametri:

1. **message**: Un messaggio testuale di risposta, in questo caso **signup successful**.
2. **user**: Un oggetto che contiene lo *user* appena registrato e aggiunto al database.
3. **token**: Il JWT generato che viene utilizzato per l'autenticazione dell'utente, valido per questa sessione.

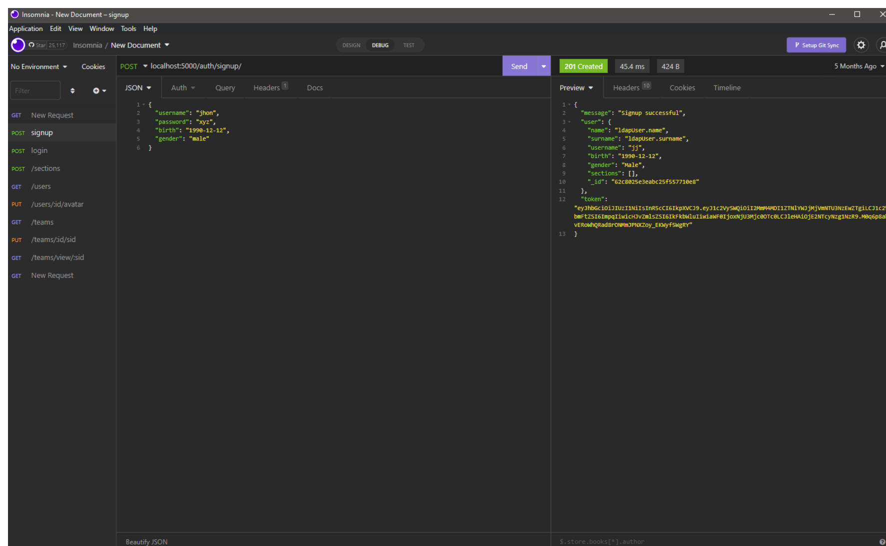


Figura 5.5: Esempio di richiesta a **POST /signup**

POST section

La figura 5.6 mostra un esempio di richiesta di aggiunta di un template di sezione. In particolare la richiesta è indirizzata all'endpoint **POST /sections**, e richiede di aggiungere una sezione chiamata **Languages**, di tipo **Standard**, con l'unica chiave condivisa con nome **Competence**, obbligatoria. Sono poi riportati i vari *subjectTemplates*, in particolare:

1. Il subject **English**, con id 1, che avrà come unica chiave la chiave condivisa **Competence**.
2. Il subject **French**, con id 2, che avrà come chiavi **Favourite French City**, obbligatoria, e la chiave condivisa **Competence**.
3. Il subject **German**, con id 3, che avrà come chiavi **Favourite German Philosopher**, opzionale, e la chiave condivisa **Competence**.
4. Il subject **Spanish**, con id 4, che avrà come unica chiave la chiave condivisa **Competence**.

Sulla destra si può vedere la risposta ricevuta, un oggetto con due parametri:

1. **message**: Un messaggio testuale di risposta, in questo caso **Section created successfully**.
2. **section**: Un oggetto che contiene la sezione appena creata ed aggiunta al database.

PUT /teams/:id/sid

La figura 5.7 mostra un esempio di richiesta di aggiornamento dello *ShareableId* di un team. In particolare la richiesta è indirizzata all'endpoint **PUT /teams/:id/sid**,

