The Mountain Pictures Application

GABRIELE PRESTIFILIPPO - 835735

Advanced Web Technologies 2014-15

Prof. Piero Fraternali

Table of Contents

Description of Application

Introduction

The application describe in this document is a mountain photo collection system (MPCS). The main objective of a MPCS system is to collect automatically from heterogeneous user-generated pictures that may contain mountain profiles, and to aggregate them in a geographical system interface.

Objectives

The main objective is to design and implement a web big data system, to achieve this there are some subsequent goals to achieve:

- Understanding the development process of a modern Web application according to a principled methodology
- Developing a small but complete rich Internet application featuring social and web interactions
- Learning the design of the architecture, data structures, and interactions of a modern Web application.
- Putting to work the acquired knowledge on advanced server-side Web application development with Node.js
- Applying client-side scripting in the JavaScript language, by implementing the front-end of the social dashboard using state-of-the-art client-side frameworks
- Learning the usage of the main Web APIs currently available on the market.

Requirement Analysis

Logic Requirements

The use case diagram for the actions that users can perform from the web page is shown below.

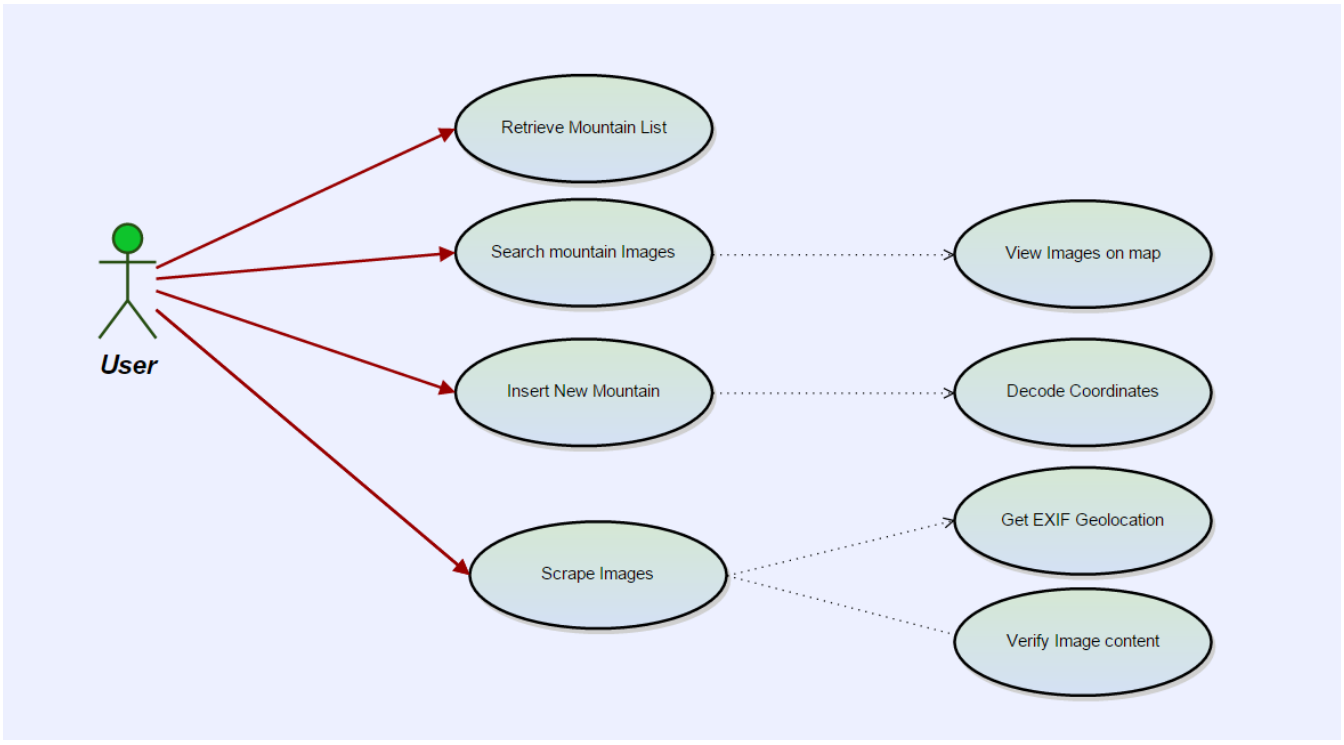


Figure 1 - Use case diagram for user

Detailed Requirements Analysis

Here will follow the modeling of the application, showing the main scenarios available to demonstrate the functionalities of it.
The *Figure 2* shows how to search for a mountain alp, providing the name.

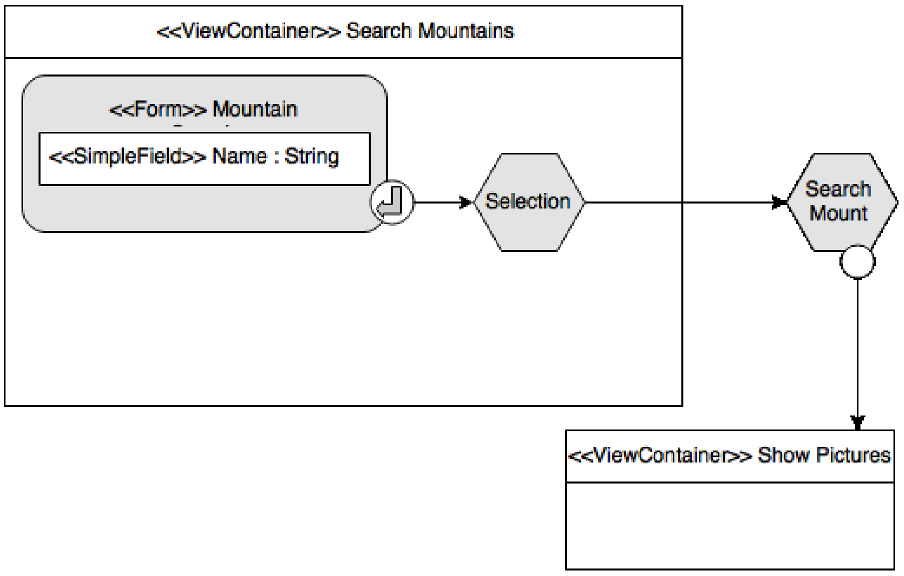

<<ViewContainer>> Search Mountains
<<Form>> Mountain
<<SimpleField>> Name : String
Selection
Search Mount
<<ViewContainer>> Show Pictures

Figure 2 - Search Mountains

Figure 3 provides a way to insert a new Mount into the application, with the possibility to decode the coordinates providing a name for the mount.

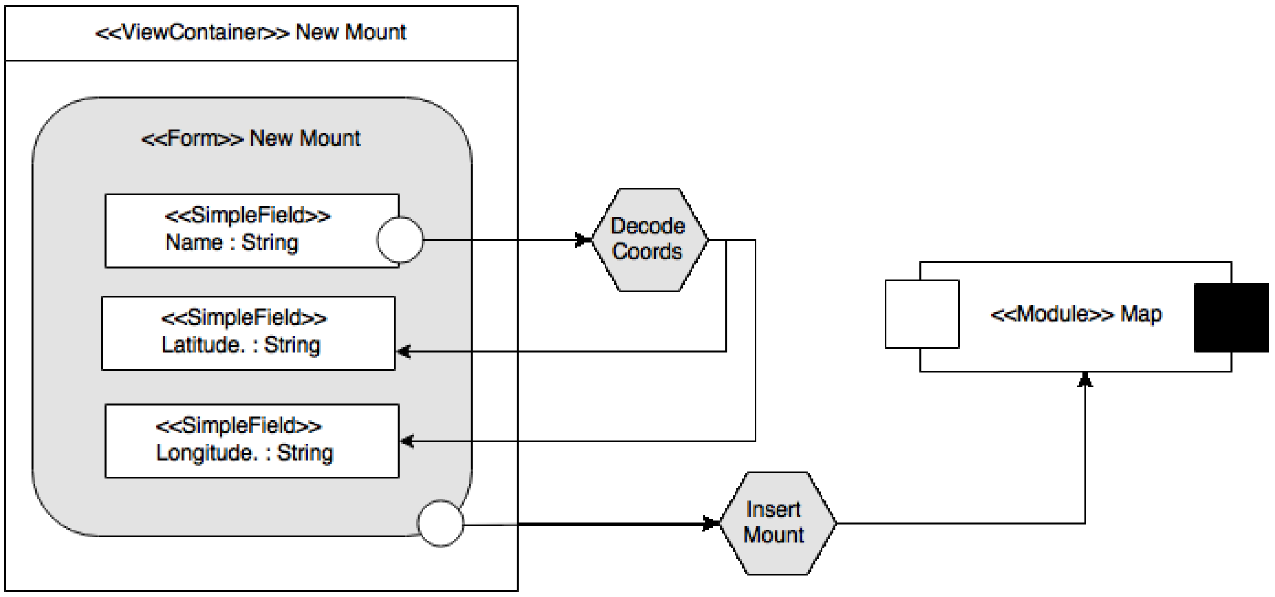

<<ViewContainer>> New Mount
<<Form>> New Mount
<<SimpleField>> Name : String
<<SimpleField>> Latitude. : String
<<SimpleField>> Longitude. : String
Decode Coords
Insert Mount
<<Module>> Map

Figure 3 - New Mount

In the *Figure 4,* we can see how to perform the scraping from various services; we can also validate the content of the image and save the retrieved images to the database.

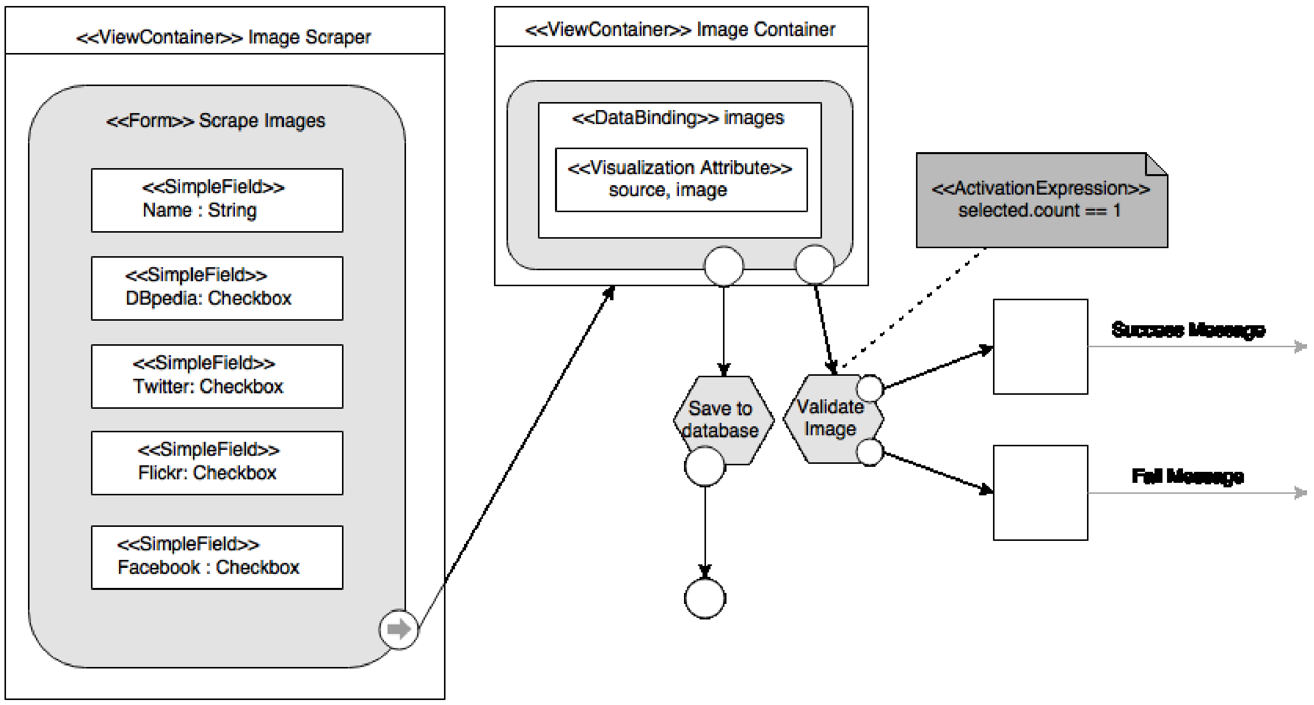

<<ViewContainer>> Image Scraper
<<Form>> Scrape Images
<<SimpleField>> Name : String
<<SimpleField>> DBpedia: Checkbox
<<SimpleField>> Twitter: Checkbox
<<SimpleField>> Flickr: Checkbox
<<SimpleField>> Facebook : Checkbox
<<ViewContainer>> Image Container
<<DataBinding>> images
<<Visualization Attribute>> source, image
<<ActivationExpression>> selected.count == 1
Save to database
Validate Image
Success Message
Fail Message

Figure 4 - Image Scraper

To show in a map the pictures that we have for a given mountain we can follow the flow of *Figure 5.* It allows selecting a Mountain Alp, searching the images that will be inserted into the map in form of *markers*, and then would be possible to view the real images, opening the container.

