



UNIVERSITY  
OF TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

CORSO DI INGEGNERIA DEL SOFTWARE



Documento di progetto: Documento di sviluppo dell'applicazione

Gruppo: T33

## Indice

<b>1. Scopo del documento .....</b>	<b>3</b>
<b>2. User Flows.....</b>	<b>4</b>
2.1 Diagramma User Flows .....	5
<b>3. Application Implementation and Documentation.....</b>	<b>6</b>
3.1. Project Structure.....	6
3.2 Project Dependencies .....	7
3.3 Project Data or DB.....	8
3.4 Project APIs.....	10
3.4.1 API description .....	10
3.4.2 Resources Extraction from the Class Diagram .....	12
3.4.3 Resources Model.....	14
<b>4. API Documentation.....</b>	<b>16</b>
<b>5. FrontEnd Implementation .....</b>	<b>30</b>
<b>6. GitHub Repository and Deployment Info .....</b>	<b>37</b>
<b>7. Testing.....</b>	<b>37</b>
<b>8. Note ed eventuali .....</b>	<b>38</b>

## 1. Scopo del documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte dell'applicazione *StayBusy*. Nello specifico, di seguito vengono presentati tutti gli artefatti necessari per la realizzazione dei servizi di gestione delle funzionalità fornite specificatamente solo alla tipologia di utente offerente dell'applicazione *StayBusy* e non all'utente studente. Questa scelta è data dal fatto che si è deciso di avere una programmazione più semplice e comprensibile sviluppando solo il lato utente offerente. Infatti, effettuare lo sviluppo dell'applicazione del lato studente avrebbe comportato una maggiore complessità nella creazione degli *User Flows* e delle *API* in quanto il login dello studente dovrebbe interfacciarsi con il sistema universitario dell'Ateneo a cui è iscritto. Avendo sviluppato un sito per gli studenti provenienti da tutta Italia sarebbe quindi stato difficile interfacciarsi ai siti di tutte le Università. Si è dunque deciso di concentrare l'attenzione sulle funzionalità associate a questo tipo di utente trattandole e sviluppandole nel modo più dettagliato possibile.

In particolare, nella parte iniziale del documento viene presentata la descrizione degli *User Flows* per il ruolo dell'utente offerente.

Gli **User Flows** si riferiscono ai diversi percorsi che gli utenti possono intraprendere mentre interagiscono con il nostro sito. Questi percorsi sono in genere visualizzati attraverso il diagramma di flusso da noi creato e mostrano i vari passaggi che un utente potrebbe eseguire mentre completa un'attività o raggiunge un obiettivo. I flussi utente vengono utilizzati nel nostro processo di progettazione e sviluppo per aiutare a identificare potenziali aree di miglioramento e per garantire che l'esperienza utente sia il più agevole e intuitiva possibile - come indicato nei requisiti non funzionali del documento **D1: Analisi dei requisiti**.

Successivamente si prosegue con la **presentazione delle API** - attraverso *l'API Model* e il *Modello delle risorse*) necessarie per il funzionamento del sito. Le API principali sono state oggetto di una **fase di testing** con input corretti ed errati per dimostrare il loro funzionamento.

Infine, sono presenti due sessioni, una dedicata alle informazioni del *Git Repository* e il *deployment dell'applicazione* stessa e un'altra destinata alla realizzazione del *front end* e del *back end* per poter permettere la visualizzazione delle API in opera in una riproduzione il più simile possibile al front end fornito nel documento **D1: Analisi dei requisiti**.

## 2. User Flows

In questa sezione del documento di sviluppo vengono riportati gli “*User Flows*” per il ruolo dell’utente offerente.

L’immagine che segue descrive lo *User Flow* relativo alla gestione delle informazioni riguardanti le funzionalità proposte all’utente offerente. Nello specifico egli ha in ogni momento la possibilità di consultare la lista dei propri annunci attivi e dei corrispettivi studenti candidati, la possibilità di modificare le impostazioni del profilo personale, di visualizzare la lista degli annunci non più attivi - ovvero degli annunci la cui data di inizio è precedente al momento della visualizzazione - ed infine di creare un nuovo annuncio.

Attraverso questo diagramma presentiamo inoltre le relazioni che sussistono tra le varie azioni che l’utente offerente può svolgere e le features descritte nella *sezione 3* del documento. In aggiunta è presente una breve legenda che descrive i simboli utilizzati.

## 2.1 Diagramma User Flows

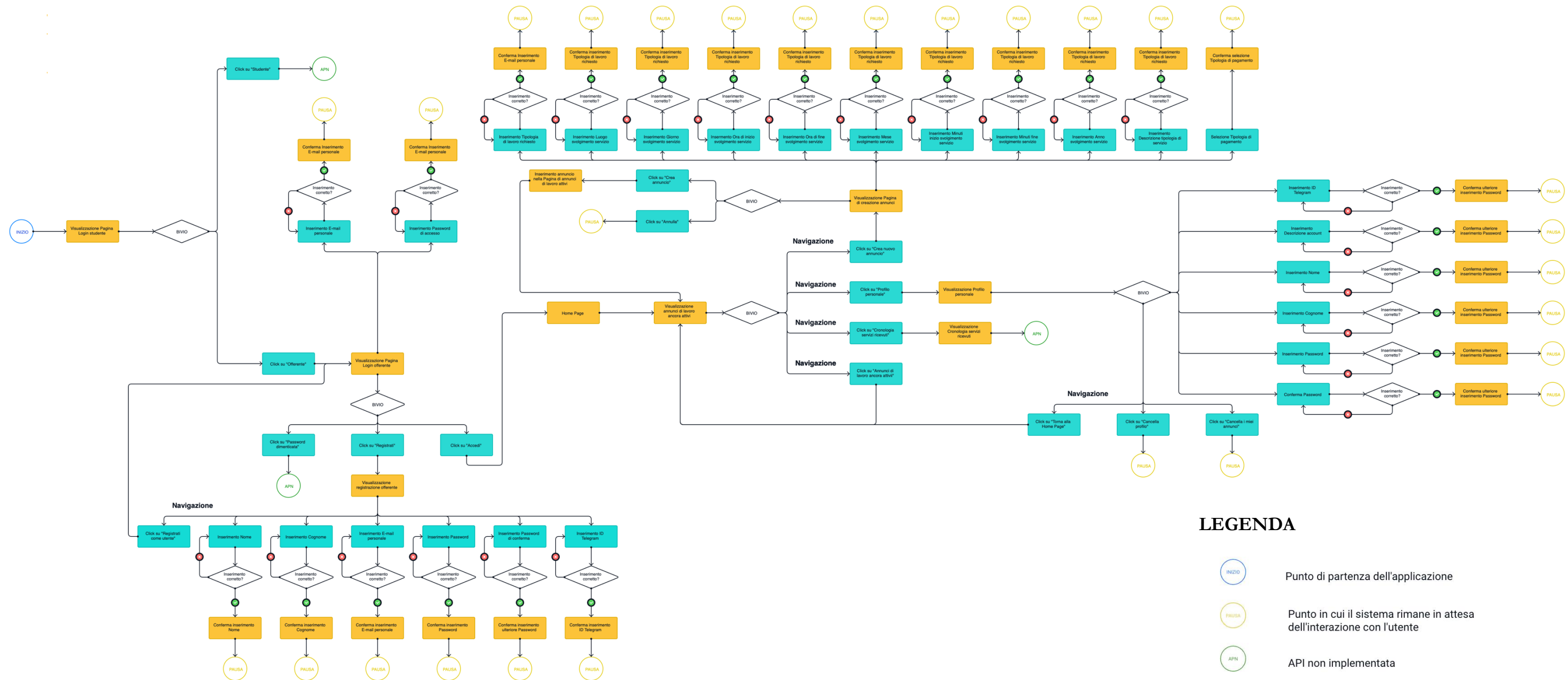


Immagine con definizione maggiore visibile [qui](#).

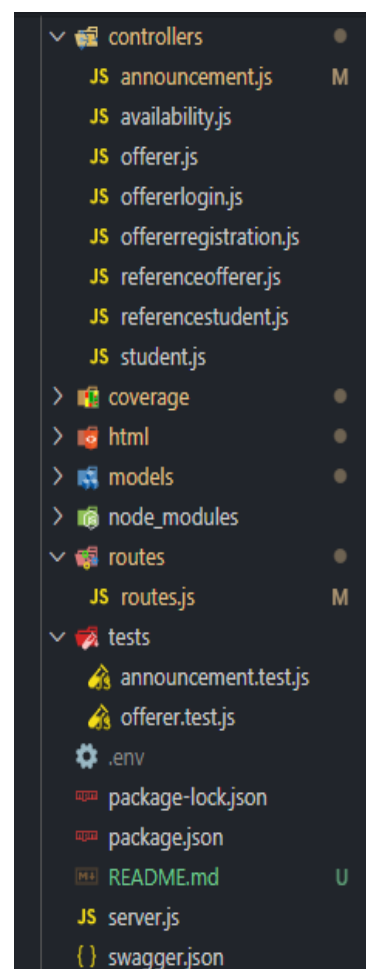
### 3. Application Implementation and Documentation

Nelle sezioni precedenti sono state identificate le *features* che devono essere implementate per l'applicazione con un'idea di come l'utente finale può utilizzarle nel suo flusso applicativo. L'applicazione è stata sviluppata utilizzando *NodeJS* per uno sviluppo efficiente ed efficace, *HTML* e *CSS* per quanto riguarda il front end e *JavaScript* per il back end e per le API. Infine, l'uso di *MongoDB* ha permesso il salvataggio e che tutti i dati vengano archiviati e gestiti in modo scalabile e flessibile. Usando queste tecnologie assieme siamo stati in grado di creare in parte un'applicazione potente e facile da usare che soddisfi le esigenze del nostro utente finale.

