

 <b>POLITECNICO</b> MILANO 1863	<b>Geopic</b> <b>Specifica Tecnica</b>	27-06-2021 Pag. 1 di 70	
--	---	----------------------------	---

# DOCUMENTAZIONE SOFTWARE GEOPIC

Flaminio Villa & Vasile Andrei Calin | Politecnico di Milano



Geopic nasce come applicazione mobile per la gestione di convenzioni e sconti per esercizi commerciali.

Grazie ad una grafica accattivante assiste l'utente nella ricerca degli esercizi commerciali convenzionati a lui più vicini, offrendo filtri accurati, descrizioni e informazioni utili.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021 Pag. 2 di 70	
---	--	----------------------------	---

## INDICE

1. OGGETTO E SCOPO
2. GLOSSARIO
3. ANALISI FUNZIONALE E CASI D'USO
4. PROGETTAZIONE FRONT-END
  - 4.1 INTRODUZIONE A FLUTTER & DART
  - 4.2 INTRODUZIONE ALLA STRUTTURA DEL PROGETTO
  - 4.3 PROGETTAZIONE SOFTWARE E CLASS DIAGRAM DI PROGETTO
  - 4.4 LE SCHERMATE DELL'APPLICAZIONE
  - 4.5 IL ROUTING
  - 4.6 GESTIONE DELLO STATO DELL'APP
    - 4.6.1 BLOC PATTERN
    - 4.6.2 GESTIONE DELLO STATO DI UN BLOC
    - 4.6.3 COMUNICAZIONE TRA BLOC
  - 4.7 GESTIONE DI EVENTI CON CONTROLLER
  - 4.8 COMUNICAZIONE BACKEND TRAMITE REPOSITORY
  - 4.9 TESTING
5. PROGETTAZIONE BACK-END
  - 5.1 ACTIVITY DIAGRAM
  - 5.2 DEFINIZIONE DELLE ENTITÀ DI SISTEMA
    - 5.2.1 DESIGN DELLA BASE DI DATI
  - 5.3 ANALISI SOFTWARE
  - 5.4 ANALISI DELL'ALGORITMO DI RICERCA DELLE STRUTTURE
    - 5.4.1 APPROFONDIMENTO ALGORITMO DI VINCENTY GEODESY
  - 5.5 SECURITY
    - 5.5.1 CORS CONFIGURATION
  - 5.6 API
  - 5.7 TESTING
  - 5.8 Docker
6. CONCLUSIONE

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021 Pag. 3 di 70	
--	--	----------------------------	---

## 1 OGGETTO E SCOPO

Il focus dell'applicativo è la visualizzazione delle strutture convenzionate in modo strutturato ed intuitivo con una suddivisione in base al tipo di struttura ed in base al tipo di servizio offerto.

È disponibile una ricerca per categorie, per tipo di struttura e per nome ed una gestione relativa alla posizione in tempo reale che mostra le strutture convenzionate più vicine all'utente.

L' applicazione ha diverse funzionalità che permettono l'accesso in modo semplice e veloce alle informazioni riportate su un sito che verranno importate tramite Web Scraping esempio news, eventi, che verranno importate automaticamente, o gestite tramite i profili amministratore.

È disponibile anche una mappa personalizzata appositamente per il servizio che permette la visualizzazione della posizione di tutte le strutture convenzionate, divise per regioni ed identificate univocamente dal tipo di struttura.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021 Pag. 4 di 70	
--	--	----------------------------	---

## 2 GLOSSARIO

### Controller

Questa è una specializzazione della classe *@Component*, che ci consente di rilevare automaticamente le classi che contengono i metodi di gestione delle richieste.

### Service

All' interno dei Service vengono inseriti tutti i controlli relativi ai parametri che verranno passati dal controller verso gli Helper.

### Helper

Gli Helper implementano la business logic ed interagiscono con le repository.

### JWT

JSON Web Token (JWT) è uno standard aperto che definisce un modo compatto e autonomo per trasmettere in modo sicuro le informazioni tra le parti come oggetto JSON

### CORS

Il Cross-Origin Resource Sharing (CORS) è un meccanismo che usa header HTTP aggiuntivi per indicare che un dominio dispone dell'autorizzazione per accedere alle risorse selezionate.

### GeoPoint

Il **GeoPoint** è un oggetto che contiene latitudine e longitudine di una coordinata geografica.

### GeoHash

Dati due valori double effettua un encoding binario che rappresenta l'unione dei **GeoPoint**

### BoundingBox

Una **BoundingBox** è un contenitore quadrato che ha come vertici le coordinate di latitudine e longitudine, grazie all'algoritmo di Vincenty geodesy verranno trovate le strutture che hanno **GeoPoint** interni alla bounding box.

### GeoHashCircle

Dato un **GeoPoint** ed un raggio crea una approssimazione di un cerchio in quadrato, che corrisponderà poi ad una **BoundingBox**.

### DTO


L'oggetto di trasferimento dati (**DTO**), è un modello di progettazione utilizzato per trasferire i dati tra i sottosistemi dell'applicazione software.

### 3 ANALISI FUNZIONALE E CASI D'USO

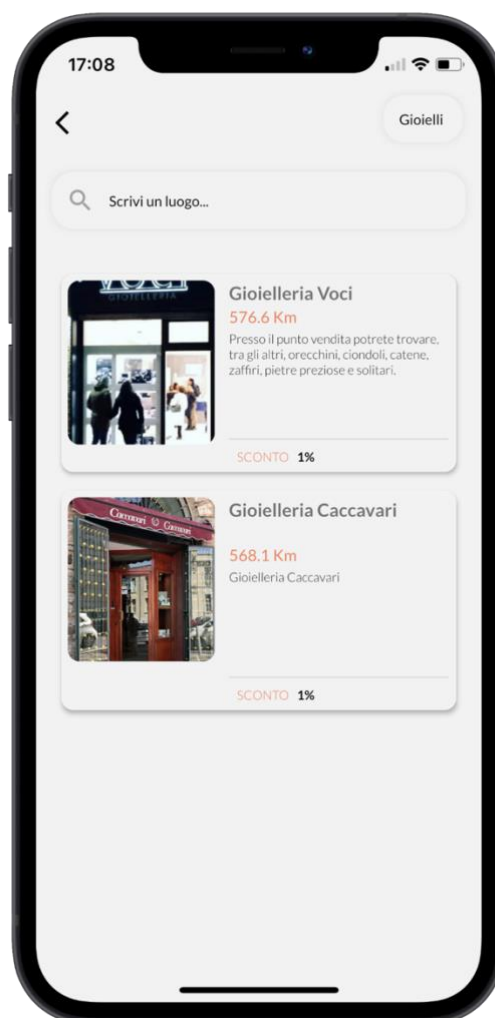
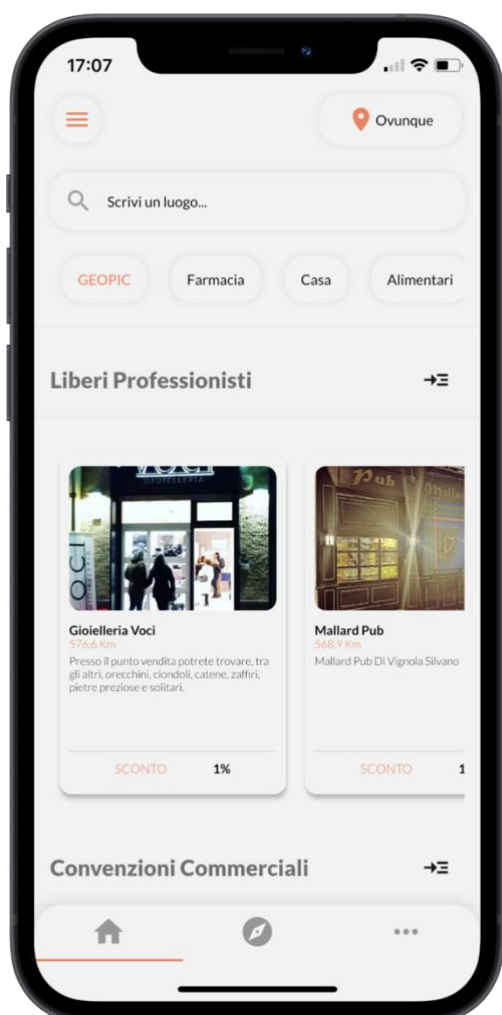
L'applicativo segue il modello client-server, permette all'utente finale l'accesso diretto ai dati delle convenzioni localizzati su un database.

Di seguito è possibile vedere la schermata principale dell'applicativo.

Dispone di una ricerca per categorie, per tipo di struttura e per nome ed una gestione relativa alla posizione in tempo reale che mostra le strutture convenzionate più vicine all'utente.

Il pulsante in alto a destra  Ovunque permette di cambiare la località di ricerca, è infatti possibile ricercare le strutture convenzionate di una determinata località, oltre a quelle vicine rispetto la posizione fisica del dispositivo.

Queste sono alcune immagini prese dall'applicazione che sta funzionando sui nostri dispositivi

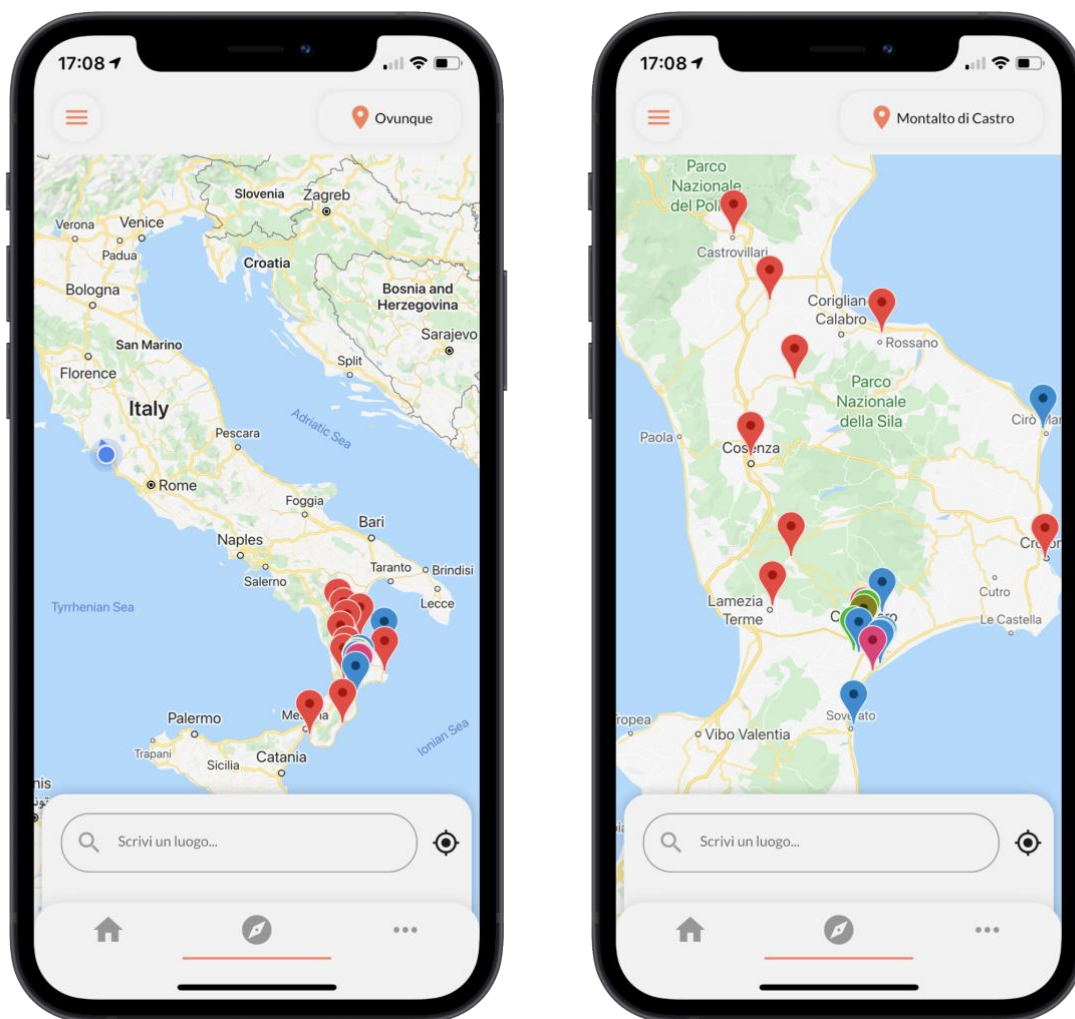


 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021 Pag. 6 di 70	
--	--	----------------------------	---


### Strutture localizzate

È disponibile anche una mappa personalizzata appositamente per il servizio che permette la visualizzazione della posizione di tutte le strutture convenzionate, divise per regioni ed identificate univocamente dal tipo di struttura. E' infatti possibile variare il colore del punto visualizzato sulla mappa, per rendere distinguibile la diversa categoria di esercizio commerciale.

Queste sono alcune immagini prese dall'applicazione che sta funzionando sui nostri dispositivi.




La mappa implementa anche una barra di ricerca delle varie località così da rendere fruibile la sua navigazione e la visualizzazione di tutte le strutture offerte.

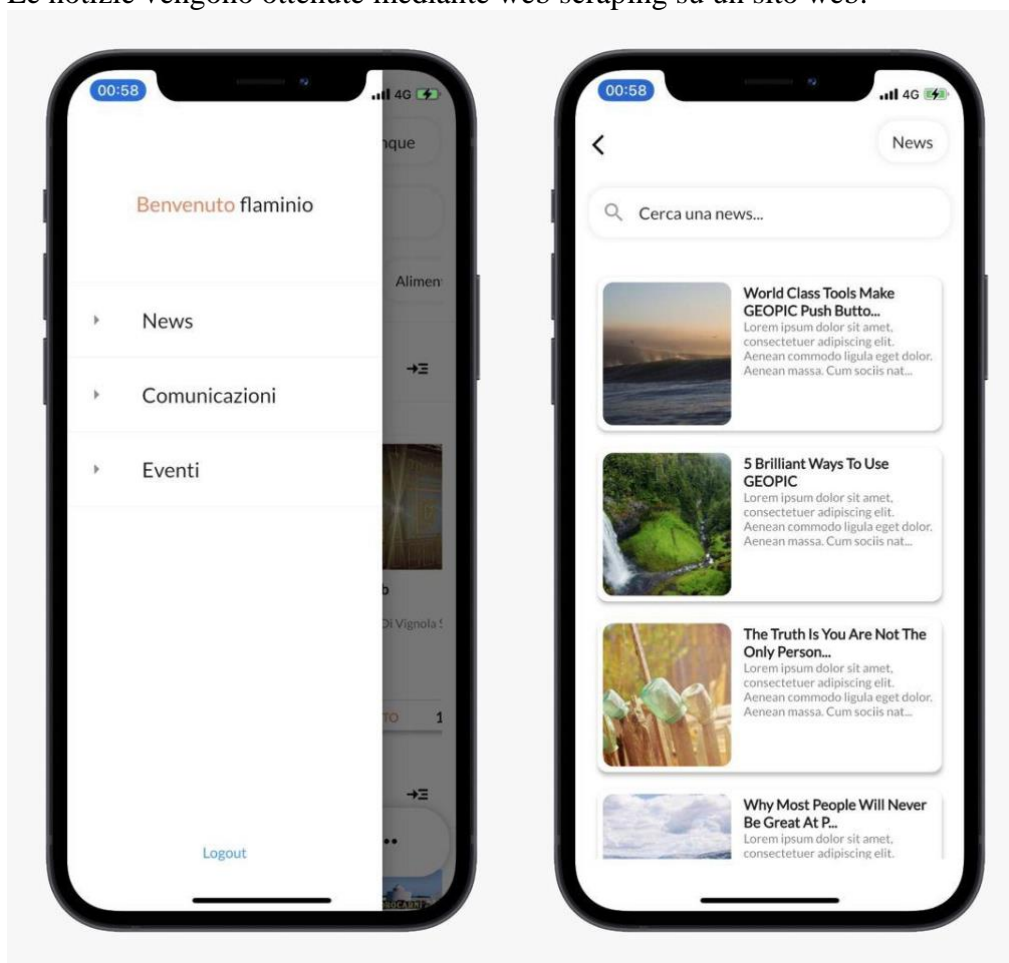
Quando viene premuto il pulsante  la mappa sposterà automaticamente il suo punto focale rispetto la posizione geografica del dispositivo.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021 Pag. 7 di 70	
--	--	----------------------------	---

I tasti di navigazione portano dalla pagina principale verso la mappa oppure verso una pagina che contiene informazioni descrittive.

Scorrendo da sinistra verso destra oppure attraverso il pulsante  si può avere accesso alla barra laterale, che contiene le sezioni di notizie.

Le notizie vengono ottenute mediante web scraping su un sito web.



E' possibile effettuare ricerche fra le varie notizie e cliccando sopra si vede il contenuto completo della news.

Per tornare al menu precedente basta usare il tasto indietro o scorrere da destra verso sinistra.

Il flusso di gestione della piattaforma è la suddivisione a zone di tutto il territorio Italiano con ogni zona gestita da del personale autorizzato che inserisce le convenzioni della zona, ogni zona ovviamente deve avere uno o più amministratori segreteria che inizialmente dovranno creare e gestire il personale di quella zona. La gestione delle zone è adibita a degli amministratori nazionali che avranno il controllo di tutte le regioni, strutture e profili del nostro servizio.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021 Pag. 8 di 70	
--	--	----------------------------	---

## 4 PROGETTAZIONE FRONT-END

Iniziamo l'introduzione del frontend con una overview delle tecnologie usate, dove la scelta è ricaduta su un framework relativamente nuovo che sta prendendo sempre più piede nello sviluppo frontend. **Flutter** è un framework realizzato da Google basato su un linguaggio di programmazione chiamato **Dart** che offre la possibilità di realizzare applicazioni cross platform per mobile, desktop e web. Flutter a differenza di altri framework come Angular o librerie come React, ha un approccio completamente diverso per la realizzazione della **UI** infatti non si basa su concetti come HTML e CSS ma implementa una sua metodologia per la realizzazione di quest'ultima che analizzeremo nel corso della documentazione.

Vorrei soffermare l'attenzione sul linguaggio di programmazione che usa, Dart, un linguaggio che rispetta tutti i concetti della programmazione ad oggetti ed inoltre è fortemente tipizzato, tutto ciò lo rende un'ottima scelta per lo sviluppo, riducendo gli errori in esecuzione che potrebbero avvenire.

### 4.1 Introduzione a Flutter & Dart

Prima di procedere con l'analisi del front end ritengo doveroso affrontare i concetti base di questo framework ed del linguaggio da esso utilizzato.

Ogni componente visivo in Flutter viene chiamato **Widget**, ogni cosa in Flutter è un widget, anche un rettangolo vuoto è un widget.

Esistono due tipi di widget, gli **StateLess Widget** e gli **StateFul Widget**, analizziamo la differenza.

Gli StateLess Widget sono dei componenti prettamente visivi dove all'interno non è contenuta nessun tipo di logica ma descrivono come un certo componente deve apparire, ovviamente è possibile passare dei parametri a questo tipo di widget così da popolare il suo contenuto!

Ogni componente dell'applicazione che svolge un ruolo prettamente visivo sarà sicuramente di questo tipo.

Osserviamo il codice per creare uno di questi Widget:

```
class CircularRoundedContainer extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return ...
  }
}
```

Come possiamo notare ogni Widget è una classe che estende StatelessWidget, infatti come introdotto prima anche la parte UI in flutter si basa su concetti di OOP, il widget preso in esame è utilizzato per la realizzazione di tutti i "container", rettangoli a bordi arrotondati nell'applicazione.



Come possiamo notare questo tipo di classe che estende StatelessWidget deve implementare il metodo build che permette il rendering del componente visivo, questo metodo infatti è chiamato ogni volta che l'applicazione deve effettuare un rendering della UI.

La classe StatelessWidget è fornita dal framework e non verrà quindi discusso più del dovuto.

Procediamo con l'analisi del StatefulWidget, quest'ultimo ha la capacità di avere uno stato, quindi possiamo vedere lo stato come l'insieme dei valori degli attributi assunti da un'istanza in uno specifico momento.

```
class ListSectionBuilder extends StatefulWidget {  
  final List<Section> sections;  
  final String location;  
  ListSectionBuilder({Key key, @required this.sections, this.location}) : super(key:  
key);  
  
  @override  
  _ListSectionBuilderState createState() =>  
  _ListSectionBuilderState.sections, location);  
}  
  
class _ListSectionBuilderState extends State<ListSectionBuilder> {  
  final List<Section> sections;  
  final String location;  
  _ListSectionBuilderState(this.sections, this.location);  
  
  @override  
  Widget build(BuildContext context) {...}  
}
```

Come possiamo vedere questo tipo di componente è formato da due classi la prima che definisce lo StatefulWidget con il metodo createState e la seconda classe che definisce lo State vero e proprio dove avviene il build del component. La particolarità di questa classe è che introduce il metodo **setState()**, questo metodo permette di effettuare il refresh della ui in base alle variabili che vengono assegnate al suo interno, quindi un vero e proprio cambiamento di stato.