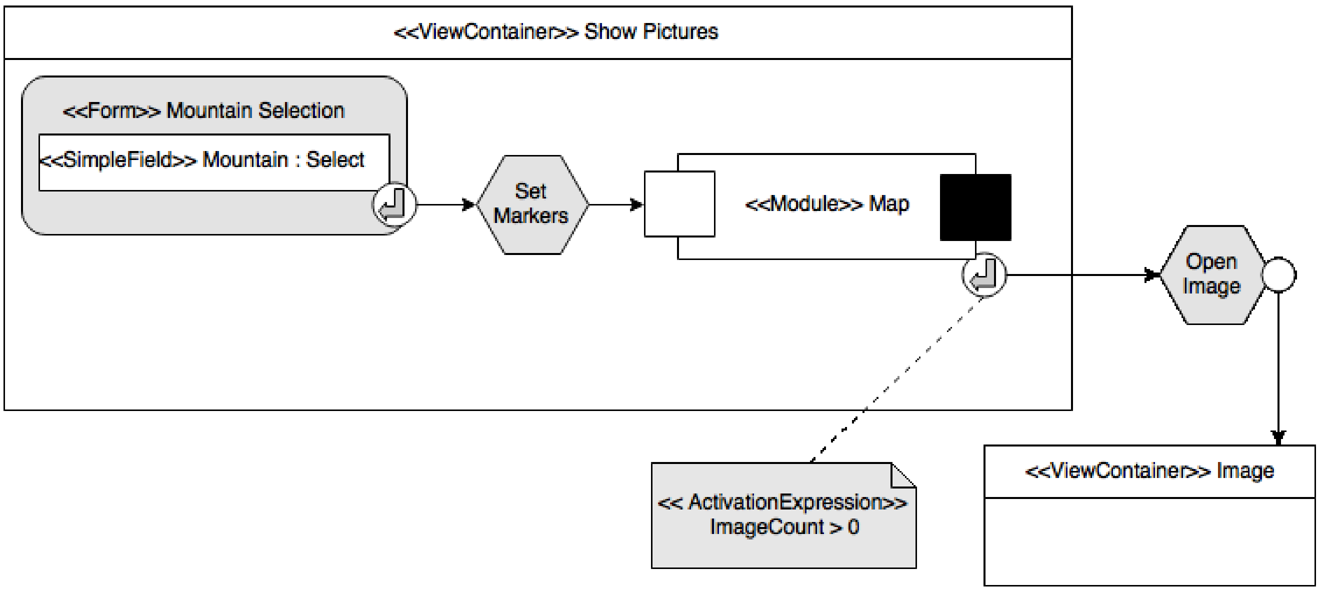

<<ViewContainer>> Show Pictures
<<Form>> Mountain Selection
<<SimpleField>> Mountain : Select
Set Markers
<<Module>> Map
Open Image
<< ActivationExpression>> ImageCount > 0
<<ViewContainer>> Image

Figure 5 - Show Pictures

The visualization of the image happens thanks to the following ViewContainer.

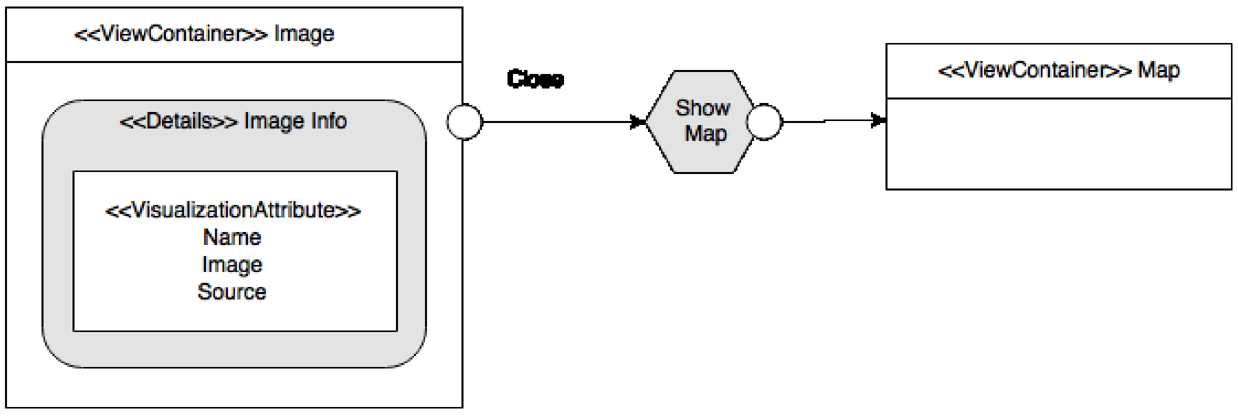


Figure 6 - Image Container

Another functionality provided by the application is the decoding of the Exif data from an image to retrieve the geolocation. From *Figure 7* we can see how it is possible to achieve this result.

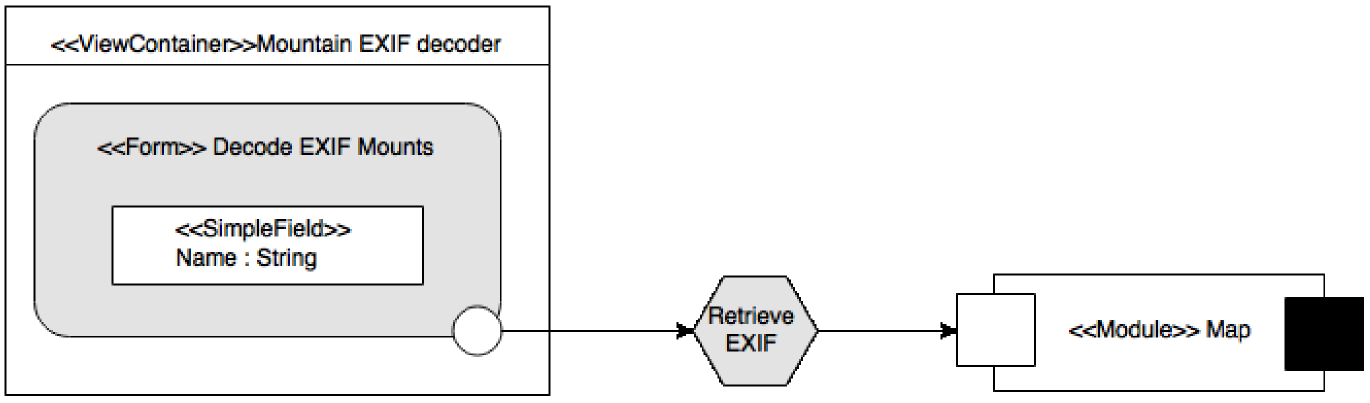

<<ViewContainer>>Mountain EXIF decoder
<<Form>> Decode EXIF Mounts
<<SimpleField>>
Name : String
Retrieve EXIF
<<Module>> Map

Figure 7 - Exif decoder

Architecture Design

The application consist in a web big data system, developed in an enterprise web application. From this comes the choice of using an MVC2 pattern, specifically adapt for web applications. To develop this kind of application, that must be scalable and should allow optimal performance, was done an accurate choice of different technologies.

Server Side

The application is running on a Linux server, exactly with CentOS 5, a community enterprise Operating System, derived from source packages of the Red Hat Enterprise Linux.

Node.js

"*Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*"[1]

The choice of Node.js derives from the requirements of scalability to handle different clients asynchronously and having good performances at the same time. The usage of Node.js is simple and intuitive and permit to create easily efficient REST APIs to allow the interaction with the client. Moreover is useful to share the same programming language between client and server, enabling to transfer objects that will be treated in the same way from the functionalities of the application.
Most important, since we are designing a scalable web application, at high levels of concurrency we can appreciate the management of non-blocking asynchronously request done by Node.js that excels over the JAVA approach.

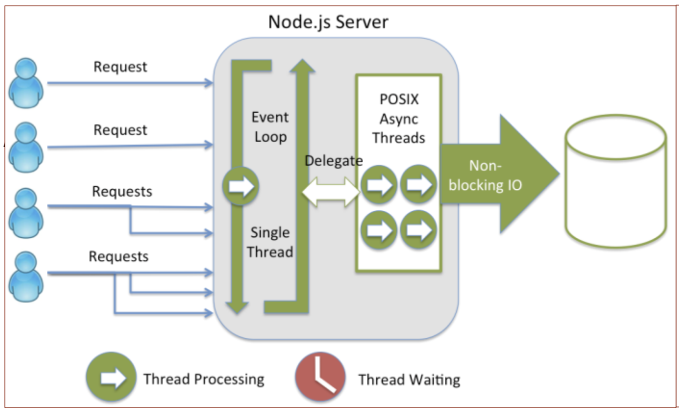


Figure 8 - Node.js IO Managing

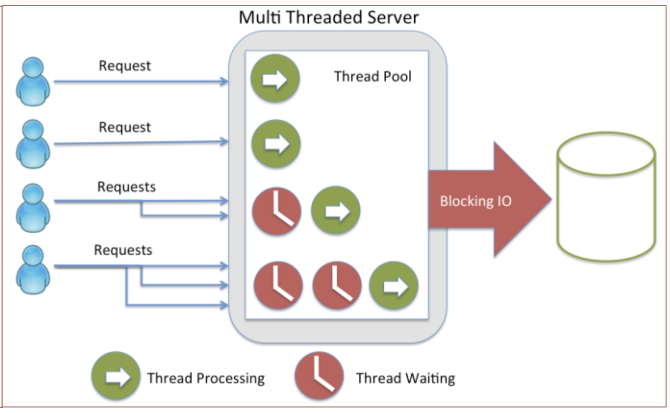


Figure 9 - General Multi-threaded Managing

[1] https://nodejs.org/

Express

"Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications."[2] The choice of Express is almost mandatory with Node.js since it implement lot of useful features, and together these two provide efficient ways to interact with the clients and handle GET and POST requests in a very easy way.

MongoDB

"*MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas*"[3]. Anew, the needs of a scalable web application brought to the choice of a NoSQL database that gives optimal performances at high level of requests. Once again, the support for Node.js makes it one of the best suitable solution for this kind of application.

Web Services APIs

To interact with external services, is needed to use the APIs these services provided. All of the following are REST based APIs. All of these services provides an interaction using JavaScript language, used for the whole project, both on server side with Node.js and on the client.

Twitter

To interact with twitter is used a library for Node.js "Twitter"[4].
It allows using the REST APIs provided by twitter. In the "crawler" model of the application a function permit to query the database of twitter inserting a query and specifying a tag, retrieving then the images associated with the tag. This functionality is used by the application to search for images of the mountain Alps and copy then the URL of the images into the database.
To use the API from twitter is required to have *consumer_key, consumer_secret*, *access_token_key* and *access_token_secret*. All these can be retrieved from the developer's website[5] of Twitter.

Flickr

The interaction with Flickr happens in the same way of Twitter, calling the APIs from a module that allows querying the Flickr database. The library for Node.js that is used is named "node-flickr"[6].
To use this service from Flickr an API key is needed and this can be retrieved from the original website of Flickr[7], upon the registration of a new application.

