Dipartimento di Ingegneria e Scienza dell'Informazione
2022/2023

YINCO

Sviluppo applicazione

Doc. Name	D4-Sviluppo di Yinco	Doc. Number	Rev 1.0
Description	Il documento include la spiegazione dello sviluppo del sistema Yinco		

INDICE

1. Scopo del documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte dell'applicazione Yinco. In particolare, presenta tutti gli strumenti necessari per realizzare i servizi di gestione delle domande proposte dagli utenti al sistema.
Partendo dalla descrizione degli user flows legati all'utilizzo del sistema da parte di un utente, il documento prosegue con la presentazione delle API tramite l'API Model e il Modello delle risorse.
Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati.
Verrà inoltre presentata un'interfaccia grafica con la quale l'utente potrà sfruttare tutte le potenzialità dell'applicazione.
Infine una sezione è dedicata alle informazioni del Git Repository e al deployment dell'applicazione stessa.

2. User flows

In questo paragrafo verranno mostrati ed analizzati due casi di user flows:

- un caso in cui un'utente, autenticato o anonimo che sia, usa il sistema Yinco per tentare di ottenere un'informazione e per navigare attraverso le altre pagine usufruendo delle loro funzioni;
- un caso in cui invece un'utente vuole solo passare da una pagina all'altra, senza effettuare alcuna azione.

Quest'ultimo caso, per quanto possa sembrare strano, è necessario sia per poter mostrare la relazione tra le varie pagine, sia perché non è escluso il caso in cui un utente voglia solo visionare il sistema per poi decidere se usufruirne in futuro oppure no.
Di seguito è possibile vedere la legenda dei simboli utilizzati nei due user flows.

Figura 2.1: Legenda dei simboli usati nei vari user flows

Alcune precisazioni sulla legenda:

- per **bivio** si intende un punto in cui la risposta del sistema cambia a seconda dell'azione dell'utente;
- per **attesa** si intende un istante in cui il sistema ha terminato di eseguire la funzione richiesta e delega quindi all'utente la scelta di cosa fare: se ripetere il processo appena concluso -se possibile, ovviamente- oppure tornare ad una pagina precedente ed effettuare ulteriori azioni;

- per **arrivo** si intende il momento in cui l'utente ha soddisfatto il motivo per cui si è interfacciato con Yinco. Nel caso del nostro sistema, esso è rappresentato dall'ottenimento dell'informazione richiesta.

2.1 Utente intenzionato ad usare il sistema

Il primo caso da analizzare è quello di un utente che vuole usufruire di tutte le funzioni del sistema.
La pagina da cui l'utente inizierà a visitare il sito sarà sempre **l'homepage**, da cui ha diverse possibilità di scelta; può effettuare il login, accedere alla pagina impostazioni, alla pagina contatti, ricaricare la homepage tramite il bottone "home" e ha due modi per poter arrivare alla pagina dedicata alla ricerca: o cliccando il bottone "Inizia!" che gli si parerà davanti alla fine della homepage, oppure cliccando il pulsante "Chatbot".
Se l'utente decide di visionare la pagina **contatti**, l'unica cosa che potrà fare poi sarà tornare indietro, in quanto la pagina Contatti non gli fornisce nessuna altra azione da effettuare.
Se l'utente decide di **autenticarsi**, verrà indirizzato alla pagina di login in cui potrà inserire le proprie credenziali. Se queste sono corrette, allora il sistema effettuerà il login e l'utente verrà riportato all'ultima pagina visitata, altrimenti verrà ricondotto di nuovo alla pagina di login, dove un avviso gli dirà che le sue credenziali sono errate.
Nel caso in cui l'utente decida di **restare nella homepage** cliccando il tasto "home", semplicemente l'utente rimarrà fermo nella homepage fino a quando non deciderà di effettuare un'altra azione.
Se l'utente vuole visionare la pagina "**Impostazioni**", premendo il tasto "Impostazioni", presente nella barra di navigazione, arriverà ad una pagina dove avrà la possibilità di effettuare tre azioni:

1) cambiare la propria preferenza mail,
2) cambiare la lingua del sistema
3) effettuare il logout.

Per cambiare la propria preferenza mail e per effettuare il logout, l'utente deve essere autenticato, altrimenti verrà di nuovo inviato alla pagina di impostazioni con un messaggio di errore. Qualsiasi utente può invece cambiare la lingua del sistema tramite l'apposita opzione.
Se l'utente vuole invece **accedere alla pagina della chatbot**, qui avrà la possibilità di digitare la domanda e di inviarla al sistema, il quale gli restituirà un errore, in base alle caratteristiche descritte nel Documento di Specifica dei Requisiti (o D2)[1], oppure un link a cui potrà trovare l'informazione trovata.
Qui l'utente potrà scegliere se

1) aprire il link che la chatbot gli ha ritornato;
2) porre ad essa una nuova domanda.

Nel primo caso, l'utente accederà alla pagina contenente le informazioni richieste – sia essa una pagina creata dal sistema oppure una pagina di WebApps UniTn – e la richiesta dell'utente potrà essere considerata soddisfatta, ponendo fine all'utilizzo

[1] Cfr *Documento di Specifica dei Requisiti (D2), paragrafo 2, sezione "Interagire con il sistema"*

dell'applicazione da parte dell'utente nel caso in cui questi non abbia altre azioni da voler intraprendere nell'ambito del sistema.
Nel secondo caso si ripeterà il processo di richiesta di informazioni.
Lo user flow appena descritto è graficamente visibile nella figura sottostante.

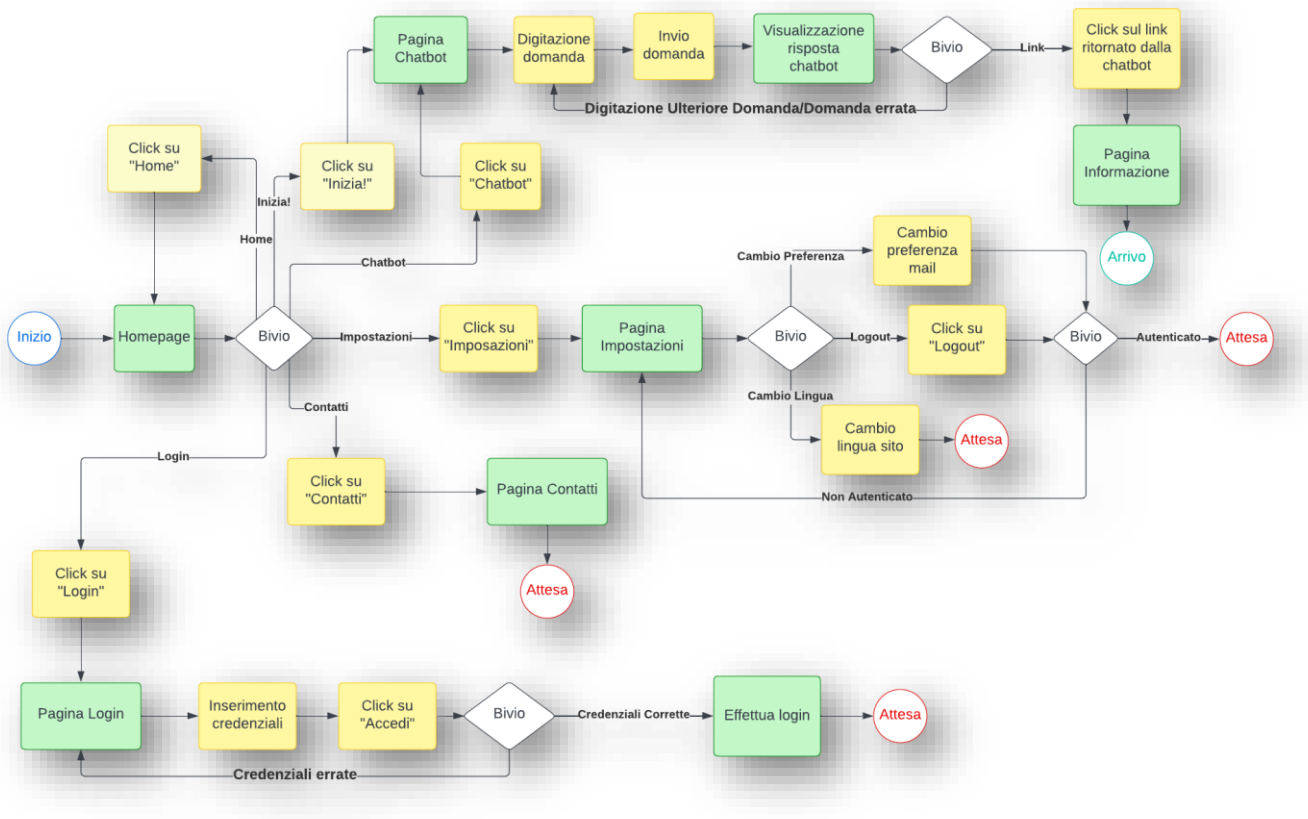

Figura 2.2: User Flow di un utente che vuole usufruire di tutte le informazioni del sistema e che vuole cercare informazioni

2.2 Flow delle pagine per l'utente

Il secondo caso da analizzare è quello di un utente che vuole solo visionare tutte le pagine, senza effettuare nessuna azione.
Questo user flow è utile per descrivere il flusso delle varie pagine del sistema e come sono interconnesse tra loro, cosa che sarebbe potuta essere aggiunta nello user flow precedente, ma che avrebbe diminuito la leggibilità dello stesso, motivo per cui ne è stato creato uno apposito.

