



UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

CORSO DI LAUREA IN INFORMATICA

SMARTAPI

DOCUMENTAZIONE

Caso di studio del corso di
Sistemi di Elaborazione per l'automazione d'Ufficio a.a. 2019-2020

A cura di:

Nicola Castiglione e Roberto Miccolis

Sommario

Introduzione	4
1.1 Architettura	5
1.2 Database	7
1.2.1 Progettazione.....	7
1.2.2 Installazione	12
1.2.3 Modalità di connessione con Node.js.....	12
1.3 Server.....	14
1.3.1 Diagramma dei componenti Node.js con Express.js	15
1.3.2 Servizi API implementati	16
1.3.3 Struttura	20
1.3.4 Installazione delle dipendenze	21
1.3.5 Driver MySQL	23
1.3.6 Autenticazione e autorizzazione con JWT	25
1.3.7 Middleware Validazione dati input.....	26
1.3.8 Controller ElaboraQuery	27
1.3.9 Controller Calcolo punteggio.....	28
1.3.10 Google API	29
1.3.11 Builder Pattern.....	30
3.4 Client.....	31
3.4.1 Diagramma dei componenti Vue App con Vuex & Vue Router	32
3.4.2 Struttura	34
3.4.3 Installazione ed utilizzo dei componenti.....	34
3.4.4 Autenticazione e autorizzazione con JWT	37
3.4.5 Componente Vuex store/auth	38

3.4.6	Componenti Login & Registrazione.....	38
3.4.7	Componente Profile	40
3.4.8	Componente Score	41
3.4.9	Libreria vue2-google-maps	44

Introduzione

SmartAPI è un'applicazione web che si propone di calcolare e restituire a chi la utilizza un indicatore di efficienza di una località (via o paese) rispetto ad altre, limitatamente al contesto nazionale (Italia).

L'intero sviluppo dell'applicazione nasce sulla base del progetto "Urban Index Integration" di Didonna Davide implementato per la sua tesi di laurea. Le differenze sostanziali risiedono sia sulla tipologia di applicazione, effettuando un porting da app Android a web application cambiandone quindi l'intera architettura, sia nel cercare di apporre miglioramenti dove fosse possibile, sia nel dotarla di una flessibilità tale da permettere modifiche future.

L'implementazione del progetto è avvenuta attraverso l'uso del linguaggio di programmazione JavaScript, sia per la realizzazione della parte front-end (client), utilizzando il framework Vue.js, e sia della parte back-end (server), con Node.js ed Express, focalizzandosi sui seguenti punti:

- realizzare un sistema di autenticazione;
- realizzare un servizio accessibile solamente all'utente che ha effettuato il login;
- realizzare un database per l'estrazione dei dati;
- implementare alcune semplici funzioni per le elaborazioni di dati;
- fornire i dati riguardanti una località (via o paese) ricercata da un determinato cliente.

Il progetto è stato realizzato da Nicola Castiglione e Roberto Miccolis per il corso di Sistemi di Elaborazione per l'automazione d'Ufficio, con l'ausilio ed il supporto del docente del corso.

1.1 Architettura

Il sistema SmartAPI è stato sviluppando seguendo un'architettura Client-Server che lavora in modalità mash-up, prelevando i dati da diverse fonti, ma che potrebbe in futuro lavorare anche in stand-alone. L'architettura si suddivide nei due seguenti livelli:

- Server: costituito dal back-end, si occupa di interagire col database per prelevare i dati richiesti dal Client;
- Client: che corrisponde al front-end, si occupa dell'interazione dell'utente via browser, con la Web App.

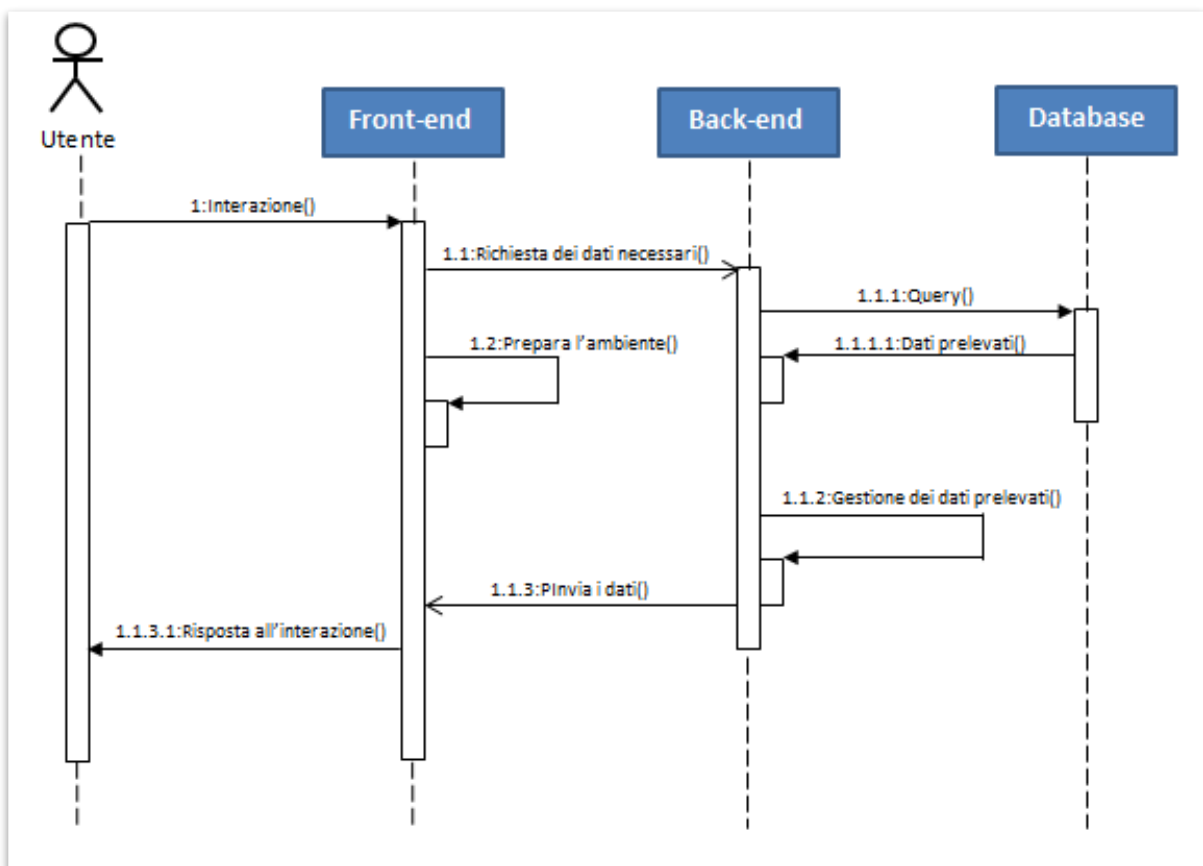


Figura 1 Diagramma di sequenza: interazione generica

Nel diagramma in Figura 1 si mostra un'interazione generica da parte dell'utente con l'applicazione web.

L'utente interagisce con l'applicazione e richiede la visualizzazione dei dati o l'esecuzione di un'operazione. La componente front-end dell'applicazione, quindi, passa la richiesta al back-end, sul server. Quest'ultimo fa quanto richiesto dall'utente, tramite l'esecuzione di una o più query sul database. I dati ottenuti dalla base di dati vengono organizzati e preparati per l'invio al front-end. Nel frattempo, il front-end esegue delle operazioni di preparazione dell'ambiente, come per esempio l'aggiornamento dell'interfaccia grafica e dati in arrivo, in risposta alla richiesta dell'utente, vengono quindi visualizzati all'interno della stessa. Le richieste da parte del front-end per il back-end sono realizzate tramite chiamate simili al concetto alla base di AJAX (Asynchronous JavaScript and XML). Tecnica che permette lo scambio di dati tra server e client in background. La pagina web si aggiorna così in maniera dinamica senza che l'utente debba ricaricarla.

Dopo il primo caricamento della pagina web, tutte le richieste di dati ulteriori, da parte dell'utente, vengono effettuate tramite chiamate http asincrone, che rendono la comunicazione client-server molto semplice.

1.2 Database

Nonostante l'adozione di Node.js come back-end non è da escludere l'utilizzo di un database relazionale per la persistenza dei dati. La necessità di tener traccia delle informazioni degli utenti, così da gestirne l'autenticazione e le ricerche effettuate (con i possibili costi associati ad essa), ha fatto ricadere la scelta su un database SQL, ovvero MySQL e in particolare la versione 5.7.11, ampiamente supportata dallo strumento visuale di progettazione, MySQL Workbench. Lo strumento oltre ad essere multiplatforma, integra lo sviluppo SQL, la gestione, la modellazione dati, la creazione e la manutenzione di un database MySQL all'interno di un unico ambiente sinergico.

1.2.1 Progettazione

Le diverse tabelle mostrate nel diagramma entità relazione (ER), in Figura 2, sono opportunamente relazionate in modo da tener traccia degli utenti registrati al sistema, di tutti i comuni d'Italia (dati aggiornati al 01/01/2020), delle attività e degli indicatori servizio per servizio di una determinata località (via o paese), così da poter ricostruire la ricerca eseguita da ogni singolo utente.

