



SAPIENZA
UNIVERSITÀ DI ROMA

PlacesJS: design e sviluppo di una Web-Application con framework Node.js

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di laurea in Ingegneria Informatica e Automatica

Daniele Iacomini
Matricola 1706790

Relatore
Roberto Beraldi

A.A. 2018-2019

*“Measuring programming progress by lines of
code is like measuring aircraft building
progress by weight”*

Bill Gates

Indice

1. Introduzione	7
1.1. Ideazione e specifica del progetto.....	7
1.2. Tecnologie utilizzate	8
2. Recap delle Activities (UNREGISTERED)	17
2.1. Static Pages.....	17
2.2. Signup	20
2.3. Login locale	22
2.4. Login con Facebook.....	24
3. Recap delle Activities (REGISTERED)	27
3.1. Profile Page	27
3.2. PlacesJS.....	30
3.3 Chat real time.....	36

4. Back-End	38
4.1 Server	38
4.2 Database	40
4.3 RabbitMQ.....	42
4.4 Web Socket.....	44
 5. Dispiegamento	 45
5.1 Installazione.....	45

Indice delle figure

Figura 1 Install Express.....	9
Figura 2 OAuth Schema.....	10
Figura 3 Install Passport	11
Figura 4 REST	12
Figura 5 API communication	13
Figura 6 Web Socket request	14
Figura 7 Web Socket response	15
Figura 8 RabbitMQ working	15
Figura 9 Home Page	17
Figura 10 About App	18
Figura 11 Contacts.....	19
Figura 12 Signup.....	20

Figura 13 Alert JavaScript.....	21
Figura 14 Login.....	22
Figura 15 Passport configuration.....	23
Figura 16 Local Login	23
Figura 17 Facebook callback.....	24
Figura 18 Facebook configuration	25
Figura 19 Auth configuration.....	26
Figura 20 Profile	27
Figura 21 Profile with Local Login	28
Figura 22 User Implementation	29
Figura 23 PlacesJS.....	30
Figura 24 initMap()	31
Figura 25 geocodeAddress().....	32
Figura 26 getNearbyPlaces()	33
Figura 27 createMarkers().....	34
Figura 28 deleteMarkers().....	34
Figura 29 handleLocationError()	35
Figura 30 Script start	35
Figura 31 Chat.....	36
Figura 32 Web Socket implementation	37
Figura 33 Node.js modules.....	38
Figura 34 Server.....	39
Figura 35 Cluster MongoDB.....	40
Figura 36 DB configuration	41

Figura 37 Route example with RabbitMQ	42
Figura 38 Reciever RabbitMQ	43
Figura 39 Web Socket on Server	44
Figura 40 GitHub repository	45
Figura 41 Command Line Interface.....	45

Capitolo 1 – Introduzione

1.1 Ideazione e specifica del progetto

La suddetta relazione viene redatta in merito al progetto PlacesJS, nato dalla volontà di sviluppare una Web-Application con il framework Node.js. L'idea alla base di questo progetto è quella di realizzare un sito Web sia dal punto di vista del front-end che dal punto di vista del back-end, che dia la possibilità agli utenti che lo visitano per la prima volta di registrarsi ed autenticarsi con le proprie credenziali, e a questi ultimi offre la funzionalità di geolocalizzarsi, per individuare la propria posizione corrente, oppure di cercare una via o una piazza a piacere nel mondo. Fatto ciò essi potranno cercare dei punti d'interesse in un raggio prestabilito. Per punti d'interesse si intende qualsiasi attività commerciale e non, che sia registrata ed individuabile. Ad esempio l'applicazione viene incontro a tutti coloro che, per esigenze diverse, organizzano pranzi o cene all'ultimo. Con PlacesJS potrai cercare tutti i bar o ristoranti nelle vicinanze oppure in un luogo prestabilito. Ma allo stesso tempo se si ha la necessità urgente di contanti oppure di un ospedale e siamo in una città che non conosciamo, PlacesJS troverà per noi la soluzione migliore nelle vicinanze.

L'applicazione offre, inoltre, la possibilità agli utenti registrati di instaurare una vera e propria chat in tempo reale, scambiandosi pareri ed opinioni sui luoghi appena cercati con altri utenti registrati.

Il sito dunque, si presenta completo nella sua semplicità e soddisfacente nel raggiungimento degli obbiettivi prefissati per una corretta esperienza d'uso da parte degli utenti che scelgono di sfruttare le potenzialità offerte da PlacesJS.

1.2 Tecnologie utilizzate

L'intera struttura dell'applicazione è stata sviluppata mediante il framework Node.js. Da un punto di vista tecnico, Node.js è un framework realizzato da V8, il motore JavaScript di Google Chrome, che permette di realizzare applicazioni web veloci e scalabili. Node.js usa un modello ad eventi e un sistema di I/O non bloccante che lo rende perfetto per applicazioni real-time che elaborano dati in modo intensivo e che può essere distribuito su più sistemi.

Leggero ed efficiente allo stesso tempo, la sua prima versione è stata rilasciata a metà del 2009, ed è tutt'ora in continua evoluzione avendo una community di sviluppatori alle spalle molto attiva(ultimo aggiornamento Agosto 2019).

L'utilizzo di JavaScript che l'ha reso celebre tra i programmatori di tutto il mondo è principalmente quello della programmazione lato client. Gli script, generalmente incorporati all'interno del codice HTML, vengono interpretati da un motore di esecuzione incorporato direttamente all'interno del proprio Browser.

Node.js amplia queste possibilità consentendo di utilizzare JavaScript anche per la scrittura di codice da eseguire lato server, ad esempio per la produzione del contenuto di pagine web dinamiche prima che la stessa venga inviata al Browser dell'utente. Grazie all'utilizzo di Node.js, è stato unificato lo sviluppo di applicazioni Web intorno ad un unico linguaggio di programmazione, Javascript per l'appunto.

Durante l'installazione di Node.js, viene installato anche un programma associato ad esso chiamato NPM, ovvero Node Package Manager. Nient'altro che una delle più grandi librerie software nella rete, in grado di scaricare ed installare in completa autonomia una serie di pacchetti che facilitano il lavoro e sviluppo del progetto in questione.

All'interno di Node.js sono state utilizzate e sviluppate altre tecnologie quali:

- *Express.js* : si tratta di un micro-framework per Node.js che offre strumenti base, come per esempio la possibilità di generare percorsi(URL) o di utilizzare i template per creare più velocemente applicazioni in Node;

Per installare Express.js è richiesto come comando da terminale:

```
1 | npm install express --save
2 | npm install express-generator -g
```

Figura 1 Install express

che oltre ad effettuare l'installazione a livello globale del pacchetto, effettua anche l'installazione del suo generatore di scaffolding.

- *OAuth* : Open Authorization è un protocollo che permette ad applicazioni di chiamare, in modo sicuro ed autorizzato, API messe a disposizione da un servizio Web. Permette di accedere alle risorse protette di un utente senza che esso debba condividere le sue credenziali(username e password).

L'autorizzazione dei trasferimenti di dati effettuati per mezzo delle API è necessaria in quanto altrimenti sussiste il rischio che terzi non autorizzati possano intercettare i dati ed utilizzarli illegittimamente per scopi personali.

Questo significa, ad esempio, che se un'applicazione deve pubblicare per conto di un utente un post su Facebook (e quindi accedere all'API di Facebook), l'utente deve fornirle l'autorizzazione necessaria.

Tramite il protocollo OAuth, l'utente può concedere tale autorizzazione senza dover fornire all'applicazione autorizzata username e password, mantenendo quindi il **pieno controllo sui propri dati**.

Nel protocollo OAuth si distinguono quattro ruoli:

- **Resource Owner** : entità che concede al client l'accesso a dati personali protetti.
- **Resource Server** : server in cui vengono salvati i dati protetti di un Resource Owner.
- **Client Application** : applicazione desktop, web o mobile che chiede l'accesso ai dati protetti del Resource Owner.
- **Authorization Server** : server che autentica il Resource Owner e rilascia un token di accesso temporaneo ad un ambito di applicazione (scope) definito dallo stesso utente. Nella realtà il server di autorizzazione e il Resource Server vengono spesso gestiti insieme.

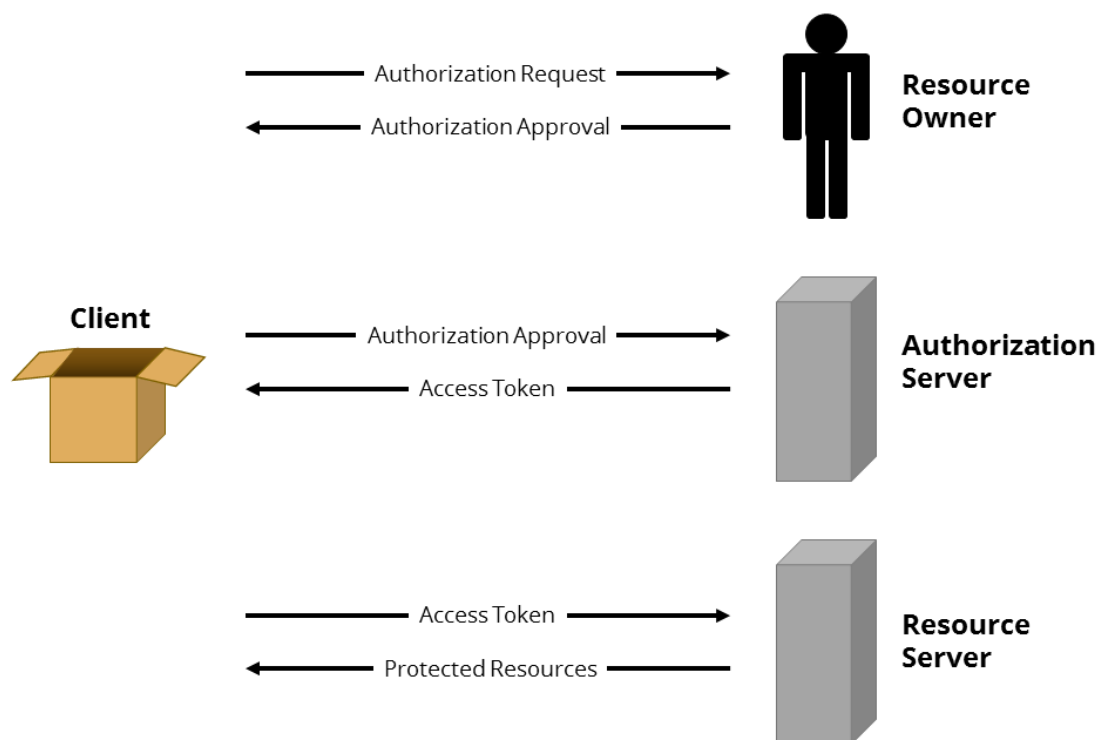


Figura 2 OAuth schema

Il protocollo prevede le seguenti fasi:

1. Il client richiede direttamente ,o tramite server di autorizzazione, l'autorizzazione al Resource Owner.
 2. Il Resource Owner concede un permesso di autorizzazione mediante un processo di autorizzazione.
 3. Il client con permesso di autorizzazione richiede un token di accesso al server di autorizzazione.
 4. Il server di autorizzazione autentica il client sulla base del suo permesso di autorizzazione e rilascia un token di accesso.
 5. Il client utilizza il token di accesso per richiedere al Resource Server i dati protetti del Resource Owner.
 6. Il Resource Server autentica il client sulla base del suo token di accesso e fornisce i dati desiderati.
- *Passport* : middleware di autenticazione per Node.js . Estremamente flessibile e modulare, Passport può essere inserito in modo discreto in qualsiasi applicazione Web basata su Express. Supporta autenticazione mediante username e password, Facebook, Twitter e molto altro.

