

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

DAIS Internship Manager è un'applicazione web realizzata con lo scopo di agevolare la creazione, la ricerca e la gestione dei tirocini del Dipartimento di Informatica, Scienze ambientali e Statistica (DAIS) sviluppata utilizzando tecnologie web-based su stack *MongoDB, Express, Angular & Node.js (MEAN)*.

Con questa piattaforma un'azienda sarà in grado di registrarsi e pubblicare un'offerta di tirocinio che, una volta approvata da un professore, verrà pubblicata e resta disponibile agli studenti. Gli studenti a loro volta potranno candidarsi alle offerte, scegliere un docente come referente per il percorso, completare il foglio presenze e infine stampare la documentazione in *PDF* precompilata.

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	11
2.1.2 Repositories	13
2.1.3 Controllers	15
2.1.4 Bootstrap	17
2.2 Architettura lato client	18
2.2.1 Moduli e divisione delle responsabilità	18
2.2.2 Servizi e recupero dei dati	19
2.2.3 Supporto multi lingua	20

3	Casi d'uso e workflow	23
3.1	Ciclo di vita di un tirocinio	23
3.2	Casi d'uso	25
3.2.1	Registrazione e login di un utente	25
3.2.2	Creazione di un tirocinio	28
3.2.3	Approvazione di un tirocinio	29
3.2.4	Candidatura ad un tirocinio	30
3.2.5	Approvazione candidatura (professore)	31
3.2.6	Approvazione candidatura (azienda)	32
3.2.7	Avvio di un tirocinio	33
3.2.8	Compilazione foglio presenze	34
3.2.9	Terminazione di un tirocinio	35
3.2.10	Generazione documentazione	36
4	API - Application Programming Interface	37
4.1	Autenticazione	37
4.1.1	Back-end	37
4.1.2	Front-end	40
4.2	Endpoints	41
4.2.1	BaseController	42
4.2.2	InternshipsController	42
4.2.3	InternshipProposalsController	43
4.2.4	RolesController	45
4.2.5	UsersController	45
4.2.6	CompaniesController	46
4.2.7	AuthenticationController	46
5	Conclusioni	47
5.1	Sviluppi futuri e nuove integrazioni	47
	Bibliografia	49
	Acronimi	51
	Glossario	53

Elenco delle figure

1.1	esempio di paragone <i>JSON-BSON</i>	3
1.2	Esempio di sintassi di <i>MongoDB</i>	4
1.3	Esempio di applicazione <i>Express.js</i>	6
1.4	Architettura di Angular	8
2.1	Esempio di <i>schema</i> dell'applicazione <i>back-end</i>	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>	15
2.4	Estratto di <i>Server.ts</i> , bootstrap del <i>back-end</i>	17
2.5	Esempio di <i>service front-end</i>	19
2.6	Estratto del file di globalizzazione <i>front-end</i>	20
2.7	Localizzare un <i>component</i> con <i>ngx-translate</i>	21
3.1	Ciclo di vita di un tirocinio	24
3.2	Flusso di login e registrazione di un utente	26
3.3	Screenshot: login e registrazione di un'azienda	27
3.4	Screenshot: aggiunta di un'offerta di tirocinio	28
3.5	Screenshot: approvazione di una nuova offerta di tirocinio	29
3.6	Screenshot: candidatura ad un'offerta di tirocinio	30
3.7	Screenshot: approvazione di una candidatura (professore)	31
3.8	Screenshot: approvazione di una candidatura (azienda)	32
3.9	Screenshot: avvio di un tirocinio	33
3.10	Screenshot: tracciamento di un tirocinio avviato	34
3.11	Screenshot: terminazione di un tirocinio	35
3.12	Screenshot: documentazione generata dal sistema	36
4.1	Metodo <i>useAuth</i> di <i>AuthController</i>	38
4.2	<i>AuthMiddleware</i> di <i>AuthController</i>	39
4.3	Esempio di metodo esposto da <i>BaseService</i>	41

4.4	Macchina a stati di un tirocinio	43
4.5	Macchina a stati di una proposta di tirocinio (candidatura)	44

Elenco delle tabelle

1.1	Confronto modello query e indicizzazione di <i>MongoDB</i> e altri database ^[5]	4
4.1	Endpoint <i>BaseController</i>	42
4.2	Endpoint <i>InternshipController</i>	42
4.3	Endpoint <i>InternshipProposalsController</i>	44
4.4	Endpoint <i>UsersController</i>	45
4.5	Endpoint <i>CompaniesController</i>	46
4.6	Endpoint <i>AuthenticationController</i>	46

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 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 una sola soluzione per raggiungere tutti i dispositivi più utilizzati — computer, smartphone e tablet.

Un requisito di fondamentale importanza è quindi un design responsivo dell'applicazione, data la diversità dei dispositivi che si intende supportare. Per favorire ulteriormente l'accessibilità dell'applicazione, inoltre, essa dovrà essere multilingua e in questa prima versione 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 *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 applicazioni web e *REST API* (*Express.js*)
3. Un *framework* per la creazione di *SPA* 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 *Binary JSON* (*BSON*). 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.

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

```
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,
9   \x00                       // 0x00 = tipo EOO ('end of object'))
```

Figura 1.1 esempio di paragone *JSON-BSON*

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 a 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 alla singola lettura per il recupero dell'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-practice).

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

```

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 1.2 Esempio di sintassi di *MongoDB* vs SQL

MongoDB presenta alcune funzionalità non previste dalle altre architetture di database. Una comparazione delle features più interessanti tra *MongoDB* e altri tipi di database è raffigurata in tabella 1.1.

Tabella 1.1 Confronto modello query e indicizzazione di *MongoDB* e altri database^[5]

	<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 <i>JOIN</i>	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.»^[2]

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. *Node.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*»^[3], unificando lo sviluppo di applicazioni web attorno ad un unico linguaggio di programmazione piuttosto che separando i 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 real-time. 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 sopra 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 manipolazione 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 1.3 Un esempio di applicazione *Express.js*

Come possiamo vedere in figura 1.3, 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 *Single Page Application (SPA)* basato su *typescript*. Gli elementi costitutivi principali di *Angular* verranno

discussi nei prossimi paragrafi.

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 essere 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* permette 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, oppure che si desidera condividere tra componenti, si crea una *service class*. La definizione di un *Service* è immediatamente preceduta dal decoratore *@Injectable*. Il decoratore fornisce i *metadati* che consentono al servizio di essere iniettato nei componenti client come dipendenza (i *metadati* permettono la *reflection* per il sistema di *DI*).

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*).

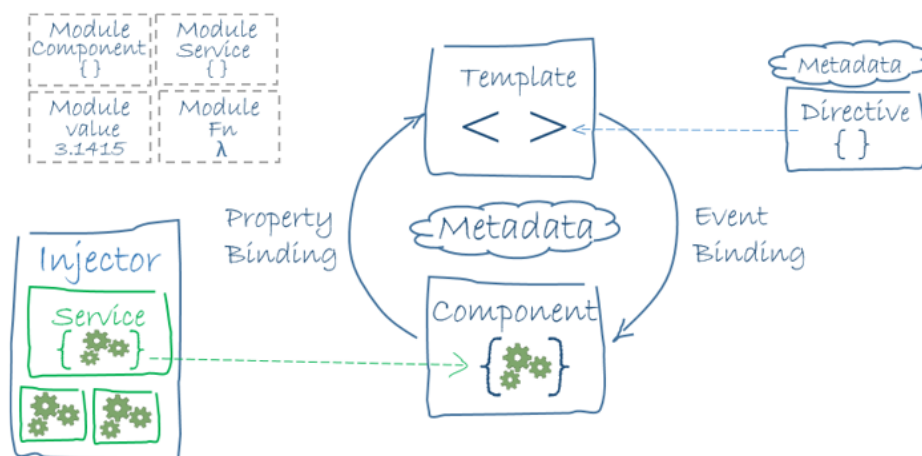


Figura 1.4 Schema dell'architettura di Angular^[4]

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 ne è previsto l'utilizzo.

1.4.1 Azienda

Rappresenta una (o più) persona fisica responsabile della gestione di un'azienda di cui si fa portavoce. 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 un tirocinio. Può completare il foglio presenze di un tirocinio in corso nella propria azienda e stamparne la documentazione precompilata al termine.

1.4.2 Professore

Rappresenta un componente interno all'ateneo che ha il compito di supervisionare le offerte e seguire gli studenti nel percorso. 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 il cui ruolo si basa 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 lo coinvolgono come referente, può completare il foglio presenze di un tirocinio in corso (di cui è referente), e stamparne la documentazione precompilata al termine.

1.4.3 Studente

Rappresenta un componente interno all'ateneo che desidera effettuare un tirocinio in un'azienda. 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

Non rappresenta un soggetto fisico, ma bensì una figura che ricopre il ruolo di amministratore dei dati, unico soggetto a non avere restrizioni sulla modifica o la cancellazione di essi. 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* di *DAIS Internship Manager*. 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 è divisa in livelli con responsabilità diverse che verranno discussi nei paragrafi seguenti.

2.1.1 Schemas

Gli *schemas* sono il livello più vicino al database, essi definiscono la struttura dei documenti delle collezioni.

Pur utilizzando *MongoDB* che è uno *schema-less* database, infatti, 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 e 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  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 });
20
21 /** The [[InternshipProposalModel]] mongoose schema model */
22 export const InternshipProposalModel = model<IIInternshipProposal>(
23   "InternshipProposal",
24   InternshipProposalSchema,
25   Defaults.collectionsName.internshipProposals
26 );
```

