

DAIS Internship Manager

Piattaforma per la gestione dei tirocini universitari



Giacomo De Liberali

Relatore: Filippo Bergamasco

**Dipartimento di Scienze Ambientali, Informatica e Statistica
Università Ca' Foscari Venezia**

Laurea in Informatica

Giugno 2018

Abstract

This is where you write your abstract ...

Indice

Elenco delle figure	vii
Elenco delle tabelle	ix
1 Introduzione	1
1.1 Da dove nasce questo progetto	1
1.2 Scelte e vincoli tecnici	2
1.3 Tecnologie adottate	2
1.3.1 <i>MongoDB</i>	2
1.3.2 <i>Node.js</i>	4
1.3.3 <i>Express.js</i>	5
1.3.4 <i>Angular</i>	6
1.4 Attori del sistema	9
1.4.1 Azienda	9
1.4.2 Professore	9
1.4.3 Studente	9
1.4.4 Admin	10
2 Architettura	11
2.1 Architettura lato server	11
2.1.1 Schemas	12
2.1.2 Repositories	13
2.1.3 Controllers	14
2.1.4 Bootstrap	15
2.2 Architettura lato client	16
2.2.1 Moduli e divisione delle responsabilità	16
2.2.2 Servizi e recupero dei dati	17
2.2.3 Supporto multi lingua	17

3	Casi d'uso e workflow	19
3.1	Casi d'uso	19
3.1.1	Registrazione e login di un utente	19
3.1.2	Creazione di un tirocinio	20
3.1.3	Modifica di un tirocinio	20
3.1.4	Approvazione di un tirocinio	20
3.1.5	Candidatura ad un tirocinio	20
3.1.6	Approvazione candidatura (professore)	21
3.1.7	Approvazione candidatura (azienda)	21
3.1.8	Avvio di un tirocinio	21
3.1.9	Compilazione foglio presenze	21
3.1.10	Generazione documentazione	21
4	Application Programming Interface	23
4.1	Autenticazione	23
4.1.1	Server side	23
4.1.2	Client side	23
4.2	Endpoints	23
4.2.1	Controller base	23
4.2.2	Internships	24
4.2.3	InternshipProposals	24
4.2.4	Roles	24
4.2.5	Users	24
4.2.6	Companies	24
4.2.7	Auth	24
5	Esempi e screenshot	25
5.1	Screenshot workflow	25
6	Conclusioni	27
6.1	Sviluppi futuri e nuove integrazioni	27
	Bibliografia	29
	Termini ed abbreviazioni	31
	Glossario	33

Elenco delle figure

1.1	Esempio di sintassi di <i>MongoDB</i>	4
1.2	Esempio di applicazione <i>Express.js</i>	6
1.3	Architettura di Angular	8
2.1	Esempio di <i>schema</i> dell'applicazione back-end	12
2.2	Esempio di <i>Repository</i>	13
2.3	Esempio di registrazione di un metodo <i>Express.js</i> in un <i>Controller</i>	14
2.4	Estratto di <i>Server.ts</i> , bootstrap del back-end	15
2.5	Esempio di <i>service</i> front-end	17
2.6	Estratto del file di globalizzazione front-end	18
2.7	Localizzare un <i>component</i> con <i>ngx-translate</i>	18
3.1	Flusso di login e registrazione di un utente	20

Elenco delle tabelle

1.1	Confronto modello query e indicizzazione di <i>MongoDB</i> e altri database ^[4]	4
4.1	Endpoint rest API	23

Capitolo 1

Introduzione

1.1 Da dove nasce questo progetto

Nella carriera universitaria di uno studente è prevista dal piano di studi l'inclusione di un tirocinio (internship) che permetterà allo studente di collaborare con un'azienda in un progetto che sia al di fuori delle mura dell'ateneo. Vi sono due diversi tipi di tirocini, curriculare ed extra-curriculare; il primo permette il riconoscimento di crediti formativi universitari (CFU), mentre il secondo mira solamente a fornire un'esperienza lavorativa allo studente.

Per avviare un tirocinio bisogna quindi mettere in comunicazione soggetti eterogenei, ovvero aziende, professori e studenti. Permettere un'efficace collaborazione tra questi attori che appartengono a categorie e ambienti diversi non è semplice e necessita di un controllo granulare e centralizzato.

Il progetto *DAIS Internship Manager* nasce proprio con l'intento di semplificare il processo di gestione degli stage universitari. Il sistema correntemente adottato dall'ateneo non permette un'efficace fruizione dei contenuti né da parte degli studenti né tanto meno dal punto di vista dei professori e delle aziende. L'intero sistema è una semplice interfaccia web che mostra agli studenti autenticati tutte le offerte pubblicate.

Il workflow da seguire per inserire, cercare e candidarsi ad un'offerta di tirocinio è piuttosto macchinoso. Se un'azienda desidera proporre un'offerta di tirocinio prima di tutto deve essere convenzionata con l'ateneo, dopodiché deve inviare un'email alla segreteria che provvederà, una volta validato il contenuto dell'offerta, alla pubblicazione della stessa. Una volta pubblicata l'offerta sarà visibile dagli studenti che potranno candidarsi contattando prima il professore e in seguito l'azienda, sempre mediante un rapporto basato su email.

Risulta quindi necessaria una soluzione che permetta di automatizzare il più possibile questo processo, che tenga traccia dell'andamento del tirocinio e ne monitori lo stato.

1.2 Scelte e vincoli tecnici

La soluzione deve essere fruibile da quanti più dispositivi possibili e per raggiungere questo obiettivo ho scelto di sviluppare un'applicazione web. Appoggiandosi infatti all'accessibilità offerta dal web sarà sufficiente mantenere un solo codebase per raggiungere tutti i dispositivi — computers, smartphones e tablets.

Un requisito di fondamentale importanza è quindi la responsività dell'applicazione, data la diversità dei dispositivi che si intende supportare. Inoltre, per favorire l'accessibilità dell'applicazione, essa dovrà essere multi lingua e in questa prima versione dovrà supportare almeno l'italiano e l'inglese.