Nelle moderne applicazioni Web, l'autenticazione può assumere una varietà di forme. Tradizionalmente, gli utenti accedono fornendo una mail ed una password. Con l'ascesa dei social network, il sign-in che utilizza un provider OAuth come Facebook o Twitter è diventato un metodo di autenticazione popolare. I servizi che espongono API richiedono spesso credenziali basate su token per proteggere l'accesso.

```
$ npm install passport
```

Figura 3 Install Passport

Servizi REST API : **RE**presentational **State** Transfer è uno stile di architettura per la progettazione di applicazioni in rete. Viene utilizzato il protocollo HTTP per gestire richieste ed effettuare chiamate tra due punti. In effetti lo stesso World Wide Web, che si basa su HTTP, può essere visto come un'immensa architettura basata su REST. Le applicazioni basate su di esso, si definiscono **RESTful** e utilizzano le richieste HTTP per inviare i dati (creazione e/o aggiornamento), effettuare query, modificare e cancellare i dati. In definitiva, REST utilizza HTTP per tutte e quattro le operazioni CRUD (Create / Read / Update / Delete).

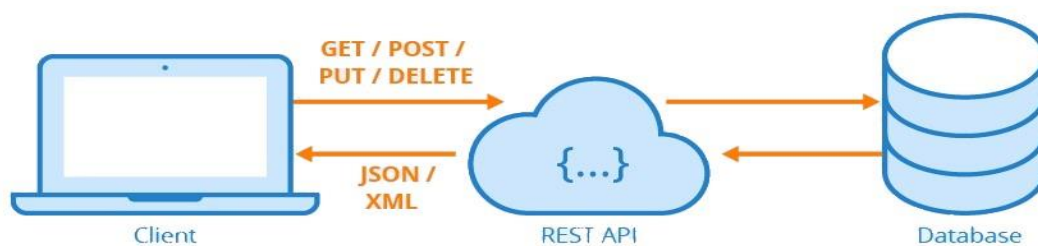


Figura 4 REST

Tale approccio architetturale è definito dai seguenti sei principi guida:

1. **Client-server** – Separando il contesto client da quello server, miglioriamo la portabilità dell'interfaccia utente su più piattaforme e la scalabilità.
2. **Stateless** - Lo stato della sessione viene mantenuto interamente sul client, senza sfruttare contesti memorizzati sul server.
3. **Cacheable** - I client possono fare caching delle risposte. Le risposte devono in ogni modo definirsi cacheable o no. Gestione corretta della cache migliora scalabilità e performance.

4. **Interfaccia uniforme** – Interfaccia di comunicazione omogenea tra client e server permette di semplificare l'architettura.
5. **Sistema a livelli**: Architettura composta da livelli gerarchici in modo tale che ciascun componente non possa "vedere" oltre il livello immediato con cui interagiscono.
6. **Codice su richiesta (opzionale)** - REST consente di estendere le funzionalità del client scaricando ed eseguendo il codice sotto forma di applet o script.

Le API, acronimo di Application Programming Interface, sono degli strumenti di programmazione che le maggiori software house e industrie del mondo informatico – come Microsoft, Google e Facebook – mettono a disposizione degli sviluppatori per facilitare il loro compito nella realizzazione di applicazioni di vario genere. Consentono ai tuoi prodotti o servizi di comunicare con altri prodotti o servizi senza sapere come vengono implementati, semplificando così lo sviluppo delle app e consentendo un netto risparmio di tempo.

Possono essere delle librerie di funzioni che permettono al programmatore di far comunicare due realtà tra loro incompatibili. Facendo ciò, si possono estendere le funzionalità di un programma ben oltre le reali intenzioni di chi lo progetta.

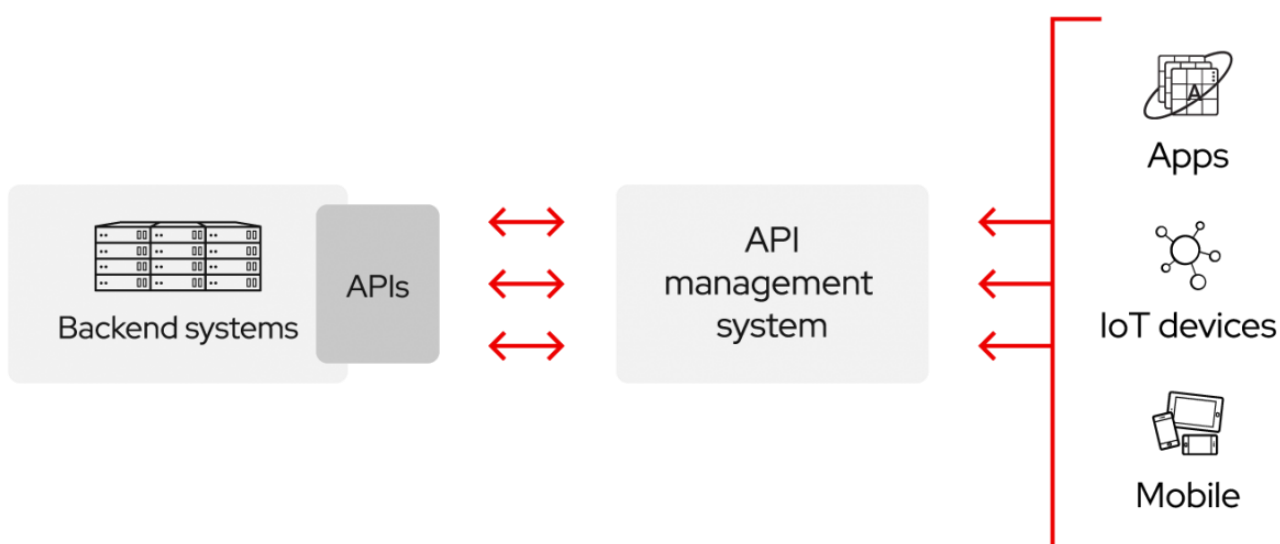


Figura 5 API communication

Le API utilizzate all'interno del progetto in questione sono offerte da Google Maps.

La casa di Mountain View mette queste API a disposizione di tutti gli sviluppatori che le volessero utilizzare per un loro programma o piattaforma Web. Sfruttando le API, ad esempio, è possibile utilizzare il servizio di cartografia digitale di Google per realizzare delle mappe personalizzate; oppure integrarle in siti web per servizi di ricerca georeferenziati; o ancora utilizzarle all'interno di applicazioni per smartphone e tablet.

- *Web Socket* : Si tratta di un protocollo di messaggistica che permette una comunicazione asincrona e full-duplex su connessione TCP. I Web Socket non sono connessioni HTTP anche se usano l'http per avviare la connessione. Si definisce sistema full-duplex quel particolare sistema che permette la comunicazione in entrambe le direzioni in maniera contemporanea.

Ogni connessione Web Socket inizia la sua vita come una richiesta HTTP, simile alle più celebri richieste salvo che nell'intestazione viene specificata l'operazione di tipo *Upgrade* che indica che il client vuole aggiornare la connessione ad un protocollo diverso, Web Socket per l'appunto. Ad aggiornamento avvenuto, viene stabilita la connessione Web Socket tra client e server e la comunicazione in entrambe le direzioni può cominciare.

Viene riportato un esempio di richiesta lato Client:

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

Figura 6 Richiesta Web Socket

Qui invece viene riportato una tipica risposta da parte del server ad una richiesta come quella precedente:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

Figura 7 Risposta Web Socket

- *RabbitMQ* : Server open-source che si basa sul principio del protocollo AMQP(Advanced Message Queuing Protocols). Per una corretta comunicazione tra mittente e destinatario è stato introdotto un servizio chiamato broker di messaggistica che si occupa della distribuzione dei messaggi. Il principio di base di tale tecnologia è piuttosto semplice: tra il produttore ed il consumatore di un messaggio c'è una coda, dove vengono depositati i messaggi.

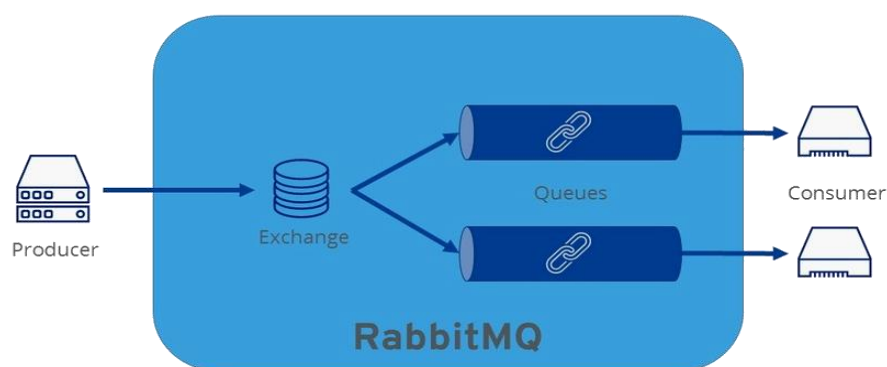


Figura 8 Funzionamento RabbitMQ

Il vantaggio di RabbitMQ è che il produttore del messaggio non deve occuparsi anche della sua trasmissione. Il broker di messaggistica consegna il messaggio permettendo al produttore di cominciare un nuovo incarico.

Il mittente, dunque, non deve attendere la ricezione del messaggio da parte del destinatario. In processi di questo tipo, il messaggio si trova in coda e può essere prelevato dal consumatore.

Quando ciò avverrà, il mittente sarà già impegnato con un nuovo incarico. Si tratta dunque di un **processo asincrono**: mittente e destinatario non devono agire nello stesso momento.

La comunicazione funziona tramite TCP, per il quale RabbitMQ necessita di porte. Quella scelta per questo tipo di utilizzo è la porta 5672 che deve essere in esecuzione su Localhost.

