Immagine che contiene testo

Descrizione generata automaticamente

Dipartimento di Ingegneria e Scienza dell’Informazione

2022/2023

**YINCO**

**Sviluppo applicazione**

|  |  |  |  |
| --- | --- | --- | --- |
| Doc. Name | D4-Sviluppo di Yinco | Doc. Number | Rev 0.9 |
| Description | Il documento include la spiegazione dello sviluppo del sistema Yinco | | |

**INDICE**

**1. Scopo del documento** 3

**2. User flows** 3

**2.1 Utente intenzionato ad usare il sistema** 4

**2.2 Flow delle pagine per l’utente** 5

**3. Application implementation and documentation** 7

**3.1 Project Structure** 7

**3.2 Project Dependencies** 8

**3.3 Project Data or DB** 8

**3.4 Project APIs** 9

**3.4.1 Resources extraction from the class diagram** 9

**3.4.2 Resources models** 12

**3.5 Sviluppo API** 15

**3.5.1 Ricerca docente** 15

**3.5.2 Ricerca informazioni** 16

**4. API documentation** 17

**5. FrontEnd implementation** 18

**5.1 Home** 18

**5.2 Chatbot** 19

**5.3 Contatti** 20

**5.4 Impostazioni** 21

**5.5 Template pagina di risposta** 21

**5.6 Login** 22

**6. GitHub Repository and Deployment info** 23

**7. Testing** 24

**7.1 Test docente** 24

**7.2 Test informazione** 24

**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 flow legate al ruolo del responsabile amministrativo (di seguito denominato RA) dell’applicazione, 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 primo 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 secondo 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 1.1: 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 azione.

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 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 o un errore in base alle caratteristiche descritte nel Documento di Specifica dei Requisiti (o D2) oppure un link a cui potrà trovare l’informazione trovata.

Qui l’utente potrà scegliere se

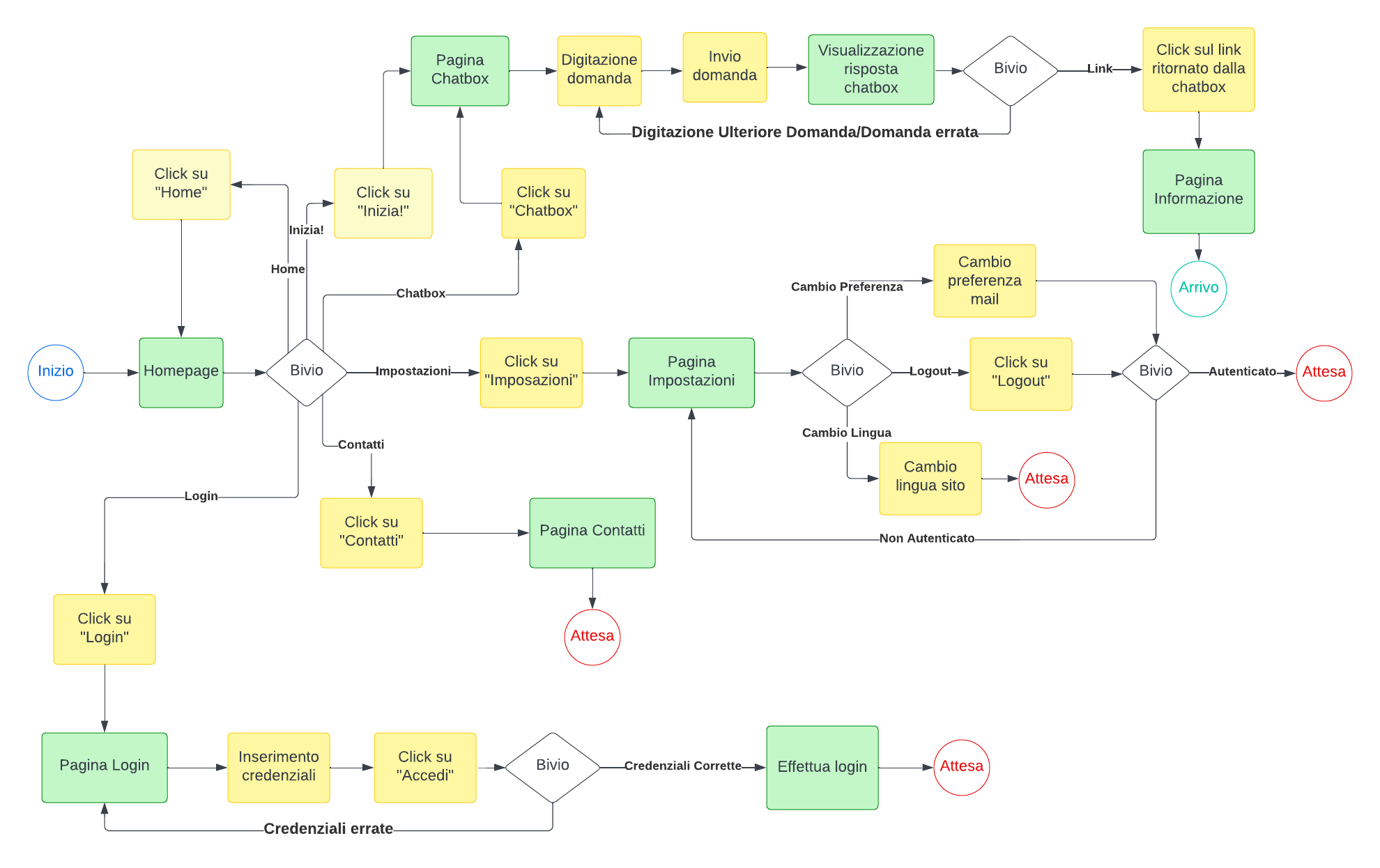
1) aprire il link che la chatbot gli ha ritornato;