---

[2] http://expressjs.com/
[3] https://en.wikipedia.org/wiki/MongoDB
[4] https://www.npmjs.com/package/twitter
[5] https://dev.twitter.com/
[6] https://www.npmjs.com/package/node-flickr
[7] https://www.flickr.com/services/api/

DBpedia

DBpedia allows users to query its database using SPARQL, an RDF[8] query language. To implement this functionality in the database the library “sparql-client”[9] is used.
To use this service there is no need for an API key. Is just needed to provide the endpoint of the query that in our case is DBpedia.
These services were used for a double purpose. As the services mentioned before to retrieve images for the mountain Alps, but another functionality used was to retrieve a list of mountain Alps, querying the database and asking for the document belonging to the section of “mountain”.
Then the list was inserted into the database to extend the set of documents that can be used by the application.

Facebook

One of the biggest archive of images is Facebook, so it could be a useful source of images for mountain Alps. It provides an API service to obtain some data but it is not possible to query the database to retrieve public posts. To achieve so a Python script was implemented. The algorithm used consist in searching through the pages with the greatest number of “like” in Facebook related the query input (the mountain name) and inside the pages retrieving the images from the public albums that may be related again with the query.

This method looks to be working for some Alps, while not for some other, since the algorithm is not very mature and may need some improvements.
To achieve this an API key is needed, obtainable from the developer website [10]of Facebook. The sequence used to provide images is the same as the previous services.

Google

The API from google were used both in the client and server part. For the client the APIs provides a way to search for the coordinates from a name. This function is used in the client to insert a new mountain.
To use this service is needed to create an application in the Google Developers website[11] and then provide to the application some permissions to use the geolocation service.

The features provided by google were used also for another scope, trying to analyze the content of an image. However, the API were not used since google does not allow any functionality to interact with its “Reverse Image Search” service.
To try to identify the content of an image a Python script was created. It takes as input the URL of an image and send a request to the “Reverse Image Search” service of Google to obtain some information. Then a crawler scrapes from the page the most relevant information and the words associated with the image. These words are then compares with a list of “bad words” and with the name of the Mountain Alp provided, trying to identify if the content is adequate.

---

[8] An RDF query language is a computer language, specifically a query language for databases, able to retrieve and manipulate data stored in Resource Description Framework format.
[9] https://github.com/thomasfr/node-sparql-client
[10] https://developers.facebook.com/
[11] https://developers.google.com/

Client Side

On the client side, a simple web page was developed, to interact and communicate with the server and thus use all the implemented functionalities. Few libraries were required to handle in an efficient and “clean” way these functions.

jQuery

“jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.”[12] jQuery allows cooperating with the *DOM*[13] and its properties thanks to simple commands. Likewise, it takes care of AJAX request to the server writing few lines of code. A constraint about the choice of using it is given by the fact that most of others external libraries require it.

Leaflet

Leaflet is an open-source JavaScript library that provide a simple way to interact with maps. It includes methods to show map from OpenStreetMap[14] and combine images and markers.

In addition, *“Leaflet.markercluster”*[15] a plug-in for Leaflet was implemented to allow a better stem of markers inside the map.

[12] https://jquery.com/
[13] DOM: Document Object Model
[14] https://www.openstreetmap.org/
[15] https://github.com/Leaflet/Leaflet.markercluster

Back-End design

The model of the server MVC architecture is represented by the following diagram, which put in relation the controller with the corresponding model

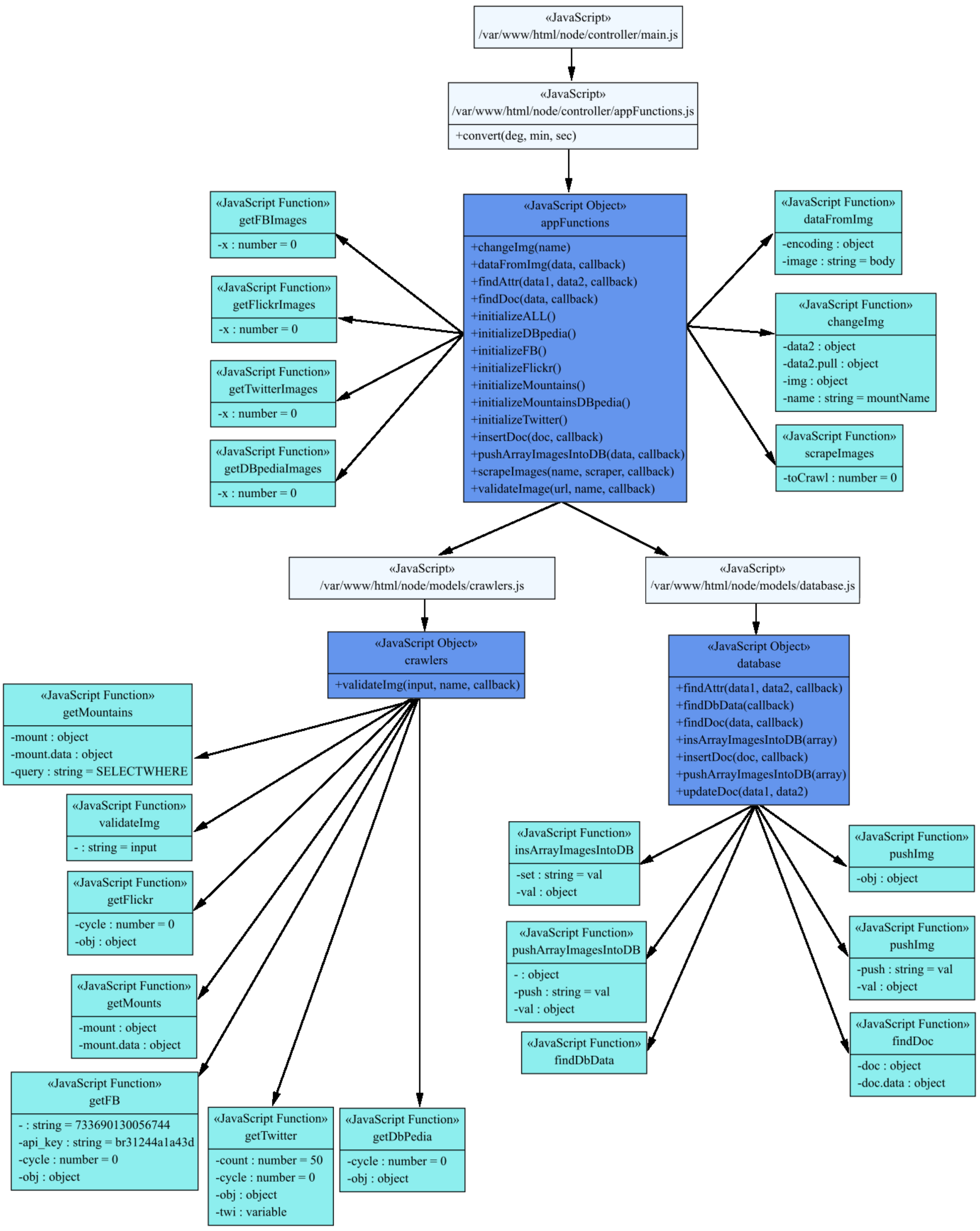

«JavaScript» /var/www/html/node/controller/main.js
«JavaScript» /var/www/html/node/controller/appFunctions.js
+convert(deg, min, sec)
«JavaScript Object» appFunctions
+changeImg(name)
+dataFromImg(data, callback)
+findAttr(data1, data2, callback)
+findDoc(data, callback)
+initializeALL()
+initializeDBpedia()
+initializeFB()
+initializeFlickr()
+initializeMountains()
+initializeMountainsDBpedia()
+initializeTwitter()
+insertDoc(doc, callback)
+pushArrayImagesIntoDB(data, callback)
+scrapeImages(name, scraper, callback)
+validateImage(url, name, callback)
«JavaScript Function» getFBImages
-x : number = 0
«JavaScript Function» getFlickrImages
-x : number = 0
«JavaScript Function» getTwitterImages
-x : number = 0
«JavaScript Function» getDBpediaImages
-x : number = 0
«JavaScript Function» dataFromImg
-encoding : object
-image : string = body
«JavaScript Function» changeImg
-data2 : object
-data2.pull : object
-img : object
-name : string = mountName
«JavaScript Function» scrapeImages
-toCrawl : number = 0
«JavaScript» /var/www/html/node/models/crawlers.js
«JavaScript» /var/www/html/node/models/database.js
«JavaScript Object» crawlers
+validateImg(input, name, callback)
«JavaScript Object» database
+findAttr(data1, data2, callback)
+findDbData(callback)
+findDoc(data, callback)
+insArrayImagesIntoDB(array)
+insertDoc(doc, callback)
+pushArrayImagesIntoDB(array)
+updateDoc(data1, data2)
«JavaScript Function» getMountains
-mount : object
-mount.data : object
-query : string = SELECTWHERE
«JavaScript Function» validateImg
- : string = input
«JavaScript Function» getFlickr
-cycle : number = 0
-obj : object
«JavaScript Function» getMounts
-mount : object
-mount.data : object
«JavaScript Function» getFB
- : string = 733690130056744
-api_key : string = br31244a1a43d
-cycle : number = 0
-obj : object
«JavaScript Function» getTwitter
-count : number = 50
-cycle : number = 0
-obj : object
-twi : variable
«JavaScript Function» getDbPedia
-cycle : number = 0
-obj : object
«JavaScript Function» insArrayImagesIntoDB
-set : string = val
-val : object
«JavaScript Function» pushArrayImagesIntoDB
- : object
-push : string = val
-val : object
«JavaScript Function» findDbData
«JavaScript Function» pushImg
-obj : object
«JavaScript Function» pushImg
-push : string = val
-val : object
«JavaScript Function» findDoc
-doc : object
-doc.data : object

Figure 12 - Class Diagram Back-End

Data Model

MongoDB Introduction

Data in MongoDB are structured in a different way than SQL Databases, where you must determine and declare a table's schema before inserting data. In MongoDB, collections do not enforce document structure.
Each document can match the data fields of the represented entity, even if the data has substantial variation.
A Database in Mongo contains Collections that may recall the ideas of tables in a relational Database. Inside each collection, there may be several Documents.
In these documents, MongoDB stores all the data, which are JSON-style data structures composed of field-and-value pairs