In generale, da ogni pagina è possibile accedere a tutte le altre.
Ciò non è però vero per la pagina di login, l'unica dalla quale non è possibile accedere a nessuna altra pagina se non dopo aver effettuato il login, il quale riporterà l'utente alla pagina da cui si è avuto accesso alla funzione di login.

Un altro caso particolare è quello della homepage, in quanto è l'unica pagina da cui è possibile arrivare alla chatbot in due modi diversi[2].
Per questo motivo, le pagine di contatti, impostazioni e chatbot hanno un loro specifico bivio, pressoché uguale a quello della homepage, ma senza la possibilità di accedere alla homepage tramite il pulsante "Inizia!".
Tale User Flow è visibile graficamente qui sotto.

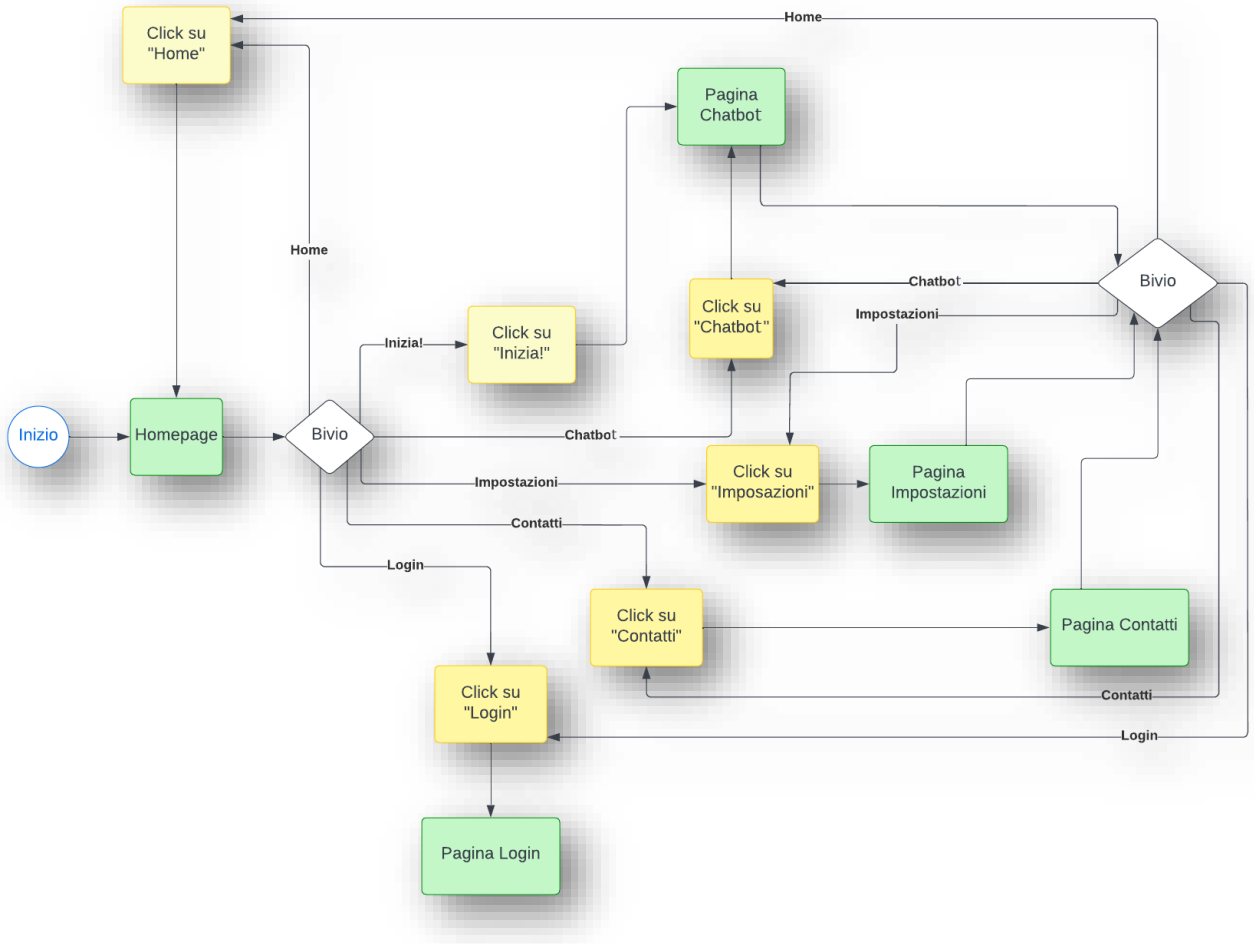

Figura 2.3: User flow che illustra l'interconnessione tra le varie pagine del sistema

[2] Cfr *Documento di Sviluppo dell'Applicazione (D4), paragrafo 2.1*

3. Application implementation and documentation

Nei capitoli precedenti abbiamo identificato le caratteristiche della nostra web app e tutte le operazioni che potrà svolgere, per dare un'idea di come funzionerà. Ora entriamo nello specifico e vediamo come è stata implementata. Abbiamo utilizzato NodeJS, HTML, CSS e Javascript per lo sviluppo vero e proprio dell'applicazione, mentre per la gestione dei dati è stato utilizzato MongoDB.

3.1 Project Structure

La struttura del progetto è visibile attraverso la figura sottostante 3.1.

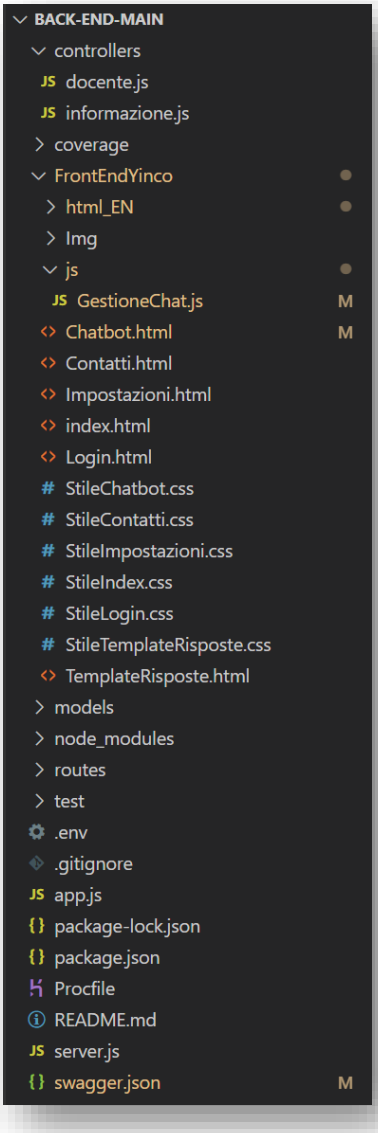

Figura 3.1: Struttura del codice

Per quanto riguarda il FrontEnd, quest'ultimo si trova tutto all'interno della cartella "**FrontEndYinco**", dove sono salvati tutti i file HTML e CSS.
Inoltre abbiamo:

- una cartella "js", all'interno della quale è salvato il file Javascript per la gestione della chat;
- una cartella "Img", nella quale sono salvate le immagini usate;
- una cartella "html_EN", in cui possiamo trovare gli stessi file della cartella "FrontEndYinco", con le opportune modifiche in inglese.

Per il BackEnd sono state sviluppate alcune API locali che, come si può notare, sono salvate nella cartella "**controllers**". La cartella "**models**" contiene i dettagli di come sono stati definiti gli oggetti di tipo Docente e di tipo Informazione contenuti in MongoDB, la cartella "**test**" contiene i vari test delle API e delle funzioni effettuati con Jest[3] e, infine, la cartella "**routes**" definisce il tipo delle API sviluppate e dove può essere trovata la loro implementazione.