Nell'ambito del progetto, RabbitMQ è stato utilizzato in maniera leggermente diversa dal suo principale utilizzo:

- Ogni task all'interno del sito Web, come per esempio navigare nella pagina Contacts, è notificato sul terminale creando un vero e proprio registro dei log, immaginando l'utente che visita il sito come il produttore di messaggi.
- Il consumatore, o destinatario dei messaggi, aprendo il proprio terminale vedrà ricevere, in maniera asincrona, lo storico delle attività effettuate dall'utente all'interno del sito Web.

Capitolo 2 – Activities (UNREGISTERED)

2.1 Static Pages

- Home Page

Una volta avviato il server di Node.js, ci si trova di fronte alla Home Page del sito Web di PlacesJS. Dall'aspetto minimale ma comunque curato, viene descritto brevemente ciò per cui l'applicazione è stata concepita.

Quindi avremo riepilogo, per completezza, delle tecnologie sfruttate per arrivare all'obiettivo prefissato:

- Rest API di Google Maps, prese direttamente dalla documentazione presente nella Google Platform;
- OAuth, che come servizio commerciale è stato scelto Facebook, in modo tale che l'utente possa decidere di accedere al sito anche mediante le proprie credenziali di Facebook;

Di seguito viene riportato il Mock-Up della Home Page.

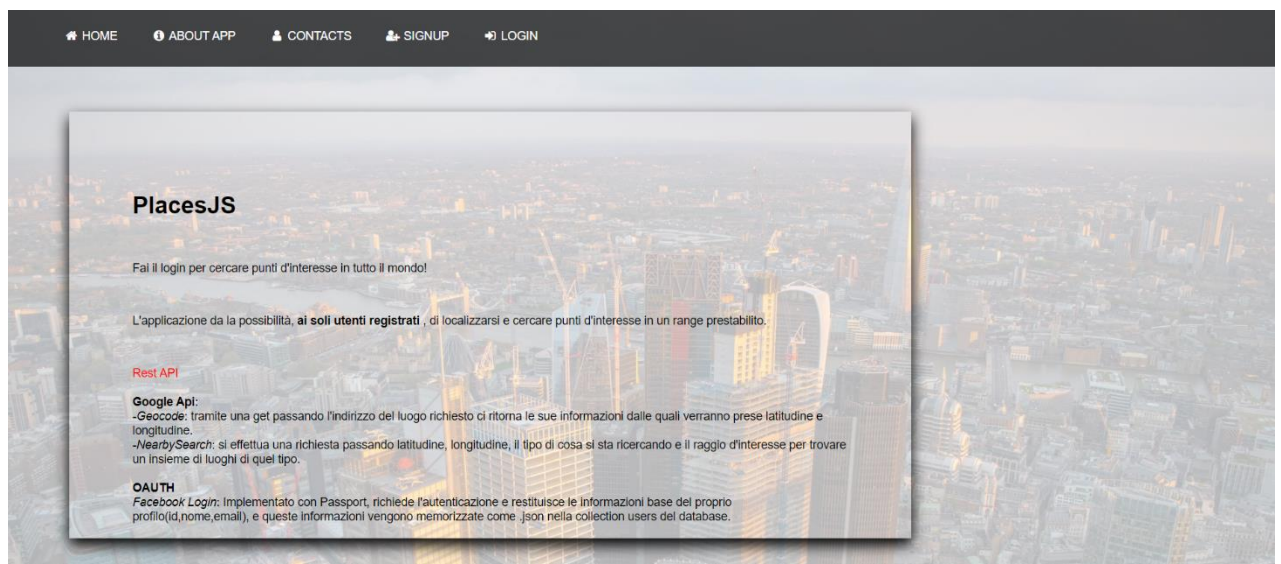


Figura 9 Home Page

- *About App*

Navigando all'interno del sito, possiamo notare come nella Navbar in alto troviamo la sezione About App.

Qui è stato chiarito lo scopo che ha spinto alla realizzazione di questo progetto, in merito all'attività di Tirocinio interno, che poi ha portato alla realizzazione della suddetta relazione.

Inoltre viene indicata anche una parte importante da cui si è preso informazioni utili durante la progettazione e scrittura del codice, nonché la documentazione ufficiale riguardo il framework Node.js .

Di seguito viene riportato il Mock-Up della pagina About App.

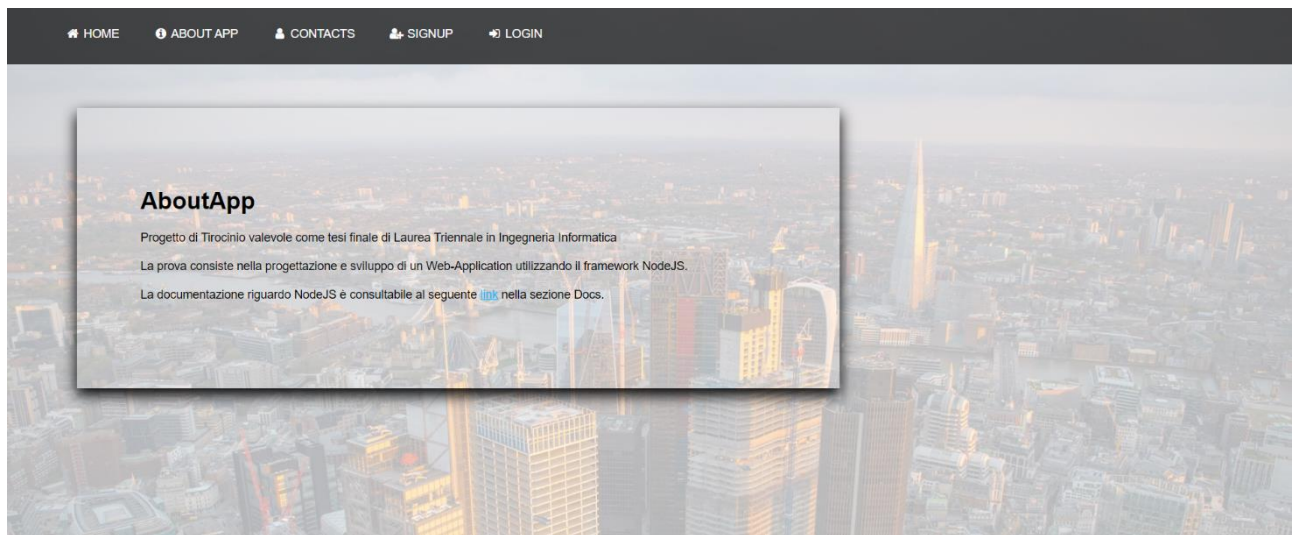


Figura 10 About App

- *Contacts*

Proseguendo nella navigazione della Navbar, troviamo la sezione Contacts, tipica di pressochè tutte le Web-Application.

Qui è esposto un rapido riepilogo dei modi per contattarmi, tra cui :

- Indirizzo mail istituzionale offerta dall'ateneo Sapienza;
- Email personale di lavoro;

Inoltre è presente il link al mio profilo sulla piattaforma GitHub, dove è stato caricato tutto il codice sorgente, compresa la directory di lavoro.

GitHub è una piattaforma di hosting per progetti software. Molto apprezzata dalla community di programmatori in tutto il mondo in quanto dà la possibilità di aprire progetti con programmatori che non risiedono necessariamente nella stessa città, e ognuno dei quali può apportare tutte le modifiche che si ritengono necessarie. Attraverso le operazioni di *"git pull"* e *"git push"* quindi, ogni sviluppatore può, rispettivamente, aggiornare la sua cartella di lavoro locale e caricare le sue modifiche sulla repository condivisa di Github.

Github tiene traccia di tutte le modifiche e di chi le opera, e attraverso dei messaggi nelle commit, lo sviluppatore può scrivere accuratamente cosa viene modificato con l'ultimo aggiornamento dei file.

Inoltre, questi lavori possono essere resi pubblici agli utenti registrati che possono fare da tester del progetto per eventuali correzioni di bug.

Di seguito viene riportato il Mock-Up della pagina Contacts.

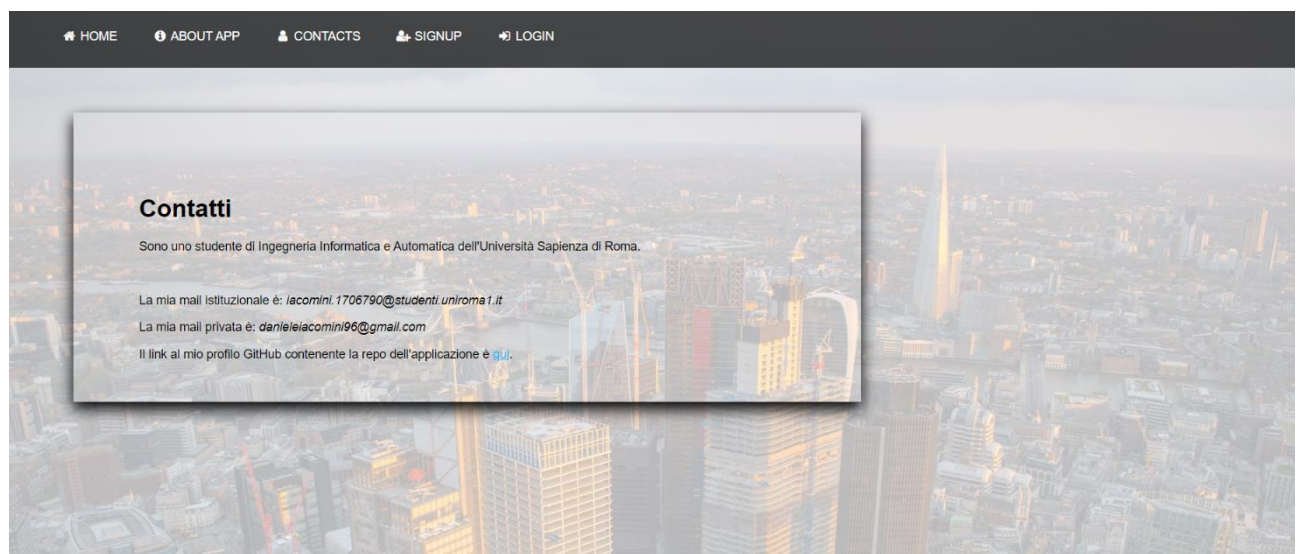


Figura 11 Contacts

2.2 Signup

La pagina di registrazione corrisponde alla prima vera attività “*dinamica*” all’interno del sito Web.

Tale contesto sancisce il confine con chi è interessato a fruire dei contenuti dell’applicazione progettata, e chi invece è solamente di passaggio all’interno del sito. La necessità è stata quella di dover implementare la registrazione e l’autenticazione degli utenti: ogni utente infatti deve poter autenticarsi utilizzando alcune credenziali per poter usufruire dei servizi a lui dedicati.