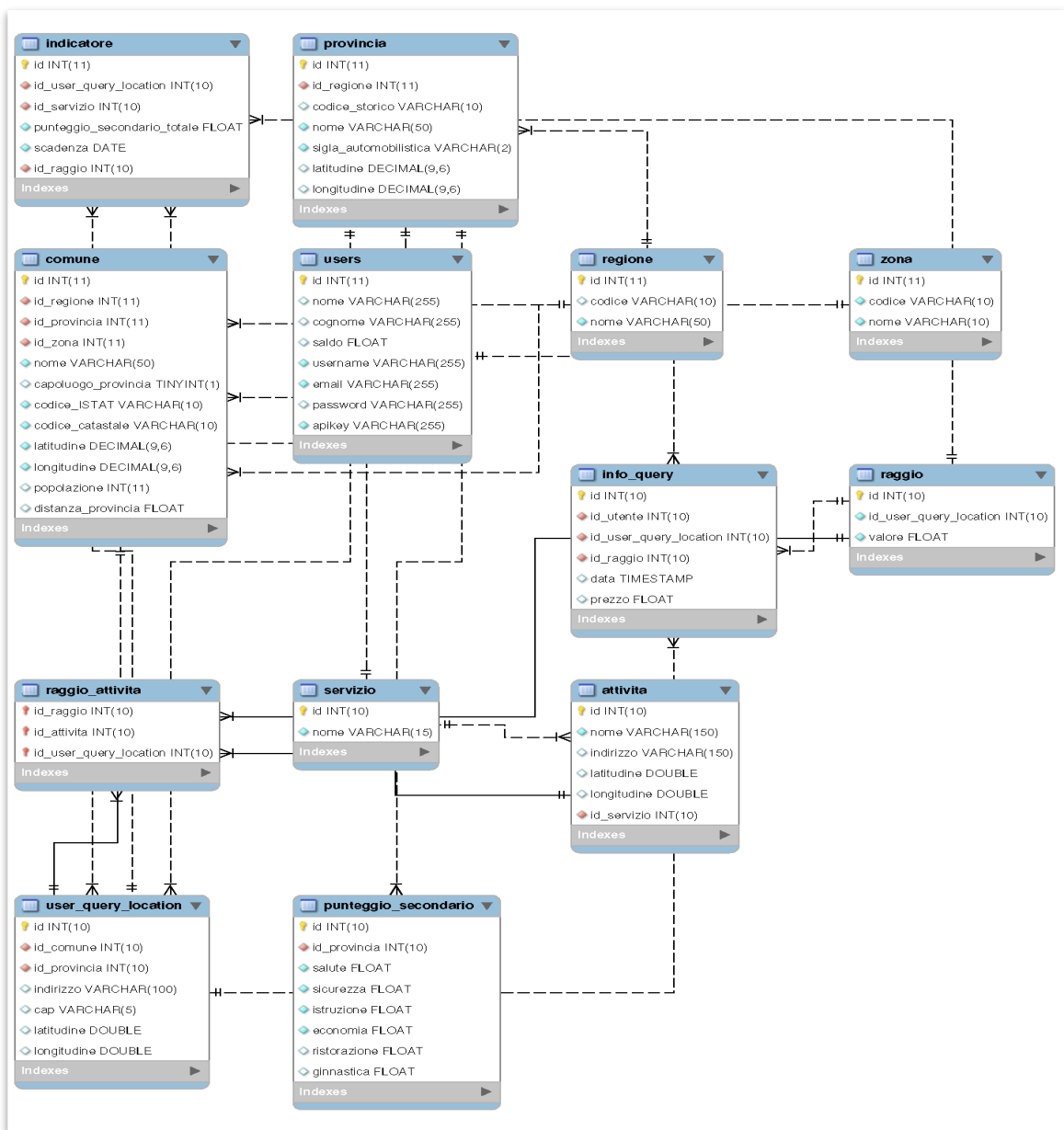


Figura 2 Diagramma Entità Relazione

Analizziamo le tabelle nel dettaglio:

- **regione**: la tabella regione contiene le informazioni base delle regioni del territorio italiano, ovvero il nome univoco (non esistono in Italia, e non si pensa possano esistere in futuro, regioni con lo stesso nome), e il codice associato ad ogni regione anch'esso univoco.
- **provincia**: la tabella provincia contiene tutte le informazioni identificative di una provincia facenti parti delle regioni italiane, ovvero il nome univoco (non esistono,

e non si pensa possano esserlo in futuro, provincie con lo stesso nome); il codice storico; la sigla automobilistica anch'essa univoca; le sue coordinate geografiche (latitudine e longitudine) registrate in forma decimale; la chiave esterna che la collega alla tabella `regione`.

- **punteggio_secondario**: la tabella `punteggio_secondario` contiene il nome di sei precisi servizi e un valore decimale indicante un punteggio qualitativo assegnato ad ogni provincia.
- **zona**: la tabella `zona` contiene le informazioni base delle zone del territorio italiano, ovvero il nome univoco e il codice univoco.
- **comune**: la tabella `comune` contiene tutte le informazioni identificative di un comune facente parte delle provincie e delle regioni italiane in una determinata zona, in particolare il nome, un valore booleano che rappresenta se il comune è anche capoluogo di provincia, il codice ISTAT univoco per ogni comune, il codice catastale anch'esso univoco, le sue coordinate geografiche (latitudine e longitudine) registrate in forma decimale, la popolazione, la distanza dalla provincia espressa in chilometri e le chiavi esterne per mantenere l'integrità con le tabelle `regione`, `provincia`, `zona`. Il nome del comune non è un campo univoco a sé stante, infatti si è pensato di renderlo univoco solo insieme alla latitudine e alla longitudine, in quanto in Italia sono presenti 8 comuni con lo stesso nome più 2 che differiscono per il solo accento.
- **users**: la tabella `users` contiene tutte le informazioni riguardanti gli utenti che utilizzano SmartAPI. Sono presenti dati che servono all'autenticazione, cioè l'username (univoco per ogni utente) e password, le informazioni generali, quali nome, cognome ed e-mail, il saldo ed una chiave API univoca assegnata ad ogni utente in fase di registrazione. L'username e l'e-mail sono campi univoci in modo tale da non permettere ad un utente di registrarsi nuovamente con la stessa e-mail.
- **user_query_location**: la tabella `user_query_location` contiene le informazioni della località (via o paese) cercata dall'utente. Non avendo a disposizione un database contenente l'intero stradario italiano, è stata lasciata la possibilità di poter aggiungere alle informazioni già presenti dei comuni e delle provincie, l'indirizzo, il cap (non obbligatorio) e le coordinate geografiche (latitudine e longitudine) registrate in forma decimale. Per mantenere l'integrità sono presenti le chiavi

esterne relative alla tabella comune e provincia, che insieme all'indirizzo sono univoci. L'indirizzo è un campo alfanumerico che a seconda della ricerca dell'utente, per semplicità, potrà contenere il solo nome del comune, il solo nome della strada, oppure il nome della strada completo di civico. Esempio: "Bari", "Via Edoardo Orabona" e "Via Edoardo Orabona, 4".

- **raggio**: la tabella `raggio` contiene le informazioni relative al raggio d'azione della località cercata, cioè il valore numerico espresso in metri e la chiave esterna al fine di collegarla alla tabella `user_query_location`. Il valore del raggio sarà univoco insieme alla chiave esterna.
- **info_query**: la tabella `info_query` contiene le informazioni relative alle ricerche effettuate dagli utenti, come la data della ricerca, il prezzo (assegnato a 0.00 di default, non essendo stata gestita la monetizzazione) e le chiavi esterne delle tabelle `users`, `raggio` e `user_query_location`.
- **servizio**: la tabella `servizio` contiene i nomi (univoci) dei servizi di interesse ai fini del calcolo dell'indicatore.
- **attività**: la tabella `attività` contiene le informazioni relative alle attività (esempio ristoranti, banche, scuole ecc.), in particolare il nome, l'indirizzo, le coordinate geografiche (latitudine e longitudine) registrate nel formato double e la chiave esterna che collega l'attività al servizio. Il nome, la chiave esterna e le coordinate sono insieme univoci, in quanto non si pensa possa esistere un'attività con lo stesso nome e servizio, nella stessa posizione. Il campo nome ed indirizzo inoltre sono state impostate con la codifica `utf8mb4`, in modo tale da poter memorizzare anche possibili simboli, come le emojis.
- **indicatore**: la tabella `indicatore` contiene tutte le informazioni degli indicatori, servizio per servizio, di una determinata località cercata. Sono presenti le chiavi esterne delle tabelle `user_query_location`, `servizio` e `raggio`, il punteggio e la scadenza dello stesso indicatore, in quanto è stato previsto che i valori calcolati non devono essere considerati sempre validi nel tempo.
- **raggio_attività**: la tabella `raggio_attività` contiene le informazioni utili per poter ricondurre l'attività alla ricerca di una località in un determinato raggio d'azione. Sono presenti le chiavi esterne relative alle tabelle `raggio`, `attività` e `user_query_location`;

Si precisa che per le tabelle: regione, provincia, comune e zona sono stati utilizzati i dati di un database non ufficiale dei comuni italiani (7904), disponibile in formato JSON al seguente indirizzo:

<https://github.com/matteocontrini/comuni-json/raw/master/comuni.json>

Il database nonostante non sia ufficiale è basato su dati ISTAT aggiornati al 01/01/2020. A questi dati sono state apportate delle integrazioni con le informazioni riguardanti la posizione geografica di ogni comune e provincia, e la distanza calcolata tra gli stessi.

Le coordinate sono state reperite dal template sinottico “Divisione Amministrativa” (più comunemente “InfoBox”), una tabella riepilogativa che dà un insieme di informazioni di base sul soggetto di una voce, dall'enciclopedia libera Wikipedia. In Figura 3 possiamo osservare un esempio relativo alla città di Roma.



Figura 3 Divisione Amministrativa del comune di Roma [Fonte: it.wikipedia.org]

La distanza in chilometri, invece, è stata calcolata per mezzo dell'API 'distance-from' disponibile su NPM. L'API sfrutta la formula dell'emisfero per il calcolo della distanza tra due coordinate con un margine di errore di +/- 0.03%

1.2.2 Installazione

In ambito di sviluppo, installare un database MySQL può essere complicato e soprattutto diverso a seconda del sistema operativo utilizzato. Per questo si è deciso di utilizzare Docker che permette di avviare dei contenitori, pronti all'uso, indipendenti sul proprio sistema, recuperandoli dall'enorme repository di cui è fornito. Prima di iniziare ad utilizzare Docker bisogna necessariamente installarlo sul proprio sistema, assicurarsi che il servizio sia in esecuzione e infine procedere ad avviare il nostro contenitore:

```
docker run --name mysql -p 3406:3306 -e MYSQL_ROOT_PASSWORD=mypassword -d mysql/mysql-server:5.7.11 --innodb-autoinc-lock-mode=0 --character-set-server=utf8mb4 --collation-server=utf8_unicode_ci
```

1.2.3 Modalità di connessione con Node.js

Ci sono diversi approcci che permettono di utilizzare un RDBMS in Node.js:

- **Driver database di basso livello:** bisogna conoscere sicuramente la sintassi del DB da utilizzare e bisogna imparare la sintassi della libreria che utilizzeremo in Node.js, ad esempio 'mysql' o 'mysql2'; sicuramente questo è il metodo più efficiente. Per il

Related / Similar Packages		
	knex, sequelize, bookshelf, hapi	knex, sequelize, hapi, bookshelf
Comparison		
Licenses	MIT	MIT
Created	9 years ago (Jan, 2011)	7 years ago (Apr, 2013)
Modified	2 months ago	a month ago
Total Versions	65	119
Version Average	every 2 months	every 21 days
Maintainers	4	3
Dependencies	4	9
Daily Downloads	101,240	63,702
Weekly Downloads	616,994	379,271
Monthly Downloads	2,553,327	1,532,437
Open Issues	149	223
Open Pull Requests	28	34
Stargazers	15,269	2,069
Subscribers	497	60
Forks	2,131	305
Wiki	✖	✓
Points		
Overall Points	1,654,403	992,034
	Points	Points

Figura 4 Confronto tra "mysql" e "mysql2" [Fonte: npmcompare.com]

progetto è stato utilizzata la libreria 'mysql2' per la documentazione disponibile, in figura si può osservare un confronto tra i due pacchetti.