1.3 Tecnologie adottate

Dal momento che ho deciso di puntare su un applicazione web, le tecnologie che andrò ad utilizzare per il *front-end* della soluzione saranno sicuramente web-based, in particolare lo stack *MongoDB*, *Express*, *Angular* & *Node.js* (*MEAN*). Questo insieme applicativo è composto da

1. Un DBMS basato su un database documentale NoSQL (*MongoDB*)
2. Un framework server-side per la creazione di applicazione web e rest *API* (*Express.js*)
3. Un framework client-side (*Angular*)

1.3.1 *MongoDB*

MongoDB è un Database Management System (DBMS) non relazionale, orientato ai documenti classificato come No Structured Query Language (NoSQL). Questo significa che rispetto ai tradizionali sistemi relazionali *MongoDB* si basa sul concetto di documento e collezione.

Il documento rappresenta un oggetto che si intende memorizzare, mentre una collezione è un insieme di documenti (una tabella se paragonata ai sistemi relazionali). *MongoDB* memorizza i dati in con una rappresentazione binaria chiamata BSON (Binary JSON). Questa codifica estende la popolare rappresentazione Javascript Object Notation (JSON) per includere tipi di dato aggiuntivi, come *int*, *long*, *date*, *floating point* e *decimal 128*. I documenti BSON possono contenere uno o più campi, ognuno dei quali contiene il valore di uno specifico tipo di dato, inclusi array, dati binari o sotto-documenti.

Un esempio di paragone JSON-BSON è il seguente:

```
1 // JSON = {"hello":"world"}
2
3 BSON:
4   \x16\x00\x00\x00           // dimensione totale documento
5   \x02                       // 0x02 = tipo String
6   hello\x00                   // nome campo. \x00 = terminatore
7   \x06\x00\x00\x00world\x00   // valore campo
8                               // (dimensione valore, valore,
8                               //      terminatore)
9   \x00                       // 0x00 = tipo EOO ('end of object')
```

Rispetto a JSON, BSON è progettato per essere efficiente sia nello spazio di archiviazione che nella velocità di scansione. Gli elementi in un documento BSON sono preceduti da un campo lunghezza per facilitare la scansione e questo in alcuni casi, quando il documento è piccolo, porterà BSON ad utilizzare più spazio di JSON proprio a causa dei prefissi di lunghezza e degli indici di array espliciti.

I documenti BSON di *MongoDB* sono concettualmente allineati alla struttura di un oggetto nei linguaggi di programmazione OOP. Questo rende più semplice e veloce per gli sviluppatori modellare la struttura dati dell'applicazione. Tendono infatti ad raggruppare tutti i dati di un record in un unico documento, in opposizione al sistema relazionale tradizionale in cui le informazioni sarebbero distribuite su diverse tabelle. Questa sorta di aggregazione del dato riduce drasticamente il bisogno di operazioni di JOIN su tabelle diverse, ottenendo performance superiori grazie ad una singola lettura per ottenere l'intero documento desiderato.

I documenti *MongoDB* possono variare nella struttura. Ad esempio, tutti i documenti che descrivono i clienti potrebbero contenere l'ID cliente e la data in cui hanno acquistato i nostri prodotti o servizi, ma solo alcuni potrebbero contenere il collegamento ai social media dell'utente o i dati sulla posizione dalla nostra applicazione mobile. I campi possono variare da un documento all'altro; non è necessario dichiarare la struttura dei documenti al sistema — essi sono auto-descrittivi. Se è necessario aggiungere un nuovo campo a un documento, è possibile crearlo senza influire sugli altri documenti nel sistema.

Le collezioni sono un insieme di documenti per definizione schema-less, ovvero contengono documenti con tipologie eventualmente diverse (non è considerata una best-practise). Rapportate al sistema relazionale rappresentano una tabella.

MongoDB, essendo basato sul modello NoSQL non utilizza il linguaggio di interrogazione proprio dei sistemi relazionali, ma ne propone uno proprio. *MongoDB* presenta alcune

```

1 // SQL
2 INSERT INTO users          SELECT * FROM users
3   (name, age, gender)      WHERE
4 VALUES                    age < 25
5   ('Jack', 22, 'M')
6
7 // NoSQL
8 db.users.insert({          db.users.find(
9   name: 'Jack',             age: {
10  age: 22,                   $lt: 25 // $lt = less than
11  gender: 'M'                }
12 })                          })

```

Figura 1.1 Esempio di sintassi di *MongoDB* vs SQL

funzionalità non previste dalle altre architetture di database. Una comparazione delle features più interessanti tra *MongoDB* e altri tipi di database è raffigurata nella tabella sottostante.

Tabella 1.1 Confronto modello query e indicizzazione di *MongoDB* e altri database^[4]

	<i>MongoDB</i>	Database relazionale	Database Key-value
Query Key-value	Si	Si	Si
Indici secondari	Si	Si	No
Intersezione indici	Si	Si	No
Range queries	Si	Si	No
Query geospaziali	Si	Aggiunta costosa	No
Faceted Search	Si	No	No
Aggregazione e trasformazione	Si	Si	No
Equi e Nonequi JOIN	Si	Si	No
Graph processing	Si	No	Si

1.3.2 *Node.js*

Node.js è un ambiente open source e cross platform sviluppato a partire dal 2009 che permette di eseguire codice javascript lato server.

«Node.js® è un runtime javascript costruito sul motore javascript V8 di Chrome. Node.js usa un modello I/O non bloccante e ad eventi, che lo rende un framework leggero ed efficiente. L'ecosistema dei pacchetti di Node.js, npm, è il più grande ecosistema di librerie open source al mondo.»^[1]

Storicamente javascript era utilizzato solamente per scripting client-side, spesso incluso all'interno delle pagine web dove veniva eseguito client-side nel browser dell'utente. *No-*

de.js permette agli sviluppatori di utilizzare scripting server-side, eseguendo comandi che producono contenuto dinamico prima che venga inviato al browser client-side. *Node.js* rappresenta il paradigma «javascript everywhere»^[2], unificando lo sviluppo di applicazioni web attorno ad un unico linguaggio di programmazione piuttosto che separando linguaggi per client e server-side.