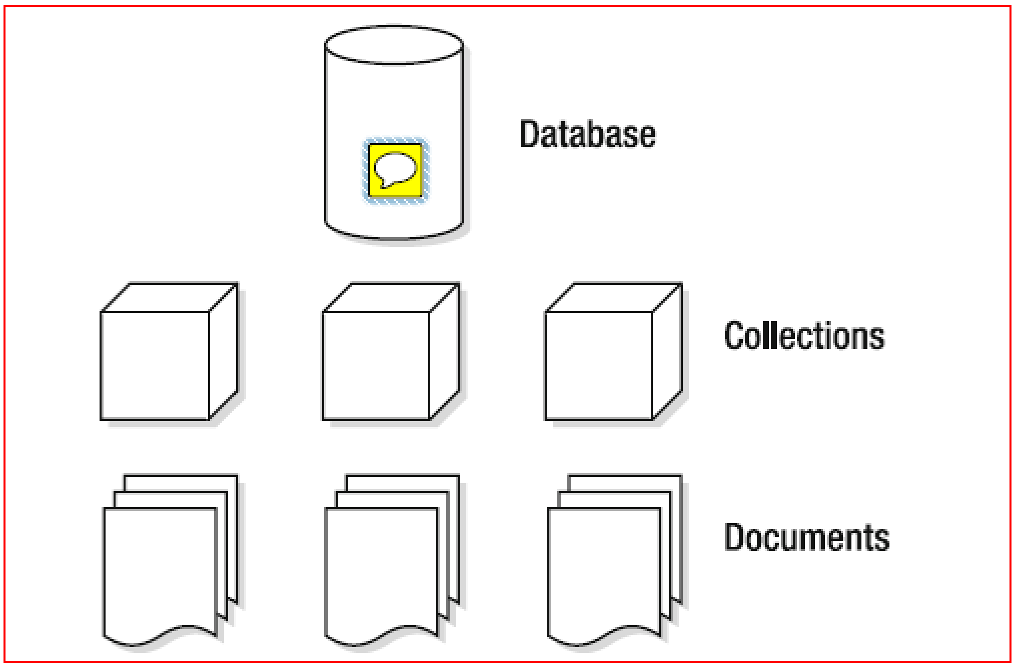


Figure 13 – Structure of MongoDB Data

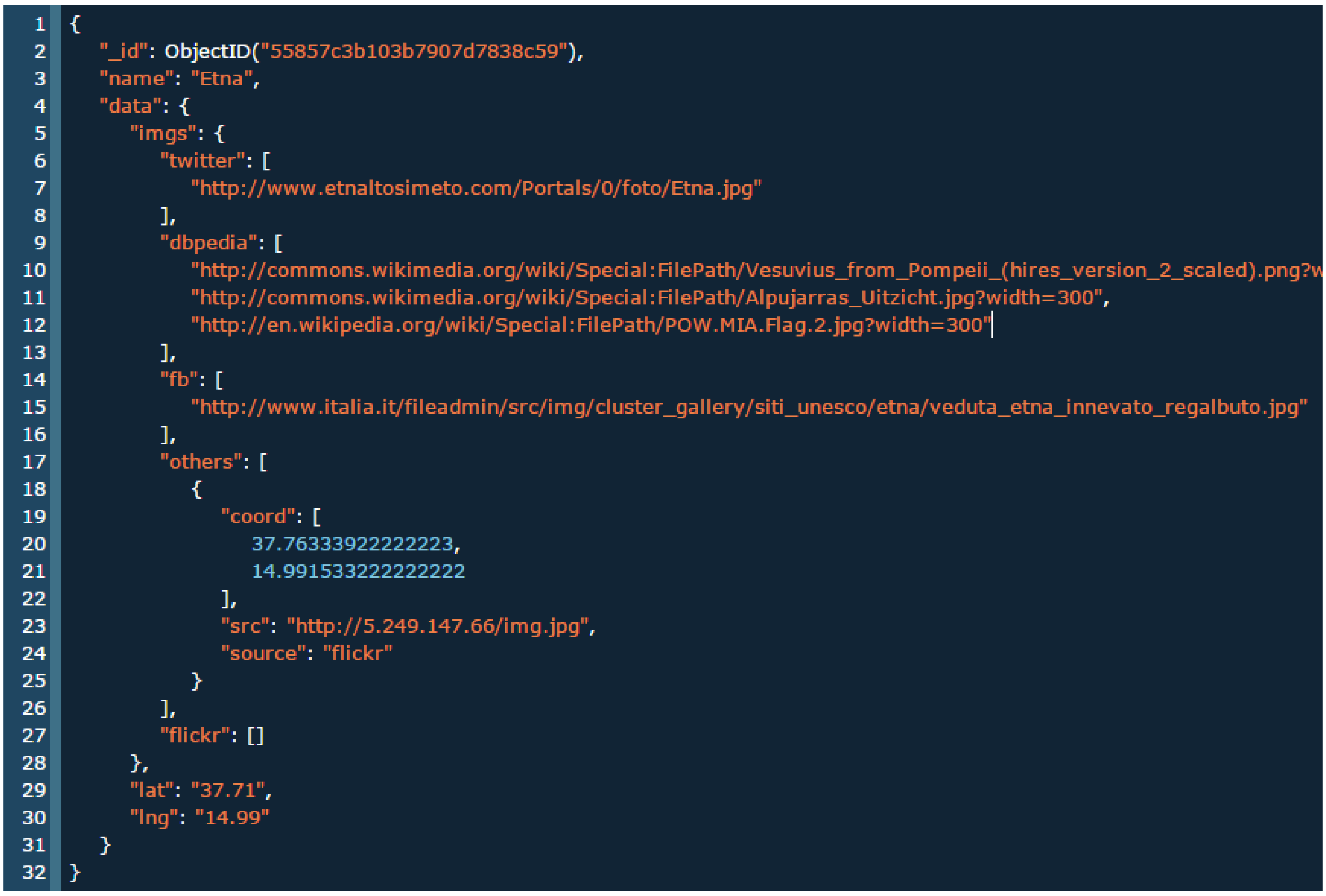
MongoDB Document Model

A JSON object composes a document in the database.
It contains a unique index "_id" defined by an "ObjectID", an index "name" referring to the Name of the Mountain Alp and a substructure "data" that contains for service provided by the crawler an array of images.

An exception is given by the group "other". It refers to the images that have their own geolocation provided by *exif* data. The structure "others" is composed by a set of coordinates, a source for the image and the *URL*.

Example Document

An Example document is provided to show how a real JSON object in this data model looks like.



```
{
   "_id": ObjectID("55857c3b103b7907d7838c59"),
   "name": "Etna",
   "data": {
      "imgs": {
         "twitter": [
            "http://www.etnaltosimeto.com/Portals/0/foto/Etna.jpg"
         ],
         "dbpedia": [
            "http://commons.wikimedia.org/wiki/Special:FilePath/Vesuvius_from_Pompeii_(hires_version_2_scaled).png?w
            "http://commons.wikimedia.org/wiki/Special:FilePath/Alpujarras_Uitzicht.jpg?width=300",
            "http://en.wikipedia.org/wiki/Special:FilePath/POW.MIA.Flag.2.jpg?width=300"
         ],
         "fb": [
            "http://www.italia.it/fileadmin/src/img/cluster_gallery/siti_unesco/etna/veduta_etna_innevato_regalbuto.jpg"
         ],
         "others": [
            {
               "coord": [
                  37.76333922222223,
                  14.991533222222222
               ],
               "src": "http://5.249.147.66/img.jpg",
               "source": "flickr"
            }
         ],
         "flickr": []
      },
      "lat": "37.71",
      "lng": "14.99"
   }
}
```

Figure 14 - Example of Document

Front-End Model

Sequence Diagrams

An example of sequence diagrams is to retrieve the names of the available mountains from the database querying it from the web page.

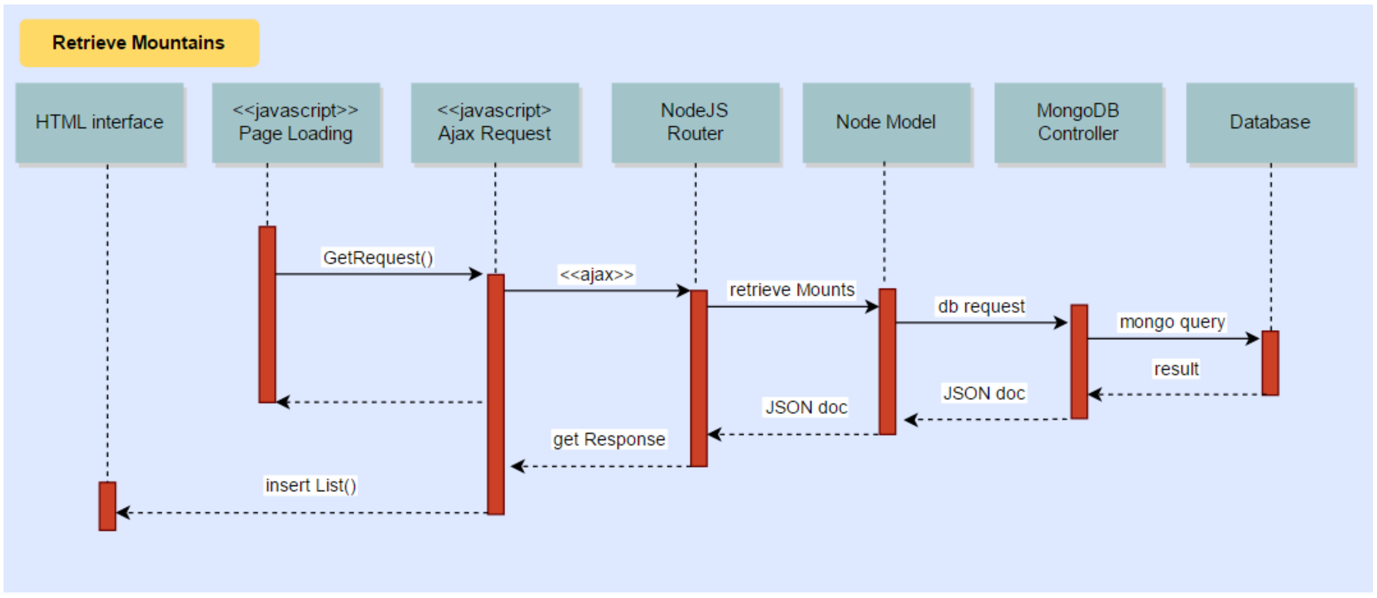


Figure 15 - Mountain List from web page

Another functionality provided by the application is to insert a new mountain alp from the web page.

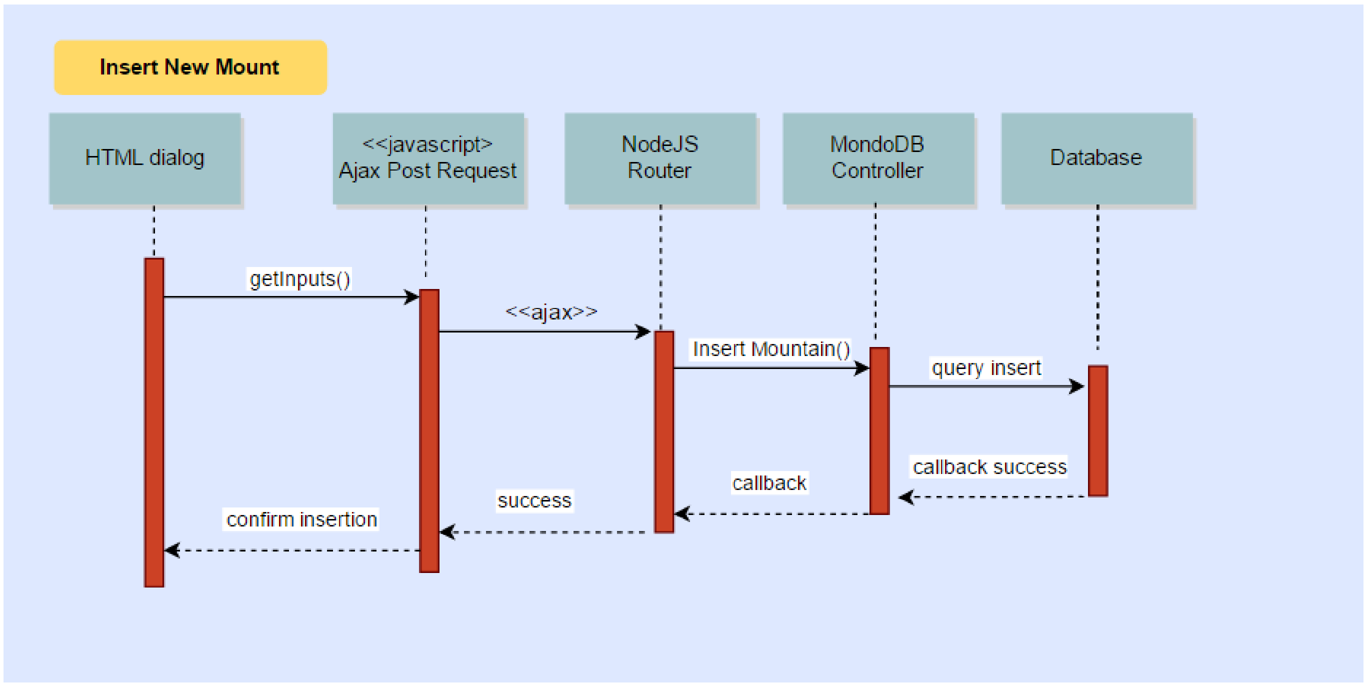


Figure 16 - Mountain Insertion

The applications implements some functionalities to get the automatically images from the web using APIs from different services. The sequence generated among the various APIs is represented by the following example of a Twitter scraping used for the initialization to fill the database.

