

# GuidApp

A Service-Oriented based Touristic Guide



Daniele Berardini, **255683**  
Stefano Cortellessa, **254260**  
Davide Iacobelli, **254435**

Service-Oriented Software Engineering  
A.A. 2018-2019

# Indice

<b>1.</b>	<b>INTRODUCTION .....</b>	<b>3</b>
<b>2.</b>	<b>STATIC VIEW.....</b>	<b>4</b>
2.1	COMPONENT DIAGRAM .....	4
2.2	DEPLOYMENT DIAGRAM.....	5
2.3	DATABASE ER-DIAGRAM.....	6
<b>3.</b>	<b>DYNAMIC VIEW .....</b>	<b>7</b>
3.1	SIGN UP.....	7
3.2	INSERT .....	8
3.3	RESEARCH .....	8
<b>4.</b>	<b>IMPLEMENTED PROVIDERS .....</b>	<b>9</b>
4.1	ROUTING REQUEST .....	9
4.2	ACCOUNT MANAGER.....	12
4.3	EVENT MANAGER .....	13
4.4	ATTRACTION MANAGER .....	13
4.5	RESEARCH MANAGER .....	13
4.6	IMPLEMENTED PROSUMER.....	13
<b>5</b>	<b>RESOURCE MANAGEMENT .....</b>	<b>13</b>
6.1	LOAD BALANCER .....	13
6.2	BALANCE AGENT .....	14
<b>6</b>	<b>PERFORMANCE .....</b>	<b>14</b>
<b>7</b>	<b>MOBILE APPLICATION .....</b>	<b>15</b>
<b>8</b>	<b>ARCHETYPE .....</b>	<b>19</b>

## 1. Introduction

The application we have developed, as its name suggests, is a tourist guide. This has been designed to be adapted to any city/region and it allows the inclusion of attractions, which can be of various kinds (art, culture, entertainment etc. etc.), and events as well. Obviously, both of them can only be inserted only by a user who has access (logged in) and who is then registered to our platform. Moreover, through the use of one of the views of our mobile application, it is possible to carry out the search specifying different parameters that are expressed in detail in section 4, where all providers developed are described.

In addition, an external and already existing service provider has been used. In particular the Google Geocoding APIs have been used to implement it.

The entire application has been developed following the principles of the Service-Oriented paradigm, achieving its aims:

- **Loose Coupling** - Less dependency on each other. This is one of the main characteristics of web services which just states that there should be as less dependency as possible between the web services and the client invoking the web service. So, if the service functionality changes at any point in time, it should not break the client application or stop it from working.
- **Service Abstraction** - Services hide the logic they encapsulate from the outside world. The service should not expose how it executes its functionality; it should just tell the client application on what it does and not on how it does it.
- **Service Reusability** - Logic is divided into services with the intent of maximizing reuse. Hence, once the code for a web service is written it should have the ability work with various application types.
- **Service Autonomy** - Services should have control over the logic they encapsulate. The service knows everything on what functionality it offers and hence should also have complete control over the code it contains.
- **Service Statelessness** - Ideally, services should be stateless. This means that services should not withhold information from one state to the other.

**Section 2** of this documentation contains a description and representation of that which is a *static view* of the system. All those that are the software components and the communications between them are described.

In **section 3** instead there is a representation of the *dynamic view* of the implemented system. Here are described various scenarios of iteration between system components through the use of Sequence Diagrams.

**Section 4** and **section 5** describes in detail all the *services* that have been implemented (both SOAPapi and RESTapi).

**Section 6** describes in detail the implementation of the *Load Balancer*, its way of interacting with other components and the 'server scheduling' policy used.

The *mobile application* developed was described in **section 7**, while finally in **section 8** was described the development of the *archetype* together with the choices made.

The code developed can be found at the following link:

[ <https://github.com/stefanocortellessa/Service-Oriented> ]

## 2. Static View

### 2.1 Component Diagram

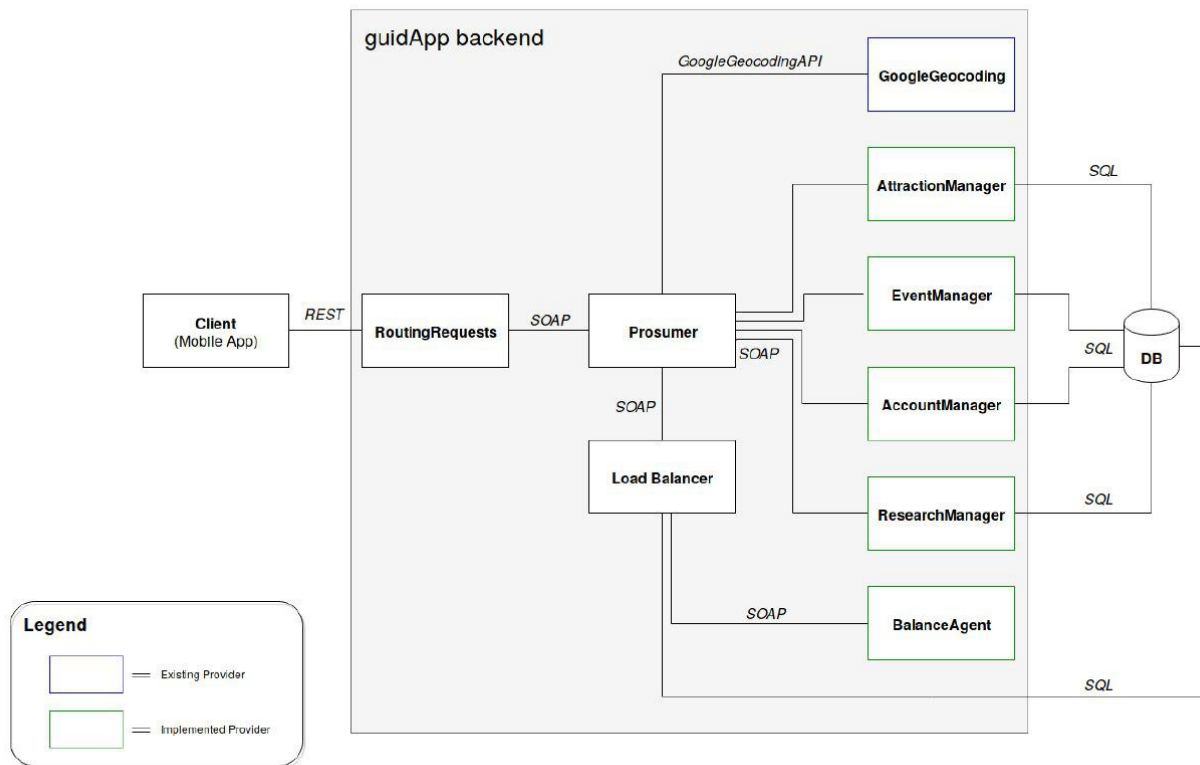


Figure 1: Component Diagram

The System was initially divided, conceptually, into three different macro components that can be identified as **Client**, **Back-End** and **DB**.

