DEPARTMENT OF TELEMATICS ENGINEERING

Design and development of a dynamic mobile guide using recommendations from affine users

Author: Mario Víktorov Mechoulam Nikolaeva
Advisor: Juan Vera del Campo

May - 2013

Acknowledgments

My most sincere gratitude to everyone who made this possible and believed in me.
To my family, whose support has always been there.
To the lecturers and friends at UPC, especially those from the Telematics Engineering department.
To the Pirate Party, flagship of knowledge freedom.
To Jeff Atwood and Joel Spolsky, for creating StackExchange and, particularly, StackOverflow.

Homo homini lupus

Abstract

En este proyecto se desarrolla LiveGuide, una guía turística para dispositivos móviles que hace especial énfasis en el dinamismo y personalización de las localizaciones recomendadas a los usuarios. Con LiveGuide, las personas tendrán el contenido de la guía en sus teléfonos móviles o tablets y deberán interactuar con él, mediante comentarios, puntuaciones y posicionamiento real GPS para avanzar y desbloquear nuevos contenidos. La información de la guía es almacenada en un servidor central y puede ser expandida y modificada fácilmente, posibilitando a los clientes sincronizar sus versiones locales. Los usuarios participan con un sistema de puntuación y opiniones que determina en todo momento la valoración de cada localización individual. Además, la información enviada por usuarios afines es analizada y, a través de un sistema de recomendaciones, a los clientes se les proponen nuevos sitios que podrían ser de su interés. El software móvil se ha desarrollado con un framework de alto nivel que abstrae la capa nativa (PhoneGap), permitiendo la portabilidad a cualquier plataforma, y mediante un servicio nativo que se ejecuta en segundo plano y controla la posición GPS del cliente.

This project describes the development of LiveGuide, a touristic guide for mobile devices focused on dynamic information and personalized recommendations of new locations. With LiveGuide, people will carry the guide content within their mobile phones or tablets and will have to interact with it, by means of comments, ratings and real GPS position in order to progress and unlock new content. The guide information is stored in a central server and can be easily expanded and modified, allowing clients to synchronize their local databases against it. In addition, the information from affine users is analyzed and, using a recommendation system, users will get a proposal of new places that may find interesting. This feature, linked to a self-categorization of the users, provides a personalized experience for every client. The mobile software is developed with a higher layer framework that abstracts the native layer (PhoneGap), allowing portability to any platform, and includes a native background service for controlling the GPS position of the client.

Table of Contents

1 Introduction

It is not a novelty that mobile electronic devices such as smartphones and tablets are present in the everyday life; it is even more known that software for them has long crossed the “phone-utility” boundary and flooded all fronts with some of the most eccentric and flamboyant applications ever imagined. In this frame, it is the author's interest to present an app that fulfills a social purpose – a tourist guide – and, at the same time, tries to offer an alternative to an already existing solution. A challenge factor to justify this project comes through the integration and interaction of different technologies and the addition of a social affinity system.

1.1 Purpose of this work

The main purpose of this work is to create a truly functional city guide application, LiveGuide, with information that changes and improves dynamically, based on the specific likes and dislikes of the users, while giving it a location-based gaming touch. The aim for this software is for it to be freely available for the greatest possible number of platforms and operating systems. From here, objectives can be divided in three categories: user oriented, technically challenging and others.

- User oriented objectives are those focused on changing the experience of the user or enhancing it. The major objective of the work is substituting printed touristic guides and make them available for mobile devices such as smartphones and tablets. However, this software wants to stray away from traditional guides and offer more of a gaming experience. As a direct consequence, the locations have to be unlocked by visiting and rating previous locations, being the user the one in charge of its own path. A parallel goal of the former is to deliver a new type of guide by the means of dynamically changing information that can adapt to the likings of each user. This is achieved first through the use of user profiles: the creation of an account under one of them produces a different output when progressing within the guide; the second step is to correlate user ratings and provide recommendations to every user on a fixed basis. In the end, the goal is a system whose database knowledge is composed by user rates and reviews– a system for the users, made by the users.
- The latter objectives are those that pose an academic or technological challenge to the fulfillment of the ideas and requirements described above. Some of them can score the difference between LiveGuide and a traditional application in terms of innovation, generated knowledge and complexity. One of these objectives is the multiple platform availability; a way to fulfill this goal is to abstract the native layer of the device and use a higher level framework which allows programming it once and running it anywhere. In order to progress in the application, the users have to physically visit available to them locations, while the detection of this action is done through the positioning providers (GPS, network). The positioning providers need to do frequent and periodic checks, even if the application is closed and the device is sleeping, while not consuming too much battery. The other major challenge from this category is the recommendation system itself. The system is a module inside the server, in charge of finding candidate users who have rated a portion of the same locations as those rated by the

soliciting client. The recommendation system will use a social metric to separate and classify users according to their similarity, and output a set of recommended locations as a result specifically returned to the client.

- Besides those, there are some other singular goals which do not really fit in neither of the other objective categories of this report. However, for the sake of completeness, they would be: try to deliver less known locations or at least not the typical places for tourists, achieve a really large user base and monetize the app. The first is not difficult to achieve, but it requires time and field work, as well as a good bit of knowledge and some luck in finding and promoting those places. The large user base comes partly by a quality and worthy app, partly by good marketing skills and visibility, which can be provided by friendly hostels and hungry businesses. Finally, the app will still remain free to use, but can accept small payments to promote worthy locations and make them appear on the app or push their position towards the first places.

The end result is a client server architecture on which the server acts as a central point, in charge of managing users, administering database, calculating recommendations and executing the requested actions by the users. The server can serve multiple clients and do so concurrently. Each client stores locally a reduced synchronized version of the server database and provides mechanisms for the user to create accounts, update and view the database information, visit and unlock locations of its choice (known as progression) and evaluating them, submitting a rate and a review to the server.

1.2 Document structure

This document is divided in several chapters in order to ease its reading and guide the reader through it by incrementally building a general picture of the system until the full disclosure of the details is given.

The first chapter is dedicated to introduce the project, define its goals and explain the structure for the rest of the document.

In the second chapter, the frame of the application is set; two typical scenarios are narrated and the potential usefulness of LiveGuide is shown. Following, the used technologies to make this possible are declared and briefly explained. The less known and used of them are given a bigger focus, while the commonly extended ones are quickly listed. Also on the second chapter is found a complete functional analysis of the application, pointing what can be done with it, how and where. To conclude it, the chapter provides the readers with the architecture the system is implementing, along a small resume of what can be called “functional modules”.

The third chapter is dedicated to the storage systems (the databases). Since their implementations have already been introduced in the former chapter and the full details of their configurations and set up will be provided in the client and server chapters respectively, this chapter is left to specifically describe the structured information model. Here, both the server and the client databases have their tables and fields explained, scripts for their creation is attached and, finally, the full relational diagram is embedded.

Chapter four belongs exclusively to the server. It starts by defining its structure and explaining how the Model-View-Controller (MVC) paradigm is implemented. Next, a section for the server-database set up is included, providing an insight of the Object-Relational-Mapping (ORM) and how it is used to interact with the model. Right after that, the available actions on the server are stated, backed up by how to invoke them, the needed parameters to be set and how the server reacts on those. Chapter four concludes dedicating two sections to two major topics: the implemented recommendation system along the mathematical theory it is based on and the security measures included on the server side, which help to protect it from the most obvious threats.

Afterwards, on chapter five, it is the client's time to be seen in-depth. Once again, the structure and the

process of interaction with other elements are detailed, but the most important of its sections is the one dedicated to the application *per-se*. A complete lifecycle of the typical usage is described; then, a technological challenge is introduced, conveniently followed by a solution proposal and its implementation; closing the section, the geo-location mechanism and the way it has been used in the application are explained.

Chapter six is dedicated to the evaluation of the results, first comparing with the initial goals that were set and afterwards adopting the point of view of a theoretical generic user. In addition, in this chapter further improvements and enhancements are proposed, as well as the future guidelines which will set the path of this project.

The last chapter is dedicated to the conclusions drawn from the design of the whole system and its comparison to the end results, joint by the generated knowledge while doing its development.