#### 3.1. Project Structure

Nella figura sottostante è resa visibile la struttura del progetto. Questo è composto dalle seguenti cartelle:

- **Controllers** è la cartella in cui sono presenti tutti i file di scripting per ogni modello, in particolare gli script che vengono utilizzati quando chiamati gli endpoint. Tali file sono scritti in linguaggio *JavaScript*.
- **Coverage** è la cartella generata automaticamente dopo aver effettuato per la prima volta il testing.
- **Html** è la cartella in cui sono presenti tutti i file *.html* e *.css* che vengono usati per mostrare i contenuti statici della pagina; inoltre, al suo interno sono presenti le immagini utilizzate nella pagina come, ad esempio, l'icona *.ico* oppure *.png* di sample per l'immagine dell'utente.
- **Models** è la cartella in cui sono presenti tutti gli schemi dei modelli definiti – attraverso gli schemi di *mongoose* – per essere contenuti nel database di *MongoDB*.
- **Node\_modules** è la cartella generata automaticamente quando viene eseguito il comando *npm init* nella directory del progetto. Al suo interno sono contenute tutte le dipendenze del progetto stesso e il suo aggiornamento avviene automaticamente ogni qualvolta viene eseguito il comando *npm install \*nome modulo\**.
- **Routes** è la cartella in cui è presente il file *routes.js* che contiene al suo interno tutti gli endpoint delle API realizzate per la web app. In particolare, viene usato il *Router di Express* per poter effettuare la chiamata all'API con il corrispondente metodo e URL.
- **Tests** è la cartella che contiene i file di testing effettuati con il modulo *supertest*. In particolare, sono state testate solamente le API che vengono utilizzate dalla nostra web app.



In aggiunta abbiamo il file `.env`, file contenente le variabili di ambiente del progetto ovvero le variabili utili per il funzionamento del sistema (link mongodb, porta server, ecc...) , nascoste nel sorgente dal file `.gitignore`.

Nel file `server.js` sono presenti i comandi per inizializzare la connessione al server mongoDB e per far partire l'applicazione e mettere in ascolto il server; inoltre al suo interno sono presenti i comandi per ereditare gli endpoint presenti in `routes.js` e sono definiti degli endpoint dell'applicazione per mostrare i contenuti statici presenti nella cartella `html`.

### 3.2 Project Dependencies

Di seguito è presente la schermata della sezione relativa ai vari moduli utilizzati e aggiunti al `Package.json`:

```
"dependencies": {
  "dotenv": "^16.0.3",
  "express": "^4.18.2",
  "jsonwebtoken": "^8.5.1",
  "mongoose": "^6.7.2",
  "multer": "^1.4.5-lts.1",
  "swagger-ui-express": "^4.6.0"
},
"devDependencies": {
  "jest": "^29.3.1",
  "supertest": "^6.3.3"
}
```

### 3.3 Project Data or DB

Per la gestione dei dati utili all'applicazione sono state definite una serie di strutture dati che presentano attributi diversi. Di seguito è presente una breve descrizione e un esempio di elemento presente nel database per ognuna delle tipologie di struttura.

#### 3.3.1 Offerers

È il tipo di dato rappresentante l'utente che si registra all'applicazione per richiedere lo svolgimento di un servizio a pagamento creando annunci di lavoro. I dati salvati per ogni elemento **Offerers** sono: *name*, *surname*, *email*, *idTelegram*, *confirmedAccount*, *password*, *relatedStudentsEmail*, *averageVotes* e *description*.

```
Offerers v {  
  name          string  
                required: true  
  surname       string  
                required: true  
  email         string  
                required: true  
  idtelegram    string  
                default:    
  confirmedaccount boolean  
                default: false  
  password      string  
                required: true  
  relatedStudentsEmail v {  
  }  
  averagevotes  number  
                default: 0  
  description   string  
                default:   
}
```

#### 3.3.2 Students

È il tipo di dato rappresentante l'utente studente che si registra all'applicazione ed è disposto a impegnare il proprio tempo libero in attività retribuite. I dati salvati per ogni elemento **Students** sono: *name*, *surname*, *email*, *city*, *idTelegram*, *confirmedAccount*, *university*, *countDoneServices*, *countDeclinedServices*, *countDeclinedServicesBefore24h*, *averageVotes*, *serviceToDo*, *serviceCandidate*, *serviceCronology* e *description*.

```
Students v {  
  name          string  
                required: true  
  surname       string  
                required: true  
  email         string  
                required: true  
  city          string  
                required: true  
  idtelegram    string  
                default:    
  confirmedaccount boolean  
                default: false  
  university    v {  
  }  
  countdoneservices number  
                default: 0  
  countdeclinedservices number  
                default: 0  
  countdeclinedservicesbefore24h number  
                default: 0  
  averagevotes  number  
                default: 0  
  servicetodo   v {  
  }  
  servicecandidate v {  
  }  
  servicecronology v {  
  }  
  description   string  
                default:   
}
```



### 3.3.3 Announcements

È il tipo di dato rappresentante l'annuncio creato dall'offerente. Ogni offerente ha infatti la possibilità di creare annunci di lavoro, specificando per ciascuno quelli che sono gli attributi che caratterizzano questo elemento. In particolare, i dati salvati per ogni elemento **Announcements** sono: *typeOfWork*, *city*, *date*, *startTime*, *endTime*, *typeOfPayment*, *description*, *candidates*, *offererEmail* e *workStudent*.

```
Announcements {
  typeofwork string
               required: true
  city        string
               required: true
  date        {
  starttime   {
  endtime     {
  typeofpayment typestringdefault
               Enum:
               [ contanti, paypal ]
  description  string
               default:
  candidates   {
  offereremail string
               required: true
  workerstudent string
               default:
}
```

### 3.3.4 ReferenceOfferers

È il tipo di dato rappresentante la referenza che è richiesta allo studente, una volta effettuato un servizio, rispetto all'attività svolta e al rapporto instaurato con l'offerente. In particolare, i dati salvati per ogni elemento **ReferenceOfferers** sono: *vote*, *review*, *studentEmail* e *offererEmail*.

```
ReferenceOfferers {
  vote      number
            required: true
  review    string
  studentemail string
            required: true
  offereremail string
            required: true
}
```

### 3.3.5 ReferenceStudents

È il tipo di dato rappresentante la referenza che è richiesta all'offerente, una volta effettuato un servizio dallo studente, rispetto a come il servizio è stato svolto. In particolare, i dati salvati per ogni elemento **ReferenceStudents** sono gli stessi di quelli di **ReferenceOfferers** ovvero: *vote*, *review*, *studentEmail* e *offererEmail*.

```
ReferenceStudents {
  vote      number
            required: true
  review    string
  offereremail string
            required: true
  studentemail string
            required: true
}
```

### 3.3.6 Availabilities

È il tipo di dato rappresentante la disponibilità specificata dall'utente studente nel “calendario di disponibilità” indicando gli orari settimanali in cui ha la possibilità di effettuare servizi. In particolare, i dati salvati per ogni elemento **Availabilities** sono: *date*, *startTime*, *endTime* e *frequencyType*.

```
Availabilities {
  date      {
  starttime {
  endtime   {
  frequencytype string
            Enum:
              [ ogni giorno, ogni settimana, ogni mese, ogni anno, ]
  studentemail string
            required: true
}
```

## 3.4 Project APIs

### 3.4.1 API description

Di seguito è presente una lista, e una breve descrizione, delle API che sono state sviluppate per questo progetto. Nella sezione [4. Api Documentation](#) è presente la descrizione completa per ogni API mentre il codice può essere trovato al link alla repository di GitHub in una sezione successiva.

- Offerers

**GET APIs:** segue la lista delle API di tipo *GET* che sono state sviluppate per il lato utente offerente dell'applicazione.

- ***E-mail***: recupera, in formato JSON, le informazioni riguardanti un utente di tipo offerente fornendo la sua e-mail personale.

**POST APIs:** segue la lista di API di tipo *POST* che sono state sviluppate per il lato utente offerente dell'applicazione.

- ***OffererCreation***: si occupa della creazione dell'account di un nuovo utente offerente controllando e inserendo tutti i dati forniti tramite formato JSON nel database del sistema.
- ***Login***: verifica se nel database del sistema è presente un utente offerente con le credenziali – ovvero con e-mail e password – fornite, se presente viene creato un token di autenticazione per l'utente stesso.
- ***IDTelegram***: permette la modifica dell'*ID Telegram* presente nell'area personale dell'utente inserito al momento della registrazione, o eventualmente successivamente cambiato.
- ***Description***: permette la modifica della descrizione personale presente nell'area personale dell'utente.
- ***Name***: permette la modifica dell'nome dell'utente presente nell'area personale dell'utente inserito al momento della registrazione o eventualmente successivamente cambiato.
- ***Surname***: permette la modifica del cognome dell'utente presente nell'area personale dell'utente inserito al momento della registrazione, o eventualmente successivamente cambiato.
- ***Password***: permette la modifica della password di accesso presente nell'area personale dell'utente inserita al momento della registrazione, o eventualmente successivamente cambiato, effettuando anche un controllo di soddisfacimento dei requisiti di *Strong Password* definiti nel documento **D1: Analisi dei requisiti** e di coincidenza tra password e conferma password.

**DELETE APIs:** segue la lista di API di tipo *DELETE* che sono state sviluppate per il lato utente offerente dell'applicazione.

- ***E-mail***: si occupa dell'eliminazione dell'account dell'utente offerente dato l'indirizzo e-mail e dell'eliminazione degli annunci attivi, se presenti.
  - [Announcements](#)

**GET APIs:** segue la lista delle API di tipo *GET* che sono state sviluppate per la gestione degli annunci dell'applicazione.

- ***E-mail***: recupera, in formato JSON, le informazioni riguardanti il/gli annuncio/i creati da un utente di tipo offerente fornendo la sua e-mail personale.

**POST APIs:** segue la lista delle API di tipo *POST* che sono state sviluppate per la gestione degli annunci dell'applicazione.

- **AnnouncementCreation**: si occupa della creazione di un nuovo annuncio di un utente offerente controllando e inserendo tutti i dati forniti tramite formato JSON nel database del sistema.

### 3.4.2 Resources Extraction from the Class Diagram

Di seguito è presente l'*UML API diagram*, ovvero una rappresentazione visiva di tutte le risorse che si vogliono gestire a livello di API e delle loro relazioni all'interno del progetto. Questo è stato utilizzato allo scopo di far comprendere come le diverse componenti di un sistema dipendono l'una dall'altra e come interagiscono.