The first macro-component, namely the *Client*, is represented by the implemented mobile application. This has been developed through the use of **ionic** that has allowed to realize a multi-platform application. Thanks to this it is possible to interact with the macro-component Back-End having at disposal a user-friendly view of easy use.

It also communicates with the Back-End exclusively through **REST** calls, interfacing with the *RoutingRequest* component. The latter was developed using *Java Spring Boot* and collects requests from the Client redirecting them to the 'Prosumer' through **SOAP** calls. This interfaces with all available providers and is the one who chooses, sbased on the request received from *RoutingRequest*, which provider to call through the SOAP call. Prosumer also dialogues with the *LoadBalancer*, which periodically communicates with the *BalanceAgent*, that collects **performance details** about the server on which it is located and through these **produces a score**, used by the *LoadBalancer* to choose the **best server** on which to perform the required operation at that time (more details are explained in section 6).

Once the server has been chosen, the *LoadBalancer* communicates its URL to the *Prosumer*, which takes care of requesting the service at the correct instance of the various providers (*AttractionManager*, *EventManager*, *ResearchManager*, *AccountManager* and *GoogleGeocoding*). Finally, these services can communicate with the latest macro-component, the DB, through SQL queries by entering, deleting, updating or selecting data from the database.

## 2.2 Deployment Diagram

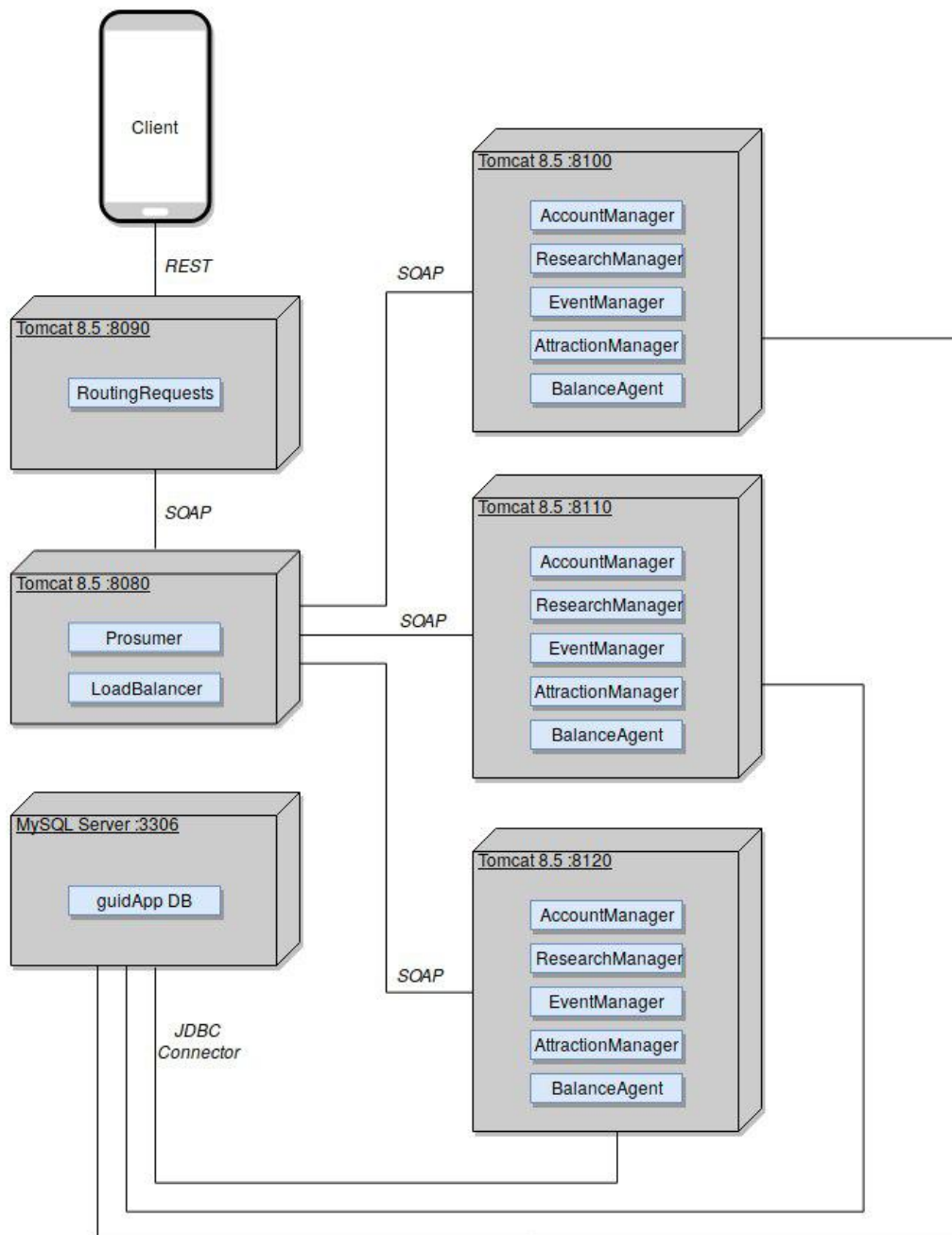


Figure 2: Deployment Diagram

As can be seen from Figure 2, **five different instances** of Tomcat have been created, more precisely on ports **8080, 8090, 8100, 8110** and **8120**.

Keeping in mind the operating logic expressed in section 2.1, respectively were deployed:

- *RoutingRequest* on 8090 instance,
- *Prosumer* and *LoadBalancer* on 8080 instance,
- *AccountManager*, *ResearchManager*, *EventManager*, *AttractionManager* and *BalanceAgent* on the remaining instances (8100, 8110, 8120).

This particular setting has been used to highlight the possibility of deploying multiple instances of providers available on different server instances. In this configuration, the *LoadBalancer* plays a key role: it takes care, in response to a request received, to select the best provider instance to request the service from (see section 6 for more details).

To do that, it uses a **thread** that *periodically* (eg: every 0.5s) *requires the various BalanceAgents* (one for each server on which providers are present) a score, calculated by making a *weighted sum* of various indicators (number of active threads on the server, server load, memory used). Once those scores are obtained, the thread uses them to populate a HashMap that maps the server's URL to its score. The *LoadBalancer* then, in response to a request, chooses to ask for the service to the provider who is on the instance with the lowest score in the HashMap.