La sua architettura è basata sul modello orientato agli eventi (EDA), ciò significa che *Node.js* richiede al sistema operativo su cui è in esecuzione di ricevere notifiche al verificarsi di determinati eventi, rimanendo in stato di *sleep* fino al ricevimento di tale notifica. Questo pattern architetturale permette una forma di comunicazione non bloccante basata sull'*asynchronous I/O* che per il programmatore finale si traduce nell'utilizzo di *callback*. Per segnalare la conclusione di un *task I/O* infatti, *Node.js* invoca la corrispondente *callback*, una semplice funzione, alla quale vengono passati i risultati dell'operazione appena conclusa. *Node.js* opera in un processo single-thread utilizzando il pattern *Observer* per la sottoscrizione e la gestione degli eventi, ottenendo così performance adatte ad applicazioni altamente realtime. Processa le richieste in arrivo in un ciclo, chiamato *event-loop*, dove ogni connessione è una piccola allocazione di memoria heap anziché un nuovo processo o thread. Alla fine della registrazione della *callback* il server rientra in modo automatico nell'*event-loop*, a differenza di altri server orientati agli eventi e vi esce solo quando non vi sono ulteriori *callback* da eseguire.

I vantaggi che hanno portato *Node.js* ad avere una diffusione così ampia sono molti. Sicuramente possiamo notare che javascript è un linguaggio ben conosciuto e largamente utilizzato, quindi la curva di apprendimento di questa tecnologia è molto più breve; offrendo inoltre la programmazione orientata agli eventi permette agli sviluppatori di creare server in grado di gestire un alto numero di richieste simultanee che siano facilmente scalabili senza l'utilizzo del *threading*. Lo svantaggio principale di *Node.js* è la mancanza al supporto per la scalabilità verticale, determinata dalla sua architettura single-thread.

1.3.3 *Express.js*

Express.js è un *framework* per costruire applicazioni web ed *API* basato sulla piattaforma *Node.js*. Nel corso del tempo è divenuto lo standard de facto per i *framework* server di *Node.js*. Si presenta in modo minimale, offrendo un sottile livello applicativo che punti a velocizzare lo sviluppo senza tuttavia oscurare le funzionalità di *Node.js*. *Express.js* è diviso in diversi moduli che possono essere innestati uno spora l'altro, rendendolo adatto ad ogni tipo di applicazione. Le sue funzionalità principali sono:

- (a) Un sistema di routing: contenuto all'interno del pacchetto 'express-router', è il modulo

per la gestione e la manipolazioni delle routes. Permette di definire in modo gerarchico un insieme di *URL*, alle quali associare una specifica azione. Un'azione è un metodo che viene invocato e che produce una risposta.

- (b) *HTTP helpers*, come *redirect* o sistemi di caching: contenuto all'interno del modulo core, mette a disposizione alcune utility class che facilitano operazioni ripetitive oppure forniscono strumenti aggiuntivi utili ad ogni tipologia di sistema che si intende sviluppare
- (c) Supporto a diversi template engines. Dal momento che si possono anche realizzare applicazioni che ritornano del contenuto, ad esempio un applicazione *Model-View-Controller (MVC)*, è necessario un interprete del template che permetta l'inserimento dinamico di contenuto all'interno di esso. Un esempio di quelli che *Express.js* supporta out of the box sono *Pug (Jade)*, *Haml.js*, *React*, *Blade* e altri.

```
1 import { express } from 'express';
2
3 const app = express()
4
5 app.get('/', (req: Request, res: Response) => {
6   res.send('Hello World')
7 })
8
9 app.listen(3000)
```

Figura 1.2 Un esempio di applicazione *Express.js*

Come possiamo vedere in figura 1.2, una volta creata l'applicazione (riga 3), viene registrata una nuova route (riga 5) alla quale viene associata una *callback*. Questa *callback* riceve due parametri, la richiesta e la risposta. Il processo di *Node.js* resterà in stato di sleep fino a che una nuova richiesta verrà inoltrata nella route appena definita (quindi fino a che non verrà eseguita una chiamata in *HTTP GET* all'indirizzo dove è in esecuzione l'applicazione). Una volta ricevuta la notifica *Node.js* entrerà nell'*event-loop* per gestirla e una volta completata l'operazione eseguirà la *callback* registrata, rispondendo al client che ha effettuato la connessione con la stringa 'Hello World!'.

1.3.4 Angular

Angular è un framework sviluppato da Google per lo sviluppo front-end di applicazioni web basato su typescript. Implementa funzionalità core e opzionali in un insieme di librerie di

typescript che bisogna importare nelle applicazioni. In seguito verranno descritti gli elementi costitutivi di *Angular*.

NgModules

Dichiarano un contesto di compilazione per un insieme di componenti dedicato a un dominio dell'applicazione, un flusso di lavoro o un insieme di funzionalità correlate. Un *NgModule* associa i propri componenti al codice relativo, come i servizi, per formare unità funzionali. Ogni applicazione *Angular* — che in genere contiene più moduli funzionali — ha un modulo radice, denominato convenzionalmente *AppModule*, che fornisce il meccanismo di *bootstrap* che avvia l'applicazione.

Come i moduli javascript, anche gli *NgModules* possono importare funzionalità da altri *NgModules* e consentire che le proprie funzionalità vengano esportate e utilizzate da altri *NgModules*. L'organizzazione del codice in moduli funzionali distinti aiuta nella gestione dello sviluppo di applicazioni complesse e nella progettazione delle stesse, aumentando la riusabilità del codice. Inoltre, questa tecnica consente di sfruttare il *lazy loading*, ovvero il caricamento dei moduli su richiesta, al fine di ridurre al minimo la quantità di codice che deve essere caricata all'avvio.

Components

Ogni applicazione *Angular* ha almeno un componente, il *root component*, che connette una gerarchia di componenti con il DOM della pagina. Ogni componente è definito mediante una classe che contiene i dati e la logica dell'applicazione a cui è associato un *template* HTML che definisce come i dati devono venire visualizzati.

I *decorators* sono funzioni che modificano le classi javascript, in particolare il decoratore *@Component* identifica la classe immediatamente sottostante come un componente *Angular*, definendo template e metadati specifici del componente stesso (come le dipendenze).

Il *template* combina HTML con un markup di *Angular* che può modificare gli elementi HTML prima che vengano visualizzati.

Le *directives* forniscono logica al componente e il *binding markup* connette i dati dell'applicazione con il Document Object Model (DOM).

Gli *event binding* permettono all'applicazione di rispondere all'input dell'utente aggiornando i dati dell'applicazione, mentre i *property binding* permettono di interpolare valori calcolati dai dati dell'applicazione all'interno del template HTML.