La creazione del diagramma di estrazione delle risorse è stata effettuata a partire dal diagramma delle classi, realizzato nel documento **D3: Documento di architettura**, identificando le classi che rappresentano le risorse e le relazioni tra tali classi.

In particolare:

- il root è l'URL del sito;
- le risorse di primo livello sono quelle ricavate dalle classi e di queste vengono considerati anche i relativi attributi;
- le risorse di secondo livello sono quelle individuate dai metodi presenti all'interno delle corrispettive classi. Per ricavarle sono state identificate le funzioni delle classi del Class Diagram, sviluppato nel documento **D3: Documento di architettura**, necessarie per lo sviluppo delle API. Ovviamente ogni funzione prende in input un parametro che varia in base a come questa è stata implementata. Nel caso in cui i parametri siano più di due è stata utilizzata la stringa *body* per rappresentarli.

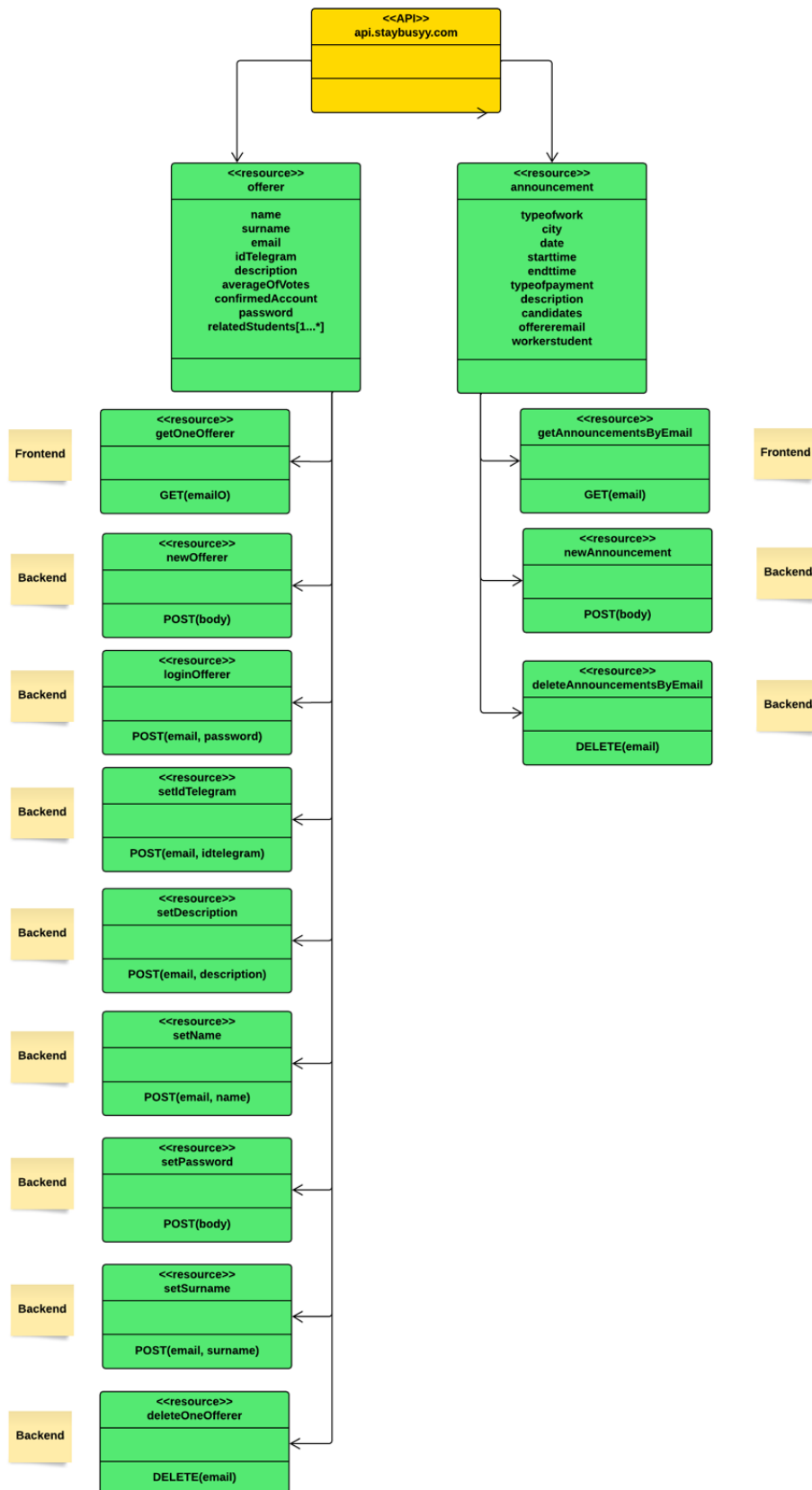
I passaggi svolti per la creazione di questo diagramma sono innanzitutto l'estrazione delle risorse dalle classi, poi l'estrazione delle risorse dai metodi, la specifica del tipo di funzione - che può essere di tipo *GET*, *POST* o *DELETE* (vedi sezione [4. Api Documentation](#)) - e dei parametri presi in input dai metodi.

L'ultimo passaggio che è stato effettuato è l'assegnazione ad ogni risorsa di una annotazione "FrontEnd" - nel caso in cui quella risorsa abbia lo scopo di mostrare qualcosa all'utente - oppure "BackEnd" - nell'eventualità in cui questa venga utilizzata per lo svolgimento di operazioni interne al sistema.

Un esempio esplicativo è la risorsa di secondo livello *loginOfferer*, la quale è stata individuata come di tipo BackEnd in quanto attraverso l'utilizzo di questa il sito presenta una pagina per inserire e-mail e password ma l'effetto di quell'API è un'operazione che avviene nel BackEnd. Questo è dovuto al fatto che i dati inseriti vengono presi in input dall'API e all'interno del codice di login è il BackEnd che svolge una serie di analisi e operazioni, ovvero effettua il collegamento con il database, controlla che username e password siano corretti e ritorna come response il risultato. È quindi etichettato come di tipo BackEnd perchè l'effetto principale dell'esecuzione dell'API viene eseguito nel BackEnd.

Un API di tipo FrontEnd ha invece lo scopo di mostrare qualcosa, non avviene alcun operazione nel database e non devono essere svolte analisi. Un esempio è rappresentato dall'utente che visualizza il suo profilo personale attraverso la risorsa *getOneOfferer*.

Si noti inoltre che nel diagramma sono state introdotte nuove classi rispetto a quelle definite nel documento **D3: Documento di Architettura**, classi necessarie per il corretto funzionamento delle API e della gestione dei dati.



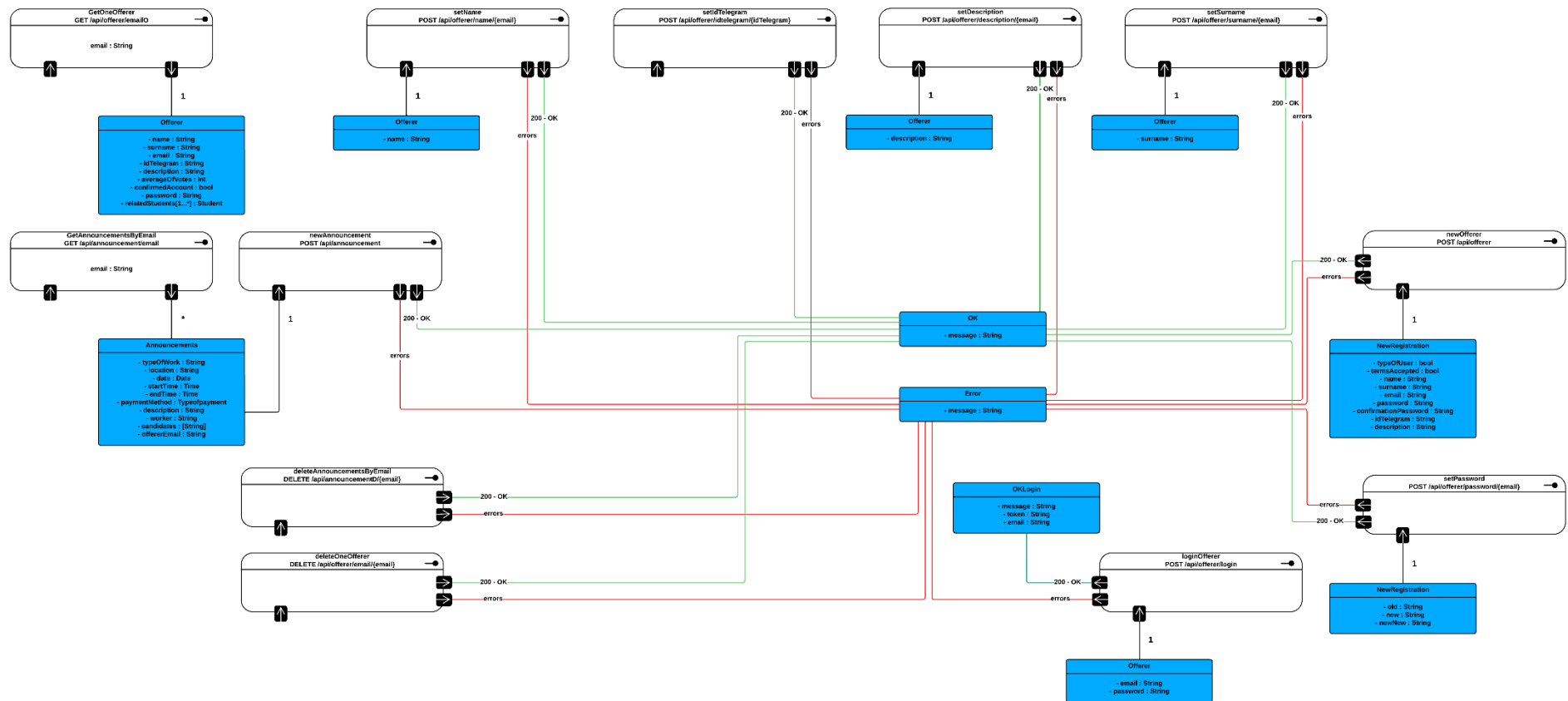
### 3.4.3 Resources Model

Il modello *Resources Model* viene utilizzato per astrarre e rappresentare graficamente le API necessarie per il funzionamento del sito. Tale modello esprime – per ogni risorsa – il nome, il metodo con cui vi è possibile accedervi e il *path* URL per raggiungerla.