2 Project insight

LiveGuide extends across several different technologies and layers. This chapter starts by framing the scenario in which the project could be used and brings two different use case examples. After that, the strong points which could provide usefulness in the described scenarios are stated, before going over all the technologies and libraries used in it. To conclude the chapter, the architecture of the whole system is shown and separations by modules are noted below the diagram.

2.1 Use cases

This section introduces a temporal and spatial frame in which the action involving the possible use of the project or of its alternatives would take place. A couple use cases are briefly displayed, only to close the section by showing the benefits of using LiveGuide in any of the former situations.

Next are some of the typical situations in which the application would prove handy. The first use case simply exposes some of the issues that may apply for specific segments of visitors or even some which are applicable to all tourists, while the second tries to show a daily usage of the here presented application.

2.1.1 Substitution of printed guides

A group of friends with similar likes is planning a trip to Barcelona; they have already purchased the tickets and booked a room near the center, but do not know anything about the city.The group would like to enjoy the town at its best, visiting well-known locations, tasting delicious food and drinks and having a lot of fun with nightlife. However, they would like to avoid the typical overcrowded places for tourists, especially the ones involving eating out, and would love instead to try out less known locations or those that are visited by the locals.

A printed city-guide might seem a solution at first, but they do not feel like spending money for information that is mostly available for free. Moreover guides do not use to be *live*, which typically leads to outdated, wrong or incomplete information and little or no variation between editions. On top of that, they perceive that they would be wasting a significant part of the guide, as the information it comprises is huge and the time to read is not abundant, not to talk about the overwhelming choice through which people risk not visiting really important locations if the guide is not examined thoroughly. This could be even worse if the group decides to look for free information on the Internet, as it is scattered and should be correlated, making them spend even more time on this matter.

2.1.2 The game of tourism

A user arrives at his hotel in the city he is visiting and sees the application being promoted at the front

desk. The user decides to try its features and downloads it via the QR code on a flier or through the application repository of his mobile brand.

After installing and opening the application, the user is asked to select a profile type that matches his likes and tourist style and prompted whether he would like to get recommendations from other users based on the rates and reviews he is going to make. The client accepts the recommendation system and selects a profile. As soon as he does this, the application goes through a couple of information screens in order to describe how to use and progress in it.

Once this information is read, the user sees the main screen of the software on which he finds three tags reading different location categories. Under each of these tags a couple of locations are placed and when the user selects any of them he is able to read some information about the place, see a photo or two and locate it on the map. With this information the user decides to visit a couple of this locations during the day and, maybe, another two at night.

The client moves to the first location and spends some time reading the information about the place, taking photos, enjoying the architectonic art... once he is done, rates it on the mobile device and writes a short review about the place, actions that reward him with the possibility to unlock more location on the categories he chooses. At lunch time, the user checks the application and chooses one of the available places to eat. While waiting for his food order he chooses to unlock a couple places, so he visits one more location in the afternoon and decides to enjoy the city's nightlife by going to a pub first and to a night club later. At the end of the day (or at the next morning) he is at his hotel at last. The system downloads any available updates and recommends additional locations to the user (on a different layer that those unlocked by tokens) based on how he has rated the locations he has already visited and correlating those with the results of other users of the application.

When the user wakes up and starts his second day at the foreign city, he not only has a much wider choice, given by how the implementation of location unlocks work, but also a more accurate one to his real likes because of the recommendation system.

2.1.3 How can be useful?

This software brings a solution to most (if not all) of the issues presented at the former section. To start off, it is completely free, can be carried around in mobile phones or tablets and its data is *alive*, meaning it is updated periodically and accordingly. In order to help the users navigate through the information and make right choices, the application categorizes them by means of profiles and delivers locations in small quantities based on importance, rates or types of location. All this conforms a suit of rules which are presented to the users as a game; in order to progress and unlock new locations the players must visit a portion of the set that is already shown and rate it. The aim of this system is to guarantee that at any time the locations offered to a given user: fit to his profiles and needs, are reduced (but not small) allowing the user to choose fast and its rates and reviews can be quickly checked to decide whether it is worth visiting them or not. On top of that, a recommendation system is bundled within the application offering additional locations to visit from time to time, by correlating the rates and reviews of the user with those of other users of the software, similar to what Netflix does. Those recommended locations presume to be extremely accurate and close to what the user likes. To conclude, an additional goal of this software is to promote the tourism to less known locations or places visited by the local population, in order to make every trip different and break up with *clichés*, a feature generally desired by most tourists.

2.2 Requeriments

During the initial stages of planning and designing of the project a group of goals were set. Each of these goals translates into a feature, which has specific needs. The next list states the requirements of the project for reference during both this reading and at the result and evaluation stage.

- ✓ Portable touristic guide.
- ✓ Compatible with most mobile operating systems.
- ✓ Dynamic and updated information.
- ✓ Gaming-like features.
- ✓ Geolocation for finding out user position.
- ✓ Progress is based on user actions:
    - Visiting a location requires to be physically present at it.
    - Users are asked to rate and review a visited location.
    - The above process gives 'tokens' for unlocking more locations.
- ✓ Location scores are based on user marks.
- ✓ Personalized experience:
    - Recommendation system offers recommended locations at some times, produced by correlating rate data against other users and calculating their affinity.
    - Different user profiles (affect unlock order using specific profile sub-segment scores).
    - User can choose the location category from which he wants to unlock more locations.
- ✓ Location and score information on mobile devices can be synchronized against the server.
- ✓ A minimum level of security to avoid user impersonation and ciphered communication.
- ✓ Get to substitute traditional paper guides (and digital where possible).
- ✓ Offer less known places, often only known by locals and difficult to find elsewhere.
- ✓ Usability aspects: smooth use, attractive design, minimum connectivity, GPS availability.
- ✓ Performance: small memory footprint, low battery drain.

2.3 Technologies

The purpose of this section is to detail the technologies that form the project and to explain the motives that led to them being chosen for the task. Since this work is a composition of different pieces of software, it is important to remember that some of the technologies are present only in the client (Phonegap, SQLite), some only in the server (JavaEE servlet, MySQL) and some are found on both (JSON). In these systems, albeit the roles of each piece of technology are clearly delimited, they are used in conjunction – boosting their synergies – for the achievement of a greater goal.

2.3.1 JavaEE

On the server side, the language of choice has been Java with the Java Enterprise Edition[1] bundle. Java has been a solid option for web services for quite some years now, with several solutions and huge support and documentation. Besides the Object-Oriented feature, the Servlet implementation and the

author's experience have been the key points for deciding to go with the language. The developer environment also plays a major role; in this case, the Netbeans IDE[2] has been selected given its excellent Web Application framework, the Persistence API support, the useful wizards for configuration and technology merging easiness, the close ties between the language and the IDE's developer (Oracle) and the wonderful option to bundle everything with a perfectly fitting, EJB 3.0 compliant Glassfish[3] server.

2.3.2 Phonegap

The idea for the client was similar to the Java *motto*, "write once, run anywhere", pursuing to overcome the handicaps of different programming languages and very heterogeneous systems for today's mobile devices. To achieve the purpose of write the application once and porting it without further (major) effort to every available platform a couple solutions were a possibility, all of them abstracting the programming to a higher layer and acting as a bridge to the native language of the platform (Fig.1).