## 2.3 Database ER-Diagram

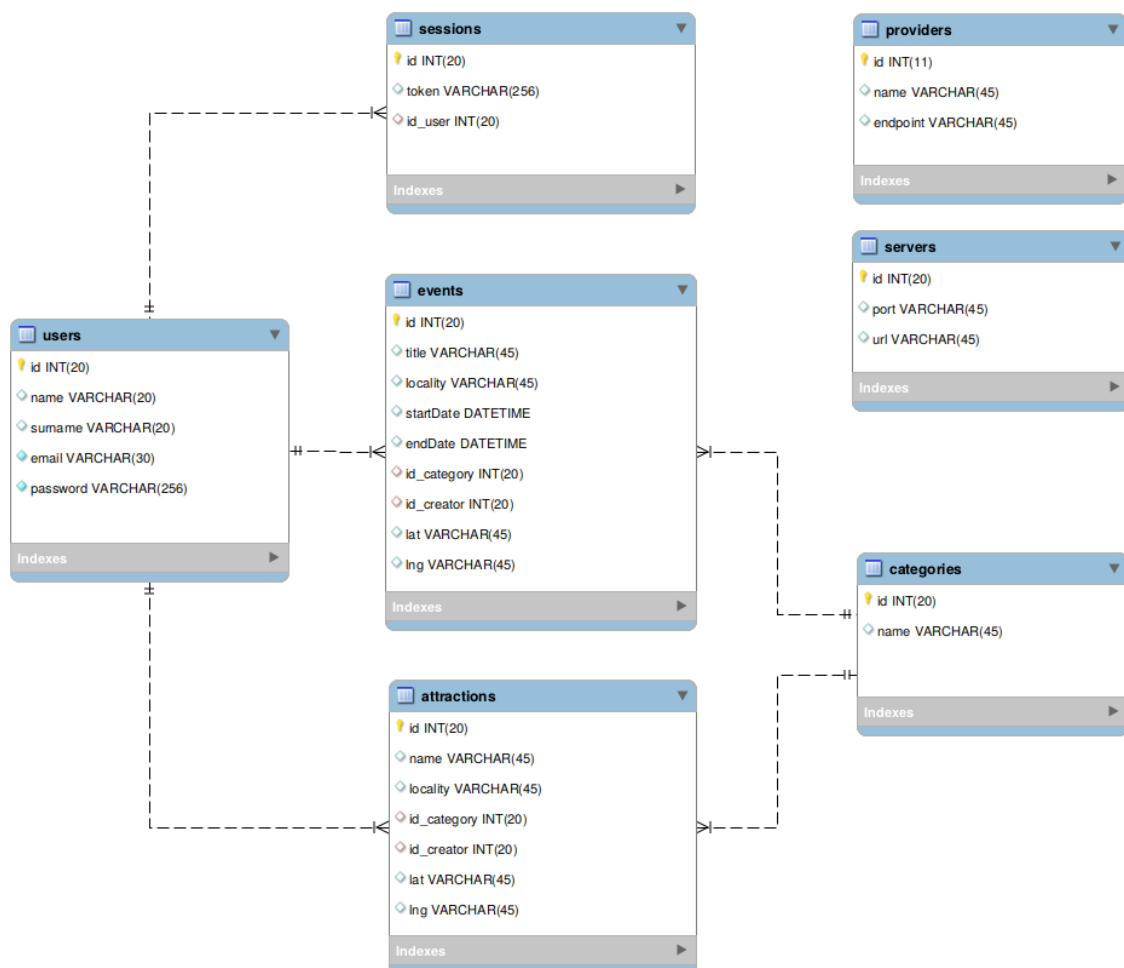


Figure 3: Database Description

### 3. Dynamic View

Keeping in mind the structure of the system (described in [section 2](#)), here there are three different examples of iteration between the different components of the system.

Within the Sequence Diagrams, **REST** and **SOAP** calls have been distinguished, for a better and easier understanding, through the addition of a "REST" label at the end of the function name. Those functions that do not have such a label are then considered SOAP.

Not all the operations that the system is able to perform have been reported since the flow is always the same.

#### 3.1 Sign Up

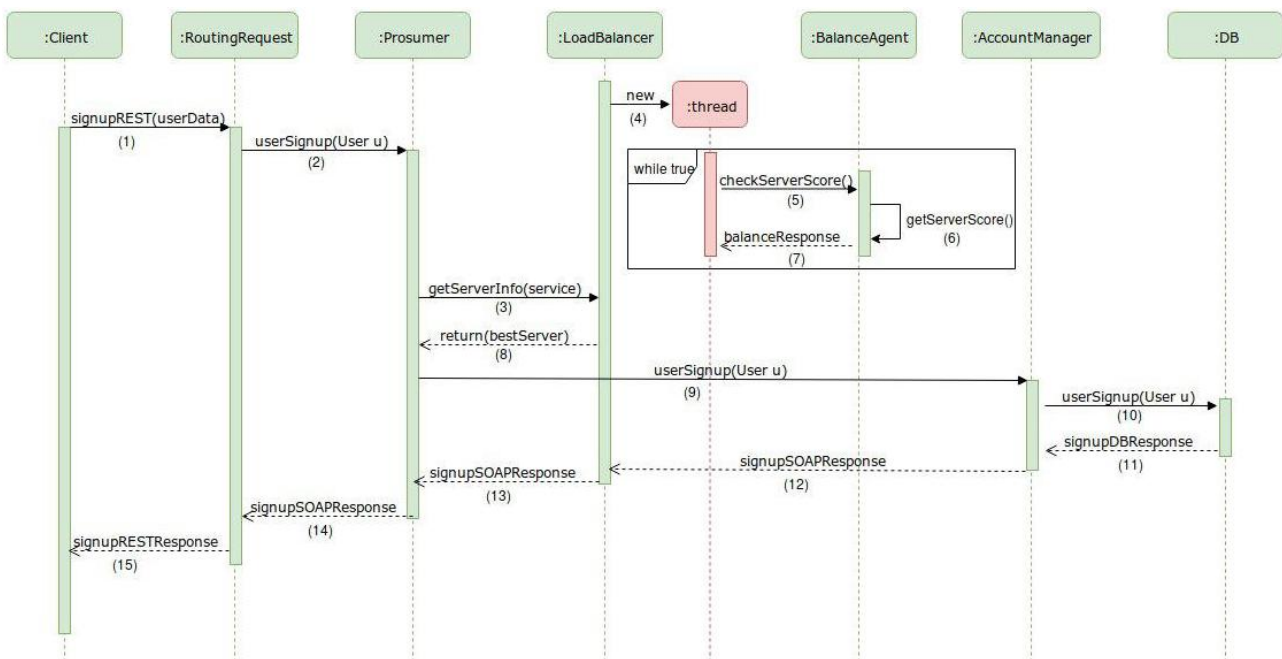


Figure 4: Sign Up Sequence Diagram

Figure 4 shows all the steps that are performed by guidApp when a user, from the registration view after filling in the form, sends the request to the provider to register himself.

At that time, a `signupREST(userData)` call is made (1) from the Client to `RoutingRequest`, which requests the `userSignUp(user)` service from the Prosumer (2).