Ogni risorsa è infatti l'unità fondamentale di una specifica API, un componente interno dell'applicazione. Questa ha il compito di elaborare dati – detti *request body*, ovvero le informazioni e i parametri che vengono passate a tale API per il suo funzionamento, e rappresentati graficamente da una freccia entrante nella risorsa – a seguito di una richiesta per fornire dei risultati – detti *response body*, ovvero la risposta dell'API all'azione effettuata, e rappresentati graficamente da una freccia uscente dalla risorsa.

Per il sito *StayBusy* sono state pensate (e successivamente realizzate) 12 API, di cui 9 per la classe **Offerer** e 3 per la classe **Announcement**. Al fine di semplificare la gestione di queste, i codici di stato utilizzati – indicati nel modello tramite etichette poste sull'associazione tra la risorsa e il corrispettivo *body response* – sono solamente di due tipologie: "200 - OK" e "errors".

Si noti inoltre che nel diagramma sono state introdotte nuove classi rispetto a quelle definite nel documento **D3: Documento di Architettura**, classi necessarie per il corretto funzionamento delle API e della gestione dei dati.



*Nota: al fine di rendere più chiara l'associazione tra ogni risorsa e i relativi body response, si è deciso di assegnare ad ogni associazione con codice di stato differente un colore; nello specifico, come visibile nel modello sovrastante, è stato scelto:*

- il colore rosso per le associazioni con codice di stato "errors"
- il colore verde per le associazioni con codice di stato "200 - OK"

## 4. API Documentation

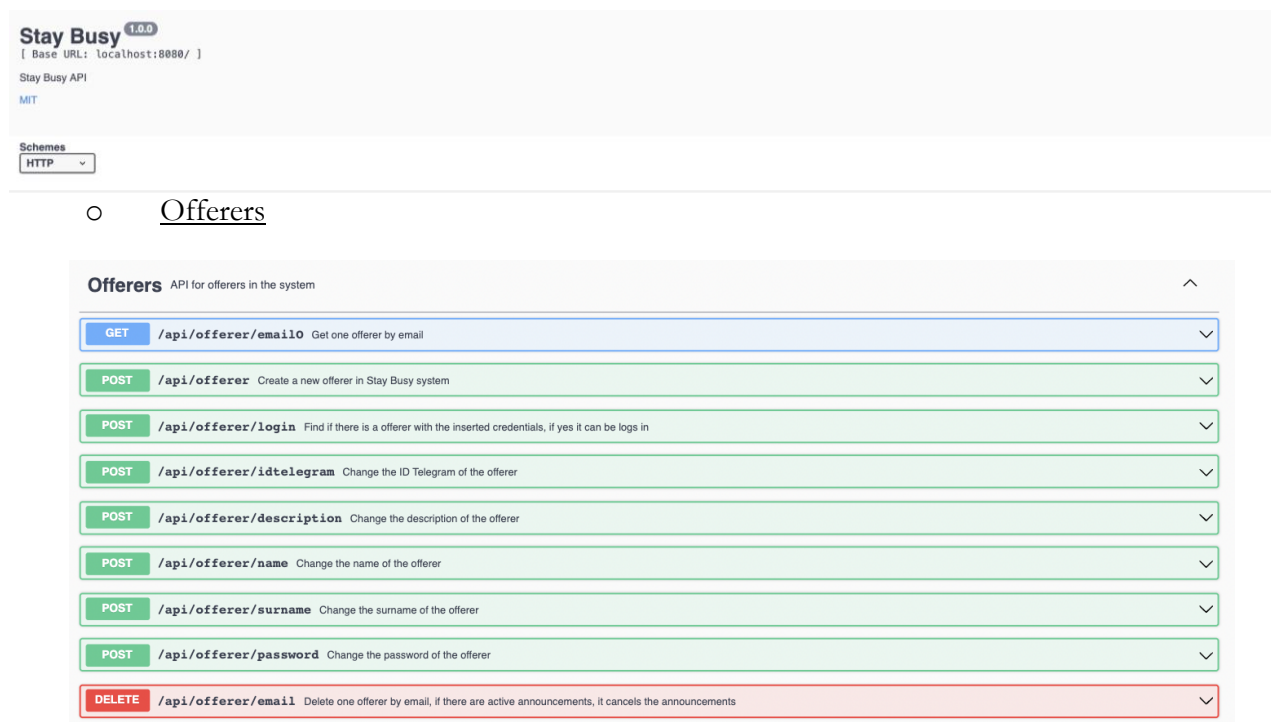
Le API locali fornite dall'applicazione *StayBusy* descritte nella sezione precedente sono state documentate utilizzando il modulo *NodeJS* chiamato **Swagger UI Express**. La documentazione relativa alle API è quindi direttamente disponibile a chiunque disponga del codice sorgente. Per poter generare l'endpoint dedicato alla presentazione delle API è stato usato *Swagger UI*, attraverso cui è stato possibile creare una pagina web dalle definizioni delle specifiche *OpenAPI*.

Di seguito è mostrata la pagina web relativa alla documentazione che presenta le API per la gestione dei dati della applicazione da noi realizzata. Nello sviluppo delle API vengono usate le funzioni *GET*, *POST* e *DELETE* per – rispettivamente – recuperare, creare ed eliminare dati. Nello specifico, la funzione *GET* viene utilizzata per recuperare i dati da un server, la funzione *POST* per creare una nuova risorsa sul server utilizzando informazioni fornite dall'utente in input mentre la funzione *DELETE* viene utilizzata per eliminare una risorsa dal server - ovvero per eliminare un utente, un annuncio o un altro tipo di risorsa che non è più necessario. In generale, queste vengono utilizzate in modo tale da consentire a diverse applicazioni e servizi di interagire tra loro, consentendo quindi la creazione di sistemi più potenti e flessibili permettendo l'utilizzo di dati e di funzionalità fornite da altri sistemi. L'endpoint da invocare per raggiungere tale documentazione è:

**<https://staybusy.herokuapp.com/api-docs>**

Si noti che non sono state implementate tutte le API ma solamente quelle necessarie per il funzionamento del sito dal lato dell'utente offerente per le motivazioni citate precedentemente nella sezione [1. Scopo del documento](#).

Di seguito riportiamo l'immagine della pagina web relativa alla documentazione:





Di seguito vengono descritti l'utilizzo e il funzionamento delle API più importanti con l'aiuto della documentazione.

#### 4.1 GET /api/offerer/emailO

The screenshot displays a REST client interface with the following sections:

- Parameters:** A table with columns 'Name' and 'Description'. It contains one parameter: 'email' (required, string, query) with the value 'prova@gmail.com' entered in a text box.
- Buttons:** 'Execute' (blue) and 'Clear' (white) buttons.
- Responses:** A section showing the response details for the executed request.
  - Request URL:** `https://staybusy.herokuapp.com/api/offerer/emailO?email=prova@gmail.com`
  - Server response:**
    - Code:** 200
    - Response body:** A JSON object: 

```
{  "_id": "63a0174d143925d9b079e295",  "name": "Riccardo",  "surname": "Fusiello",  "email": "prova@gmail.com",  "telegram": "",  "confirmedaccount": false,  "password": "Riccardo-2002",  "relatedStudentsEmail": [],  "averagevotes": 0,  "description": "Ciao! Sono uno studente universitario e mi piace molto l'informatica.",  "__v": 0}
```
    - Response headers:**

```
content-length: 304
content-type: application/json; charset=utf-8
date: Wed, 28 Dec 2022 17:21:22 GMT
etag: W/"130-LVt6XKh6yH0bq5fkTPK9y/MC0ZBE"
server: Cowboy
via: 1.1 vegur
x-powered-by: Express
```
  - Responses table:** A table with columns 'Code' and 'Description'. It shows a 200 status with the description 'OK'.
  - Example Value / Model:** A JSON schema: 

```
{  "name": "string",  "surname": "string",  "email": "string",  "telegram": "",  "confirmedaccount": false,  "password": "string",  "relatedStudentsEmail": "string",  "averagevotes": 0,  "description": ""}
```
  - 404:** A row showing the status '404' with the description 'Offerer doesn't exist'.

Questa API di tipo *GET* si occupa di effettuare la connessione con il database *MongoDB* eseguendo la funzione *findOne()* con l'e-mail che l'utente passa come *path parameter*, ovvero come parametro di ingresso. Questa funzione restituisce i dati di registrazione dell'utente nel caso in cui egli si sia precedentemente registrato mentre, in caso contrario, viene restituito il messaggio "*Offerer doesn't exist*".