2) porre ad essa una nuova domanda.

Nel primo 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 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: 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, diagramma che poteva essere aggiunta nello user flow precedente, ma che avrebbe diminuito la leggibilità dello stesso, motivo per cui è stato creato uno user flow 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.

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 Fow è visibile graficamente qui sotto.

Immagine che contiene testo, parcheggio

Descrizione generata automaticamente

**Figura 3: User flow che illustra l’interconnessione tra le varie pagine del sistema**

**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 MongoDB.

**3.1 Project Structure**

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

Immagine che contiene testo

Descrizione generata automaticamente

**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**”.

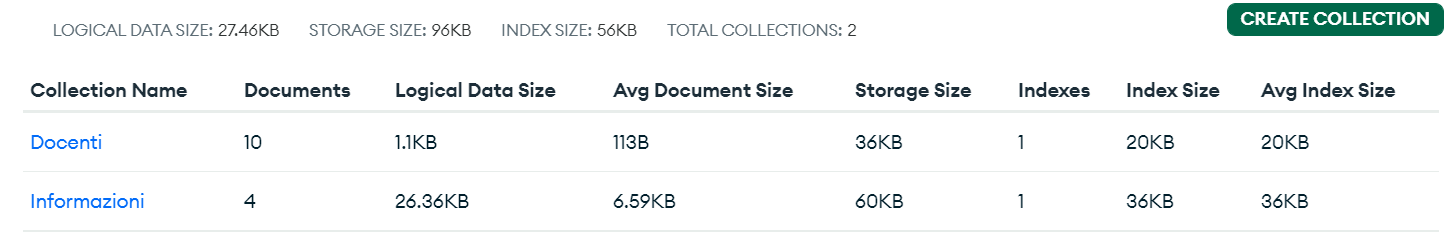
**3.2 Project Dependencies**

I moduli Node che sono stati aggiunti nel file “package.json”, ed utilizzati, 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.

****

**Fig. 3.2: Collezioni dei dati**

Abbiamo creato una collezione “**Docenti**” per salvare le informazioni dei professori, e la struttura è la seguente.

****

**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 nel seguente modo:

**Immagine che contiene testo

Descrizione generata automaticamente**

**Fig. 3.4: Tipo di dato “Informazione”**

**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 alcuni API le potevamo implementare direttamente in quest’ultimo, dunque quelle che il sistema utilizzerà effettivamente sono 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:

* ‘**link’**
* ‘**docente’**
* ‘**database’**

L’API ‘**docente’** è una GET, cioè utilizza il metodo GET per recuperare la rappresentazione di una risorsa, nel nostro caso la pagina url del docente.