3.2 Project Dependencies

I moduli Node utilizzati nel progetto e, di conseguenza, aggiunti nel file "package.json" sono i seguenti:

- Swagger
- Multer
- Mongoose
- Express

3.3 Project Data or DB

Per la gestione delle informazioni abbiamo definito due strutture principali, come si vede dalla figura 3.2.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
Docenti	10	1.1KB	113B	36KB	1	20KB	20KB
Informazioni	8	49.52KB	6.19KB	76KB	1	36KB	36KB

Fig. 3.2: Collezioni dei dati

Abbiamo creato una collezione "**Docenti**" per salvare le informazioni dei professori, la cui struttura è la seguente:

[3] Il seguente argomento verrà trattato in dettaglio nel paragrafo 7 del seguente documento.

```
_id: ObjectId('639ad74b8bcc02436be46ea6')
cognome: "Bucchiarone"
url: "https://webapps.unitn.it/du/it/Persona/PER0053043/Didattica"
```

Fig. 3.3: Tipo di dato "Docente"

Per quanto riguarda le informazioni generali che uno studente potrebbe voler sapere, abbiamo creato una collezione "**Informazioni**", all'interno della quale i dati sono salvati, sia in italiano che in inglese, nel seguente modo:

```
_id: ObjectId('63988859246d499d4829d11f')
title: "Tasse anno accademico 2022/2023"
year: 2022
body: "Importi tasse e simulatore
      Gli importi delle tasse e dei contributi un…"
tags: "tasse,rate,contributi"
Privilege_level: 0
```

Fig. 3.4: Tipo di dato "Informazione" in italiano

```
_id: ObjectId('63a87eadc82ffd2c230e9e86')
title: "Academic year 2022/2023 taxes"
year: 2022
body: "Tuition and simulator amounts
      The amounts of tuition and university co…"
tags: "taxes, fees, contributions"
Priviledge_level: 0
```

Fig. 3.4: Tipo di dato "Informazione" in inglese

3.4 Project APIs

3.4.1 Resources extraction from the class diagram

Analizzando il **Class Diagram**, siamo arrivati alla conclusione che le API utilizzate dal nostro sistema sono quelle visibili nella figura 3.5, però, una volta iniziata l'implementazione del front-end, ci siamo accorti che alcune API potevano essere

implementate direttamente in quest'ultimo. Per questa ragione, le API che il sistema utilizzerà effettivamente sono quelle presenti nell'immagine 3.6.
Di queste abbiamo deciso di implementarne solo alcune.
Le API che abbiamo effettivamente implementato sono le API della classe '**Ricerca**', quindi le API:

- '**docente**'
- '**database**'

L'API '**docente**' è una GET, cioè utilizza il metodo GET per recuperare la rappresentazione di una risorsa, nel nostro caso l'url della pagina WebApps.unitn del docente.
L'API '**informazione**, anch'essa una GET, è la risorsa che ritorna un'informazione richiesta in base alle keyword inserite dall'utente.
Nelle API 'informazione' e 'docente', il dato di partenza su cui si basano le loro funzionalità è una **keyword**.
Le API non implementate della classe 'ricerca' sono 'privilegi', anch'essa è una GET, la quale ritorna se l'utente è attualmente autenticato o no, e link, una GET anch'essa, la quale prende in input il dato ritornato dall'API "informazione" e ritorna all'utente un link ad una pagina contenente le informazioni così ritornate.
Analizzando anche il resto del diagramma notiamo che sono presenti altre classi con le loro relative API.
L'API della risorsa '**Utente_Anonimo**' è '**login**': è una GET, e permette all'utente anonimo di diventare un utente autenticato. Questa API prende come parametri l'email e la password dell'utente.
Le API della classe '**Utente_autenticato**' sono '**logout**' e '**login_automatico**', ambedue dei metodi di tipo GET. L'API 'logout' permette all'utente autenticato di poter diventare un utente anonimo e l'API '**login_automatico**' permette ad un utente di accedere automaticamente.
Per fare ciò l'API necessita dell'informazione '**tempo_trascorso_ultimo_accesso**', in quanto l'utente può effettuare il login automatico se e solo se il tempo passato dall'ultimo logout è inferiore a 15 minuti. I dati di partenza su cui si basa questa funzionalità sono l'email e la password dell'utente, le quali verranno poi ad essere prese da MongoDB attraverso il metodo GET.
La risorsa '**impostazione**' ha come sua unica API '**notifiche**'. Questa API permette ad un utente autenticato di poter aggiornare la propria preferenza notifiche, la quale di default viene disattivata.
Dovendo semplicemente aggiornare una risorsa il metodo di questa API è PATCH.
L'API della risorsa '**mail**' è '**send_mail**' di tipo POST. Questa API si interfaccia con le API 'tasse' ed 'esami' della risorsa ESSE3, in quanto il body della mail conterrà informazioni su ciò che verrà ritornato da queste due API.
Il metodo di 'send_mail' è POST, in quanto viene creata una nuova risorsa, cioè la mail che dovrà essere inviata.
La risorsa '**ESSE3**' ha a sua volta due API: '**tasse**' e '**esami**'. Entrambe sono di tipo GET e rispettivamente servono ad ottenere da ESSE3 informazioni relative alle tasse e agli esami.

Fig. 3.5: Diagramma con tutte le API

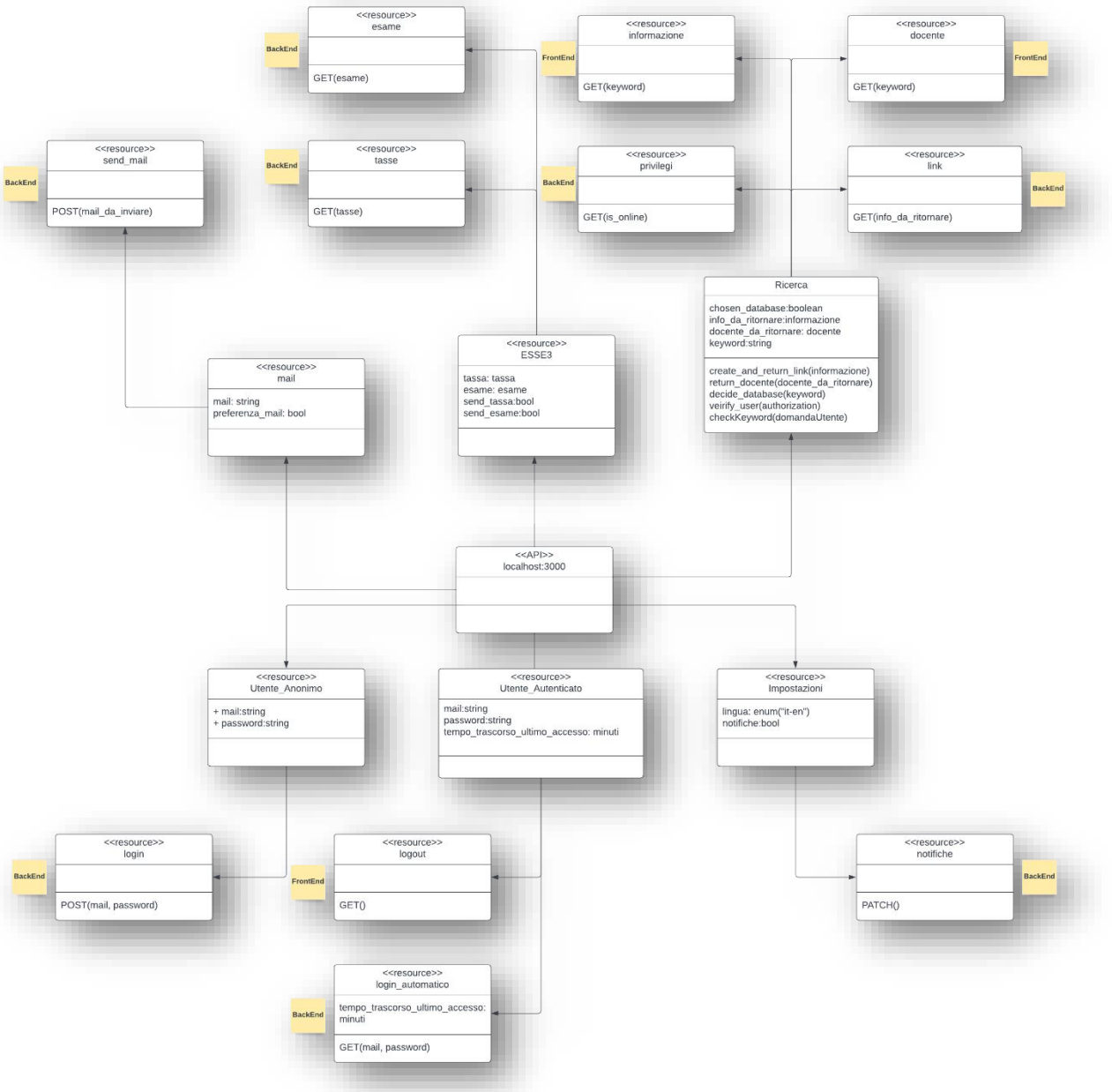

Fig. 3.6: Diagramma con le API implementate

3.4.2 Resources models

Anche nel **Resource Model Diagram** abbiamo descritto delle API che il sistema utilizza, ma che abbiamo deciso di non implementare.
Le due API che abbiamo deciso di implementare sono quelle di ricerca '**docenti'** e '**informazioni'**. Entrambe sono del tipo GET e hanno come URI '**ricerca'**.
Alle due API viene passato come body request la risorsa '**ricerca'** la quale ha come parametri '**keyword'** di tipo stringa, '**is_online'** di tipo booleano e '**chosen_database'**, anch'esso di tipo booleano.
L'attributo 'chosen_database' è stato deciso di inizializzarlo come booleano, in quanto, avendo solo due database sui quali possiamo andare a reperire le informazioni, abbiamo individuato questa come soluzione più logica, in quanto

abbiamo attribuito 'false' al database dei docenti e 'true' al database delle informazioni.
Le response body di queste API possono essere quattro:

1. **'found'**
2. '**not_found'**
3. '**unathorized**'
4. '**forbidden**'

ovvero:

1. **'found'**: l'API è riuscita a trovare ciò che le è stato richiesto, e il codice HTTP che ritorna è 200 OK;
2. '**not_found'**: l'API o non è riuscita a trovare l'informazione o il docente richiesto, e il codice HTTP che ritorna è 404 NOT FOUND;
3. '**unauthorized'**: l'informazione richiesta non può essere ritornata in quanto per accedervi bisogna essere un utente autenticato (cfr D1 obiettivi), e il codice HTTP che ritorna è 401 UNAUTHORIZED.
4. **'forbidden'**: l'API non permette agli utenti di poter accedere a quelle specifiche informazioni, in quanto sono in manutenzione da parte degli amministratori del sistema. Il codice http che deve essere ritornato in questa situazione è 403 FORBIDDEN.