Prima che un componente venga visualizzato, *Angular* valuta le direttive e risolve la sintassi di *binding* nel template per modificare gli elementi HTML e il DOM a seconda

dei dati dell'applicazione e della loro logica. *Angular* supporta il *two-way databinding*, che significa che cambiamenti nel DOM, come input dell'utente, possono venire riflessuti all'interno dei dati dell'applicazione, e viceversa.

Services e dependency injection

Per dati o logica non associati ad una specifica visualizzazione grafica che si desidera condividere tra componenti, si crea una *service class*. Una definizione di una classe di servizio è immediatamente preceduta dal decoratore *@Injectable*. Il decoratore fornisce i metadati che consentono al servizio di essere iniettato nei componenti client come dipendenza.

La Dependency injection (DI) consente di mantenere le classi dei componenti snelle ed efficienti delegando logiche di business ai servizi iniettati.

Routing

L'*NgModule Router* fornisce un servizio che consente di definire un percorso di navigazione tra i diversi componenti dell'applicazione e visualizzarne le gerarchie.

Il router mappa percorsi simili a URL per componenti anziché per pagine, ovvero quando un utente esegue un'azione, ad esempio facendo clic su un collegamento — che dovrebbe caricare una nuova pagina nel browser — il router intercetta il comportamento del browser e mostra o nasconde le gerarchie di componenti definite nel modello di routing.

Se il router determina che lo stato dell'applicazione corrente richiede funzionalità particolari e il modulo che le definisce non è stato caricato, il router può caricare il modulo su richiesta (*lazy loading*).

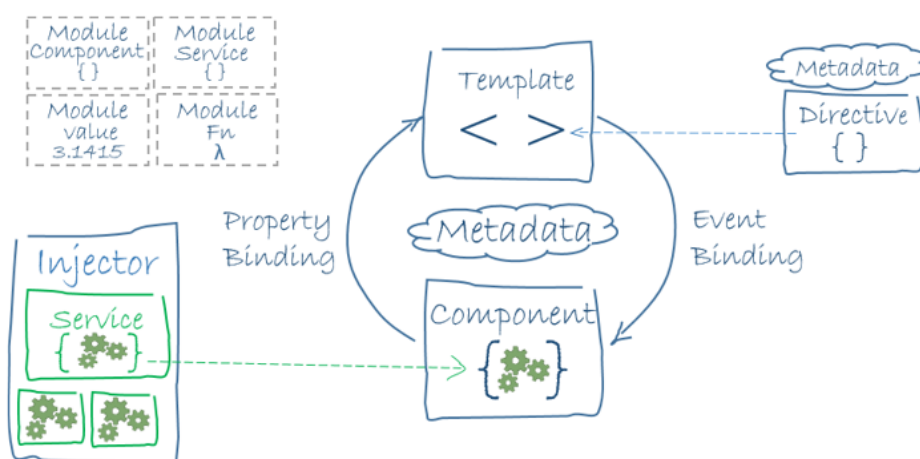


Figura 1.3 Schema dell'architettura di Angular^[3]

1.4 Attori del sistema

Dal momento che vi sono più soggetti diversi che accedono alla piattaforma è necessario definire i ruoli dei soggetti coinvolti. Il sistema prevede la gestione di quattro tipi di account — ognuno con permessi diversi — e la personalizzazione dell'interfaccia in base all'utente correntemente autenticato. Un account può avere anche più ruoli, ma al momento non esiste un'apposita interfaccia per la gestione; l'assegnazione di un ruolo multiplo deve avvenire tramite accesso diretto al database.

1.4.1 Azienda

Deve effettuare l'accesso come un membro esterno dell'ateneo indicando le informazioni della propria azienda e quelle dei suoi amministratori. Inserisce le offerte di tirocinio presso una delle proprie sedi, visualizza le candidature ricevute e approva l'inizio di uno stage. Può completare il foglio presenze di un tirocinio in corso (nella propria azienda) e stamparne la documentazione precompilata al termine. Un'azienda ha uno o più amministratori che ne gestiscono le offerte di tirocinio e le candidature.

1.4.2 Professore

Deve effettuare l'accesso come un membro dell'ateneo utilizzando l'email istituzionale. Una volta eseguito il login per la prima volta viene creato un account con il questo ruolo basandosi sull'email fornita: se essa termina con *@unive.it* rappresenta un professore, mentre se termina con *@stud.unive.it* rappresenta uno studente. Un professore approva le offerte inserite dalle aziende prima che vengano pubblicate, approva e visualizza le richieste di candidatura degli studenti che li coinvolgono come referenti. Può completare il foglio presenze di un tirocinio (do cui è referente) in corso e stamparne la documentazione precompilata al termine.

1.4.3 Studente

Deve effettuare l'accesso come un membro dell'ateneo utilizzando l'email istituzionale. Visualizza le offerte di tirocinio pubblicate e propone una candidatura indicando un professore come referente. Può completare il foglio presenze di un suo tirocinio in corso e stamparne la documentazione precompilata al termine.

1.4.4 Admin

Ricopre il ruolo di amministratore dei dati ed è l'unico soggetto a non avere restrizioni sulla modifica o la cancellazione dei dati. Non ha un'influenza sul processo di gestione dei tirocini in quanto gli altri account formano un ecosistema che si alimenta e gestisce in modo autonomo.

Capitolo 2

Architettura

Questo capitolo illustra il funzionamento e la struttura delle architetture software, sia per quanto riguarda il front-end che il back-end. Dal momento che entrambe le applicazioni *Angular* e *Node.js* operano sugli stessi oggetti, le loro definizioni risiedono in un pacchetto Node Package Manager (npm) condiviso che esporta tutte le entità necessarie, in modo da riutilizzare il codice e renderlo facilmente manutenibile. Le principali entità esportate sono:

- *User*: rappresenta un utente del sistema con uno specifico ruolo
- *Company*: rappresenta un'azienda
- *Internship*: rappresenta un'offerta di tirocinio di un'azienda
- *InternshipProposal*: rappresenta una proposta di candidatura di uno studente per un tirocinio

2.1 Architettura lato server

Il back-end di *DAIS Internship Manager* è un'applicazione *Node.js* che si appoggia sul framework *Express.js* per l'esposizione di REST API che interagiscono con *MongoDB* attraverso l'ORM *Mongoose.js*.