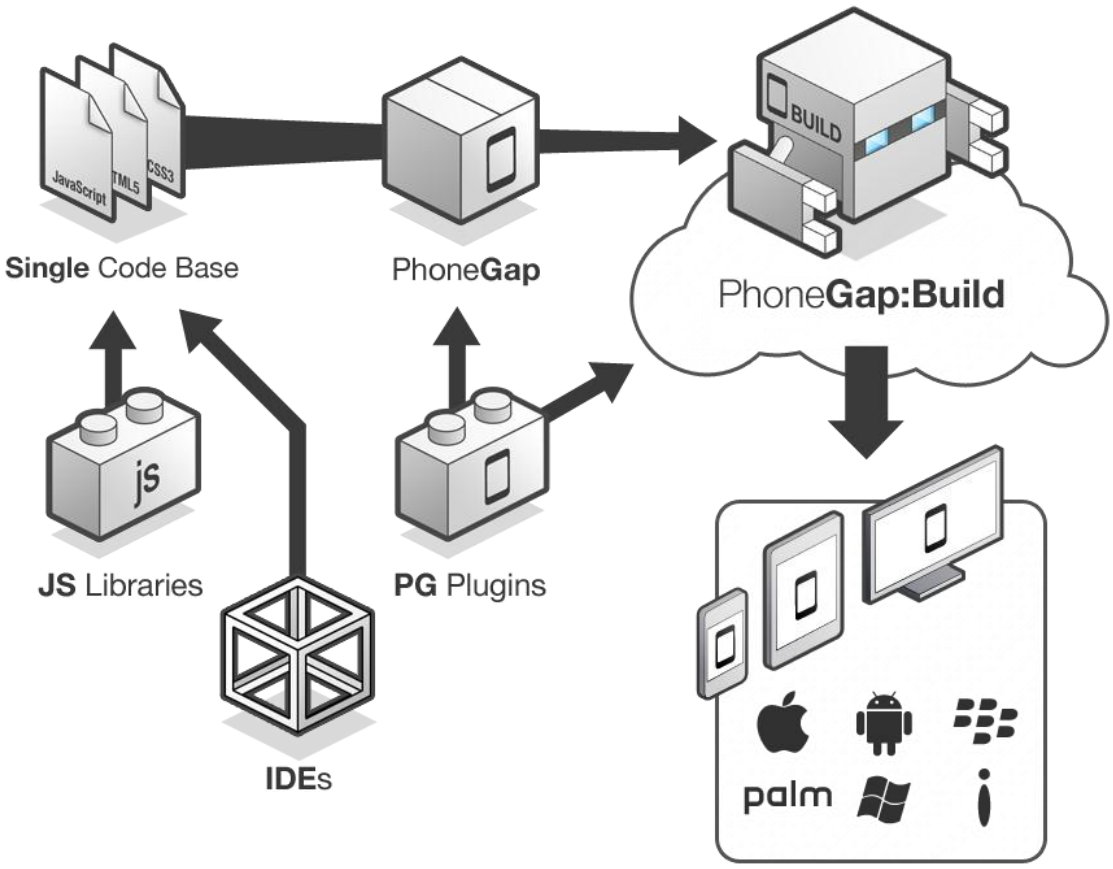


Figure 1- PhoneGap Build

Among these, the PhoneGap framework (now Apache Cordova)[4] is the most appealing one, offering the closest true cross-mobile building through the use of standard web-based technologies to bridge web applications with mobile devices. On top of that it is free and open source, very valuable characteristics in the IT and software environments. To code an application with PhoneGap the standard web languages may be used; this includes HTML (full support to version 5)[5], CSS[6] and JavaScript[7]. In addition, network protocols such as XMLHttpRequests[8] or WebSockets[9] can also

be used to enhance the project with enterprise level processes access on remote servers (Fig.2).

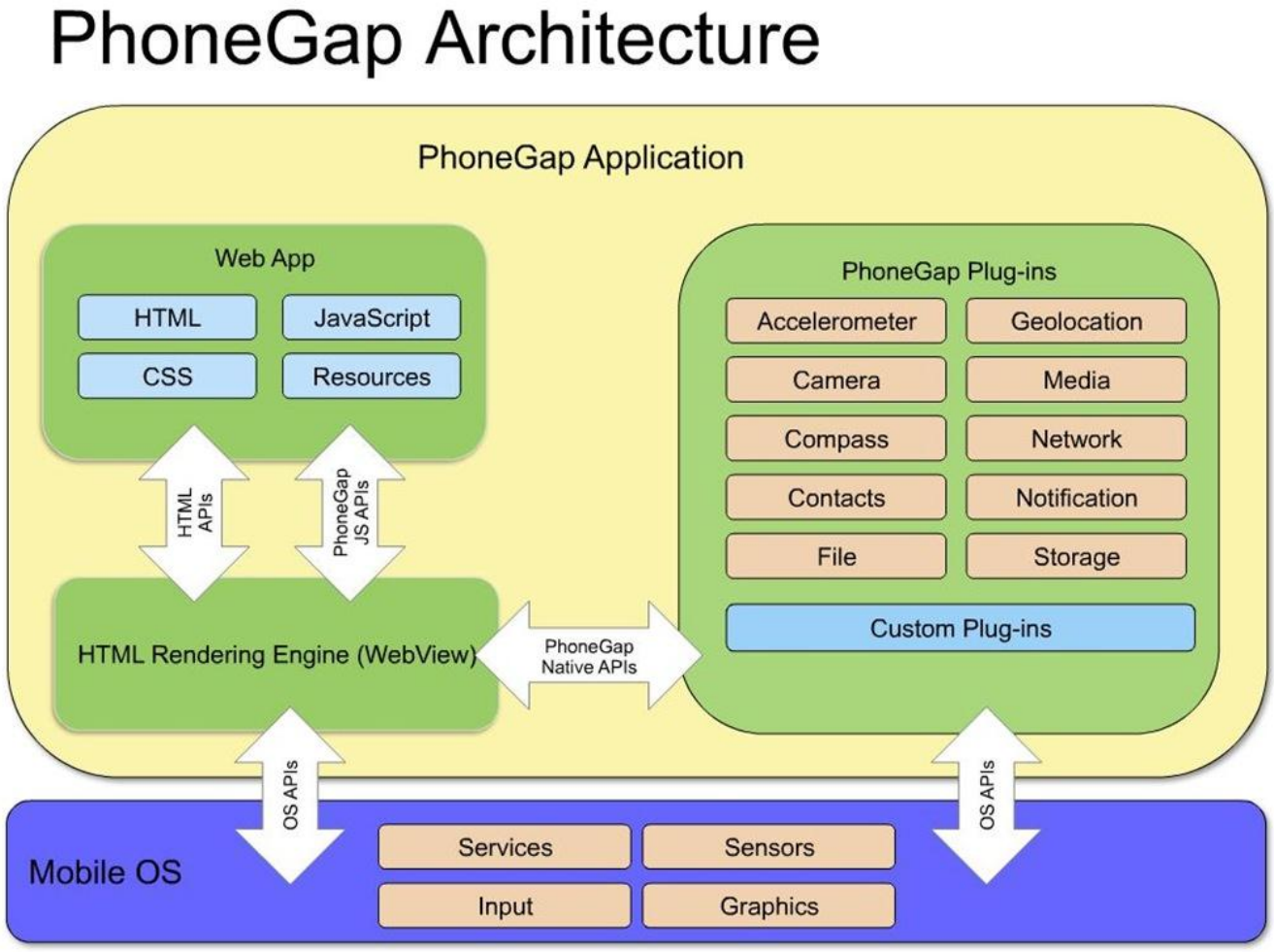


Figure 2 - PhoneGap Architecture

The demonstrator of this project is one such application running on the Android[10] platform (the author's mobile brand operating system), handling all presentation layers, inputs and events from a web-browser like Android WebView component, started within a main Activity. For other platforms, this initial setup could differ a bit, but it is still handled by the PhoneGap frame work

2.3.3 jQuery Mobile

Besides the use of the standard jQuery library, a mobile version of the framework is available. jQuery Mobile[11] is a user interface framework based on jQuery that works across all popular phones, tablet, e-reader, and desktop platforms. Built with accessibility and universal access in mind, it follows progressive enhancement and Responsive Web Design (RWD) principles. HTML5 Markup-driven configuration makes it easy to learn, but a powerful API makes it easy to deeply customize the library. jQuery Mobile comes with an integrated user interface built on jQuery Core, backed by the bundled CSS files and is structured in 'pages' and other defined sections for the HTML documents. The logic and event handling can still be managed through JavaScript or standard jQuery.