Di seguito viene riportato il Mock-Up della pagina Signup.

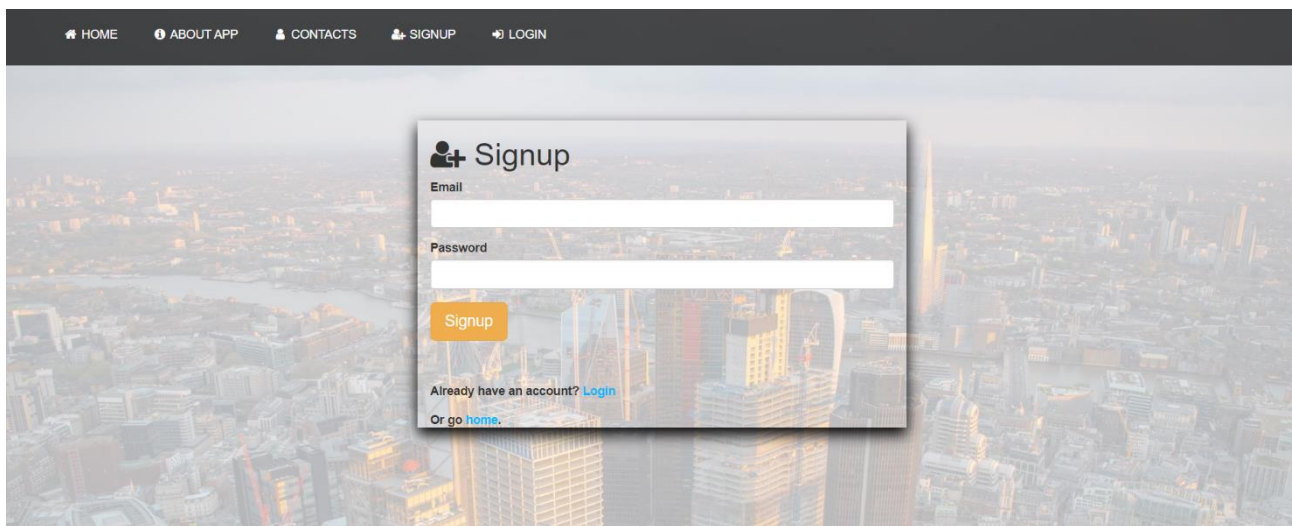


Figura 12 Signup

Il form di registrazione prevede che l’utente non registrato, per accedere ai contenuti dell’applicazione, inserisca anzitutto una mail ed una password.

I campi della form di registrazione sono soggetti al controllo di funzioni JavaScript per la sua corretta validazione, per cui se si prova ad inserire, ad esempio, un indirizzo mail senza l’uso del carattere “@”, il Browser informerà l’utente che la registrazione non è andata a buon fine in quanto è richiesto un indirizzo mail valida.

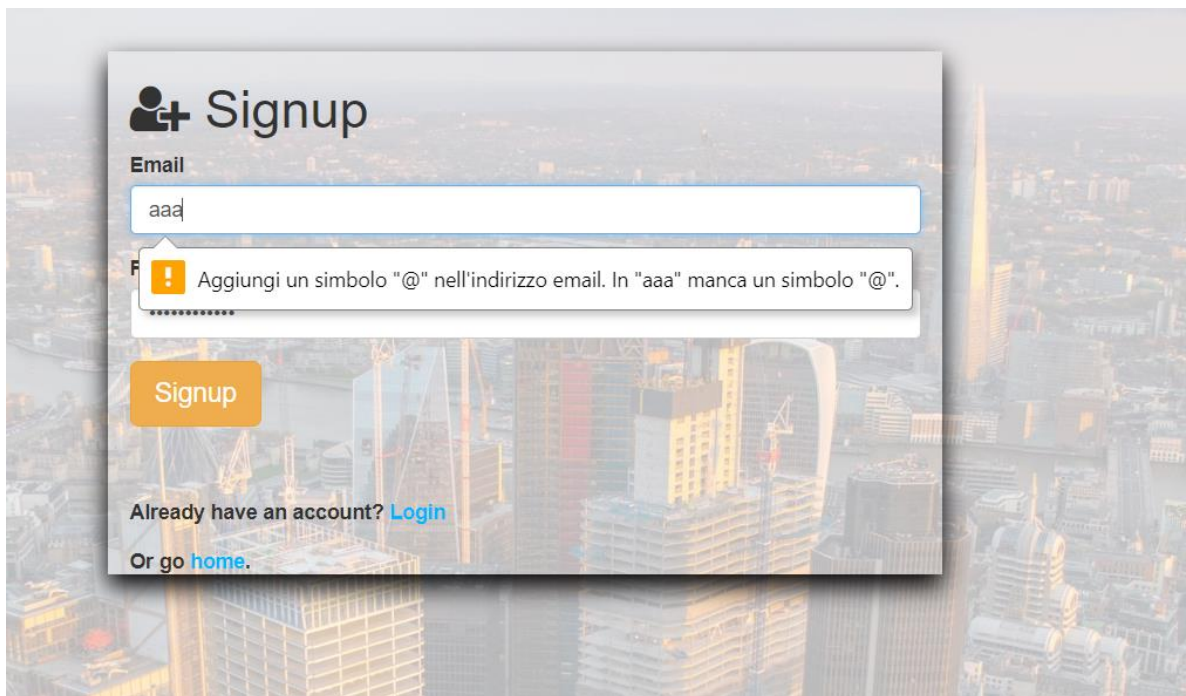


Figura 13 Alert JavaScript

Il tutto è notificato poi da una funzione di Alert, tipica di chi progetta Web-Application con Javascript ed HTML.

Se la registrazione è andata a buon fine, allora si verrà automaticamente re-indirizzati all'interno dell'applicazione, nella pagina di Profilo, con il riepilogo delle informazioni utilizzate.

Lato Back-end del sito, questo tipo di azione corrisponde a creare all'interno del Database utilizzato(MongoDB) una tupla, contenente due campi separati, email e password per l'appunto. Quest'azione è fondamentale, in quanto per i futuri accessi da parte dell'utente, senza tale memorizzazione, il sistema di Login non sarebbe in grado di riconoscerlo e il sito richiederebbe una nuova fase di Signup.

2.3 Login locale

La fase di Login è, temporalmente parlando, ciò che qualsiasi utente effettua subito dopo essersi registrato al sito Web.

Per la fruizione dei contenuti dell'applicazione è necessario inserire email e password precedentemente scelti in fase di Registrazione.

Di seguito viene riportato il Mock-up della pagina Login.

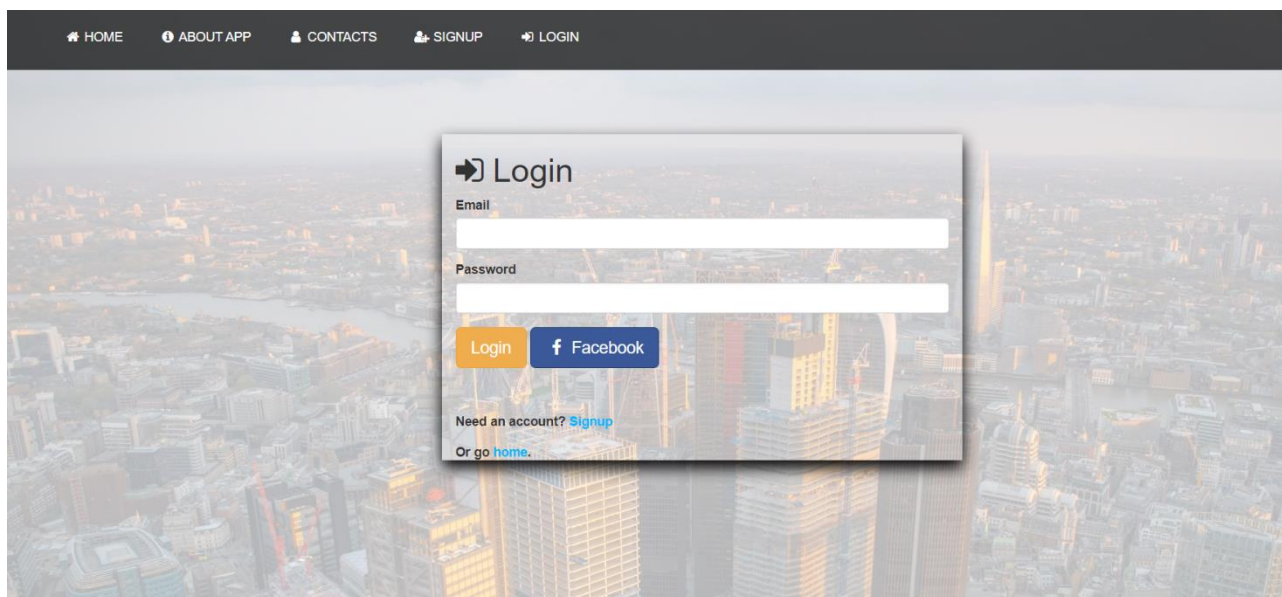


Figura 14 Login

Il sito offre la possibilità di effettuare il Login in due differenti modi:

- Login locale, tramite modulo Passport;
- Login con Facebook;

Per quanto riguarda il Login locale si fa riferimento ai dati inseriti in fase di Signup. Dal punto di vista della progettazione e scrittura del codice, l'utilizzo del modulo Passport è stato molto utile ed ha reso decisamente più snello il codice rispetto ad altre soluzioni che erano anch'esse praticabili.

```

const passport      = require('passport')
  , FacebookStrategy = require('passport-facebook').Strategy
  , session          = require('express-session')
  , cookieParser     = require('cookie-parser')
  , config            = require('./configuration/config')

require('./configuration/passport')(passport);
app.use(passport.initialize());
app.use(passport.session());
app.use(flash());

app.use(cookieParser());
app.use(session({ secret: 'key', saveUninitialized: true, resave: true }));
app.use(passport.initialize());
app.use(passport.session());

```

Figura 15 Configuration Passport

È necessario richiedere il modulo Passport per poterlo utilizzare, così come *'session'*, necessario a mantenere la sessione aperta finché l'utente non decide di effettuare il Logout dal sito.

Di seguito il codice, presente nel file passport.js, per la gestione del login locale.

```

passport.use('local-login', new LocalStrategy({
  usernameField: 'email',
  passwordField: 'password',
  passReqToCallback: true
},
function(req, email, password, done){
  process.nextTick(function(){
    User.findOne({ 'local.username': email }, function(err, user){
      if(err)
        return done(err);
      if(!user)
        return done(null, false, req.flash('loginMessage', 'No
User found'));
      if(!user.validPassword(password)){
        return done(null, false, req.flash('loginMessage',
'invalid password'));
      }
      return done(null, user);
    });
  });
});

```

Figura 16 Local Login

2.4 Login con Facebook

Il login con Facebook è di fatto uno dei tipi di accesso più utilizzati dalle Web-Application al giorno d'oggi. Consente all'utente di inserire solo parte dei dati richiesti da una form di Registrazione, in quanto email, nome e cognome, ed altre svariate informazioni vengono prese direttamente dal proprio profilo di Facebook. Questo è possibile grazie al servizio offerto da OAuth che tramite le API di Facebook rende l'esperienza utente ancor più semplice ed intuitiva.