L'infrastruttura server è divisa in diversi livelli con responsabilità diverse che interagiscono fra loro che verranno discusse nei paragrafi seguenti.

2.1.1 Schemas

Pur utilizzando *MongoDB* che è uno schema-less DBMS ho preferito appoggiarmi su un sistema che mi permettesse di validare i dati e la loro struttura. Per fare ciò ho utilizzato *Mongoose.js*, un Object-relational mapping (ORM) che mi permette di definire la struttura dei documenti nelle collezioni del database, mi fornisce metodi di validazione e manipolazione basati su oggetti.

Gli *schemas* sono la definizione dei documenti delle collezioni del database come previsti da *Mongoose.js*. Essi contengono la definizione del documento, dei suoi campi e del loro tipo. Possiamo notare che in figura 2.1, all'interno della definizione della struttura del documento,

```
1  /** The [[InternshipProposal]] mongoose schema model */
2  export const InternshipProposalSchema: Schema = new Schema({
3    attendances: [
4      {
5        date: {
6          type: Schema.Types.Date,
7          required: true
8        },
9        ...
10     }
11   ],
12   internship: {
13     type: Schema.Types.ObjectId,
14     ref: 'Internship',
15     autopopulate: true
16   },
17   status: Schema.Types.Number,
18   ...
19 });
```

Figura 2.1 Esempio di *schema* dell'applicazione back-end

vi sono tre proprietà — *attendances*, *internship* e *status*. Ognuno dei campi ha tipo diverso:

- *attendances* è un array di oggetti con una proprietà *date* di tipo *Date* obbligatoria
- *internship* è una referenza di un altro schema (*Internship*), che verrà popolato automaticamente in fase di lettura (effettua un JOIN in automatico con il plugin npm 'mongoose-autopopulate')
- *status* è semplice numero

Nel caso l'applicazione cerchi di salvare un oggetto che non rispetti i vincoli imposti dallo *schema* viene sollevata un'eccezione che impedisce di rendere inconsistente il database.

2.1.2 Repositories

I *repositories* sono classi legate ad uno specifico oggetto che esportano operazioni su di esso. Interrogano uno o più *schemas* per leggere, scrivere o aggregare dati, e contengono solamente la logica di accesso ai dati, senza nessuna logica di business (ad esempio il controllo dei permessi). Tutti i *repositories* derivano dal *BaseRepository* che esporta le operazioni di base — Create, Read, Update e Delete (CRUD) — oltre che un metodo per eseguire query personalizzate. Ogni *repository* può esporre ulteriori metodi personalizzati ed eventualmente

```

1  /**
2   * The [[InternshipsProposalsRepository]] repository
3   */
4  @injectable()
5  export class InternshipsProposalsRepository extends
6      BaseRepository<IIInternshipProposal, InternshipProposal> {
7
8      /**
9       * Initialize [[InternshipsProposalsRepository]]
10      */
11      constructor(
12          // The injected [[InternshipProposalModel]] model
13          @inject(types.Models.InternShipProposal)
14          protected internshipProposalModel: Model<IIInternshipProposal>,
15
16          // The lazy-injected [[InternshipsRepository]] repository
17          // (lazy to avoid circular-dependencies errors)
18          @inject(new LazyServiceIdentifier(() => InternshipsRepository))
19          private internshipRepository: InternshipsRepository) {
20
21          // Initialize [[BaseRepository]]
22          super(internshipProposalModel,
23              Defaults.collectionsName.internshipProposals);
24      }
25
26      /**
27       * Return the number of available places for an internship
28       * @param internshipId The internship id
29       */
30      public async getAvailablePlaces(internshipId: string) {
31          ...
32      }
33
34  }

```

Figura 2.2 Esempio di *repository* dell'applicazione back-end

accedere e ad altri *repositories* iniettati dal sistema di Dependency injection (DI). Il sistema di DI adottato dal sistema si basa sul pacchetto npm 'inversify', che fornisce anche un modo

di aggirare le dipendenze circolari (ad esempio tra due *repositories*) tramite un meccanismo di *lazy-inject*.

2.1.3 Controllers

I *controllers* espongono su architettura REST API un metodo di *Express.js* che internamente utilizza le operazioni dei *repositories*. Anche i *controllers* sono legati ad un'unica entità e derivano da un *BaseController* che di default espone i metodi per le operazioni di CRUD dell'entità. I *controllers* sono responsabili di gestire l'autenticazione dell'utente e le eventuali eccezioni sollevate dai *repositories*. Ogni *controller* può esporre metodi che richiedono ruoli diversi, quindi ogni metodo definisce tramite uno *scope* quale siano gli utenti abilitati ad eseguire quel metodo e in caso negativo ritorna un errore di autenticazione. Come possiamo

```
1  /**
2   * Update the state of an [[Internship]] following the [[
3     InternshipStatusTypeMachine]] transition function
4   */
5   private useUpdateStates() {
6     this.router.put('/status', [ownInternshipProposal],
7       async (req, res) => {
8         try {
9           const internshipProposalId = req.body.id;
10          const newState:
11            InternshipProposalStatusType = req.body.status;
12
13          // Get the internship
14          const internshipProposal = await
15            this.internshipProposalsRepository
16              .get(internshipProposalId);
17
18          // ... do stuff ...
19
20          // Return the updated internship proposal
21          return new ApiResponse({
22            data: internshipProposal,
23            httpCode: 200,
24            response: res
25          }).send();
26        } catch (ex) {
27          // return ex
28        }
29      });
30  }
```

Figura 2.3 Esempio di registrazione di un metodo *Express.js* in un *Controller*

notare in figura 2.3 il *controller* sta registrando una route di *Express.js* il cui secondo parametro è un array. Questo array contiene un insieme di *middleware* che *Express.js* si occuperà di invocare prima di eseguire il codice all'interno della callback. Il *middleware*, in questo caso la funzione *ownInternshipProposal*, verifica che l'utente che ha effettuato la chiamata sia effettivamente un soggetto (azienda, professore o studente) della proposta di tirocinio di cui vuole aggiornare lo stato. In caso negativo rifiuta la richiesta con un errore e la termina. L'autenticazione del sistema verrà discussa in dettaglio nel capitolo 4.1 - *Autenticazione*.