The navigation through the different sections of the site is done using AJAX[12], which intercepts button clicks or forms submission, allowing also for an internally coded back button. Inside the section containers, a wide choice of widgets can be added in the form of elements: from buttons, listviews and compounds, to dialogs, pop-ups, navigation bars and floating panels; the options are endless. In

addition, an easy integrated mechanism of changing themes of the elements is also provided. All in all, it has proved a worthy and helpful framework for the development of this project. Following, some examples can be seen in code snippets (Code.Snippet 1, 2).

```
<div data-role="navbar" data-grid="c">
    <ul>
        <li><a href="#" class="ui-btn-active">One</a></li>
        <li><a href="#">Two</a></li>
        <li><a href="#">Three</a></li>
        <li><a href="#">Four</a></li>
    </ul>
</div><!-- /navbar -->
```

Code.Snippet 1 - Navbar

```
<ul data-role="listview" data-inset="true">
    <li><a href="#">
        <img src="../../_assets/img/album-bb.jpg">
        <h2>Broken Bells</h2>
        <p>Broken Bells</p></a>
    </li>
    <li><a href="#">
        <img src="../../_assets/img/album-hc.jpg">
        <h2>Warning</h2>
        <p>Hot Chip</p></a>
    </li>
    <li><a href="#">
        <img src="../../_assets/img/album-p.jpg">
        <h2>Wolfgang Amadeus Phoenix</h2>
        <p>Phoenix</p></a>
    </li>
</ul>
```

Code.Snippet 2 - Page Structure

2.3.4 HTML5SQL

Transactions have been done using the html5sql library[13]: a module coded in JavaScript, implementing the WebSQL[14] API that eases the work with HTML5 Web Databases. The API is supported by the major web browsers (Chrome, Firefox, Opera, Safari...) including those available on the smartphones. The module provides a frame to sequentially process SQL within a single transaction, whether they are single strings, arrays, statement objects, multiple concatenated query strings or even loading the statements from a separate file. On top of that, it simplifies the version control of a database. Some details of the module can be found next, while its usage is shown in the respective section in the client chapter (Code.Snippet 3, 4).

```
html5sql.openDatabase(
     "com.mycompany.appdb",
     "The App Database"
     3*1024*1024);
```

Code.Snippet 3 - Database creation/access

```
html5sql.process(
     [
       {
         "sql": "INSERT INTO contacts (name, phone) VALUES (?, ?)",
         "data": ["Joe Bob", "555-555-5555"],
         "success": function(transaction, results){
             //Just Added Bob to contacts table
         },
       },
       {
         "sql": "INSERT INTO contacts (name, phone) VALUES (?, ?)",
         "data": ["Mary Bob", "555-555-5555"],
         "success": function(){
             //Just Added Mary to contacts table
         },
       }
     ]
);
```

Code.Snippet 4 - Queries

2.3.5 Databases

Databases are powerful instruments useful as a storage container for structured data. The schemes can be as complex as required by the application and with adequate actions the information stored inside them can be retrieved, modified or deleted.

- ***MySQL***

The implementation of the database on the server side has been done using MySQL[15], however its design should allow for an implementation on any other system. The reasons behind this choice have been a highly scalable and free system, good performance and availability, support of robust transactions and ease of management. On top of that, the author already had former experience with it in several other projects. The storage engine of choice has been InnoDB, given its transaction and foreign key support. Also on the plus side is its row locking features and capability to group multiple INSERT queries.

- ***SQLite***

On the mobile devices the adopted database implementation is SQLite[16] for all the major platforms and operating systems. The adaptation for most of the tables, variables and column characteristics was pretty straightforward; some minor modifications had to be made though. The approach consisted in dumping the server database with the *mysqldump*[17] tool and parsing it to SQLite with mysql2sqlite[18] script. Nevertheless, some variable types had to be manually edited (like Timestamp fields), engine enforcing rules for tables had to be removed and every start or end transaction mark had to be deleted, since the project relies on specific libraries to control that behavior.

An initial version of the database (up to date with the current application release on the repository) is bundled within the project. During the first execution, the tables are created and the information is stored inside. Code can be examined at the Annex.

Finally, transactions have been done using html5sql library, as described in the former section.

2.3.6 Communication

LiveGuide is an application composed by applications or, as it is known best, a distributed application. The need of interaction and communication between the parts is nowadays a hot topic; different solutions exist for different scenarios, technologies and hardware. The solution for LiveGuide comes by the usage of two different technologies: XMLHttpRequest (XHR) for message transmission and JSON for a common way to represent the contents of those messages.

- ***XMLHTTPRequest***

The XMLHttpRequest (XHR) is a specification that provides functionality through an API for data transfers between client and server. It is *de facto* implemented within JavaScript.. Albeit the official name, any data can be interchanged with its use, secure communications are also supported (HTTPS) and 'HTTP Requests' is used in a general sense, meaning that all the HTTP methods are actually supported. In order to use an XHR, an XMLHttpRequest object implementing the interface with the same name has to be created. Communication can be prepared with the 'open' method, indicating the end-side URL, included parameters and synchrony (among other options) and later executed through the 'send' method. A listener can check and inform the client end of changes in the state of the connection; for LiveGuide client, only the 'onreadystatechange' status is reacted upon, attaching a function to check for the request '200 OK' status and process the data inside the response. Next, a simple example of this functionality is presented (Code.Snippet 5).

```
var request = new XMLHttpRequest();
    request.open("GET", url, false);
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
                if (request.status == 200 || request.status == 0) {
                    if(request.responseText != null &&
request.responseText != ""){
                        var data = JSON.parse(request.responseText);
                        //Do stuff
                    }
                }
        }
    }
    request.send();
```

Code.Snippet 5 - XHR

- ***JSON***

JavaScript Object Notation (JSON)[19] is a lightweight data-interchange format. It is said to be the ideal language for this purpose given it is easily parsed and unparsed, presents a human readable format and it is language independent. Two types of structures are allowed in JSON: a collection of name-value pairs (known as an object in the language) and an ordered list of values (representing the array).

- Any object is bound by the '{' (left brace) and '}' (right brace) characters. Inside any object one or more members may exist. A member is a string-value pair separated by a colon ':' and with both the string and the value is surrounded by double quotes in most of the cases. The different string-value pairs are separated by commas.
- The array is bound by '[' (left bracket) and ']' (right bracket) and contains a variable number of what are called elements. Elements are just like the values and they can be anything, from a

primitive type to other objects, arrays, null...