## 4.2 Introduzione alla struttura del progetto

Adesso che sono state introdotte le basi fondamentali di questo framework procediamo con l'analisi del progetto, partendo dall'architettura del progetto e la suddivisione delle cartelle/file.

In un progetto Flutter la cartella che contiene i codici è nominata **lib** , e questa è la struttura adottata dal progetto:

- app
  - files...
- app\_widgets
  - files...
- core
  - app\_theme
    - cubit
    - files...
  - controller
    - implementations
    - files...
  - models
    - files...
  - repositories
    - implementations
    - files...
- pages
  - browse
    - bloc
      - files...
    - view
      - files...
  - card
    - view
      - files...
- //stessa struttura interna
  - category
  - home
  - login
  - macro\_category
  - settings
  - splash\_screen
- routing
  - files...
- tad\_widgets
  - view
    - //stessa struttura interna view/bloc
      - app\_bar
      - drawer
      - tab\_bar

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 11 di <b>70</b>	
--	--	---	---

Come best practice flutter usa una nomenclatura lowercase con divisione delle parole tramite underscore, quindi differisce completamente dalla nomenclatura comune a java. Internamente per le classi e variabili viene utilizzata una nomenclatura camelCase.

Il progetto si basa su una struttura model view controller rivisitata, infatti vengono divise le varie schermate in sottocartelle, e per ogni schermata si cerca di dividere il più possibile la business logic dalla presentation.

Partendo dall'alto verso il basso nella cartella **app** sono contenuti solo i file per l'inizializzazione dell'applicazione dove vengono definiti i parametri di default per l'applicazione.

Nella cartella **app\_widgets** sono contenuti tutti i file che definiscono i widgets che sono condivisi tra più schermate.

Nella sezione **core** si trovano tutti le componenti che vengono utilizzate da più elementi dell'app, come le repository che vengono implementate dentro la business logic di ogni pagina, o dei controller per la gestione avanzata di particolari eventi. In oltre sono contenuti le configurazioni del tema della **UI**, costanti o altre variabili per la parte grafica.

Nella sezione **pages**, che è anche la parte più complessa dove sono effettivamente contenute tutte le schermate della ui, possiamo trovare una cartella per ogni pagina che viene suddivisa a sua volta in **view** e **bloc** (pattern utilizzato per la parte di business logic).

Procedendo ulteriormente nella folder structure troviamo la cartella **routing** dove sono definite tutte le path delle schermate e dove viene definita la navigazione da una schermata all'altra.

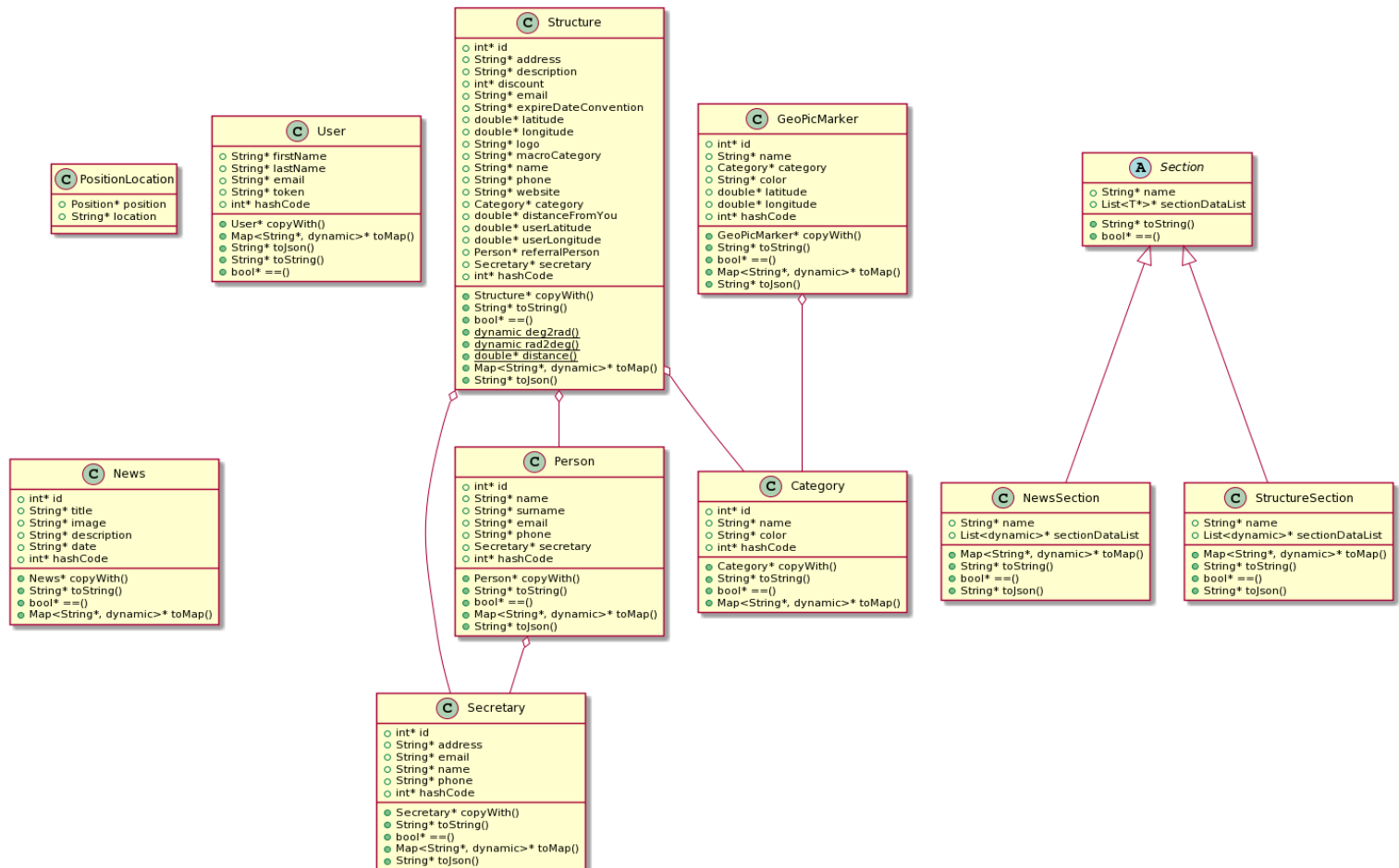
Per concludere troviamo la cartella **tad\_widgets** dove TAD è un acronimo che sta per (TabBar,AppBar,Drawer) che sono dei componenti core dell'applicazione e necessitano quindi di una loro sezione apposta così da non essere confusi con altri widget.

## 4.3 Progettazione Software e Class Diagram di progetto

Dopo aver introdotto brevemente di cosa si tratta in ogni macro categoria della folder structure procediamo nel vedere e analizzare realmente i modelli della nostra applicazione rappresentando come sono collegati tra loro. L'applicazione presenta undici modelli distinti che servono per garantire la concreta fruibilità del contenuto offerto e per avere una vera e propria architettura anche a livello UI. Molti modelli "ovviamente" sono simili o uguali a quelli lato backend, ci possono essere differenze o dati in più che servono per gestire la parte UI.

Modelli:

- Category
- GeopicMarker
- News
- StructureSection
- NewsSection
- Section
- Person & User
- PositionLocation
- Secretary
- Structure



Come è possibile vedere nel nel class diagram notiamo che il modello della struttura è il modello più complicato dove abbiamo dei collegamenti di aggregazione con le classi Person (1 a N), Secretary (1 a 1) e Category (1 a 1).

Possiamo inoltre notare anche le classi NewsSection e StructureSection che estendono la classe Section. Questa classe serve per la creazione di una “Sezione” che rappresenta la parte di ui dove vengono mostrati i componenti “Card” che mostrano le relative info.

Le classi PositionLocation, GeoPicMarker sono classi helper che permettono una gestione facilitata rispettivamente della gestione della localizzazione e della gestione dei marker della mappa.

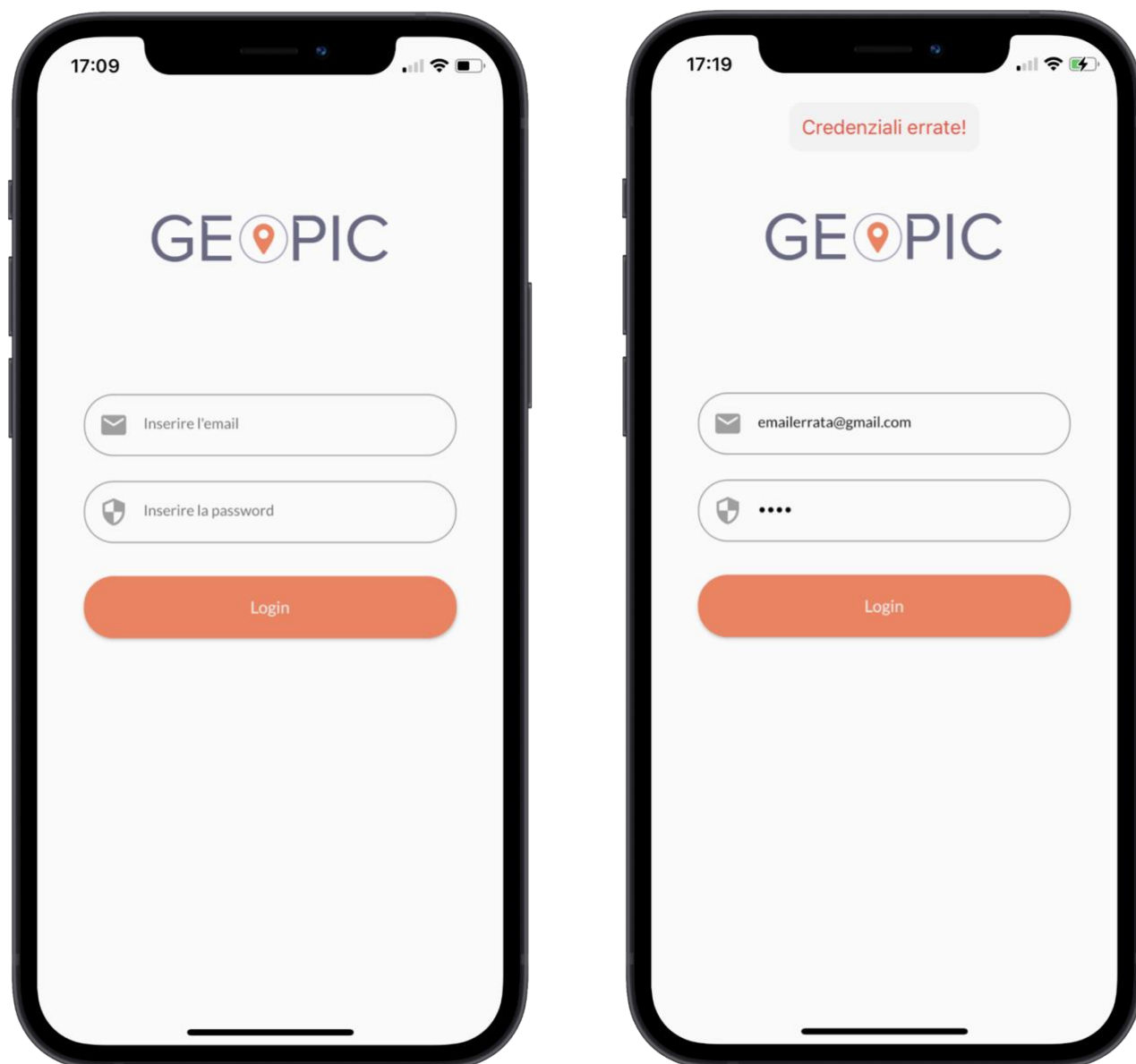
La classe User serve per la gestione dell’autenticazione di un utente, infatti contiene le informazioni dell’utente attualmente connesso e del token JWT token che permette di autenticarsi quando effettua una chiamata backend.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 13 di 70	
---	--	---------------------------------	---

## 4.4 Le schermate dell'applicazione

Introduciamo le varie schermate dell'applicazione e cosa un utente può fare e visualizzare in ognuna di esse.

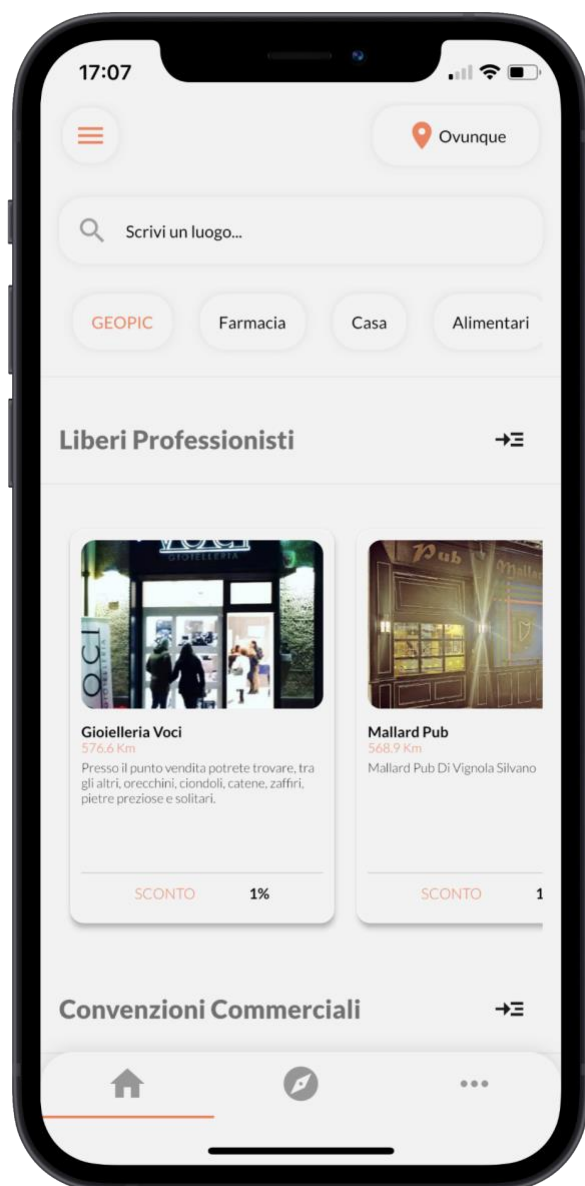
L'applicazione presenta come prima schermata quella di login, necessaria per potersi autenticare e accedere al servizio.



Sono presenti due capi di testo per inserire email e password e un pulsante di login, Se le credenziali sono errate verrà mostrato a schermo un popup con il messaggio di errore.

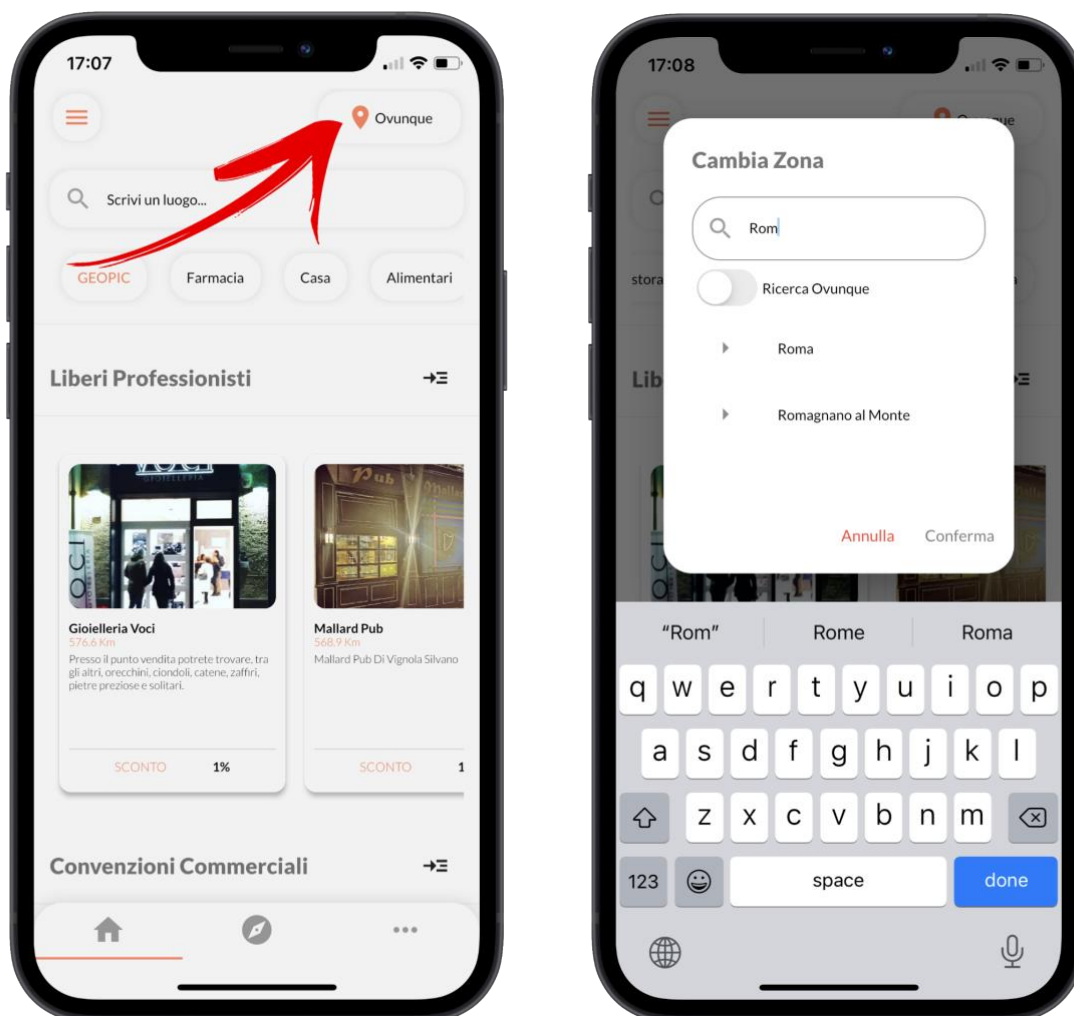
 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 14 di 70	
---	--	---------------------------------	---

Se il login avviene con successo, l'utente viene reindirizzato alla home page dell'applicazione dove dopo un breve caricamento è possibile visualizzare le strutture convenzionate in zona. Questa pagina si suddivide in quattro parti fondamentali, l'App Bar (la barra in alto), il campo di ricerca, la lista delle categorie e la lista delle strutture suddivise per macro categorie.



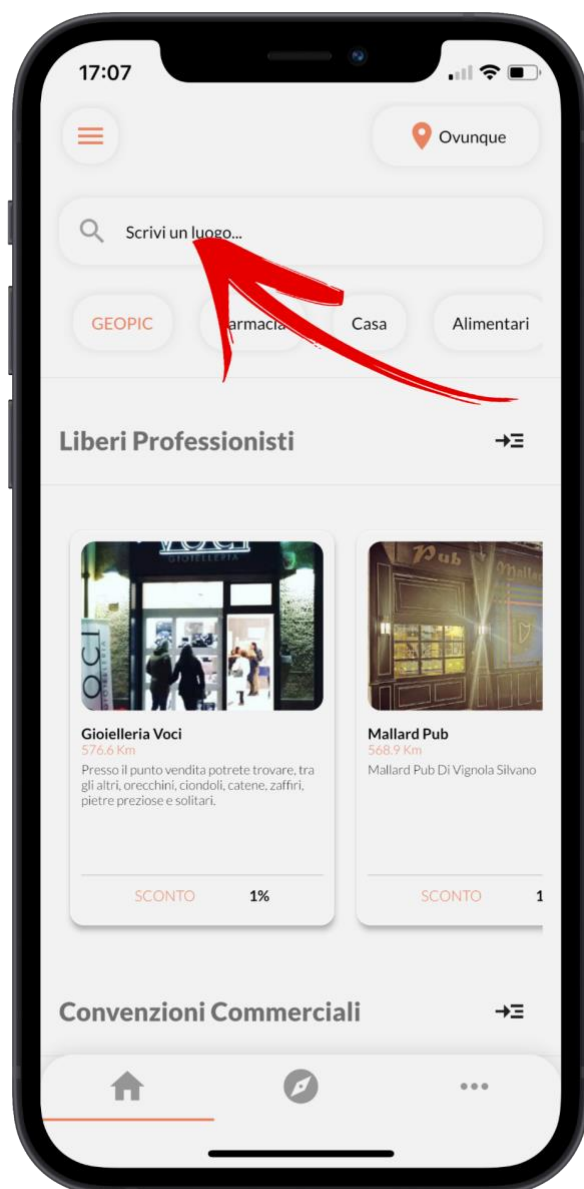
Procediamo con l'analisi di ogni sua componentistica.