Di seguito il codice della funzione utilizzata:

```
const getOneOfferer = (req, res, next) => {
  let email = req.query.email;

  Offerer.findOne({email : email}, (err, data) => {
    if(err || !data) return res.status(404).json({message : "Offerer doesn't exist"});
    return res.status(200).json(data);
  })
}
```

## 4.2 POST /api/offerer

POST /api/offerer Create a new offerer in Stay Busy system

Parameters

Cancel

Name	Description
<b>parameterstosent</b> * required (body)	Parameters to be sent Example : OrderedMap { "typeofuser": false, "termsaccepted": true, "name": "Riccardo", "surname": "Fusiello", "email": "prova@gmail.com", "password": "Prova-1990", "confirmationpassword": "Prova-1990", "idtelegram": "", "description": "" }

Edit Value | Model

```
{
  "typeofuser": false,
  "termsaccepted": true,
  "name": "Riccardo",
  "surname": "Fusiello",
  "email": "T33@gmail.com",
  "password": "W3-L0V3-U-BuCH14R0N3",
  "confirmationpassword": "W3-L0V3-U-BuCH14R0N3",
  "idtelegram": "",
  "description": ""
}
```

Cancel

Parameter content type  
application/json

Execute Clear

Responses

Response content type application/json

Curl

```
curl -X 'POST' \
  'http://localhost:8080/api/offerer' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "typeofuser": false,
    "termsaccepted": true,
    "name": "Riccardo",
    "surname": "Fusiello",
    "email": "T33@gmail.com",
    "password": "W3-L0V3-U-BuCH14R0N3",
    "confirmationpassword": "W3-L0V3-U-BuCH14R0N3",
    "idtelegram": "",
    "description": ""
  }'
```

Request URL

http://localhost:8080/api/offerer

Server response

Code Details

200

Response body

```
{
  "message": "Offerer already exists"
}
```

Download

Response headers	
<pre> connection: keep-alive content-length: 36 content-type: application/json; charset=utf-8 date: Wed,14 Dec 2022 16:53:51 GMT etag: W/"24-7ofW8AIsj0JN4j+oVg8Mq8W0Y0" keep-alive: timeout=5 x-powered-by: Express </pre>	
Responses	
Code	Description
200	OK
Example Value   Model	
<pre> {   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" } </pre>	
404	NOT OK
Example Value   Model	
<pre> {   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" } </pre>	

Questa API di tipo *POST* gestisce la creazione dell'utente offerente. In particolare, si occupa di verificare che i dati inseriti dall'utente rispettino le regole implementate dall'applicazione e, attraverso il metodo *findOne()* – a cui viene passata come parametro di ingresso l'e-mail personale – ha il compito di verificare che nel database non sia già presente un utente registrato con lo stesso indirizzo e-mail. In quest'ultimo caso, l'API restituisce il messaggio *"Offerer already exists"*.

Di seguito il codice della funzione utilizzata:

```

const newOfferer = (req, res, next) => {
  var typeofuser = req.body.typeofuser;
  var termsaccepted = req.body.termsaccepted;
  var name = req.body.name;
  var surname = req.body.surname;
  var email = req.body.email;
  var password = req.body.password;
  var confirmationpassword = req.body.confirmationpassword;
  var idtelegram = req.body.idtelegram;
  var description = req.body.description;

  if(typeofuser == "true"){
    return res.json({message : "Wrong type of user"});
  }
  if(termsaccepted == "false"){
    return res.json({message : "You need to accept the Terms & Conditions"});
  }
  if(password != confirmationpassword){
    return res.json({message : "Two inserted password are not equal"});
  }

  if(!checkPasswordRequirements(password)){
    return res.json({message : "Wrong password"});
  }

  Offerer.findOne({email : req.body.email}, (err, data) => {
    if(!data){
      const newOfferer = new Offerer({
        name : name,
        surname : surname,
        confirmedaccount : "true",
        email : email,
        idtelegram : idtelegram,

```

```

        description : description,
        password : password,
        relatedStudents : []
    })
    newOfferer.save((err, data) => {
        if(err) return res.json({Error : err});
        return res.json({message : "OK"});
    })
} else {
    if(err) return res.json("Something went wrong, please try again: " + err);
    return res.json({message: "Offerer already exists"});
}
};

```

### 4.3 POST /api/offerer/login

**POST** **/api/offerer/login** Find if there is a offerer with the inserted credentials, if yes it can be logs in

Try it out

Name	Description
<b>parameters</b> <small>* required</small> <small>(body)</small>	Parameters to be sent Example : <code>OrderedMap { "email": "prova@gmail.com", "password": "Prova-1990" }</code> Example Value   Model <pre>{   "email": "prova@gmail.com",   "password": "Prova-1990" }</pre> Parameter content type application/json

Responses

Response content type application/json

Code	Description
200	OK Example Value   Model <pre>{   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" }</pre>
404	NOT OK Example Value   Model <pre>{   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" }</pre>

Questa API di tipo *POST* gestisce il controllo dell'esistenza dell'account di un utente – ovvero il controllo di e-mail e password inseriti – restituendo il messaggio "*NOT OK*" nel caso in cui queste non coincidano, siano inserite in modo errato o in generale non siano presenti all'interno del database.

Di seguito il codice della funzione utilizzata:

```
const loginOfferer = (req, res, next) => {  
  let email = req.body.email;  
  let password = req.body.password;  
  
  Offerer.findOne({email : email, password : password}, (err, data) => {  
    if(!data){  
      return res.status(404).json({message : "NOT OK"});  
    }else{  
      var payload = {  
        email: data.email,  
        id: data._id  
      }  
      var options = {  
        expiresIn: 86400  
      }  
      var token = jwt.sign(payload, process.env.SUPER_SECRET, options);  
  
      return res.status(200).json({  
        message: 'OK',  
        token: token,  
        email: data.email  
      });  
    }  
  })  
};
```

Le successive API di tipo *POST* sono praticamente equivalenti, e differiscono solamente per i file *JSON* restituiti diversi. Il loro funzionamento è molto simile – ovvero permettono tutte la modifica di uno specifico dato del proprio profilo personale inserito dall'offerente al momento della registrazione ed eventualmente successivamente modificato. Ciò che quindi varia sono solamente i parametri in ingresso passati alla funzione *findOneAndUpdate()*, cioè i parametri che verranno poi modificati attraverso l'utilizzo di quest'ultima.

Per non creare un documento troppo lungo, essendoci il codice disponibile su *GitHub* e dato che, come detto, le successive API *POST* sono tutte abbastanza simili, si riporta solamente l'esempio dell'API di modifica del nome utente.

#### 4.4 POST /api/offerer/name

The screenshot shows a REST client interface for a POST request to `/api/offerer/name`. The description is "Change the name of the offerer".

**Parameters:**

Name	Description
<b>email</b> * required string (query)	Parameters to be sent T33@gmail.com
<b>name</b> * required (body)	Parameters to be sent Example : OrderedMap { "name": "Prova" }

**Body:**

```
{  
  "name": "Prova"  
}
```

**Parameter content type:** application/json

Execute

Clear

Responses

Response content type application/json

Curl

```
curl -X 'POST' \
  'http://localhost:8080/api/offerer/name?email=T33%40gmail.com' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Prava"
  }'
```

Request URL

http://localhost:8080/api/offerer/name?email=T33%40gmail.com

Server response

Code

Details

200

Response body

{
 "message": "OK"
}

Download

Response headers

connection: keep-alive  
 content-length: 16  
 content-type: application/json; charset=utf-8  
 date: Wed, 14 Dec 2022 20:13:39 GMT  
 etag: W/"10-MzBdybUxsfG0spb6vyp7iPMFo"  
 keep-alive: timeout=5  
 x-powered-by: Express

Responses

Code

Description

200

OK

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "email": "string",
  "idtelegram": "",
  "confirmedaccount": false,
  "password": "string",
  "relatedStudentsEmail": "string",
  "averagevotes": 0,
  "description": ""
}
```

404

NOT OK

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "email": "string",
  "idtelegram": "",
  "confirmedaccount": false,
  "password": "string",
  "relatedStudentsEmail": "string",
  "averagevotes": 0,
  "description": ""
}
```

Di seguito il codice della funzione utilizzata:

```
const setName = (req, res, next) => {
  let email = req.query.email;

  Offerer.findOneAndUpdate({email : email}, {name : req.body.name}, {new: true}, function(err, response) {
    if (err || !response) {
      return res.status(404).json({message : "NOT OK"});
    } else {
      return res.status(200).json({message : "OK"});
    }
  });
}
```

#### 4.5 POST /api/offerer/password

POST

/api/offerer/password

Change the password of the offerer

Parameters

Cancel

POST

/api/offerer/password

Change the password of the offerer

Try it out

Parameters

Name	Description
<b>email</b> * required string (query)	Parameters to be sent Example : <a href="#">prova@gmail.com</a> <input type="text" value="email"/>
<b>password</b> * required (body)	Parameters to be sent Example : <code>OrderedMap { "old": "Prova-1990", "new": "Prova-2020", "newnew": "Prova-2020" }</code> Example Value   Model <pre>{   "old": "Prova-1990",   "new": "Prova-2020",   "newnew": "Prova-2020" }</pre> Parameter content type <input type="text" value="application/json"/>

Responses

Response content type

Code	Description
200	OK Example Value   Model <pre>{   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" }</pre>
400	BAD REQUEST
404	NOT OK

L'API di tipo *POST* che permette di modificare la password di accesso al sistema passando come parametro in ingresso l'e-mail dell'offerente è leggermente differente rispetto a quelle precedenti in quanto è stato necessario anche implementare un controllo che verificasse che la password fosse inserita correttamente due volte consecutive.

Di seguito il codice della funzione utilizzata:

```
const setPassword = (req, res, next) => {
  let email = req.query.email;
  let oldPassword = req.body.old;
  let newPassword = req.body.new;
  let newPasswordConfirm = req.body.newnew;

  if(newPassword !== newPasswordConfirm){
    return res.status(400).json({message : "BAD REQUEST"});
  }

  Offerer.findOneAndUpdate({email : email, password : oldPassword},{password : newPassword, {new: true}, function(err, response) {
    if (err || !response) {
      return res.status(404).json({message : "NOT OK"});
    } else {
      return res.status(200).json({message : "OK"});
    }
  });
}
```

## 4.6 DELETE /api/offerer/email

**DELETE** /api/offerer/email Delete one offerer by email, if there are active announcements, it cancels the announcements

Parameters

Name	Description
<b>email</b> * required string (query)	Parameters to be sent

T33@gmail.com

Execute

Clear

Responses

Response content type application/json

Curl

curl -X 'DELETE' \  
'http://localhost:8080/api/offerer/email?email=T33W4@gmail.com' \  
-H 'accept: application/json'

Request URL

http://localhost:8080/api/offerer/email?email=T33W4@gmail.com

Server response

Code	Details
200	<div><div>Response body</div><div>{   "message": "OK" }</div><div>Download</div></div> <div><div>Response headers</div><div>connection: keep-alive content-length: 14 content-type: application/json; charset=utf-8 date: Wed, 14 Dec 2022 20:15:18 GMT etag: W/"10-RxB4yqMLcx6Qdnp8b8vgp7LPMFo" keep-alive: timeout=5 x-powered-by: Express</div></div>

Responses

Code	Description
200	OK

Example Value | Model

{  
  "name": "string",  
  "surname": "string",  
  "email": "string",  
  "idtelegram": "",  
  "confirmedaccount": false,  
  "password": "string",  
  "relatedStudentsEmail": "string",  
  "averagevotes": @,  
  "description": ""  
}

404 NOT OK |

Example Value | Model

{  
  "name": "string",  
  "surname": "string",  
  "email": "string",  
  "idtelegram": "",  
  "confirmedaccount": false,  
  "password": "string",  
  "relatedStudentsEmail": "string",  
  "averagevotes": @,  
  "description": ""  
}

Questa API di tipo *DELETE* si occupa dell'eliminazione dell'account dell'offerente attraverso l'utilizzo della funzione *deleteOne()* passando come parametro in ingresso la propria e-mail personale.