In tutti e quattro i casi viene inviato un messaggio all'utente.
Le altre API presenti nel Resource Model sono '**login_automatico**' e '**logout'**, entrambe di tipo GET, '**login'**, di tipo POST, e '**notifiche'**, di tipo PATCH.
L'API 'login' ha come body request la risorsa 'utente', la quale ha come parametri mail e password, (tipo: stringa).
La reponse body che questa API può ritornare è una tra le seguenti:

- '**access'**: la mail e la password inserite dall'utente sono corrette, dunque il login è stato effettuato;
- '**error'**: la mail, o la password, inserite, sono errate;
- '**server_error'**; è presente un errore nel server dell'università.

L'API '**login_automatico'**, anch'essa di tipo GET, ha come body request la risorsa 'utente', la quale ha come parametri mail, password e tempo_trascorso_ultimo_accesso.
Mail e password sono di tipo string, mentre tempo_trascorso_ultimo_accesso è di tipo int.
Le responde body fornite da questa API sono due:

- '**access'**: il login_automatico è stato effettuato, ed ha come codice HTTP 200 OK;
- **'error'**: il login non è stato effettuato, e questo viene fatto notare all'utente attraverso un messaggio di tipo stringa. Il codice HTTP di questa response body è 400 BAD REQUEST.

L'API '**logout'** ha come body request la risorsa 'utente', la quale ha come parametri 'is_online', di tipo booleano, e 'mail', di tipo stringa.
Le responde body di questa API sono:

- **'logout'**: il logout è avvenuto correttamente, e il codice di questa response è 200 OK;

- **'error'**: c'è stato un errore durante l'operazione di logout;
- **'unauthorized'**: un utente non autenticato ha cercato di effettuare il logout.

L'API '**notifiche'** ha come request body la risorsa 'utente', la quale ha come parametri 'is_online', di tipo booleano, 'mail', di tipo stringa, e 'preferenza_notifiche', anch'essa di tipo stringa.
Sono due le response body di questa API:

- '**updated'**: la richiesta di aggiornare la propria preferenza notifiche è stata aggiornata: il codice HTTP è 200 OK.
- '**unauthorized'**: un utente non autenticato ha cercato di aggiornare la propria preferenza mail e il codice HTTP è 401 UNAUTHORIZED.

Entrambe queste risposte ritornano all'utente un messaggio di tipo stringa.
L'API '**send_mail**' ha come request body la risorsa 'utente' la quale ha come parametri 'mail', di tipo stringa, e 'preferenza_notifiche', di tipo booleano.
Sono quattro le response body di questa API:

- '**mail**': l'email da inviare è stata creata, e ha come suo unico parametro 'body', di tipo stringa: il codice HTTP è 201 CREATED;
- '**error**': pur essendo stata creata la mail, questa non è stata inviata: può essere dovuto al fatto che la mail non sia stata inviata correttamente. Il codice HTTP è 400 BAD REQUEST;
- '**server_error**': c'è stato un errore sul server di gmail, software utilizzato per poter inviare la mail: il codice HTTP è 500 SERVER ERROR.

Queste API ritornano un messaggio di tipo stringa.
Per concludere, le ultime due API che compongono il diagramma di Resources Model sono '**tasse**' ed '**esami**'.
Entrambe hanno come request body la risorsa 'utente', la quale ha come suoi parametri 'mail' e 'password', entrambi di tipo stringa.
Le request body dell'API **'tasse'** sono tre:

- '**found**': l'informazione 'tasse' è stata trovata, e ciò che viene ritornato è l'informazione 'tassa' di tipo tassa: il codice HTTP è 200 OK;
- '**not_found**': c'è stato un errore sul server di ESSE3, software utilizzato per poter accedere all'informazione: il codice HTTP è 500 SERVER ERROR.
- '**server_error**': l'informazione riguardante le tasse non è stata trovata: il codice HTTP è 404 NOT FOUND.

Sia in 'not_found' che in 'server_error' viene ritornato un messaggio di tipo stringa.
L'API **'esami'** ha anch'essa tre response body:

- '**found**': l'informazione 'esame' è stata trovata, e il codice HTTP è 200 OK; ciò che viene ritornato è l'informazione 'esame' di tipo esame;
- '**not_found**': c'è stato un errore sul server di ESSE3, software utilizzato per poter accedere all'informazione esame; il codice HTTP è 500 SERVER ERROR;
- '**server_error**' : l'informazione riguardante le tasse non è stata trovata; il codice HTTP è 404 NOT FOUND.

Sia in 'not_found' che in 'server_error' viene ritornato un messaggio di tipo stringa.

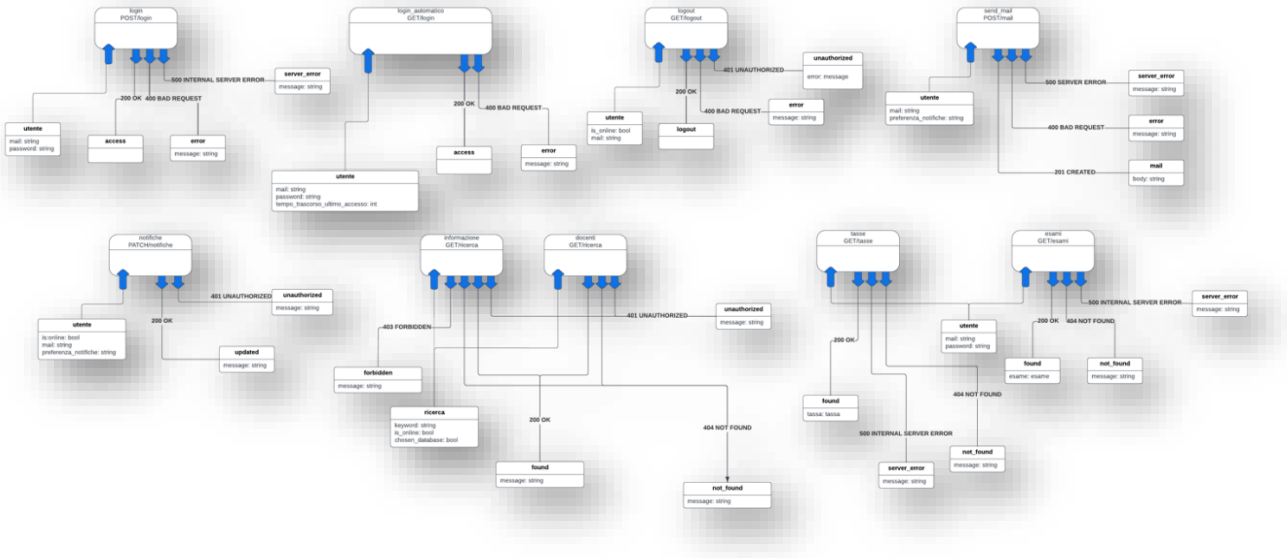

Fig. 3.7: Diagramma del resources model

3.5 Sviluppo API

Le API implementate nel programma sono le API ‘docente’ e ‘informazione’.

3.5.1 Ricerca docente

La funzione utilizzata da questa API ha come compito quello di trovare un docente in MongoDB in base al cognome inserito dal docente: nel caso in cui questo venga trovato, la funzione restituisce l’**url** della pagina Webapps.unitn del docente, in caso contrario restituisce un messaggio d’errore.
La funzione richiede che si utilizzi il modello ‘docente’, presente nella cartella ‘models’, questo perché il cognome del docente verrà estratto dal modello e utilizzato come variabile, che poi dovrà essere ricercata in MongoDB.
Nella funzione ‘**getOneDocente**’ viene utilizzato anche il metodo ‘**findOne**’ della libreria ‘mongoose’, il quale prende come parametro il cognome del docente.
Nel caso il ‘docente’ sia stato trovato, l’API ritorna il codice 200 OK, in caso contrario, ritorna il codice 404 NOT FOUND.
Per finire, questa funzione viene esportata, in quanto dovrà essere poi richiamata in ‘routes’.

```
const docente = require('../models/docente');

const getOneDocente = async(req, res) => {
    let surname = req.query.cognome;
    console.log(surname);
    let cognome = await docente.findOne({ cognome: req.query.cognome }, {url: 1}, (err, data) => {
        if (err || !data){
            return res.status(404).json("Il docente non è presente");
        }
        else {
            return res.status(200).json(data);
        }
    }).clone().catch(function(err){console.log(err)});
};
module.exports = {
    getOneDocente,
};
```