This format allows for parsing of complex data structures with any desired depth since, as it can be seen, nesting is possible. As noted in the former chapter, the parsing is done by the Google GSON[20] free libraries on the server and by the native JavaScript implementation on the client. The resulting JSON data structures are sent and received between the server and the client as an easy way to transmit any type of objects, arrays or primitive types. The following are some examples of JSON parsed data used in this project (Code.Snippet 6).

```
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup
Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used to
create markup languages such as DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

Code.Snippet 6 - JSON example

2.4 Functional analysis

This software is thought only for mobile platforms such as smartphones and tablets. Given the technologies used for its crafting, it can be tested up to a certain degree by any device that supports databases and has a browser for HTML/CSS/JS processing and displaying. Nevertheless, to truly utilize its features, a GPS for geolocation and wireless connectivity for communicating are a must.

2.4.1 Download

The application has to be downloaded from any of the major OS' repositories, the central storage and information point where submitted apps are uploaded, either navigating manually to it or scanning a QR code. With the aim of maintaining consistency between versions and, most importantly, database updates, the latest version of the database is bundled along with the application. The key factors here are the date and time values of this database, as that represents the latest update that has been made to it; more about this in this same section.

2.4.2 Installation and database setup

The application is installed and unpackaged in its natural location depending on the hosting operating system. At the first execution, a small script code runs to ensure that the database is created from within the application. A variable is set inside the app's 'localStorage' to mark the database as installed and to ensure this step is avoided in further executions, allowing for a smoother boot. Specific code for this functionality can be checked at the client disclosure chapter.

2.4.3 Account and profile setup

After the former has been done internally, the user is presented a single select element with different profiles and asked to choose the role that defines him best. Next, the user is presented with the 'Log in'-'Sign in' screen; a simple prompt with text areas in which to input a user name and a password and two buttons below. The following situations may be possible:

- If the client has no account on the server, the 'create' button press, after selecting a profile and filling the input fields will yield him one (supposing there is no conflict with existing user names).
- If the client has an account on the server, after filling this information he should use the other button for log in purposes.
- If the client has an account on the server, the application remembers his info; the text on the 'Login' button is slightly modified and labeled after the user name, directly allowing him to login, while the other allows creating a new account.

Only in the case of creating a new user does communication occur with the server (as the user table is also present locally) and the sent parameters are the chosen profile and the last updated date and time of the database. The server response is either an error or a complete 'user' table entry, which the client stores locally. In addition, the server sends three ordered arrays with the locations in a sequence determined by the scores received by the user's chosen profile. This will allow the ‘unlock’ to occur in a from-best-to-worst fashion. Diagram can be seen at (Fig. 3).

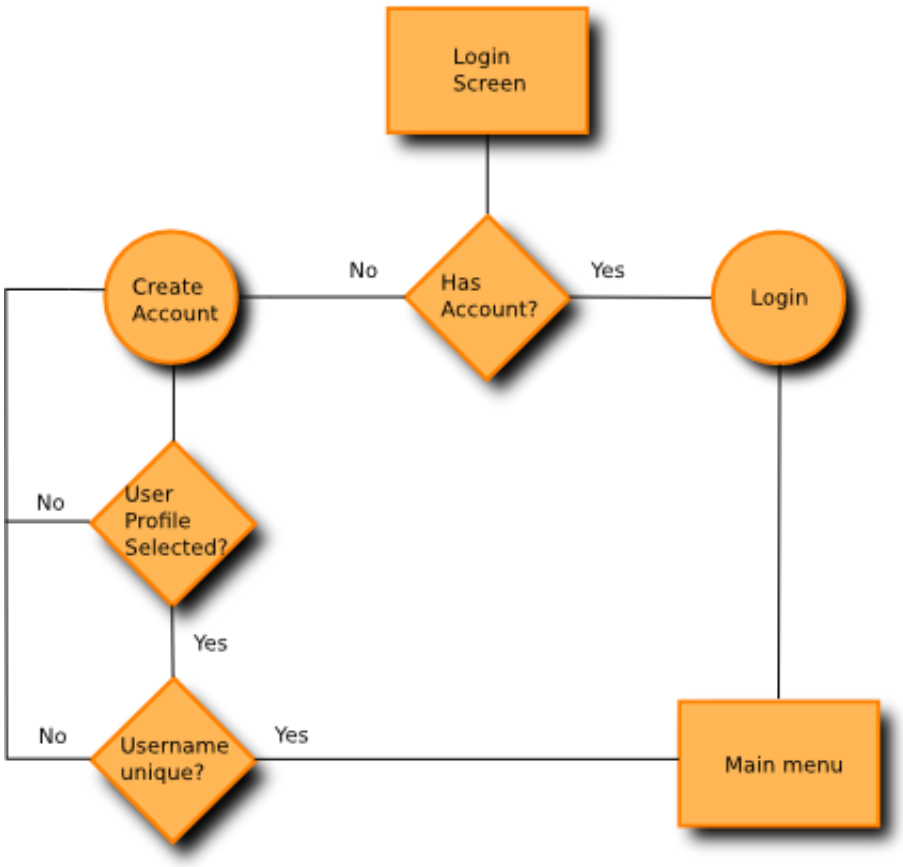


Figure 3 - Login diagram

2.4.4 Navigating the app

Once logged-in, the user can start to discover the offered locations, conveniently orchestrated by the software. These locations are displayed by clicking on the 'Discover' button and are presented grouped in four different tabs through which the client can flow. The tabs represent categories with a certain orientation, such as sightseeing, historical and architectonic sites; eating out and *tapas* places; or nightlife points like pubs, discos or musical events. The last tab is used to store recommended locations made by other users, based on the affinity deducted by made rates; it is initially empty.

On this same screen, there are three more actions that a user can do:

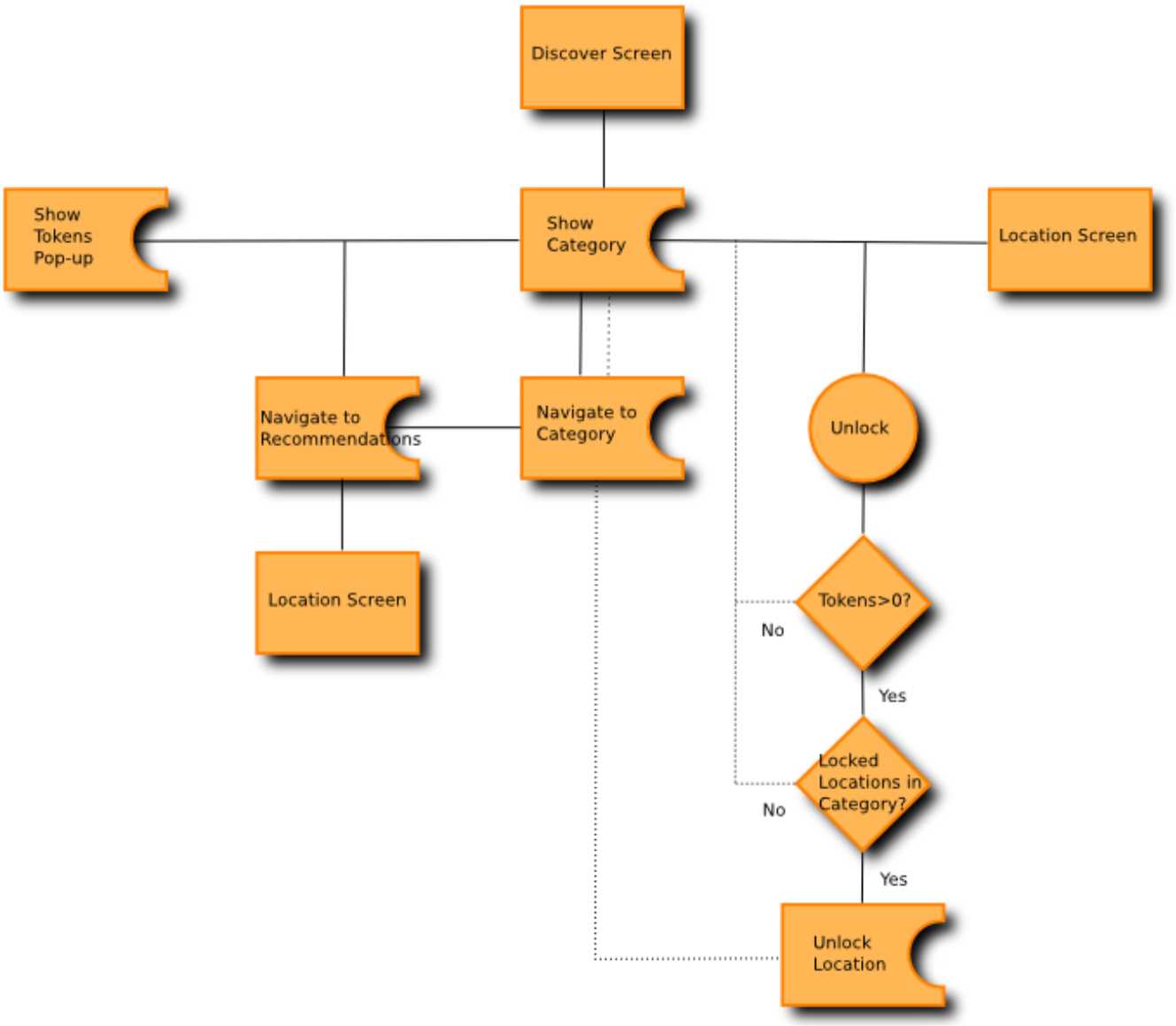
- Unlock a location in one of the categories; this is not possible for the tab containing recommendations for obvious reasons. The outcome depends on two factors: whether there are available places to be unlocked under that category and the user having enough tokens to unlock a location.
- Check the tokens in possession. The number is displayed at all times in the upper right corner of this screen – clicking on it displays some additional information.
- By clicking on any of the locations, the user can review detailed information about it and proceed towards additional actions in a new screen.

The location screen presents all the data available in the database about the place. This includes the full name, the average score derived by the user's rates, some photos that expand to full-size when clicked, information about the place and two buttons at the bottom. The buttons are used to summon a sliding

panel with the most recent reviews about the place made by other users and to open a pop-up dialog with the placement of the location on the map. Ideally, a dynamic map engine like Google Maps would be much more helpful, but since one of the main purposes of the project is to be a strong candidate to replace printed guides and most of the foreigners do not come with a data connectivity plan (Wi-Fi is still far from being ubiquitous as in other big cities) static image maps are the embedded solution. Diagram is found in (Fig. 4, 6).

2.4.5 Visiting a location

The GPS of the device checks periodically for a match between the current position and those of the unlocked not-visited locations. A marginal error is included on purpose to allow for a looser wake-up of the system for the mathematical operations and to prevent missing a visited target. In case of a match, a notification pops-up on the device's toolbar and when the user opens it, he is informed of the name of the location and that it has been marked as visited.



[1]

Figure 4 - Discover and unlock

2.4.6 The rate and review system

On the location screen, the client can check useful information about the selected physical place (Fig. 5). The official name, an informative text, images, a map and even some of the latest reviews submitted by other users about the place are included in this page. Also on this screen, users can confirm how much they have progressed with the selected location: whether it has already been visited or not and whether the rate and the review have been submitted. In addition, if the location was the result of a recommendation process it is also conveniently stated here. Finally, if one location has been unlocked and visited, the rate and review action can be done from within this same page, as explained in the next section and in (Fig. 5).