L'**App Bar** permette di selezionare una località diversa da quella attuale o scegliere di visualizzare tutte le strutture selezionando l'opzione ovunque.



 <p><b>POLITECNICO</b> MILANO 1863</p>	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	<p>27-06-2021</p> <p>Pag. 16 di 70</p>	
---	--	--	---

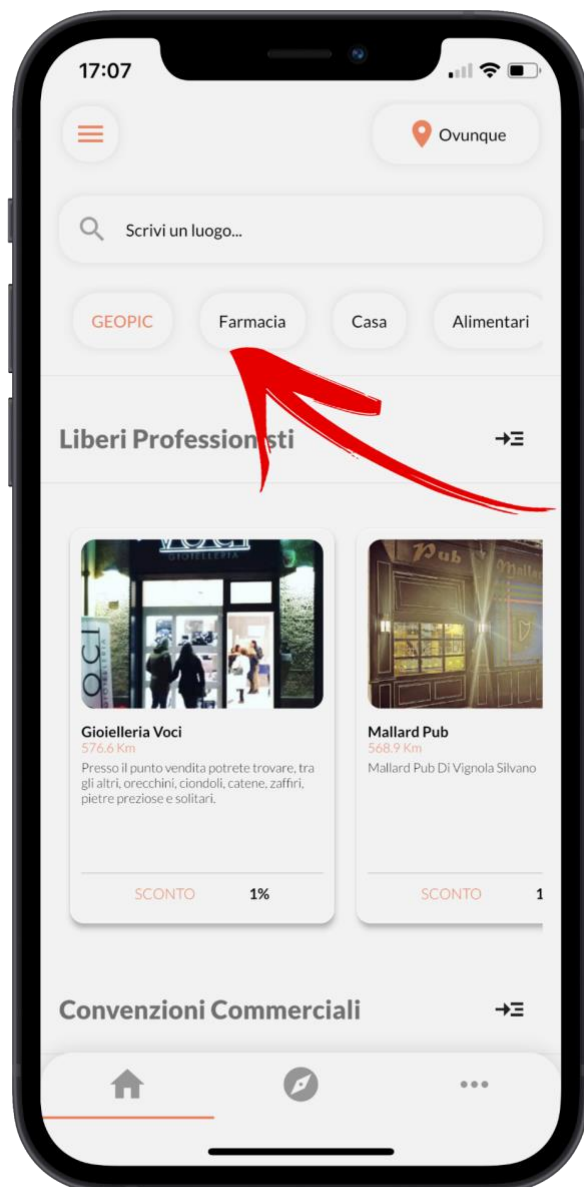
Il **campo di ricerca** permette di filtrare le strutture convenzionate in base al loro nome.





 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 17 di <b>70</b>	
---	--	---	---

La **lista delle categorie** permette di aprire una nuova pagina dove vengono visualizzate solo le strutture appartenenti a quella categoria. (Verrà affrontata in seguito questa pagina).

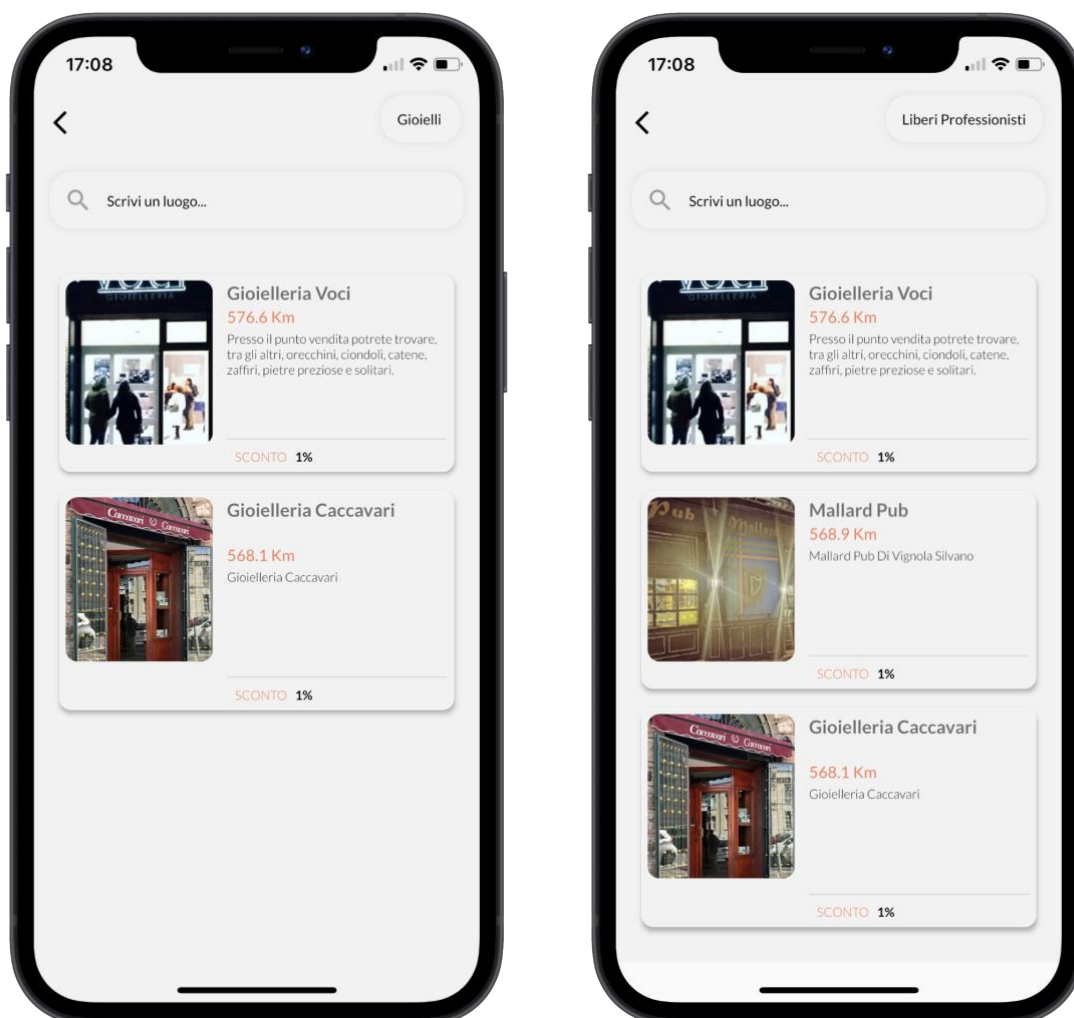


La lista delle categorie è una lista con scorrimento orizzontale (che viene fornita dal backend) con tutte le categorie presenti nel servizio, così da poter aggiungere una nuova categoria senza avere la necessità di aggiornare l'app.

La **lista delle macro-categorie** dove ognuna di essa contiene all'interno una lista di strutture, è possibile aprire questa sezione tramite l'apposito pulsante così da avere una migliore fruizione dei dati.

Procediamo con l'analisi delle pagine di visualizzazione di categoria e macro-categorie che si basano sullo stesso layout.

La schermata è suddivisa in titolo, barra di ricerca e la lista effettiva per la visualizzazione.



Non verranno approfonditi i componenti dato che sono simili tra le varie schermate.

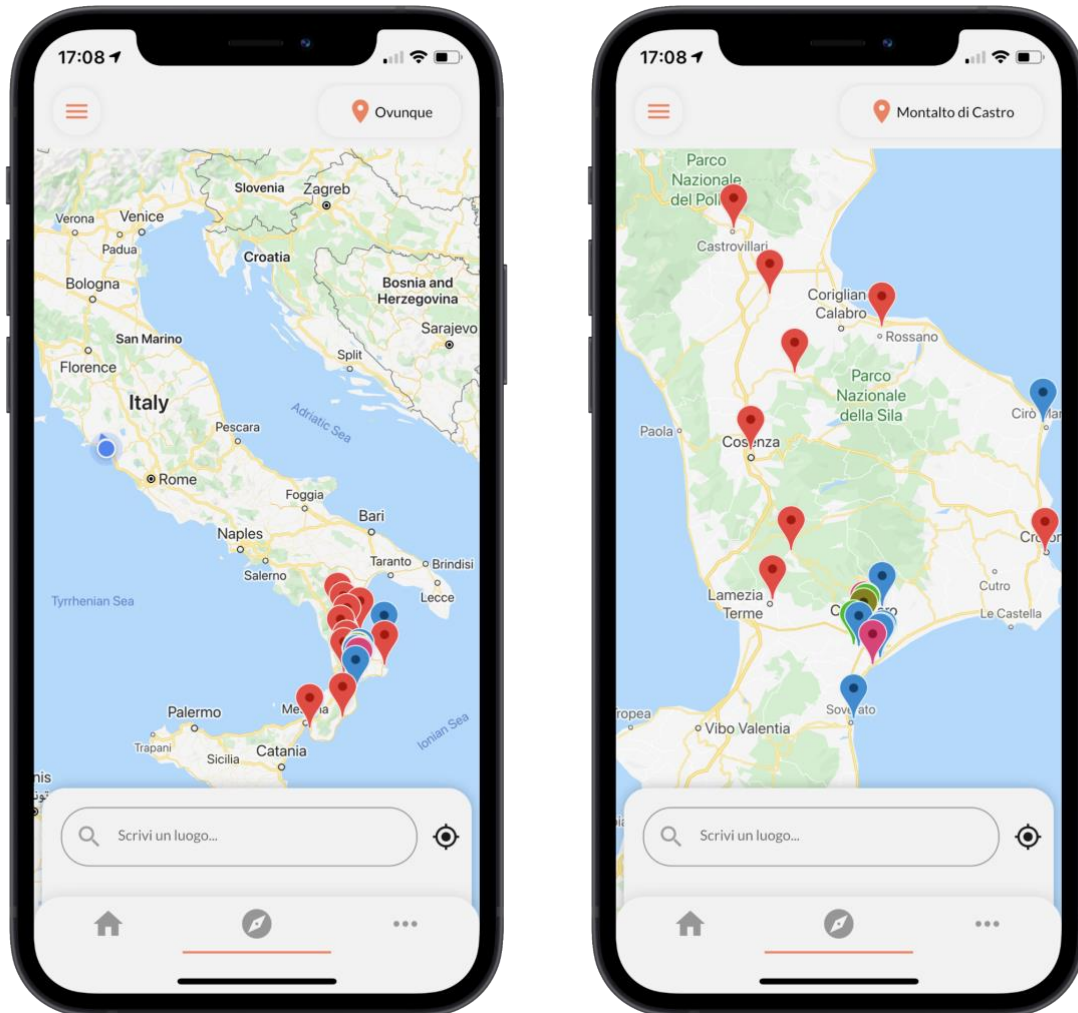
 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 19 di 70	
--	--	---------------------------------	---

Cliccando su una struttura (da qualsiasi parte dell'applicazione quindi da una card o da un marker) è possibile vedere tutte le informazioni di una determinata struttura, come è possibile vedere dalle foto.



Per migliorare l'esperienza utente è stata introdotta una visualizzazione tramite mappa delle strutture, questa schermata è accessibile dalla barra di navigazione in basso.

 <p><b>POLITECNICO</b> MILANO 1863</p>	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	<p>27-06-2021</p> <p>Pag. 20 di 70</p>	
---	--	--	---



Come è possibile vedere questa schermata condivide con l'home page , l'App Bar che permette di fare tutto ciò che era possibile fare prima.

Questa schermata mostra sulla mappa tramite dei Marker la posizione delle strutture convenzionate in base alla zona dove ci si trova, è possibile cercare il nome di una città tramite la barra di ricerca per essere reindirizzati alla posizione desiderata.

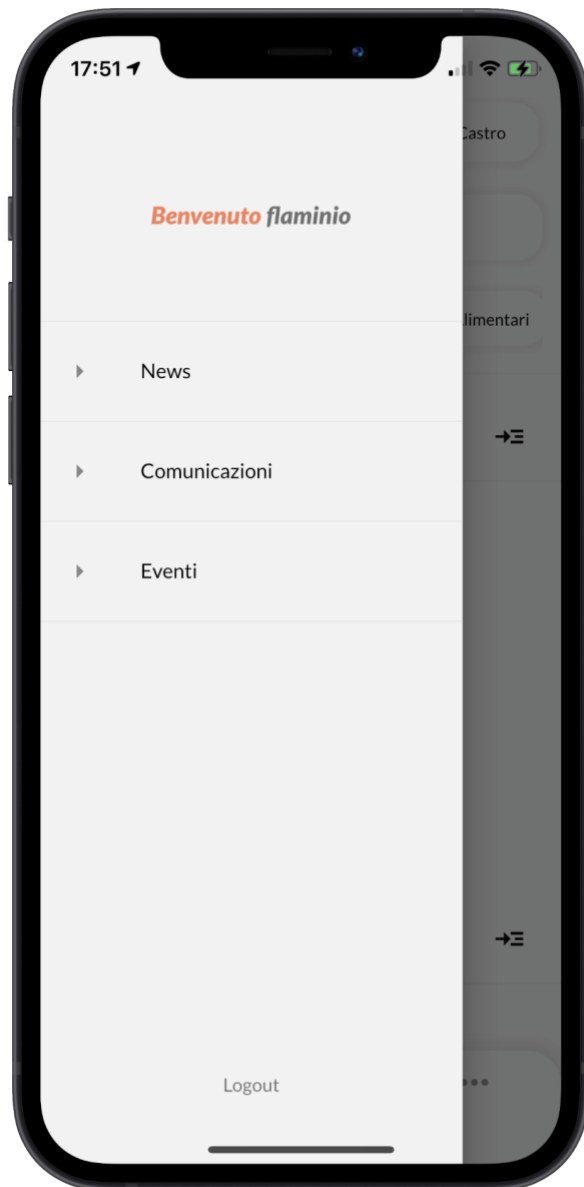
 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 21 di <b>70</b>	
--	--	---	---

L'ultima sezione della nostra barra di navigazione ci riconduce alla schermata delle impostazioni, dove in questo caso è presente solo una breve descrizione dell'applicazione.



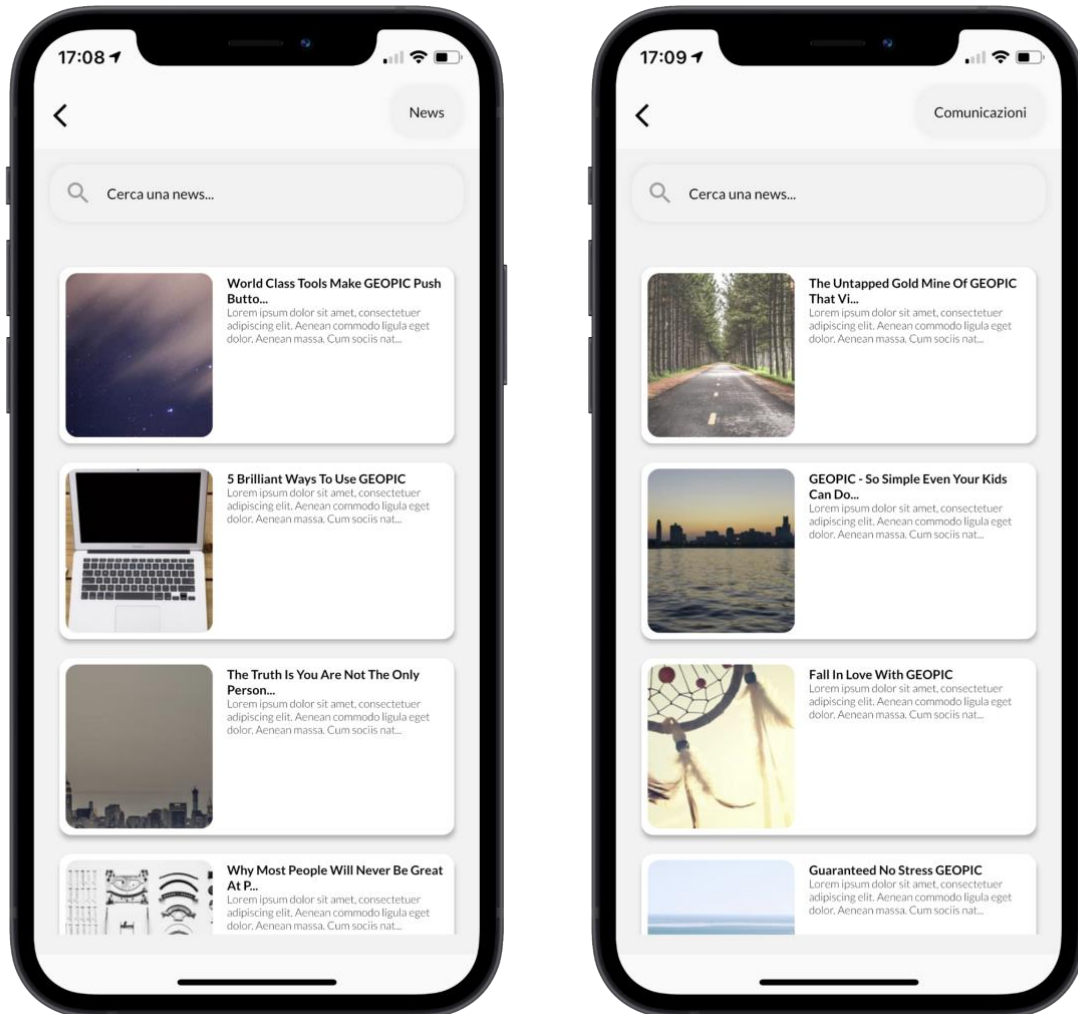
 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 22 di <b>70</b>	
--	--	---	---

Un altro componente che non abbiamo trattato è il drawer dell'app che ci permette di accedere alle varie sezioni : news, eventi e comunicazioni.



Il drawer mostra il nome dell'utente che è attualmente loggato, sotto di esso sono presenti le opzioni prima elencate.

Cliccando su una di queste opzioni ci troviamo in una pagina simile a quella per le categorie , in questo caso le “Card” per le notizie sono diverse e studiate appositamente per questo scopo.



Come già visto in precedenza per la pagina delle categorie e macro categorie , questa sezione è molto simile vengono solo cambiate le “Card” che mostrano le varie notizie.



 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 24 di <b>70</b>	
--	--	---	---

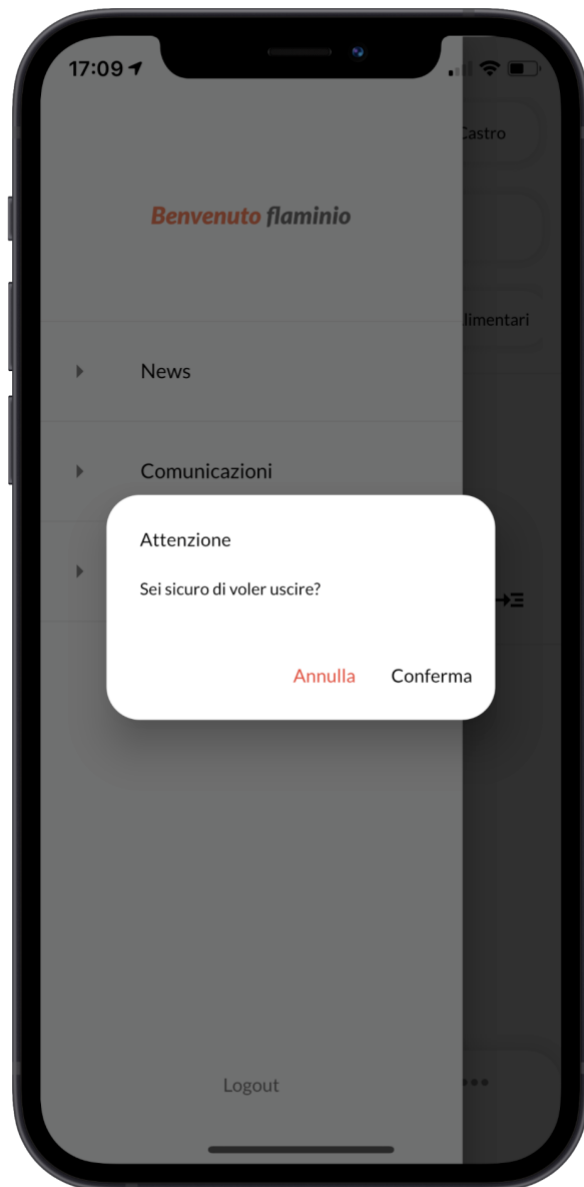
Cliccando su una notizia viene aperta la pagina di dettaglio per la visualizzazione della notizia.