Fig. 3.8: API ricerca docente

3.5.2 Ricerca informazioni

La funzione utilizzata da questa API ha come compito quello di trovare un'informazione in MongoDB, in base ad uno o più tags inseriti dall'utente: nel caso in cui venga trovata un'informazione con i tags di ricerca, la funzione restituisce il titolo, l'anno ed il body dell'informazione, mente, in caso contrario, restituisce un messaggio d'errore.
La funzione utilizzata da questa API richiede che si utilizzi il modello 'informazioni', presente nella cartella 'models', questo perché i 'tags' dell'informazione verranno estratti dal modello, e utilizzati come variabile, che poi dovranno essere ricercate in MongoDB.
Nella funzione '**getOneInfo**' viene utilizzato anche qui il metodo '**findOne**' della libreria 'mongoose', il quale prende come variabile da ricerca i tags dell'informazione.
Nel caso l'informazione venga trovata, l'API ritorna il codice 200 OK mentre, in caso contrario, ritorna il codice 404 NOT FOUND.
Per finire, questa funzione viene estratta, in quanto dovrà essere poi richiamata in 'routes'.

```
const informazione = require('../models/informazione');

const getOneInfo = async (req, res) => {
    let tags = req.query.tags; //prende le tag dell'info
    console.log(tags);
    let information = await informazione.findOne({tags: {$regex:req.query.tags}}, {title:1, year:1, body:1}, (err, data) => {
        if (err || !data){
            return res.status(404).json("L'informazione non è presente");
        }
        else {
            return res.status(200).json(data);
        }
    }).clone().catch(function(err){console.log(err)});
};
module.exports = {
    getOneInfo,
};
```

Fig. 3.9: API ricerca informazione

4. API documentation

Le API Locali fornite dalla webapp Yinco e descritte nella sezione precedente sono state documentate utilizzando il modulo NodeJS chiamato **Swagger UI Express**, grazie al quale possiamo generare una pagina web dove è possibile trovare la documentazione delle varie API usate nel progetto, oltre a poterle provare con mano.

In particolare, di seguito mostriamo la pagina web relativa alla documentazione che presenta le 2 API (GET) per la gestione dei dati della nostra applicazione.
La prima GET viene utilizzata per estrarre il link alla pagina web di unitn di un docente inserito dall'utente; la seconda invece, per estrarre l'informazione salvata nel database grazie ai tag inseriti dall'utente.

L'endpoint da invocare per raggiungere la seguente documentazione è:
https://yinco-co.herokuapp.com/api-docs/

Se invece si volesse fare il deployment in locale dell'applicazione, allora la documentazione è disponibile al seguente endpoint:
localhost:3000/api-docs

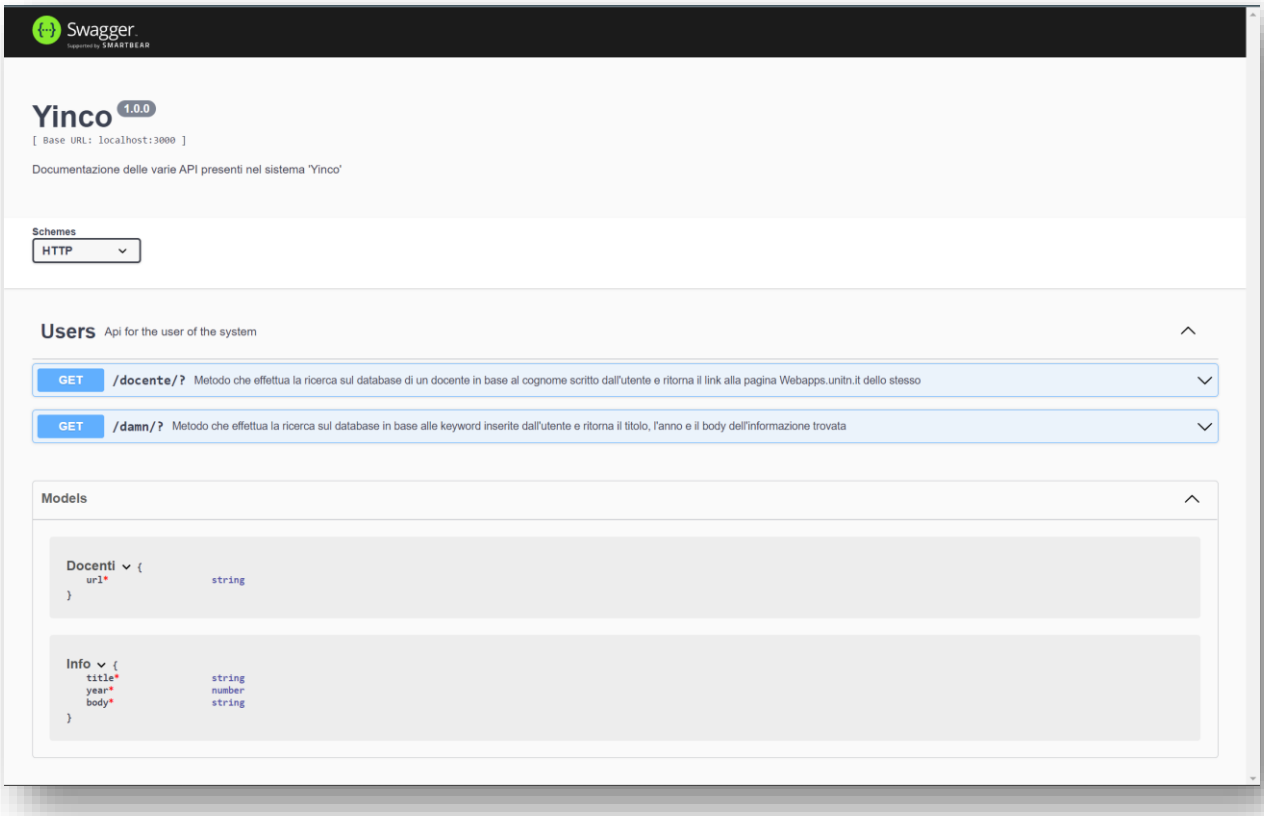

Fig. 4.1: pagina Swagger delle API

5. FrontEnd implementation

In questo capitolo affronteremo il frontend dell'applicazione Yinco, ovvero l'interfaccia che l'utente si troverà di fronte per interagire col sistema.
Troveremo essenzialmente 6 pagine utilizzabili dall'utente:

- **Home**
- **Chatbot**
- **Contatti**
- **Impostazioni**
- **Template di risposta**
- **Login**

Inoltre, tutte le pagine del sito sono responsive, ovvero adattabili alle dimensioni del dispositivo su cui lavorano.