- **Costruttore di query:** bisogna conoscere una libreria, ad esempio 'knex', così da essere in grado di generare programmaticamente query dinamiche in un modo molto più conveniente, soprattutto se bisogna concatenare stringhe per formare la query SQL.

- **Alto livello ORMs** (object-relational mapping): non bisogna conoscere la sintassi SQL, ma bisogna conoscere la sintassi della libreria ORM che utilizzeremmo, ad esempio 'sequelize'. Una libreria ORM fornisce, mediante un'interfaccia orientata agli oggetti, tutti i servizi inerenti alla persistenza dei dati, astruendo allo stesso tempo le caratteristiche implementative dello specifico RDBMS utilizzato. Purtroppo, la sintassi di una libreria non è identica ad un'altra e le query generate non sono sempre efficienti, quindi, potrà essere necessario scrivere query manualmente, in questo caso sarà necessario conoscere la sintassi SQL. Il suo utilizzo ha quindi i suoi pro e contro.

1.3 Server

Per lo sviluppo del server si è utilizzato il design pattern architetturale MVC (Model-View-Controller) (Figura 5).

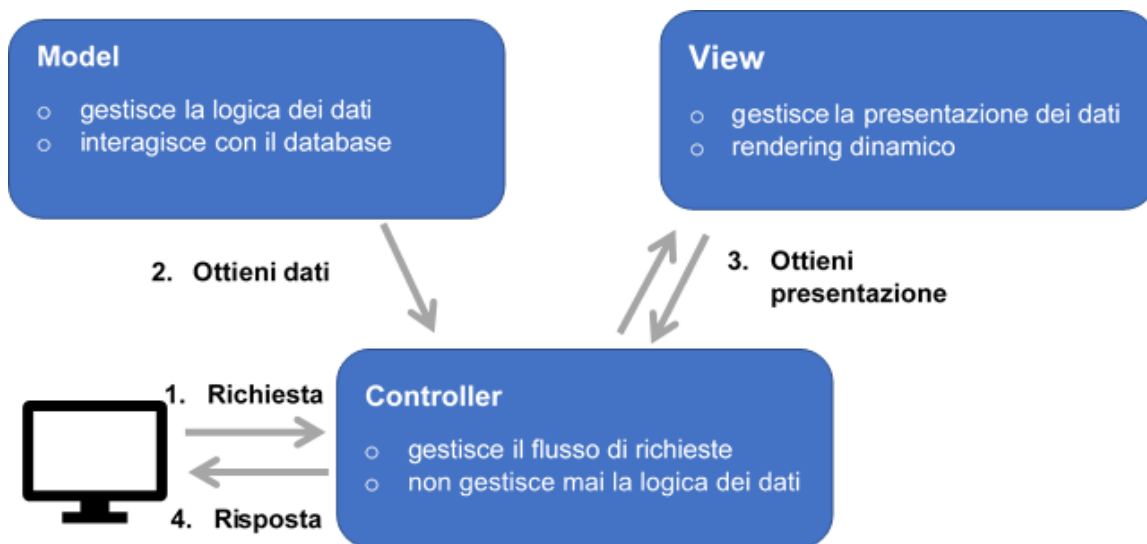


Figura 5 Design pattern architetturale MVC

Un'applicazione progettata utilizzando un pattern MVC è divisa in tre parti separate, che interpretano tre ruoli principali e comunicano tra loro: il model e il controller che costituiscono il lato server e la view, implementata sul lato client.

Il model è un oggetto che si occupa della gestione dei dati persistenti interagendo con il database, può ricevere i dati dal controller e inviare i dati alla view. Il controller rappresenta la spina dorsale dell'applicazione e contiene tutta la logica necessaria al suo funzionamento, come le azioni di aggiunta, modifica, cancellazione e il recupero dei dati dal model. La view si occupa della gestione dell'interfaccia, è ciò che l'utente vede e con cui interagisce. Attraverso la view gli utenti forniscono i dati che, dopo essere stati gestiti dal controller, vengono nuovamente reindirizzati alla view. Quanto concerne quest'ultima parte del pattern verrà meglio analizzata nel capitolo riguardante il client.

L'uso del modello MVC, separando la logica dell'applicazione e la gestione dell'interfaccia utente, rende l'applicazione maggiormente flessibile e riutilizzabile.

1.3.1 Diagramma dei componenti Node.js con Express.js

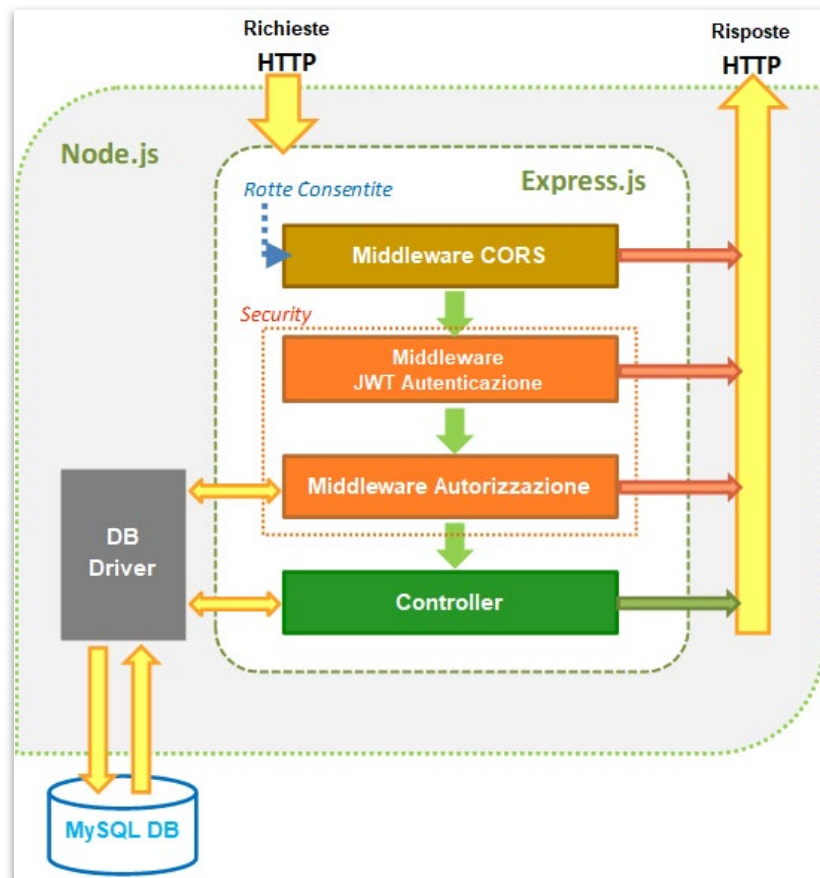


Figura 6 Diagramma dei componenti del back-end

Il diagramma in Figura 6 mostra il funzionamento del lato back-end dell'applicazione web.

Attraverso il framework Express e alle rotte definite dal router, le richieste http saranno controllate, e tutte abilitate dal 'Middleware CORS', prima di arrivare al livello di sicurezza. Il livello di sicurezza include: 'Middleware JWT Autenticazione' che verifica il token ricevuto dall'utente; 'Middleware Autorizzazione' che verifica se l'utente è autorizzato ad utilizzare la risorsa.

Se si verifica un determinato errore nei middleware, verrà inviato un messaggio come risposta http al client.

I controller interagiscono con il database MySQL tramite il DB driver e inviano la risposta http (token, informazioni sull'utente, calcolo dell'indicatore) al client.

1.3.2 Servizi API implementati

La parte server, come è stato detto precedentemente, ha lo scopo di fornire i seguenti servizi:

- **Registrazione utente**

URI: /api/auth/signup;

Metodo: POST;

Content-Type: Application/JSON;

Descrizione: salva i dati passati in input nel database, previa verifica degli stessi;

Input: come parametri alla chiamata http vengono passati i seguenti campi: nome, cognome, username, email, password;

Output: come risposta http viene ritornato un JSON nella seguente forma:

```
{
    message: "valore"
}
```

- **Login utente**

URI: /api/auth/signin;

Metodo: POST;

Content-Type: Application/JSON;

Descrizione: verifica se l'utente è registrato;

Input: come parametri alla chiamata http vengono passati i campi username e password;

Output: come risposta http viene ritornato un JSON nella seguente forma:

```
{
    id: "valore1",
    username: "valore2",
    email: "valore3",
    accessToken: "valore4",
    apiKey: "valore5"
}
```


- **Calcolo indicatore con autenticazione**

URI: /api/user/score;

Metodo: POST;

Content-Type: Application/JSON;

Descrizione: verifica se l'utente è loggato al sistema e previa verifica dei dati di input passati, restituisce il calcolo dell'indicatore di una località (via o paese) in un determinato raggio d'azione;

Input: come parametri alla chiamata http vengono passati oltre all'id, un JSON nella seguente forma:

```
{
    street_number: 'valore1',
    route: 'valore2',
    locality: 'valore3',
    administrative_area_level_2: 'valore4',
    country: 'valore5',
    postal_code: 'valore6',
    radius: valore7
}
```

Output: come risposta http viene ritornato un JSON nella seguente forma:

```
{
    total_score: valore1,
    address: 'valore2',
    population: valore3,
    distance: valore4,
    radius: valore5,
    coords: { lat: valore6, lng: valore7 },
    index: {
        bank: {
            nome: 'economia',
            score: valore8,
            places: [],
        },
        restaurant: {
            nome: 'ristorazione',
            score: valore9,
            places: [],
        },
        police: {
            nome: 'sicurezza',

```

```

        score: valore10,
        places: [],
    },
    gym: {
        nome: 'ginnastica',
        score: valore11,
        places: [],
    },
    hospital: {
        nome: 'salute',
        score: valore12,
        places: [],
    },
    school: {
        nome: 'istruzione',
        score: valore13,
        places: [],
    }
},
success: { status: valore14, error: valore15, message: valore16 }
}

```

- **Calcolo indicatore con API_KEY**

URI: /api/user/score;

Metodo: GET;

Content-Type: Application/JSON;

Descrizione: verifica se l'utente possiede una chiave API valida e previa verifica dei dati di input passati, restituisce il calcolo dell'indicatore di una località (via o paese) in un determinato raggio d'azione;