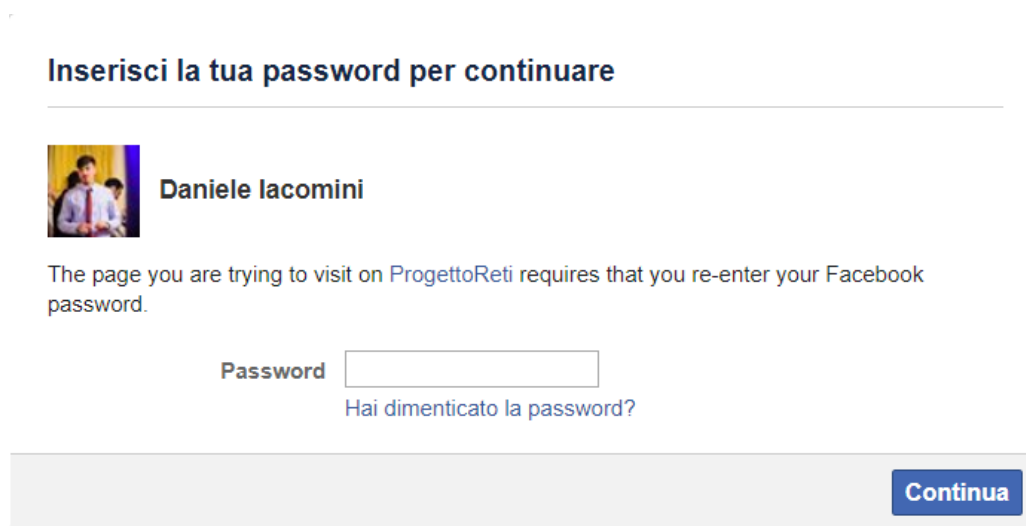


Figura 17 Facebook Callback

Se si sceglie di continuare, Facebook reindirizza l'utente all'URL impostato nell'applicazione di Facebook for Developers: "localhost:3000/auth/facebook/callback". Insieme al redirect nell'URL di callback, Facebook nella risposta fornisce i dati richiesti dall'applicazione PlacesJS che sono i semplici dati personali come nome, cognome ed e-mail. Tali dati vengono gestiti da una funzione opportunamente definita: questo controlla se è la prima volta che l'utente richiede di autenticarsi con Facebook o se già in precedenza aveva dato il suo consenso al trattamento dei dati.

La piattaforma Facebook for Developers, una volta registrata la propria applicazione, concede due dati strettamente personali dell'app, quali :

- Facebook_api_key;
- Facebook_api_secret;

Queste due variabili identificano la nostra applicazione alla piattaforma che darà quindi all'utente la possibilità di utilizzare come modalità di accesso il login con Facebook.

Di seguito è riportato il codice di implementazione del modulo per l'accesso con Facebook. Anche in questo caso ci si è servito di Passport.

```
passport.use('facebook', new FacebookStrategy({
  clientID: configAuth.facebookAuth.clientID,
  clientSecret: configAuth.facebookAuth.clientSecret,
  callbackURL: configAuth.facebookAuth.callbackURL,
  profileFields: ['id', 'displayName', 'photos', 'email'],
  enableProof: true, // For security
  //passReqToCallback: true // Pass request from route
}),
function(accessToken, refreshToken, profile, done) {
  process.nextTick(function(){
    User.findOne({'facebook.id': profile.id}, function(err,
user){
      if(err)
        return done(err);
      if(user)
        return done(null, user);
      else {
        var newUser = new User();
        newUser.username = profile.displayName;
        newUser.facebook.id = profile.id;
        newUser.facebook.token = accessToken;
        newUser.facebook.name = profile.displayName;
        newUser.facebook.email = profile.emails[0].value;

        newUser.save(function(err){
          if(err)
            throw err;
          return done(null, newUser);
        })
      }
    });
  });
}
```

Figura 18 Facebook Configuration

La funzione `passport.use` abilita il modulo `FacebookStrategy` di Passport, e le variabili `configAuth.facebookAuth.clientID`, `configAuth.facebookAuth.clientSecret` corrispondono esattamente ai dati passati dalla piattaforma per sviluppatori di Facebook. Di un nuovo utente che accede con Facebook vogliamo che venga salvato all'interno del Database:

- Username;
- Id;
- Token di accesso;
- Nome;
- Email;

Di seguito il contenuto del file `auth.js` necessario alla corretta configurazione del Login con Facebook.

```
module.exports = {
  'facebookAuth' : {
    clientID: process.env.FB_ID,
    clientSecret: process.env.FB_SECRET,
    callbackURL: 'http://localhost:3000/auth/facebook/callback',
    profileURL:
    'https://graph.facebook.com/v2.5/me?fields=first_name,last_name,email',
    //profileFields: ['id', 'displayName', 'email', 'first_name',
    'middle_name', 'last_name']
  }
}
```

Figura 19 Auth configuration

Capitolo 3 – Activities (REGISTERED)

3.1 Profile Page

Il Profilo rappresenta la prima pagina che un utente, nel momento in cui decide di registrarsi al sito, può visitare. Subito dopo la registrazione, o il login a seconda delle esigenze, si viene indirizzati a questa pagina, che contiene il riepilogo delle informazioni che l'utente ha scelto di inserire per poter accedere all'applicazione.

Ci sono due sezioni, che corrispondono alle due differenti modalità d'accesso offerte all'utente. Una per il Login Locale, e la seconda per chi decide di accedere mediante Facebook.

Di seguito viene riportato il Mock-up della pagina Profilo.

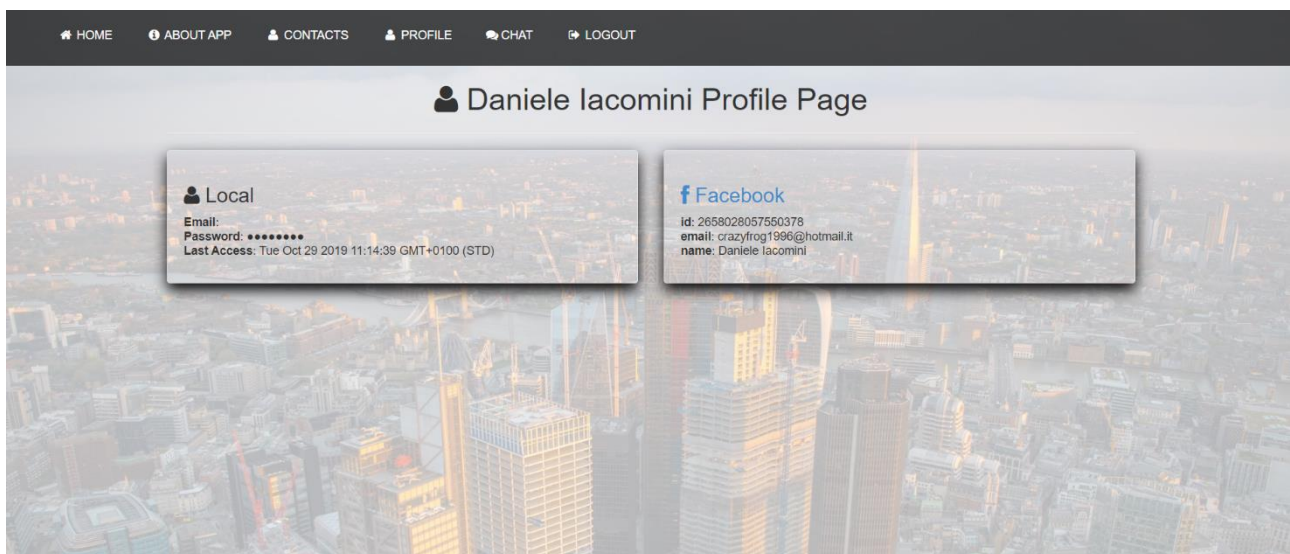


Figura 20 Profile

A tal proposito, è possibile notare come siano presenti solamente i dati che si è deciso di salvare dal proprio profilo di Facebook, quali:

- Id;
- Email di accesso;
- Nome e Cognome;

La sezione riguardo il Login Locale è composta invece da:

- Email di accesso;
- Password criptata;
- Ultimo accesso;

In fase di scrittura del codice è stata sviluppata una semplice funzione JavaScript che consente di visualizzare solamente le informazioni relative alla tipologia di accesso che si è scelto di utilizzare, in modo tale che, in caso di Login Locale, la tabella contenente le informazioni riguardanti Facebook, risulti vuota e priva di informazioni.



Figura 21 Profile with Local Login

Lato Back-End dell'applicazione l'utente è stato gestito come una tupla all'interno del Database, servendoci:

- Del modulo Mongoose, per l'inserimento e la gestione all'interno del Database dei singoli utenti;
- Del modulo Bcrypt, per la cifratura della password, in modo da tener sempre nascosti i dati sensibili degli utenti che decidono di registrarsi;

Il file user.js, contenuto nella directory *"models"* è così composto:

```
var mongoose = require('mongoose');
var bcrypt = require('bcrypt');

var userSchema = mongoose.Schema({
  local: {
    username: String,
    password: String,
    time : { type : Date, default: Date.now }
  },
  facebook: {
    id: String,
    token: String,
    name: String,
    email: String
  }
});

userSchema.methods.generateHash = function(password) {
  return bcrypt.hashSync(password, bcrypt.genSaltSync(9));
}

userSchema.methods.validatePassword = function(password) {
  return bcrypt.compareSync(password, this.local.password);
}

module.exports = mongoose.model('User', userSchema);
```

Figura 22 User implementation

3.2 PlacesJS

Una volta visionate le poche ma essenziali informazioni di riepilogo nella pagina di Profilo, si passa ufficialmente alla parte principale e più funzionale di tutta l'applicazione, l'idea cardine su cui si basa poi tutto il progetto.

Di seguito viene riportato il Mock-up della pagina PlacesJS.