Di seguito il codice della funzione utilizzata:

T33 24



```
const deleteOneOfferer = (req, res, next) => {
  let email = req.query.email;

  Announcement.deleteMany({offereremail : email}, (err, data1) => {
  })

  Offerer.deleteOne({email : email}, (err, data) => {
    if(data.deletedCount === 0) return res.status(404).json({message : "NOT OK"});
    return res.status(200).json({message : "OK"});
  })
};
```

## ○ Announcements

Announcements API for announcements in the system	
GET	/api/announcement/email Get the announcement of an offerer by email
POST	/api/announcement Create an announcement of an offerer by email

### 4.7 GET /api/announcement/email

GET /api/announcement/email Get the announcement of an offerer by email

Parameters

Name	Description
email * required	Parameters to be sent
string (query)	<input type="text" value="G_T33@gmail.com"/>

Execute

Clear

Responses

Response content type application/json

Curl

curl -X 'GET' \ 'http://localhost:8080/api/announcement/email?email=G\_T33@gmail.com' \ -H 'accept: application/json'

Request URL

http://localhost:8080/api/announcement/email?email=G\_T33@gmail.com

Server response

Code	Details
200	<div>Response body</div> <pre>{   "city": "Trento",   "typeofpayment": "paypal",   "description": "Domani non riesco a portare la spesa ai miei nonni. Cerco qualcuno disponibile!",   "offereremail": "G_T33@gmail.com",   "candidates": [],   "_v": 0 }, {   "date": {     "day": 22,     "month": 12,     "year": 2022   },   "starttime": {     "hour": 14,     "minutes": 30   },   "endtime": {     "hour": 16,     "minutes": 30   },   "_id": "639af42d44ef3e76fc0bbfc8",   "typeOfWork": "Tutor di programmazione per la fisica di base" } </pre> <div>Download</div>

Response headers

```
connection: keep-alive
content-length: 1268
content-type: application/json; charset=utf-8
date: Thu, 15 Dec 2022 10:17:37 GMT
etag: W/"4f4-shVv3tpoUlpnxzGO+4UrkPPuo"
keep-alive: timeout=5
x-powered-by: Express
```

Responses

Code	Description
200	OK
404	NOT OK

Questa API di tipo *GET* si occupa di effettuare la connessione con il database *MongoDB* eseguendo la funzione *findOne()* con il *typeOfWork* che l'utente passa come *path parameter*, ovvero come parametro di ingresso. Questa funzione restituisce i dati relativi all'annuncio creato dall'offerente nel caso in cui egli abbia effettivamente precedentemente creato tale annuncio mentre, in caso contrario, viene restituito il messaggio "*Announcement doesn't exist*".

Di seguito il codice delle funzioni utilizzate:

```
const getAllAnnouncementByEmail = (req, res, next) => {
  Announcement.find({offereremail : req.query.email}, (err, data) => {
    if(err || !data) return res.status(404).json({message : "NOT OK"});
    return res.status(200).json(data);
  })
}
```

```
Announcement.findOne({typeofwork : req.body.typeofwork}, (err, data) => {
  if(!data){
    const newAnnouncement = new Announcement({
      typeofwork : req.body.typeofwork,
      city : req.body.city,
      date : {
        day : req.body.day,
        month : req.body.month,
        year : req.body.year
      },
      starttime : {
        hour : req.body.hour1,
        minutes : req.body.minutes1
      },
      endtttime : {
        hour : req.body.hour2,
        minutes : req.body.minutes2
      },
      typeofpayment : req.body.typeofpayment,
      description : req.body.description,
      offereremail : req.body.offereremail
    })
    newAnnouncement.save((err, data) => {
      if(err || !data) return res.json({message : "NOT OK"});
      return res.json({message : "OK"});
    })
  }else{
    if(err || !data) return res.json({message : "NOT OK"});
    return res.json({message : "NOT OK"});
  }
})
```

## 4.8 POST /api/announcement

POST

/api/announcement

Create an announcement of an offerer by email

^

Parameters

Try it out

Name	Description
<b>parameterstosent</b> <small>required</small> <small>(body)</small>	<p>Parameters to be sent</p> <p><i>Example</i> : <code>OrderedMap { "hour1": 18, "hour2": 20, "minutes1": 30, "minutes2": 30, "day": 15, "month": 12, "year": 2023, "typeofwork": "Prova annuncio", "city": "Prova città", "typeofpayment": "contanti", "description": "", "offereremail": "prova@gmail.com" }</code></p> <p>Example Value   Model</p> <pre>{   "hour1": 18,   "hour2": 20,   "minutes1": 30,   "minutes2": 30,   "day": 15,   "month": 12,   "year": 2023,   "typeofwork": "Prova annuncio",   "city": "Prova città",   "typeofpayment": "contanti",   "description": "",   "offereremail": "prova@gmail.com" }</pre> <p>Parameter content type</p> <div>application/json</div>

Responses

Response content type

application/json

Code	Description
200	<p>OK</p> <p>Example Value   Model</p> <pre>{   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" }</pre>
404	<p>NOT OK</p> <p>Example Value   Model</p> <pre>{   "name": "string",   "surname": "string",   "email": "string",   "idtelegram": "",   "confirmedaccount": false,   "password": "string",   "relatedStudentsEmail": "string",   "averagevotes": 0,   "description": "" }</pre>

Questa API di tipo *POST* si occupa della creazione di un annuncio. In particolare, sono stati implementati tutti i vincoli imposti sull'orario di inizio e fine de servizio richiesto definiti nel documento **D3: Documento di Architettura**.

Di seguito il codice della funzione utilizzata:

```

const newAnnouncement = (req, res, next) => {
  var hour1 = req.body.hour1;
  var hour2 = req.body.hour2;
  var minutes1 = req.body.minutes1;
  var minutes2 = req.body.minutes2;
  var day = req.body.day;
  var month = req.body.month;
  var year = req.body.year;
  if(hour1 < 0 || hour1 > 23){
    return res.json({message : "Wrong hour start time inserted"});
  }
  if(hour2 < 0 || hour2 > 23){
    return res.json({message : "Wrong hour end time inserted"});
  }
  if(minutes1 < 0 || minutes1 > 59){
    return res.json({message : "Wrong minute start time inserted"});
  }
  if(minutes2 < 0 || minutes2 > 59){
    return res.json({message : "Wrong hour end time inserted"});
  }
  if(hour1 > hour2){
    return res.json({message : "Wrong hours inserted"});
  }else if(hour1 == hour2 && minutes1 > minutes2){
    return res.json({message : "Wrong minutes inserted"});
  }
  if(year < 1850){
    return res.json({message : "Wrong year inserted"});
  }
  if(month < 1 || month > 12){
    return res.json({message : "Wrong month inserted"});
  }
  if(day < 1 || day > 31){
    return res.json({message : "Wrong day inserted"});
  }
  Announcement.findOne({typeofwork : req.body.typeofwork}, (err, data) => {
    if(!data){
      const newAnnouncement = new Announcement({
        typeofwork : req.body.typeofwork,
        city : req.body.city,
        date : {
          day : req.body.day,
          month : req.body.month,
          year : req.body.year
        },
        starttime : {
          hour : req.body.hour1,
          minutes : req.body.minutes1
        },
        endtime : {
          hour : req.body.hour2,
          minutes : req.body.minutes2
        },
        typeofpayment : req.body.typeofpayment,
        description : req.body.description,
        offereremail : req.body.offereremail
      })
      newAnnouncement.save((err, data) => {
        if(err || !data) return res.json({message : "NOT OK"});
        return res.json({message : "OK"});
      })
    }else{
      if(err || !data) return res.json({message : "NOT OK"});
      return res.json({message : "NOT OK"});
    }
  })
};

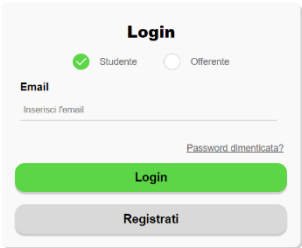
```

## 5. FrontEnd Implementation

Questa sezione del documento fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati fornite nell'applicazione *StayBusy* all'utente offerente attraverso l'utilizzo delle varie API sviluppate e precedentemente descritte.

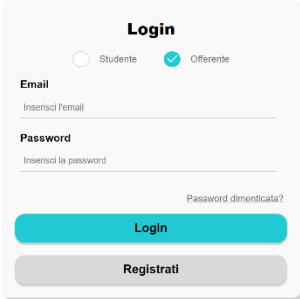
Nonostante nel documento **D1: Analisi dei requisiti** sia stato scritto che la prima schermata mostrata dall'applicazione è la pagina di **Login dell'account studente**, in questa implementazione parziale è stato deciso di mostrare come pagina index il **Login dell'account offerente** in quanto le funzionalità dello studente non sono state realizzate - per i motivi precedentemente specificati nella sezione [1. Scopo del documento](#) - bensì sono state implementate di *FrontEnd* solamente le schede **Login dell'account studente** e **Registrazione dell'account studente**.

Di seguito è presente la schermata del **Login dell'account studente**.