Come è possibile vedere questa pagina presenta tre componenti fondamentali, l'immagine della Notizia-Comunicazione-Evento, il titolo e la descrizione.



Dal drawer l'ultima azione possibile è il logout dell'utente tramite l'apposito pulsante che chiede anche la conferma per approvare l'azione.



Dopo aver eseguito il logout i dati dell'utente verranno cancellati dall'applicazione e verrà richiesto di effettuare nuovamente il login per poter accedervi nuovamente. Dopo aver visto tutte le pagine e componentistiche di cui è composta l'app procediamo nella prossima sezione per vedere come viene gestito ed effettuato il routing quindi la navigazione tra una pagina ed un'altra.

## 4.5 Il Routing

Lo spostamento da una schermata ad un'altra avviene tramite il routing, processo che definisce su quale pagina, con quale animazione e con quali dati l'applicazione deve passare da una schermata A ad una schermata B.

La gestione del routing se non effettuata nel modo corretto può provocare errori di transizione ad una pagina.

In questo caso la gestione del routing è divisa in due file, il primo si chiama "routing\_constants" dove vengono definite tutte le possibili path dell'applicazione.

Il secondo file è il router che contiene un metodo per la generazione di una route, quindi effettua la vera e propria transizione da una pagina.

Questo metodo riceve in input una RouteSettings, classe fornita dal framework che ha due attributi fondamentali, name e arguments.

L'attributo name è definito da uno delle costanti dentro il file routing\_constants e invece l'attributo arguments contiene i vari parametri che servono per la transizione a quella rotta.

Questo metodo quindi, tramite uno switch basato sul nome della RouteSettings fornita sceglie la transizione da effettuare.

Ecco un pezzo di codice per capire meglio il funzionamento.

```
Route<dynamic> generateRoute(RouteSettings settings) {  
  switch (settings.name) {  
    case tabControllerPageRoute:  
      return MaterialPageRoute(builder: (context) => TabBarController());  
    case homePageRoute:  
      return MaterialPageRoute(builder: (context) => HomePage());  
    case settingsPageRoute:  
      return MaterialPageRoute(builder: (context) => SettingsPage());  
    .  
    .  
    .  
  }  
}
```

## 4.6 Gestione dello stato dell'app

La gestione dello stato nell'applicazione è gestita tramite un pattern chiamato:

BLOC (Business Logic Components)

Questo pattern ha lo scopo di dividere la parte di UI dalla parte di Business Logic, per riuscire a fare ciò il pattern si basa su stream di dati che forniscono le informazioni da mostrare tramite i widget e permette di gestire anche le callback che sono effettuate da un widget a seguito di un'azione da parte dell'utente. L'unico componente che comunica con il backend è la repository, nel nostro caso sono presenti tre repository per la gestione dell'autenticazione, dei dati delle strutture ecc. e della sezione new, eventi e comunicazioni.

Quindi la nostra struttura avrà questo flusso:



Ricordiamo che un widget o un block non effettuano direttamente nessuna chiamata al backend ma devono fare richiesta ad una repository.

Usando questo pattern un widget non deve eseguire nessuna Business Logic ma può solo emettere un nuovo evento nel bloc.

## 4.6.1 BLOC Pattern

Vediamo come è costituito un BLOC:

Un BLOC si suddivide in Bloc State, Bloc Event , Event Status e il BLOC vero e proprio. Tramite la parte di Bloc State si definiscono i vari stati del widget, prendiamo in esempio la home page:

Viene definito uno stato di base abstract:

```

abstract class HomeState extends Equatable {
  @override
  List<Object> get props => [];
}
  
```

Successivamente vengono definiti i vari stati della home page, come: stato iniziale, in caricamento, caricato e stato di errore.

```

class HomeInitState extends HomeState {}
  
```

```

class HomeLoading extends HomeState {}
  
```

Vengono definiti anche tutti gli attributi che uno stato deve contenere in quella fase, questi sono i dati che popoleranno la UI.

```

class HomeLoaded extends HomeState {
  final List<Section> sections;
  final String location;
  final List<Category> categories;
  HomeLoaded({this.sections, this.location, this.categories});
}
  
```

```
}
```

```
class HomeError extends HomeState {
  final error;
  HomeError({this.error});
}
```

Avendo definito tutti gli stati che la nostra pagina può assumere, lato ui sapremo sempre cosa accade!

Procediamo con l'analisi del Bloc Event e dell'Event Status.

Un evento è l'azione che viene emessa dalla UI o da un altro BLOC per effettuare un cambiamento di stato.

Lo Status definisce l'azione vera e propria da compiere .

```
enum HomeStatus { Fetch, Init, FetchFromInput}
```

L'evento contiene tutti i dati necessari per effettuare un azione.

```
class HomeEvent{
  final String location;
  final HomeStatus status;
  final String structureName;
  HomeEvent({@required this.status,@required this.location,this.structureName});
}
```

Quindi un evento è una classe contenente tutti i dati necessari per aggiornare lo stato del nostro widget.

Adesso analizziamo il vero e proprio bloc.

```
class HomeBloc extends Bloc<HomeEvent,HomeState > {...}
```

Come possiamo vedere il nostro bloc estende la classe BLOC (libreria esterna che semplifica la gestione degli stream e del flusso di dati).

Dentro al bloc sono contenuti vari attributi per il suo funzionamento come le repository per comunicare con il backend o variabili per la gestione degli stati.

Esempio:

```
final MainRepository mainRepository;
final LocationAppCubit locationAppCubit;
List<Section> sections;
List<Category> categories;
String location;
Completer locationLoaded =new Completer();
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 29 di <b>70</b>	
--	--	---	---

E' presente un costruttore che riceve come parametri di solito le repository o altri variabili necessarie per altre operazioni.

```
HomeBloc({this.mainRepository, this.locationAppCubit}) : super(HomeInitState()) {...}
```

Il vero core di tutta la classe è l'ovverdie del metodo mapEventToState della classe estesa BLOC.

```
@override
```

```
Stream<HomeState> mapEventToState(HomeEvent event) async* {
  switch (event.status) {
    case HomeStatus.Init:
      yield HomeLoading();
      ...
      yield HomeLoaded(sections: sections, location: location, categories:
categories);
      break;
    case HomeStatus.Fetch:
      try{
        yield HomeLoading();
        ...
        yield HomeLoaded(
          sections: sections, location: location, categories: categories);
      }catch(err){
        yield HomeError(error: err);
      }
      break;
    case HomeStatus.FetchFromInput:
      yield HomeLoading();
      ...
      yield HomeLoaded(sections: sections, location: location, categories:
categories);
      break;
  }
}
```

Questo metodo converte un evento fornito in input in una sequenza di istruzioni, ogni evento passa per la fase di Loading dove lato UI viene gestito tramite un qualche tipo di caricamento per poi concludersi in una fase di Loaded dove vengono mostrati i dati o in uno stato di errore nel caso qualcosa non andasse a buon fine.

La parte di gestione della business logic si conclude qui, ovviamente per ogni metodo vengono eseguite chiamate al backend (analizzate in seguito) e azioni interne, che non affronteremo poiché ridondanti.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 30 di <b>70</b>	
--	--	---	---

## 4.6.2 Gestione dello Stato di un BLOC

Lato UI il pattern viene implementato nel seguente modo, prendiamo in esempio nuovamente la home page:

@override

```
Widget build(BuildContext context) {
  return BlocBuilder<HomeBloc, HomeState>(builder: (_, HomeState state) {
    if (state is HomeLoaded) {
      String _location = state.location;
      List<Category> _categories = state.categories;
      List<Section> _sections = state.sections;
      return _buildHomeLoaded(_location, _categories, _sections);
    } else if (state is HomeError) {
      return Center(
        child: Text('Ops qualcosa è andato storto...'),
        style: Theme.of(context).textTheme.headline5,
      );
    } else {
      return Center(
        child: CircularProgressIndicator(),
      );
    }
  });
}
```

Come è possibile vedere il componente BlocBuilder fornisce la gestione di un bloc, infatti ad ogni build dell'applicazione (quindi ad ogni refresh) viene fatto un controllo su quale UI renderizzare in base allo stato del bloc, ogni stato contiene, come visto prima, degli attributi che verranno forniti al widget per popolarlo.

Un'azione lato ui viene eseguita e manata al bloc tramite un'istruzione del tipo:

```
BlocProvider.of<HomeBloc>(context).add(new HomeEvent(status: HomeStatus.Fetch))
```

Come è possibile vedere un'istruzione viene “aggiunta” al bloc, questo perché gli eventi sono gestiti tramite uno stream quindi con il metodo “add” viene aggiunto l'evento tramite un “sink” allo stream degli eventi in ingresso.

### 4.6.3 Comunicazione tra BLOC

La comunicazione tra bloc è una parte fondamentale, infatti in molti casi è necessario avere una qualche informazione di un bloc in un altro. In questa applicazione l'App Bar con il suo bloc gestiscono tutta la parte di localizzazione quindi tutti gli altri bloc dovranno in qualche modo sottoscrivere a questa informazione.

Il processo di comunicazione tra bloc è relativamente semplice, un bloc può essere passato ad un altro bloc o tramite costruttore o tramite un metodo. Dopo che si ha un bloc dentro la classe si può effettuare una subscription ad esso e quindi mettersi in ascolto dello stream che fornisce, che è la stessa logica usata per la creazione della UI.

Prendiamo in considerazione un bloc chiamato: locationAppBloc che è contenuto all'interno del nostro bloc, possiamo metterci in ascolto dei cambiamenti su esso tramite un listener.

Ad ogni aggiornamento del bloc verrà fornito un nuovo state che potremo usare per gestire la nostra app come in questo esempio.

```
locationAppCubit.stream.listen((LocationAppState locationAppState) {  
  if (locationAppState.status == LocationAppStatus.Loaded) {  
    ...  
    add(new HomeEvent(status: HomeStatus.Fetch, location: location));  
  }  
});
```

Come è possibile notare in questo caso non stiamo usando un bloc ma un cubit che è esattamente un bloc ma come differenza ha che non usa uno stream di eventi in ingresso ma solo chiamate a funzioni, questo riduce la sua complessità per alcune funzionalità. In questo caso creare un bloc per la gestione della sola posizione risultava non necessario ed ho optato per questa alternativa.

Il codice interno è superfluo per questo esempio ma possiamo capire che il cambiamento di stato del bloc locationAppBloc può comportare un aggiornamento della nostra app in determinate circostanze.

Inoltre come già visto in precedenza possiamo immettere un evento nel bloc esterno con il metodo “add”.

E' inoltre possibile accedere direttamente ad una variabile dentro il bloc :

```
locationAppBloc.nomeVariabile
```

Questo metodo però non prende in considerazione lo stato dell bloc quindi non sappiamo se quel valore è aggiornato oppure no, o se è stato inizializzato quindi occorre una logica di supporto che ci renda certi che il dato che stiamo cercando sia presente.

## 4.7 Gestione di Eventi con Controller

La gestione di un evento di un BLOC potrebbe risultare complesso da eseguire all'interno del metodo di mapping, creare metodi dentro al bloc lo renderebbe troppo complesso da leggere e gestire. Per questo problema si usano dei "Controller" che sono delle classi helper che controllano per l'appunto una parte di logica.

Nell'applicazione sono presenti due controller, uno per la gestione della mappa e uno per la gestione della posizione.

Il controller per la gestione della mappa : **BrowseController** , presenta i metodi per muovere la camera della mappa così da permettere lo spostamento ad una nuova località, i metodi per la creazione dei markers nella mappa e per la gestione dei permessi della posizione del dispositivo (si basa sul controller della location).

Il secondo controller è quello per la gestione della posizione: **LocationController** dove ci sono i metodi per richiedere la posizione attuale, e la richiesta dei permessi lato dispositivo.

## 4.8 Comunicazione con il Backend tramite Repository

Come già introdotto prima la comunicazione con il backend avviene tramite delle classi Repository, queste classi contengono al loro interno un metodo per ogni tipo di chiamata che devono effettuare. Prendiamo in esempio la **mainRepository** dell'applicazione che esegue tutte le chiamate al backend per ricevere le strutture ,categorie, macrocategorie e markers da visualizzare nell'applicazione.

Come possiamo vedere questa classe contiene un solo attributo che è l'header per tutte le richieste e contiene ovviamente anche tutti i metodi che può eseguire.

```
class MainRepository{  
    //Defining the request headers  
    Map<String, String> headers = {'Content-Type': 'application/json'};  
  
    MainRepository();  
    ... i vari metodi  
}
```

Analizziamo come è strutturato un metodo:



 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 33 di <b>70</b>	
--	--	---	---

Una richiesta al backend vera è propria è svolta dai metodi `http.get` , `http.post` ecc...  
Questi metodi prendono come parametro l'url del backend che nell'applicazione è salvata all'interno dentro le variabili d'ambiente, più precisamente in un file `.env` presente nel progetto dove sono contenuti i dati sensibili.  
Tramite il metodo `env` della classe `DotEnv` è possibile accedere all'url del nostro backend tramite dizionario in : `BACKEND_URL` e concatenando ad esso l'endpoint del backend riusciamo a formare il nostro URL.

Il tipo `Future` indica che il dato di ritorno è di tipo asincrono, in Flutter un `Future` è un oggetto che deve essere ancora risolto (chiamata al backend che non ha dato ancora una risposta), quindi si deve effettuare un'operazione di “`await`” e gestire l'attesa del dato.

```

///Returns all the db structures
Future<List<Structure>> getStructures(Position userPosition) async {
  List<Structure> structures = [];
  try {
    final structuresRequest =
      await http.get(Uri.parse(DotEnv.env["BACKEND_URL"] + 'structure/findAll'));
    if (structuresRequest.statusCode == 200) {
      var parsedSection = json.decode(structuresRequest.body) as List;
      structures = parsedSection.map((section) =>
        Structure.fromMap(section,userPosition.latitude,userPosition.longitude)).toList();
      return structures;
    }
  } catch (err) {
    return structures;
  }

  return structures;
}

```

All'interno del metodo viene effettuato il controllo riguardo l'esito della richiesta tramite lo status code, convertendo successivamente la risposta del body di tipo JSON in una lista di oggetti definiti dai modelli.

Questo approccio viene sfruttato per la gestione di tutti i metodi della repository.

## 4.9 TESTING

Il testing lato frontend permette il controllo delle parte critiche del codice, ovviamente eseguire UnitTesting sui componenti UI non ha senso e sarebbe superfluo.

Il testing in flutter avviene tramite la libreria test fornita dal Framework in oltre viene usata la libreria test\_bloc che permette di testare la business-logic della nostra app.

Procediamo nel analizzare il testing effettuato per quest'applicazione:

La cartella “test” contiene tutti i file che ci interessano, osserviamo subito il file mock\_data che contiene una struttura di dati falsi che ci permetteranno di falsificare i dati così da verificare se i dati che richiediamo vengono elaborati come richiesto.

```
class MockData{
  static String description = 'testDescription';
  static String location = 'Roma';
  static Position positionRomeMock = new Position(latitude: 41.9027835,...);
  static PositionLocation positionLocation = new PositionLocation(...);
  static Map positionMap = {'latitude':positionRomeMock.latitude... };
  static Secretary secretary = new Secretary(id: 0, address: 'address'...);
  static Person referralPerson = new Person(id: 0, name: 'name',...);
  static Category category = new Category(id: 0, name: 'name', color: 'color');
  static Category categoryForSection = new Category(id: 0, name: 'sectionTest'...);
  static List<Category> categories = [category];
  static Structure structure = new Structure(...);
  static List<Structure> structures = [structure];
  static StructureSection structureSectionInit = new StructureSection(...);
  static StructureSection structureSection = new StructureSection(...);
  static List<StructureSection> structureSections = [structureSection];
  static List positionSuggestion = [{"comune":"Roma"}];
  static GeoPicMarker geoPicMarker = new GeoPicMarker(id: 0, name: 'name'...);
  static List<GeoPicMarker> geoPicMarkers = [geoPicMarker];
  static MarkerId markerId = new MarkerId('markerIdTest');
  static Marker marker = new Marker(markerId: markerId);
  static Set<Marker> markers = new Set.from([marker]);
}
```

Questi sono i dati “mockati”, termine usato per indicare dei dati falsi.

Per procedere con il testing una parte fondamentale era mockare anche i controller e le repository dei vari block, processo che permette di farci ritornare i dati che abbiamo mockato.

Dentro alla cartella “test” sono contenute le cartelle: “mock\_repository” e “mock\_controllers”, procediamo nel analizzare come si struttura una repository mockata.

```
class MockMainRepository implements ImplMainRepository {
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 35 di <b>70</b>	
--	--	---	---

```

@Override
Future<List<StructureSection>> findAdvancedStructuresByInputName(String structureName, body, Position
userPosition) {
    return Future.delayed(Duration(seconds: 2), () => MockData.structureSections);
}

@Override
Future<List<StructureSection>> findStructuresByInputName(String structureName, String location, Position
userPosition) {
    return Future.delayed(Duration(seconds: 2), () => MockData.structureSections);
}

@Override
Future<List<GeoPicMarker>> getAllGeoPicMarkers() {
    return Future.delayed(Duration(seconds: 2), () => MockData.geoPicMarkers);
}
.
.
.
}

```

Come prima cosa è necessario avere una classe abstract d'implementazione che definisce tutti i metodi della nostra repository (`ImplMainRepository`) e va implementata sia nella classe mock che nella repository originale.

Come è possibile vedere dal codice sopra, i metodi ritornano un Future (perché i metodi sono di tipo asincrono) e ritornano un attributo della classe MockData.

Con questa logica alla scrittura di tutte le repository di mock.

Per i controller il ragionamento è lo stesso, si crea una classe abstract d'implementazione e successivamente la si implementa sia nel controller originale che in quella di mock.

```

class MockLocationController implements ImplLocationController {

    Future<Position> getPermissionAndPosition() {
        return Future.delayed(Duration(seconds: 2), () => MockData.positionRomeMock);
    }
    ....
}

```

Procediamo con l'analisi del testing vero e proprio.

Sono stati testati i BLOC dei vari widget che definiscono la logica della nostra app, testando i vari eventi critici di ognuno di essi.

Prendiamo in esempio il testing del CategoryBloc:

```

void main() {

```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 36 di <b>70</b>	
--	--	---	---

```

group('CategoryBloc', () {

  MockMainRepository mockMainRepository;
  LocationAppCubit locationAppCubit;
  CategoryBloc categoryBloc;

  setUp() {
    EquatableConfig.stringify = true;
    mockMainRepository = new MockMainRepository();
    locationAppCubit = new LocationAppCubit(mainRepository: mockMainRepository, locationController: new
MockLocationController(), initialPositionLocation: MockData.positionLocation);
    categoryBloc = CategoryBloc(
      mainRepository: mockMainRepository,
      locationAppCubit: locationAppCubit);
  });
  ...
}

```

Come prima cosa viene creato un gruppo di testing in questo caso CategoryBloc, successivamente vengono definiti tutti le repository (mockate) e altri bloc usati.

Nella fase di setUp vengono inizializzati repository e bloc.  
Osserviamo ora il testing di una funzionalità del bloc:

```

blocTest<CategoryBloc, CategoryState>(
  'Test Category Init',
  build: () => categoryBloc,
  act: (bloc) => {
    bloc.add(new CategoryEvent(status: CategoryStatus.Init, location: 'Roma', category:
MockData.categoryForSection)),
  },
  expect: () => [
    CategoryLoading(),
    CategoryLoaded(section: MockData.structureSection),
  ],
);

```

Come possiamo vedere tramite il parametro build definiamo il bloc, tramite il parametro act possiamo eseguire una serie di eventi, in questo caso aggiungiamo un nuovo evento al bloc e successivamente tramite il parametro expect definiamo quali stati sono stati emessi dal primo emesso (primo nella lista), all ultimo emesso (ultimo nella lista).

Con questa logica sono stati analizzati tutti i casi critici dell'applicazione contenuti nel bloc che non affronteremo perché di facile interpretazione.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 37 di <b>70</b>	
--	--	---	---

## 5 PROGETTAZIONE BACK-END

Dal punto di vista back-end, si è scelto di utilizzare **Spring Boot**, un framework per lo sviluppo di applicazioni web basate su codice Java che offre un ulteriore livello di astrazione rispetto a Spring Framework, di più complicato utilizzo e adatto a progetti aziendali più elaborati.

Spring Boot ha numerosi vantaggi tra cui:

- Incorporare direttamente applicazioni web server/container come Apache Tomcat, per cui non è necessario l'uso di file WAR (Web Application Archive);
- Configurazione di **Maven** semplificata grazie ai POM "Starter" (Project Object Models);
- Caratteristiche non funzionali come metriche o configurazioni esterne automatiche.