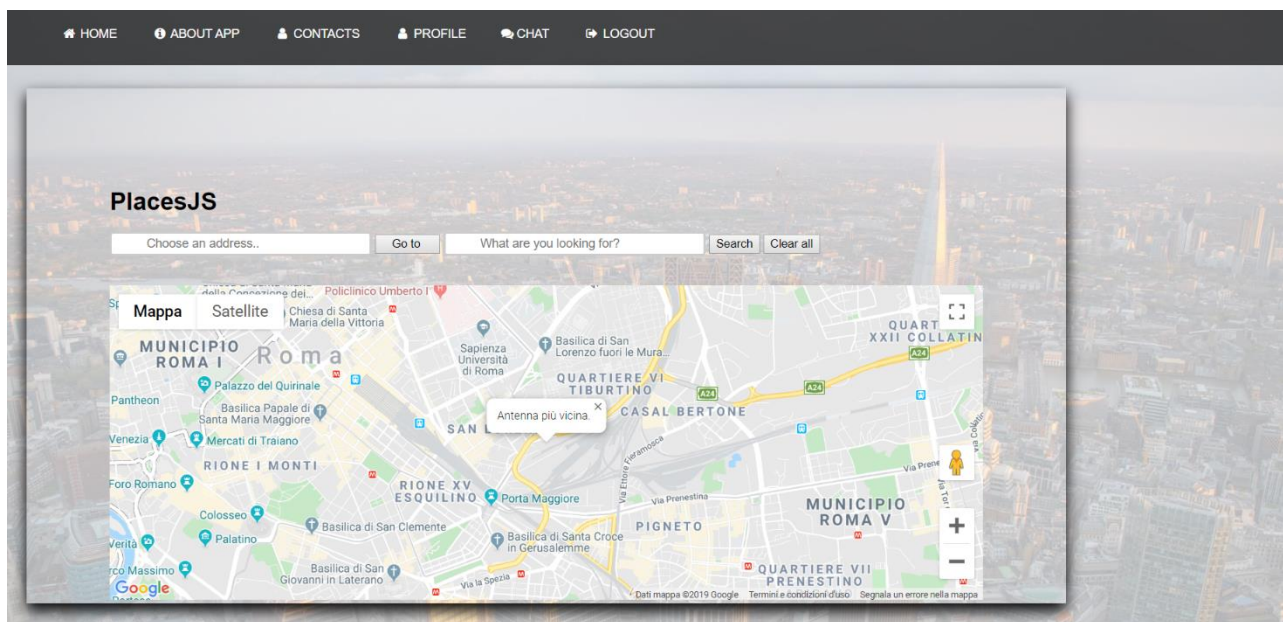


Figura 23 PlacesJS

La pagina è costituita da un contenitore principale chiamato “map”, che conterrà tutte le componenti dinamiche che sono state progettate.

Al centro della pagina troviamo una mappa, ottenuta tramite funzioni appositamente scritte per interagire con le API di Google Maps.

```

function initMap() {
    // Initialize variables
    bounds = new google.maps.LatLngBounds();
    infoWindow = new google.maps.InfoWindow;
    currentInfoWindow = infoWindow;
    var geocoder = new google.maps.Geocoder();

    // Try HTML5 geolocation
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(position => {
            pos = {
                lat: position.coords.latitude,
                lng: position.coords.longitude
            };
            map = new google.maps.Map(document.getElementById('map'), {
                center: pos,
                zoom: 14
            });
            bounds.extend(pos);

            infoWindow.setPosition(pos);
            infoWindow.setContent('Antenna più vicina.');
```

function() {

```

            infoWindow.open(map);
            map.setCenter(pos);
            //BUTTON LISTENERS
            document.getElementById('submit').addEventListener('click',
                geocodeAddress(geocoder, map);
            });
            document.getElementById('sushi').addEventListener('click',
function() {
            // Call Places Nearby Search on user's location
            deleteMarkers();
            getNearbyPlaces(pos);
            });
            document.getElementById('delete').addEventListener('click',
function() {
            //Delete markers in the map
            deleteMarkers();
            });
        }, () => {
            // Browser supports geolocation, but user has denied permission
            handleLocationError(true, infoWindow);
        });
    } else {
        // Browser doesn't support geolocation
        handleLocationError(false, infoWindow);
    }
}

```

Figura 24 initMap()

La funzione `initMap()` ha lo scopo di impostare preliminarmente la mappa dal satellite di Google.

Vengono definite delle variabili utilizzate all'interno dello script, tra cui `geocoder`.

Geocoder è un tool per la gestione della geocodifica e geocodifica inversa.

Per geocodifica si intende quel processo che trasforma un indirizzo, o altra descrizione di una posizione, in una coordinata formata da latitudine e longitudine.

Per geocodifica inversa si intende quel processo che trasforma una coordinata, formata sempre da latitudine e longitudine, in un indirizzo.

Navigator, invece, è un servizio per cui tramite il Browser, vengono richieste informazioni riguardo latitudine e longitudine. Tramite questo servizio, è possibile sfruttare la funzione `getCurrentPosition()` che restituisce l'attuale posizione dell'utente che l'ha richiesto.

Fatto questo, tramite la funzione `google.maps.Map` viene generata la mappa con dei valori di default, quali il posizionamento e lo zoom.

Successivamente vengono implementati i Listener dei button presenti nell'area sopra la mappa. Ognuno dei bottoni è stato progettato per fare una specifica azione, che ora vedremo nel dettaglio.

```
function geocodeAddress(geocoder, resultsMap) {
    var address = document.getElementById('address').value;
    geocoder.geocode({'address': address}, function(results,
status) {
        if (status === 'OK') {
            resultsMap.setCenter(results[0].geometry.location);
            var marker = new google.maps.Marker({
                map: resultsMap,
                position: results[0].geometry.location
            });
            markers.push(marker);
            pos = results[0].geometry.location;
        } else {
            alert('Geocode was not successful for the following
reason: ' + status);
        }
    });
}
```

Figura 25 `geocodeAddress()`

La funzione è costituita, anzitutto, da una variabile *address* che è costituita dal contenuto della casella di testo, in modo da essere sempre diversa in base alle ricerche dei differenti utenti.

Qui interviene il geocoder che, preso in input l'indirizzo, lo trasforma in una coordinata, formata sempre da latitudine e longitudine. Fatto ciò, centra la mappa in base alla coordinata che ottiene precedentemente e crea il primo Marker, tramite la funzione *google.maps.Marker*.

Il Marker non è nient'altro che la simbologia che utilizza Maps per individuare un luogo, e tramite la funzione *push* lo inserisce all'interno della mappa, nella posizione individuata dalle coordinate.

```
// Perform a Places Nearby Search Request
function getNearbyPlaces(position) {
    let request = {
        location: position,
        radius: 600,
        keyword: document.getElementById('what').value
    };

    service = new google.maps.places.PlacesService(map);
    service.nearbySearch(request, nearbyCallback);
}

// Handle the results (up to 20) of the Nearby Search
function nearbyCallback(results, status) {
    if (status == google.maps.places.PlacesServiceStatus.OK) {
        createMarkers(results);
    }
}
```

Figura 26 *getNearbyPlaces()*

La funzione prende in input la posizione ottenuta precedentemente, e crea un array contenente:

- Posizione;
- Raggio d'interesse;
- Keyword;

Sull'oggetto *service* viene invocata la funzione *google.maps.places.PlacesService* che ha il compito di trovare i luoghi d'interesse sulla mappa.

La funzione *nearbySearch()* richiama al suo interno la callback function *nearbyCallback()* che si occupa di creare e posizionare, tramite apposita funzione, il Marker per tutti gli elementi che sono stati trovati.

```
// Set markers at the location of each place result
function createMarkers(places) {
  places.forEach(place => {
    var marker = new google.maps.Marker({
      position: place.geometry.location,
      map: map,
      title: place.name,
    });
    // Adjust the map bounds to include the location of this
marker
    bounds.extend(place.geometry.location);
    markers.push(marker);
  });
}
```

Figura 27 createMarkers()

La funzione, partendo da ogni luogo trovato, si serve dell'API di Google per la creazione dei marker tramite la funzione *google.maps.Marker*, a cui viene passato la posizione, la mappa, e il nome del risultato(es. Gelateria Fassi). Come visto in precedenza, tramite la funzione *push*, viene inserito correttamente il Marker all'interno della mappa.

```
function deleteMarkers() {
  for (var i = 0; i < markers.length; i++) {
    markers[i].setMap(null);
  }
  markers = [];
}
```

Figura 28 deleteMarkers()

Questa semplice funzione, abbinata al button “Clear All” ha il compito tra una ricerca e l'altra di cancellare e ripulire la mappa da tutti i Marker precedentemente posizionati, in modo da poter iniziare una nuova ricerca.

```
// Handle a geolocation error
function handleLocationError(browserHasGeolocation, infoWindow)
{
    // Set default location to Sydney, Australia
    pos = { lat: -33.856, lng: 151.215 };
    map = new google.maps.Map(document.getElementById('map'), {
        center: pos,
        zoom: 15
    });

    // Display an InfoWindow at the map center
    infoWindow.setPosition(pos);
    infoWindow.setContent(browserHasGeolocation ?
        'Geolocation permissions denied. Using default location.' :
        'Error: Your browser doesn\'t support geolocation. ');
    infoWindow.open(map);
    currentInfoWindow = infoWindow;

    // Call Places Nearby Search on the default location
    getNearbyPlaces(pos);
}
}
```

Figura 29 handleLocationError()

Questa funzione ha il compito di gestire eventuali errori in fase di ricerca di un luogo. In tutti quei casi in cui lo script non riesce a riconoscere l'indirizzo selezionato, entra in azione questa funzione che posiziona l'utente in un luogo di default preimpostato con delle coordinate già assegnate.

```
<script async defer
src="https://maps.googleapis.com/maps/api/js?key=AIzaSyAQqagAKv0m6sSP-
n5J4si0umvvtwFaRJ4&libraries=places&callback=initMap">

</script>
```

Figura 30 Avvio Script

L'atto conclusivo dell'integrazione con le API di Google Maps è il momento in cui viene richiamato lo script dalla Google Platform. L'attributo script, tipico dell'HTML, è stato ampliato con due attributi quali:

- **Async:** evita di interrompere il rendering dell'HTML durante il download dello script, che di fatto avviene in parallelo;
- **Defer:** evita di interrompere il rendering dell'HTML, il parsing della pagina non viene mai messo in pausa;

3.3 Chat real time

Terminata l'esperienza d'uso con le funzionalità offerte dalla sezione "PlacesJS", si passa infine alla Chat real time.

È stata concepita per dare la possibilità agli utenti che navigano il sito Web di potersi scambiare informazioni utili riguardo i luoghi d'interesse cercati, di poter dare consigli o recensioni riguardo uno di essi, oppure per consigliare agli utenti online uno specifico bar o ristorante, ad esempio.

Di seguito viene riportato il Mock-up della pagina Chat.