5.1 Home

La pagina **home** si presenterà nel seguente modo

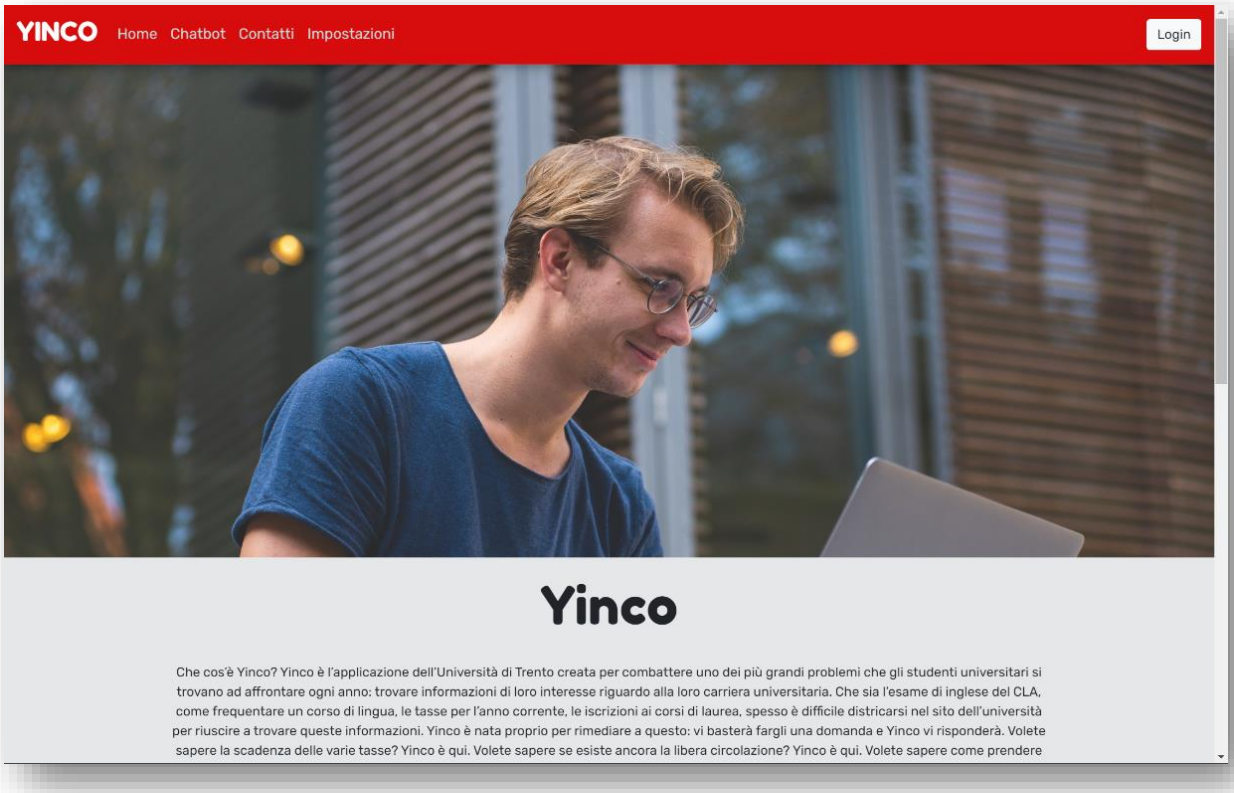

Fig. 5.1: Home

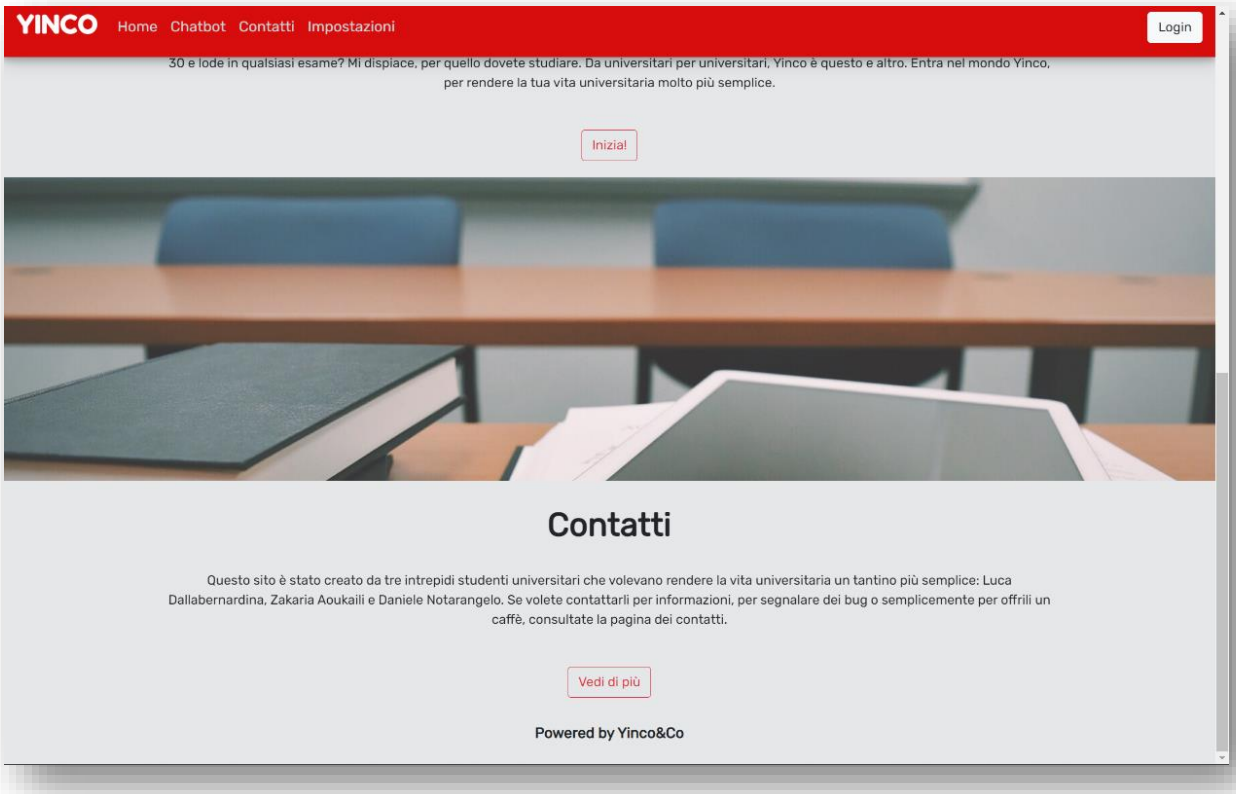

Fig. 5.2: Home

Il primo elemento che troviamo, e che si ripeterà anche nelle prossime schermate, è la barra di navigazione, grazie alla quale possiamo raggiungere tutte la parti del nostro sito, anche la home stessa, tramite gli appositi bottoni.
Qui, inoltre, troviamo anche una piccola presentazione di che cos'è Yinco e perché sia stato creato e, grazie al bottone "Inizia!", possiamo procedere direttamente alla pagina della chatbot, nella quale potremo porre al bot tutti i nostri dubbi.
Sotto possiamo anche trovare una presentazione dei creatori di questa web app e, tramite il bottone "vedi di più", verremo portati nella pagina apposita dei contatti che vedremo tra poco.

5.2 Chatbot

La pagina della **chatbot** presenta la solita barra di navigazione e, al di sotto, la chat, con un messaggio iniziale di presentazione da parte del bot.
A questo punto, tramite la casella di testo e il pulsante invia, possiamo inserire ed inviare una o più domande al sistema.

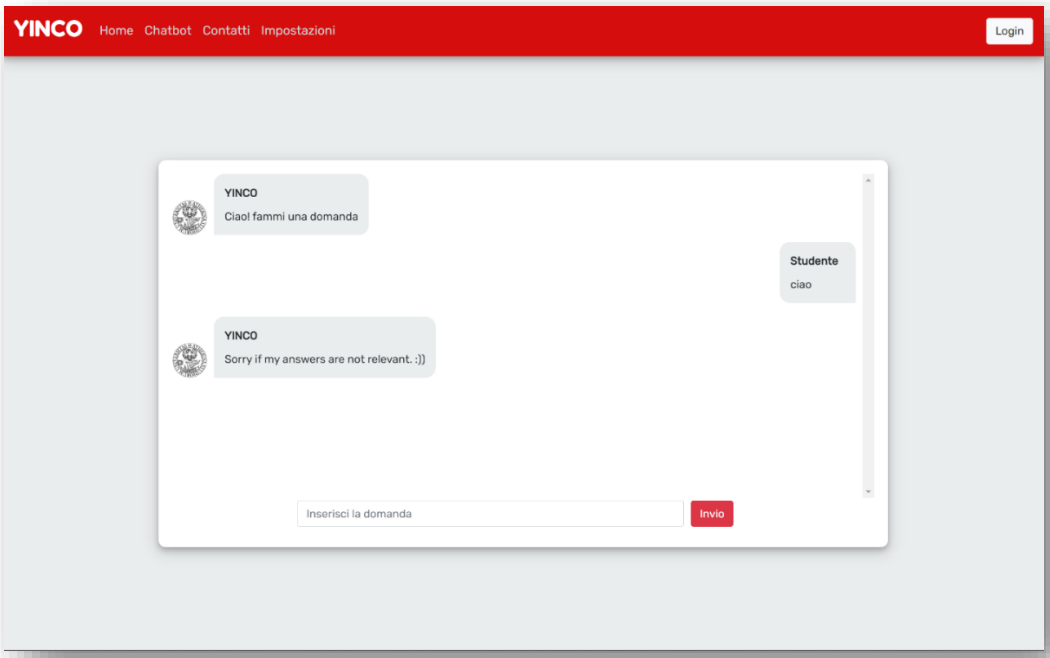

Fig. 5.3: Chatbot

5.3 Contatti

La pagina dei **contatti** presenta anch'essa la barra di navigazione e tutte le informazioni riguardanti l'azienda Yinco. Troveremo la sede, i contatti, sia email che di telefono, dei creatori ed infine la mappa con un tag sulla sede su cui è presente un link che ci riporta alla pagina di google maps.

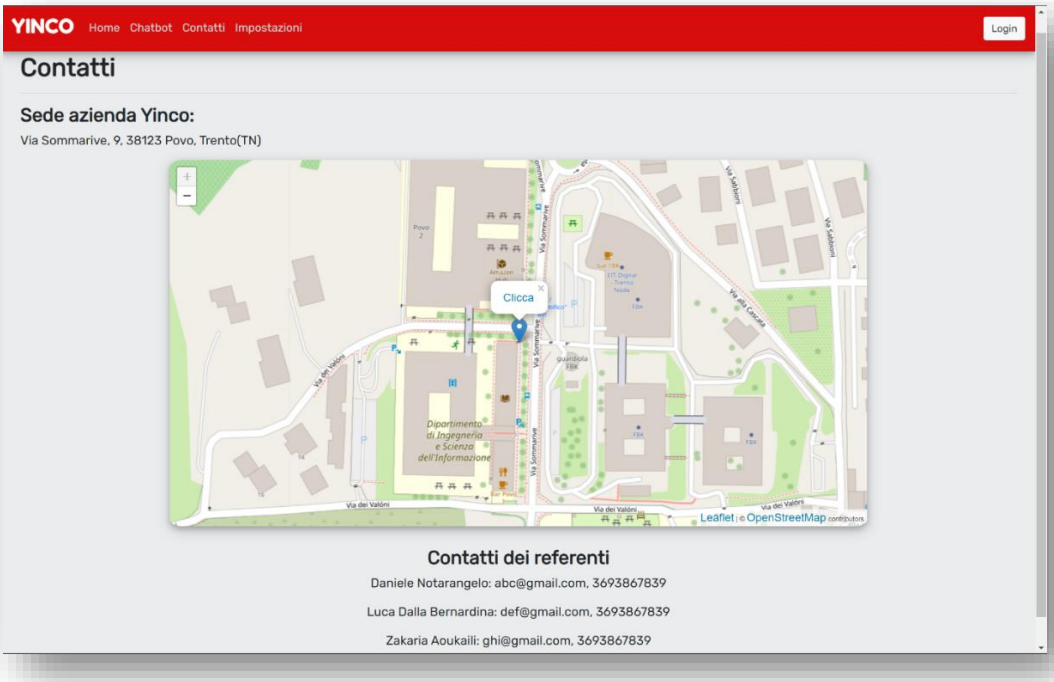

Fig. 5.4: Contatti

5.4 Impostazioni

La pagina di **impostazioni** è l'ultima delle principali riguardanti il sito.
Qui troviamo la barra di navigazione e 3 opzioni che offre l'applicazione:

- attivazione delle notifiche riguardanti le scadenze per tasse ed esami;
- il logout;
- la scelta della lingua (tra italiano o inglese).