[1] Rectangular shaped boxes indicate application screens; diamond shaped refer to condition evaluation; circular elements are server interaction through a request-response action; puzzle shaped elements are local actions such as button pushes or navigation.

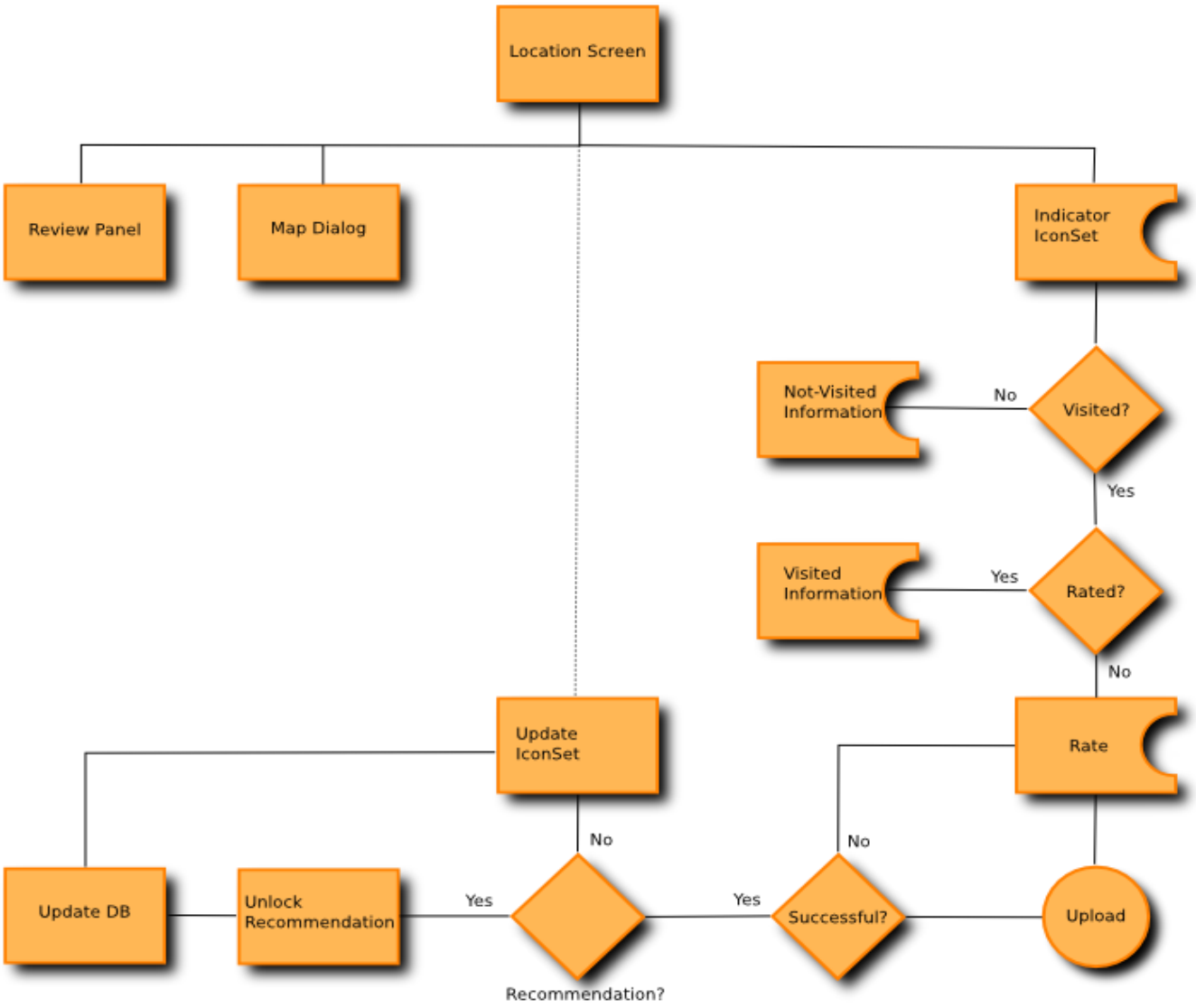


Figure 5 - Location, rate and upload

2.4.7 Upload

The upload process is triggered at the moment at which the user submits his valuation. If there is connectivity, this information is forwarded straight to the server; otherwise it is stored in a local table, waiting for a possibility to be sent. It is the mobile app's task to decide if the user applies for a recommendation, based on the number of rated locations, and to append an additional parameter to the upload 'rate' table entry. The response of the server ranges the location identifier of a new recommendation or an informative integer (Fig. 5).

2.4.8 Update

The information of this application is live, meaning it can change and grow dynamically if there is need to. Locations can be added, modified or disabled; scores change with every single rate submitted... it is useful for the user to be up to date in these aspects and also to have the latest reviews at his disposal. The update process is in charge of managing all these modifications and to synchronize the database on the server with those on the mobile terminals. It can be triggered at any moment by the user, being the only obvious condition an Internet connection (Fig. 6).

For this request, the petition data sent is the user identifier and the date and time of the last update made – extracted from the 'user' table in the local database. The server performs a check to detect what entries have been modified on a date and time posterior to the last flush to the user and sends all those entries back in the response. All the entries are properly updated in the local database at their arrival.

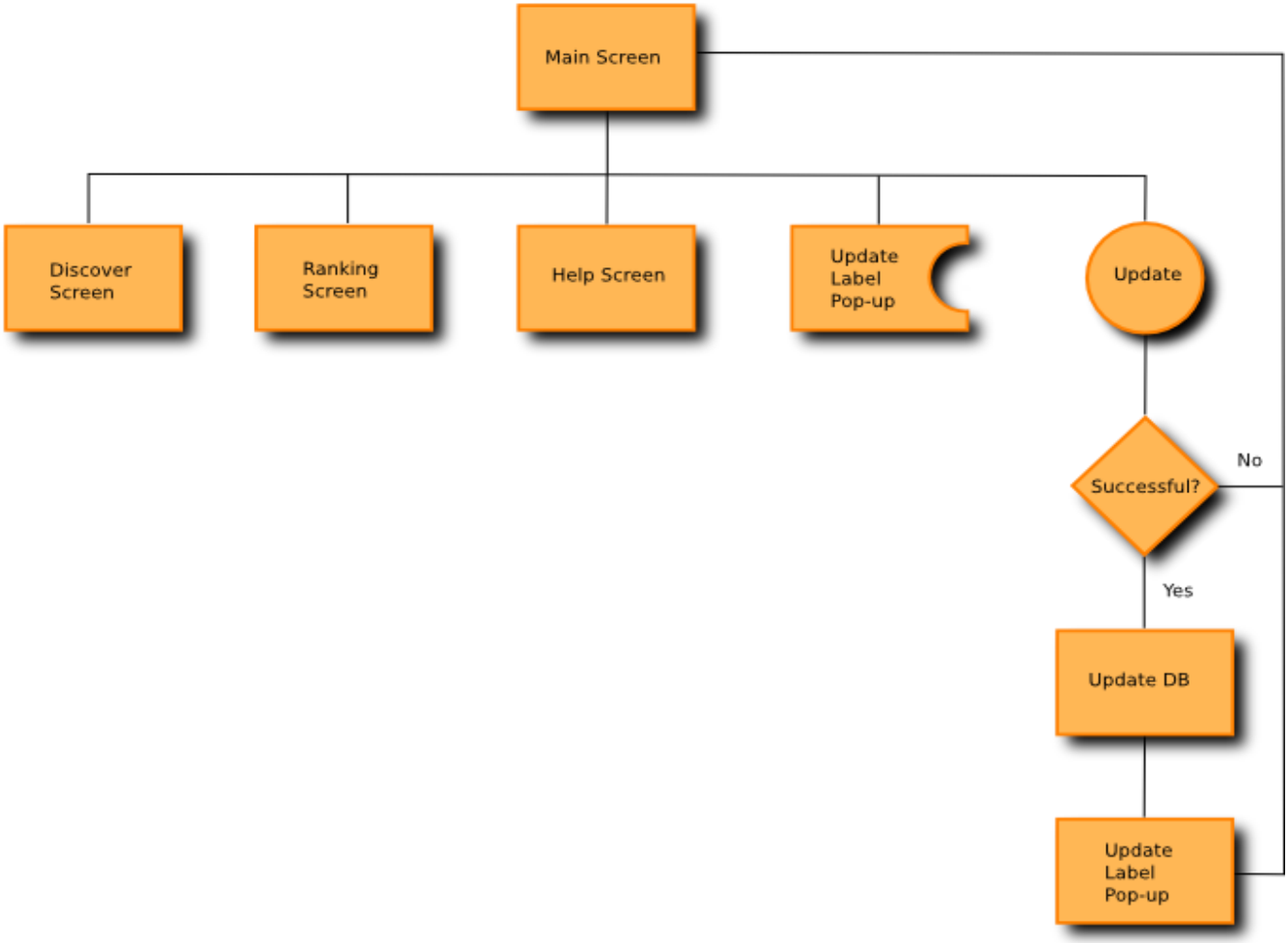


Figure 6 - Main and update

2.4.9 Additional features

Some of the additional props planned for the application are a help menu – at which the user can find useful information and understand how the progress works – a section with the top visited and top rated locations or a table with overall statistics about the usage and its population: number of clients, average number of visited places, most predominant profile, accuracy of the recommendations...

It has also been contemplated to add a price range for the locations which apply (such as pubs, restaurants, museums...) and a “how to reach” widget.

2.5 Architecture

The whole system is broken down by functionality for easiness, giving birth to what is here documented as modules. Modules are units of technologies, mechanisms and operations that are in charge of doing certain work, as for example the communication module is in charge of fulfilling the transmission needs between all the parts (Fig. 7).

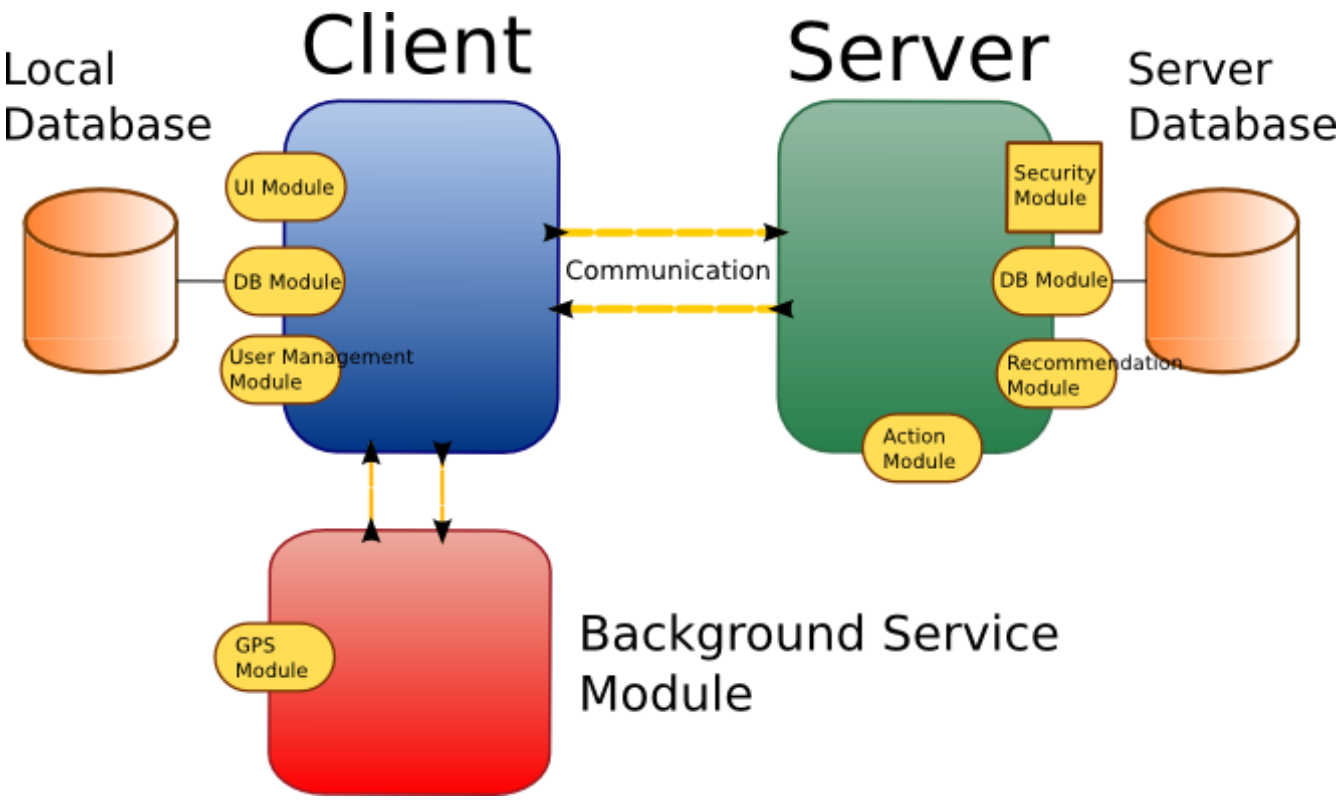


Figure 7- System architecture

2.5.1 Modules

Some of modules detailed here have a slightly different implementation depending on whether they are used in the client side or on the server. Nonetheless, the major principles of action remain the same in both cases.

- ***UI Module***

The UI Module is in charge of the visualization and user event interaction. Represented by the PhoneGap framework and the jQuery Mobile library.

- ***DB Module***

Set of instructions for database CRUD operations. In the client the implementation is natively handled by the web component, however html5sql library has been use to ease some situations. In the server it manages the DB through the Persistence API.

- ***UserManagement Module***

Controls the progress of the user, its configuration options, tokens and unlocked locations.

- ***Recommendation Module***

Its task is to calculate a valid recommendation outcome for the current user. Works against the DB module on the server side.

- ***Background Service Module***

Does all the operations when the phone's main application is dormant. In addition, it is in charge of executing native code, push notifications and interacting with the GPS module.

- ***GPS Module***

Reads and calculates the client's position and correlates the information with the DB parameters.

- ***Action Module***

Module dedicated to intercept and react upon client's actions such as user creation, update or upload. It interacts closely to other modules.

- ***Communication Module***

Manages the communication link and primitives between a client and the server.

- ***Security Module***

The set of security measures and mechanisms to prevent the most obvious threats to the system.

2.5.2 Conclusions

Next, an insight of the content for the next chapters is stated. The three major components of the diagram (Fig. 7) are broken down and detailed separately in order to acquire a better understanding of the system.

The modules listed above fit in the next chapters; some have a dedicated single section, others span across several and there are some that span across multiple chapters. The database module is the first to be described in chapter 3, but also has specific sections for configuring the module inside other chapters. The server and its components are described in chapter 4, as well as Security issues and the Recommendation module. The UI, GPS and the Background Service modules inside the client have their specific sections in chapter 5 of this document. Communication, Action and UserManagement modules are reviewed in chapters 4 and 5, with the former two being predominantly addressed in the server chapter and the latter being majorly addressed inside the client.

3 Database

The technologies involved in the creation of database on both parts of the system have just been reviewed and justified. This new chapter will focus in detailing the inner structure of the information for the application, and will do so by presenting a full diagram for both sub-systems and attaching the needed scripts to generate them. In addition, all the tables and their fields are explained at the end of each section.

3.1 MySQL

The implementation of the database on the server side has been done using MySQL, however its design should allow for an implementation on any other system. The reasons behind this choice have been a highly scalable and free system, good performance and availability, support of robust transactions and ease of management. On top of that, the author already had former experience with it in several other projects. The storage engine of choice has been InnoDB, given its transaction and foreign key support. Also on the plus side is its row locking features and capability to group multiple INSERT queries.

The server database design can be seen following (Fig. 8). It contains the bigger version of the structure as the server is a central point that manages the information flow towards and from clients.

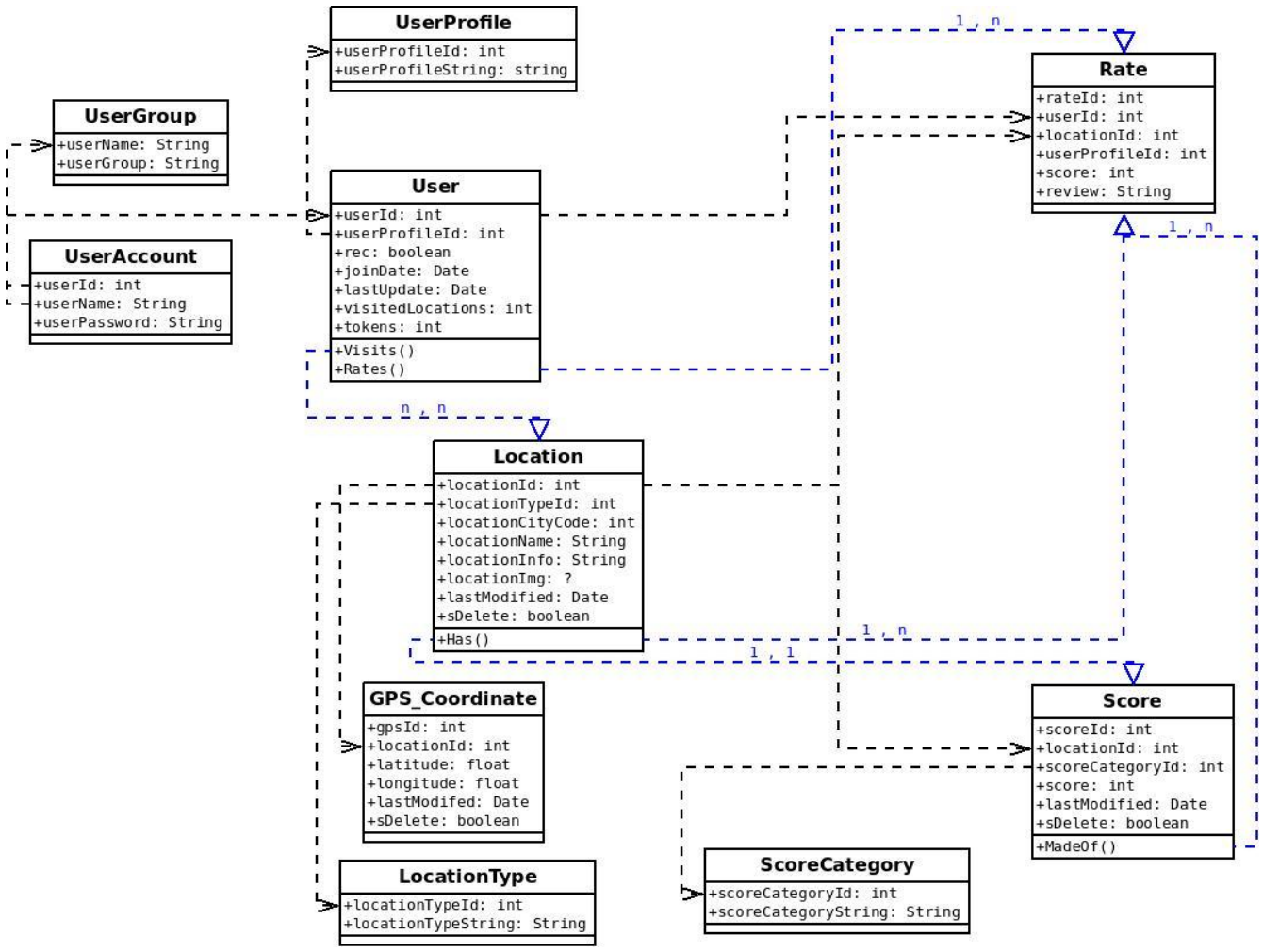


Figure 8 - Server DB

The server database script can be seen next (Code.Snippet 7).