Input: come parametri alla chiamata http vengono passati i seguenti campi:

```
street_number="valore1"&route="valore2"&locality="valore3"&administrative_area_level_2="valore4"&country="valore5"&postal_code="valore6"&radius="valore7"&apikey="valore8"
```

Output: come risposta http viene ritornato un JSON nella seguente forma:

```

{
    total_score: valore1,
    address: 'valore2',
    population: valore3,
    distance: valore4,
    radius: valore5,

```

```
coords: { lat: valore6, lng: valore7 },
index: {
  bank: {
    nome: 'economia',
    score: valore8,
    places: [],
  },
  restaurant: {
    nome: 'ristorazione',
    score: valore9,
    places: [],
  },
  police: {
    nome: 'sicurezza',
    score: valore10,
    places: [],
  },
  gym: {
    nome: 'ginnastica',
    score: valore11,
    places: [],
  },
  hospital: {
    nome: 'salute',
    score: valore12,
    places: [],
  },
  school: {
    nome: 'istruzione',
    score: valore13,
    places: [],
  }
},
success: { status: valore14, error: valore15, message: valore16 }
}
```

1.3.3 Struttura

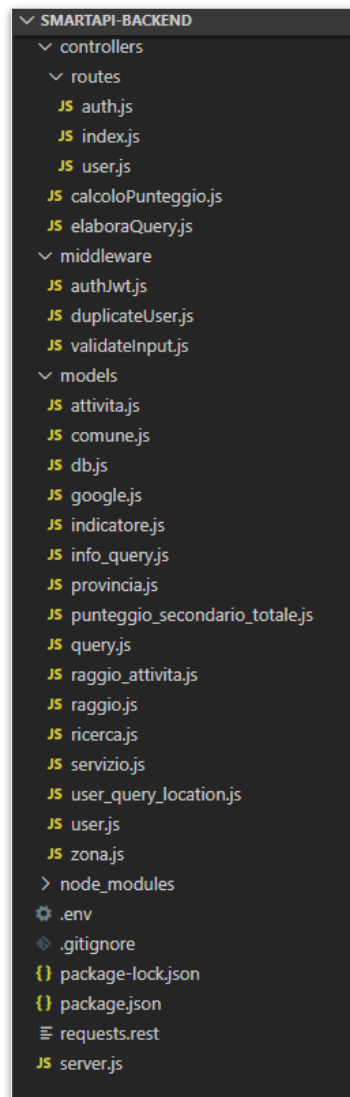


Figura 7 File e cartelle del server

Il diagramma in Figura 6 spiega la struttura del back-end che ha portato alla realizzazione del progetto cos  come in Figura 7.

L'applicazione consiste in diverse cartelle e file per implementare la logica dell'applicazione in modo che la struttura rimanga semplice. I models, i controllers (che racchiudono anche le routes) e i middleware dell'applicazione sono collocati in modo da seguire l'architettura MVC. La directory 'node_modules' contiene tutte le dipendenze richieste dal lato server. Il file '.gitignore'   costituito da tutti i file e le cartelle

non tracciati come `node_modules` e lo stesso file `.env` che contiene tutte le variabili d'ambiente utili per la configurazione del database e altre chiavi API.

I models sono stati creati per definire una struttura standard per le tabelle che vengono memorizzate nel database e da cui bisogna recuperare le informazioni. Ogni model è stato esportato in modo tale da poter essere utilizzato in altre parti dell'applicazione.

La cartella `'routes'` contiene tutta la logica lato server dell'applicazione e le API così come il routing sono implementate in questa sezione. Il file `'auth.js'` contiene la logica e il percorso per tutte le attività relative all'autenticazione degli utenti, come la registrazione di un utente nel database (signup) e la verifica delle informazioni dell'utente (signin); il file `'user.js'` contiene la logica e il percorso per le attività che può svolgere un utente, come il calcolo dell'indicatore, e si occupa di verificare ogni volta che un utente cerca di accedere all'applicazione.

1.3.4 Installazione delle dipendenze

La realizzazione del back-end in Node.js ha visto l'utilizzo delle dipendenze che si ritroveranno nella cartella `'node_modules'` e sono le seguenti:

- `express`
- `bcrypt`
- `jsonwebtoken`
- `mysql2`
- `cors`
- `node-fetch`
- `uuid`

In ambito di sviluppo sono state inoltre utilizzati:

- `nodemon`
- `dotenv`

L'installazione degli stessi è stata eseguita tramite il package manager NPM

```
npm install express
```

```
npm install bcrypt
```

```
npm install jsonwebtoken
npm install mysql2
npm install cors
npm install node-fetch
npm install uuid
npm install --save-dev nodemon dotenv
```

Nella Figura 8 il file 'package.json' successivamente generato.

```
"dependencies": {
  "bcrypt": "^4.0.1",
  "cors": "^2.8.5",
  "ejs": "^3.0.1",
  "express": "^4.17.1",
  "express-ejs-layouts": "^2.5.0",
  "jsonwebtoken": "^8.5.1",
  "mysql2": "^2.1.0",
  "node-fetch": "^2.6.0",
  "uuid": "^7.0.2"
},
"devDependencies": {
  "dotenv": "^8.2.0",
  "nodemon": "^2.0.2"
}
```

Figura 8 package.json server

Il nodo principale del server che si fa carico di configurare il middleware CORS e il framework Express in modo tale da poter gestire il resto dell'applicazione è definita dal file server.js, in Figura 9.

La configurazione del framework Express prevede la definizione delle rotte da intraprendere per le richieste http e configura quindi il server http per l'ascolto sulla porta definita dall'ambiente o 3000 come predefinita.

```

JS server.js > ...
1  if (process.env.NODE_ENV !== 'production') {
2    require('dotenv').config();
3  }
4
5  const express = require('express');
6  const cors = require('cors');
7  const app = express();
8
9  const indexRouter = require('./controllers/routes/index');
10 const authRouter = require('./controllers/routes/auth');
11 const userRouter = require('./controllers/routes/user');
12
13 app.use(express.json());
14 app.use(express.urlencoded({ extended: false }));
15 app.use(cors());
16
17 app.use('/', indexRouter);
18 app.use('/api/auth', authRouter);
19 app.use('/api/user', userRouter);
20
21 const db = require('./models/db.js');
22
23 checkMysqlConnection()
24   .then(() => console.log('Connected to Mysql'))
25   .catch(error => console.log(error.message));
26
27
28 app.listen(process.env.PORT || 3000);
29
30 function checkMysqlConnection() {
31   return new Promise((resolve, reject) => {
32     resolve(db.query("SELECT 1"));
33   });
34 }

```

Figura 9 server.js nodo principale del server

1.3.5 Driver MySQL

Attraverso il driver 'mysql' della libreria mysql2 è stato eseguito il collegamento al database. Per questo collegamento è stato definito nel file 'models/db.js' un pool, atto a ridurre il tempo di connessione al server MySQL, che viene esportato per renderlo disponibile all'interno dell'applicazione (Figura 10). Il pool di connessione permette di riutilizzare una connessione precedente che verrà lasciata aperta invece di chiudersi nell'immediato, migliorando così la latenza delle query in quanto si evitano tutti i sovraccarichi che si verificano quando si stabilisce una nuova connessione. Il pool non

crea tutte le connessioni in anticipo, ma le crea su richiesta fino al raggiungimento del limite di connessione.

Le query con `mysql2` sono state eseguite con il metodo `'execute()'`. Questo metodo permette di utilizzare le dichiarazioni preparate ovvero una caratteristica usata per eseguire le stesse (o simili) dichiarazioni SQL ripetutamente con alta efficienza. Le istruzioni preparate funzionano fondamentalmente su più step: viene creato ed inviato al database un modello di dichiarazione SQL dove alcuni valori, chiamati parametri, vengono non specificati ed etichettati con il simbolo `"?"`; il database analizza, compila ed esegue l'ottimizzazione delle query sul modello di dichiarazione SQL e memorizza il risultato senza eseguirlo; in un secondo momento, l'applicazione lega i valori ai parametri e il database esegue la dichiarazione.

L'applicazione può eseguire le dichiarazioni tutte le volte che vuole con valori diversi. Rispetto all'esecuzione diretta delle istruzioni SQL, le istruzioni preparate presentano tre vantaggi principali:

- riducono il tempo di analisi, poiché la preparazione della query viene eseguita una sola volta (anche se la dichiarazione viene eseguita più volte);
- i parametri vincolati riducono al minimo la larghezza di banda al server in quanto è necessario inviare ogni volta solo i parametri, e non l'intera query;
- sono utili contro gli attacchi di tipo SQL Injection, perché i valori dei parametri, che vengono trasmessi in un secondo momento con un protocollo diverso, non hanno bisogno di un corretto escape.

```
models > JS dbjs > ...
1  const mysql = require('mysql2');
2
3  const pool = mysql.createPool({
4    host: process.env.DB_HOST,
5    port: process.env.DB_PORT,
6    user: process.env.DB_USER,
7    password: process.env.DB_PASSWORD,
8    database: process.env.DB_SCHEMA,
9    namedPlaceholders: true,
10 });
11
12 const db = pool.promise();
13
14 module.exports = db;
```

Figura 10 Creazione pool connessione con `mysql2`

1.3.6 Autenticazione e autorizzazione con JWT

Json Web Token (JWT) è sicuramente il modo migliore per creare un canale di comunicazione con l'utente perché non lo obbliga ad effettuare il login ad ogni richiesta.

In particolare, permette di cifrare, tramite un codice segreto, le informazioni dell'utente che ha eseguito l'autenticazione al server, creando un token che verrà inviato all'utente.

Il client si occuperà di memorizzare nell'area di lavoro (nel caso specifico il Local Storage) del browser il token ricevuto così da poter eseguire ogni nuova richiesta al server antepoendo questo token. Il server solo grazie al codice segreto sarà in grado di verificare l'autenticità del token, precedentemente inviato all'utente, senza dover mantenere una sessione per ogni utente nella propria area di lavoro.

Il JWT è costituito da una struttura standard: header.payload.signature e il tipicamente allega il JWT all'header di autorizzazione il prefisso "Bearer": `Authorization: Bearer [header].[payload].[signature]`. In Figura 11 viene mostrato in sintesi il funzionamento di JWT.