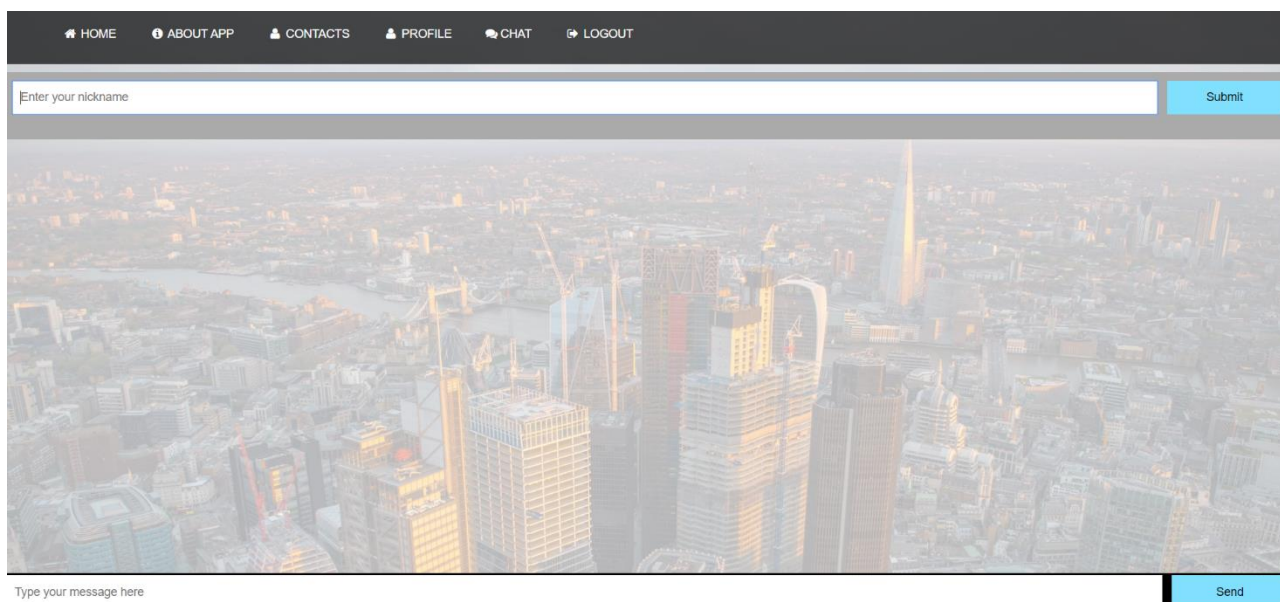


Figura 31 Chat

L'utente ha la possibilità di inserire un nickname per identificarsi con gli altri utenti, che rimarrà invariato per tutta la durata della sessione. Al momento del Logout, l'utente potrà, una volta autenticato nuovamente, poter scegliere un nuovo nickname.

```

<script>
  websocket = new WebSocket("ws://localhost:8080/");
  $('form').submit(function() {
    name = $('#name').val() ? $('#name').val() : 'Anonymous';
    $('#name-div').hide();
    $('#welcome').text('Hello ' + name);
    websocket.send(JSON.stringify({
      name: name,
      message: $('#message').val()
    }));
    $('#message').focus();
    $('#message').val('');
    return false;
  });
  websocket.onmessage = function(evt) {
    $('#messages').append($('- ').html(evt.data));
  };
  websocket.onerror = function(evt) {
    $('#messages').append($('- ').text('<span style="color:
red;">ERROR:</span> ' + evt.data));
  };
</script>

```

Figura 32 WebSocket implementation

Viene istanziato un oggetto WebSocket sulla porta di default 8080, e tramite il JQuery andiamo a reperire le informazioni da quello che abbiamo chiamato “form”.

Tramite la funzione *send*, verrà inviato il contenuto del file JSON che è stato precedentemente parsato a stringa, contenente:

- Messaggio;
- Nome di chi ha inviato il messaggio;

Le successive funzioni di *onmessage* e *onerror* servono rispettivamente per:

- Inserire il messaggio inviato all'interno del contenitore che rappresenta il corpo della chat;
- Gestire eventuali errori in fase di invio del messaggio;

Capitolo 4 – Back-End

4.1 Server

In questa sezione viene trattato il Back-end dell'applicazione, cioè tutta la struttura che non è visibile all'utente, ma che costituisce parte essenziale al funzionamento di tutto il sistema.

È stato stabilito che il Server dovesse girare localmente su **localhost:3000**. Questo file Javascript è costituito per la maggior parte da funzioni *require* che chiamano moduli.

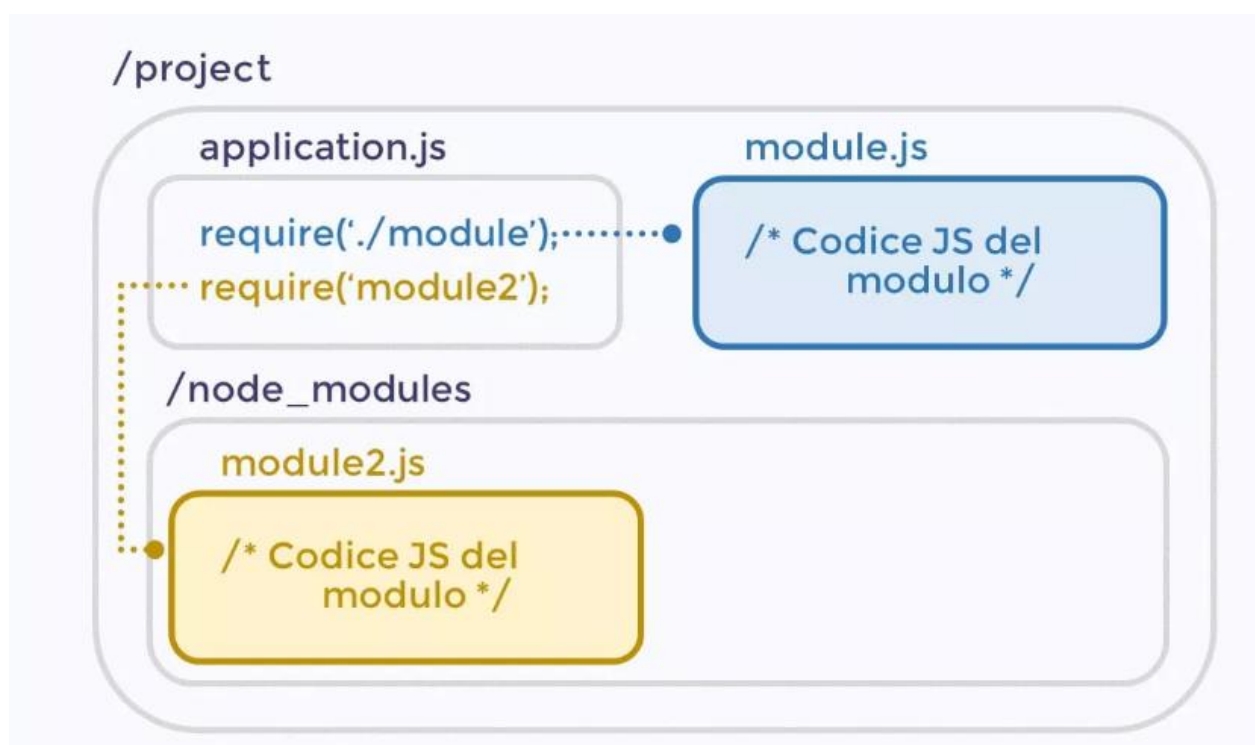


Figura 33 modulo NodeJS

Un modulo non è altro che un classico file JavaScript, il quale verrà trovato da Node e caricato per poter essere utilizzato. I moduli risiedono, solitamente, nella directory denominata *"node_modules"*, ma se il file non è presente in tale cartella, allora Node andrà a cercare un file che ha lo stesso nome più in alto nella struttura ad albero delle directory.

Tutte le funzioni all'interno dei moduli devono avere la dicitura **export**, in quanto tutte le funzioni non esportate nel file del modulo rimarranno private.

```

const express = require('express');
const app = express();
const bodyParser = require('body-parser');
var mongoose = require('mongoose');
var cors = require('cors'); // We will use CORS to enable cross origin domain
requests.
require('dotenv/config');
var flash = require('connect-flash');

const passport      =      require('passport')
  , FacebookStrategy =      require('passport-facebook').Strategy
  , session          =      require('express-session')
  , cookieParser     =      require('cookie-parser')
  , config            =      require('./configuration/config')

require('./configuration/passport')(passport);
app.use(passport.initialize());
app.use(passport.session());
app.use(flash());

app.use(cookieParser());
app.use(session({ secret: 'key', saveUninitialized: true, resave: true}));
app.use(passport.initialize());
app.use(passport.session());

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

app.use(express.static(__dirname + '/views'));
app.set('view engine', 'ejs'); //template engine ejs per la gestione dell'html

```

Figura 34 Server

Il primo modulo da analizzare è quello si è occupato della gestione lato client delle pagine HTML, ovvero **Express**.

Si tratta di un micro-framework per Node.js che offre strumenti base, come per esempio la possibilità di generare percorsi(URL) o di utilizzare i template per creare più velocemente applicazioni in Node.

Il modulo **Body-Parser** invece si occupa di gestire le richieste HTTP di tipo POST, estraendo l'intera parte del corpo di un flusso di richieste in entrata e lo espone sul body. Questo modulo inoltre analizza i dati JSON, e URL codificati inviati utilizzando la richiesta HTTP di tipo POST.

Il modulo **Cors**(cross-origin HTTP request) si occupa di gestire le richieste di un client che invoca una risorsa di un differente dominio, protocollo o porta.

4.2 Database

Si tratta di un archivio strutturato che permette di memorizzare permanentemente le informazioni, rendendole disponibili per successive elaborazioni.

Il compito principale di un database è quello di effettuare ricerche tra le tuple, nonché le utenze dell'applicazione, dopo averle memorizzate all'interno del proprio sistema.

Il database che è stato utilizzato per l'implementazione di questo progetto è MongoDB. **MongoDB** è un database *document-oriented*, il che significa che i dati vengono archiviati nella forma di document, il document rappresenta la singola informazione, e sono le **collections** che racchiudono più documents al loro interno.