2.1.4 Bootstrap

L'avvio dell'applicazione avviene nel file *server.ts*, che si preoccupa di registrare tutti i *repositories* nel sistema di Dependency injection e infine di risolvere i *controllers*. L'applicazione

```
1 // Bind all repositories in DI container
2 container.bind<UsersRepository>(UsersRepository)
3   .to(UsersRepository).inTransientScope();
4
5 container.bind<InternshipsRepository>(InternshipsRepository)
6   .to(InternshipsRepository).inTransientScope();
7
8 // Bind all repositories...
9
10 // And then resolve and register controllers
11 const internshipsController = container
12   .resolve(InternshipsController) // resolve dependencies
13   .useAuth() // use auth middleware
14   .useCustoms() // use custom methods
15   .useCrud({ // use CRUD operation
16     delete: {
17       middleware: [adminScope] // optional middleware
18     },
19     update: {
20       middleware: [ownInternship] // optional middleware
21     }
22   })
23   .register(); // register routes
24
25 // Resolve all controllers...
```

Figura 2.4 Estratto di *Server.ts*, bootstrap del back-end

si preoccupa anche di connettersi al database *MongoDB* e registrare nel container l'istanza dell'applicazione in modo sia accessibile dove necessario. Definisce inoltre un *middleware* per catturare eventuali eccezioni non gestite dai *controllers*.

2.2 Architettura lato client

Il front-end di *DAIS Internship Manager* è un'applicazione *Angular* (v6) composta da diversi moduli (*NgModules*). Vi sono due moduli principali, uno che contiene le pagine che può visualizzare un utente non autenticato (*NoAuthModule*) e uno che contiene le pagine visibili agli utenti autenticati (*AuthModule*). Il modulo per gli utenti non autenticati viene caricato per primo, mentre il modulo che contiene le pagine protette viene caricato con la tecnica del *lazy-loading* solamente una volta effettuato il login. Esiste poi un altro macro modulo, *SharedModule*, che contiene dipendenze utilizzare in entrambi gli altri due moduli e che viene infatti importato in essi.

2.2.1 Moduli e divisione delle responsabilità

SharedModule

Contiene le dipendenze, i *components*, le *pipes* e le *directives* utilizzate negli altri moduli. Viene importato sia in *NoAuthModule* che in *AuthModule*.

NoAuthModule

Contiene la pagine che solo un utente non autenticato può raggiungere, ovvero *login* di un utente e *registrazione* di un'azienda. Una volta effettuato il login, il *router* carica il modulo *AuthModule* che contiene invece tutte le pagine protette e fintanto che l'utente non effettua il *logout* non può più raggiungere le pagine in *NoAuthModule*.

AuthModule

Contiene la pagine che solo un utente autenticato può raggiungere, ovvero il cuore dell'applicazione. Data la sua dimensione questo modulo contiene degli altri sotto moduli:

- *UserModule*: contiene la parte di gestione dell'account di un utente, con la possibilità di modificarlo
- *Internship*: contiene la parte di gestione dei tirocini (aggiunta, modifica, approvazione, dettaglio, candidatura)
- *InternshipProposal*: contiene la parte di gestione dei tirocini (modifica, approvazione, dettaglio, tracciamento)
- *Shared*: contiene i componenti comuni a questi sotto moduli (header, footer e sidebar)

2.2.2 Servizi e recupero dei dati

I moduli visti poco sopra contengono i componenti responsabili della visualizzazione dei dati, che tuttavia non si preoccupano di recuperare in autonomia. Essi si affidano infatti ad uno strato di servizi che interagiscono con il back-end dell'applicazione, i *services*. I *services* di *Angular* seguono lo standard del back-end, ovvero ognuno di essi si preoccupa di gestire una sola entità. All'interno dei componenti che contengono il *template* verranno iniettati uno

```
1  /**
2   * The [[InternshipProposal]] service
3   *
4   * @extends {BaseService}
5   */
6  @injectable()
7  export class InternshipProposalService extends BaseService {
8
9      /**
10       * Return a list of all proposals that
11       * reference the given professor id
12       *
13       * @param {string} professorId The professor id
14       */
15     getByProfessorId(professorId: string):
16         Promise<ApiResponseDto<Array<InternshipProposal>>> {
17
18         return this.getVerb(
19             `${Defaults.collectionsName.internshipProposals}/
20             getByProfessorId/${professorId}`
21         );
22     }
23
24     // ...
25 }
```

Figura 2.5 Esempio di *service* front-end

o più *services* che permetteranno di recuperare i dati da visualizzare.

2.2.3 Supporto multi lingua

La gestione della localizzazione dell'app è gestita tramite il pacchetto npm *ngx-translate*, che fornisce dei componenti per la traduzione delle stringhe nei *template* dei *components* e nei relativi *view-model*. Le traduzioni sono salvate in un file JSON ed è possibile recuperare in differenti modi:

- *Pipe*: da utilizzare nel *template* quando ci sono espressioni da valutare

```
1 {
2   "Dictionary": {
3     "Back": "Indietro",
4     ...
5   },
6   "Pages": {
7     "Index": {
8       "Title": "DAIS Internship Manager",
9       "SubTitle": "Benvenuto {{fullname}}, sono le {{datetime}}"
10    },
11    ...
12  },
13  ...
14 }
```

Figura 2.6 Estratto del file di globalizzazione front-end

- *Directive*: da utilizzare nel *template* quando la stringa da tradurre è cablata
- *Service*: da utilizzare nel *view-model* dei *components*

```
1 // Pipe
2 <p>
3   {{'Dictionary.Back' | translate}}
4 </p>
5
6 // Directive
7 <p translate [translate-params]="currentUser">
8   Pages.Index.SubTitle
9 </p>
10
11 // Service
12 const translatesString = await
13   this.translateService.get('Pages.Index.Title');
```

Figura 2.7 Localizzare un *component* con *ngx-translate*

Un servizio registrato al *bootstrap* dell'applicazione si preoccupa di leggere la lingua corrente e caricare il file di traduzione relativo, oppure uno di *fallback*.

Capitolo 3

Casi d'uso e workflow

3.1 Casi d'uso

Nel seguente paragrafo verranno descritti i casi d'uso che l'applicazione supporta.