La memorizzazione delle informazioni è affidata ad un Database **SQL**, in particolare MySQL, che viene gestito a livello di persistenza dall'applicazione tramite l'uso di **JPA** (Java Persistence API) e della sua più famosa implementazione framework **Hibernate**.

I dati contenuti vengono monitorati tramite l'uso di Datagrip un visual tool che permette di verificare che i contenuti del database siano coerenti con le operazioni eseguite.

La security invece viene gestita mediante Spring Security.

**Spring Security** è un framework di autenticazione e controllo degli accessi potente e altamente personalizzabile, si concentra sulla fornitura di autenticazione e autorizzazione alle applicazioni Java. Come tutti i progetti Spring, il vero potere di Spring Security si trova nella facilità con cui può essere esteso per soddisfare i requisiti personalizzati.

**SpringBoot** è stato affiancato a **Maven**, progetto open-source che permette di organizzare al meglio il progetto Java, poiché ha la caratteristica di essere modulare, cioè in fase di progettazione si scelgono in base alle funzionalità che deve avere l'applicativo, le librerie (chiamate **dependencies**) da aggiungere al progetto in un file denominato pom.xml (Project Object Model).

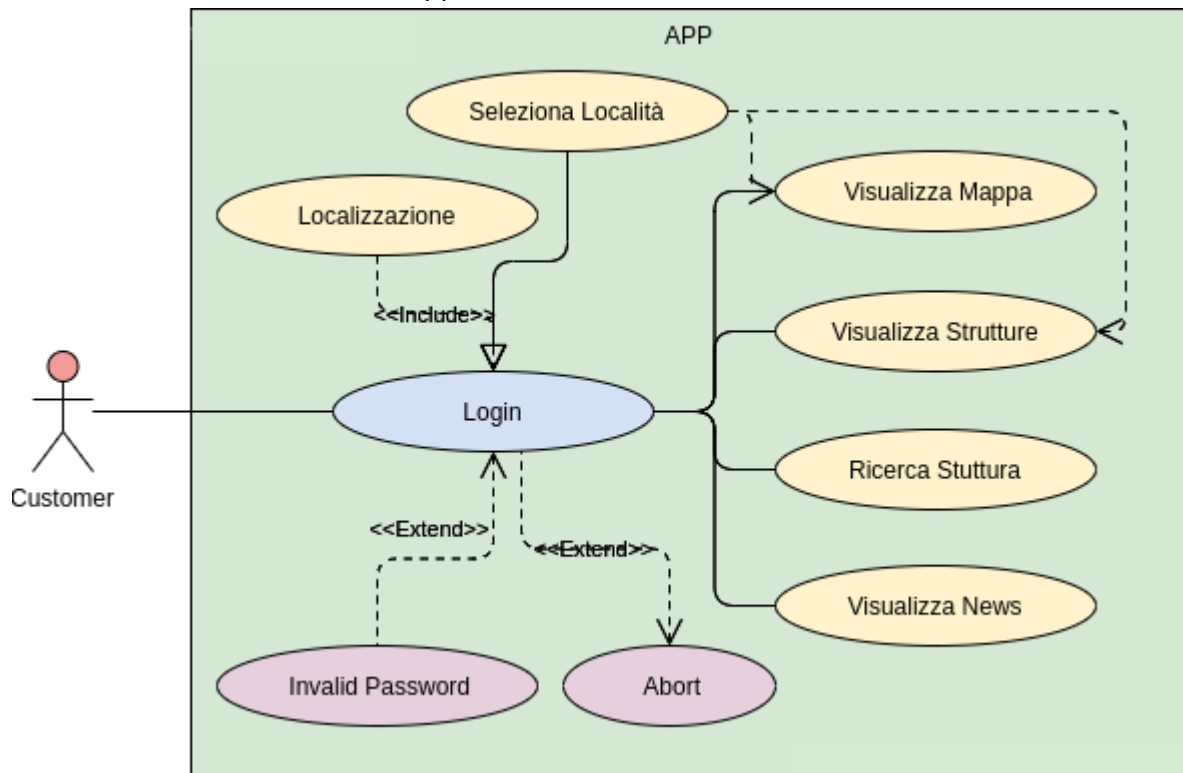
Di fondamentale importanza sono le **Annotations** di SpringBoot, indicate nelle classi Java con la **@** che precede il loro nome e che specificano informazioni aggiuntive utili a definire valori, comportamenti, proprietà delle classi o degli attributi.

In quanto framework ORM (Object / Relational Mapping), **Hibernate** si occupa della persistenza dei dati in quanto si applica ai database relazionali (tramite **JPA**); **Hibernate** è ben noto per la sua eccellente stabilità e qualità, dimostrata dall'accettazione e dall'utilizzo da parte di decine di migliaia di sviluppatori Java.

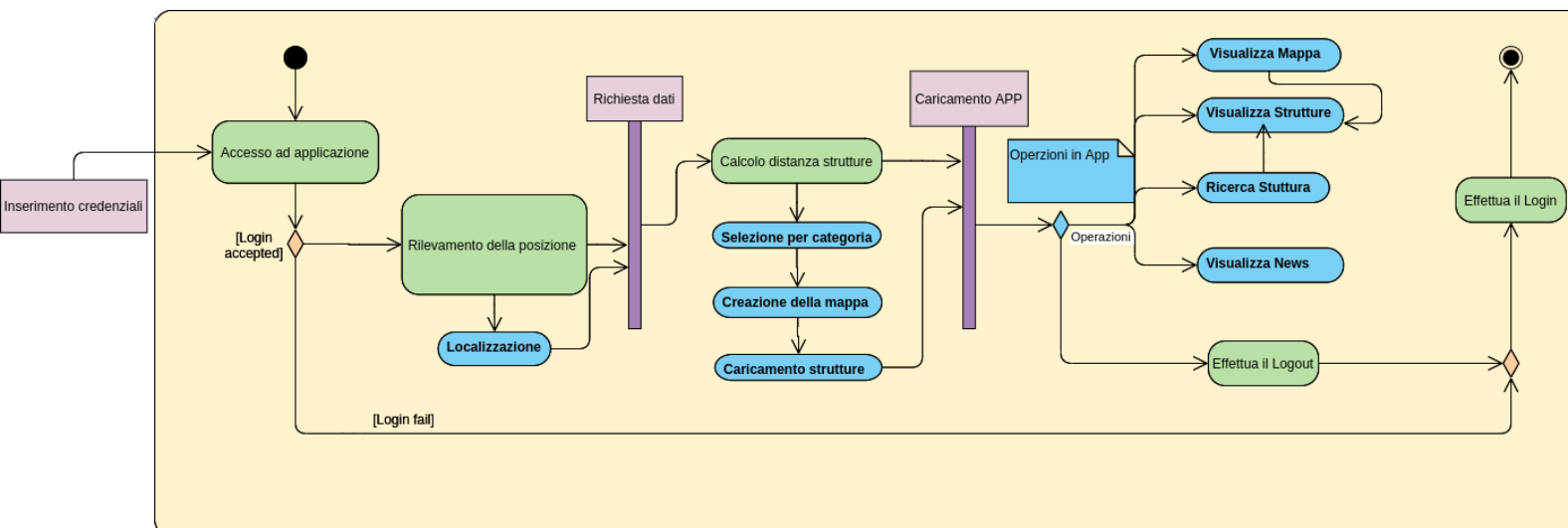
## 5.1 ACTIVITY DIAGRAM

Gli Use Cases catturano il comportamento del sistema, illustrando quali sono i suoi requisiti funzionali.

Lo use case realizzato in figura permette di visualizzare tutte le funzionalità a cui può accedere l'utente scaricando l'applicazione mobile.



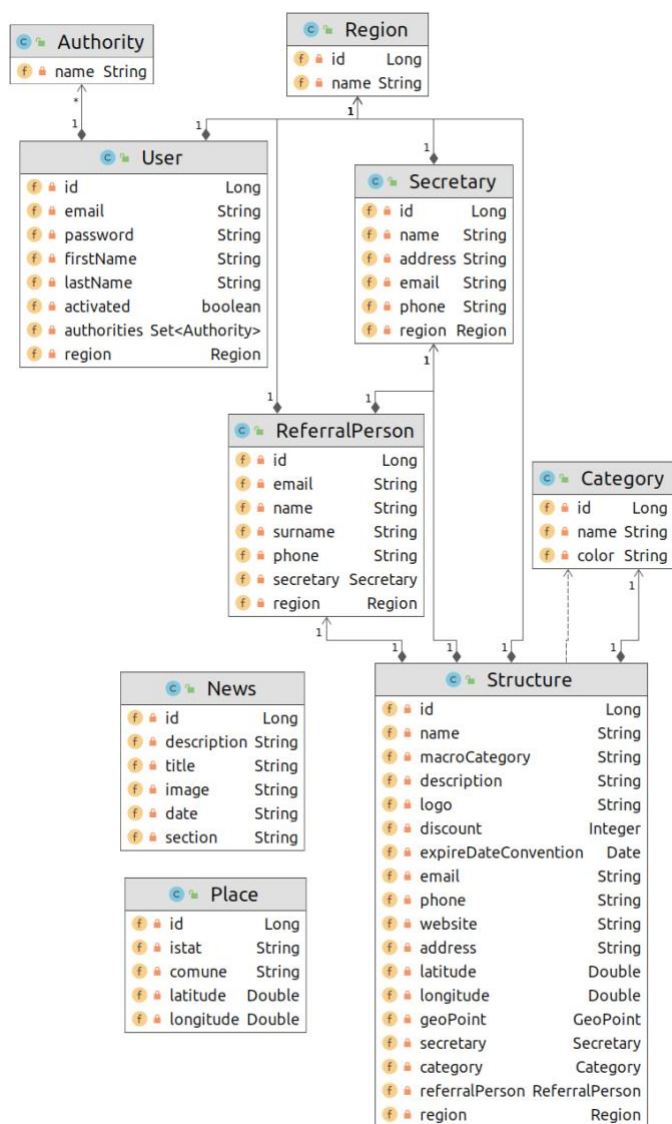
Come si può osservare, un cliente si autentica sul sistema, cioè effettua il login e dopo di che può eseguire una serie di funzionalità.





## 5.2 DEFINIZIONE DELLE ENTITÀ DI SISTEMA


Il Class Diagram mostra le classi che compongono il software e come esse si relazionano l'una con l'altra.



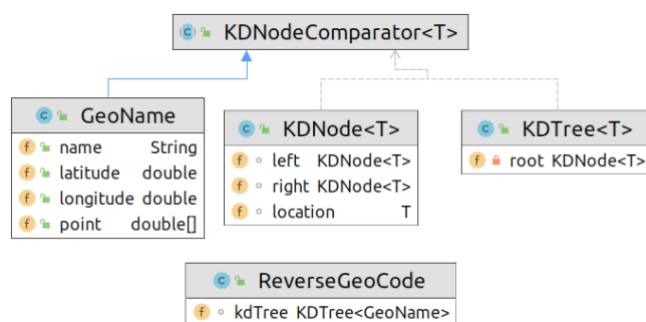
Il diagramma in figura mostra le classi che costituiscono i “*model*” del progetto. Per questioni di visibilità grafica non sono stati riportati i costruttori e i metodi all’interno delle singole classi.

Le classi rappresentano le Entità di cui è composto il Database e ne riprendono le relazioni, in particolare:

- La classe **Authority** contiene il nome dei possibili ruoli assumibili dagli utenti
- La classe **UserAuthority** associa ad ogni **User** i ruoli corrispondenti ruoli

- La classe **User** contiene le informazioni principali necessarie per gli utenti della piattaforma, ogni User è associato ad una **Region**.
- La classe **Region** contiene nome e id di tutte le regioni d'Italia, serve per dividere settorialmente utenti e strutture.
- La classe **Structure** è l'entità principale, contiene infatti tutti dati di una struttura convenzionata da esporre sulla piattaforma, vengono divise per **Category**.  
Ogni struttura viene gestita da una **ReferralPerson** associata ad una **Secretary** e localizzata in una **Region**, viene memorizzato anche il **GeoPoint**.
- La classe **Category** specifica il nome ed il colore da usare nei  personalizzati della mappa
- La classe **ReferralPerson** definisce le generalità del personale gestore delle convenzioni
- La classe **Secretary** contiene le informazioni delle sedi di controllo in cui lavorano le **ReferralPerson**

All'interno del progetto possono essere individuati altri tre gruppi di entity:



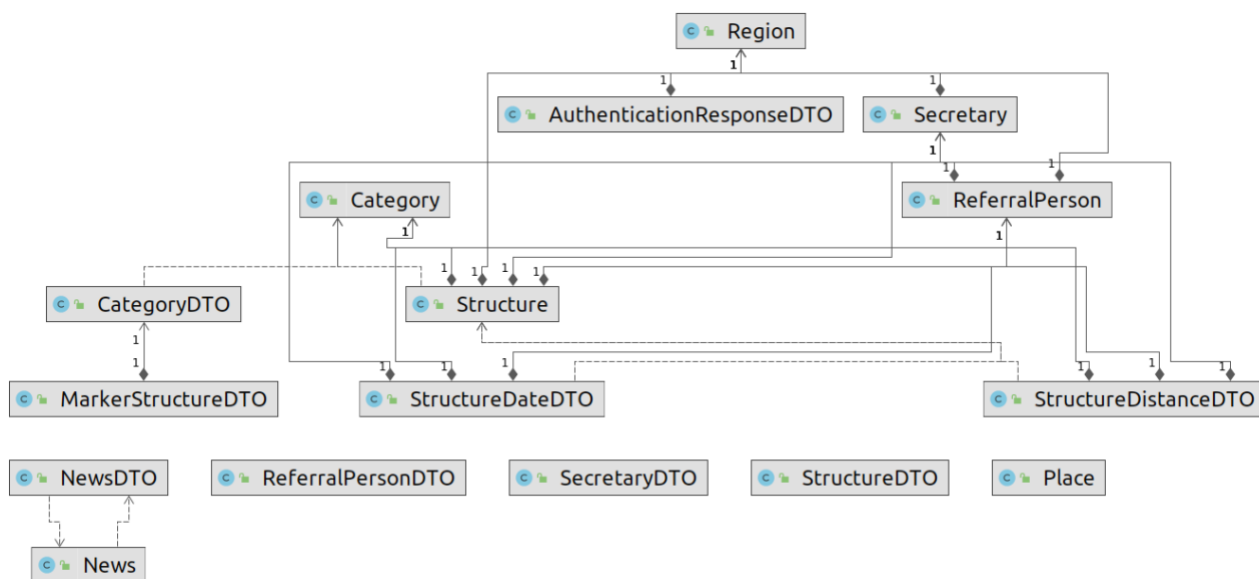
Per effettuare un **Reverse geocoding** il più veloce ed efficiente possibile, è stato creato un albero binario, che effettua memorizzazione e ricerca di coordinate geografiche.

Analizzando meglio le classi:

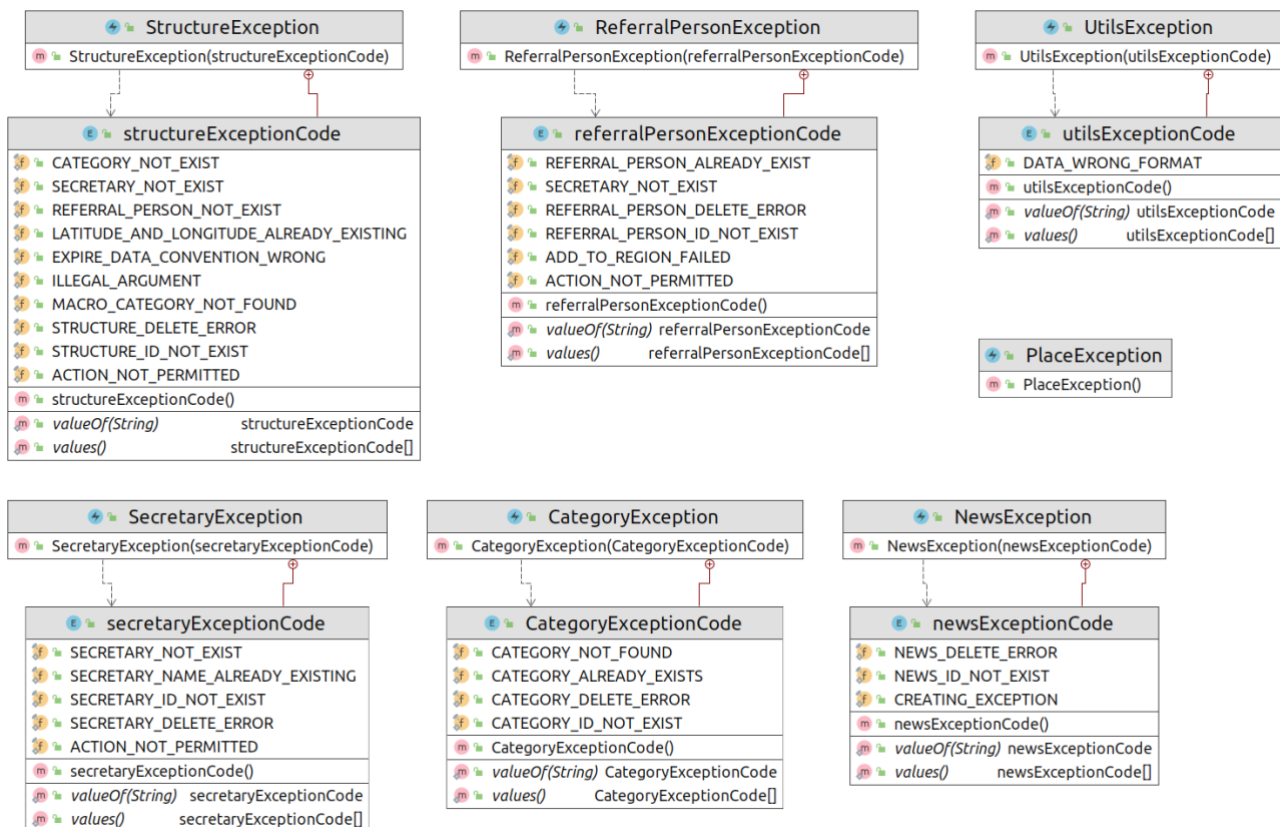
- La classe **Geoname** contiene i parametri necessari all'identificazione della città italiana corrispondente.
- La classe **KDNode** costituisce la struttura della foglia dell'albero, che verrà creato tramite **KDTree**
- La classe **ReverseGeoCode** effettua le operazioni di ricerca all'interno dell'albero.
- La classe **KDNodeComparator** ridefinisce i metodi comparator.