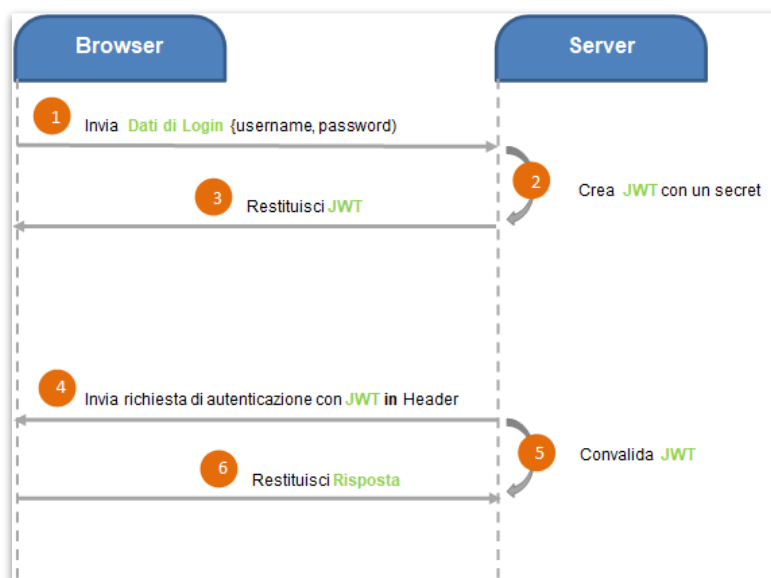


Figura 11 Schema rappresentativo di JWT

In Figura 12 un diagramma di flusso per mostrare in che modo viene eseguita la registrazione degli utenti, il login e l'autorizzazione alle risorse.

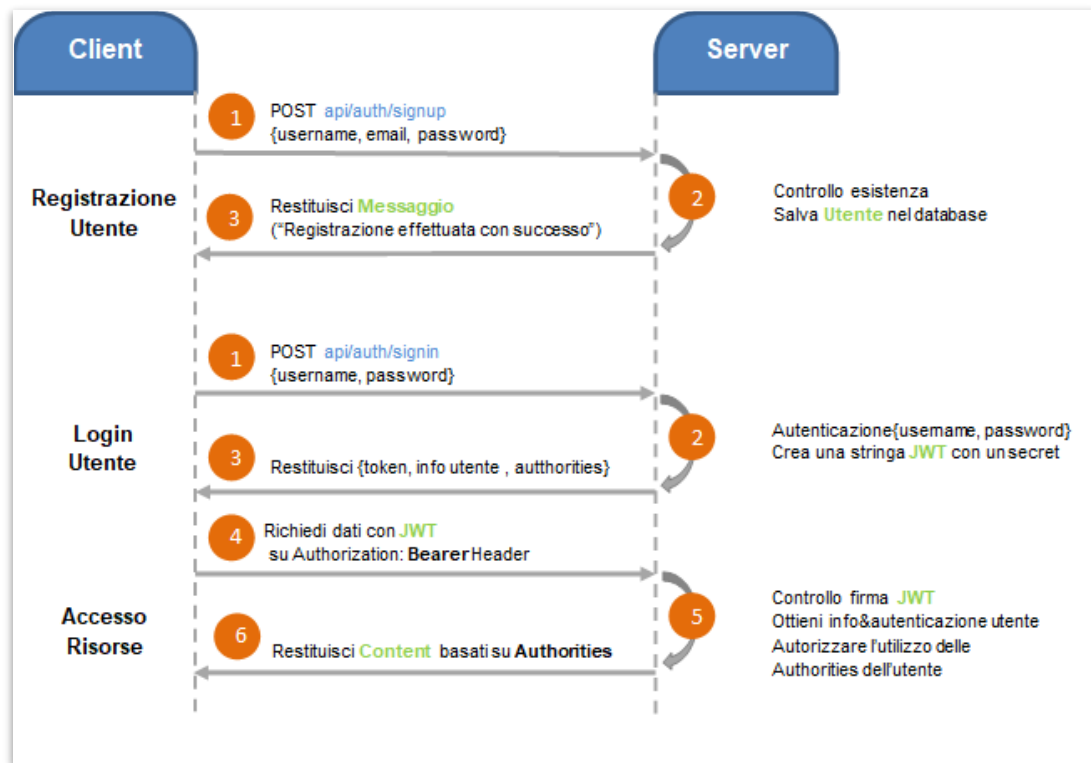


Figura 12 Flusso del processo di registrazione utente, login utente e autorizzazione

1.3.7 Middleware Validazione dati input

La risorsa accessibile dagli utenti è quella atta al calcolo dell'indicatore, ovvero `/api/user/score`, raggiungibile mediante due tipologie di richieste http, POST e GET. Il middleware per la validazione dei dati di input, `middleware/validateInput.js`, è fondamentale in quanto si occupa di validare i dati richiesti dal client e allo stesso tempo di verificare se i dati di input dell'utente corrispondono ad una nuova ricerca, ad una ricerca già effettuata in precedenza (anche da altri utenti), o ad una ricerca scaduta.

Il middleware si occupa di gestire entrambe le richieste http e per questo motivo controlla, in caso di richiesta GET, se tra i parametri di input della stessa è presente una chiave API valida. In seguito, verranno controllati in ordine di necessità tutti i restanti parametri di input passati alla richiesta e costruito l'oggetto query che avrà il

necessario per poter effettuare il calcolo dell'indicatore. In caso di errori verrà inviato immediatamente un messaggio esplicativo al client. È opportuno ricordare che l'applicazione web gestisce soltanto località nel territorio italiano e che la ricerca dovrà necessariamente contenere come parametri il nome del comune, il nome della provincia e il raggio di cui si vuole conoscere l'indicatore. Questa ricerca potrebbe essere già stata effettuata, in tal caso verrà interrogato il database per conoscere sia se è presente l'id della `user_query_location` sia se è presente l'id del raggio per la stessa. Bisognerà inoltre confrontare la data di scadenza della tabella indicatore. Questa scadenza è stata determinata in quanto gli elementi alla base del calcolo possono subire delle variazioni nel tempo che andrebbero a modificare i valori del calcolo finale, pertanto si è scelto di fissarla a sei mesi dal primo inserimento. Nel caso in cui fosse necessario quindi l'aggiornamento dei dati viene fissato il flag `statoQuery` dell'oggetto query su `'aggiorna'`, in caso contrario il flag sarà `'leggi'`. Nel caso in cui la ricerca è da effettuare, verrà verificato se per questa è stato specificato anche l'indirizzo (strada con o senza civico) e verrà quindi effettuata una richiesta all'API di Google per ricevere le coordinate, in caso contrario verranno prelevate dal database e aggiunte all'oggetto query. Questa scelta è stata fatta sia per limitare le richieste a Google, sia per poter agevolmente sostituire, in un secondo momento, Google con altri servizi o metodi più efficienti. Lo `statoQuery` verrà impostato su `'scrivi'`.

1.3.8 Controller `ElaboraQuery`

Per gestire situazioni che potrebbero rendere il server unresponsive, come in casi di calcoli complessi o che bloccano il thread principale, in Node.js è stato aggiunto il supporto al multi thread, con il modulo `'worker_threads'`, un pacchetto che permette di creare applicazioni multi thread pienamente funzionali. Questo modulo è stato sfruttato nella costruzione dell'applicazione web per generare ad ogni richiesta da parte degli utenti un nuovo thread. Nel controller `elaboraQuery` viene creato l'oggetto ricerca, ovvero l'output in uscita dal thread, ed eseguite delle funzioni a seconda dello `statoQuery` dell'oggetto query: se corrisponderà a `'leggi'` il controller si interfacerà con il model ricerca per ottenere le informazioni dal database; se corrisponderà ad `'aggiorna'` o `'scrivi'`, verrà prima effettuato il calcolo del punteggio, in seguito verranno eseguite le operazioni di aggiornamento e scrittura nel database.

1.3.9 Controller Calcolo punteggio

In merito al calcolo del punteggio questo non subisce variazione nella web app rispetto all'app Android. La differenza sostanziale sta nel controller che al posto di ricavare il punteggio secondario da tabelle .xlsx lo ricava dalla tabella `punteggio_secondario` del database attraverso il model `punteggio_secondario`. Nella tabella risultano già inseriti i risultati, ovvero la media dei domini, dei calcoli necessari. Basterebbe così aggiornare esclusivamente la tabella di riferimento limitando così le dipendenze e la lettura di diversi file per ogni nuova ricerca.

Nel dettaglio si è proseguito ai calcoli del punteggio nella seguente maniera:

- **Il criterio di valutazione è basato sulla somma di due punteggi:**
 - Punteggio primario (PR): il prodotto del coefficiente x e il numero delle attività rilevate nell'area d'esame.
 - Punteggio secondario (PS): media dei domini analizzati dal dataset dell'ISTAT assegnati proporzionalmente alla distanza Provincia – Comune.

Il numero delle attività rilevabili e mostrate sulla mappa presenta un limite di 60 attività dovuto alle politiche di Google; per ovviare al problema del limite massimo sono state suddivise le attività mostrabili per ogni settore in tre fasce, create in base alla stima della popolazione totale della località cercata, e ad ogni attività è stato assegnato il coefficiente x come mostrato in tabella.

Fascia	Popolazione (p)	N° Attività	Coefficiente (x)
1	$p < 100.000$	20	0.25
2	$100.000 < p < 1.000.000$	40	0.125
3	$p > 1.000.000$	60	0.0833

Per quanto riguarda il PS non sono stati ritrovati dataset per i tutti i settori, per cui ai settori ginnastica e ristorazione è stato riconosciuto un PS pari a 0.00; per gli altri, a partire dai dataset .xls utilizzati, è stata ottenuta la media dei domini (**md**) in base a

valori definiti da uno schema stabilito dallo sviluppatore stesso. Per ottenere il PS la media è stata poi assegnata proporzionalmente alla distanza in chilometri (Km) tra Provincia e Comune, anch'essa suddivisa in fasce, come in tabella.