At this point, the Prosumer asks the `LoadBalancer` for the best instance of server to request the Signup service from (3). To obtain this information, the `LoadBalancer`, at the start-up of the server on which it is located, automatically launches a cyclic thread (every 0.5s) that communicates with the `BalanceAgent` of each server used, requesting the scores via `checkServerScore()` (5). The generic `BalanceAgent` then calculates the score of the server on which it is located using the `getServerScore()` function and returns the result to the `LoadBalancer` thread (7) which updates the `HashMap` about scores. In this way, when the Prosumer asks the `LoadBalancer` for the best server instance (8), it responds directly indicating the instance with minimum score in the `HashMap` filled by the thread.

Once the server is selected, the prosumer asks the `AccountManager` provider deployed on that server to register via `userSignUp(user)` (9) which queries the DB (10) and returns the response to the same provider (11). Finally, the same steps are followed in the reverse direction until the response is returned to the Client that shows it to the user.

### 3.2 Insert

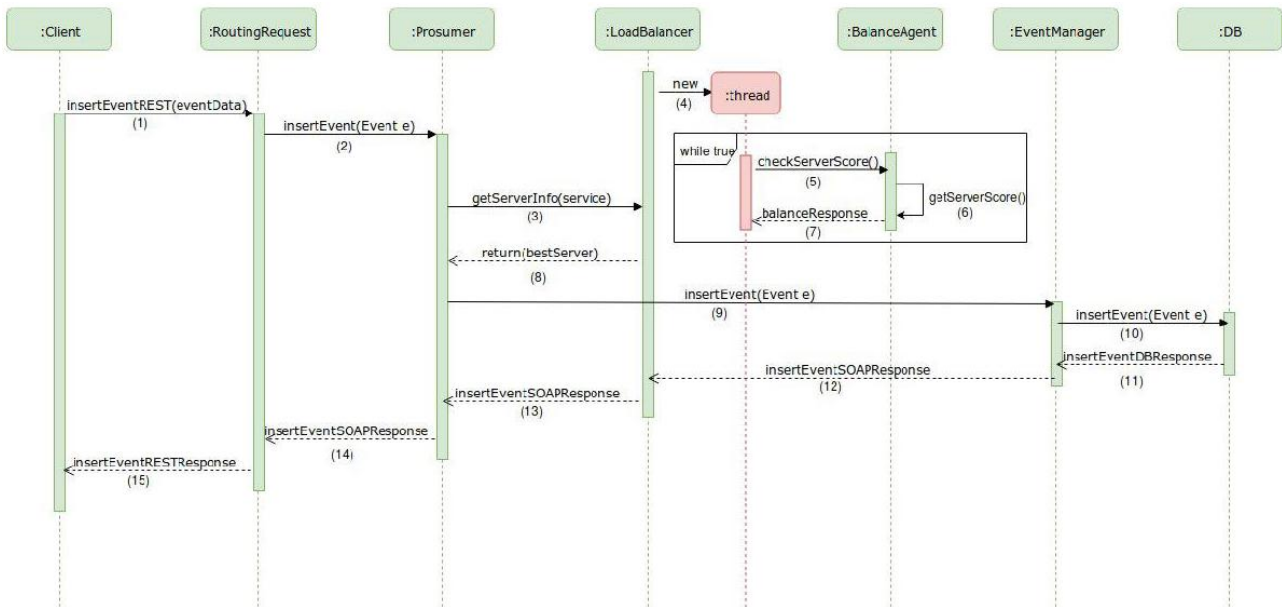


Figure 5: Insert Sequence Diagram

Figure 5 shows all steps needed to insert an event into the DB. These are exactly the same as described in section 3.1, so the explanation is not repeated. What changes is only the data sent (since they have different structures) and the name of the functions that perform the various operations.

### 3.3 Research

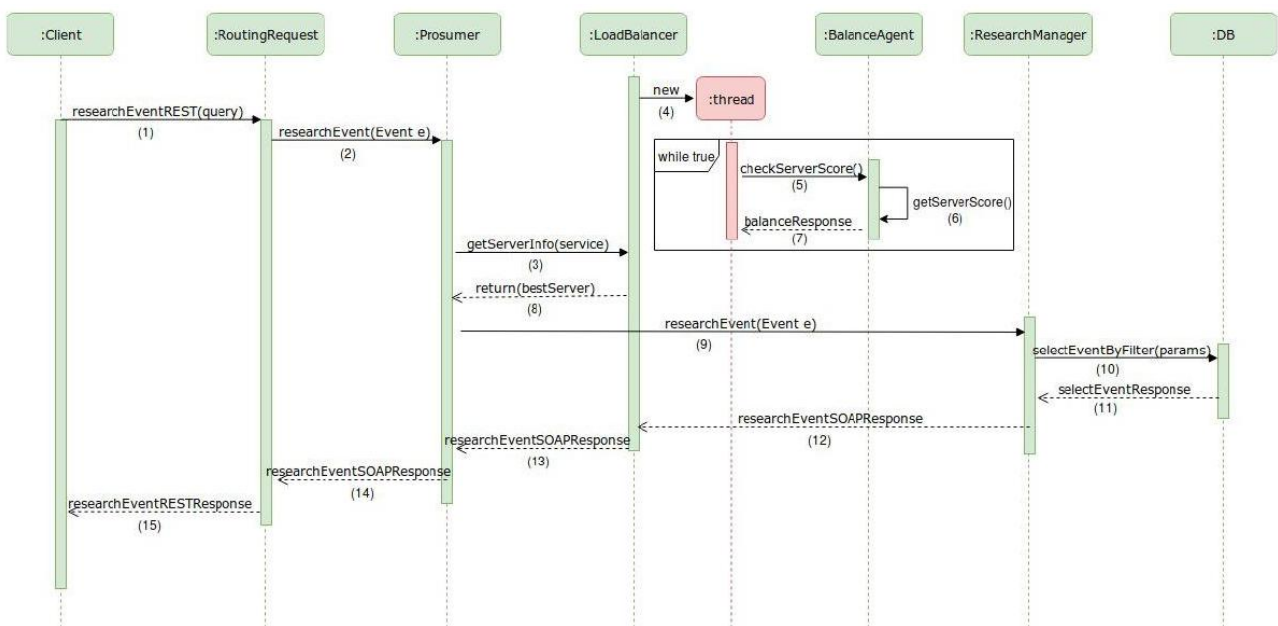


Figure 6: Research Sequence Diagram



Figure 5 shows all the steps needed to research an event into the DB.

These are exactly the same as described in section 3.1, so the explanation is not repeated. What changes is only the data sent (since they have different structures) and the name of the functions that perform the various operations.

## 4. Implemented Providers

In this section all providers implemented are described in detail, indicating the functions that every service is able to perform.