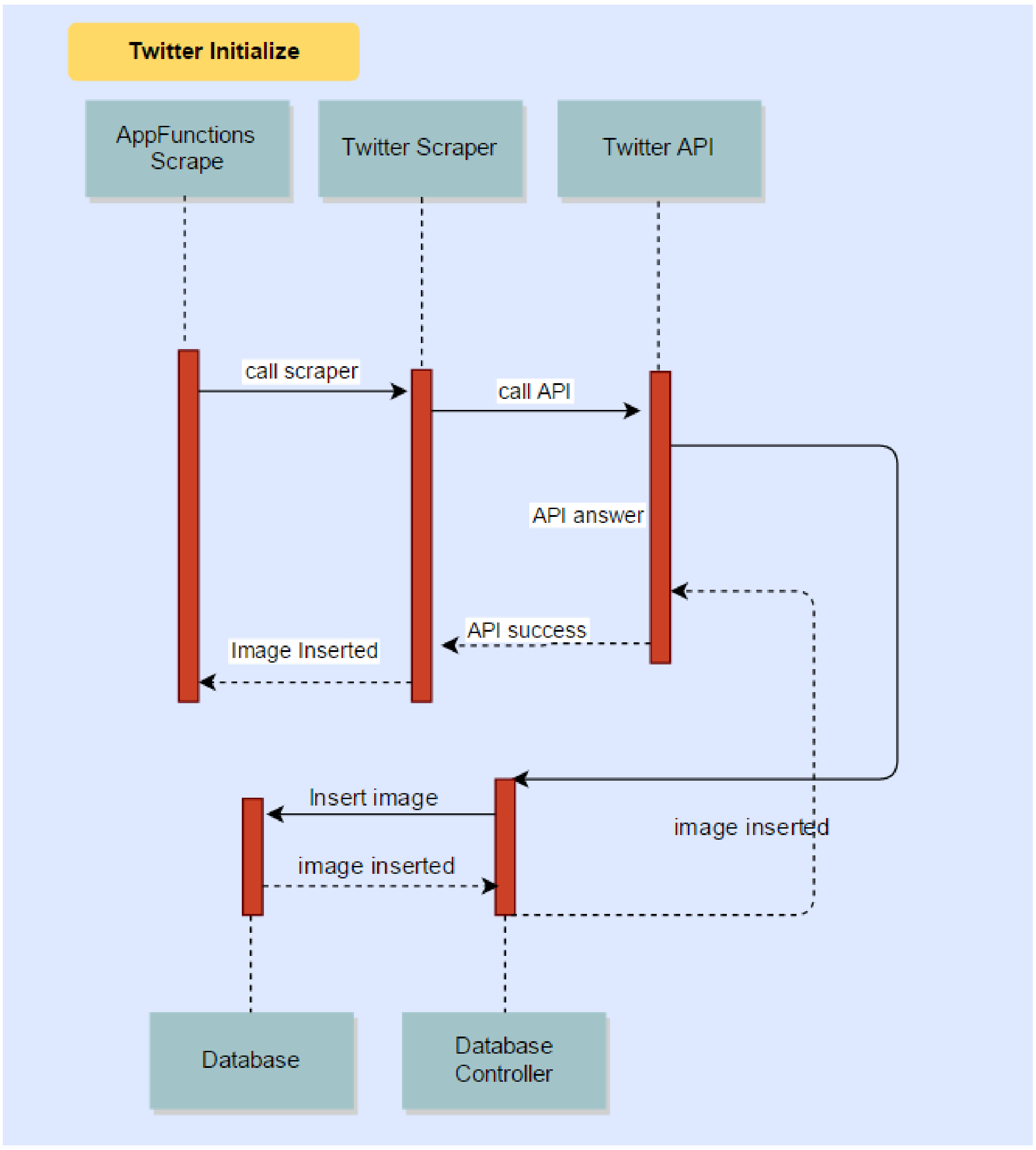


Figure 17 - Twitter Scraper

While is also possible from the front-end to retrieve some images using different APIs, to show them in a list, to perform later further operations.

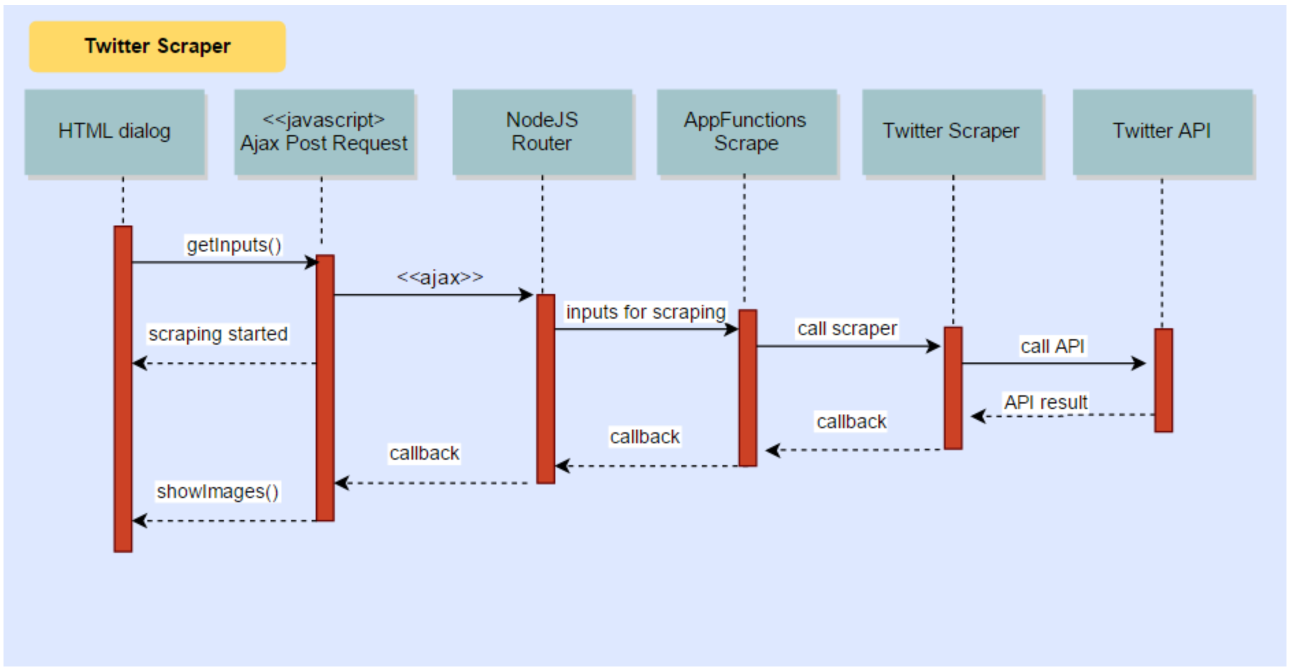


Figure 18 - Twitter Scraper

It is possible to extract the EXIF data from an image and search for the coordinates from the geo location. After the function will update the document in the database, moving the image to the part of object where there are images with their own geo location. To do so we should follow the sequence diagram reported that refers to the single instance of the function.

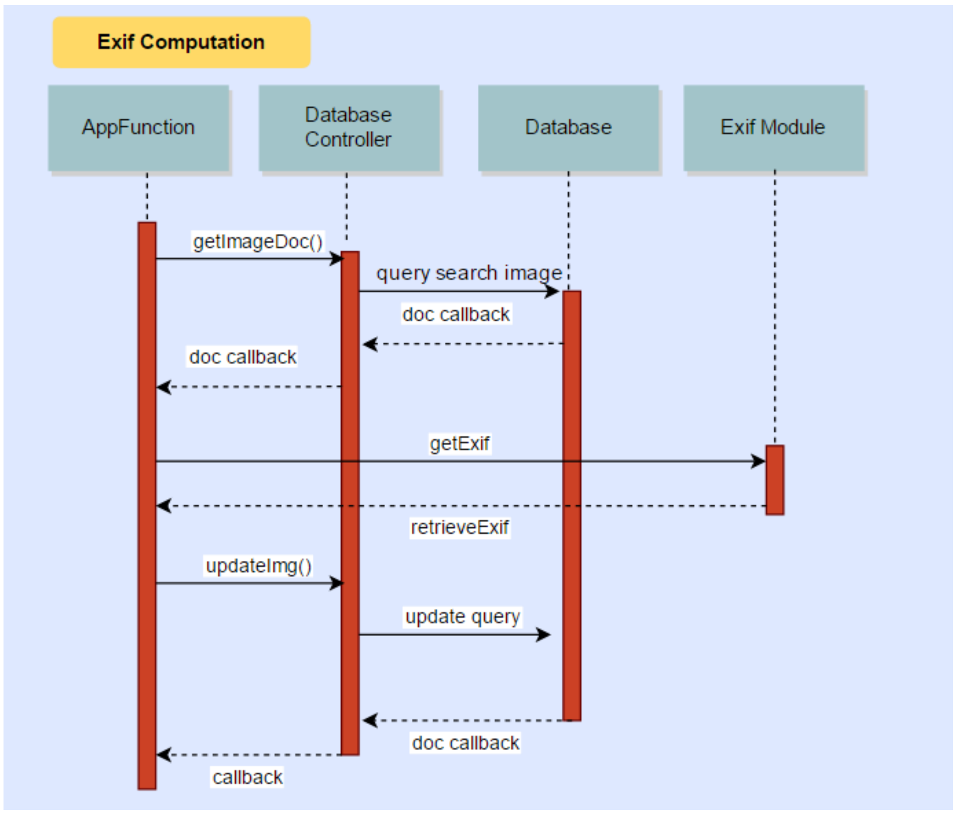


Figure 19 - Extracting EXIF

Usage test

The application was tested on the web Browser *Google Chrome*[16] based on the layout engine software of *WebKit*[17]to render the web pages.

The main page of the application shows a simple vector map, the source comes from OpenStreetMap as mentioned in the client section. Here we find a button on the top-right part to show a menu.

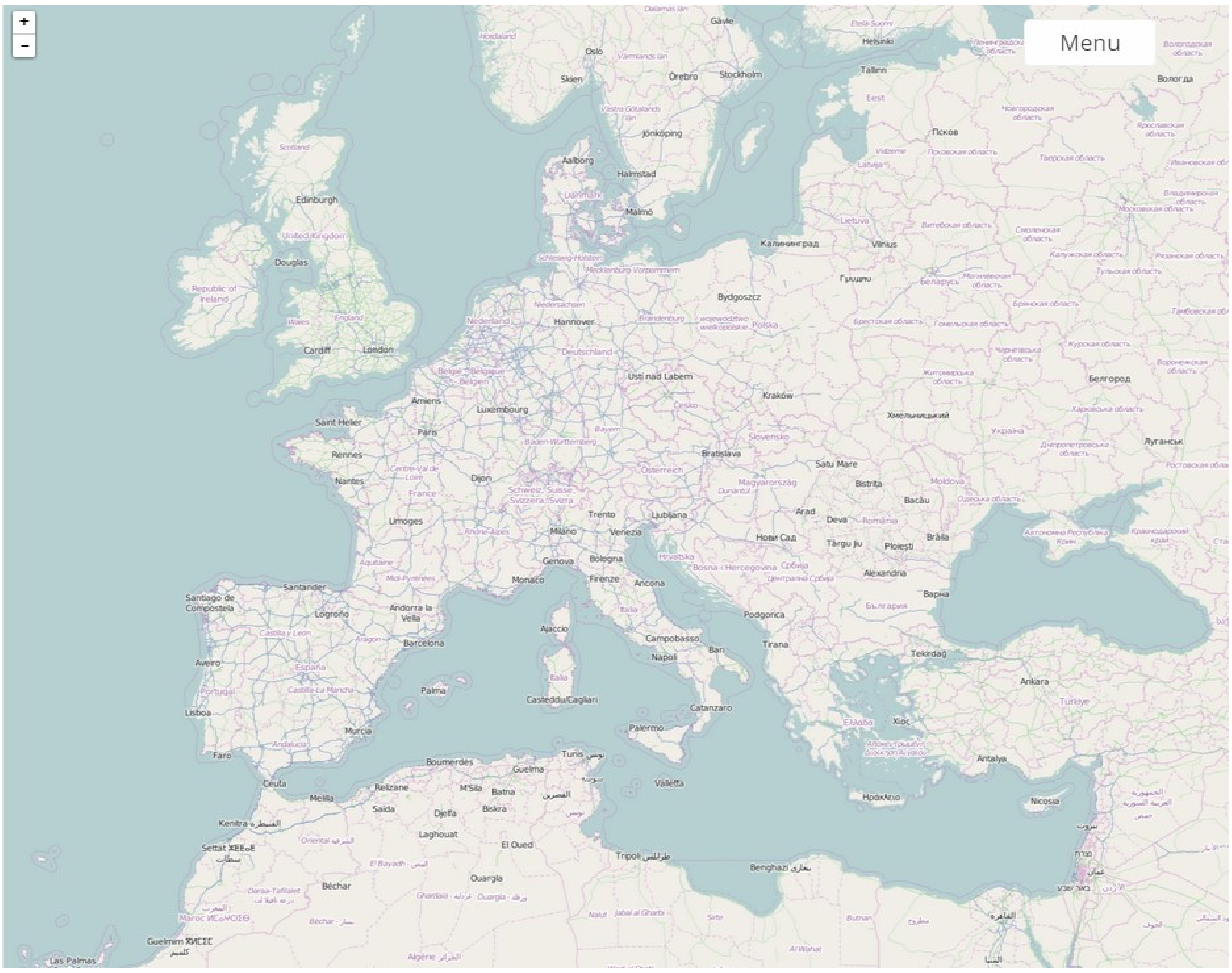


Figure 20 - Main page of the application

16 https://www.google.com/chrome/browser/desktop/index.html
17 https://en.wikipedia.org/wiki/WebKit

From the *menu,* we can interact with the server and execute the needed functionality. From the dropdown list, supported by an accelerator, we can write the name of the mount we are looking for and clicking on *Search* to show on the map the related pictures.