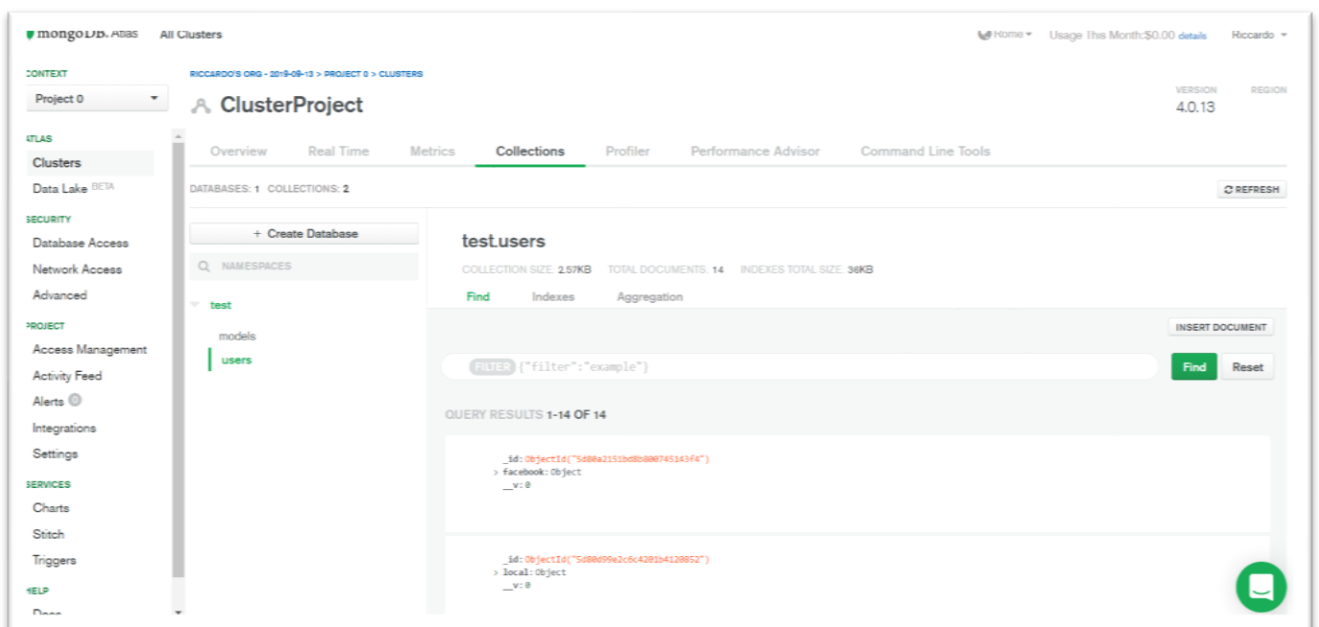


Figura 35 Cluster MongoDB

L'interfaccia MongoDB Atlas è l'ambiente Cloud in cui vengono ospitati i database MongoDB. Completata la configurazione iniziale, sarà quindi possibile creare utenti ed assegnare autorizzazioni limitate.


```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  console.log('Connected to DB');
});
//Connessione al DB
mongoose.connect(process.env.DB_CONNECTION,
  { useNewUrlParser: true,
    useUnifiedTopology: true } );
```

Figura 36 DB configuration

La creazione del database all'interno del file `server.js` è affidata a **Mongoose**. Si tratta di un modellatore di oggetti del nostro database e fornisce una soluzione semplice e basata su schema per modellare i dati dell'applicazione. Include il cast di tipo integrato, la convalida e la creazione di query. Formando uno schema di lavoro, detto *Model*, viene usato per instanziare oggetti che saranno automaticamente dotati di metodi per svolgere le classiche operazioni di CRUD.

La funzione `db.on` si occupa di aprire l'istanza del database all'interno della nostra applicazione, e tramite terminale se tutto è andato a buon fine, viene notificata la corretta connessione al database.

La funzione `mongoose.connect` è il modo più semplice per connettersi a mongoDB usando Mongoose. Una volta connesso si potrà creare un modello Mongoose ed iniziare a interagire con il database.

4.3 RabbitMQ

La tecnologia offerta da RabbitMQ è stata utilizzata all'interno del progetto per sviluppare un riepilogo dei task che l'utente effettua all'interno dell'applicazione. Ogni visita di una pagina viene infatti notificata con un semplice messaggio sul terminale, immaginando l'utente come fosse il “*sender*” di messaggi in uno schema di comunicazione asincrona.

Mittente e destinatario, infatti, non devono agire nello stesso momento.

Tale comunicazione sfrutta le opportunità messe a disposizione dal protocollo TCP, eseguendo il server di RabbitMQ sulla porta 5672 su Localhost.

```
var amqp = require('amqplib/callback_api');

module.exports = function(app, passport){

app.get('/about', function(req,res) {
  amqp.connect('amqp://localhost', function(error0, connection) {
    if (error0) {
      throw error0;
    }
    connection.createChannel(function(error1, channel) {
      if (error1) {
        throw error1;
      }

      var queue = 'queue_example';
      var msg = 'You are at about page!';

      channel.assertQueue(queue, {
        durable: false
      });
      channel.sendToQueue(queue, Buffer.from(msg));

      console.log(" ---- Sent: %s ----", msg);
    });
  });

  res.render('./about.ejs', { user: req.user });
})
```

Figura 37 route example with RabbitMQ

La route presa in esame nell'esempio precedente è quella relativa alla redirect verso la pagina About App.

La variabile *amqp* rappresenta la richiesta di utilizzo del modulo di libreria *amqp*, che offre le funzionalità per l'uso corretto di RabbitMQ.

All'interno della classica struttura di una get, troviamo la callback, che per prima cosa si occupa della connessione del server di Rabbit al Localhost.

Successivamente si crea il cosiddetto **canale di comunicazione**, formato da una coda dove risiederanno i messaggi prima del loro invio e dal messaggio stesso che si desidera inviare.

La funzione *assertQueue* afferma l'esistenza della coda e impostando il flag *durable* a *false*, la coda verrà resettata dopo ogni riavvio del server.

Infine la funzione *sendToQueue*, che prende in input la coda e il messaggio, ha il compito di inviare al destinatario il messaggio contenuto all'interno della coda.

Il reciever, che è in attesa di ricevere messaggi, avrà pressochè la stessa struttura, fatta eccezione per la funzione *sendToQueue*, che è stata sostituita dalla funzione *consume*.

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(error0, connection) {
  if (error0) {
    throw error0;
  }
  connection.createChannel(function(error1, channel) {
    if (error1) {
      throw error1;
    }

    var queue = 'queue_example';

    channel.assertQueue(queue, {
      durable: false
    });

    console.log("Waiting for messages in %s.\n", queue);

    channel.consume(queue, function(msg) {
      console.log(" ---- Task: %s ----", msg.content.toString());
    }, {
      noAck: true
    });
  });
});
```

Figura 38 reciever RabbitMQ

4.4 Web Socket

Le Web Socket sono uno strumento molto utile che viene incontro a tutti quei sviluppatori che hanno intenzione di progettare una chat.

Una volta registrati e loggati all'interno del sito Web, l'utente ha la possibilità di comunicare con altri utenti collegati per scambiarsi informazioni riguardo luoghi d'interesse, o anche per chiedere consiglio su uno di essi.

```
WebSocketServer = require('ws').Server,  
wss = new WebSocketServer({  
  port: 8080  
});  
  
wss.broadcast = function broadcast(data) {  
  wss.clients.forEach(function each(client) {  
    client.send(data);  
  });  
};  
  
wss.on('connection', function(ws) {  
  console.log('connected');  
  ws.on('message', function(msg) {  
    data = JSON.parse(msg);  
    if (data.message) wss.broadcast('<strong>' + data.name + '</strong>:  
    ' + data.message);  
  });  
});
```

Figura 39 Web Socket on Server

L'implementazione della Web Socket inizia dalla richiesta del modulo 'ws' che permetterà di utilizzare le funzioni e metodi offerti da tale tecnologia.

La porta di default su cui risiede la Web Socket è la 8080.

Tramite la funzione di broadcast, Web Socket invierà a tutti gli utenti, che in questo caso sono contrassegnati come client, i messaggi che si desidera inviare.

Una volta stabilita, infine, la connessione tramite la funzione *wss.on*, si notifica sul terminale la reale connessione alla Web Socket e sul client degli utenti connessi, nella sezione Chat, verrà mostrato in broadcast il messaggio inviato, precedentemente convertito in formato JSON.

Capitolo 5 – Dispiegamento

5.1 Installazione

La fase d'installazione della directory del progetto prevede, una volta connessi ad Internet, di scaricare tramite piattaforma GitHub, la repository PlacesJS tramite il seguente link: <https://github.com/DanieleIacomini/PlacesJS>

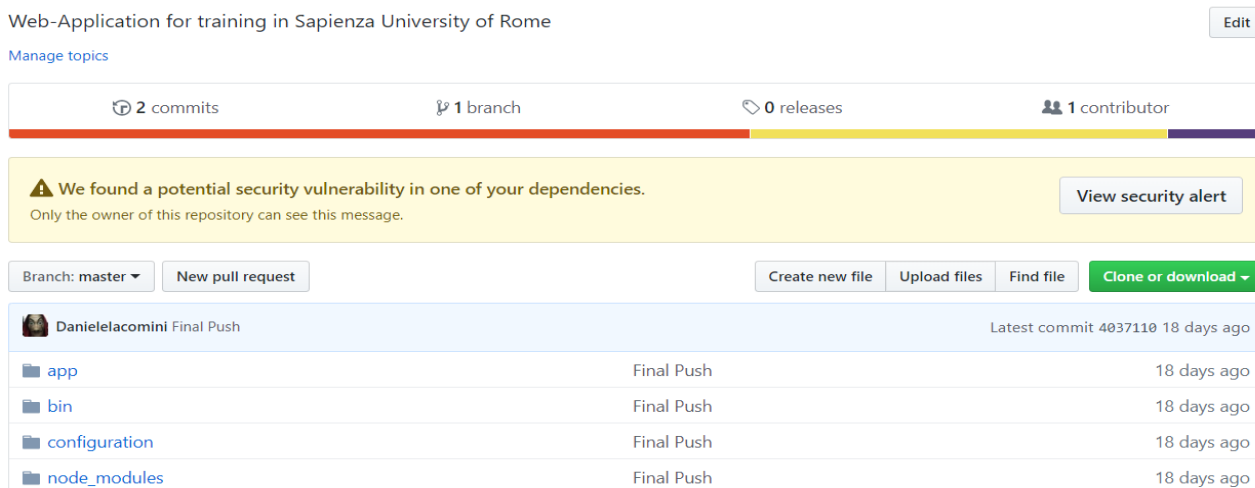


Figura 40 GitHub repository

Prima di avviare il server, bisogna eseguire alcuni comandi per l'installazione dei moduli utili al funzionamento di Node.js . Eseguire, quindi, nell'ordine i seguenti comandi:

- **sudo apt-get update**: aggiorna prima l'indice del pacchetto locale;
- **sudo apt-get install nodejs**: installa il pacchetto relativo a Node.js;
- **sudo apt-get install npm**: installa Npm, cioè il gestore dei pacchetti più usato dagli sviluppatori Node.js;

Una volta eseguite queste operazioni, il comando da eseguire per avviare il server, che partirà su un indirizzo IP locale (localhost) e sulla porta 3000, è il seguente: **node server.js** (o in alternativa **npm start**).