```
DROP DATABASE IF EXISTS `liveguide`;
CREATE DATABASE IF NOT EXISTS `liveguide`;
USE `liveguide`;

DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
 `userId` INTEGER (10) NOT NULL AUTO_INCREMENT,
 `userProfileId` INTEGER (2) NOT NULL,
 `rec` BOOLEAN DEFAULT FALSE,
 `joinDate` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
 `lastUpdate` TIMESTAMP,
 `visitedLocations` INTEGER (5) DEFAULT 0,
 `tokens` INTEGER (3) DEFAULT 0,
 PRIMARY KEY (`userId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `userAccount`;
CREATE TABLE `userAccount` (
 `userId` INTEGER (10),
 `userName` VARCHAR (30) NOT NULL,
 `userPassword` VARCHAR (30),
 CONSTRAINT USER_PK PRIMARY KEY(userName)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `userGroup`;
CREATE TABLE `userGroup` (
 `userName` VARCHAR (30) NOT NULL,
 `userGroup` VARCHAR (30) NOT NULL,
 CONSTRAINT GROUP_PK PRIMARY KEY(userName, userGroup),
 CONSTRAINT USER_FK FOREIGN KEY(userName) REFERENCES
userAccount(userName)
 ON DELETE CASCADE ON UPDATE RESTRICT
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `userProfile`;
CREATE TABLE `userProfile` (
 `userProfileId` INTEGER (2) NOT NULL AUTO_INCREMENT,
 `userProfileString` VARCHAR (20),
 PRIMARY KEY (`userProfileId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `location`;
CREATE TABLE `location` (
 `locationId` INTEGER (10) NOT NULL AUTO_INCREMENT,
 `locationTypeId` INTEGER (2) NOT NULL,
 `locationCityCode` INTEGER (7) ZEROFILL NOT NULL,
 `locationName` VARCHAR (40) NOT NULL,
 `locationInfo` TEXT NOT NULL,
 `locationImg` LONGBLOB,
 `lastModified` TIMESTAMP,
 `sDelete` BOOLEAN DEFAULT FALSE,
 PRIMARY KEY (`locationId`)
) ENGINE=InnoDB;
```