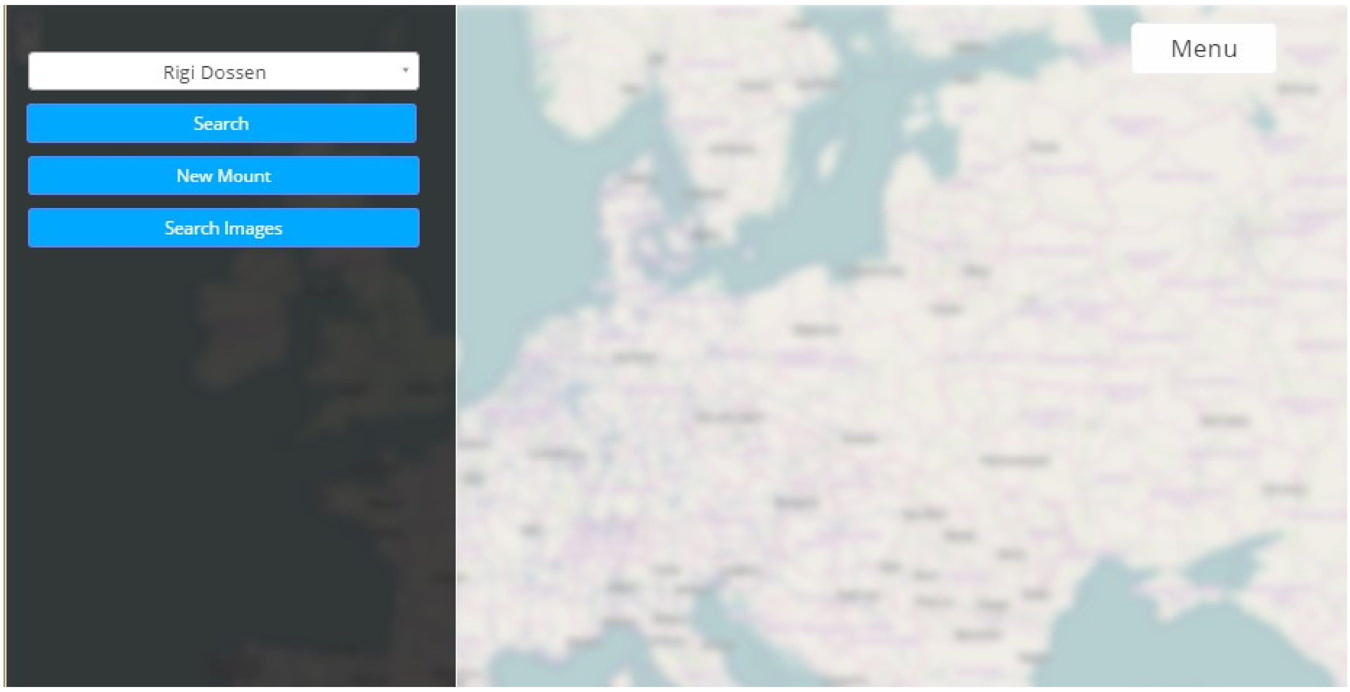


Figure 21 - Left Menu showing list of Mounts

The pictures are shown on a marker, and clicking on it, we can open the cluster to see all the available pictures.

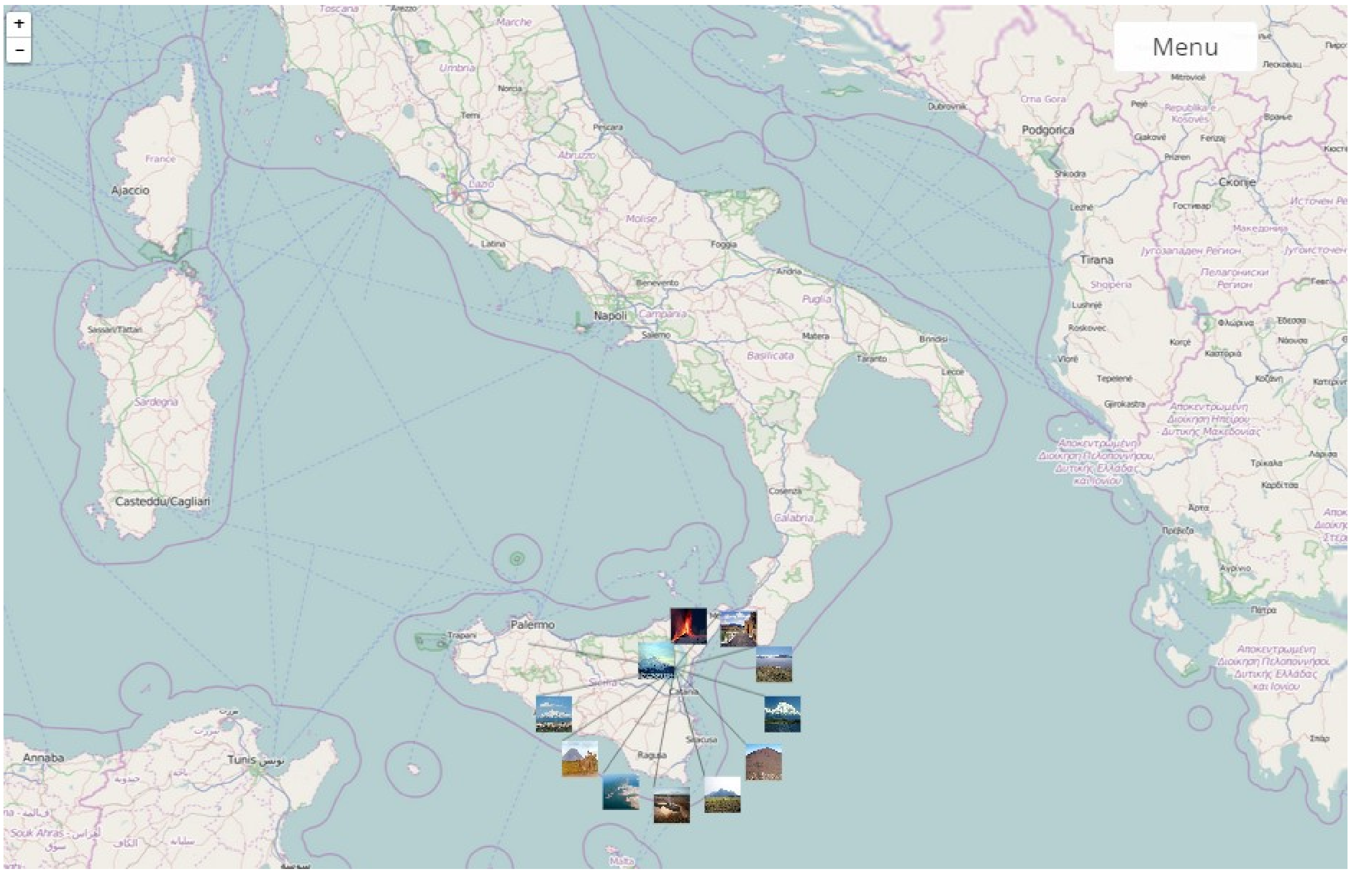


Figure 22 - Available Images for a mount

Here we can select a specific image and a pop-up will open showing some information related to the picture and the source of the image.

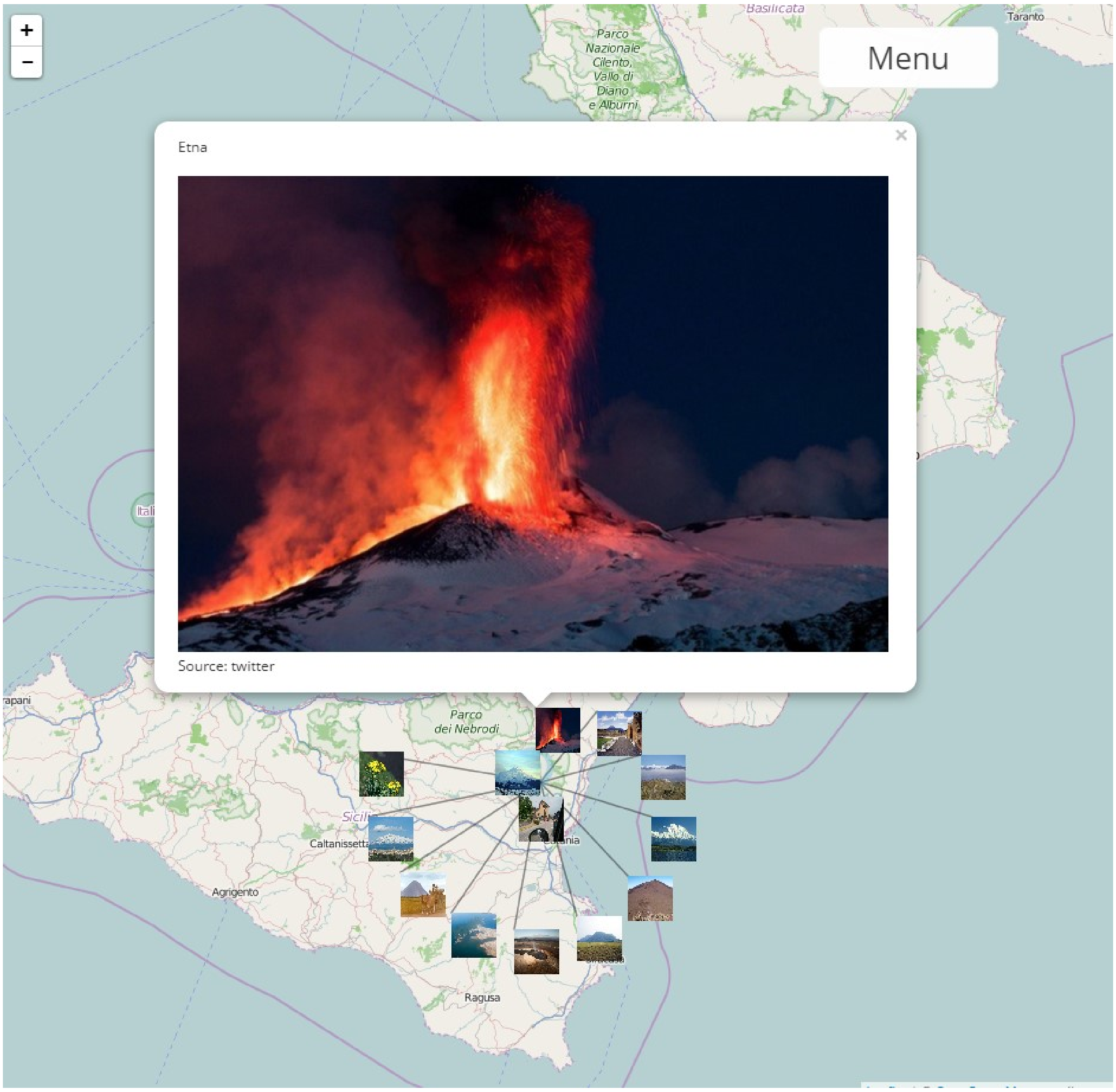
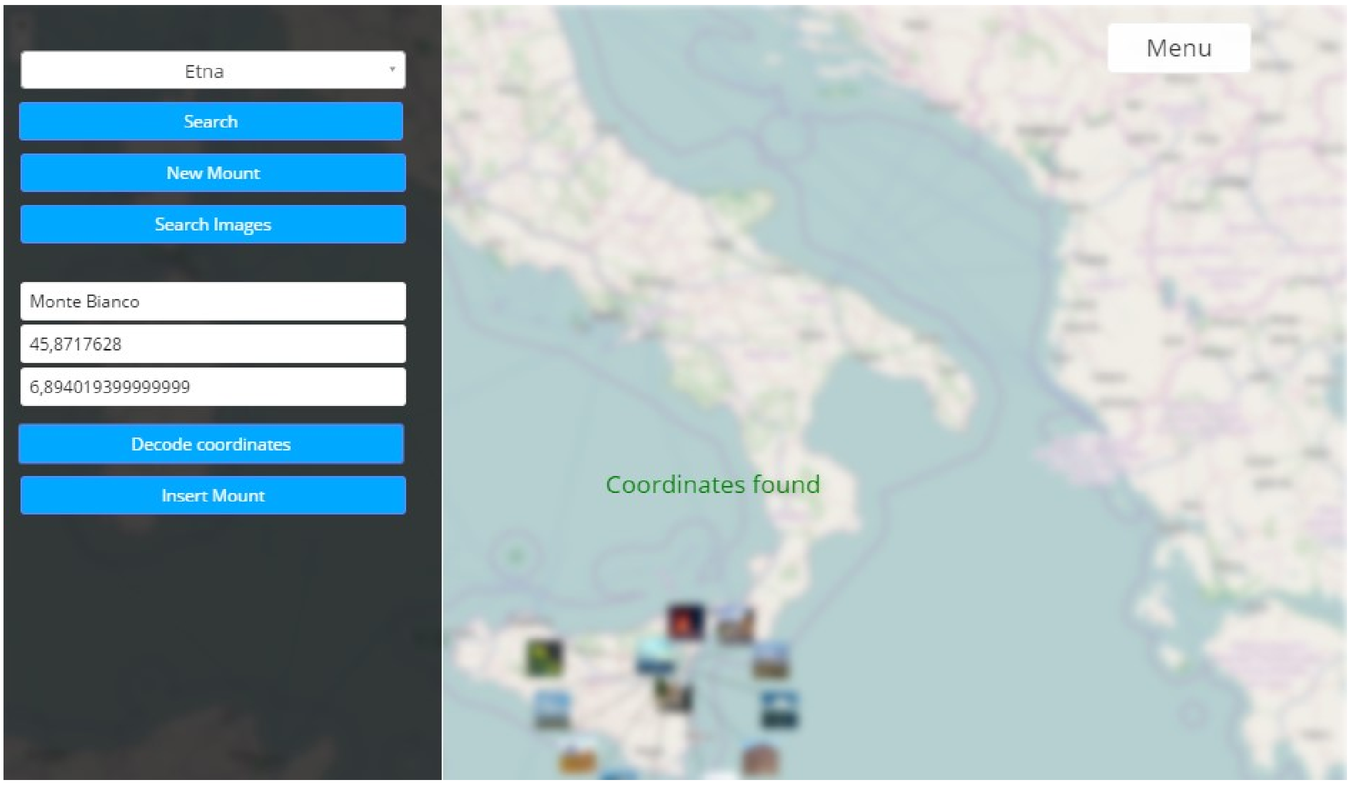