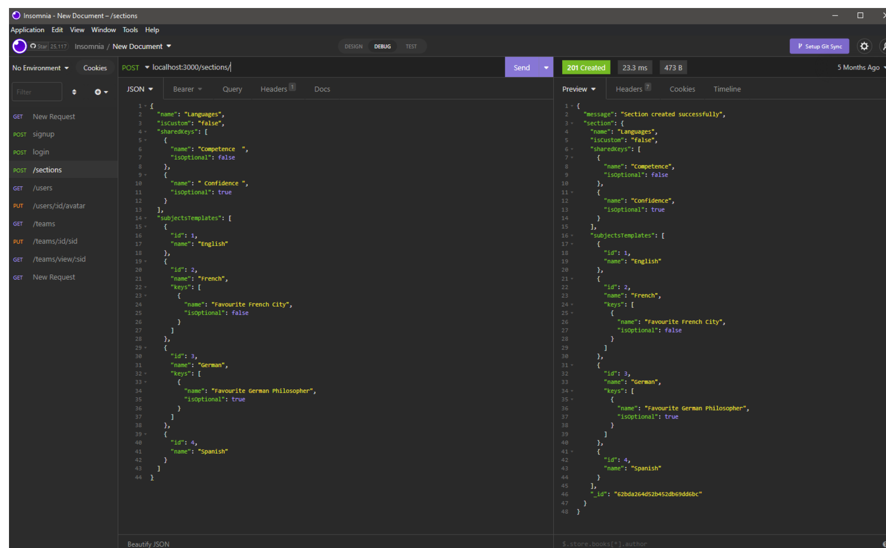


Figura 5.6: Esempio di richiesta a **POST /sections**

dove con **:id** si fa riferimento al team identificato dall'id **62bdac9724ff109c56ee0daf**, e si imostano i seguenti parametri:

1. **isPublic** viene impostato a **true**, quindi chiunque sia in possesso del sid sarà in grado di visualizzarlo (in particolare con una richiesta a **GET /view/:sid**).

2. **expiration** viene impostato a **12/12/2022**, quindi chiunque sia in possesso del sid sarà in grado di visualizzarlo se in data precedente a quella indicata.

Sulla destra si può vedere la risposta ricevuta, un oggetto con due parametri:

1. **message**: Un messaggio testuale di risposta, in questo caso **ShareableId updated successfully**.

2. **ShareableId**: Un oggetto che contiene lo *ShareableId* appena aggiornato.

Una volta ricevuta la risposta, tramite il parametro **sid** di **ShareableId** si può facilmente condividere il team, in quanto basterà effettuare una richiesta a **GET /view/62bdaccd24ff109c56ee0db0**.

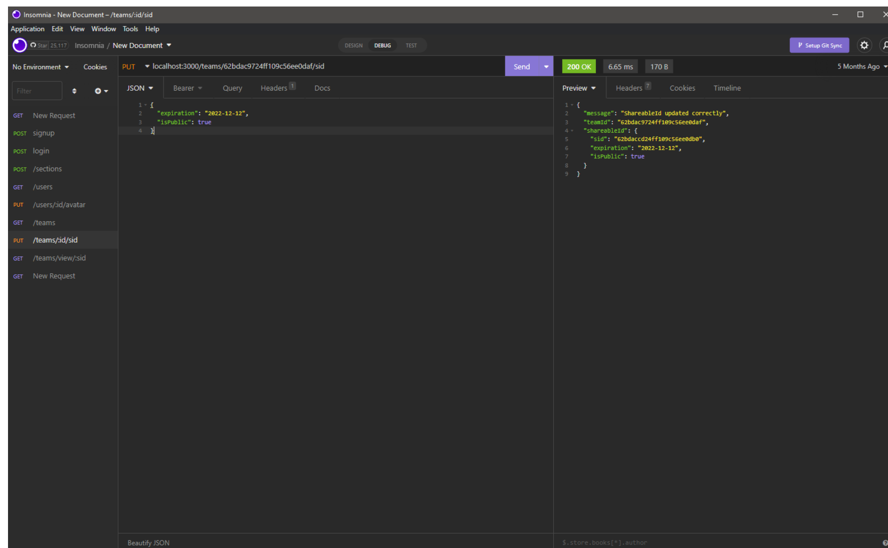


Figura 5.7: Esempio di richiesta a **PUT /teams/:id/sid**

Capitolo 6

Verifica e validazione

6.1 Test di unità

I test di unità non sono stati identificati dall'azienda come essenziali, e anzi sono stati considerati come requisiti desiderabili ma non obbligatori.
Ciononostante sono stati implementati dei test per verificare almeno le funzionalità più imoprtanti dell'applicazione.

6.1.1 Tecnologie utilizzate

I test di unità sono stati sviluppati utilizzando le tecnologie più utilizzate in ambito *Node.js*, *Mocha* e *Chai*. *Mocha* é un framework per effettuare test del codice sviluppato con *Node.js*, mentre Chai é una *assertion library* per *Node.js* che può essere facilmente utilizzata insieme a qualsiasi framework per testing di javascript.
Grazie a queste tecnologie risulta molto semplice organizzare, sviluppare ed eseguire test di unità per il codice che é stato prodotto.
Oltre a queste due tecnologie, dove necessario, é stato fatto utilizzo della libreria *Sinon* per facilitare la definizione di *stub* e *mock*, necessari in alcuni casi di test.
I test sviluppati sono stati organizzati in cartelle suddivise seguendo la divisione del codice sorgente, e sono facilmente eseguibili direttamente su container docker tramite il comando **docker-compose run app npm run test**, in quanto è stato definito all'interno del file **package.json** lo script per l'esecuzione dei test.