### 4.1 Routing Request

The *RoutingRequest* was developed using Spring Boot and implements REST APIs used by the mobile client. In particular, in implementing these APIs, it takes care of requesting to the Prosumer the service necessary to complete the request. The latter will then request the service from the interested provider, forwarding the response to the *RoutingRequest*, so that it can be viewed through the app (RESTapi responses).

REST calls are:

#### 1) *AccountController*:

- *URL*: `http://localhost:8090/signup`
- *HTTP method*: POST
- Sign Up a user in the database. The JSON passed is like:

```
{
  "name": "user_name",
  "surname": "user_surname",
  "email": "user_email",
  "password": "user_password"
}
```

---

- *URL*: `http://localhost:8090/login`
- *HTTP method*: POST
- Login a user creating a token into the DB related to him. The JSON passed is like:

```
{
  "email": "user_email",
  "password": "user_password"
}
```

---

- *URL*: `http://localhost:8090/logout/{token}`
- *HTTP method*: DELETE
- Logout a user deleting the token related to the user into the. It doesn't require a JSON.

## 2) *AttractionController*:

- *URL*: `http: //localhost:8090/attractions`
- *HTTP method*: POST
- Insert an attraction in the database. The JSON passed is like:

```
{
  "name": "attraction_name",
  "locality": " attraction_locality ",
  "category":
  {
    "name":" category_name ",
    "id": category_id
  },
  "creator":
  {
    "id": user_id
  }
}
```

---

- *URL*: `http: //localhost:8090/attractions`
- *HTTP method*: PUT
- Update an attraction referring to the *attraction\_id* into the DB. The JSON passed is like:

```
{
  "id": attraction_id,
  "name": "attraction_name",
  "locality": " attraction_locality",
  "category":
  {
    "name": "category_name ",
    "id": category_id
  },
  "creator":
  {
    "id": user_id
  }
}
```

---

- *URL*: `http: //localhost:8090/attractions`
- *HTTP method*: DELETE
- Delete an attraction referring to the *attraction\_id* and to the creator (*user\_id*). The JSON passed is like:

```
{
  "id": attraction_id,
  "creator":
  {
    "id": user_id
  }
}
```

---

### 3) *EventController*:

- *URL*: `http://localhost:8090/events`
- *HTTP method*: POST
- Insert an event in the database. The JSON passed is like:

```
{
  "name": "attraction_name",
  "locality": " attraction_locality ",
  "category":
  {
    "name": " category_name ",
    "id": category_id
  },
  "creator":
  {
    "id": user_id
  }
}
```

---

- *URL*: `http://localhost:8090/events`
- *HTTP method*: PUT
- Update an event referring to the *event\_id* into the DB. The JSON passed is like:

```
{
  "id": event_id,
  "title": " event_title",
  "locality": " event_locality",
  "startDate": " event_startDate",
  "endDate": " event_endDate ",
  "lat": " event_lat",
  "lng": " event_lng",
  "category":
  {
    "name": "category_name ",
    "id": category_id
  },
  "creator":
  {
    "id": user_id
  }
}
```

---

- *URL*: `http://localhost:8090/events`
- *HTTP method*: DELETE
- Delete an attraction referring to the *event\_id* and to the creator (*user\_id*). The JSON passed is like:

```
{
  "id": event_id,
  "creator":
  {
    "id": user_id
  }
}
```

---

#### 4) *ResearchController*:

- *URL*: `http: //localhost:8090/user/{id}/events`
  - *HTTP method*: GET
  - Return all events inserted from the user that has id equal to {id}.
- 

- *URL*: `http: //localhost:8090/user/{id}/attractions`
  - *HTTP method*: GET
  - Return all attractions inserted from the user that has id equal to {id}.
- 

- *URL*: `http: //localhost:8090/attractions`
  - *HTTP method*: GET
  - Return all attractions.
- 

- *URL*: `http: //localhost:8090/events`
  - *HTTP method*: GET
  - Return all events.
- 

- *URL*: `http: //localhost:8090/attractions/{id}`
  - *HTTP method*: GET
  - Return the attraction with id equal to {id}.
- 

- *URL*: `http: //localhost:8090/events/{id}`
  - *HTTP method*: GET
  - Return the event with id equal to {id}.
- 

- *URL*: `http: //localhost:8090/categories/{id}`
  - *HTTP method*: GET
  - Return the category with id equal to {id}.
- 

## 4.2 Account Manager

This provider offers all services regarding accounts. In particular here is reported the list of services offered by it:

- |                |   |                              |
|----------------|---|------------------------------|
| ○ userSignup   | → | Signup service               |
| ○ userLogin    | → | Login service                |
| ○ userLogout   | → | Logout service               |
| ○ checkSession | → | Check token validity service |

### 4.3 Event Manager

This provider offers all services regarding events. In particular here is reported the list of services offered by it:

- insertEvent → Insert new event service service
- deleteEvent → Delete existing event (by event creator) service
- updateEvent → Update existing event (by event creator) service

### 4.4 Attraction Manager

This provider offers all services regarding attractions. In particular here is reported the list of services offered by it:

- insertAttraction → Insert new attraction service
- deleteAttraction → Delete existing attraction (by creator) service
- updateAttraction → Update existing attraction (by creator) service

### 4.5 Research Manager

This provider offers all services regarding research operations. In particular here is reported the list of services offered by it:

- researchEventByCreatorId → Get list of event by their creator id service
- researchAttractionByCreatorId → Get list of attractions by their creator id service
- researchEvent → Search events by params (e.g: date) service
- researchAttraction → Search attractions by params service
- researchEventDetail → Get event detail by its id service
- researchAttractionDetail → Get attraction detail by its id service

### 4.6 Implemented Prosumer

The Prosumer takes care of taking charge of the requests from the *RoutingRequest* and to require their processing from the provider that implements the requested service. In particular, it interrogates the best instance of that provider, after having asked the *LoadBalancer* for the address of the server on which the instance is located.

## 5 Resource Management

### 6.1 Load Balancer

*The purpose of the LoadBalancer is to **optimize the performance** of the system.* In particular, this component has been designed as a stand-alone service that takes care of choosing the best instance of current server to which it is convenient to request the service. As specified in the "deployment diagram" (section 2.2), in fact, each provider has been deployed on 4 different Tomcat servers.

When a new request arrives at the Prosumer, the *LoadBalancer* is responsible for **choosing the best provider instance** (i.e. the server on which it is located) at that time and communicating it to the Provider, who only at that point will request the correct instance service.

As it is easy to see, choosing the best provider instance when the Prosumer receives a new request extends the response time of the system. For this reason, it was decided to do this work **offline**. At the *LoadBalancer* start-up, in fact, this creates a **thread** responsible of collecting every 0.5s the scores related to the instantiated servers. The lower a score is, the better a server is considered to be.