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 *schema* che esportano operazioni su di esso. Interrogano il proprio *schema* 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

```

1  /**
2   * The [[InternshipsProposalsRepository]] repository
3   */
4  @injectable()
5  export class InternshipsProposalsRepository extends
6      BaseRepository<IInternshipProposal, InternshipProposal> {
7
8      /**
9       * Initialize [[InternshipsProposalsRepository]]
10      */
11     constructor(
12         // The injected [[InternshipProposalModel]] model
13         @inject(types.Models.InternShipProposal)
14         protected internshipProposalModel: Model<IInternshipProposal>,
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*

base — *Create, Read, Update e Delete (CRUD)* — oltre che un metodo per eseguire query

personalizzate. Ogni *repository* può esporre ulteriori metodi personalizzati ed eventualmente accedere 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 per aggirare le dipendenze circolari (ad esempio tra due *repositories*) tramite un meccanismo di *lazy-inject*.

Lista *repositories* pubblici

I *repositories* presenti nell’applicazione server sono:

- *CompaniesRepository*
- *InternshipProposalsRepository*
- *InternshipsRepository*
- *RolesRepository*
- *UsersRepository*

ognuno dei quali si occupa di una singola entità ed interagisce con un singolo *schema*.

2.1.3 Controllers

I *controllers* espongono su architettura *REST API* un metodo di *Express.js* che internamente utilizza le operazioni pubbliche dei *repositories* iniettati nel *controller* stesso. 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. Tutti i metodi esposti ritornano il risultato racchiuso all'interno di un oggetto, *ApiResponseDto*, che contiene anche informazioni aggiuntive, come lo stato *HTTP* ed eventuali errori.

```
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     return this;
31 }
```

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

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 un *authentication middleware* quale siano gli utenti abilitati ad eseguire quel metodo e in caso negativo ritorna un errore di autenticazione.

Come possiamo 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*.

```
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*

L'applicazione 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*. 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 utilizzate 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 in entrambi gli 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 ritornare alle pagine in *NoAuthModule*. Questa protezione viene effettuata tramite un *service* chiamato *NoAuthGuard*, eseguito dal router stesso di *Angular*, che prima di attivare una route verifica che l'utente non sia autenticato.

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
- *InternshipModule*: contiene la parte di gestione dei tirocini (aggiunta, modifica, approvazione, dettaglio, candidatura)
- *InternshipProposalModule*: contiene la parte di gestione dei tirocini (modifica, approvazione, dettaglio, tracciamento)

- *SharedModule*: contiene i componenti comuni a questi sotto moduli (header, footer e sidebar)

Questo modulo viene caricato solamente una volta che l'utente ha effettuato il login, e tutte le pagine sono protette da un *service* chiamato *AuthGuard*, eseguito dal router stesso di *Angular*, che prima di attivare una route verifica che l'utente sia autenticato e abbia i permessi necessari. L'autenticazione e l'autorizzazione client verranno discusse in dettaglio nel capitolo 4.1.2 - Autenticazione *front-end*

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, mappando uno ad uno i rispettivi *controllers*. All'interno dei componenti che

```
1  /**
2   * The [[InternshipProposal]] service
3   *
4   * @extends {BaseService}
5   */
6  @injectable()
7  export class InternshipProposalService extends BaseService {
8
9      /**
10       * Return a list af 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  }
```

Figura 2.5 Esempio di *service front-end*

contengono il *template* verranno iniettati uno 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*

```
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*

ed è possibile recuperare in differenti modi:

- *Pipe*: da utilizzare nel *template* quando ci sono espressioni da valutare
- *Directive*: da utilizzare nel *template* quando la stringa da tradurre è cablata
- *Service*: da utilizzare nel *view-model* dei *components*

Ogni stringa inserita nel file di traduzione può contenere dei parametri da interpolare, che dovranno essere popolati quando si richiede la traduzione. L'organizzazione del file di traduzione è basata sul concetto di contesto: una traduzione può essere generica (come 'Back' in figura 2.6) oppure specifica per un componente, come il titolo di una pagina. Nel primo caso la risorsa potrà essere riutilizzata in contesti differenti, mentre nel secondo caso no. Tutte le risorse riutilizzabili sono salvate all'interno della voce "Dictionary", mentre tutte le altre sono annidate in un oggetto che auto descrive dove esse vengono utilizzate.

```
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*

La lingua dell'applicazione viene recuperata da un servizio di *SharedModules* registrato al *bootstrap*, che si preoccupa di leggere la lingua del browser, oppure l'ultima preferenza dell'utente, caricare il file di traduzione relativo (oppure uno di *fallback*) e propagare la scelta a tutti gli altri moduli. Utilizzando però il meccanismo di *lazy-loading*, non è possibile cambiare la lingua a run-time, ed è necessario un ricaricamento della pagina per propagare le traduzioni in tutti i componenti.

Capitolo 3

Casi d'uso e workflow

DAIS Internship Manager prevede i seguenti casi d'uso:

1. *Registrazione e login di un utente*
2. *Creazione di un tirocinio*
3. *Approvazione di un tirocinio*
4. *Candidatura ad un tirocinio*
5. *Approvazione di una candidatura (professore)*
6. *Approvazione di una candidatura (azienda)*
7. *Avvio di un tirocinio*
8. *Compiazione del foglio presenze*
9. *Terminazione di un tirocinio*
10. *Generazione documentazione precompilata*

I casi d'uso saranno illustrati nei paragrafi seguenti.

3.1 Ciclo di vita di un tirocinio

Prima di definire in dettaglio i casi d'uso è opportuno spiegare l'intero ciclo di vita di un tirocinio, dalla sua creazione alla sua chiusura. Schematicamente il processo di gestione è rappresentato in figura 3.1.

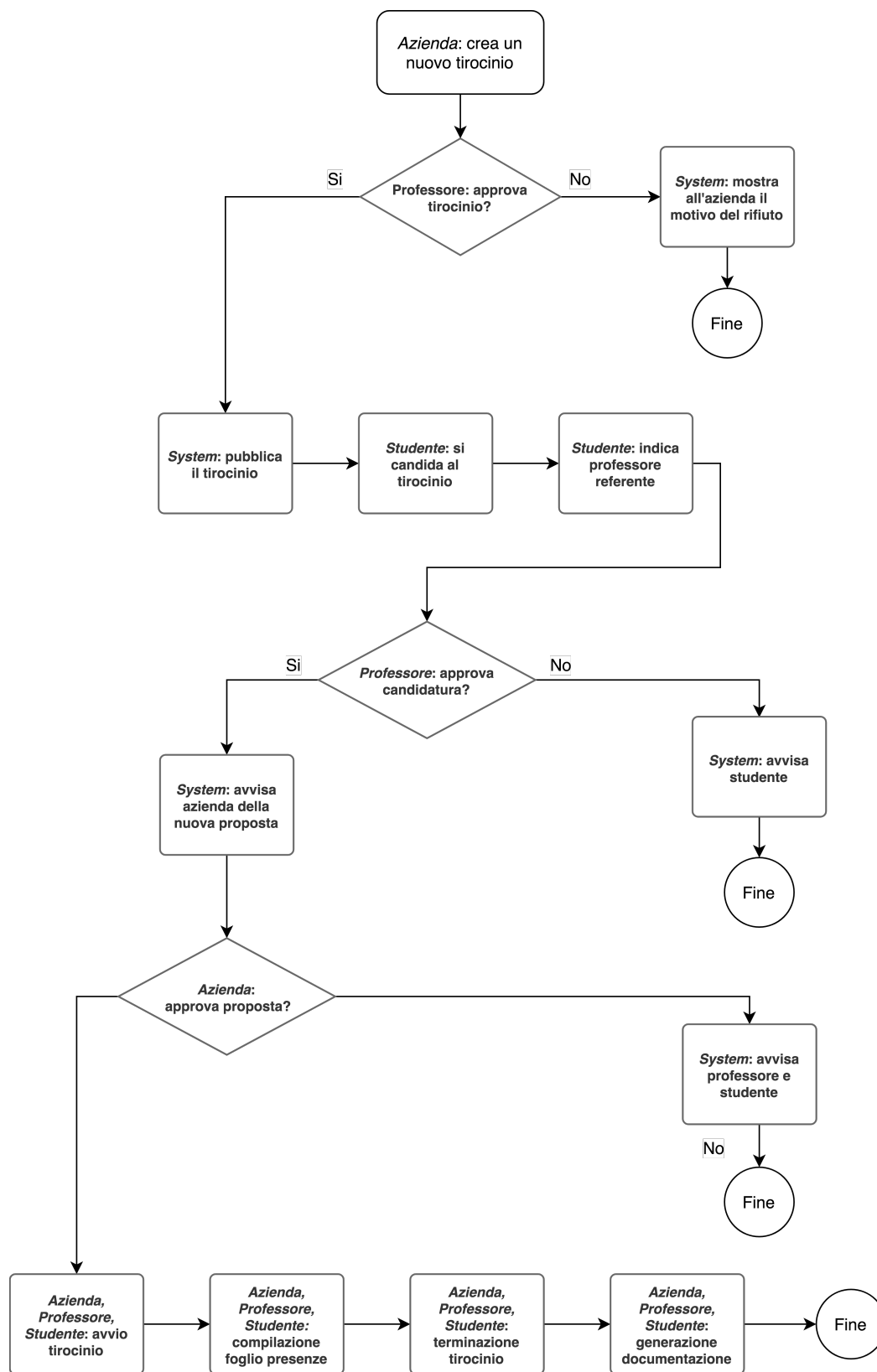


Figura 3.1 Ciclo di vita di un tirocinio

3.2 Casi d'uso

Nel seguente paragrafo verranno descritti i casi d'uso previsti dall'applicazione.