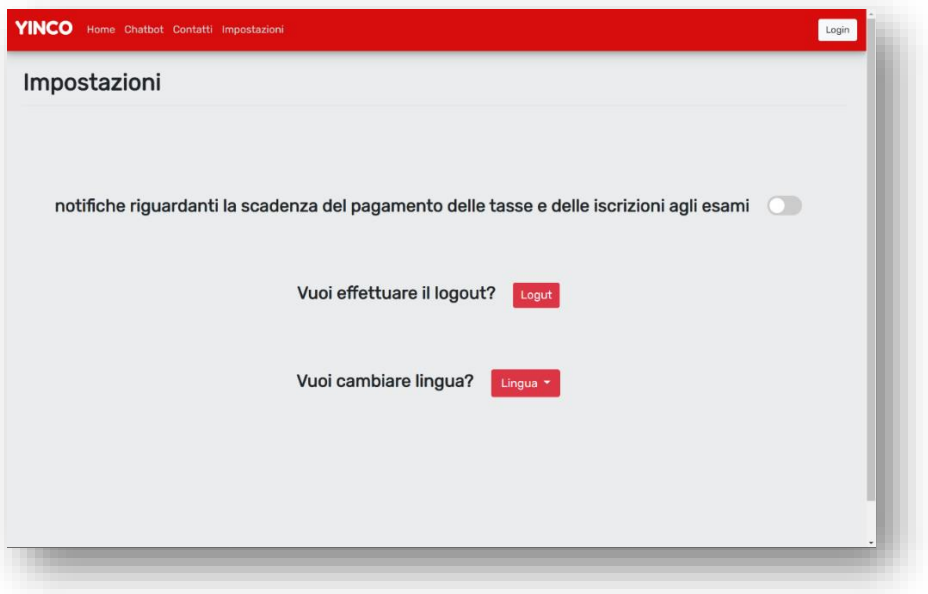

Fig. 5.5: Impostazioni

5.5 Template pagina di risposta

Abbiamo anche un template per una pagina di risposta tipo, riguardo le informazioni che richiede l'utente. Tuttavia, per problemi di implementazione, questa funzionalità non è stata aggiunta, perciò la risposta verrà stampata nella chat come un semplice messaggio

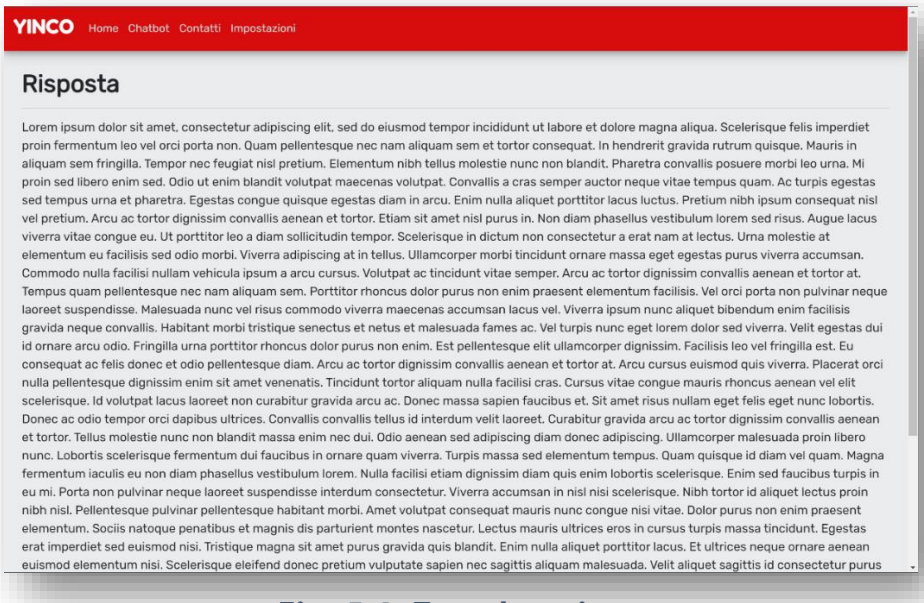

Fig. 5.6: Template risposta

5.6 Login

La pagina di login, come era stato presentato nel documento di analisi dei requisiti, sarebbe dovuta essere quella dell'università di Trento, dato che avremmo dovuto sfruttare l'API universitaria, ma, a causa dell'impossibilità nell'utilizzo di quest'ultima, abbiamo implementato noi una semplice pagina di login, raggiungibile da tutte le pagine del sito tramite la barra di navigazione.

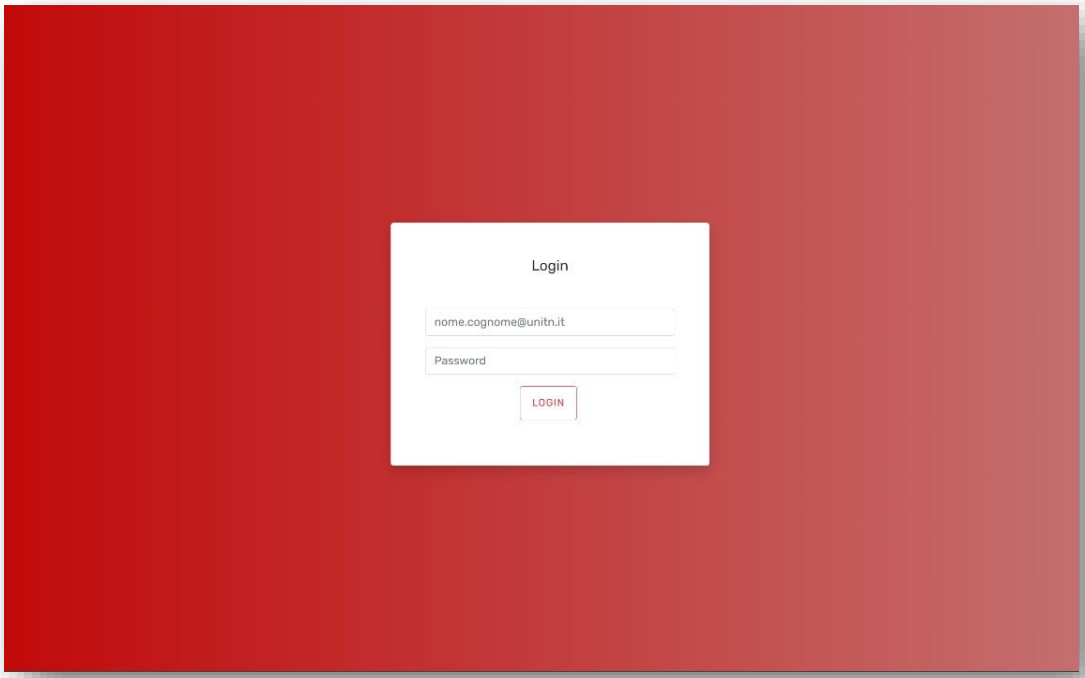

Fig. 5.7: Login

6. GitHub Repository and Deployment info

Il GitHub del nostro progetto si suddivide essenzialmente in tre parti:

- **Documentazione**
- **FrontEnd**
- **BackEnd**

Nella repository della **documentazione** troviamo 4 sottocartelle, all'interno delle quali sono presenti tutti i file utilizzati per la produzione dei 4 documenti di specifica di Yinco.
Nella repository di **FrontEnd** troviamo per lo più file HTML, CSS e un file Javascript, utilizzati per la costruzione dell'interfaccia utente descritta nel capitolo precedente.
Infine troviamo la repository di **BackEnd** nella quale troviamo pressoché tutti i file descritti nella Project Structure, inclusa anche una copia dei file che si possono trovare nella repository di FrontEnd.

Per quanto riguarda il deployment, abbiamo sfruttato il servizio di **Heroku**.
Qui abbiamo creato una macchina chiamata "yinco-co", nella quale abbiamo caricato il sistema Yinco.
Il sito è raggiungibile attraverso il seguente link: https://yinco-co.herokuapp.com/

In caso invece si voglia eseguire il deployment in locale del sistema, basta scaricare i file contenuti nella repository BackEnd o, in alternativa, scaricare i file delle cartelle BackEnd e FrontEnd e inserire nella cartella del BackEnd contenete i file del Front-End quelli scaricati dalla repository FrontEnd. Fatto ciò, l'unica cosa da fare è avviare il sistema, scrivendo da terminale *node server.js,* e, fatto ciò, connettersi all'url localhost:3000/, da cui poi si potrà avere accesso a tutte le funzioni del sito.