Collected scores are stored by the thread in a *HashMap*, associated with the URL of the server instance to which they refer.

In this way, when the Prosumer asks the *LoadBalancer* to tell it the best instance of a server, the only thing the *LoadBalancer* does is to select the instance with minimum score from the *HashMap* provided by the thread.

By structuring the procedure in this way, the response times of the system are *not* influenced (if not minimally) by the choice of the best server. Moreover, in this way the workload required from the system is **balanced** on the various available servers, improving performances.

The policy of choosing the best server (minimum score) considers a *weighted sum* of some server statistics. This score is calculated in particular by the *BalanceAgent*, a service deployed on each server (among those with providers) invoked by the *LoadBalancer* thread. In the next section the functioning of the *BalanceAgent* is explained.

## 6.2 Balance Agent

The *BalanceAgent* is the service responsible of collecting data about the server on which it is deployed and calculating the score obtained by them. This service is deployed on each server on which providers are deployed (e.g. Tomcat port 8100, 8110, 8120 deployment diagram, described in section 2.2).

When the *BalanceAgent* is invoked by the *LoadBalancer* thread, using *JMX* calls to the Tomcat server on which it is located, it collects the following data:

- **Number of Active Threads on the Server (*t*)**
- **Server Load (*s*)**
- **Server Memory Usage (*m*)**

Using this information, it then calculates the following **weighted sum**:

$$[ s * 0.1 + m * 0.6 + t * 0.4 / 100 ]$$

The weights were chosen on the basis of various tests carried out and allow, especially in the face of a very high volume of requests, to distribute almost equally the requests between the various servers available. The results obtained using the structure and policy described are reported in the following section.

## 6 Performance

Below are shown the results obtained by performing various stress tests using the *load tests* provided by SoapUI tool.

These results are expressed through the following quantities:

- **Test:** test id.
- **#Threads:** Number of *concurrent* simulated clients.
- **Delay:** Delay from a client request to another, in seconds.
- **Request per Thread:** Number of requests executed by each thread.
- **Total Requests:** Number of total requests processed by the system during the test.
- **Server 8100, 8110, 8120:** Number of requests taken into account by each server.
- **Min:** Shortest time a request has taken, in seconds.
- **Max:** Longest time a request has taken, in seconds.
- **Avg:** Average time a request has taken, in seconds.
- **Bps:** Byte per second processed by the system.

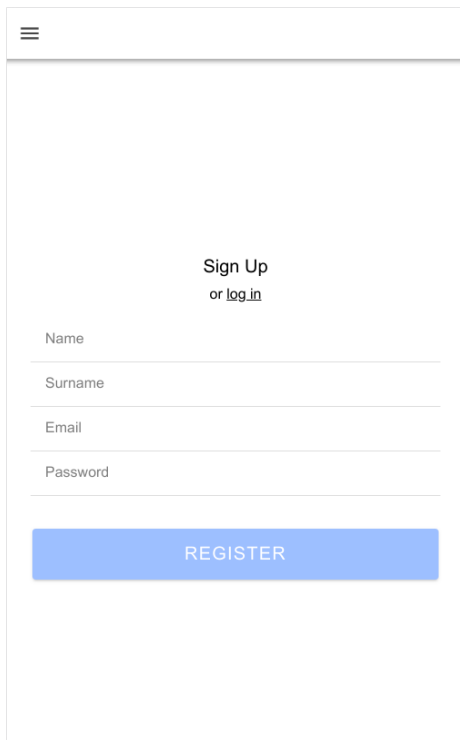
Test	#Thread	Delay (s)	Request per Thread	Total Requests	Server 8100	Server 8110	Server 8120	Min (s)	Max (s)	Avg (s)	Bps
1	100	10	10	1000	475	304	221	0.039	0.58	0.56	447
2	1000	10	10	10000	3563	3745	2692	0.036	0.48	0.53	838
3	100	1	10	1000	552	148	300	0.043	1.85	0.86	2107
4	100	0.5	10	1000	471	161	368	0.044	2.97	1.29	2150
5	500	0.5	10	5000	2696	1282	1022	0.052	6.3	2.45	2360
6	1000	0.5	10	10000	4139	3571	2289	0.04	7.32	2.41	2577
7	500	0	10	5000	3085	1005	910	0.07	7.42	2.62	2552
8	1000	0	10	10000	3209	3298	3493	0.06	10.54	3.41	2116
9	2000	0.5	10	20000	8698	7239	5131	0.12	11.64	4.25	1708
10	1000	0	100	100000	31837	35279	32884	0.07	21.07	5.3	1348

Table 1: Test results

## 7 Mobile Application

To represent the Client side of the system, a mobile application named *guidApp* has been developed. This has been done using *ionic* that is a free, open source mobile UI toolkit for developing cross-platform apps from a single codebase.

Different views have been developed:



Sign Up  
or [log in](#)

Name

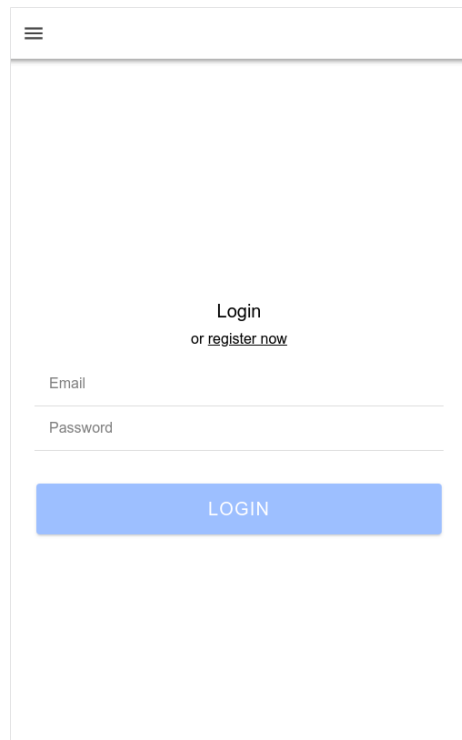
Surname

Email

Password

REGISTER

Figure 7: Sign up view



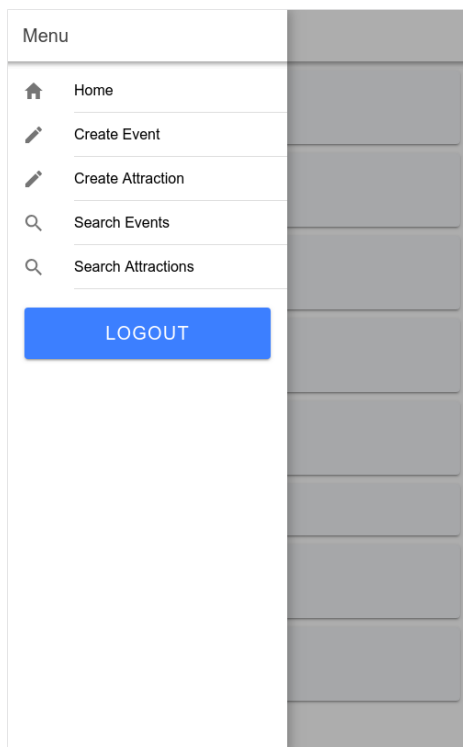
Login  
or [register now](#)