3.2.1 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 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. 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 verranno create 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.

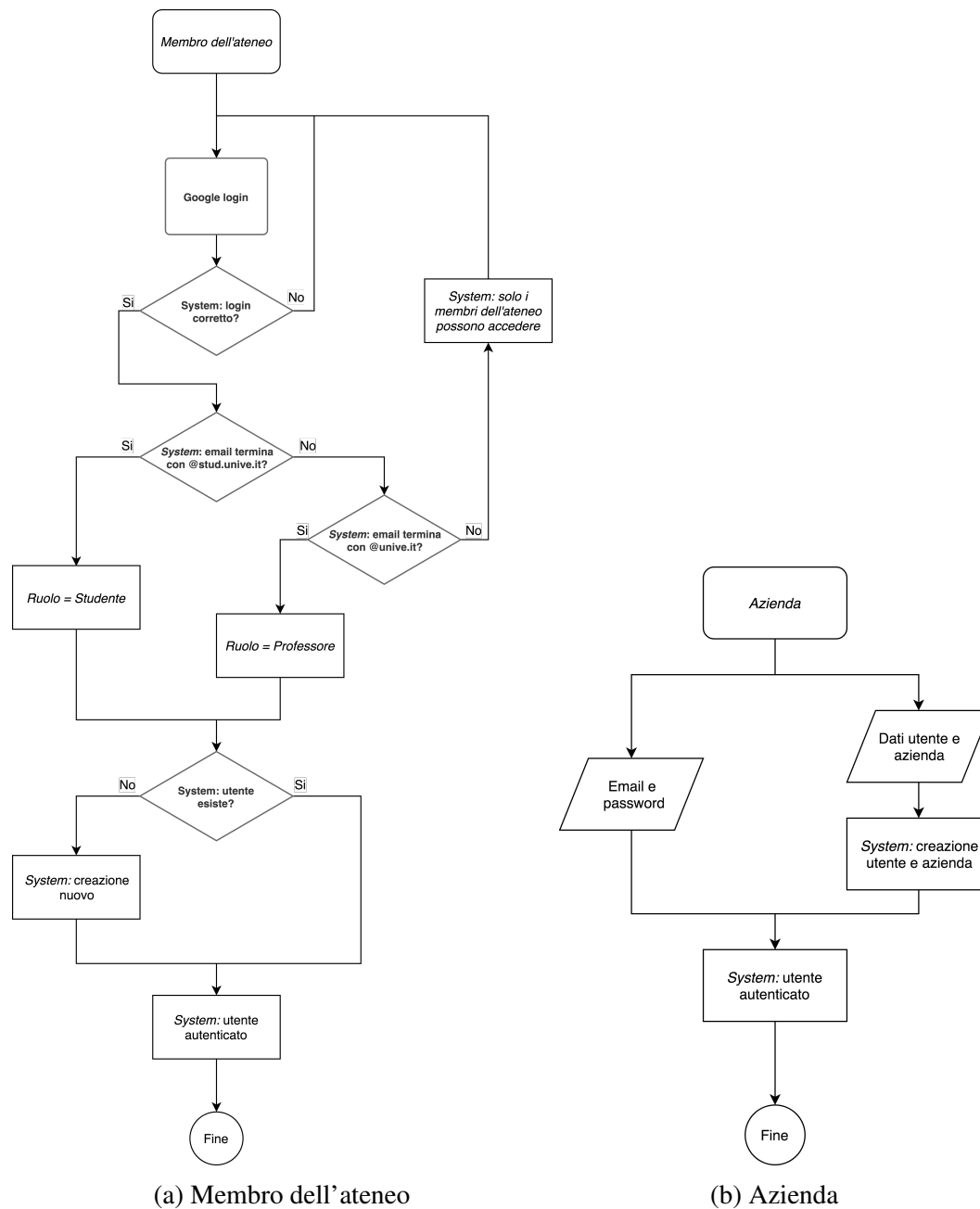
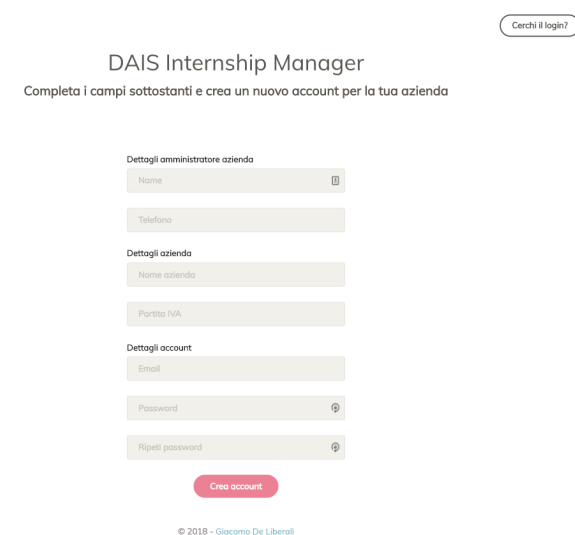


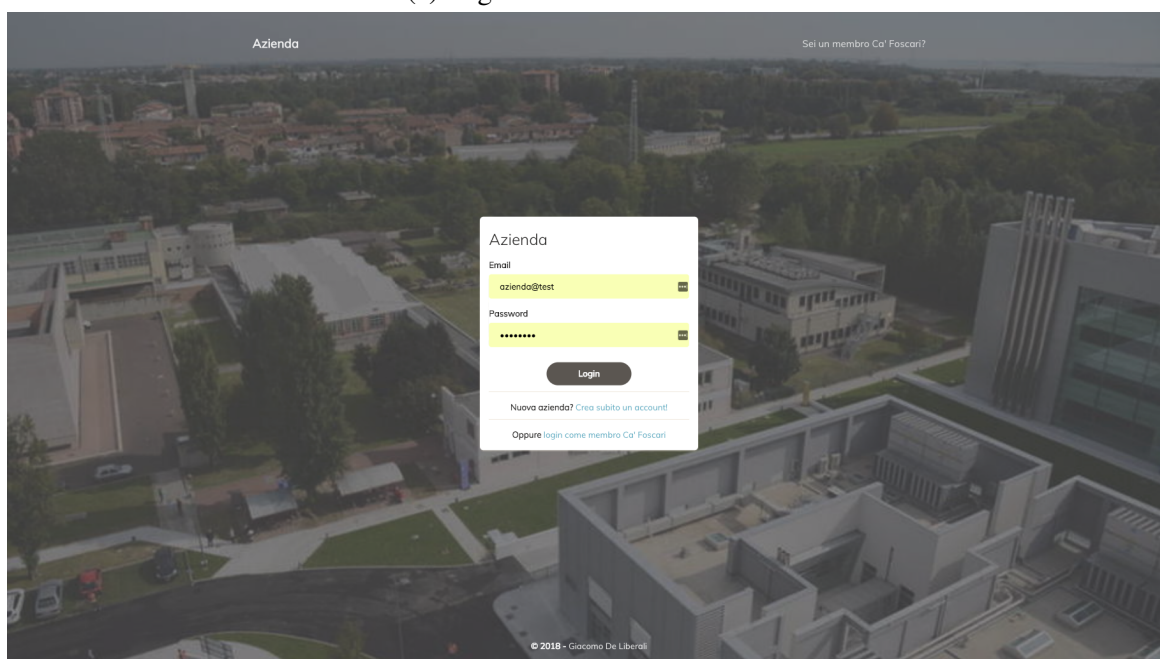
Figura 3.2 Flusso di login e registrazione di un utente

Screenshots



The screenshot shows the 'DAIS Internship Manager' registration page. At the top right is a link 'Cerchi il login?'. The main heading is 'DAIS Internship Manager' with the instruction 'Completa i campi sottostanti e crea un nuovo account per la tua azienda'. The form is divided into three sections: 'Dettagli amministratore azienda' with fields for 'Nome' and 'Telefono'; 'Dettagli azienda' with fields for 'Nome azienda' and 'Partita IVA'; and 'Dettagli account' with fields for 'Email', 'Password', and 'Ripeti password'. A red 'Crea account' button is at the bottom. The footer reads '© 2018 - Giacomo De Liberali'.