L’API ‘**link’**, anch’essa una GET, è la risorsa che ritorna l’url di una pagina la quale contiene le informazioni richieste.

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**.

Nell’API ‘link’, il dato di partenza è “cosa verrà ritornato dall’API ‘informazione”.

L’API non implementata della classe ‘ricerca’ è ‘privilegi’ e anch’essa è una GET e ritorna se l’utente è attualmente autenticato o no.

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 informazioni l’email e la password dell’utente.

Le API della classe ‘**Utente\_autenticato’** sono ‘**logout’** e ‘**login\_automatico’**. 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.

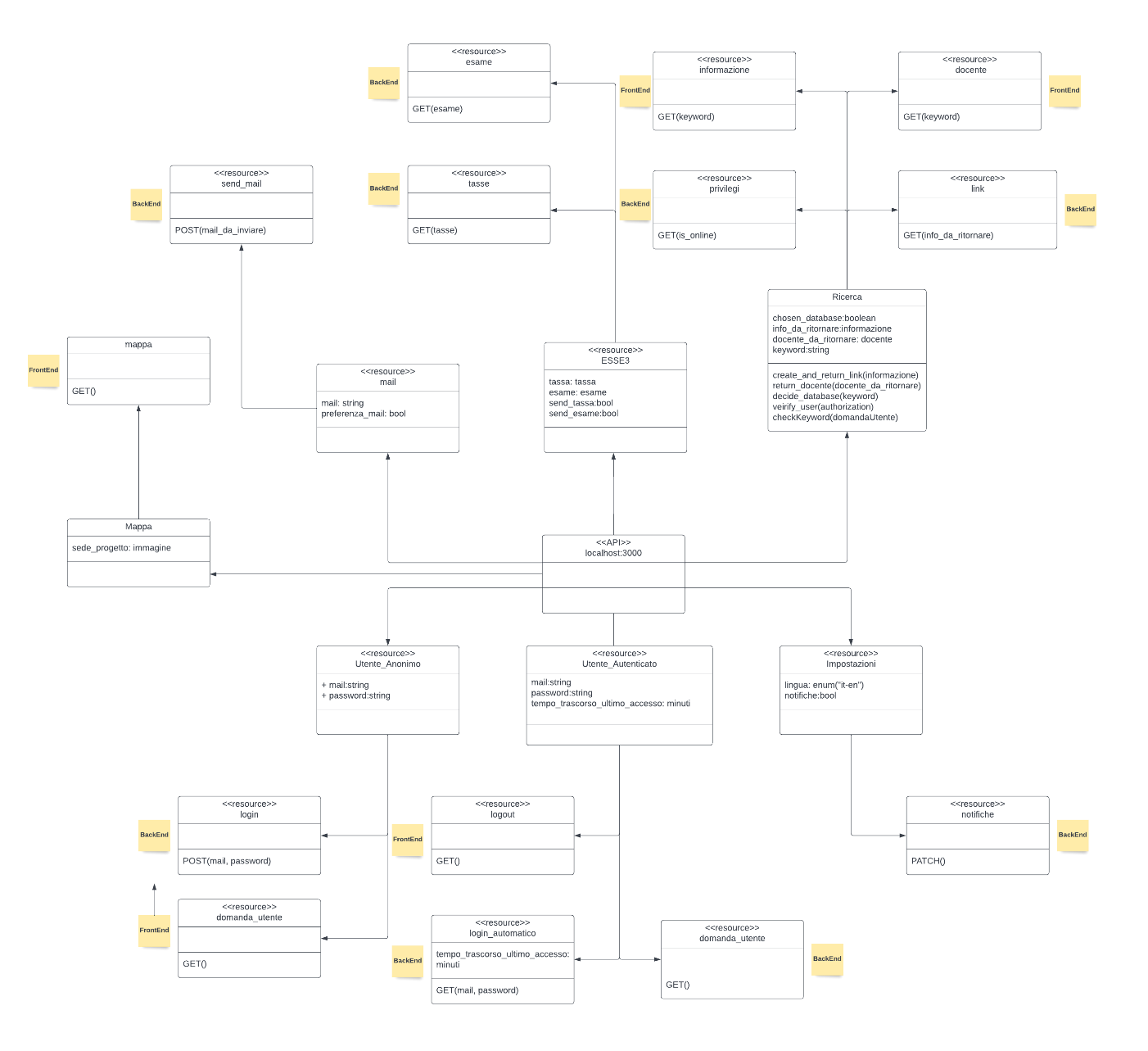
Per fare ciò l’API necessita dell’informazione ‘**tempo\_trascorso\_ultimo\_accesso’**, in quanto l’utente può effettuare il login automatico se e solo se il tempo passato dall’ultimo logout è inferiore a 15 minuti. I dati di partenza su cui si basa questa funzionalità sono l’email e la password dell’utente, 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.