Email

Password

LOGIN

Figure 8: Login view

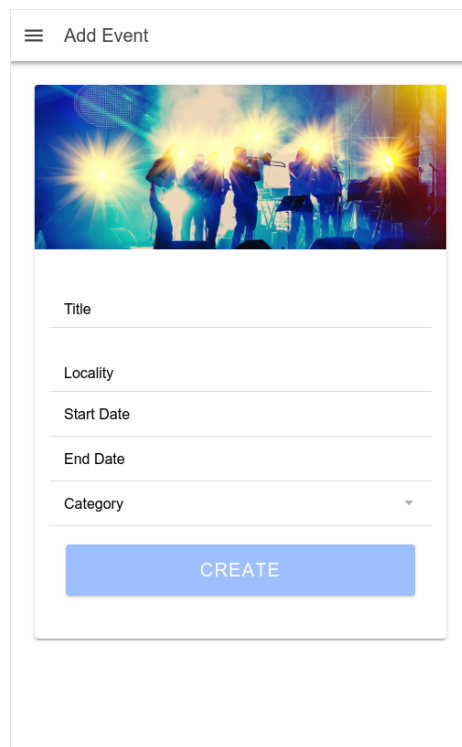


Menu


- Home
- Create Event
- Create Attraction
- Search Events
- Search Attractions

LOGOUT

Figure 9: User view when logged



Add Event



Title

Locality

Start Date

End Date

Category

CREATE

Figure 10: Insert Event view



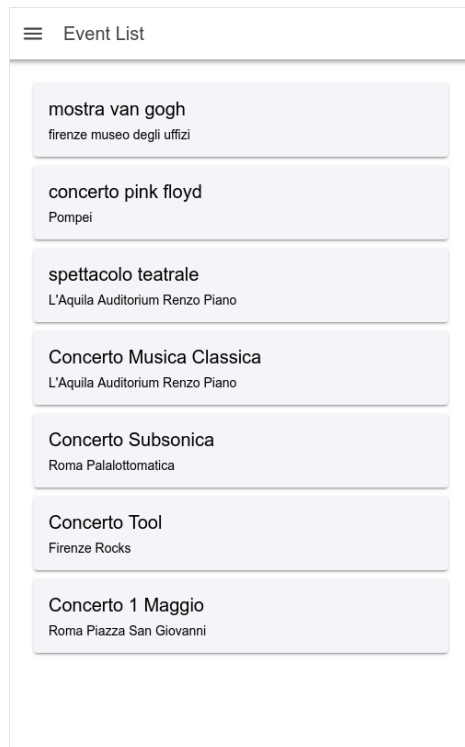


Figure 11: Event List view

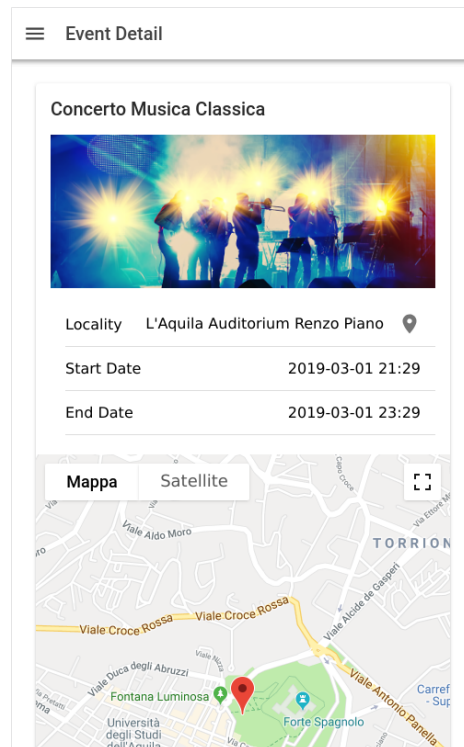


Figure 11: Event Detail view

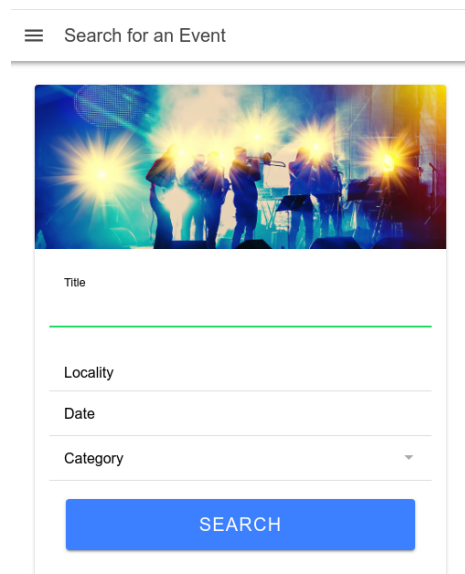


Figure 12: Search Event view

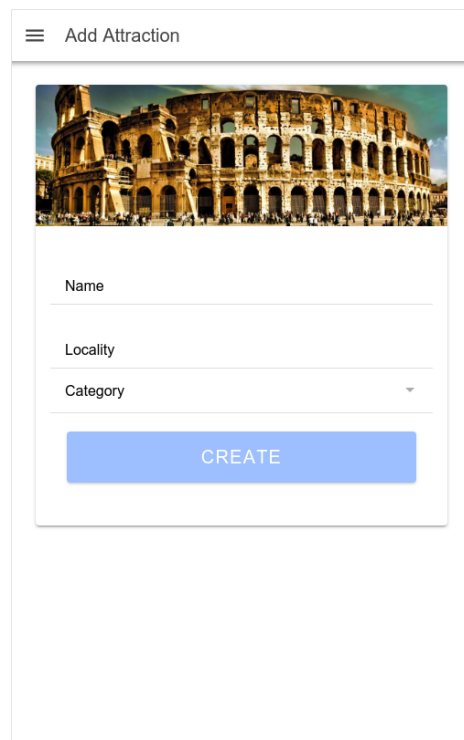


Figure 13: Insert Attraction view

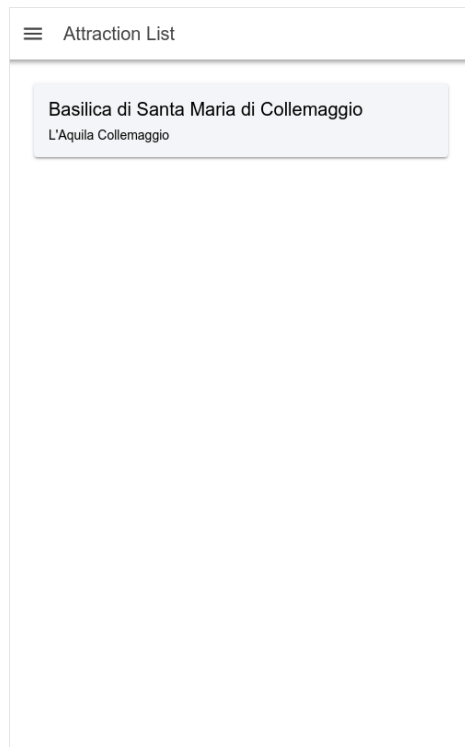


Figure 14: Attraction List view

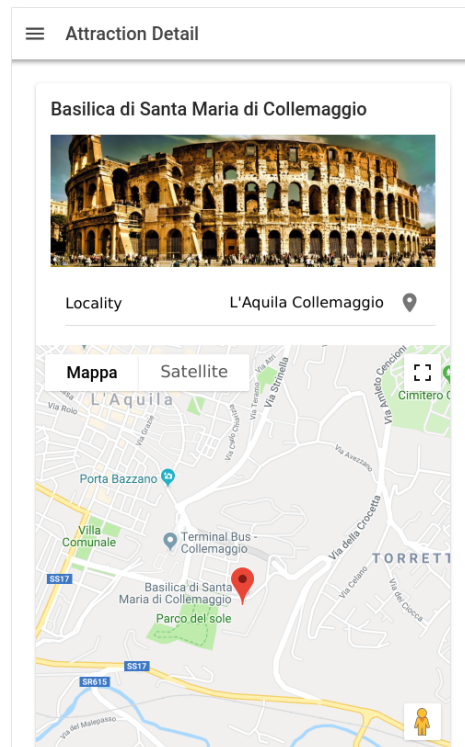


Figure 15: Attraction Detail view

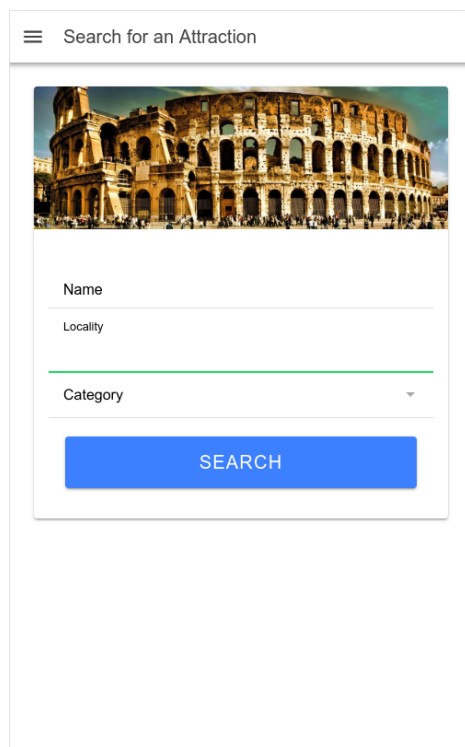


Figure 16: Search Attraction view

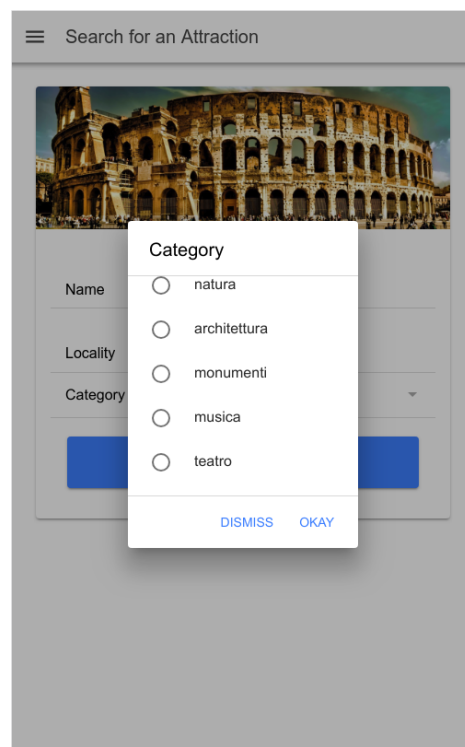


Figure 17: Search Attraction by Category

## 8 Archetype

For the generation of project structures, both consumer and prosumer, has been created an Archetype, called ***guidApp\_archetype***. Here is the structure:



Figure NUMBER: Archetype Structure

Once the Archetype is invoked with the appropriate Maven command, a new project with the following structure:



Figure NUMBER: Project generated from the archetype

As you can see from the *figure NUMBER* above, the archetype allows you to create the structure for a new provider/prosumer.

In addition to the structure, the necessary configuration and code files are generated. In particular, we can see that apache-cxf configuration files, log4j configuration files and, above all, a WSDL containing a dummy operation with request, response and fault are generated. In this way, in addition to having everything already configured, the user will have an example of WSDL to follow in the development phase, so as to comply with the standard and the structure used in other providers/prosumers. In addition to this, the JDBC is also configured automatically, using a parameter of the generated command that will be reported in this section.