(a) Registrazione di un'azienda



The screenshot shows the login form overlaid on an aerial view of a city. The form is titled 'Azienda' and has fields for 'Email' (containing 'azienda@test') and 'Password' (masked with dots). A 'Login' button is below the fields. At the bottom of the form, there are two links: 'Nuova azienda? Crea subito un account!' and 'Oppure login come membro Ca' Foscari?'. The background image shows a city with a large building in the foreground. The footer reads '© 2018 - Giacomo De Liberali'.

(b) Login di un'azienda

Figura 3.3 Login e registrazione di un'azienda

3.2.2 Creazione di un tirocinio

Deve essere possibile — per un utente con il ruolo azienda — l'aggiunta di una nuova offerta di tirocinio.

Una volta effettuato l'accesso con le proprie credenziali, sarà sufficiente seguire la voce di menù "Tirocini > Aggiungi tirocinio". Apparirà una pagina con un *form* che permette l'inserimento di un'offerta. Una volta salvata, prima di essere pubblicata deve essere approvata da un professore. Le offerte pubblicate dall'azienda correntemente autenticata possono essere visualizzate seguendo la voce di menù "Azienda > I miei tirocini".

Screenshots

The screenshot shows the 'Aggiungi stage' (Add internship) form in the 'INTERNSHIP MANAGER' application. The left sidebar contains the user profile 'Mario Rossi' and navigation links for 'HOME', 'TIROCINI' (highlighted), 'Visualizza offerte', 'Aggiungi tirocinio', and 'AZIENDA'. The main form area is titled 'Aggiungi stage' and includes the following fields:

- Azienda:** A dropdown menu with the selected value 'ProxiMax SRL - 987654321'.
- Indirizzo:** A grid of input fields for 'Via' (Desman), 'Numero' (175/A), 'Piano' (Numero), 'Città' (Santa Maria di Sala), 'Provincia' (Venezia), 'CAP' (30036), and 'Nazione' (Italia).
- Dettagli:** A section containing a 'Titolo' field with the value 'Sviluppatore Angular 2+', a 'Descrizione' field with a rich text editor, and a 'Requisiti' (Requirements) list containing 'Redux', 'Ionic', and 'Node.js'.

The bottom of the form shows a status bar with the text 'div ul li strong'.

Figura 3.4 Aggiunta di un'offerta di tirocinio

3.2.3 Approvazione di un tirocinio

Deve essere possibile per un professore approvare o rifiutare un'offerta di tirocinio di un'azienda prima che venga pubblicata.

Una volta che un'azienda pubblica una nuova offerta, essa è visibile da tutti gli utenti con il ruolo di professore alla pagina `"/auth/internship/approve"`, raggiungibile dalla voce "Professore > In attesa di approvazione". Da questa pagina, che contiene una tabella, è possibile approvare o rifiutare singolarmente le offerte visionandone il contenuto. Nel caso in cui l'offerta non venga approvata, è possibile specificare un testo libero contenente il motivo del rifiuto che sarà visualizzato dall'azienda.

Screenshots

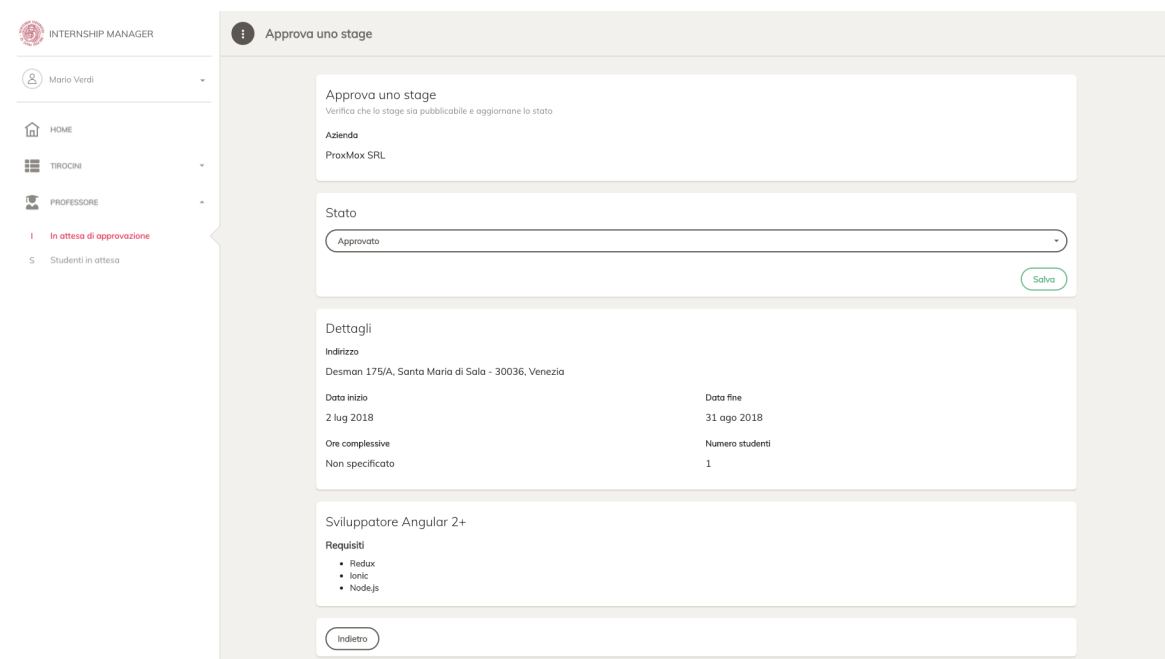


Figura 3.5 Approvazione di una nuova offerta di tirocinio

3.2.4 Candidatura ad un tirocinio

Deve essere possibile per uno studente candidarsi ad un'offerta di tirocinio pubblicata da un'azienda.

Una volta effettuato l'accesso, lo studente può visualizzare tutte le offerte di tirocinio pubblicate. Una volta selezionata quella di suo gradimento, può — se vi sono ancora posti liberi — candidarsi indicando un professore suo referente. La candidatura si effettua tramite il pulsante "Candidati" a fondo pagina di un'offerta di tirocinio. Una volta avviata la candidatura, il professore indicato dovrà accettarla o rifiutarla e solo successivamente sarà rimbalzata all'azienda per la conferma finale.

Screenshots

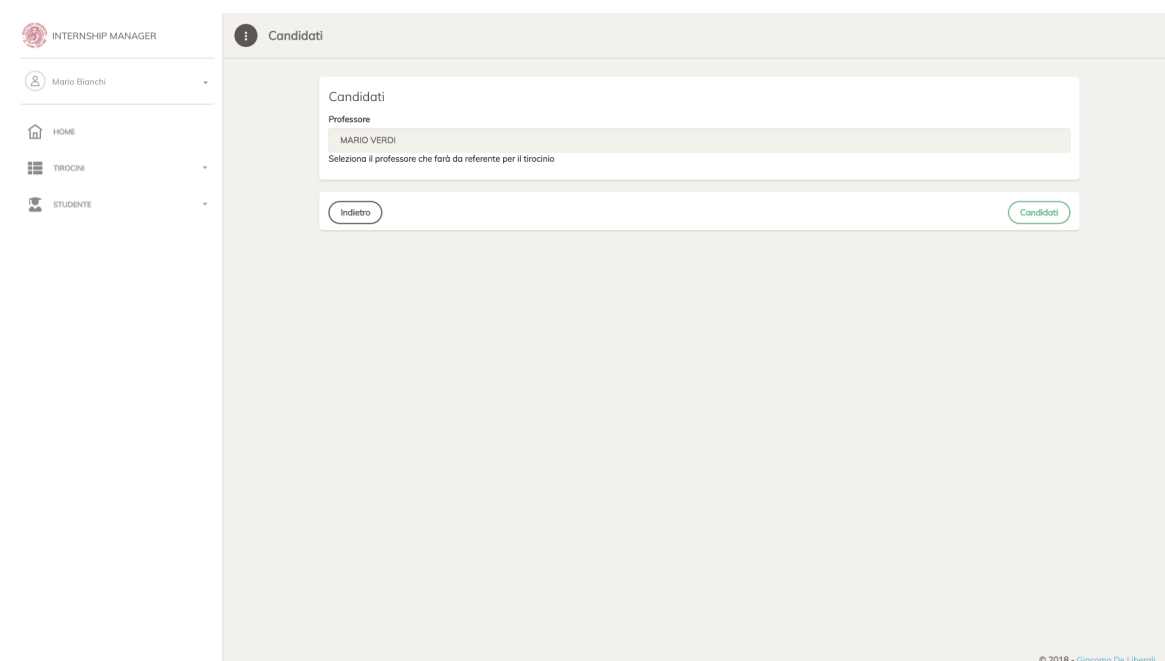


Figura 3.6 Candidatura ad un'offerta di tirocinio

3.2.5 Approvazione candidatura (professore)

Deve essere possibile per un professore visualizzare, accettare e rifiutare le proposte di candidatura che lo coinvolgono come referente.