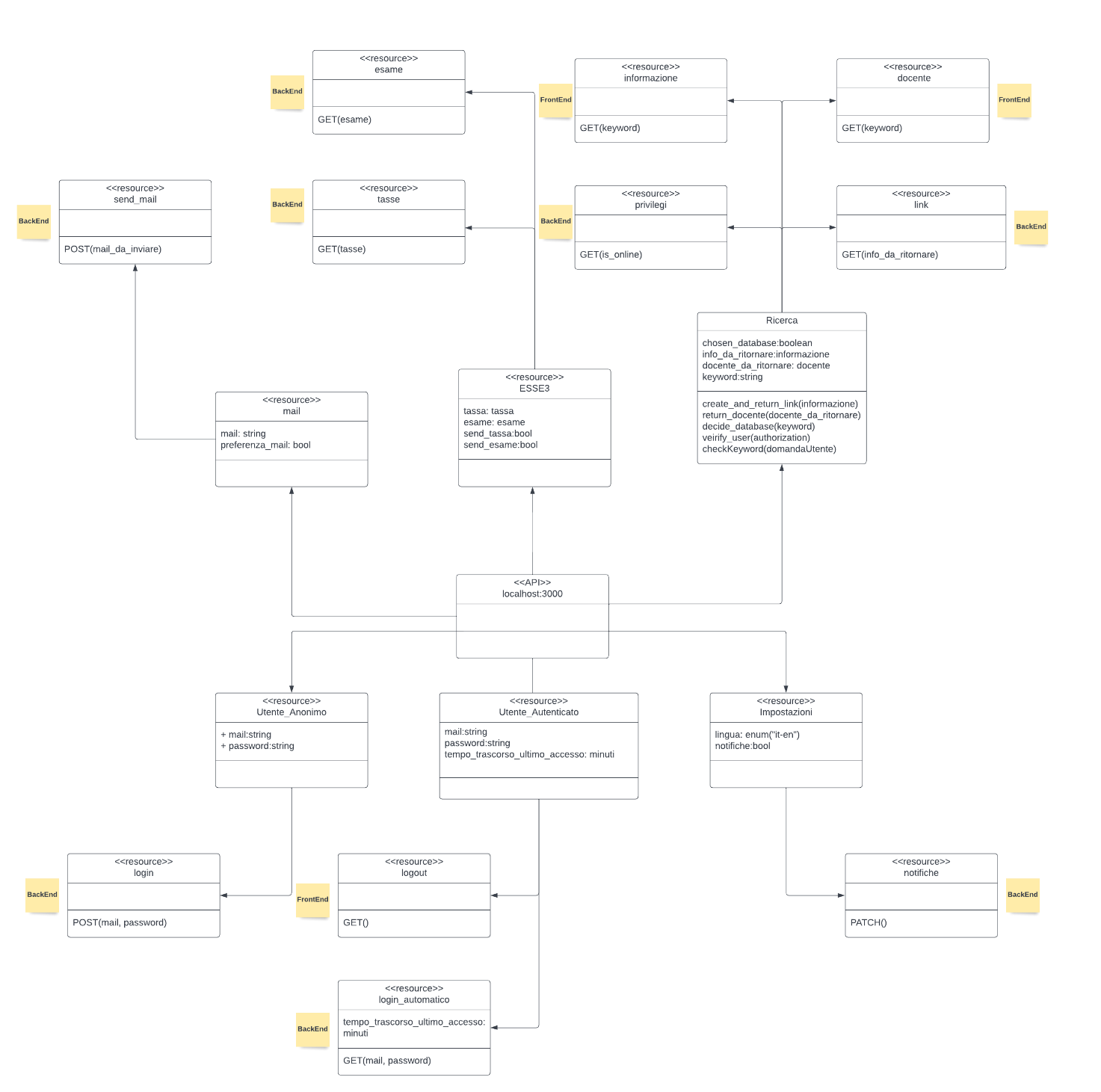
Dovendo semplicemente aggiornare una risorsa il metodo di questa API è PATCH. L’API della risorsa ‘**mail’** è ‘**send\_mail’** di tipo POST. Questa API si interfaccia con le API ‘tasse’ e ‘esami’ della risorsa ESSE3, in quanto il body della mail conterrà informazioni su ciò che verrà ritornato da queste 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 sulle tasse e sugli 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 Models 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 booleano.