Distanza Provincia-Comune (d)	Punteggio Secondario
$d = 0$	md
$5 \text{ km} < d < 15 \text{ km}$	$2/3 \text{ md}$
$15 \text{ km} < d < 25 \text{ km}$	$1/3 \text{ md}$
$d > 25 \text{ km}$	0

- **Il punteggio totale è dato dalla media della somma dei PR+PS di ogni singolo settore.**

1.3.10 Google API

Si è fatto uso dell'API di Google unicamente per quanto riguarda le coordinate e le attività settore per settore, necessarie per il calcolo del punteggio totale, di uno specifico set di latitudine e longitudine. In particolare, nel file 'models/google.js', tramite la libreria 'node-fetch', sono state eseguite delle richieste GET per ottenere un oggetto JSON alle API 'geocode' per le coordinate e 'place' per le attività. Tutte necessitano della chiave API precedentemente ottenuta da Google e sono di seguito indicate:

- **coordinate**

```
https://maps.googleapis.com/maps/api/geocode/json?address=NOME_LOCALITA&key=API_KEY
```

- **attività**

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=LATITUDINE,LONGITUDINE&radius=RAGGIO&type=SETTORE&key=API_KEY
```

Bisogna precisare che la singola richiesta per le attività prevede che vengano restituiti un massimo di venti risultati. Nel caso in cui i risultati fossero maggiori, a questi venti risultati se ne aggiungerebbe un ultimo rappresentato da un token. Per poter ottenere i risultati ulteriori sarà quindi necessario effettuare nuovamente la richiesta (fino ad un

massimo di tre) aggiungendo il parametro '&pagetoken=', con il token ottenuto precedentemente, all'URI delle attività. Queste operazioni di richiesta a Google non vengono effettuate nell'immediato, ma necessitano dei tempi tecnici, stabiliti dallo stesso, che ne influenzano la risposta per il risultato del calcolo del punteggio. Diversamente quando la richiesta è già stata fatta e quindi memorizzata nel database i tempi si riducono considerevolmente.

1.3.11 Builder Pattern

Un design pattern è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente". Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software, ancor prima della definizione dell'algoritmo risolutivo della parte computazionale. Conoscere i design pattern è fondamentale per qualsiasi progettista in quanto facilita l'attività di progettazione stessa, ne favorisce la riusabilità e la manutenibilità del codice.

Esistono diversi design pattern a cui riferirsi per la progettazione, nello sviluppo del back-end nel caso particolare è stato utilizzato il Builder pattern. Questo è uno dei migliori modelli di design creativo per creare oggetti complessi senza complicare i costruttori o il codice.

Nella programmazione ad oggetti tale pattern separa la costruzione in un oggetto complesso dalla sua rappresentazione, cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. In questo modo l'algoritmo per la creazione di un oggetto complesso è indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate. Ciò ha l'effetto immediato di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla corretta costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti. Questo è particolarmente utile quando bisogna assicurarsi che un oggetto sia valido prima di istanziarlo, e non si vuole che la logica di controllo appaia nei costruttori degli oggetti.

Un builder permette inoltre di costruire un oggetto passo-passo, come si è fatto per la creazione dell'oggetto query nel middleware Verifica dati input.

3.4 Client

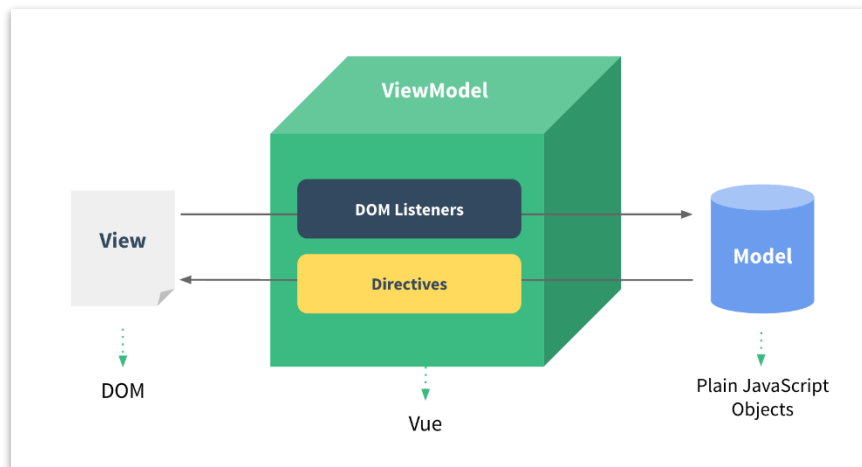


Figura 13 Design Pattern MVVM [Fonte: vuejs.org]

La componente front-end, influenzata dall'uso del framework Vue.js, sfrutta il design pattern Model-View-ViewModel (MVVM), una declinazione del Model-View-Controller (MVC), i cui componenti principali sono:

- il Modello (Model) che rappresenta l'implementazione del dominio dati gestito dall'applicazione (come per la 'M' in MVC);
- la Vista (View) che rappresenta il componente grafico renderizzato all'utente, composta da HTML e CSS;
- Il Modello per la Vista (ViewModel) che rappresenta il collante tra i precedenti componenti e fornisce alla view sia i dati in un formato consono alla presentazione che il comportamento di alcuni componenti dinamici.

La grossa differenza sta nelle componenti del Controller e del ViewModel: il primo è di fatto una porzione di codice che esegue particolari logiche di business (grazie al Model) e che ritorna una View da mostrare all'utente; il secondo rappresenta di fatto un modello parallelo al Model, che viene direttamente collegato alla View e descrive il comportamento di quest'ultima con funzioni associate, ad esempio al click di un elemento. Il Controller esegue logiche di business prima del rendering della View, il ViewModel definisce il comportamento dell'applicazione a runtime.

In Vue.js è interessante l'uso delle direttive ovvero speciali attributi HTML, che iniziano con prefisso `v-` e che permettono di aggiungere comportamenti a livello di presentazione e sono in grado di svolgere vari compiti, come la ripetizione di operazioni (`v-for`) ed il collegamento con il model (`v-model`).

Come è stato già descritto in precedenza uno dei vantaggi del framework Vue.js è la sua flessibilità che permette di decidere quali componenti integrare nel progetto. Al fine di poter gestire l'autenticazione sono stati utilizzati in particolare le componenti Vuex, Vue Router e VeeValidate, in modo tale da supportare l'autenticazione con JWT, vista nel back-end.

Lo sviluppo del progetto non si è focalizzato sulla componente front-end, pertanto non è stato realizzato uno stile particolare per la presentazione. Si è comunque resa l'applicazione responsive, ovvero si è fatto in modo che il layout si adattasse alla dimensione del display in cui viene eseguita, affiancando Vue.js al framework Bootstrap.

3.4.1 Diagramma dei componenti Vue App con Vuex & Vue Router

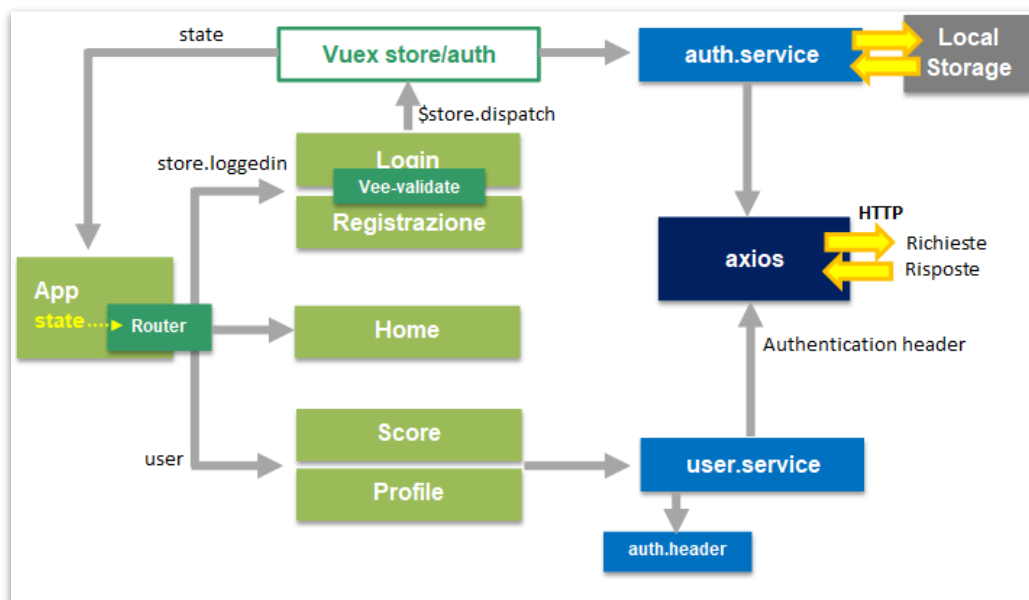


Figura 14 Diagramma dei componenti del front-end

Il diagramma in Figura 21 mostra i componenti che costituiscono la Vue application:

Componente App: è un contenitore con l'oggetto Router che riceve lo stato di app da Vuex store/auth che permette la gestione dello stato e funziona da contenitore centralizzato per tutti i componenti con regole che garantiscono che lo stato possa essere mutato solo in modo prevedibile. Verificato lo stato, la barra di navigazione ne visualizzerà il contenuto a questo riferito. Il componente App passa lo stato anche ai suoi componenti figli.

Componenti Login & Registrazione: contengono il modulo per l'invio dei dati con il supporto di vee-validate che permette di validare i dati di input dell'utente. Sarà chiamata la funzione dispatch() di Vuex store per effettuare le azioni di login e registrazione.

Componente Vuex store/auth: chiama i metodi auth.service, che usano axios per fare richieste http, e memorizza negli stessi metodi il JWT, che verrà ricevuto o prelevato dal Local Storage del browser.

Componente Home: rappresenta la schermata pubblica a tutti i visitatori.

Componente Profile: ottiene i dati dell'utente dal suo componente genitore e visualizza le informazioni dell'utente.

Componente Score: ottiene i dati dell'utente dal suo componente genitore e permette di calcolare l'indicatore. In questo componente, usiamo user.service per ottenere risorse protette dalle API fornite dal back-end. Il modulo user.service per poter effettuare la richiesta http aggiunge JWT all'HTTP Authorization header, per mezzo della funzione 'auth-header()', che restituirà un oggetto contenente il JWT dell'utente attualmente connesso da Local Storage.

3.4.2 Struttura

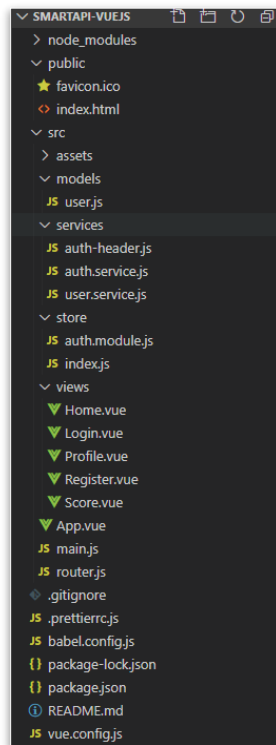


Figura 15 Cartelle e file della Vue Application

Il diagramma dei componenti visto precedentemente spiega la struttura del front-end realizzato, composto dalle cartelle e dai file visibili nella Figura 15.

3.4.3 Installazione ed utilizzo dei componenti

La realizzazione del front-end ha visto l'utilizzo dei seguenti moduli:

- vue: 2.6.11
- vue-router: 3.1.6
- vuex: 3.1.3
- axios: 0.19.2
- vee-validate: 2.2.15
- bootstrap: 4.4.1
- vue-fontawesome: 0.1.7
- vue2-google-maps: 0.10.7
- vue-google-charts: 0.3.2

L'installazione degli stessi è stata eseguita tramite il package manager NPM

```
npm install vuex
```

```
npm install vue-router
```

```
npm install vee-validate@2.2.15
```

```
npm install axios
```

```
npm install bootstrap jquery popper.js
```

```
npm install @fortawesome/fontawesome-svg-core @fortawesome/free-solid-svg-  
icons @fortawesome/vue-fontawesome
```

```
npm install vue2-google-maps vue-google-charts
```

Nella Figura 16 il file 'package.json' successivamente generato.