3.1.1 Registrazione e login di un utente

Deve essere possibile registrare un nuovo utente nel sistema e permettergli di autenticarsi.

Vi sono due tipologie di utenti, quelli interni all'ateneo (professori e studenti) e quelli esterni (aziende). Gli utenti interni devono effettuare l'accesso con il proprio account istituzionale utilizzando Google come provider federato, mentre le aziende possono inserire un'email qualunque.

Una volta che un membro dell'ateneo esegue l'accesso mediante Google, viene reindirizzato indietro al sistema, che verifica l'email utilizzata nella fase di accesso a Google: se termina con *@stud.unive* rappresenta un utente con il ruolo di studente, mentre se termina con *@unive.it* rappresenta il un utente con il ruolo di professore. Una volta determinato il ruolo, il sistema verifica l'esistenza di un utente nella relativa *collection* del database e in caso negativo lo crea, registrando un nuovo utente. A questo punto l'utente (professore o studente) è autenticato e viene indirizzato nella parte del portale protetta.

Per quanto riguarda un utente con il ruolo di azienda, invece, la procedura di *login* e registrazione è divisa in pagine diverse. Nella pagina di registrazione l'utente deve indicare i dati della propria azienda, un'indirizzo email e una password, dopodiché verrà creato sia una nuova azienda che un nuovo utente, associando l'utente come amministratore della nuova azienda. Nella procedura di login invece, l'utente dovrà inserire l'email e la password utilizzate in fase di registrazione.

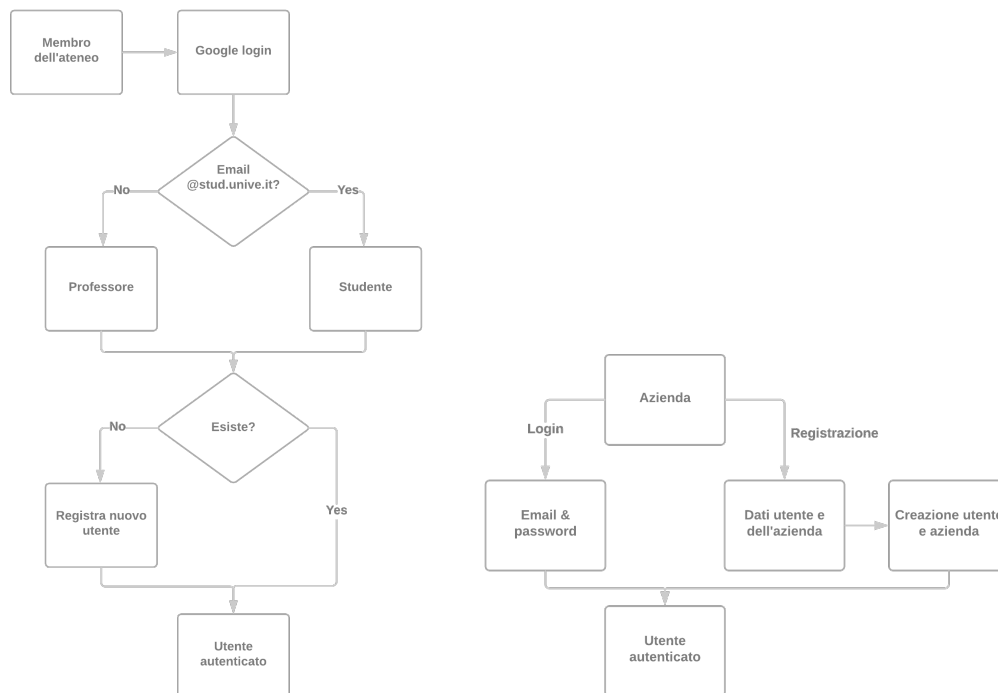


Figura 3.1 Flusso di login e registrazione di un utente

3.1.2 Creazione di un tirocinio

Tirocinio

3.1.3 Modifica di un tirocinio

And some more ...

3.1.4 Approvazione di un tirocinio

And some more ...

3.1.5 Candidatura ad un tirocinio

And some more ...

3.1.6 Approvazione candidatura (professore)

And some more ...

3.1.7 Approvazione candidatura (azienda)

And some more ...

3.1.8 Avvio di un tirocinio

And some more ...

3.1.9 Compilazione foglio presenze

And some more ...

3.1.10 Generazione documentazione

And some more ...

Capitolo 4

Application Programming Interface

4.1 Autenticazione

Express middleware, auth controller

4.1.1 Server side

Utilizzo JWT per gestire lo stato del sistema

4.1.2 Client side

Utilizzo JWT per gestire lo stato del sistema

4.2 Endpoints

4.2.1 Controller base

Descrizione controller base

Tabella 4.1 Endpoint rest API

URL	Metodo	Parametri	Risposta
/api/internship	GET		ApiResponseDto<Array<Internship>>
I1LL	7.48	0.56	8.7
I2MD	3.99	0.63	4.2
I2LL	6.81	0.02	6.66
CMD	13.47	0.09	10.55
CBL	11.88	0.05	13.11

4.2.2 Internships

Metodi custom

4.2.3 InternshipProposals

Metodi custom

4.2.4 Roles

Metodi custom

4.2.5 Users

Metodi custom

4.2.6 Companies

Metodi custom

4.2.7 Auth

Metodi custom

Capitolo 5

Esempi e screenshot

5.1 Screenshot workflow

Capitolo 6

Conclusioni

6.1 Sviluppi futuri e nuove integrazioni

- Stage curriculari/extra curriculari
- Rimborso spese
- Assicurazione
- Amministratore (collegamento segreteria per riconoscimento crediti)

Bibliografia

- [1] Node.js Foundation. Node.js, 2018. URL <https://nodejs.org/it/>.
- [2] Wikimedia Foundation. Node.js - wikipedia, 2018. URL <https://en.wikipedia.org/wiki/Node.js>.
- [3] Google. Angular - architecture overview, 2018. URL <https://angular.io/guide/architecture>.
- [4] MongoDB Inc. MongoDB architecture, 2018. URL <https://www.mongodb.com/mongodb-architecture>.
- [5] MongoDB Inc. Sql to mongodb mapping chart, 2018. URL <https://docs.mongodb.com/manual/reference/sql-comparison/>.