7. Testing

Per quanta riguarda il testing delle API, abbiamo testato le uniche API da noi create, ovvero la ricerca del **docente** e la ricerca dell'**informazione.** Inoltre è stato effettuato il testing anche per la funzione **filter,** in quanto dal suo risultato dipende l'input delle due API dedicate alla ricerca rispettivamente di un docente o di un'informazione. I risultati ottenuti sono rappresentati nell'immagine sottostante. Vediamo come sono stati ottenuti questi risultati.

Nota: Sia in 'docente' che in 'informazione' non si ha il 100% di coverage in quanto in entrambe è presente una riga di codice (riga 13) che permette a MongoDB di poter eseguire le query duplicate. È possibile consultare il codice delle seguenti due API nel paragrafo 'Sviluppo API'

docente.js		90%	9/10	100%	4/4	66.66%	2/3	90%	9/10
filter.js		100%	10/10	100%	4/4	100%	3/3	100%	8/8
informazione.js		90%	9/10	100%	4/4	66.66%	2/3	90%	9/10

Fig. 7.1: risultati test API

7.1 Test docente

Il test dell'API del docente prevede due controlli: il primo, il caso positivo, in cui l'API ritorna 200, quindi l'informazione corretta; il secondo, in cui ritorna un errore 404, perciò l'informazione non è stata trovata.
Nel primo caso, controlliamo sia che lo **statusCode** della get che chiamiamo sia uguale a 200, sia che l'url trovato nel database sia quello che ci aspettiamo grazie all'utilizzo di una **toEqual.**
Per il secondo caso, controlliamo solo che lo statusCode sia 404.

```
const request = require('supertest');
const app = require('../app');


describe('Test dell\'API Docente', () => {
  test('Deve restituire il docente con il cognome specificato', async () => {
    const res = await request(app).get("/docente/?cognome=Bucchiarone");
    expect(res.statusCode).toEqual(200);
    expect(res.body.url).toEqual('https://webapps.unitn.it/du/it/Persona/PER0053043/Didattica');
  }, 20000);

  test('Deve restituire un errore se il docente non esiste', async () => {
    const res = await request(app).get("/docente?cognome=Bianchi");
    expect(res.statusCode).toEqual(404);
  }, 20000);
});
```

Fig. 7.2: test API docente

7.2 Test informazione

Il test dell'API delle **informazioni** è più complesso e prevede sei controlli:

1. è il caso positivo, in cui controlliamo che l'API ritorni 200 e che il titolo dell'informazione sia corretta;
2. è quello in cui ritorna un errore 404, poiché il tag inserito dall'utente è errato;
3. controlliamo che l'API ritorni 200 e che il titolo dell'informazione sia corretta, nonostante il tag inserito dall'utente sia incompleto;
4. prevede che l'API ritorni sempre tutte le informazioni corrette, ma questa volta l'utente ha inserito più tag corretti: da notare che i tag sono inseriti nell'ordine con cui sono stati inseriti in MongoDB;
5. controlliamo che l'API ritorni errore 404 nel caso in cui i tag siano stati inseriti in un ordine diverso da quello previsto;
6. verifichiamo che l'API ritorni un errore 404 quando l'utente inserisce dei tag sbagliati.

```
const request = require('supertest');
const app = require('../app');


describe('Test dell\'API Informazione', () => {
  test('Deve restituire l\'informazione riferita al tag dato', async () => {
    const res = await request(app).get("/damn/?tags=libera circolazione");
    expect(res.statusCode).toEqual(200);
    expect(res.body.title).toEqual('Libera Circolazione: abbonamento per trasporto urbano ed extraurbano');
  }, 20000);

    test('Deve restituire un errore se il tag non è stato scritto bene', async () => {
      const res = await request(app).get("/damn/?tags=libea circolazione");
      expect(res.statusCode).toEqual(404);
    }, 20000);

    test('Deve restituire l\'informazione riferita al tag incompleto dato', async () => {
        const res = await request(app).get("/damn/?tags=libera");
        expect(res.statusCode).toEqual(200);
        expect(res.body.title).toEqual('Libera Circolazione: abbonamento per trasporto urbano ed extraurbano');
    }, 20000);

    test('Deve restituire l\'informazione riferita ai tag dati. Vanno nello stesso ordine di MongoDB', async () => {
        const res = await request(app).get("/damn/?tags=trasporti, bus");
        expect(res.statusCode).toEqual(200);
        expect(res.body.title).toEqual('Libera Circolazione: abbonamento per trasporto urbano ed extraurbano');
    }, 20000);

    test('Deve restituire un errore se i tag inseriti sono in ordine diverso rispetto a MongoDB', async () => {
        const res = await request(app).get("/damn/?tags=bus, trasporti");
        expect(res.statusCode).toEqual(404);
    }, 20000);

  test('Deve restituire un errore se il tag non si riferisce a nessuna informazione', async () => {
    const res = await request(app).get("/damn/?tags=ciao");
    expect(res.statusCode).toEqual(404);
  }, 20000);
});
```

Fig. 7.3: test API informazione in italiano

Le stesse operazioni vengono svolte anche per le informazioni inglese, come si può notare nell'immagine sottostante.

```
describe('Test dell\'API Informazione in inglese', () => {
  test('Deve restituire l\'informazione riferita al tag dato', async () => {
    const res = await request(app).get("/damn/?tags=free circulation");
    expect(res.statusCode).toEqual(200);
    expect(res.body.title).toEqual('Free circulation pass for in-town and out-of-town public transport');
  }, 20000);

    test('Deve restituire un errore se il tag non è stato scritto bene', async () => {
      const res = await request(app).get("/damn/?tags=fre circulatin");
      expect(res.statusCode).toEqual(404);
    }, 20000);

    test('Deve restituire l\'informazione riferita al tag incompleto dato', async () => {
        const res = await request(app).get("/damn/?tags=free");
        expect(res.statusCode).toEqual(200);
        expect(res.body.title).toEqual('Free circulation pass for in-town and out-of-town public transport');
    }, 20000);

    test('Deve restituire l\'informazione riferita ai tag dati. Vanno nello stesso ordine di MongoDB', async () => {
        const res = await request(app).get("/damn/?tags=transportation, buses");
        expect(res.statusCode).toEqual(200);
        expect(res.body.title).toEqual('Free circulation pass for in-town and out-of-town public transport');
    }, 20000);

    test('Deve restituire un errore se i tag inseriti sono in ordine diverso rispetto a MongoDB', async () => {
        const res = await request(app).get("/damn/?tags=buses, transportation");
        expect(res.statusCode).toEqual(404);
    }, 20000),

    test('Deve restituire un errore se il tag non si riferisce a nessuna informazione', async () => {
      const res = await request(app).get("/damn/?tags=ciao");
      expect(res.statusCode).toEqual(404);
    }, 20000);
});
```

Fig. 7.3: test API informazione in inglese

Fig. 7.4: test API informazione in inglese

7.3 Test filter

La funzione "**filter**" viene testata con diverse stringhe di testo e gli array "docenti" e "info" per verificare che restituisca il valore corretto in base ai casi sottostanti e presenti nella figura 7.5. I casi considerati durante la fase di testing sono stati i seguenti:

1) verifica che la funzione restituisca 1 quando il messaggio continene una parola presente nell'array docenti;
2) verifica che la funzione restituisca 2 quando il messaggio contiene una parola presente nell'array info;
3) verifica che la funzione restituisca 0 quando il messaggio non contiene parole presenti negli array docenti o info.

```
const filter = require('../controllers/filter');
const docenti = ["Ranise", "Passerone", "Giorgini", "Tomasi", "Bucchiarone", "Casari", "Bouquet", "Velha", "Montresor", "Iacca"];
const info = ["tasse", "rate", "contributi", "esoneri", "borsa di studio", "opera universitaria",
              "invalidità", "alloggio", "casa", "libera circolazione", "trasporti", "bus", "treni",
              "taxes", "fees", "contributions", "tuition fees", "exemptions", "scholarship", "disability",
              "housing", "home", "free circulation", "transportation", "buses", "trains"];

describe('La funzione filter', () => {
  test('restituisce 1 quando il messaggio contiene una parola presente nell\'array docenti', () => {
    const msgText = "Fornisci informazioni sul prof Bucchiarone";
    expect(filter(msgText, docenti)).toEqual(1);
  });

  test('La funzione filter restituisce 2 quando il messaggio contiene una parola presente nell\'array info', () => {
    const msgText = "Dammi informazioni sulle tasse";
    expect(filter(msgText, docenti)).toEqual(2);
  });

  test('restituisce 0 quando il messaggio non contiene parole presenti negli array docenti o info', () => {
    const msgText = "Ciao, come va?";
    expect(filter(msgText, docenti)).toEqual(0);
    expect(filter(msgText, info)).toEqual(0);
  });
});
```

Fig. 7.5: test API filter