Figure 23 - Popup showing Image Information

The second button in the Menu allows inserting a new Mount, clicking on it a new form will be brought to the menu and we will be able to insert the name and decode the coordinates.

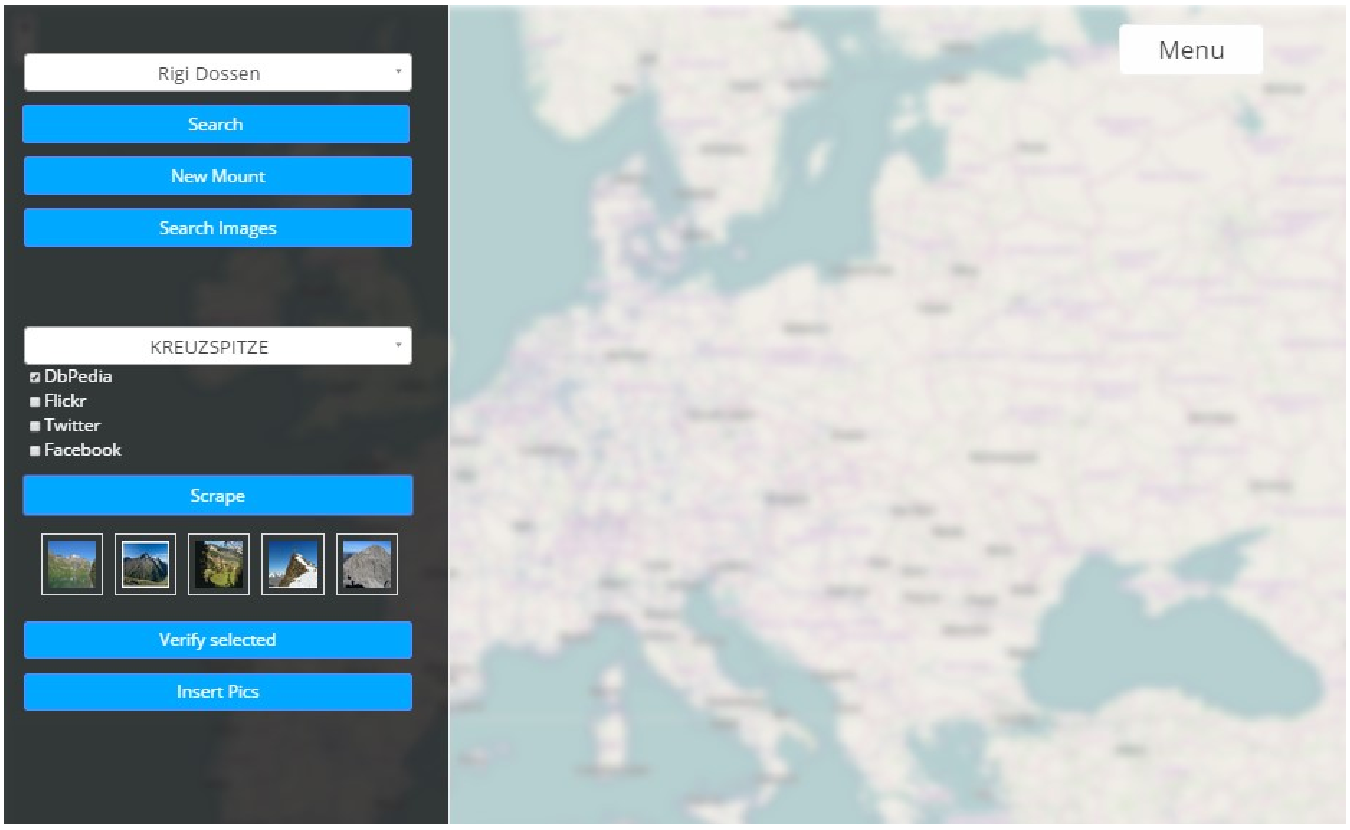


Monte Bianco
45,8717628
6,894019399999999

Coordinates found

Figure 24 - Coordinates decoding for a new Mount

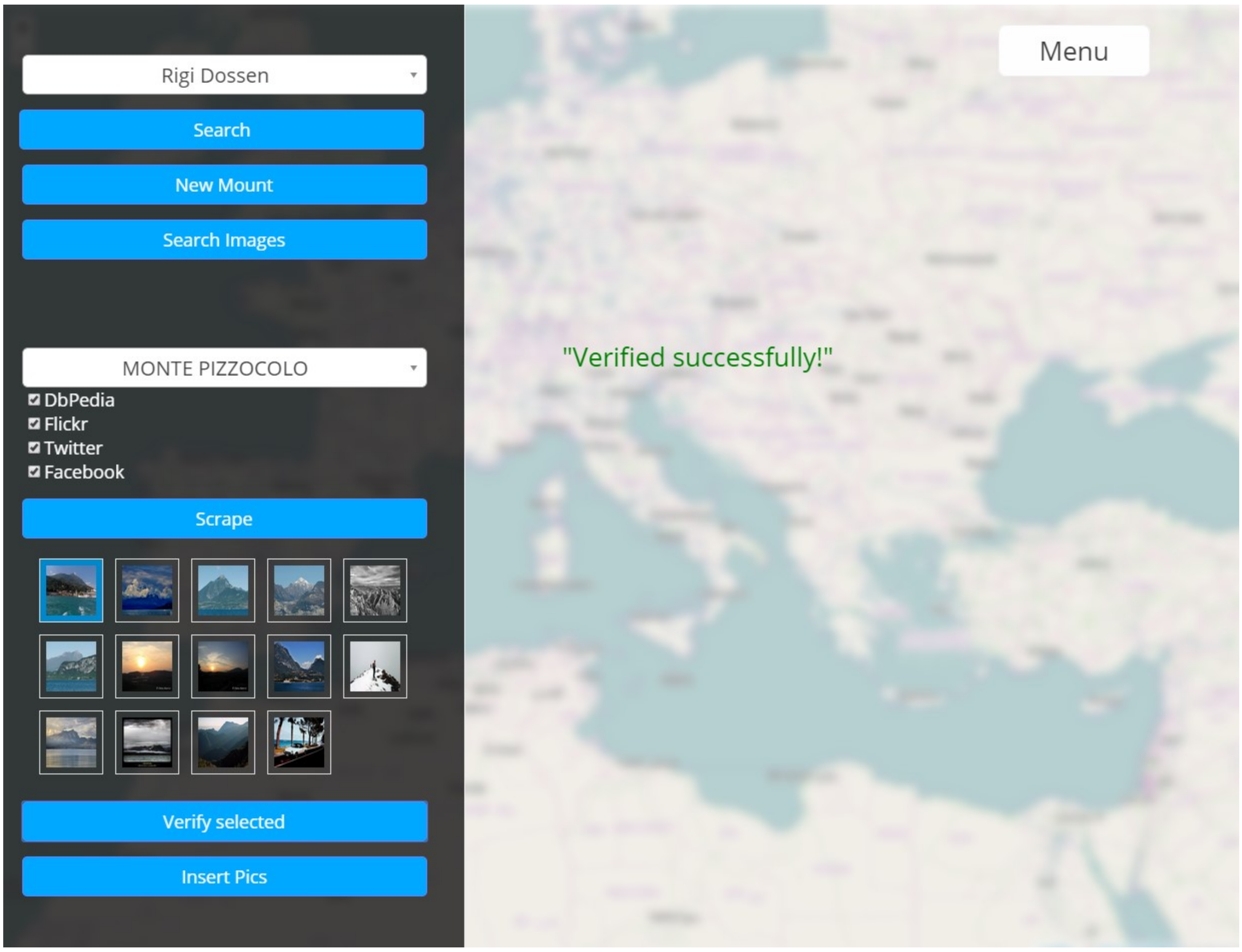
The third button permit to scrape for images over the available source, for a selected mount in the database.



KREUZSPITZE
☑ DbPedia
■ Flickr
■ Twitter
■ Facebook

Figure 25 - Image Scraping over DBpedia

When the images will be available, we can select an image to verify the content sending the request to the server.



"Verified successfully!"

Figure 26 - Content Image Verification

Then clicking on *Insert Pics* the selected images will be inserted into the database and will be immediately available for the selected mount.

Node.js Code

The structure of the code as mentioned previously follows an MVC pattern. It contains a controller named *main.js* that enable the routes answering to GET and POST requests of the view.
To have a better separation of the code all the functionalities to interact with the *crawlers.js* and *database.js*, which are the models, are implemented separately in a file *appFunctions.js.* This is still a controller on the MVC structure but just execute the call from the router *main.js.*

Here reported some of the key functionalities implemented in Node.js, regarding the controller and the modules.