Di seguito vengono riassunti i principali **DTO** utilizzati all'interno del progetto





Le **exception** di ogni **Entity** vengono gestite singolarmente:

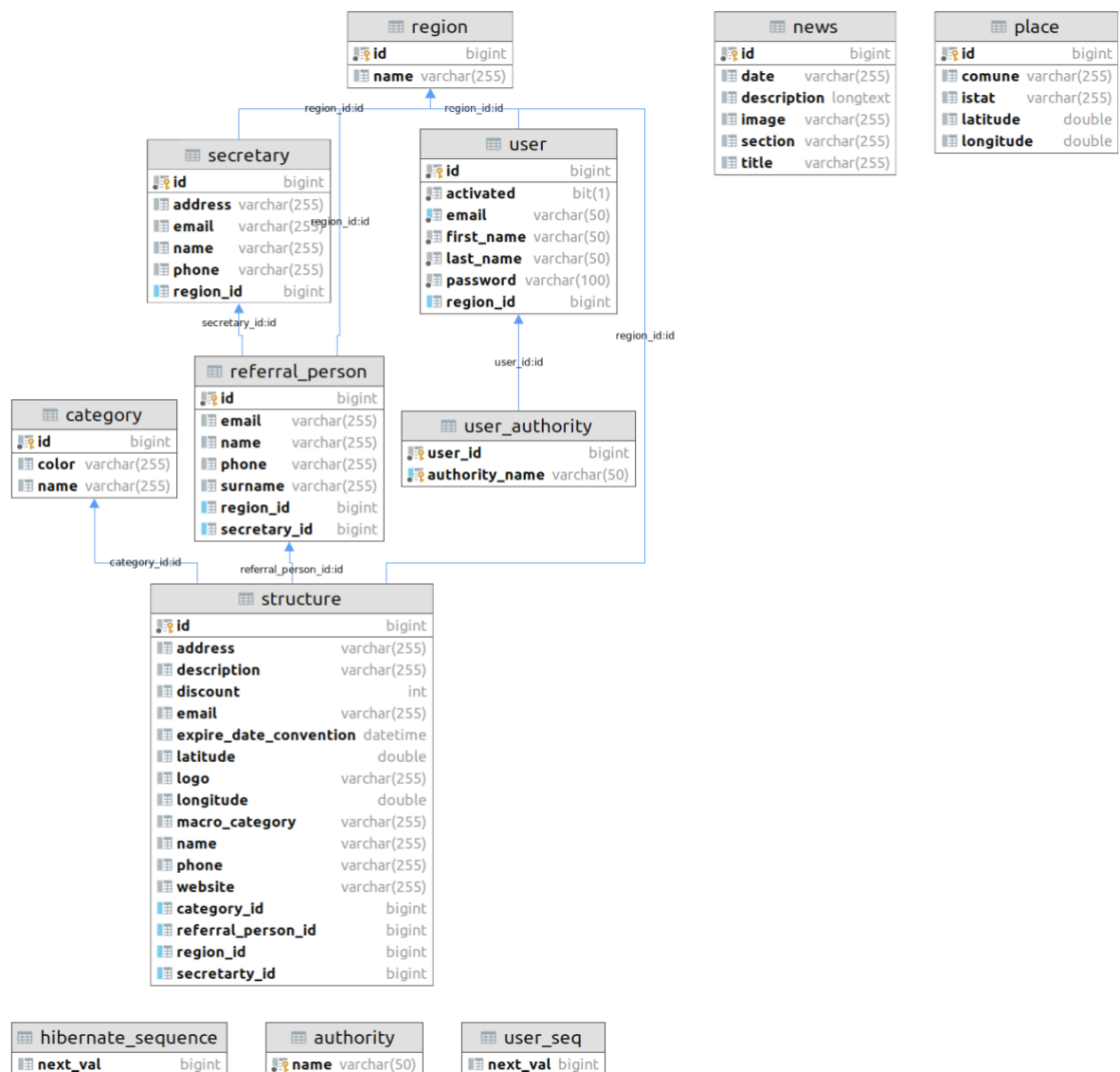


## 5.2.1 DESIGN DELLA BASE DI DATI

Il Database, come spiegato in precedenza, è di tipo relazionale.

E' strutturato sulla base delle classi presenti all'interno del package "model", contenente le Entity di sistema.

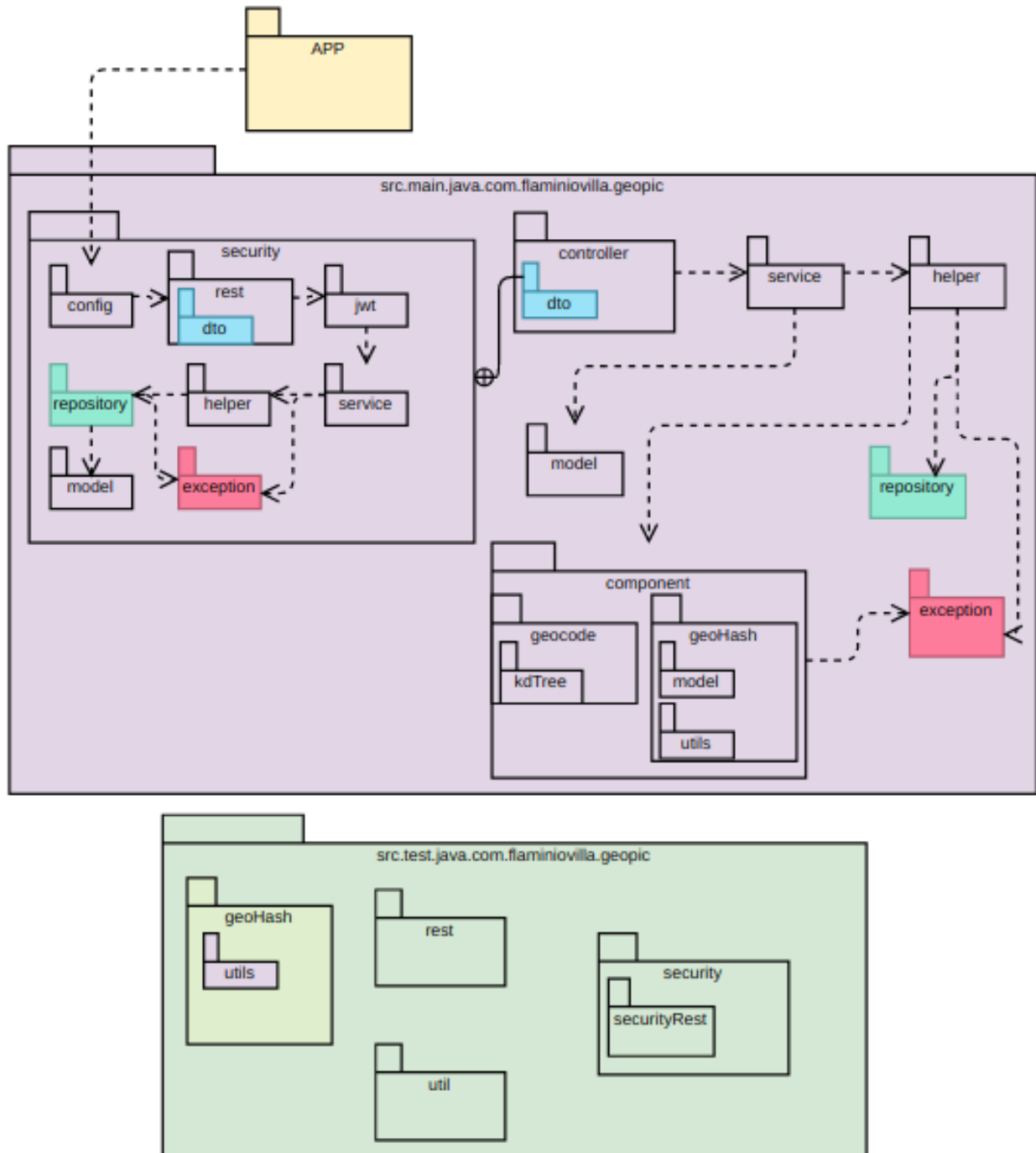
Il diagramma Entità-Relazioni del Database è il seguente:



**hibernate\_sequence** e **user\_seq** servono a memorizzare l'id utilizzato, utile per avere una successione esatta per la generazione degli ID.

## 5.3 ANALISI SOFTWARE

Il progetto è suddiviso in **32** package mostrati in figura



 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 44 di <b>70</b>	
--	--	---	---

### Model

Il package “model” contiene le classi Entity che rappresentano le tabelle del Database. Ogni classe è dichiarata con l’annotazione **@Entity** facente parte di Jpa Hibernate. L’annotazione **@Data** genera getter per tutti i campi, un utile metodo toString e hashCode. Genererà anche setter per tutti i campi non finali, oltre ai costruttori che verranno specificati tramite **@AllArgsConstructor** e **@NoArgsConstructor**. L’annotazione **@Builder** è un meccanismo utile per utilizzare il **pattern Builder** senza scrivere codice ridondante.

### Security

La security verrà analizzata nel dettaglio nel punto **4.5**.

### Controller

Il package “controller ” contiene al suo interno il package “dto” contenente le classi che definiscono gli standard per i passaggi di dati tra il back-end e il front-end, utilizzando come tipo il **JSON**. I dati successivamente verranno trasferiti verso i sottosistemi dell'applicazione software.

Sono divisi in *nameController* e *nameCRUDController*, come si può intuire dal nome avranno tipi di accesso differenti, i primi si occuperanno solo di fornire dati, i secondi effettueranno operazioni di interfaccia con il database.

le classi Controller, sono annotate con **@RestController**. I controller fanno uso dei service (richiamati con **@Autowired**) per prelevare i dati secondo le logiche di business, mappando le richieste HTTP chiamate dalla view ai microservizi sviluppati e rispondendo utilizzando il medesimo protocollo e lo standard **JSON**.

Le richieste GET vengono mappate usando **@GetMapping()** con tra parentesi indicato il path dell’endpoint. Le richieste POST vengono mappate usando **@PostMapping()**. Le richieste POST vengono mappate usando **@PostMapping()**. Le richieste DELETE vengono mappate usando **@DeleteMapping()**.

### Service

Il package “service ” contiene classi che svolgono tutti i controlli relativi ai parametri ricevuti dai controlli, e tramite **@Autowired** richiamano gli helper.

### Helper

Il package “helper ” contiene la **business logic** di progetto, si interfaccia tramite **@Autowired** verso le repository , definendo le operazioni sul database.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 45 di <b>70</b>	
---	--	---	---

### Repository

Il package “repository” contiene le interfacce che estendono *JpaRepository*<*T*, *ID*> e sono annotate con @Service.

Esse si occupano di eseguire operazioni CRUD sulle Entity nel Database.

L’interfaccia *JpaRepository* offre alcuni metodi base come findById() e findAll(), invece se si ha bisogno di eseguire una query SQL nativa si può utilizzare l’annotazione @Query.

### Exceptions

Il package “exceptions” contiene le classi che estendono la classe ***RuntimeException*** (lanciate da Java).

Queste classi rappresentano eccezioni personalizzate create per mandare dettagli più precisi alla view (front-end), sia per quanto riguarda lo status HTTP dell’errore generato, sia per quanto riguarda un messaggio che descriva brevemente il problema.

### Component

Il package “component” contiene I package “**geoCode**” e “**geoHash**”, oltre ad alcune classi di utils, contiene anche la classe Scraper (svolge il ruolo di web scraper).

Una parte molto importante dell’applicativo è relativa a la **geolocalizzazione** del dispositivo.

Per effettuare un Reverse geocoding il più veloce ed efficiente possibile, è stato creato un albero binario, che effettua memorizzazione e ricerca di coordinate geografiche, questo meccanismo è sviluppato all’interno del package “**geoCode**”.

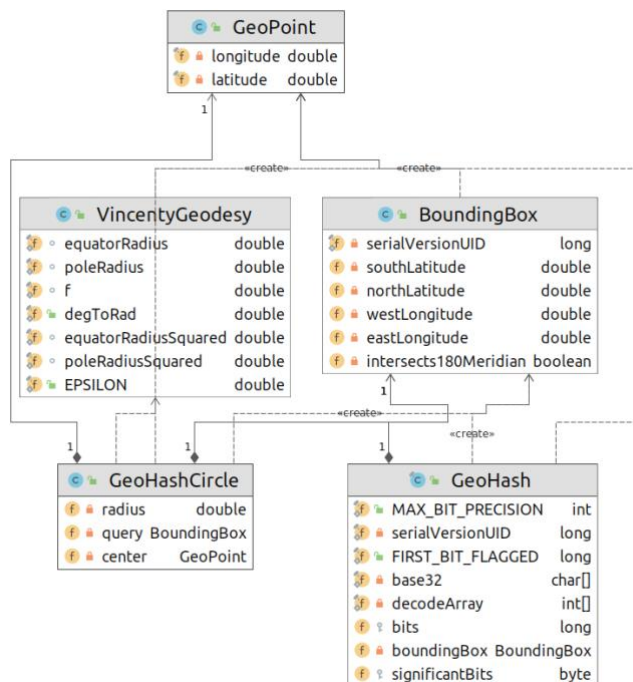
Un’altro punto focale è la funzione che permette di mostrare all’utente le strutture presenti vicine a la sua posizione.

Non è per niente banale la sua implementazione, infatti effettuare un semplice controllo di distanza tra la posizione gps del dispositivo e tutte le strutture del DB sarebbe inefficiente e lento.

E’ stato quindi implementato una funzione che sfrutta l’algoritmo di Vincenty's geodesy per effettuare tale misurazione.

## 5.4 ANALISI DELL'ALGORITMO DI RICERCA DELLE STRUTTURE

Il blocco contenuto all'interno del package "component" -> "geoCode" principale è la componente che ci permette di identificare le strutture presenti in una determinata area geografica.



Per dare una breve descrizione delle classi sopra riportate :

- La classe **GeoPoint** è un oggetto che contiene latitudine e longitudine di una coordinata geografica.
- La classe **GeoHash** effettua un encoding binario che rappresenta l'unione dei **GeoPoint**.
- La classe **BoundingBox** definisce un contenitore quadrato che ha come vertici le coordinate di latitudine e longitudine
- La classe **GeoHashCircle** viene creata da un **GeoPoint** ed un raggio crea una approssimazione di un cerchio in quadrato, che corrisponderà poi ad una **BoundingBox**.
- La classe **GeoHashCircle** Incapsula l'algoritmo di Vincenty's geodesy.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 47 di <b>70</b>	
--	--	---	---

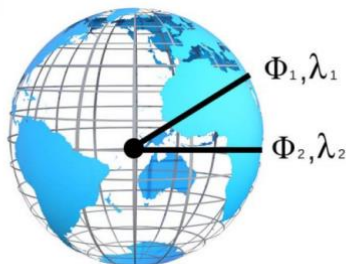
### 5.4.1 APPROFONDIMENTO ALGORITMO DI VINCENTY GEODESY

La formula di Vincenty è un metodo iterativo per calcolare la distanza tra due punti sulla superficie di uno sferoide, sviluppati da Thaddeus Vincenty (1975a). Si basa sul presupposto che la figura della Terra sia uno sferoide oblato, e quindi sono più accurati dei metodi che assumono una Terra sferica, come la distanza del cerchio maggiore.

- La latitudine deve essere compresa tra  $0^\circ$  e  $\pm 90^\circ$  e le latitudini sud sono negative (ad esempio,  $-35^\circ 55' 56,12''$ );
- La longitudine deve essere compresa tra  $0^\circ$  e  $\pm 180^\circ$  e le longitudini ovest sono negative (ad esempio,  $-148^\circ 56' 25,12''$ );

Le formule sono processi iterativi, il che significa che viene calcolata una sequenza di equazioni in cui l'output viene reinserito nella stessa sequenza di equazioni. L'obiettivo è ridurre al minimo il valore di output dopo un determinato numero di iterazioni.

$$\sin \sigma = \sqrt{(\cos U_2 \sin \lambda)^2 + (\cos U_1 \sin U_2 - \sin U_1 \cos U_2 \cos \lambda)^2}$$



Dove  $d$  è la distanza tra due punti con longitudine e latitudine  $(\lambda, \Phi)$  e  $a$  è il raggio della terra.

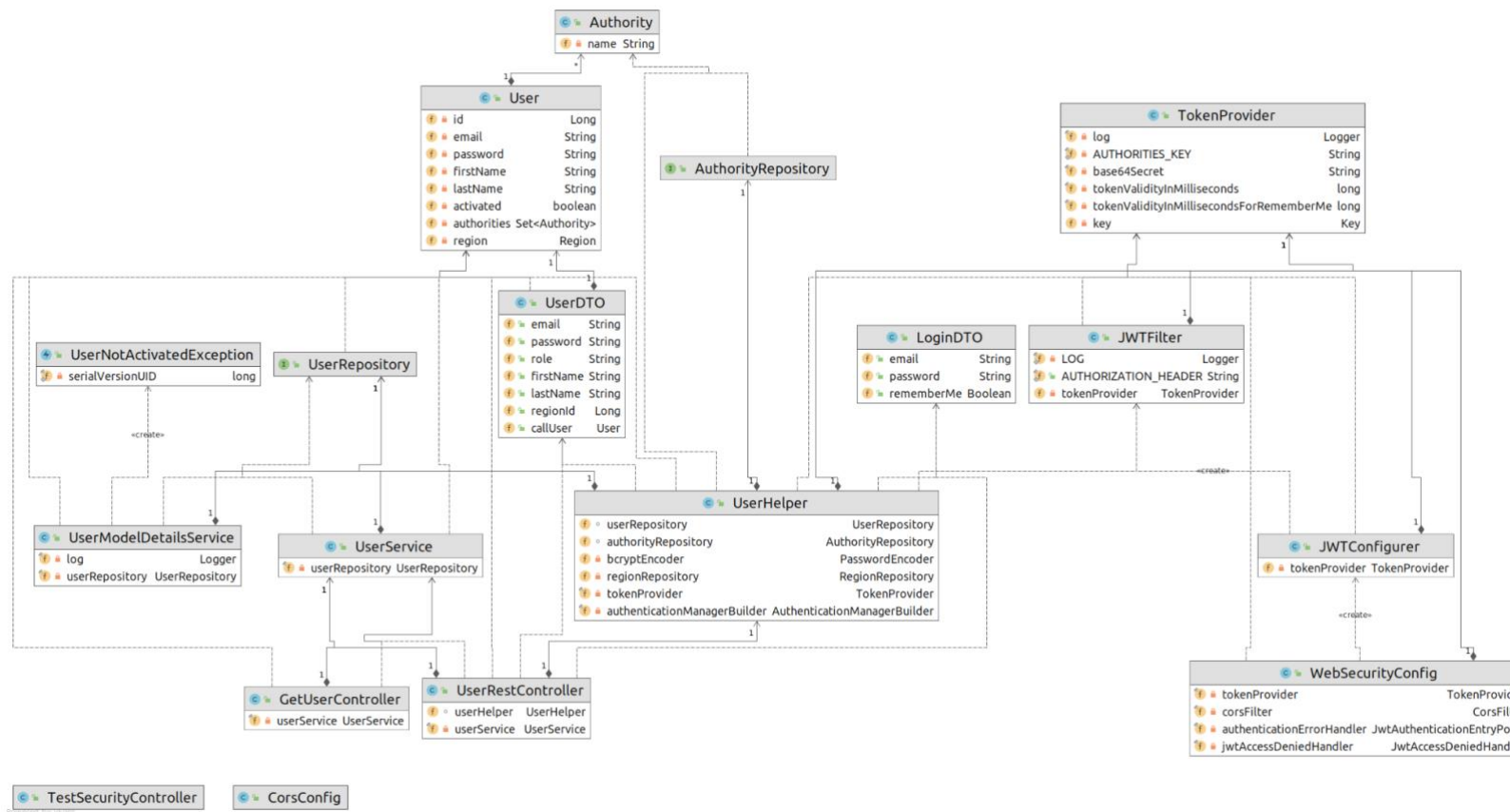
$$u^2 = \cos^2 \alpha \frac{a^2 - b^2}{b^2}$$

Le formule di Vincenty sono indicate come accurate entro circa lo **0,3%**, il che è ottimo.



## 5.5 SECURITY

Il package “security” è strutturato come un micro servizio di login indipendente, infatti gestisce tutta la configurazione per l’accesso verso la piattaforma, con l’annotazione **@EnableWebSecurity**.