```
"dependencies": {  
  "@fortawesome/fontawesome-svg-core": "^1.2.28",  
  "@fortawesome/free-solid-svg-icons": "^5.13.0",  
  "@fortawesome/vue-fontawesome": "^0.1.9",  
  "axios": "^0.19.2",  
  "bootstrap": "^4.4.1",  
  "core-js": "^2.6.11",  
  "jquery": "^3.4.1",  
  "popper.js": "^1.16.1",  
  "vee-validate": "^2.2.15",  
  "vue": "^2.6.11",  
  "vue-google-charts": "^0.3.2",  
  "vue-router": "^3.1.6",  
  "vue2-google-maps": "^0.10.7",  
  "vuex": "^3.1.3"  
},
```

Figura 16 Dipendenze Vue application

Il punto di partenza della Vue Application che si fa carico di importare tutti i componenti utili, è definita dal file `main.js` in Figura 17. In particolare: store per Vuex implementato nel file `'src/store'`; router per Vue Router implementato nel file `'src/router.js'`; bootstrap per la presentazione (CSS); vee-validate per la verifica del form di registrazione e di login; vue-fontawesome per le icone utilizzate nella barra di navigazione; vue2-google-maps per l'impiego dell'autocomplete e delle mappe di Google utilizzabili, come visto per le altre librerie di Google, solo dopo aver impostato una valida `API_KEY`.

```

src > JS main.js > ...
1  import Vue from 'vue';
2  import App from './App.vue';
3  import { router } from './router';
4  import store from './store';
5  import 'bootstrap';
6  import 'bootstrap/dist/css/bootstrap.min.css';
7  import VeeValidate from 'vee-validate';
8  import Vuex from 'vuex';
9  import * as VueGoogleMaps from 'vue-google-maps'
10 import { library } from '@fortawesome/fontawesome-svg-core';
11 import { FontAwesomeIcon } from '@fortawesome/vue-fontawesome';
12 import {
13   faHome,
14   faUser,
15   faUserPlus,
16   faSignInAlt,
17   faSignOutAlt
18 } from '@fortawesome/free-solid-svg-icons';
19
20 Vue.use(VueGoogleMaps, {
21   load: {
22     key: process.env.API_KEY,
23     libraries: 'places',
24   }
25 })
26
27 library.add(faHome, faUser, faUserPlus, faSignInAlt, faSignOutAlt);
28
29 Vue.config.productionTip = false;
30
31 Vue.use(VeeValidate);
32 Vue.component('font-awesome-icon', FontAwesomeIcon);
33
34 Vue.use(Vuex);
35
36 new Vue({
37   router,
38   store,
39   render: h => h(App)
40 }).$mount('#app');

```

Figura 17 main.js Vue Application

Per rendere disponibili tutti i percorsi nell'applicazione, in aggiunta al main.js, è stato definito, nel file routes.js, l'oggetto route, che rappresenta la condizione del percorso attivo e contiene le informazioni analizzate dell'URL corrente e i record di rotta corrispondenti all'URL. L'oggetto route è immutabile e ogni navigazione riuscita darà luogo a un nuovo oggetto. Il metodo 'beforeEach()' dell'oggetto è stato utilizzato per impedire l'accesso ai percorsi di cui non si ha l'autorizzazione ad accedervi, controllandone lo stato.

3.4.4 Autenticazione e autorizzazione con JWT

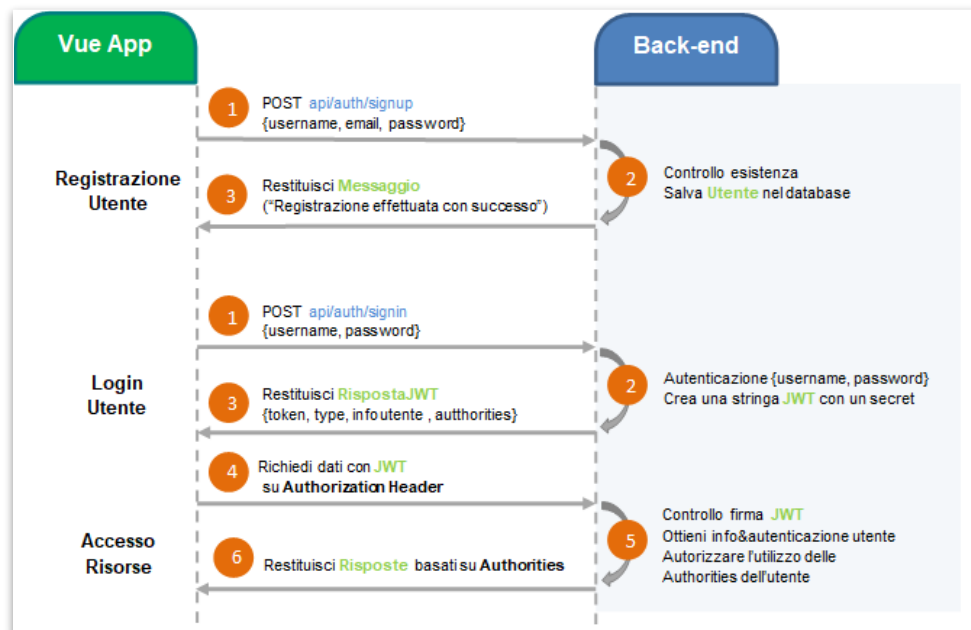


Figura 18 Flusso per la registrazione e il login dell'utente

L'autenticazione con JWT è stata realizzata chiamando i due endpoint:

- `POST api/auth/signup` per la registrazione dell'utente
- `POST api/auth/signin` per il login utente

L'autorizzazione, che permette l'accesso a risorse protette, è stata invece realizzata aggiungendo JWT all'Authorization Header prima di inviare la richiesta http. Nella Figura 18 è possibile osservare una panoramica delle richieste e delle risposte della Vue Application.

L'implementazione degli stessi è stata effettuata nella cartella `src/services` dove sono stati creati i servizi: autenticazione e dati.

Il servizio di autenticazione fornisce tre importanti metodi che, tranne per il 'logout', utilizzano axios per le richieste POST e per le risposte http.

- **login(user):** `POST {username, password}` che salva JWT nella memoria locale in caso positivo;
- **logout():** rimuove JWT dalla memoria locale

- **register(user):** POST {username, email, password}

Il servizio dati definisce le richieste che necessitano di accedere a risorse protette e si occupa sia di aggiungere dei possibili oggetti alla richiesta da effettuare al back-end e sia di aggiungere all'header (della richiesta http) l'autorizzazione JWT. Per far ciò è stata creata una funzione di aiuto, chiamata 'authHeader()'. La funzione controlla il Local Storage del browser dell'utente e restituisce l'accessToken (JWT) nel caso ci sia un utente registrato, altrimenti restituisce un oggetto vuoto.

3.4.5 Componente Vuex store/auth

Il componente Vuex, presente nella cartella 'src/store' contiene:

- state: { status, user }
- actions: { login, logout, register }
- mutations: { loginSuccess, loginFailure, logout, registerSuccess, registerFailure }

Al centro di questo componente c'è `store` che è fondamentalmente il contenitore dello stato dell'applicazione. Ci sono due motivazioni che rendono un Vuex store diverso da quello che può sembrare un semplice oggetto globale, ovvero:

- reattività: quando i componenti Vue recuperano lo stato dallo store o lo stato cambia, si aggiornano in modo reattivo ed efficiente.
- Mutabilità non diretta dello stato: l'unico modo per cambiare lo stato di uno store è commettere (commit) esplicitamente delle mutazioni. Ciò assicura che ogni cambiamento di stato lasci una traccia e fornisce strumenti che permettono di capire meglio le applicazioni.

3.4.6 Componenti Login & Registrazione

I componenti Login e Registrazione, invece di utilizzare direttamente axios o AuthService, funzionano grazie al componente Vuex Store che gli permette di:

- ottenere lo stato, con `this.$store.state.auth`;
- fare richiesta, eseguendo il metodo: `this.$store.dispatch()`.

Per rendere il codice chiaro e facile da leggere, è stato definito il modello User, nella cartella 'src/models'.

Login

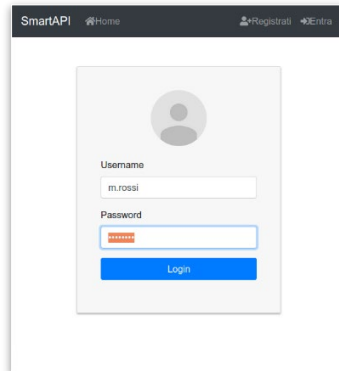


Figura 19 Pagina addetta al login

La pagina che contiene il modulo con nome utente e password per loggarsi al sistema è `Login.vue`, visibile in Figura 19, `vee-validate` si occupa di convalidare i dati inseriti prima di inviare il modulo così da mostrare un messaggio di errore nel caso in cui un campo non è valido. Viene controllato lo stato dell'utente connesso usando Vuex Store (`this.$store.state.auth.status.loggedIn`) e se lo stato è vero (utente loggato), Vue Router indirizzerà l'utente alla pagina predisposta per il calcolo dell'indicatore.

Registrazione

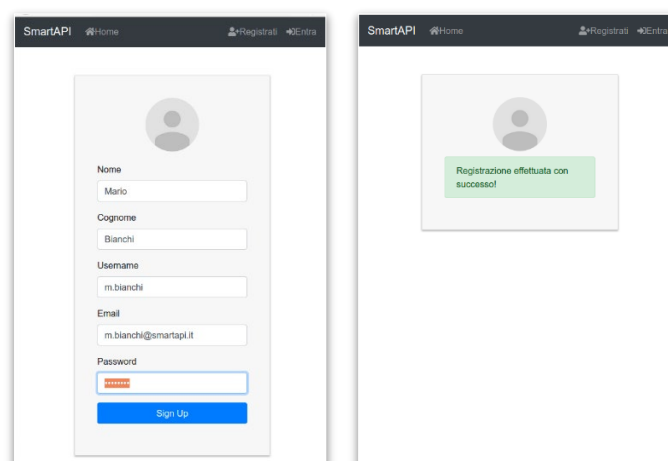


Figura 20 Pagina addetta alla registrazione

La pagina che è simile a quella del Login e permette invece di registrarsi al sistema è `Register.vue`, visibile in Figura 20. Per la validazione del modulo, però sono stati definiti dei dettagli in più:

- `username`: è un campo di tipo 'testo' richiesto, con un minimo di carattere di 3 e un massimo di 20;
- `email`: è un campo di tipo 'email' richiesto, con un massimo di 50 caratteri;
- `password`: è un campo di tipo 'password' richiesto, con un minimo di carattere di 6 e un massimo di 40;

L'invio del modulo viene eseguito con il metodo Vuex 'auth/register'.

3.4.7 Componente Profile

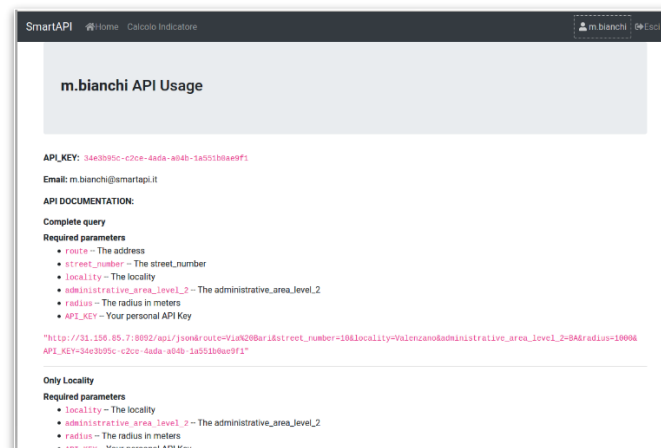


Figura 21 Pagina relativa al componente Profile

La pagina `profile`, visibile nella Figura 21, riceve l'utente corrente da Vuex Store, se l'utente è loggato viene mostrata la chiave API dell'utente e le istruzioni per poter utilizzare il sistema in alternativa all'interfaccia creata nel componente `Score`, altrimenti si viene indirizzati alla pagina Home.

3.4.8 Componente Score

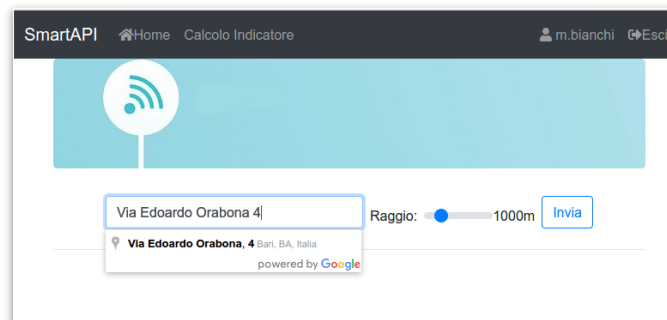


Figura 22 Pagina relativa al componente Score

La pagina `score` riceve l'utente corrente da Vuex Store e se l'utente è loggato viene mostrata la pagina per il calcolo dell'indicatore, altrimenti si viene indirizzati alla pagina Home. In questo componente, usiamo `user.service` per ottenere risorse protette dalle API fornite dal back-end, in particolare il metodo `postUserQuery()`.

Nella Figura 22 possiamo osservare il modulo che permette all'utente di cercare la località (via o paese), di cui si vuole conoscere l'indicatore, in un determinato raggio d'azione. Il modulo contiene un campo testuale per inserire la località, uno slider per selezionare il raggio d'azione che se non modificato corrisponde al valore 1000 metri, ed un pulsante, inizialmente disabilitato, per inviare la richiesta.

Per evitare l'immissione di dati non validi, in particolare per quanto riguarda la località, è stato utilizzato il componente `gmap-autocomplete` della libreria `vue2-google-maps`.

Il pulsante che verrà abilitato solo dopo che l'utente avrà selezionato un elemento dall'autocomplete esegue l'azione `handleQuery` che invierà l'oggetto JSON in **Errore**. **L'origine riferimento non è stata trovata.** al server.