L’attributo ‘chosen\_database’ abbiamo 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 ‘true’ al database dei docenti e ‘false’ 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.

In tutti e quattro i casi viene inviato un messaggio all’utente.

Le altre API presenti nel Resource Models 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 queste:

* ‘**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 Resource Models 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.

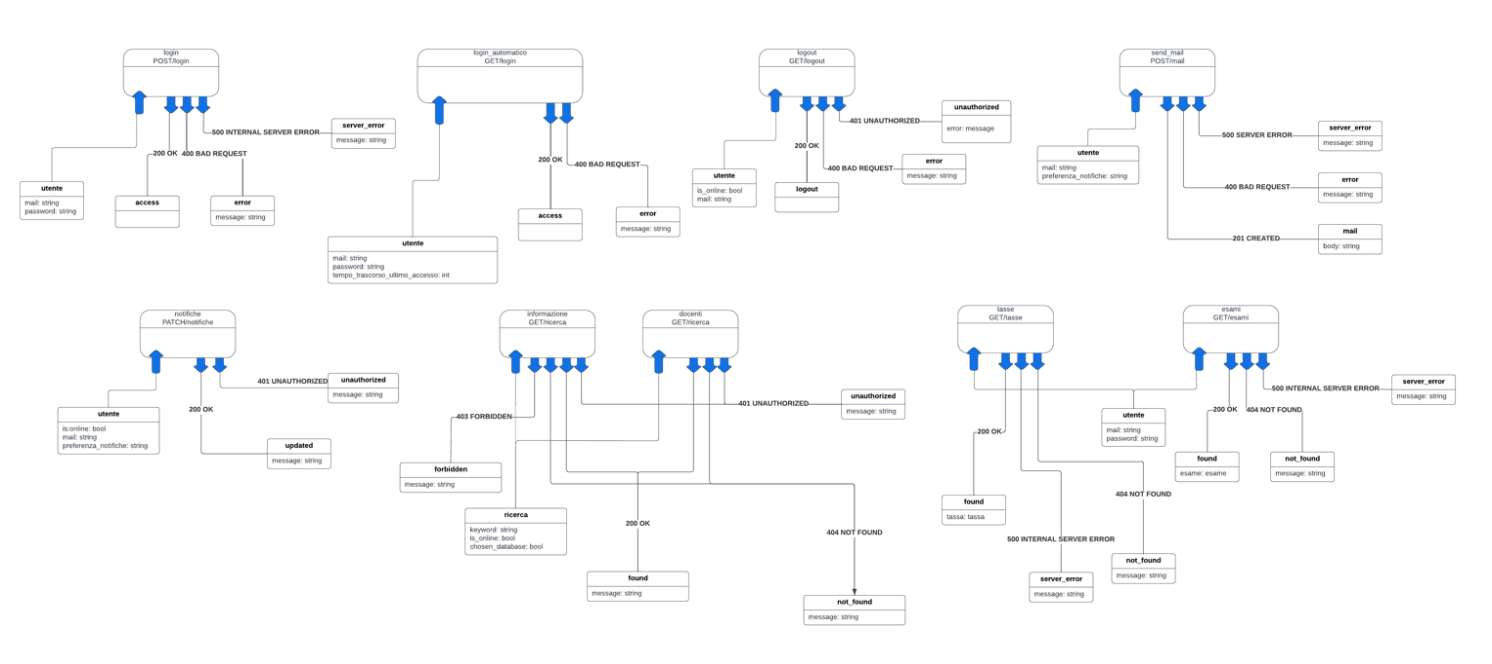
Sia in ‘not\_found’ che in ‘server\_error’ viene ritornato un messaggio di tipo stringa. L’API **‘esami’** ha anch’essa tre response body:

‘**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 models**

**3.5 Sviluppo API**

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

**3.5.1 Ricerca docente**

La funzione utlizzata 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 del docente, in caso contrario, 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.