Spring Security si trova sul classpath, **Spring Boot** protegge automaticamente tutti gli endpoint HTTP con l'autenticazione "di base". Analizziamo il workflow del login :

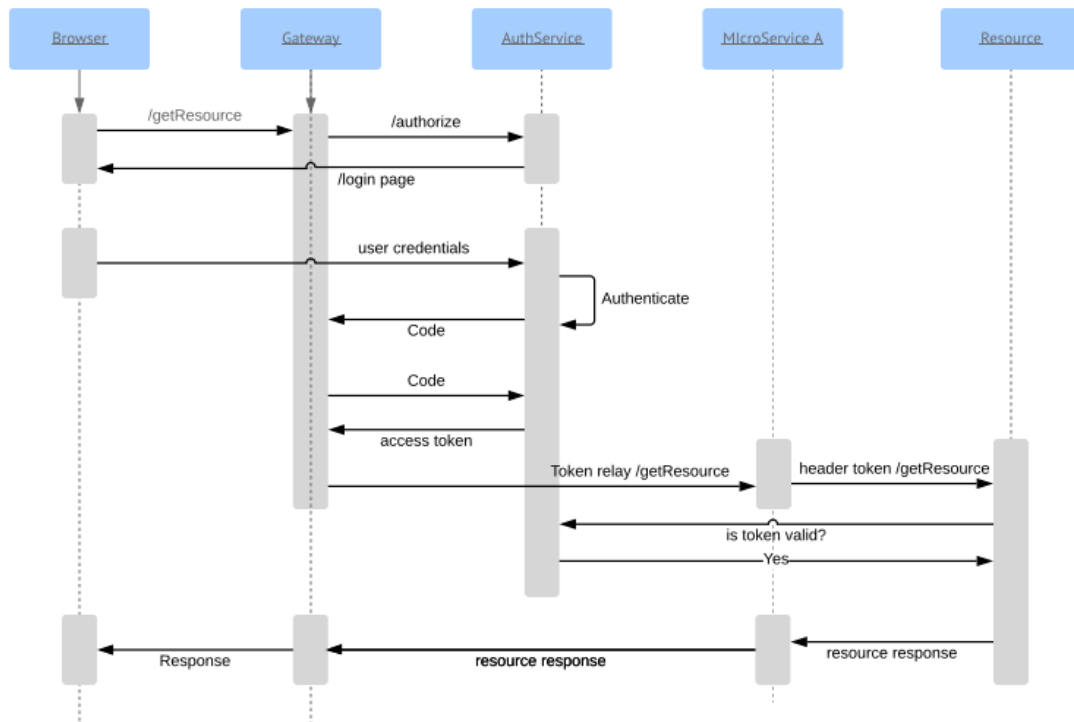
- 1) La richiesta di login arriva a AuthenticationRestController con **/api/authenticate**.
- 2) Il metodo authorize genera un token con username e password.
- 3) Il token viene aggiunto a Authentication e salvato nel **SecurityContextHolder**.
- 4) Bisogna generare il jwt token chiamo quindi tokenProvider.createToken.
- 5) Da application.yml prendo i valori che indicano la durata dei token.  
\*\*createToken
- 6) prendo le authorities da Authentication e genero un jwt token con nome, authorities, chiave di crittografia e durata.
- 7) Genero gli headers http e aggiungo il **jwt Token**
- 8) Nella risposta aggiungo JWTToken(jwt), httpHeaders, HttpStatus.OK

Analizziamo il workflow delle richieste :

- 1) La richiesta viene ricevuta da **JWTFilter** che prende la servletRequest, e risolve il token a partire dalla **HttpServletRequest**

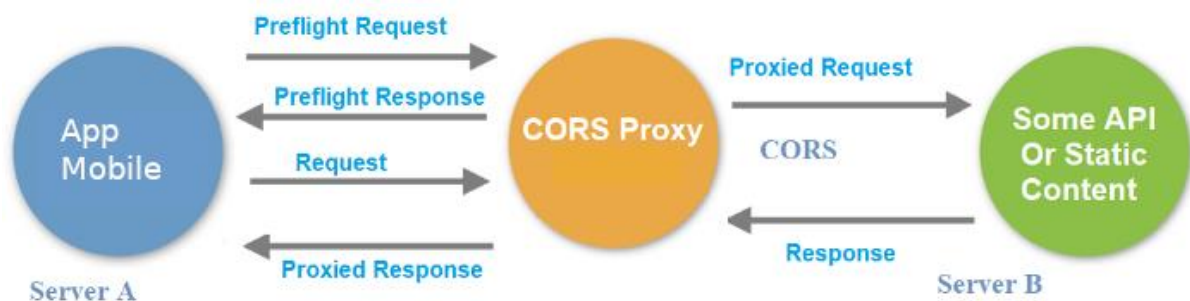


- 2) il token viene validato da tokenProvider  
 Authentication-> tokenProvider.getAuthentication(jwt);
- 3) Authentication viene inserita nel **SecurityContextHolder**
- 4) Adesso è garantito l'accesso verso gli endPoint gestiti dai controller



### 5.5.1 CORS CONFIGURATION

Il Cross-Origin Resource Sharing (CORS), un meccanismo che usa header HTTP aggiuntivi per indicare che un dominio dispone dell'autorizzazione per accedere alle risorse selezionate.



## 5.6 API

La documentazione ha l'obiettivo di mostrare le chiamate API di cui si compone l'applicativo e le quattro classi Java annotate con **@Controller**.

Questo applicativo RESTful è progettato per restituire dati (**JSON**) che verranno inviati dai controller direttamente tramite la risposta HTTP.

L'annotazione **@RestController** di Spring Boot è fondamentalmente una scorciatoia rapida che ci evita di dover sempre definire **@ResponseBody**.

### Login

Per effettuare il login bisogna fare una richiesta di tipo POST, con i seguenti header json:

```
POST https://geopic.flaminiovilla.it/api/authenticate
```

```
@CrossOrigin(origins = "*")
@PostMapping("/authenticate")
public ResponseEntity<UserHelper.JWTToken> authorize(@Valid @RequestBody
LoginDTO loginDto)
)
```

Request	Response
<pre>{   "email" : "viflaadmin@gmail.com",   "password" : "flaminio" }</pre>	<pre>{   "id_token": "nygsvsgsodjpa64s...",   "user": {     "email": "viflaadmin@gmail.com",     "password": null,     "role": ADMIN,     "firstName": "flaminio",     "lastName": "villa",     "regionId": 99   } }</pre>

Chiaramente la password dell'utente non viene inserita all'interno della risposta, ma verrà sostituita con il valore null.

Ad ogni mail sono associati ruoli particolari, quindi in base al tipo di utente che andrà ad effettuare il login, sono previsti diversi tipi di profili, con autorizzazioni differenti. Questa parte non viene gestita, ma verranno utilizzati principalmente i profili di tipo **ADMIN** e **USER**.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 51 di <b>70</b>	
---	--	---	---

### Registrazione

Per effettuare la registrazione bisogna fare una richiesta di tipo POST, i ruoli seguono la seguente gerarchia, è necessario inserire il proprio token durante le richieste:

- Un **admin** può creare un altro **admin**

```
POST https://geopic.flaminiovilla.it/api/register/admin
```

```
@PostMapping("/register/admin")
public User registerAdmin(@Valid @RequestBody UserDTO userDTO)
```

- Un **admin** può creare un profilo di tipo **secretary**

```
POST https://geopic.flaminiovilla.it/api/register/adminSecretary
```

```
@PostMapping("/register/adminSecretary")
public User registerAdminSecretary(@Valid @RequestBody UserDTO userDTO)
```

- Una **secretary** può generare i profili **adminSecretary**

```
POST https://geopic.flaminiovilla.it/api/register/secretary
```

```
PostMapping("/register/secretary")
public User registerSecretary(@Valid @RequestBody UserDTO userDTO)
```

- Chiunque può registrarsi come **user**

```
POST https://geopic.flaminiovilla.it/api/register/user
```

```
@PostMapping("/register/user")
public User registerUser(@Valid @RequestBody UserDTO userDTO)
```

Per testare che l'utente creato ha i permessi necessari sono stati creati specifici endpoint di test, che date le credenziali di login, restituiscono i ruoli posseduti.

```
POST https://geopic.flaminiovilla.it/api/testAdmin
POST https://geopic.flaminiovilla.it/api/testAdminSecretary
POST https://geopic.flaminiovilla.it/api/testSecretary
POST https://geopic.flaminiovilla.it/api/testUser
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 52 di <b>70</b>	
--	--	---	---

### Category

Le **API** della sezione Category sono gestite da due diversi **@RestController** :

#### CategoryController :

Restituisce un json contenente tutte le `category` presenti :

```
POST https://geopic.flaminiovilla.it/category/findAll
RESPONSE[ { "id": 1, "name": "GEOPIC", "color": "#ffff00"}, ... ]
```

```
@GetMapping("/findAll")
List<Category> findAll()
```

#### CategoryCRUDController :

Restituisce un json contenente tutte le `category` presenti ma in forma ridotta, mostrando solo id e nome :

```
POST https://geopic.flaminiovilla.it/api/category/findAllShort
RESPONSE { "id": 1, "name": "GEOPIC" }
```

```
@GetMapping("/findAllShort")
String findAllShort()
```

Dati nome e colore creo un nuovo obj di tipo `category`, se l'inserimento riesce lo restituisco in formato json con l' id (che viene generato automaticamente) :

```
POST https://geopic.flaminiovilla.it/api/category/create
INPUT { "name": "GEOPIC", "color": "#ffff00" }
RESPONSE { "id": 1, "name": "GEOPIC", "color": "#ffff00" }
```

```
@PostMapping("/create")
@ResponseBody
public Category create(@RequestBody CategoryDTO categoryDTO)
```

Specificando id , name e color sovrascrive la `category` con lo stesso id :

```
POST https://geopic.flaminiovilla.it/api/category/update
INPUT { "id": 1, "name": "GEOPIC", "color": "#aaaa11" }
RESPONSE come INPUT
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 53 di <b>70</b>	
--	--	---	---

```
@PutMapping("/update")
@ResponseBody
public Category update(@RequestBody CategoryDTO categoryDTO)
```

Ricerca dato un id e restituisce la *category* corrispondente in formato json.

```
POST https://geopic.flaminiovilla.it/api/category/findById
INPUT {"id": 1}
RESPONSE { "id": 1, "name": "GEOPIC", "color": "#ffff00" }
```

```
@PostMapping("/findById")
Optional<Category> findById(@RequestBody CategoryDTO categoryDTO)
```

Elimina dato un id :

```
POST https://geopic.flaminiovilla.it/api/category/delete
INPUT {"id": 1}
RESPONSE 200
```

```
@DeleteMapping("/delete")
public Boolean delete(@RequestBody CategoryDTO categoryDTO)
```

## News

Le **API** della sezione **News** sono gestite da due diversi **@RestController** :

### NewsController :

Restituisce tutte le *news* che contengono la stringa di ricerca nel titolo, se non trovo niente cerco {search} all'interno della description.

```
POST https://geopic.flaminiovilla.it/news/search/{search}
RESPONSE [{
  "id": 16,
  "description": "Lorem ipsum",
  "title": "10 Ways To Immediately Start Selling POSIZIONE",
  "image": "https://picsum.photos/200/300?random=7",
  "date": "2021-03-02 00:00:00",
  "section": "News"} , ...]
```

```
@PostMapping("/search/{search}")
List<NewsDTO> search(@PathVariable("search") String search, @RequestBody
StructureDTO structureDTO)
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 54 di <b>70</b>	
--	--	---	---

Restituisce tutte le **news** di una section:

```
POST https://geopic.flaminiovilla.it/news/section
INPUT { "section": "News" }
RESPONSE [{"id": 1, "description": "...", "title": "...", "image": "...",
"date": "2021-03-02", "section": "News"}, ...]
```

```
@PostMapping("/section")
List<NewsDTO> section(@RequestBody StructureDTO structureDTO)
```

Ricerca dato un id e restituisce la **news** corrispondente in formato json.:

```
POST https://geopic.flaminiovilla.it/news/findById
INPUT { "section": "News" }
RESPONSE [{"id": 1, "description": "...", "title": "...", "image": "...",
"date": "2021-03-02", "section": "News"}, ...]
```

```
@PostMapping("/findById")
Optional<News> findById(@RequestBody NewsDTO newsDTO)
```

### NewsCRUDController :

Crea una nuova **news** ,Date e' in formato stringa e verrà convertito in in Sql.Date ,section puo' essere: ["News","Comunicazioni","Eventi"], se l'inserimento riesce lo restituisco in formato json con l' id (che viene generato automaticamente).

```
POST https://geopic.flaminiovilla.it/api/news/create
INPUT {"id": 1, "description": "...", "title": "...", "image": "...",
"date": "2021-03-02", "section": "News"}
```

```
@PostMapping("/create")
@ResponseBody
public News create(@RequestBody NewsDTO newsDTO) {}
```

Specificando l'id , e modificando eventuali campi vado a sovrascrivere la **news** :

```
POST https://geopic.flaminiovilla.it/api/news/update
INPUT {"id": 1, "description": "...", "title": "...", "image": "...",
"date": "2021-03-02", "section": "News"}
RESPONSE come INPUT
```

```
@PutMapping("/update")
@ResponseBody
public News update(@RequestBody NewsDTO newsDTO)
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 55 di <b>70</b>	
--	--	---	---

Elimina dato un id :

```
POST https://geopic.flaminiovilla.it/api/news/delete
INPUT {"id": 1}
RESPONSE 200
```

```
@DeleteMapping("/delete")
public Boolean delete(@RequestBody NewsDTO newsDTO)
```

### Place

Questo gruppo di api si occupa della ricerca delle strutture in una determinata località :

Aiuto al completamento del nome della località , mostra solo i primi 10 record

esempio : ...api/place/rom

```
POST https://geopic.flaminiovilla.it/place/{initStr}
RESPONSE[{
  "id": 4792,
  "istat": "58091",
  "comune": "Roma",
  "latitude": 41.89277044,
  "longitude": 12.48366723
}, ...]
```

```
@GetMapping("/{initStr}")
List<Place> findPlaceByNameBeginning(@PathVariable("initStr") String
initialString)
```

Data latitudine e longitudine restituisce il nome del comune più vicino.

```
POST https://geopic.flaminiovilla.it/place/cityname
INPUT { "latitude": 41.89277044,"longitude": 12.48366723 }
RESPONSE { "comune": "Roma" }
```

```
@PostMapping("/cityname")
public String cityname(@RequestBody StructureDTO structureDTO)
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 56 di <b>70</b>	
--	--	---	---

### Referral Person

**ReferralPerson** definisce le generalità del personale gestore delle convenzioni e sono gestite da due diversi **@RestController** :

### ReferralPersonController :

Specificando l'id , e modificando eventuali campi vado a sovrascrivere la **referralPerson** :

**POST** <https://geopic.flaminiovilla.it/referralPerson/findAll>

```

RESPONSE{
  "id": 1,
  "email": "vifla01@gmail.com",
  "name": "Flaminio",
  "surname": "Villa",
  "phone": "3775093443",
  "secretary": {
    "id": 1,
    "name": "geopic",
    "address": "via sandro sandri 81",
    "email": "geopic@flaminiovilla.it",
    "phone": "3775093443E9",
    "region": {
      "id": 3,
      "name": "Calabria"
    }
  },
  "region": {
    "id": 3,
    "name": "Calabria"
  }
}, ...]

},

```

```

@GetMapping("/findAll")
List<ReferralPerson> findAll()

```



 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 57 di <b>70</b>	
--	--	---	---

### ReferralPersonCRUDController :

Restituisce un json contenente tutte le `referralPerson` presenti ma in forma ridotta, mostrando solo id e nome :

```
POST https://geopic.flaminiovilla.it/api/referralPerson/findAllShort
RESPONSE { "id": 1, "name": "Andrei" }
```

```
@GetMapping("/findAllShort")
String findAllShort()
```

Ricerca dato un id e restituisce la `referralPerson` corrispondente in formato json.

```
POST https://geopic.flaminiovilla.it/api/referralPerson/findById
INPUT { "id": 1 }
RESPONSE { "id": 1, "email": "", "name": "", "phone": "", "address": "", "regionId": }
```

```
@PostMapping("/findById")
Optional<ReferralPerson> findById(@RequestBody ReferralPersonDTO
referralPersonDTO)
```

Crea una nuova `referralPerson` e se l'inserimento va a buon fine la restituisce in formato json

```
POST https://geopic.flaminiovilla.it/api/referralPerson/create
INPUT { "email": "", "name": "", "phone": "", "address": "", "regionId": 1 }
RESPONSE come INPUT + ID
```

```
@PostMapping("/create")
@ResponseBody
public ReferralPerson create(@RequestBody ReferralPersonDTO
referralPersonDTO)
```

Specificando l'id , e modificando eventuali campi vado a sovrascrivere la `referralPerson` :

```
POST https://geopic.flaminiovilla.it/api/referralPerson/update
INPUT Come nel caso dell'inserimento ma specificando l'id
```

```
@PutMapping("/update")
@ResponseBody
public ReferralPerson update(@RequestBody ReferralPersonDTO
referralPersonDTO)
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 58 di <b>70</b>	
--	--	---	---

Elimina dato un id (Se presente) :

```
POST https://geopic.flaminiovilla.it/api/referralPerson/delete
INPUT {"id": 1}
RESPONSE 200
```

```
@DeleteMapping("/delete")
public Boolean delete(@RequestBody SecretaryDTO secretaryDTO)
```

### Secretary

La classe **Secretary** contiene le informazioni delle sedi di controllo in cui lavorano le **ReferralPerson** sono gestite da due diversi **@RestController** :

SecretaryController :

Restituisce un json contenente tutte le **secretary** presenti :

```
POST https://geopic.flaminiovilla.it/secretary/findAll
RESPONSE [{
  "id": 1,
  "name": "geopic",
  "address": "via sandro sandri 81",
  "email": "geopic@flaminiovilla.it",
  "phone": "3775093443E9",
  "region": {
    "id": 3,
    "name": "Calabria"
  }
}, ... ]
```

```
@GetMapping("/findAll")
List<Secretary> findAll()
```

SecretaryCRUDController :

Ricerca dato un id e restituisce la **secretary** corrispondente in formato json.

```
POST https://geopic.flaminiovilla.it/api/secretary/findById
INPUT {"id": 1}
RESPONSE {"id": 1,"email": "", "name": "", "phone": "", "address": "", "regionId": }
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 59 di <b>70</b>	
--	--	---	---

```
@PostMapping("/findById")
Optional<Secretary> findById(@RequestBody SecretaryDTO secretaryDTO)
```

Restituisce un json contenente tutte le `secretary` presenti ma in forma ridotta, mostrando solo id e nome :

```
POST https://geopic.flaminiovilla.it/api/secretary/findAllShort
RESPONSE [{ "id": 1, "name": " ", ... }]
```

```
@GetMapping("/findAllShort")
String findAllShort()
```

Crea una nuova `secretary` e se l'inserimento va a buon fine la restituisce in formato json

```
POST https://geopic.flaminiovilla.it/api/secretary/create
INPUT { "email": " ", "name": " ", "phone": " ", "address": " ", "regionId": 1 }
RESPONSE come INPUT + ID
```

```
@PostMapping("/create")
@ResponseBody
public Secretary create(@RequestBody SecretaryDTO secretaryDTO)
```

Specificando l'id , e modificando eventuali campi vado a sovrascrivere la `secretary` :

```
POST https://geopic.flaminiovilla.it/api/secretary/update
INPUT Come nel caso dell'inserimento ma specificando l'id
```

```
@PutMapping("/update")
@ResponseBody
public Secretary update(@RequestBody SecretaryDTO secretaryDTO)
```

Elimina dato un id (Se presente) :

```
POST https://geopic.flaminiovilla.it/api/secretary/delete
INPUT { "id": 1 }
RESPONSE 200
```

```
@DeleteMapping("/delete")
public Boolean delete(@RequestBody SecretaryDTO secretaryDTO)
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 60 di <b>70</b>	
--	--	---	---

### Structure

La classe **Structure** è l'entità principale, contiene infatti tutti dati di una struttura convenzionata da esporre sulla piattaforma, analizziamola per ultimo , dopo aver già analizzato tutte le entity che contiene:

sono gestite da due diversi **@RestController**

**StructureController :**

Restituisce un json contenente tutte le `structure` presenti :

```
POST https://geopic.flaminiovilla.it/structure/findAll
RESPONSE [{
  "id": 30,"name": "Geopic","macroCategory": "Sanitaria",
  "category": {"id": 1,"name": "GEOPIC","color": "#ffff00"},
  "description": "Geopic","logo": "", "discount": 90,"distance": null,
  "expireDateConvention": "2050/12/12","email": "...","phone": "...",
  "website": "https://www.geopic.com",
  "address": "via sandro sandri 71, 00184 Roma RM, Italy",
  "latitude": 41.887701387610505,
  "longitude": 12.496582779052002,
  "secretary": {"id": 2,"name": "Elis","address": "via sandro sandri 71",
    "email": "elis@elis.it","phone": "...",
    "region": {"id": 7,"name": "Lazio" }
  },
  "referralPerson": {
    "id": 2,
    "email": "calinvasileandrei@gmail.com",
    "name": "Andrei",
    "surname": "Calin",
    "phone": "3775093443",
    "secretary": {
      "id": 2,
      "name": "Elis",
      "address": "via sandro sandri 71",
      "email": "elis@elis.it",
      "phone": "0.0",
      "region": {"id": 7, "name": "Lazio"
    }}, "region": {"id": 7,"name": "Lazio"}
  },
  "geoPoint": {
    "longitude": 12.496582779052002,
    "latitude": 41.887701387610505
  }
},,, ... ]
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 61 di 70	
--	--	---------------------------------	---

```
@GetMapping("/findAll")
List<Secretary> findAll()
```

Ricerca dato un id e restituisce la **structure** corrispondente in formato json.

```
POST https://geopic.flaminiovilla.it/structure/findById
INPUT {"id": 1}
RESPONSE {"id": 1,"email": "", "name": "", "phone": "", "address": "", "regionId": }
```

```
@PostMapping("/findById")
StructureDateDTO findById(@RequestBody StructureDTO structureDTO)
```

Restituisco tutte le **structure** che sono di una determinata macro categoria e contengono la posizione (ESEMPIO roma ) all'interno del campo indirizzo dopo / .

```
POST https://geopic.flaminiovilla.it/structure/findSection/{search}
INPUT {"macroCategory": "Convenzioni Commerciali"}
```

```
@PostMapping("/findSection/{search}")
List<StructureDateDTO> findSection(@PathVariable("search") String search,
@RequestBody StructureDTO structureDTO)
```

Restituisco tutte le **structure** che sono di una determinata macro categoria e località.