```
toSend: {
  street_number: '', // civico
  route: '', // indirizzo
  locality: '', // città
  administrative_area_level_1: '', // regione
  administrative_area_level_2: '', // provincia
  country: '', // nazione
  postal_code: '', // cap
  radius: 1000 //raggio default
}
```

Figura 23 JSON inviato al server tramite POST

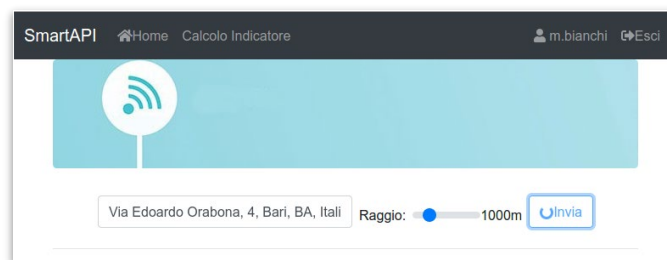


Figura 24 Stato del modulo dopo aver inviato la richiesta

In attesa di ricevere i dati dal server, uno spinner affiancherà il pulsante, come in Figura 24, e in caso di esito positivo il pulsante verrà nuovamente disabilitato e verranno restituite parte delle informazioni che riguardano l'indicatore calcolato per la località cercata: il raggio, la popolazione, la distanza (dalla provincia), le coordinate, il punteggio totale (l'indicatore) e un grafico a torta basato sul numero delle attività ricevute dei vari settori. Il grafico è stato realizzato grazie al componente GChart della libreria `vue-google-charts`.

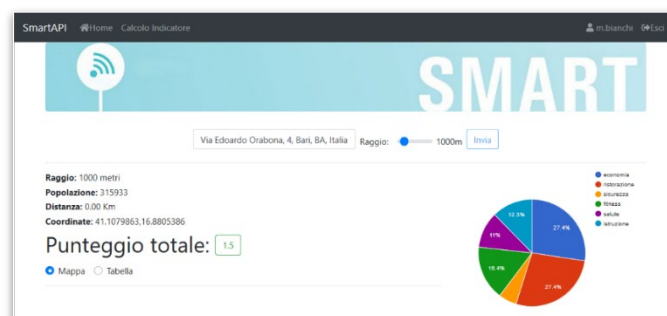


Figura 25 Risultato ottenuto dal server nello stato di default

La pagina sarà composta da due elementi attivi ovvero elementi che cambiano lo stato in base alla selezione, cioè: un pulsante di opzione che permette di selezionare il modo in cui si vorranno mostrare le attività ricevute, mappa o tabella, e un pulsante in corrispondenza del punteggio totale che oltre a mostrare l'indicatore complessivo permetterà di mostrare gli indicatori di tutti i settori.



Figura 26 Evento eseguito dopo il click del pulsante

L'evento associato al click del pulsante in corrispondenza del punteggio totale modificherà lo stato della variabile 'mostra', che nel suo stato vero, mostrerà gli indicatori dei vari settori, Figura 26. Ogni indicatore è a sua volta un pulsante associato ad un evento che modificherà lo stato di una variabile in modo tale da mostrare la mappa delle attività o la tabella in base alla selezione effettuata in precedenza, Figura 27. La mappa è stata realizzata grazie ai componenti `gmap-map`, `gmap-marker` e `gmap-info-window` della libreria `vue2-google-maps`, che si occupano rispettivamente di generare la mappa in base alle coordinate della località, di creare dei marker in base alle coordinate delle singole attività e di una icona personalizzata, e di creare la finestra di informazioni in base al nome delle singole attività che verrà mostrata nel caso in cui si seleziona un marker.

La combinazione delle direttive di `vue.js`: `v-for` e `v-if` rendono i cambiamenti di visualizzazione trasparenti all'utente.

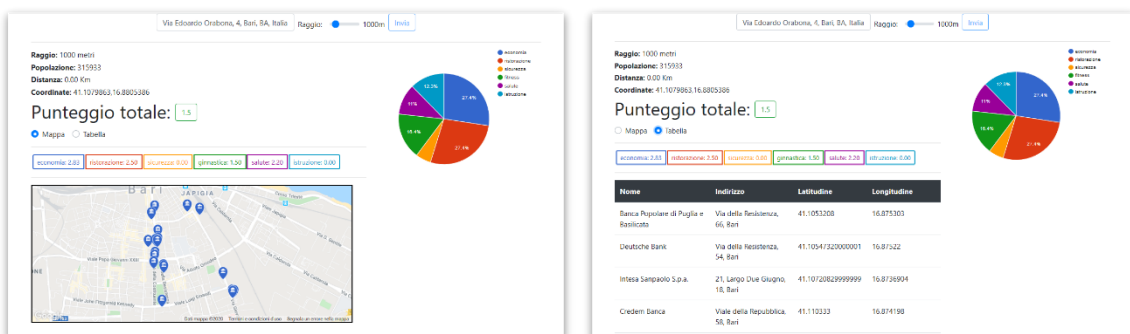


Figura 27 Evento eseguito dopo il click del pulsante economia

L'utente modificando il raggio d'azione potrà riattivare il pulsante di invio ed effettuare una nuova ricerca.

Se diversamente si è verificato qualche errore nel calcolo dell'indicatore, verrà mostrato un rettangolo in rosso subito sotto il modulo di ricerca, Figura 28

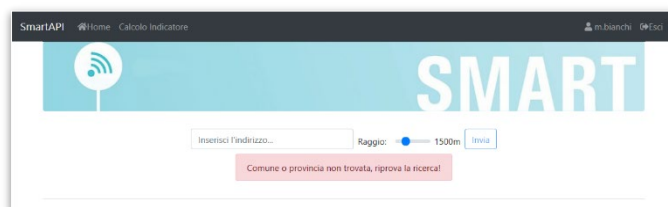


Figura 28 Esempio errore calcolo indicatore

3.4.9 Libreria vue2-google-maps

La libreria `vue2-google-maps` sviluppata da `xkjyeah` che racchiude i componenti utilizzati per validare la ricerca e per la creazione della mappa, essendo stata realizzata sulla base della libreria `Places` delle `API Maps JavaScript` offerta da Google, è utilizzabile solo previa creazione della propria `API_KEY`, l'accesso alle richieste non è supportato senza chiave ed è necessario abilitare la fatturazione su ciascuno dei progetti che ne hanno bisogno.

In particolare, il componente `gmap-autocomplete` permette quindi di dare alle applicazioni il comportamento di ricerca utilizzato nella barra di Google Maps, ovvero quando un utente inizia a digitare un indirizzo, il completamento automatico compilerà il resto. I campi dei dati del luogo richiesti influiscono sul costo di ogni richiesta per questo è stato specificato il solo campo `address_components` che consente di ricavare le informazioni utili da inviare al server: `street_number`, `route`, `locality`, `administrative_area_level_2`, `country`, `postal_code`.