Una volta effettuato l'accesso, il professore può recarsi alla pagina `"/auth/proposals/professor"` dalla voce "Professore > Studenti in attesa" e visualizzare tutte le candidature che lo coinvolgono. Può accettarle o rifiutarle singolarmente accedendo alla pagina di dettaglio.

Screenshots

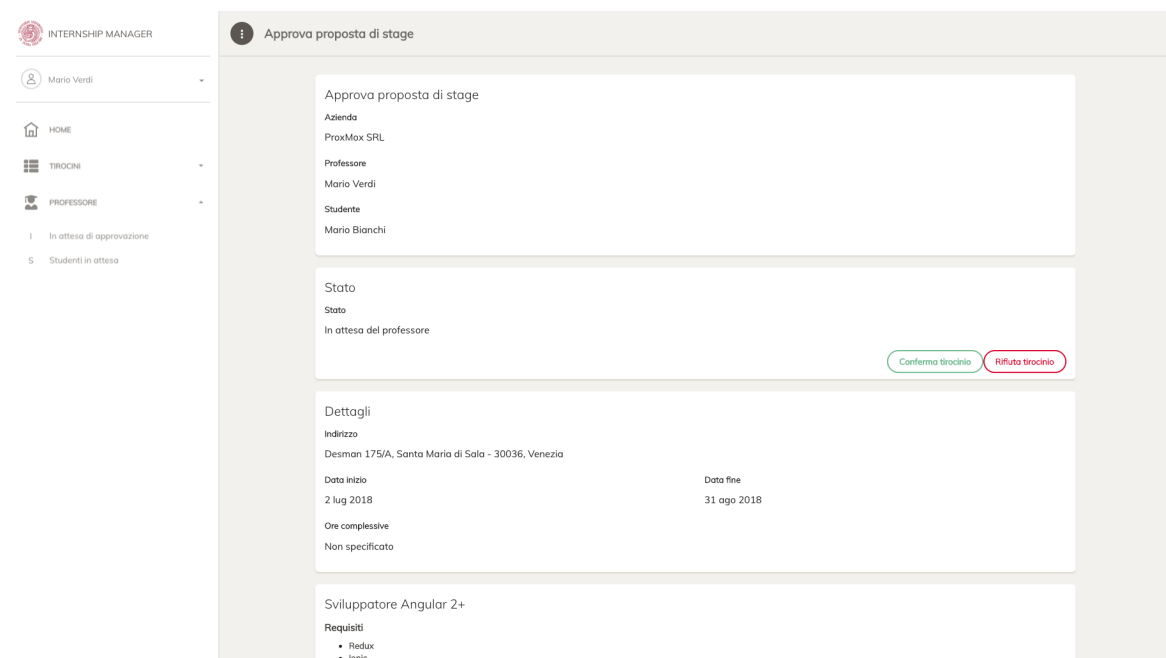


Figura 3.7 Approvazione di una candidatura (professore)

3.2.6 Approvazione candidatura (azienda)

Deve essere possibile per un'azienda visualizzare, accettare e rifiutare le proposte di candidatura alle proprie offerte di tirocinio.

Una volta effettuato l'accesso, l'azienda può recarsi alla pagina `"/auth/proposals/company` dalla voce "Azienda > Proposte ricevute" e visualizzare tutte le candidature che lo coinvolgono. Può accettarle o rifiutarle singolarmente accedendo alla pagina di dettaglio.

Screenshots

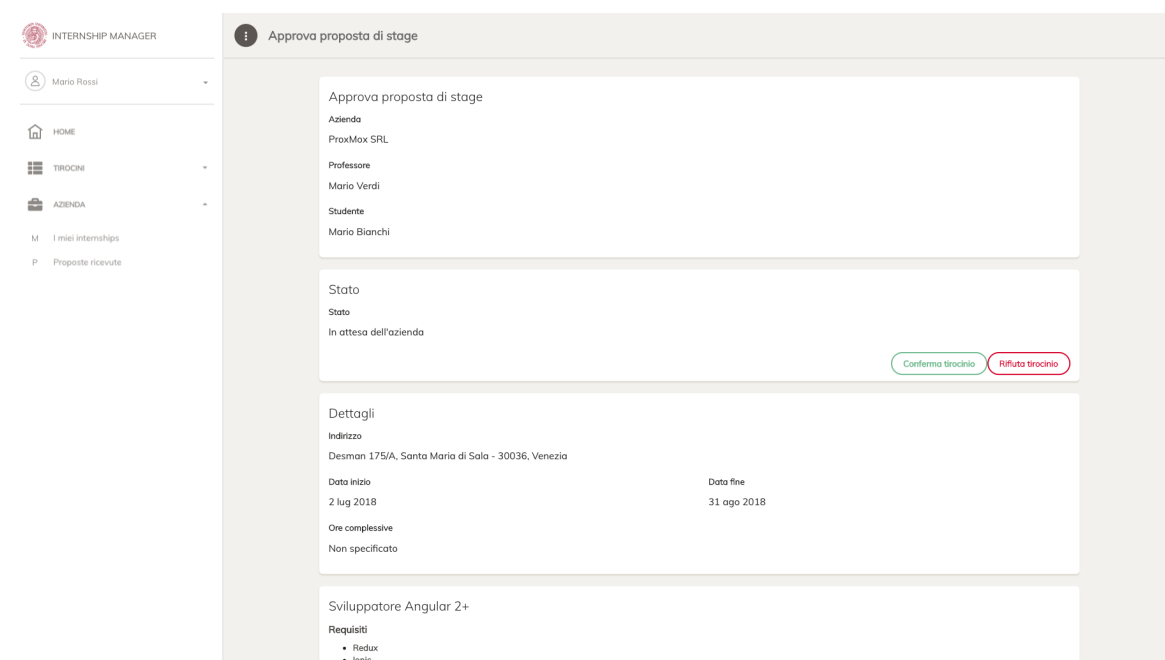


Figura 3.8 Approvazione di una candidatura (azienda)

3.2.7 Avvio di un tirocinio

Deve essere possibile, una volta che sia il professore che l'azienda hanno approvato una candidatura di uno studente, avviare il tirocinio per iniziare a tracciarlo.

Una volta individuato il tirocinio in questione, sarà sufficiente entrare nella pagina di dettaglio della candidatura e premere il pulsante "Avvia tirocinio". Da questo momento in poi sarà possibile completare il foglio presenze e terminare il tirocinio.

Screenshots

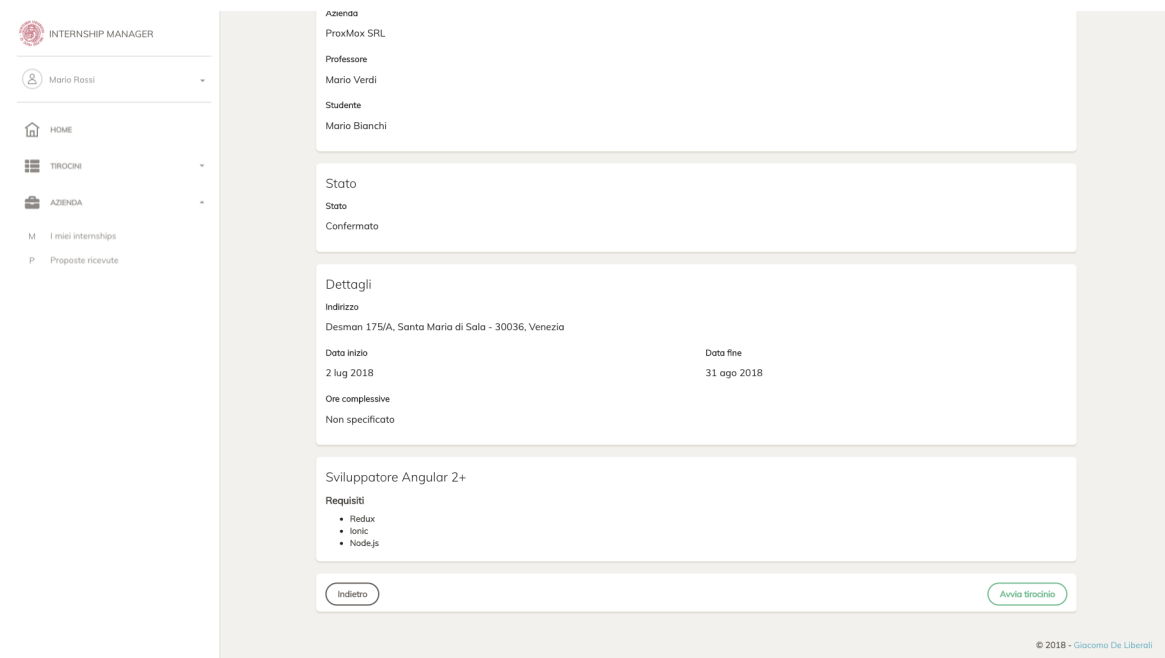


Figura 3.9 Avvio di un tirocinio

3.2.8 Compilazione foglio presenze

Deve essere possibile, una volta avviato un tirocinio, completare il foglio presenze dello studente.