The screenshot shows a login form titled "Login". At the top, there are two radio buttons: "Studente" (selected with a green checkmark) and "Offerente". Below this is an "Email" label and a text input field with the placeholder "Inserisci l'email". To the right of the input field is a link "Password dimenticata?". At the bottom of the form are two buttons: a green "Login" button and a grey "Registrati" button.

Qui è possibile selezionare l'opzione offerente e visualizzare la pagina **Login dell'account offerente** da cui è possibile accedere all'applicazione come utente offerente di servizi.



The image shows a login form titled "Login" centered on a light gray background. At the top, there are two radio buttons: "Studente" (unselected) and "Offertante" (selected with a blue checkmark). Below this, there are two input fields: "Email" with the placeholder text "Inserisci l'email" and "Password" with the placeholder text "Inserisci la password". To the right of the password field is a link that says "Password dimenticata?". At the bottom of the form are two buttons: a blue "Login" button and a gray "Registrati" button. In the bottom right corner of the overall page, there is a small circular logo featuring a stylized bird or figure.

Nella parte centrale viene richiesto l'inserimento dell'*e-mail di accesso* e della *password* impostate dall'utente al momento della registrazione. Cliccando su *Password dimenticata?* l'utente può inoltre recuperare la password di accesso nel caso in cui se la sia dimenticata. Questa funzionalità non è ancora stata implementata ed infatti cliccandoci sopra il sito per il momento mostrerà per il momento un messaggio di alert con scritto "*Funzionalità non ancora implementata*".

In basso è presente il pulsante *Accedi* che, se premuto e se inserite le credenziali corrette, consente di effettuare l'accesso all'applicazione con i dati precedentemente inseriti. Se i dati di accesso fossero errati comparirebbe il messaggio di alert "*Attenzione, credenziali errate*". Per fornire questo tipo di servizio vengono utilizzate l'*API login* e l'*API email*.

Se invece non si è mai effettuato l'accesso è possibile effettuare la registrazione cliccando l'apposito bottone *Registrati* che invece manda l'utente alla schermata di registrazione.

Nella pagina di **Registrazione** dell'utente l'offerente dovrà fornire alcune informazioni, quali *nome*, *cognome*, *e-mail* con cui si desidera effettuare la registrazione, *password* che rispetti i *requisiti di Strong Password* definiti nel documento **D1: Analisi dei requisiti**, *conferma della password* precedentemente inserita e *ID Telegram*. Per completare la registrazione è necessario accettare i *Termini e Condizioni e Privacy Policy* spuntando l'apposita casella, altrimenti comparirà un messaggio di alert con scritto " *Bisogna accettare i termini e le condizioni*". Per fornire questo tipo di servizio viene utilizzata l'*API OffererCreation*.

Anche in questo caso, l'utente ha la possibilità di selezionare l'opzione di registrazione come utente studente. Per effettuare la registrazione è richiesto allo studente il solo inserimento dell'*Ateneo universitario di appartenenza* poiché premendo il tasto *Accedi* lo studente verrà indirizzato alla pagina di accesso della propria università, dove avverrà il vero e proprio login con credenziali.





**Registrazione**

☒ Studente ☐ Offerente

**Università**

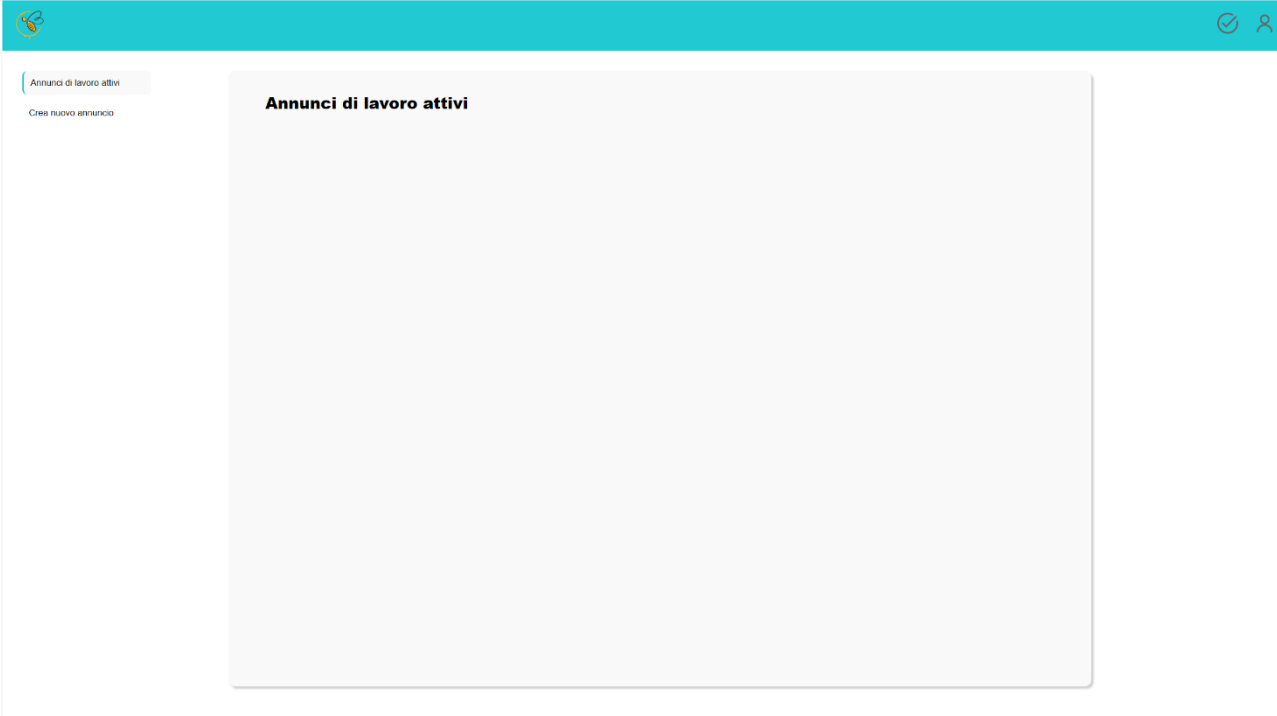
Inserisci nome università

☐ Accetto i Termini e Condizioni e Privacy Policy

**Registrati come studente**

**Accedi**

Effettuati correttamente login o registrazione, l'utente offerente visualizza l'**Home Page** del sito. Questa propone in primo piano la lista di annunci creati dall'offerente stesso - ovviamente inizialmente vuota - mostrando per ognuno i dati inseriti dall'offerente al momento della creazione dell'annuncio, il numero di persone da cui è stato visualizzato, gli studenti che effettivamente si sono proposti per svolgere il lavoro e lo stato dell'annuncio (assegnato, non assegnato). Cliccando sull'anteprima dell'annuncio, l'offerente viene rimandato all'annuncio vero e proprio.






Annunci di lavoro attivi


Crea nuovo annuncio

**Annunci di lavoro attivi**

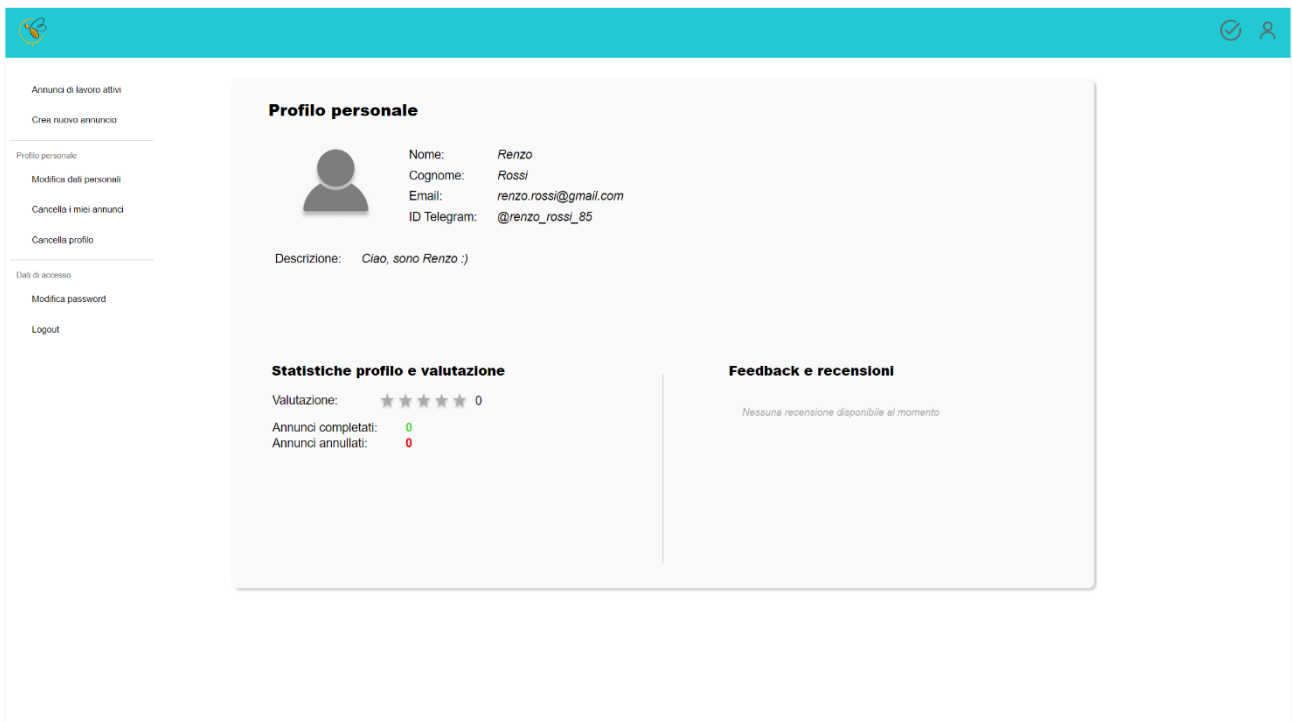
Le funzionalità Persone che hanno visualizzato l'annuncio, Richieste di effettuare il lavoro e Stato non sono per ora state implementate quindi cliccando su una di esse compare il messaggio di alert "*Funzionalità non ancora implementata*".