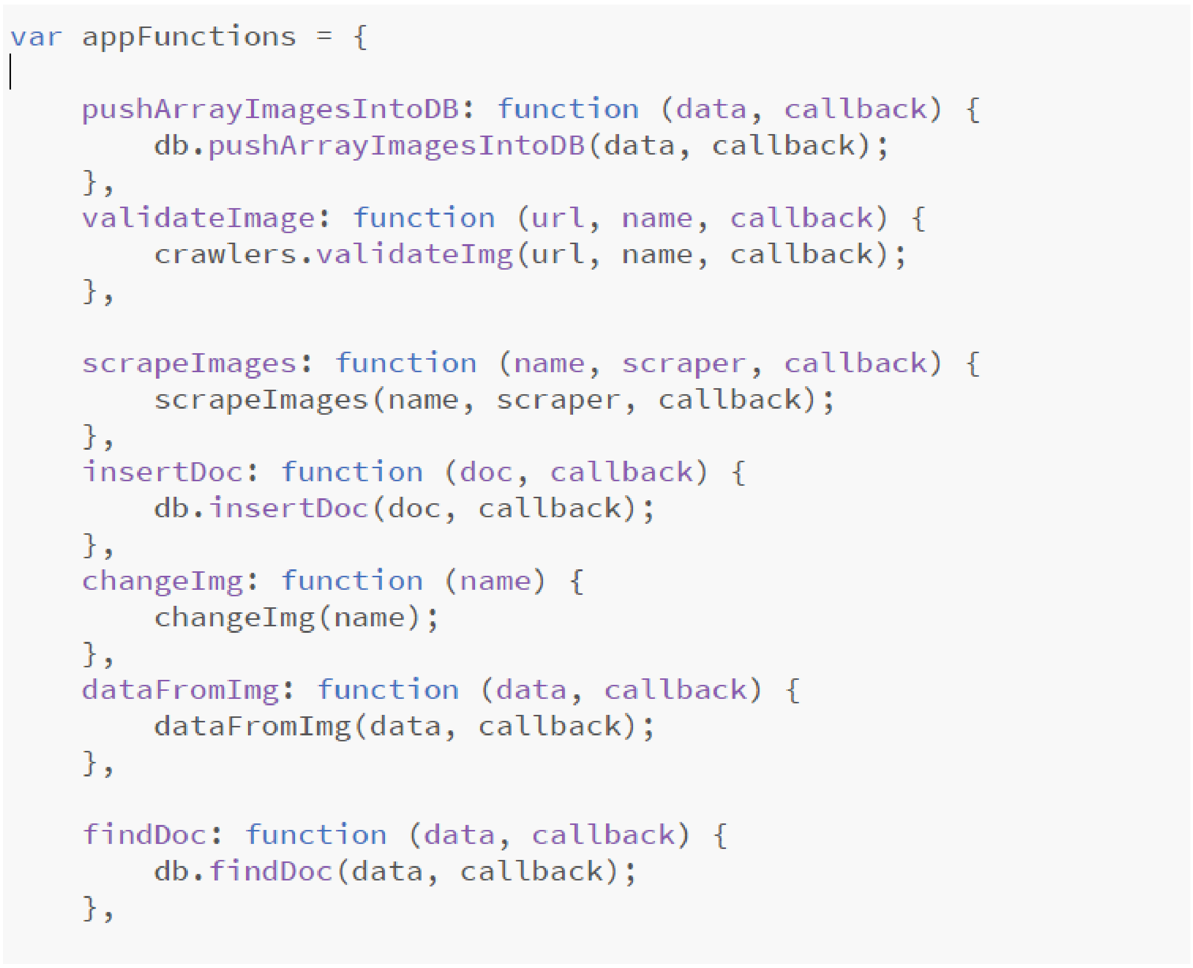
Some parts of the *main.js*, that implements the routes, is report here, in particular the POST method *new* to create a new Mountain Alps, reading from the AJAX request the useful information.

Also the POST for scraper that consists in scraping images from the services used given a name and an array of scrapers that have to be used.

```
app.post('/new', function (req, res) {
    var name = req.body.name;
    var lat = req.body.lat;
    var lng = req.body.lng;
    var mount = {};
    mount.name = name;
    mount.data = {};
    mount.data.lat = lat;
    mount.data.lng = lng;
    mount.data.img = [];
    console.log(mount);
    appFunctions.insertDoc(mount, function (dbres) {
        if (dbres != undefined) {
            res.send(JSON.stringify(dbres));
        }
    });

});

app.post('/scraper', function (req, res) {
    var scraper = req.body.scraper;
    var name = req.body.name;
    appFunctions.scrapeImages(name, scraper, function (img) {
        res.send(JSON.stringify(img));
    });

});
```

Figure 27 - main.js. POST for /new and /scraper

As mentioned, the *appFunctions.js* exposes its functionalities to the *main.js*, some of these are reported here.



```
var appFunctions = {

    pushArrayImagesIntoDB: function (data, callback) {
        db.pushArrayImagesIntoDB(data, callback);
    },
    validateImage: function (url, name, callback) {
        crawlers.validateImg(url, name, callback);
    },

    scrapeImages: function (name, scraper, callback) {
        scrapeImages(name, scraper, callback);
    },
    insertDoc: function (doc, callback) {
        db.insertDoc(doc, callback);
    },
    changeImg: function (name) {
        changeImg(name);
    },
    dataFromImg: function (data, callback) {
        dataFromImg(data, callback);
    },

    findDoc: function (data, callback) {
        db.findDoc(data, callback);
    },
```

Figure 28 - appFunctions.js exposed functionalities

Reported here, still in *appFunctions.js* the function to extract EXIF data from an image and get the geolocation.

```
function convert(deg, min, sec) {
    var dec_min = (min * Number(1.0) + (sec / 60.0));
    var answer = deg * Number(1.0) + (dec_min / 60.0);
    return answer;
}

function dataFromImg(img, callback) {
    var request = require('request').defaults({
        encoding: null
    });
    request.get(img, function (err, res, body) {
        try {
            new ExifImage({
                image: body
            }, function (error, exifData) {
                if (error) {
                    console.log('Error: ' + error.message);
                } else {

                    var gps = exifData.gps;
                    var lat = convert(gps.GPSLatitude[0], gps.GPSLatitude[1], gps.GPSLatitude[2]);
                    var lng = convert(gps.GPSLongitude[0], gps.GPSLongitude[1], gps.GPSLongitude[2]);
                    var coord = [lat, lng];
                    callback(coord);
                }
            });
        } catch (error) {
            console.log('Error: ' + error.message);
        }
    });
```

Figure 29 - appFunctions.js Extracting EXIF

Going then to the *crawlers.js*, that includes the methods to scrape the images from the various services, we can find the initialization of the APIs, setting the API key and other required settings.

```
//FLICKR SETUP
var Flickr = require("node-flickr");
var keys = {
    "api_key": "aed4cb6393cc26a9cfc293a48903d2b3"
};
var flickr = new Flickr(keys);

//TWITTER SETUP
var Twitter = require('twitter');
var client = new Twitter({
    consumer_key: '4edRbNwdrhTlb2HDQRXIaXjUn',
    consumer_secret: 'Ax0y2XYjfTFMA1HoOtdvKTL7ityYxIswXJZYokEhmhvx9sZABk',
    access_token_key: '469171537-q1yDPUUEoZV0g6bbi5iy55Ys8cPCz2usuXRei0aW',
    access_token_secret: 'z66J1OPb9tiHIVdbqTIXN9w7qBS5OgwyZ3FmrZtYwzdLQ'
});


//DBPEDIA SETUP
var SparqlClient = require('sparql-client');
var endpoint = 'http://dbpedia.org/sparql';
```

Figure 30 - crawlers.js Initializing APIs

Using the Sparql client, we created a method for the *crawlers.js* to retrieve all the mountains in DBpedia, getting the information that we need to store in the database. This functionality was used to enrich the number of mountains available in the database.

```
    getMountains: function (callback) {
        var query = 'SELECT * WHERE {  ?mountain a <http://dbpedia.org/ontology/Mountain> .  ?mountain <http://dbpedia
?name .  ?mountain <http://dbpedia.org/property/latD> ?latD .  ?mountain <http://dbpedia.org/property/longD> ?longD .}
        var client = new SparqlClient(endpoint);
        client.query(query).execute(function (error, results) {
            var x;
            var doc = [];
            var allName = [];
            for (x in results.results.bindings) {
                if (results.results.bindings.hasOwnProperty(x)) {
                    var mount = {};
                    var name = results.results.bindings[x].name.value;
                    var lat =  results.results.bindings[x].latD.value;
                    var lng =  results.results.bindings[x].longD.value;
                    mount.name = name;
                    mount.data = {};
                    mount.data.lat = lat;
                    mount.data.lng = lng;
                    mount.data.img = [];
                    if (allName.indexOf(mount.name) === -1) { //avoid dbpedia duplicates
                        doc.push(mount);
                        allName.push(mount.name);
                    }
                }
            }
            callback(doc);
        });
    }

};
```

Figure 31 - crawlers.js, querying DBpedia to retrieve mountains

Another method we used aims to retrieve pictures from Flickr, given an array of mountain names. It executes for each mountain a scraping query, and creates the link to be inserted into the database.

```
getFlickr: function (input, callback) {
    var y, arr = [], cycle=0;
    for (y = 0; y < input.length; y = y + 1) {
        flickr.get("photos.search", {
            "tags": input[y]
        }, function (result) {
            var obj = {};
            var name = input[cycle];
            obj.name = name;
            obj.service = "flickr";
            var images = [];
            var x;
            cycle = cycle + 1;
            if (result && result.photos) {
                for (x in result.photos.photo) {
                    if (result.photos.photo.hasOwnProperty(x)) {
                        var farm = String(result.photos.photo[x].farm);
                        var server = String(result.photos.photo[x].server);
                        var secret = String(result.photos.photo[x].secret);
                        var id = String(result.photos.photo[x].id);
                        images.push("https://farm" + farm + ".staticflickr.com/" + server + "/" + id + "_" + secret + ".jpg");
                    }
                }
            }
            obj.images = images;
            if (obj.images.length > 0) {
                arr.push(obj);
            }
            if (cycle == input.length) {
                callback(arr);
            }
        });
    }
},
```

Figure 32 - crawlers.js, scrape images from Flickr

Moving to the *database.js*, here the method for querying the database to obtain the name of all the available Mountain Alps.

```
function findDbData(callback) {
    MongoClient.connect(url, function (err, db) {
        assert.equal(null, err);
        var data = [];
        db.collection('mountain').find().toArray(function (err, docs) {
            docs.forEach(function (doc) {
                data.push(doc.name);
            });
            db.close();
            callback(data);
        });
    });
}
```

Figure 33 - database.js, querying database to retrieve mountains

Another method used in the *database.js*, consists in the functions to retrieve from the database, given a name, the document associated.

```
function findDoc(data, callback) {
    MongoClient.connect(url, function (err, db) {
        assert.equal(null, err);
        var doc = {};
        db.collection('mountain').find(data).toArray(function (err, docs) {
            docs.forEach(function (elem) {
               doc=elem;
            });
            db.close();
            if (doc) {
                callback(doc);
            }
        });

    });
}
```

Figure 34 - database.js, retrieving the document by name

Evaluation of Experience

The project has brought to the experimentation of new efficient technologies to develop web applications, scalable and fast at the same time. The MVC pattern has guaranteed the separation among the different components and thus the code can be reusable easily since the various components are also almost independents.
Some remarks can be mentioned about the decision to use some components and frameworks in a late phase of the project. An example can be given by Mongoose[18] a MongoDB modelling tool that would have simplify the management of the database operations.

Future Development

The project could be improved in the future and some new functionalities could be incorporated.
An important work could be done on the algorithm to recognize the suitability of the images related to the context. Regarding this, the service provided by google is for sure a very powerful approach but more researches should be done on how let this functionality work properly.

A new functionality that can be helpful into the context could be to let the application work in a reverse way, letting the user upload an image, to position then the image in the appropriate Mountain Alp, decoding the geolocation from the EXIF or positioning thanks to coordinates. This can be implemented efficiently, offering moreover an innovative service to let people contribute to the application.

[18] http://mongoosejs.com/

Table of Figures