Una volta individuato il tirocinio in questione, sarà sufficiente entrare nella pagina di dettaglio della candidatura, spostarsi all'interno della sezione "Foglio presenze" e premere il pulsante "Aggiungi". Verranno chiesti data e ore lavorate, e un nuovo record sarà aggiunto alla tabella.

Screenshots

INTERNSHIP MANAGER

Mario Rossi

HOME

TIROCINI

AZIENDA

M I miei internships

P Proposte ricevute

Stato corrente

Data di inizio reale: 11 giu 2018

Data di fine reale: Non specificato

Ore rimanenti: Non specificato

Foglio presenze

Aggiungi

Data	Ore lavorate
23 mag 2018	8
24 mag 2018	8
25 mag 2018	8
4 giu 2018	8
5 giu 2018	8
6 giu 2018	8
7 giu 2018	8
8 giu 2018	8
11 giu 2018	8
Totale	72

Indietro

Termina tirocinio

© 2018 - Giacomo De Liberati

Figura 3.10 Tracciamento di un tirocinio avviato

3.2.9 Terminazione di un tirocinio

Deve essere possibile, una volta avviato un tirocinio, terminarlo.

Una volta individuato il tirocinio in questione, sarà sufficiente entrare nella pagina di dettaglio della candidatura e premere il pulsante "Termina candidatura". Da questo momento sarà possibile generare la documentazione associata al tirocinio eseguito.

Screenshots

The screenshot displays the 'Internship Manager' web application. On the left is a sidebar with the user 'Mario Rossi' and navigation links for HOME, TIROCINI, AZIENDA, and a list of internships. The main content area shows details for a specific internship, including 'Requisiti' (Requirements) like Redux, Ionic, and Node.js. Below this, the 'Stato corrente' (Current status) is shown as 'Data di inizio reale' (11 giu 2018) and 'Data di fine reale' (11 giu 2018). A 'Foglio presenze' (Attendance sheet) table follows, listing dates from June 4 to June 11, 2018, with a total of 72 hours worked. At the bottom, there are buttons for 'Indietro' (Back) and 'Genera documenti' (Generate documents). A copyright notice '© 2018 - Giacomo De Liberali' is visible in the bottom right corner.

Data	Ore lavorate
11 giu 2018	8
8 giu 2018	8
7 giu 2018	8
6 giu 2018	8
5 giu 2018	8
4 giu 2018	8
25 mag 2018	8
24 mag 2018	8
23 mag 2018	8
Totale	72

Figura 3.11 Terminazione di un tirocinio

3.2.10 Generazione documentazione

Deve essere possibile, una volta terminato un tirocinio, generarne la relativa documentazione precompilata in PDF..

Una volta individuato il tirocinio in questione, sarà sufficiente entrare nella pagina di dettaglio della candidatura e premere il pulsante "Genera documentazione". Verrà aperta una nuova finestra del browser contenente il documento PDF precompilato contenente i dettagli del tirocinio, dell'azienda, del professore, dello studente e la tabella del foglio presenze.

Screenshots

Resoconto tirocinio
Sviluppatore Angular 2+

Studente: Mario Bianchi
Professore: Mario Verdi
Azienda: ProxMox SRL

Dettagli tirocinio

Data inizio: 11 giu 2018
Data fine: 11 giu 2018
Ore lavorate: 72
Luogo di lavoro: Santa Maria di Sala

Foglio presenze

Le seguenti tabelle riportano le giornate lavorate e la quantità di ore.

Data	Ore lavorate
11 giu 2018	8
8 giu 2018	8
7 giu 2018	8

Data	Ore lavorate
6 giu 2018	8
5 giu 2018	8
4 giu 2018	8

Data	Ore lavorate
25 mag 2018	8
24 mag 2018	8
23 mag 2018	8



Firma professore

Firma azienda

Firma studente

11 giugno 2018 15:42

Figura 3.12 Documentazione generata dal sistema

Capitolo 4

API - Application Programming Interface

4.1 Autenticazione

Il sistema gestisce l'autenticazione degli utenti e le loro autorizzazioni utilizzando *JSON Web Token (JWT)*. Il *back-end* dell'applicazione espone delle *REST API state-less*, ovvero che non mantengono nessuna sessione sugli utenti autenticati, ma che si aspettano piuttosto di ricevere in ogni chiamata un *token* che contiene l'identificazione del chiamante.

Nella fase di login, quando un utente effettua l'accesso al proprio account, viene invocato il metodo *login* di *AuthController*, che ritorna un oggetto contenente:

- l'utente che ha effettuato l'accesso
- un flag booleano che indica se l'utente è nuovo o meno
- un *token JWT*, il cui contenuto è l'utente che ha eseguito il login, firmato con la chiave privata del server

Il *token* generato dovrà essere incluso in ogni chiamata autenticata che l'applicazione *front-end* desidera effettuare.

4.1.1 *Back-end*

Express.js fornisce la possibilità di utilizzare delle funzioni, chiamate *middleware*, prima che un metodo venga eseguito. Esse vengono eseguite in serie, permettendo di definire in funzioni atomiche controlli di validazione sulla chiamata in cui vengono innestate.

Tutti i *Controllers* dell'applicazione *Node.js* derivano da un *BaseController* che espone un metodo, *useAuth*.

```
1 /**
2  * Enable JWT token verification
3  */
4 public useAuth() {
5     this.router.use('*', AuthenticationController.AuthMiddleware);
6     return this;
7 }
```

Figura 4.1 Metodo *useAuth* di *AuthController*

Questo metodo registra un nuovo *middleware* che ha due funzioni:

1. verifica che la chiamata corrente contenga nell'*header HTTP* un *token* valido e non scaduto
2. aggiunge al parametro *req.body* una nuova proprietà che contiene il risultato della decodifica del token *JWT* (ovvero un oggetto *User*, con i relativi ruoli)

Il *middleware* di autenticazione viene eseguito in tutte le chiamate che vengono registrate dopo di lui. Il metodo *useAuth* deve venire infatti chiamato prima di tutti i metodi che utilizzano l'autenticazione e dopo di tutti quelli che non la utilizzano. Per esempio, in figura 2.4 il metodo *useAuth* è chiamato all'inizio, il che significa che tutti i metodi di *InternshipController* sono protetti da autenticazione.


```
1 /**
2  * The authentication middleware. Populate the request.body
3  * with the [[ServerDefaults.authUserBodyPropertyName]] property
4  * containing the token user
5  */
6 public static AuthMiddleware = async (req: Request,
7                                     res: Response,
8                                     next: Function) => {
9
10     const token = req.headers[ServerDefaults.jwtTokenHeaderName];
11     if (token) {
12         const isValid = AuthenticationController.ValidateToken(token);
13         // Is is valid proceed
14         if (isValid) {
15             req.body[ServerDefaults.authUserBodyPropertyName] =
16                                                         decode(token);
17             return next();
18         }
19         // return auth exception
20     } else {
21         // return missing token
22     }
23 }
```

Figura 4.2 *AuthMiddleware* di *AuthController*

Authentication scopes

Tutti i metodi registrati dopo l'utilizzo di *useAuth* vengono quindi eseguiti se e soltanto se la chiamata contiene un *token* valido. Durante la loro esecuzione esisterà dunque una proprietà che contiene l'oggetto *User* che ha effettuato la chiamata, e sarà quindi possibile permettere di arrestarne l'esecuzione se l'utente corrente non dovesse disporre del ruolo necessario per eseguirla.

Con questa filosofia ho creato dei *middleware* di autorizzazione predefiniti, chiamati *scopes*, che permettono di autorizzare solo una tipologia di utenti. Gli *scopes* inseriti all'interno dell'applicazione sono:

- *adminScope*: permette l'esecuzione se l'utente corrente contiene almeno il ruolo *Admin*
- *companyScope*: permette l'esecuzione se l'utente corrente contiene almeno il ruolo *Company*

- *studentScope*: permette l'esecuzione se l'utente corrente contiene almeno il ruolo *Student*
- *professorScope*: permette l'esecuzione se l'utente corrente contiene almeno il ruolo *Professor*
- *ownCompanyScope*: permette l'esecuzione se l'utente corrente è un amministratore dell'azienda su cui sta eseguendo l'operazione
- *ownInternshipScope*: permette l'esecuzione se l'utente corrente è proprietario del tirocinio su cui sta eseguendo l'operazione
- *ownInternshipProposalScope*: permette l'esecuzione se l'utente corrente è un soggetto (azienda, professore o studente) della proposta di tirocinio su cui sta eseguendo l'operazione

Gli *scopes* si possono anche combinare insieme in serie per ottenere autorizzazioni più articolate.