La riga 13 viene utilizzata per non stampare in console un grande numero di errori. Per finire questa funzione viene estratta, in quanto dovrà essere poi richiamata in ‘routes’.

Immagine che contiene testo

Descrizione generata automaticamente

**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 un tag inserito dall’utente: nel caso in cui venga trovato, la funzione restituisce il body dell’informazione, in caso contrario, 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 sia trovata , l’API ritorna il codice 200 OK, in caso contrario, ritorna il codice 404 NOT FOUND.

La riga 13 viene utilizzata per non stampare in console un grande numero di errori. Per finire, questa funzione viene estratta, in quanto dovrà essere poi richiamata in ‘routes’.

Immagine che contiene testo

Descrizione generata automaticamente

**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 l’endpoint dedicato alla presentazione delle API, in quanto crea una pagina web dalle definizioni delle specifiche API.

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/>

**Immagine che contiene testo

Descrizione generata automaticamente**

**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.

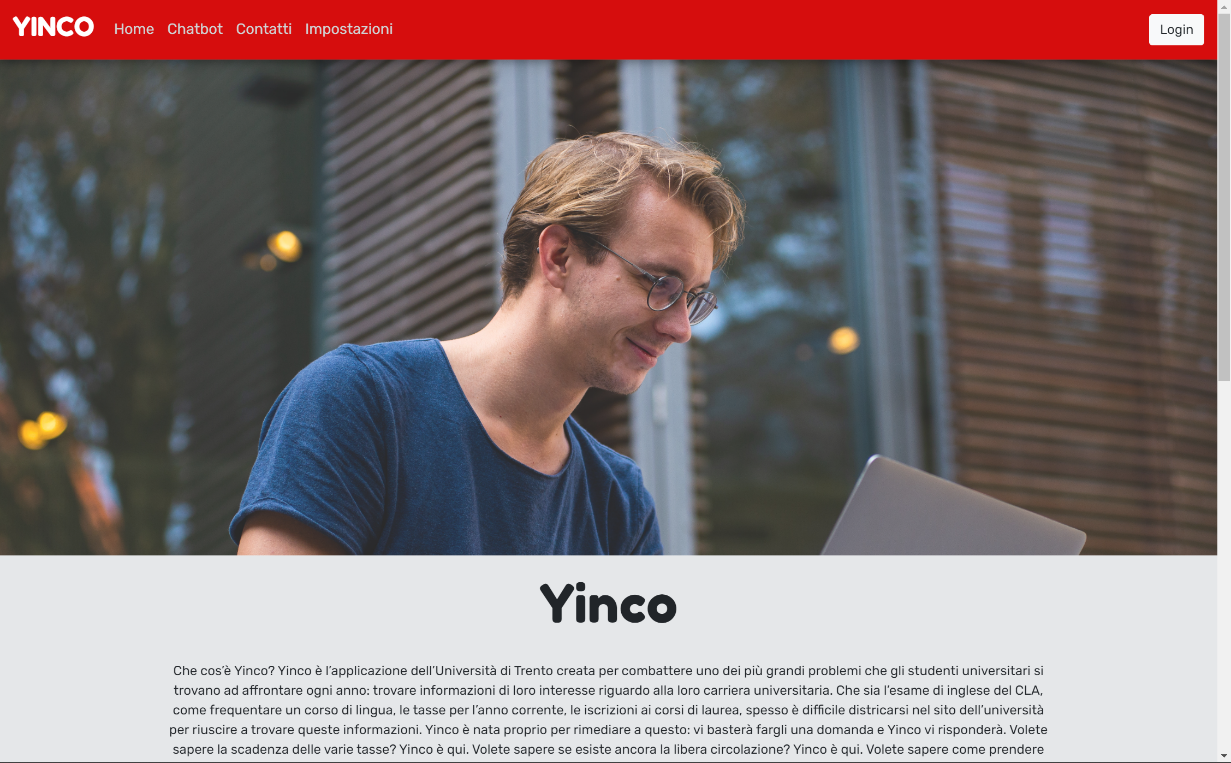
Troveremo essenzialmente 6 pagine utilizzabili dall’utente:

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

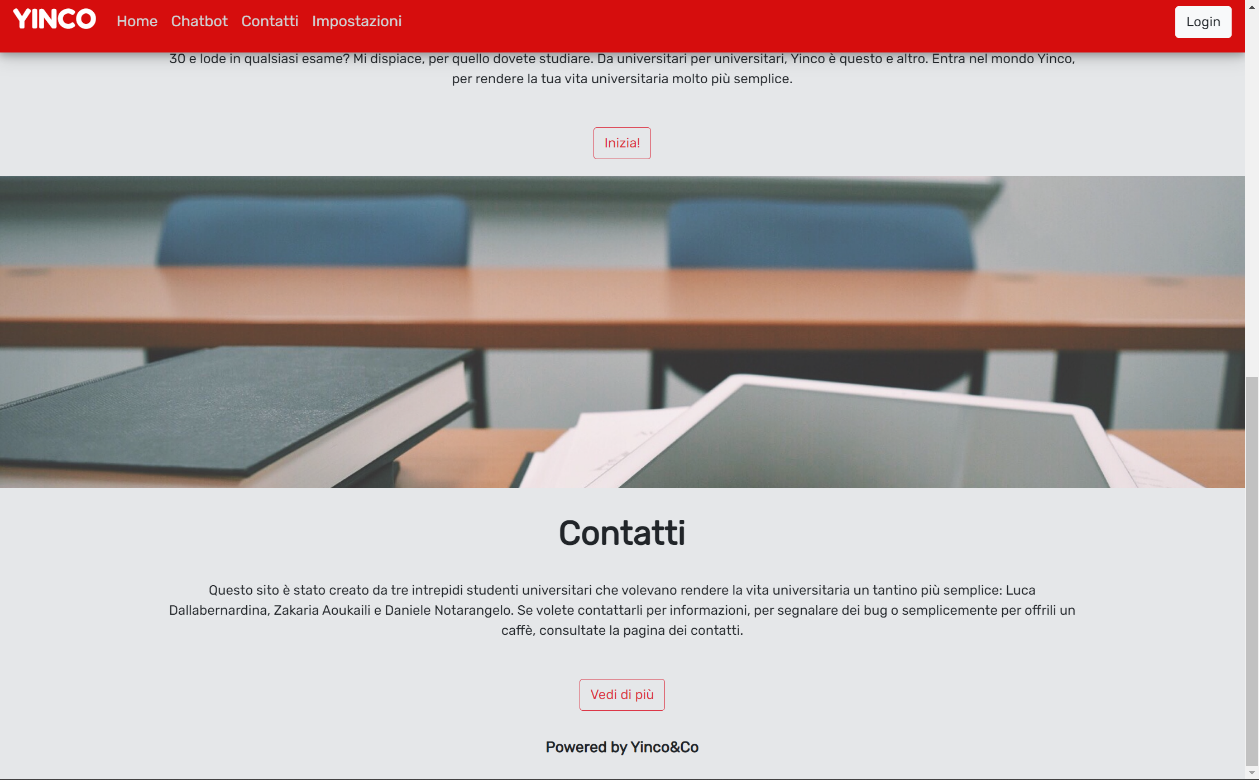
Inoltre, tutte le pagine del sito sono responsive, ovvero adattabili alle dimensioni del dispositivo su cui lavorano.

**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 cos’è Yinco e perché sia stato creato, e grazie al bottone “Inizia!”, possiamo procedere direttamente alla pagina di 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 di **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 tutte le domande che vogliamo.

Immagine che contiene testo

Descrizione generata automaticamente

**Fig. 5.3: Chatbot**

**5.3 Contatti**

La pagina di **contatti** presenta anch’essa la barra di navigazione, e subito 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.