La barra superiore della **Home page** mette a disposizione all'offerente diverse funzionalità quali, rispettivamente da sinistra verso destra:

-  tornare alla Home page
-  visualizzare la cronologia dei servizi ricevuti;
-  visualizzare il proprio profilo, eventualmente modificandolo.

La schermata di **Cronologia dei servizi ricevuti** mostra all'offerente la lista di tutti i servizi non più attivi al momento della visualizzazione - ovvero dei servizi eliminati o che sono stati svolti in passato da uno studente. Questa funzionalità per ora non è stata implementata quindi cliccando sopra all'icona  comparirà il messaggio di alert "*Funzione non ancora implementata*".

La pagina di **Profilo personale** mostra tutti i dati personali inseriti dall'utente, la possibilità di tornare alla pagina Home page, di effettuare il log out dall'applicazione, di cancellare il profilo ed infine di eliminare tutti gli annunci ancora attivi. In questi due ultimi casi è possibile effettuare queste operazioni solo se vengono rispettate le imposizioni descritte nel documento **D2: Specifica dei requisiti**.



All'offerente è proposta la possibilità di modificare tutti i dati personali impostati al momento della registrazione - ovvero *nome*, *cognome*, *ID Telegram* e *descrizione* - a differenza del campo *e-mail* che non può essere modificato. Cliccando il pulsante *Salva* il sistema effettua la modifica definitiva dei dati personali che possono successivamente essere cambiati in ogni momento mentre cliccando il tasto *Annulla* la

modifica non verrà effettuata. Per fornire questo tipo di servizi vengono utilizzate l'*API IDTelegram*, l'*API Description*, l'*API name*, l'*API surname* e l'*API e-mail* della sezione *Delete*.

The screenshot shows a web application interface with a teal header bar containing a logo and user icons. A left sidebar lists navigation options: 'Annuncio di lavoro attivo', 'Crea nuovo annuncio', 'Profilo personale' (with sub-options 'Modifica dati personali', 'Cancella i miei annunci', and 'Cancella profilo'), and 'Dati di accesso' (with sub-options 'Modifica password' and 'Logout'). The main content area is titled 'Profilo personale - Modifica dati personali' and features a user profile icon with a yellow edit pencil. To the right of the icon are input fields for 'Nome' (filled with 'Renzo'), 'Cognome' (filled with 'Rossi'), 'Email' (filled with 'renzo.rossi@gmail.com'), 'ID Telegram' (filled with '@renzo\_rossi\_85'), and 'Descrizione' (filled with 'Ciao, sono Renzo :)'). At the bottom of the form are two buttons: 'Annulla' and 'Salva'.

Nella sezione *Profilo personale* l'utente ha in aggiunta la possibilità di cliccare su "*Modifica password*" e cambiare la password di accesso al sito inserendo la password vecchia - ovvero della password che si vuole modificare -, la nuova password - che deve rispettare i requisiti di *Strong password* definiti nel documento **D1: Analisi dei requisiti** - e la conferma della nuova password. Nel caso in cui queste due password coincidessero e i requisiti di *Strong password* fossero rispettati, cliccando il tasto *Salva*, il sistema effettua la modifica definitiva della password di accesso che può successivamente essere cambiata in ogni momento. In tutti gli altri casi o nell'eventualità in cui si cliccasse il tasto *Annulla* la modifica non verrà effettuata. Per fornire questo tipo di servizio viene utilizzata l'*API password*.

**Dati di accesso - Modifica password**

Vecchia password:

Nuova password:

Conferma nuova password:

Inoltre, tutte le schermate del profilo forniscono una barra laterale per navigare all'interno del proprio profilo: **Annunci di lavoro attivi** permette la visualizzazione della **Home page** mentre **Crea un nuovo annuncio** permette la creazione di un nuovo annuncio.

**Crea un nuovo annuncio**

**Tipo di lavoro**  
Inserisci il tipo di lavoro

**Luogo**  
Inserisci il luogo

**Data**  
Inserisci giorno    Inserisci mese    Inserisci anno

**Ora**  
Inserisci ora inizio    Inserisci minuti inizio  
Inserisci ora fine    Inserisci minuti fine

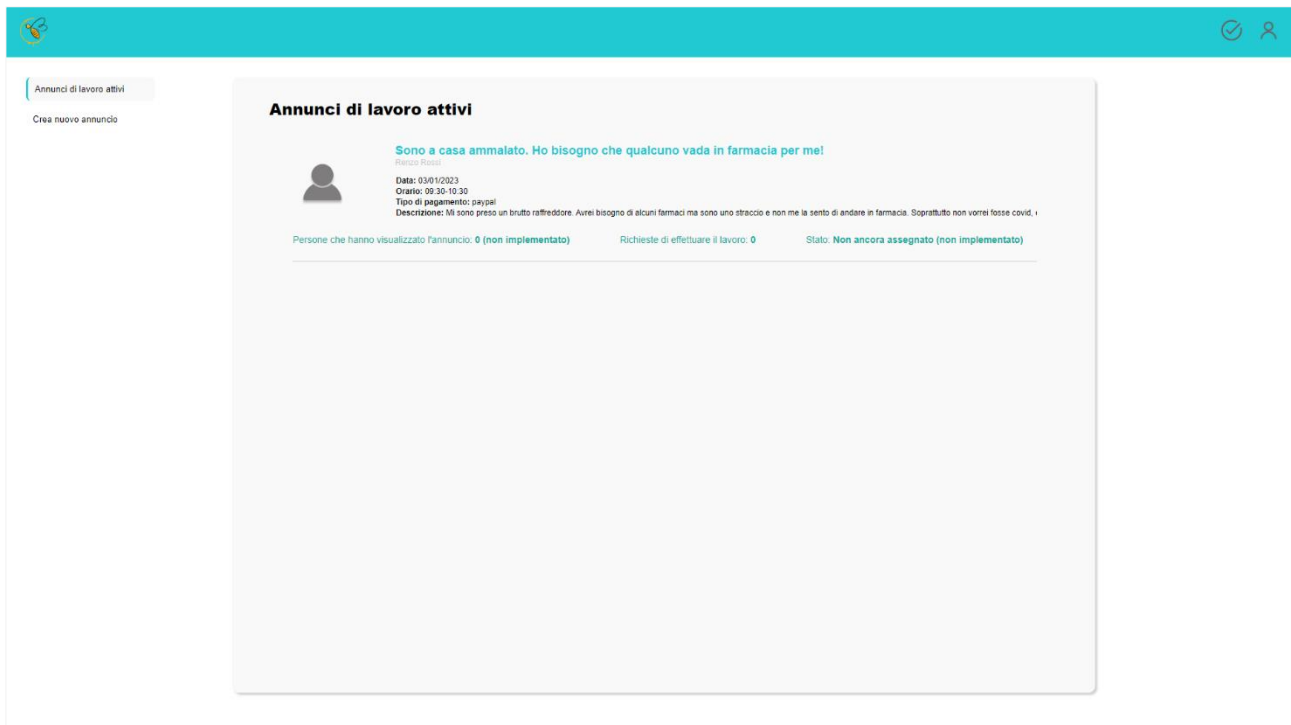
**Tipo di pagamento**  
Selezione il tipo di pagamento

**Descrizione del lavoro**  
Inserisci descrizione del lavoro

Tramite un'apposita etichetta sul menù di sinistra è infatti possibile accedere nella pagina di **Creazione nuovo annuncio**. Questa permette all'offerente di creare un nuovo annuncio inserendo dei dati specifici, quali il *tipo di lavoro*, ovvero un insieme di parole chiave per descrivere il servizio richiesto, il *luogo* dove

verrà svolto il lavoro, la *data* e l'*ora* in cui deve essere svolto il lavoro, la *tipologia di pagamento* prevista e una *breve descrizione* del lavoro che lo studente dovrà effettuare.

L'annuncio verrà creato ed inserito a sistema premendo l'apposito pulsante *Crea annuncio*. Cliccando questo tasto verrà dunque modificata la pagina **Home Page** alla quale verrà aggiunto l'annuncio appena creato. Per fornire questo tipo di servizio viene utilizzata l'*API AnnouncementCreation*.



## 6. GitHub Repository and Deployment Info

Il progetto StayBusy è disponibile al seguente link:

<https://github.com/riccardoFus/Stay-Busy>

Il deployment del progetto è stato effettuato con la piattaforma *Heroku*. Di seguito il link per visualizzare la web app:

<https://staybusyy.herokuapp.com/>

## 7. Testing

Per eseguire il testing è stata definita nel nostro Progetto la cartella “tests” al cui interno sono presenti due file *.test.js*. In particolare, questi file sono “offerer.test.js” e “announcement.test.js” ed entrambi vanno a definire dei test cases per le API utilizzate nella nostra web app.

Inoltre, sono stati creati diversi casi di test nei quali si è cercato di verificare tutti i comportamenti delle API in base all’input ricevuto dall’utente. Nello specifico, si è provato a verificare se il funzionamento dell’API venisse svolto correttamente.

Per eseguire il test abbiamo installato il modulo *supertest* con il quale siamo andati a testare le API. Per avviare i test basta entrare nella cartella contenente il file *server.js* ed eseguire il comando “npm test”. Si noti che dal momento che il server è in ascolto, il testing non termina; per farlo terminare bisogna bloccare il server dal mettersi in ascolto. Questo è il risultato del testing:

```
PASS tests/offerer.test.js (10.059 s)
  • Console

  console.log
    Your app is listening on port: 59537

    at Server.log (server.js:85:13)

  console.log
    MongoDB Connection -- Ready state is: 1

    at log (server.js:98:17)

PASS tests/announcement.test.js
  • Console

  console.log
    Your app is listening on port: 59656

    at Server.log (server.js:85:13)

  console.log
    MongoDB Connection -- Ready state is: 1

    at log (server.js:98:17)
      at runMicrotasks (<anonymous>)
```

## 8. Note ed eventuali

Il diagramma *User Flow* presente in questo sito è stato realizzato interamente dal nostro gruppo tramite il sito [www.flowmapp.com](http://www.flowmapp.com). Per una visualizzazione maggiormente definita di tutti i diagrammi presenti nel documento cliccare [qui](#).