4.1.2 *Front-end*

Autenticazione nei *Services*

L'applicazione *Angular* deve ovviamente preoccuparsi di inserire nell'*header* (*HTTP*) di tutte le chiamate autenticate il *token* ricevuto al login, dal momento che il server non mantiene nessuna sessione per gli utenti autenticati. Questa logica è inserita all'interno del *BaseService* da cui tutti i *Services* derivano: questa classe espone dei metodi nominati con il *verb HTTP* che si intende utilizzare. Ha una dipendenza nei confronti di *AuthService* — che non deriva da *BaseService*, ma espone i metodi di login e registrazione — per il recupero del *token* corrente.

```
1 /**
2  * Make a GET request to the specified path
3  * @param path The path (baseUrl already included)
4  * @param options The options to pass to the HttpClient
5  */
6 protected getVerb<T>(path: string, options?: HttpOption):
7     Promise<ApiResponseDto<T>> {
8
9     const httpOptions = Object.assign({
10         headers: {
11             [Defaults.JwtHeaderName]: this.authService.token || ''
12         }
13     }, options || {});
14
15     return this.httpClient
16         .get(`${environment.apiServicesBaseUrl}/${path}`, httpOptions)
17         .toPromise();
18 }
```

Figura 4.3 Esempio di metodo esposto da *BaseService*

Autenticazione nei *NgModules*

Il modulo che contiene la porzione di applicazione visibile dopo l'autenticazione, *AuthModule*, per essere caricato con il *lazy-loading* richiede solamente che l'utente sia autenticato e disponga di un token valido. A questo livello non è ancora necessaria l'autorizzazione basata sui ruoli in quanto questo modulo e i suoi sotto moduli comprendono pagine condivise raggiungibili da utenti con ruoli differenti.

L'autorizzazione basata sui ruoli viene infatti applicata a livello di pagina: ad ogni componente registrato nel router di *Angular*, sono associati uno o più ruoli necessari per accedervi. Nel caso in cui il *token* non contenga un *claim* valido, l'utente viene reindirizzato in una pagina di errore 404.

4.2 Endpoints

Tutti i *controllers* dell'applicazione *Express.js* ritornano i risultati delle chiamate all'interno di un oggetto standard chiamato *ApiResponseDto* che contiene oltre al risultato dell'operazione delle altre informazioni, come lo stato *HTTP* ed eventuali errori. Tutti i *controllers* sono esposti sotto al path convenzionale */api*.

4.2.1 BaseController

Il *BaseController* espone di default i metodi *CRUD* dell'entità parametrica del *controller* che lo estende, in modo tale da non dover riscrivere codice duplicato. I metodi che rende disponibile ai figli sono esposti nella tabella 4.1.

Tabella 4.1 Endpoint *BaseController*

URL	Metodo	Parametri	Risposta	Scope
/<controllerName>	GET		Array<T>	
/<controllerName>/:id	GET	string	Array<T>	
/<controllerName>	POST	T	T	
/<controllerName>	PUT	T	T	
/<controllerName>/:id	DELETE	string	boolean	adminScope

Nel caso un *controller* derivato non intenda esporre alcuni di questi metodi, è sufficiente in fase di registrazione del *controller* stesso passare delle opzioni al metodo *useCrud(options: CrudOptions)*, specificando i metodi disabilitati oppure *scopes* aggiuntivi.

Il *BaseController* ha una dipendenza dall'applicazione di *Express.js*, in quanto deve averne accesso per registrare nuovi metodi.

4.2.2 InternshipsController

Questo *controller* gestisce le operazioni sui tirocini. Il suo path è `"/internships"`.

Dipendenze

- *internshipsRepository: InternshipsRepository*

Endpoint

Tabella 4.2 Endpoint *InternshipController*

URL	Metodo	Scope
/getByCompanyOwnerId/:ownerId	GET	
/getApproved	GET	
/getNotApproved	GET	
/status	PUT	professorScope
/status/force	PUT	adminScope
/status/:status	GET	

Gli ultimi tre metodi in tabella sono utilizzati per aggiornare lo stato di un tirocinio:

- PUT `"/status"`: permette di aggiornare lo stato di un tirocinio (es. da "Non approvato" ad "Approvato"). Dal momento che gli stati dei tirocini seguono una macchina a stati ben precisa, questo metodo verifica che il nuovo stato sia raggiungibile dallo stato corrente e che l'utente che ha effettuato la richiesta abbia il permesso di effettuare questa transizione. La definizione della macchina a stati è riportata in figura 4.4.

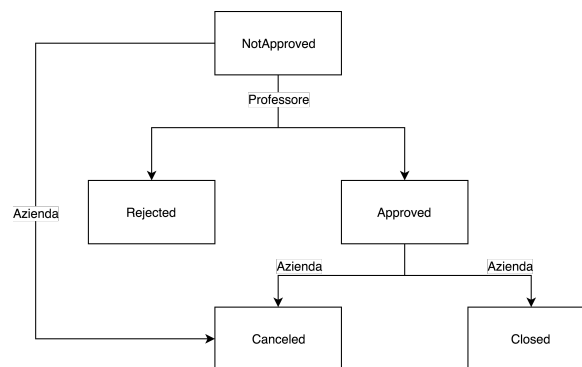


Figura 4.4 Macchina a stati di un tirocinio

Nel caso l'utente non abbia il permesso di effettuare la transizione viene ritornato un errore di autorizzazione, mentre se lo stato non è raggiungibile viene ritornato un messaggio di errore che riporta l'errata transizione.

- PUT `"/status/force"`: raggiungibile solamente da un utente amministratore, permette di aggiornare lo stato di un tirocinio senza il vincolo della macchina a stati di figura 4.4.
- GET `"/status/:status"`: ritorna tutti gli stati che l'utente corrente può raggiungere dallo stato passato come parametro.

4.2.3 InternshipProposalsController

Questo *controller* gestisce le operazioni sulle candidature ai tirocini. Il suo path è `"/proposals"`.

Dipendenze

- *internshipProposalsRepository*: *InternshipsProposalsRepository*
- *companiesRepository*: *CompaniesRepository*
- *internshipsRepository*: *InternshipsRepository*

Endpoint

Tabella 4.3 Endpoint *InternshipProposalsController*

URL	Metodo	Scope
/getByProfessorId/:professorId	GET	
/getByCompanyOwnerId/:companyOnwerId	GET	
/getByStudentId/:studentId	GET	
/availableplaces/:internshipId	GET	
/addAttendances	POST	ownInternshipProposalScope
/generateDocs/:internshipProposalId	GET	ownInternshipProposalScope
/status	PUT	ownInternshipProposalScope
/status/force	PUT	adminScope
/status/:status	GET	

Anche in questo caso gli ultimi tre metodi in tabella sono utilizzati per aggiornare lo stato di un tirocinio:

- PUT `"/status"`: permette di aggiornare lo stato di una candidatura (es. da "Iniziata" ad "Terminata"). Dal momento che gli stati di una proposta di tirocinio seguono una macchina a stati ben precisa, questo metodo verifica che il nuovo stato sia raggiungibile dallo stato corrente e che l'utente che ha effettuato la richiesta abbia il permesso di effettuare questa transizione. La definizione della macchina a stati è riportata in figura 4.5.

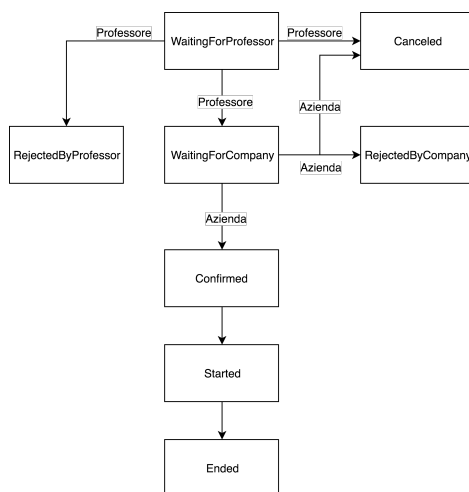


Figura 4.5 Macchina a stati di una proposta di tirocinio (candidatura)

Nel caso l'utente non abbia il permesso di effettuare la transizione viene ritornato un errore di autorizzazione, mentre se lo stato non è raggiungibile viene ritornato un messaggio di errore che riporta l'errata transizione.

- PUT "/status/force": raggiungibile solamente da un utente amministratore, permette di aggiornare lo stato di un tirocinio senza il vincolo della macchina a stati di figura 4.5
- GET "/status/:status": ritorna tutti gli stati che l'utente corrente può raggiungere dallo stato passato come parametro.

4.2.4 RolesController

Questo *controller* gestisce le operazioni sui ruoli degli utenti. Il suo path è "/roles".

Dipendenze

- *rolesRepository*: *RolesRepository*

Endpoint

Non ha metodi custom, ma espone solamente le operazioni *CRUD* offerte da *BaseController*.