Immagine che contiene mappa

Descrizione generata automaticamente

**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 loguot;
* 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

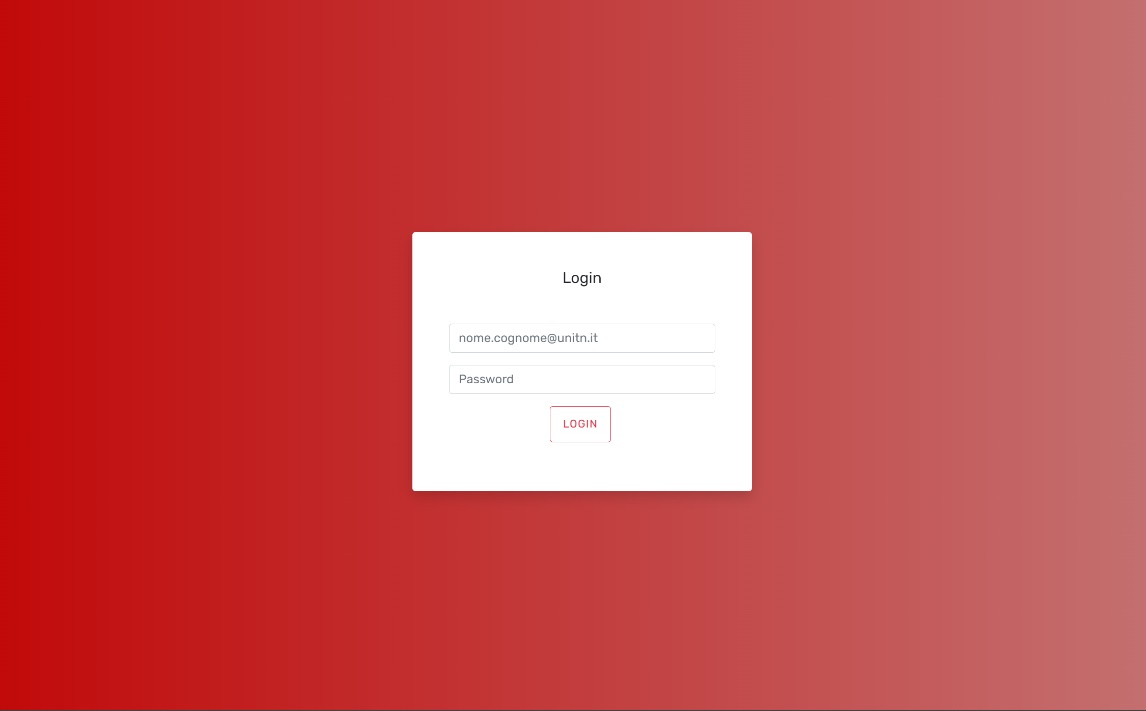
Immagine che contiene testo

Descrizione generata automaticamente

**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.

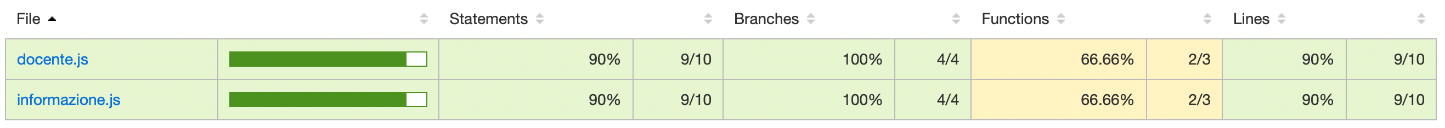
Per quanto riguarda il deployment, abbiamo sfruttato il servizio di **Heroku**.

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/>

**7. Testing**

Per quanta riguarda il testing delle API, ovviamente, abbiamo testato le uniche API da noi create, quindi, la ricerca del **docente** e la ricerca dell’**informazione.** I risultati ottenuti sono rappresentati nell’immagine sottostante. Vediamo come sono stati implementati.



**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.

Immagine che contiene testo

Descrizione generata automaticamente

**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.

Immagine che contiene testo

Descrizione generata automaticamente

**Fig. 7.3: test API informazione**