Arconimi

API Application Programming Interface. 2, 5, 11, 23, 25, *Glossario*: Application Programming Interface

JSON Binary JSON. 2, 3, 23, 25, *Glossario*: Binary JSON

DBMS Database Management System. 2, 12, 23, 25, *Glossario*: Database Management System

DI Dependency injection. 8, 23, 25, *Glossario*: Dependency injection

DOM Document Object Model. 7, 8, 23, 25, *Glossario*: Document Object Model

EDA Event-driver architecture. 5, 23, 25, *Glossario*: Event-driver architecture

HTML HyperText Markup Language. 7, 23, 25, 27, *Glossario*: HyperText Markup Language

HTTP Hyper Text Transfer Protocol. 6, 23, 25, 29, *Glossario*: Hyper Text Transfer Protocol

JSON Javascript Object Notation. 2, 3, 23, 25, 27, *Glossario*: Javascript Object Notation

MEAN MongoDB, Express, Angular & Node.js. 2, 23, 25, *Glossario*: MongoDB, Express, Angular & Node.js

MVC Model-View-Controller. 6, 23, 25, *Glossario*: Model-View-Controller

NoSQL No Structured Query Language. 2, 3, 23, 25, *Glossario*: No Structured Query Language

npm Node Package Manager. 11, 12, 23, *Glossario*: Node Package Manager

Observer Observer pattern. 5, 23, 25, *Glossario*: Observer pattern

OOP Object-oriented programming. 3, 23, 25, 29, *Glossario*: Object-oriented programming

ORM Object-relational mapping. 11, 12, 23, 26, *Glossario*: Object-relational mapping

REST Representational State Transfer. 11, 23, 26, *Glossario*: Representational State Transfer

URL Uniform Resource Locator. 6, 8, 23, 26, *Glossario*: Uniform Resource Locator

Glossario

Application Programming Interface Un insieme di definizioni di metodi, protocolli e strumenti per la creazione di software applicativo. In termini generali, si tratta di un insieme di metodi di comunicazione definiti in modo chiaro tra i vari componenti software. 23, 25

Back-end Si intende la parte di applicazione non visibile all'utente finale che manipola, gestisce e fornisce i dati alla parte di front-end . 11, 23

Binary JSON Una rappresentazione binaria di JSON. 23, 25

Callback Rappresenta un codice eseguibile che viene passato come argomento ad un'altra funzione da cui ci si aspetta che venga richiamata (eseguita) in un dato momento. L'esecuzione potrebbe essere immediata come nei callbacks sincroni oppure potrebbe verificarsi in un momento successivo, come nel caso dei callbacks asincroni . 5, 6, 23

Database Management System Un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database. 2, 23, 25

Dependency injection Un design pattern della programmazione orientata agli oggetti il cui scopo è quello di semplificare lo sviluppo e migliorare la testabilità di software di grandi dimensioni. 8, 23, 25

Document Object Model Una forma di rappresentazione dei documenti strutturati come modello orientato agli oggetti. Nativamente supportato dai browser per modificare gli elementi di un documento HTML, DOM è un modo per accedere e aggiornare dinamicamente il contenuto, la struttura e lo stile dei documenti.. 7, 23, 25

Event-driver architecture Un pattern di architettura software che promuove la produzione, l'individuazione, il consumo e la reazione agli eventi. 23, 25

- Event-loop** Un costrutto di programmazione che attende e invia eventi o messaggi in un programma . 5, 6, 23
- Framework** Fornisce un modo standard per creare e distribuire applicazioni . 5, 6, 11, 23
- Front-end** Si intende la parte di applicazione visibile all'utente finale, che nasconde i dettagli implementativi . 2, 6, 11, 23, 27
- Hyper Text Transfer Protocol** Protocollo di comunicazione che sta alla base del World Wide Web, Supporta diversi metodi, detti verbi, quali *GET*, *PUT*, *POST*, *PATCH*, *DELETE*, *OPTIONS*, *HEAD*. 23, 25
- HyperText Markup Language** Un linguaggio di markup per la formattazione e impaginazione di documenti ipertestuali disponibili nel web. 23, 25
- Javascript** Un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi . 4, 5, 7, 23, 28, 29
- Javascript Object Notation** Una rappresentazione testuale che permette di codificare un oggetto javascript. 2, 23, 25
- JOIN** Una clausola del linguaggio SQL che serve a combinare le tuple di due o più relazioni di un database tramite l'operazione di congiunzione dell'algebra relazionale . 3, 4, 12, 23
- Model-View-Controller** Un pattern architetturale comunemente usato per lo sviluppo di software che divide un'applicazione in tre parti interconnesse. Separa le rappresentazioni interne delle informazioni dal modo in cui vengono presentate all'utente. Il modello di progettazione MVC disaccoppia questi componenti principali consentendo un riutilizzo efficiente del codice. 6, 23, 25
- MongoDB, Express, Angular & Node.js** Uno stack di tecnologie utilizzate insieme al fine di creare applicazioni web. 2, 23, 25
- No Structured Query Language** Archivi di dati che il più delle volte non richiedono uno schema fisso (schemaless), evitano spesso le operazioni di giunzione (join) e puntano a scalare in modo orizzontale. 2, 23, 25

Node Package Manager Un package manager per javascript, è quello di default per il runtime di *Node.js*. 11, 23

Object-oriented programming Un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso uno scambio di messaggi. 23, 25, 29

Object-relational mapping Una tecnica di programmazione per convertire dati tra sistemi incompatibili utilizzando linguaggi Object-oriented programming (OOP). 12, 23, 26

Observer pattern Un design pattern in cui un oggetto, chiamato *subject*, mantiene un elenco dei suoi dipendenti, chiamati *observers* e li notifica automaticamente di qualsiasi cambiamento di stato, di solito chiamando uno dei loro metodi. 23, 25

Representational State Transfer Un'architettura software per trasmettere dati su HTTP in modo stateless. 23, 26

Typescript Un linguaggio programmazione open-source sviluppato di Microsoft, super-set di javascript, estende la sintassi di javascript introducendo lo static typing . 6, 7, 23

Uniform Resource Locator Rappresenta una referenza ad una risorsa web. Specifica la sua posizione nella rete e il meccanismo per recuperarla. 23, 26