4.2.5 UsersController

Questo *controller* gestisce le operazioni sugli utenti. Il suo path è "/users". Limita l'utilizzo del metodo di aggiornamento ed eliminazione solamente agli utenti con il ruolo di amministratore, per impedire modifiche non autorizzate agli utenti.

Dipendenze

- *usersRepository*: *UsersRepository*

Endpoint

Tabella 4.4 Endpoint *UsersController*

URL	Metodo	Scope
/getByRole/:role	GET	
/own	PUT	ownUserScope
/professors/lookup	POST	

4.2.6 CompaniesController

Questo *controller* gestisce le operazioni sulle aziende. Il suo path è `"/companies"`.

Dipendenze

- *companiesRepository*: *CompaniesRepository*

Endpoint

Tabella 4.5 Endpoint *CompaniesController*

URL	Metodo	Scope
<code>/getByOwnerId/:ownerId</code>	GET	

4.2.7 AuthenticationController

Questo *controller* gestisce le operazioni di login e registrazione degli utenti. Il suo path è `"/auth"`. Non deriva da *BaseController* in quanto non esporta operazioni sulle entità, ma si preoccupa solamente di verificare l'identità degli utenti e di generare un *token* di accesso. Tutti i suoi metodi sono, per ovvie ragioni, privi di autenticazione e liberamente accessibili.

Dipendenze

- *usersRepository*: *UsersRepository*
- *rolesRepository*: *RolesRepository*

Endpoint

Tabella 4.6 Endpoint *AuthenticationController*

URL	Metodo	Scope
<code>/login</code>	POST	
<code>/register</code>	POST	
<code>/google</code>	GET	
<code>/token/validate</code>	GET	
<code>/token/decode</code>	GET	(solo debug)

Capitolo 5

Conclusioni

L'applicazione, sebbene soddisfi i casi d'uso previsti, è da considerarsi come una versione non ancora in grado di ricoprire tutti gli scenari realmente esistenti nel processo di gestione dei tirocini. Quello che potrebbe completare *DAIS Internship Manager* è l'integrazione dell'autenticazione con il sistema federato dell'ateneo e l'aggiunta del ruolo utente "Garante" — che potrebbe convergere con il ruolo di professore — che si occupa di accertare che il tirocinio richiesto da uno studente possa permettere il riconoscimento dei crediti. In questo momento, infatti, la candidatura ad un tirocinio è da considerarsi come extra-curricolare. Per permettere il riconoscimento di crediti formativi è necessario che l'azienda sia convenzionata con l'ateneo e che il contenuto del tirocinio sia valido e inerente al percorso di studi. Per questo si potrebbe pensare di creare una sezione dell'applicazione che permetta di convenzionare le aziende interessate, e infine di permettere al professore di verificare che il piano dell'azienda proposto per il tirocinio sia sufficiente al riconoscimento dei crediti richiesti dallo studente.

5.1 Sviluppi futuri e nuove integrazioni

I possibili sviluppi futuri dell'applicazione potrebbero dunque coinvolgere:

- la gestione di stage anche curriculari, permettendo il riconoscimento dei crediti formativi universitari (CFU)
- l'autenticazione mediante il sistema federato dell'ateneo, permettendo un controllo degli utenti più restrittivo
- la possibilità di convenzionare le aziende online, favorendo il numero di imprese che interagiscono con l'ateneo

Bibliografia

- [1] Automattic. Mongodb object modeling designed to work in an asynchronous environment, 2018. URL <https://github.com/Automattic/mongoose>.
- [2] Node.js Foundation. Node.js, 2018. URL <https://nodejs.org/it/>.
- [3] Wikimedia Foundation. Node.js - wikipedia, 2018. URL <https://en.wikipedia.org/wiki/Node.js>.
- [4] Google. Angular - architecture overview, 2018. URL <https://angular.io/guide/architecture>.
- [5] MongoDB Inc. Mongodb architecture, 2018. URL <https://www.mongodb.com/mongodb-architecture>.
- [6] MongoDB Inc. Sql to mongodb mapping chart, 2018. URL <https://docs.mongodb.com/manual/reference/sql-comparison/>.
- [7] Inversify. A powerful and lightweight inversion of control container for javascript and node.js apps powered by typescript, 2018. URL <https://github.com/inversify/InversifyJS>.
- [8] Jakesgordon. A javascript finite state machine library, 2018. URL <https://github.com/jakesgordon/javascript-state-machine>.
- [9] Ngx-translate. The internationalization (i18n) library for angular, 2018. URL <https://github.com/ngx-translate/core>.

Arconimi

API Application Programming Interface. vi, 2, 5, 11, 15, 37, 38, 40, 42, 44, 46, 49, *Glossario: Application Programming Interface*

BSON Binary *JSON*. vii, 2, 3, 49, *Glossario: Binary JSON*

CRUD Create, Read, Update e Delete. 13, 15, 42, 45, 49, *Glossario: Create, Read, Update e Delete*

DBMS Database Management System. 2, 49, *Glossario: Database Management System*

DI Dependency injection. 8, 14, 17, 49, *Glossario: Dependency injection*

DOM Document Object Model. 7, 8, 49, *Glossario: Document Object Model*

EDA Event-driver architecture. 5, 49, *Glossario: Event-driver architecture*

HTML HyperText Markup Language. 7, 49, *Glossario: HyperText Markup Language*

HTTP Hyper Text Transfer Protocol. 6, 15, 38, 40, 41, 49, *Glossario: Hyper Text Transfer Protocol*

JSON Javascript Object Notation. vii, 2, 3, 20, 37, 49, *Glossario: Javascript Object Notation*

JWT *JSON* Web Token. 37, 38, 49, *Glossario: JSON Web Token*

MEAN MongoDB, Express, Angular & Node.js. iii, 2, 49, *Glossario: MongoDB, Express, Angular & Node.js*

MVC Model-View-Controller. 6, 49, *Glossario: Model-View-Controller*

NoSQL No Structured Query Language. 2, 3, 49, *Glossario: No Structured Query Language*

npm Node Package Manager. 11, 12, 14, 20, 49, *Glossario*: Node Package Manager

Observer Observer pattern. 5, 49, *Glossario*: Observer pattern

OOP Object-oriented programming. 3, 49, *Glossario*: Object-oriented programming

ORM Object-relational mapping. 11, 12, 49, *Glossario*: Object-relational mapping

REST Representational State Transfer. 2, 11, 15, 37, 49, *Glossario*: Representational State Transfer

SPA Single Page Application. 2, 6, 49, *Glossario*: Single Page Application

URL Uniform Resource Locator. 6, 8, 49, *Glossario*: Uniform Resource Locator

Glossario

JSON Web Token Un open-standard basato su *JSON* per la creazione di token di accesso che contengono un certo numero di *claims*. 37, 49

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. vi, 37, 38, 40, 42, 44, 46, 49

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*. vi, vii, 11–13, 17, 19, 37, 49

Binary JSON Una rappresentazione binaria di *JSON*. 2, 49

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, 16, 49

Create, Read, Update e Delete Rappresentano le operazioni di base per un modello di memorizzazione permanente. 13, 49

Database Management System Un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database. 2, 49

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, 14, 17, 49

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, 49

Event-driver architecture Un pattern di architettura software che promuove la produzione, l'individuazione, il consumo e la reazione agli eventi. 49

Event-loop Un costrutto di programmazione che attende e invia eventi o messaggi in un programma . 5, 6, 49

Framework Fornisce un modo standard per creare e distribuire applicazioni . 2, 5, 6, 11, 49

Front-end Si intende la parte di applicazione visibile all'utente finale, che nasconde i dettagli implementativi . vi, vii, 2, 6, 11, 18–20, 37, 40, 49

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*. 49

HyperText Markup Language Un linguaggio di markup per la formattazione e impaginazione di documenti ipertestuali disponibili nel web. 49

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, 49

Javascript Object Notation Una rappresentazione testuale che permette di codificare un oggetto *javascript*. 2, 49

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, 49

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, 49

MongoDB, Express, Angular & Node.js Uno stack di tecnologie utilizzate insieme al fine di creare applicazioni web. iii, 49

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, 49

Node Package Manager Un package manager per *javascript*, è quello di default per il runtime di *Node.js*. 11, 49

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. 49

Object-relational mapping Una tecnica di programmazione per convertire dati tra sistemi incompatibili utilizzando linguaggi *Object-oriented programming (OOP)*. 12, 49

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. 49

Representational State Transfer Un'architettura software per trasmettere dati su *HTTP* in modo state-less. 49

Single Page Application Un'applicazione web che può essere utilizzata su una singola pagina web con l'obiettivo di fornire una esperienza utente più fluida e simile alle applicazioni desktop dei sistemi operativi tradizionali. 6, 49

Typescript Un linguaggio programmazione open-source sviluppato di Microsoft, super-set di *javascript*, estende la sintassi di *javascript* introducendo lo static typing . 6, 49

Uniform Resource Locator Rappresenta una referenza ad una risorsa web. Specifica la sua posizione nella rete e il meccanismo per recuperarla. 49