6.1.2 Test sul modello

* Test classe **Section**
  - **TU1**: il metodo *build* solleva un errore quando il parametro *subjectData* non ha riferimenti validi ad un *subjectTemplate* del parametro *sectionTemplate*.
  - **TU2**: il metodo *build* solleva un errore quando nel parametro *subjectData* è presente una *key* che non compare nel *subjectTemplate* del parametro *sectionTemplate* a cui fa riferimento.
  - **TU3**: il metodo *build* solleva un errore quando nel parametro *subjectData* non è presente il valore per una *key* con **isOptional===false**.
  - **TU4**: il metodo *build* costruisce correttamente una *Section* dati parametri di input validi.

* Test classe **User**
  - **TU5**: il metodo di costruzione di *User* solleva un errore quando *sectionData* non fa riferimento ad un *sectionTemplate* nel database.
  - **TU6**: il metodo di costruzione di *User* ha successo quando *sectionData* contiene dati validi.
  - **TU7**: il metodo *getById* solleva un errore quando viene chiamato con parametro *id* non presente nel database (figura 6.1).
  - **TU8**: il metodo *getById* ritorna l'utente corretto quando viene chiamato con con parametro *id* presente nel database (figura 6.1).
  - **TU9**: il metodo *queryById* ritorna il valore *null* quando viene chiamato con parametro *id* non presente nel database (figura 6.1).
  - **TU10**: il metodo *queryById* ritorna l'utente corretto quando viene chiamato con con parametro *id* presente nel database (figura 6.1).
* Test classe **Team**
  - **TU11**: il metodo *getBySID* solleva un errore quando viene chiamato con parametro *sid* non valido.
  - **TU12**: il metodo *getBySID* solleva un errore quando viene chiamato con parametro *sid* che fa riferimento ad uno *ShareableId* con **isPublic===false**.
  - **TU13**: il metodo *getBySID* solleva un errore quando viene chiamato con parametro *sid* che fa riferimento ad uno *ShareableId* con *expiration* scaduta.
  - **TU14**: il metodo *getBySID* ritorna correttamente il team a cui fa riferimento quando il parametro *sid* fa riferimento ad uno *ShareableId* valido, pubblico e non scaduto.

6.1.3 Test dei middleware

* Test middleware **isAuthN**
  - **TU15**: viene sollevato un errore quando la richiesta non presenta un *Authorization Header*.
  - **TU16**: viene sollevato un errore quando la richiesta presenta un *Authorization Header* con formattazione scorretta.
  - **TU17**: viene sollevato un errore quando la richiesta presenta un *Authorization Header* con un JWT che fallisce la validazione.
  - **TU18**: aggiunge correttamente un oggetto *userData* con i dati corretti alla richiesta ricevuta, dopo aver validato e decodificato correttamente il JWT.

```
describe('Finder Methods', () => {

    const userId = new ObjectId();
    collections.users = {
        findOne: async ({ _id }) => {
            if (userId.equals(_id))
                return { name, surname, username, birth, gender, sections: [], _id: userId, avatar: null }
        }
    }

    describe('User.getById Method', () => {

        it('should throw an error when no user is found (TU7)', () => {
            return expect(User.getById(new ObjectId())).to.eventually.be.rejectedWith(/No user found/i);
        });

        it('should return a user object when user is found (TU8)', () => {
            return expect(User.getById(userId)).to.eventually.have.property('username');
        });
    });


    describe('User.queryById Function', () => {

        it('should return null when no user is found (TU9)', () => {

            return expect(User.queryById({ id: new ObjectId() })).to.eventually.equal(null);
        });

        it('should return a user object when user is found (TU10)', () => {
            return expect(User.getById(userId)).to.eventually.have.property('username');
        });
    });

});
```

Figura 6.1: Definizione ed implementazione dei test per i metodi *getById* e *queryById* della classe **User**, che corrispondono ai test **TU7**, **TU8**, **TU9** e **TU10**