As far as Java files are concerned, the ones common to all providers/processors are generated, such as the interface containing the services and their implementation, as well as the dummy `Utility.java` and `DummyModel.java` classes that can be replaced with various utility classes and with the models used by the providers/processors. The names of the files and classes are also generated automatically on the basis of the 'generate' command, so as to comply with the structure and standards used for the other providers/prosumers.

Moreover, common or dummy portions of code have been included in the Java files, which can help the user with development. For example, in the code generated for the `JDBCWebServiceArchetypeTestServiceImpl` class, in which the services and their interface with the db are to be implemented, a dummy method is generated in which the code is already provided to open the connection to the DB close it correctly, etc., so that the user only has to insert the ad-hoc logic.

The following is an example of a generate command:

- *`mvn archetype:generate -DgroupId=it.univaq.disim.sose.newprovider -DartifactId=archetypetest -DarchetypeArtifactId=guidApp_archetype -DarchetypeVersion=1.0.0-SNAPSHOT -DarchetypeGroupId=it.univaq.disim.sose.touristicguide -DinteractiveMode=false -DarchetypeCatalog=local -DdbName=projectName -DartifactIdClass=ArchetypeTest -X`*

In particular, they are indicated in this command:

- `-DgroupId`: Provider/prosumer group id that has to be generated
- `-DartifactId`: Provider/prosumer artifact id that has to be generated (lower case)
- `-DdbName`: Name of the DB to connect the JDBC to
- `-DartifactIdClass`: Artifact id used to denominate files and classes (lower case)

In addition to these parameters, additional parameters can be entered to indicate the versions of spring, log4j, etc. If no particular versions are indicated, the default values indicated in the *archetype-metadata.xml* file will be used.