```
POST https://geopic.flaminiovilla.it/structure/findSection
INPUT {"macroCategory": "Convenzioni Commerciali", "location": "roma"}
```

```
@PostMapping("/findSection")
List<StructureDateDTO> findSection(@RequestBody StructureDTO structureDTO)
```

Grazie all'algoritmo di ricerca delle **structure** , analizzato precedentemente, dato un punto geografico ed una distanza in km , effettua la ricerca delle strutture comprese in questa distanza

```
POST https://geopic.flaminiovilla.it/structure/findStructuresNearGeoLocation
INPUT {
  "latitude": 38.9027885,
  "longitude": 16.5944429,
  "distance": 6
}
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 62 di <b>70</b>	
--	--	---	---

```
@PostMapping("/findStructuresNearGeoLocation")
List<StructureDateDTO> findStructuresNearGeoLocation(@RequestBody
StructureDTO structureDTO)
```

Data la posizione per cui si vogliono avere le `structure` restituisce una lista di strutture divise per macro categorie .

```
POST https://geopic.flaminiovilla.it/structure/findAllByCategory
INPUT {
  "location": "catanzaro",
  "category": "farmacia"
}
```

```
@PostMapping("/findAllByCategory")
List<StructureDateDTO> findAllByCategory(@RequestBody StructureDTO
structureDTO)
```

Restituisce tutte le `structure` che contengono `"location"` all'interno dell'indirizzo

```
POST https://geopic.flaminiovilla.it/structure/findAllStructuresByLocation
INPUT {
  "location": "catanzaro"
}
```

```
@PostMapping("/findAllStructuresByLocation")
String findAllStructuresByLocation(@RequestBody StructureDTO structureDTO)
```

Data la `"location"` restituisce i customMarker di quella località:

```
POST https://geopic.flaminiovilla.it/structure/markers
RESPONSE {
  "id": 39,
  "name": "Geopic",
  "latitude": 38.89357180922953,
  "longitude": 16.587004309020646,
  "category": {
    "id": 1,
    "color": "#ffff00",
    "name": "GEOPIC" }}
```

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 63 di <b>70</b>	
--	--	---	---

```
@PostMapping("/markers")
@ResponseBody
public List<MarkerStructureDTO> markers(@RequestBody StructureDTO
structureDTO)
```

Ricerca dentro il db le **structure** che contengono le parole cercate nel nome, data una localita' .

```
POST https://geopic.flaminiovilla.it/structure/search/{search}
INPUT {
  "location": "catanzaro"
}
```

```
@PostMapping("/search/{search}")
List<StructureDateDTO> findStructureByContainsLetters(@PathVariable("search")
String search, @RequestBody StructureDTO structureDTO)
```

StructureCRUDController :

Elimina dato un id (Se presente) :

```
POST https://geopic.flaminiovilla.it/api/structure/delete
INPUT {"id": 1}
RESPONSE 200
```

```
DeleteMapping("/delete")
public Boolean delete(@RequestBody StructureDTO structureDTO)
```

Crea una nuova **structure** e se l'inserimento va a buon fine la restituisce in formato json

```
POST https://geopic.flaminiovilla.it/api/structure/create
INPUT
{
  "name" : "", "macroCategory" : "", "description" : "", "discount" : ,
  "expireDateConvention" : "", "email" : "", "phone" : "", "website" : "",
  "address" : "", "latitude" "", "longitude" "", "categoryId" : 1,
  "referralPersonId" : 1, "secretaryId" : 1 ,
}
RESPONSE come INPUT + ID
```

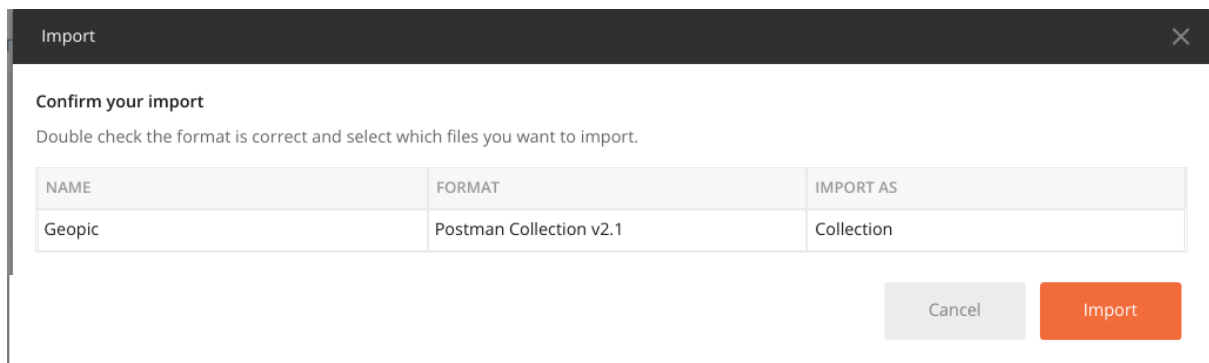
```
@PostMapping("/create")
@ResponseBody
public Structure create(@RequestBody StructureDTO structureDTO)
```

Specificando l'id , e modificando eventuali campi vado a sovrascrivere la **structure** :

**POST** <https://geopic.flaminiovilla.it/api/structure/update>  
**INPUT** Come nel caso dell'inserimento ma specificando l'id

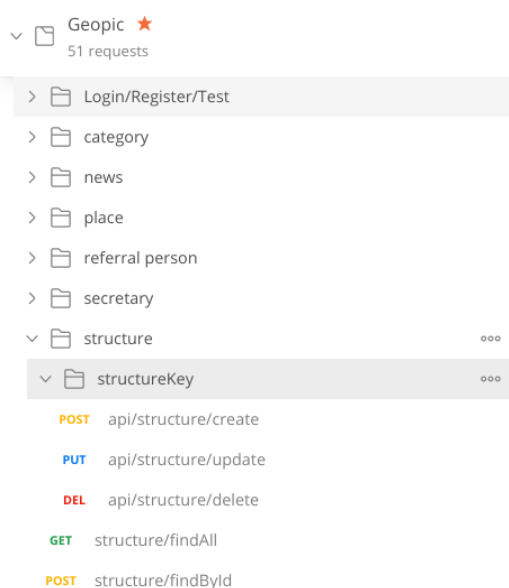
```
@PutMapping("/update")
@ResponseBody
public Structure update(@RequestBody StructureDTO structureDTO)
```

Tutte le **API** sono state testate con **Postman** (<https://www.postman.com/>) , di seguito riporto il link per importare i test relativi a tutte le singole api sopra descritte:



E' possibile scaricare le collection di postman dal seguente link:

[https://polimi365-my.sharepoint.com/:u:/g/personal/10713791\\_polimi\\_it/EXycM3PA4qlMk5Qt-B4tnYYB\\_pxAeeznYR4Ogh1TgUE42g?e=hCdoWJ](https://polimi365-my.sharepoint.com/:u:/g/personal/10713791_polimi_it/EXycM3PA4qlMk5Qt-B4tnYYB_pxAeeznYR4Ogh1TgUE42g?e=hCdoWJ)



Per accedere ad alcune api, è necessaria l'autenticazione occorre quindi effettuare prima il login e successivamente includere il token nella richiesta, esempio:  
 Authorization + Bearer eyJhbGciOiJIUzUxMiJ9....



## 5.7 TESTING

Il testing sfrutta le classi di SpringBootTest e la piattaforma JUnit 5, I Test d'integrazione per l'applicazione sono contenuti all'interno del package:

```
src/test/java/com/flaminiovilla/geopic/
```

I test sono stati realizzati con SpringBootTest, una libreria Maven che permette di effettuare un setup automatico dell'ambiente di test, viene richiamata tramite l'annotazione @SpringBootTest all'inizio di ogni classe.

Il metodo testato va annotato con @Test e al suo interno si richiamano i metodi della classe Assertions (facente parte della libreria JUnit 5 prevista da SpringBootTest) per verificare la coerenza dei dati.

I metodi della classe Assertions usati sono:

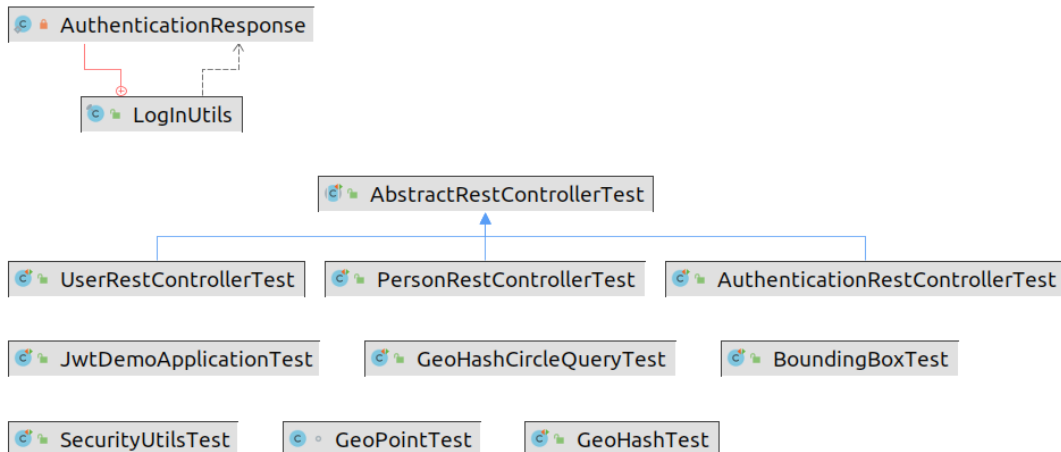
- **assertTrue()** verifica che la condizione passata nelle parentesi tonde sia vera;
- **assertFalse()** verifica che la condizione passata nelle parentesi tonde sia falsa;
- **assertNull()** verifica che la condizione passata nelle parentesi tonde ritorni null;
- **assertNotNull()** verifica che la condizione passata nelle parentesi tonde non ritorni null;
- **assertThat()** verifica che il valore atteso sia uguale al valore effettivo;
- **assertContains()** verifica che un array contiene un valore;
- **assertNotContains()** verifica che un array non contiene un valore;

Per quanto riguarda il testing delle api è stato fatto uso dei MockMvc, definiti come punto di ingresso principale per i test Spring MVC lato server. I test si MockMvc si trovano a metà strada tra i test di unità e quelli di integrazione.

I controlli implementati sono:

- **.contentType()** verifica il tipo di parametri ricevuti ( esempio formato JSON);
- **.content()** verifica se sono presenti determinati parametri;
- **.addApect()** verifica se il tipo di risposta lato server coincide (esempio : unauthorized);
- **.getResponse()** analizza il risultato della servlet dispatcher;

La struttura dei test è la seguente:



## REPORT DI TESTING:

✓ <default package>	3 sec 73 ms
✓ JwtDemoApplicationTest	82 ms
✓ contextLoads	82 ms
✓ GeoHashCircleQueryTest	2 ms
✓ testIssue3WithCircleQuery	2 ms
✓ test180MeridianCircleQuery	
✓ PersonRestControllerTest	73 ms
✓ getPersonForAnonymous	73 ms
✓ UserRestControllerTest	7 ms
✓ getActualUserForUserWithoutToken	7 ms
✓ SecurityUtilsTest	69 ms
✓ getCurrentUsernameForNoAuthenti	68 ms
✓ getCurrentUsername	1 ms
✓ BoundingBoxTest	9 ms
✓ testEqualsObject	
✓ testContains	9 ms
✓ testHashCode	
✓ GeoHashTest	
✓ testEqualsAndHashCode	
✓ AuthenticationRestControllerTest	2 sec 831 ms
✓ unsuccessfulAuthenticationWithNo	895 ms
✓ unsuccessfulAuthenticationWithWr	402 ms
✓ successfulAuthenticationWithUser	675 ms
✓ unsuccessfulAuthenticationWithDis	374 ms
✓ successfulAuthenticationWithAdmi	485 ms

Essendo molto vasta la struttura sono riportati i test dei punti critici principali del sistema:

**JwtDemoApplicationTest**

Inizializza i test con `@SpringBootTest`

Per i test che coprono l'intera applicazione Spring Boot dalla richiesta in entrata al database, o per i test che coprono alcune parti dell'applicazione che sono difficili da configurare manualmente, usiamo `@SpringBootTest`.

`@SpringBootTest` per impostazione predefinita inizia la ricerca nel pacchetto corrente della classe di test e quindi cerca verso l'alto nella struttura del pacchetto, cercando una classe annotata con la `@SpringBootTestConfiguration` e legge la configurazione per creare un contesto dell'applicazione.

Utilizziamo poi `MockMvc` per eseguire una POST richiesta, per verificare che risponda come previsto.

 <b>POLITECNICO</b> MILANO 1863	<h1 style="text-align: center;">Geopic</h1> <h2 style="text-align: center;">Specifica Tecnica</h2>	27-06-2021  Pag. 67 di <b>70</b>	
--	--	---	---

Analizziamo di seguito l'utilità delle varie classi di test.

### **GeoHashTest**

Controlla che l'operazione di hashing di latitudine e longitudine, specificando la precisione dei bit, avvenga correttamente.

### **GeoPointTest**

Crea diversi **GeoPoint** e successivamente associa i valori corrispondenti, creandone delle copie e verificando che il processo di confronto vada a buon fine.

Inoltre viene effettuato il test dell'algoritmo di VincentyGeodesy.

### **BoundingBoxTest**

Definite delle **BoundingBox** verifico se contengono dei **GeoPoint**, definiti internamente secondo specifiche criticità, per esempio punti molto esterni o molto vicini al centro della **BoundingBox**.

### **GeoHashCircleQueryTest**

Dati dei **GeoPoint** ed un raggio crea una approssimazione di un cerchio in quadrato, che corrisponderà poi ad una **BoundingBox** analizzando se contiene dei punti critici.

In alcuni punti è precedentemente è stata calcolata la distanza relativa grazie a google maps, e successivamente è stata effettuata la verifica della distanza calcolata mediante l'algoritmo di VincentyGeodesy.

### **PersonRestControllerTest**

Tramite `assertSuccessfulPersonRequest` verifico se tutti i parametri vengono trasmessi correttamente, e se il token viene aggiunto correttamente nell'Header.

### **AuthenticationRestControllerTest**

Esegue dei test relativi l'accesso:

1. `successfulAuthenticationWithUser`
2. `successfulAuthenticationWithAdmin`
3. `unsuccessfulAuthenticationWithDisabled`
4. `unsuccessfulAuthenticationWithWrongPassword`
5. `unsuccessfulAuthenticationWithNotExistingUser`

### **UserRestControllerTest**

Provo a risalire all'utente loggato, estraendo il token dall'Header della request.

### **LogInUtils**

Effettuo il test della generazione del token con `getTokenForLogin` e ne controllo la validità.

 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 68 di <b>70</b>	
--	--	---	---

## 5.8 Docker

**Docker** è uno strumento progettato per semplificare la creazione, la distribuzione e l'esecuzione di applicazioni utilizzando i **container**.

I **container** consentono di creare un pacchetto di un'applicazione con tutte le parti di cui ha bisogno, e distribuirla come un unico pacchetto. In questo modo, grazie al **container**, l'applicazione verrà eseguita su macchina Linux.

Per effettuare il deploy di Geopic grazie a **docker compose** si unisce **mariadb** con l'app **spring**.

**Compose** è uno strumento per la definizione e l'esecuzione di applicazioni Docker multi-container, si utilizza un file YAML per configurare i servizi per l'applicazione. L'utilizzo di **Compose** è fondamentalmente un processo in tre fasi: Definizione dell'ambiente della applicazione con un in Dockerfile in modo che possa essere riprodotta ovunque. Definizione dei servizi che la compongono in docker-compose.yml per essere eseguiti insieme in un ambiente isolato.

1. Esecuzione di **docker compose up --build** (il comando di composizione Docker avvia ed esegue l'intera app).

Contenuto del **docker-compose.yml** :

```
version: '3.1'
services:
  database:
    image: mariadb
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: -----@
    volumes:
      - /home/ubuntu/docker/geopic:/var/lib/mysql
    ports:
      - 3309:3306
  backend:
    depends_on:- database
    build: .
    restart: always
    command:
      java -jar /app.jar
    --
    spring.datasource.url="jdbc:mariadb://database/geopic?createDatabaseIfNotExists=true&autoReconnect=true&useSSL=false"
    --spring.datasource.password="-----@"
    ports:
      - 5001:8080
```

## 6 Conclusioni

La documentazione di questa piattaforma formata dal server backend e dell'applicazione frontend fornisce le informazioni necessarie per capire il progetto al meglio, capire il suo funzionamento e come queste due parti interagiscono tra di loro.

La documentazione definisce la logica e l'architettura utilizzata per la realizzazione della piattaforma, non viene trattato il codice riga per riga dato che è stato commentato adeguatamente, così che possa essere compreso al meglio.

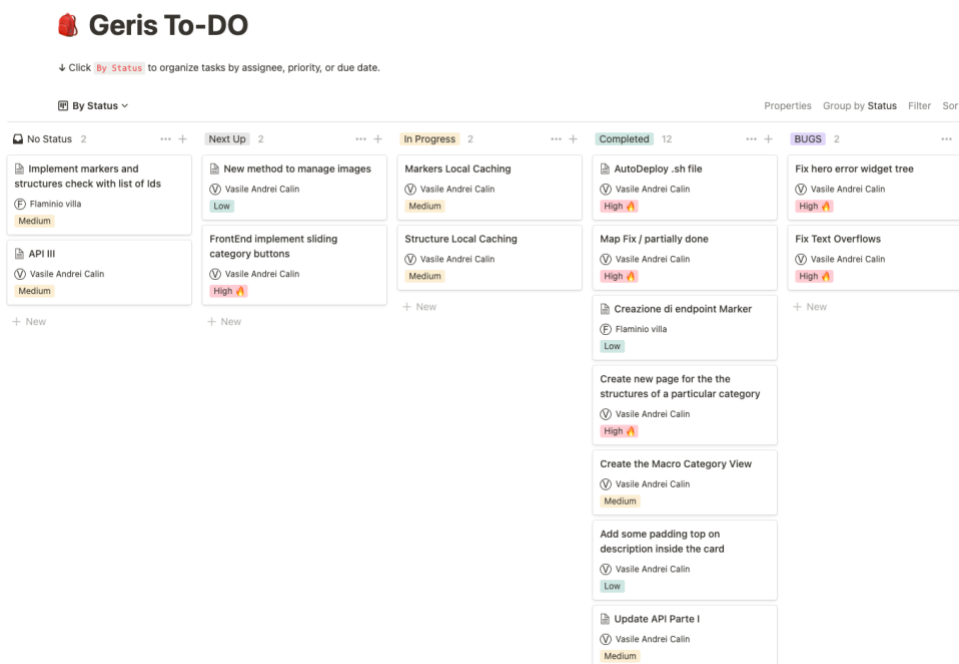
Per collaborare efficacemente durante la realizzazione, è stata usata una task-board che permette la comunicazione di problematiche o richieste nel team. Usando questo tipo di board è possibile creare una "issue" che rappresenta un problema, una funzionalità o una feature da implementare così da garantire una certa autonomia nello sviluppo del lato frontend/backend. Analizziamo una casistica:

Il frontend necessita di un endpoint che ancora non è stato sviluppato dal backend.

La persona che lavora sullo sviluppo frontend crea una issue per esempio (Implementazione Endpoint per la ricerca delle strutture), a questa issue è possibile assegnare una priorità che rappresenta l'urgenza.

Lato backend lo sviluppatore dovrà cercare di risolvere le issue a lui assegnate così da garantire il corretto proseguimento del lavoro.

Ovviamente con la stessa logica può avvenire la situazione opposta dei Team di sviluppo o l'interazione di più Team nel caso di progetti di grandi dimensioni formati da sviluppatori backend, frontend, designer ecc.



 <b>POLITECNICO</b> MILANO 1863	<h1>Geopic</h1> <h2>Specifica Tecnica</h2>	27-06-2021  Pag. 70 di <b>70</b>	
---	--	---	---

I sorgenti sia della parte front end che della parte backend sono pubblici sui nostri github:



Back-end

<https://github.com/villaflaminio/geopic.git>

Front-end

[https://github.com/calinvasileandrei/geopic\\_polimi](https://github.com/calinvasileandrei/geopic_polimi)

E' anche possibile scaricare in formato .zip i sorgenti e l'APK (Disponibile per android)



[Link Drive](#)



[Link Video](#)

**Speriamo che il nostro progetto abbia soddisfatto tutte le aspettative create, stiamo provando a concretizzarlo, rendendolo disponibile presso gli store.**

**Per qualunque domanda, o problema di configurazione siamo reperibili tramite:**

Vasile Andrei Calin , [vasileandrei.calin@mail.polimi.it](mailto:vasileandrei.calin@mail.polimi.it), 10735153, 939029

Flaminio Villa, [flaminio.villa@mail.polimi.it](mailto:flaminio.villa@mail.polimi.it) , 10713791, 938967