6.2 Validazione

Terminati i lavori si è tenuto un collaudo con il tutor aziendale, che ha verificato che tutti i requisiti obbligatori fossero stati raggiunti. In questa occasione si è inoltre discusso con il tutor aziendale di come alcune soluzioni trovate potrebbero essere state diversamente implementate, e di altre idee maturate durante i lavori. Tra le soluzioni alternative pensate, ci si sarebbe concentrati principalmente sul semplificare la logica delle sezioni, in modo da renderle più facilmente gestibili. Si sarebbero inoltre implementati più endpoint per gli utenti, in modo da gestire le sezioni di un utente più granularmente, e non andare ogni volta a intoccarle tutte ad ogni singola modifica. Alternativamente si sarebbe potuto strutturare il database in maniera più efficiente, facendo più attenzione a come salvare i dati. Ad esempio le sezioni dei vari utenti si sarebbero potute salvare in una *collection* a parte, e poi queste sarebbero state referenziate dagli utenti tratmite il loro id.

Capitolo 7

Conclusioni

7.1 Raggiungimento degli obiettivi

Tutti i requisiti obbligatori sono stati soddisfatti, e in aggiunta anche alcuni requisiti desiderevoli; in particolare sono stati soddisfatti i seguenti requisiti desiderevoli:

* **FD01:** Implementazione di test di unità.
* **FD02:** Possibilità di ricerca profili e team tramite semplici filtri.
* **FD05:** Implementazione codice per la visualizzazione degli avatar 3D con Threejs.
* **FD06:** Organizzazione dei dockerfiles tramite l'utilizzo di Makefiles.

7.2 Conoscenze acquisite

Grazie all'esperienza avuta ho potuto imparare e sperimentare con molte tecnologie e concetti nuovi, che mi hanno permesso di ampliare vastamente le mie conoscenze e le mie abilità pratiche.
Ho imparato cosa siano, come si utilizzino e come si predispongano i container Docker, come progettare ed implementare un back-end, utilizzando *Node.js* ed *Express*, in particolare utilizzando Typescript, come si gestisce un database non relazionale come MongoDB.

7.3 Valutazione personale

Tutto considerato mi ritengo soddisfatto dell'esperienza avuta. In particolare ho apprezzato il dover affrontare tecnologie e strumenti completamente sconosciuti, arrivare a comprenderli e sapreli utilizzare ed applicare.
Ho apprezzato la flessibilità del tutor aziendale, che non ha avuto problemi nel riadattare lo stage dando importanza agli aspetti che personalmente mi interessavano di più.

Come aspetti negativi considero il fatto di aver dovuto lavorare ad un progetto da solo, non avendo quindi avuto la possibilità di sperimentare il lavorare in un gruppo di lavoro, con tutti i benefici e gli svantaggi che porterebbe, in quanto il lavorare da solo in certi momenti è sembrato quasi più un *assignment* universitario piuttosto che un progetto aziendale.

Glossario

API in informatica con il termine *Application Programming Interface API* (interfaccia di programmazione di un'applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 28, 43

back-end in informatica con il termine *back-end* si indica l'insieme delle applicazioni e dei programmi informatici con cui l'utente non interagisce direttamente, e che è spesso responsabile dell'utilizzo e della memorizzazione dei dati.. iii, ix, 6, 7, 9, 10, 12, 13, 15, 22, 27, 41, 43

endpoint in informatica con il termine *endpoint* ci si riferisce ad un indirizzo digitale dove un server riceve richieste, le processa ed invia delle risposte. Solitamente si tratta di un *URL*.. 9, 13, 23, 27–31, 33, 34, 39, 43

framework in informatica con il termine *framework* si indica un'architettura logica di supporto sulla quale un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore.. 6, 10, 13, 37, 43

front-end in informatica con il termine *front-end* si indica l'insieme delle applicazioni e dei programmi informatici con cui l'utente interagisce direttamente.. 6, 15, 27, 43

Bibliografia

Siti web consultati

Documentazione Docker. URL: https://docs.docker.com/.

Documentazione Ready Player Me. URL: https://docs.readyplayer.me/ready-player-me/.

Express. URL: https://expressjs.com/.

JSONWebToken. URL: https://jwt.io/.

MongoDB. URL: https://www.mongodb.com/.

Node.js. URL: https://nodejs.org/en/.

Red Hat. URL: https://www.redhat.com/en/topics.

Riferimenti git. URL: https://www.atlassian.com/git/tutorials.

Riferimenti REST API. URL: https://restfulapi.net.

Typescript. URL: https://www.typescriptlang.org/.

Wikipedia. URL: https://www.wikipedia.org/.