```
DROP TABLE IF EXISTS `gps_coordinate`;
CREATE TABLE `gps_coordinate` (
 `gpsId` INTEGER (10) NOT NULL AUTO_INCREMENT,
 `locationId` INTEGER (10) NOT NULL,
 `latitude` FLOAT NOT NULL,
 `longitude` FLOAT NOT NULL,
 `lastModified` TIMESTAMP,
 `sDelete` BOOLEAN DEFAULT FALSE,
 PRIMARY KEY (`gpsId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `locationType`;
CREATE TABLE `locationType`(
 `locationTypeId` INTEGER (2) NOT NULL AUTO_INCREMENT,
 `locationTypeString` VARCHAR (20) NOT NULL,
 PRIMARY KEY (`locationTypeId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `rate`;
CREATE TABLE `rate` (
 `rateId` INTEGER (12) NOT NULL AUTO_INCREMENT,
 `userId` INTEGER (10) NOT NULL,
 `locationId` INTEGER (10) NOT NULL,
 `userProfileId` INTEGER (2) NOT NULL,
 `score` INTEGER (1) NOT NULL,
 `review` TEXT,
 PRIMARY KEY (`rateId`),
 UNIQUE KEY `ulx` (`userId`, `locationId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `score`;
CREATE TABLE `score` (
 `scoreId` INTEGER (10) NOT NULL AUTO_INCREMENT,
 `locationId` INTEGER (10) NOT NULL,
 `scoreCategoryId` INTEGER (2) NOT NULL,
 `score` DECIMAL (4,2) NOT NULL,
 `timesEval` INTEGER (10) NOT NULL,
 `lastModified` TIMESTAMP,
 `sDelete` BOOLEAN DEFAULT FALSE,
 PRIMARY KEY (`scoreId`)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS `scoreCategory`;
CREATE TABLE `scoreCategory` (
 `scoreCategoryId` INTEGER (2) NOT NULL AUTO_INCREMENT,
 `scoreCategoryString` VARCHAR (20) NOT NULL,
 PRIMARY KEY (`scoreCategoryId`)
) ENGINE=InnoDB;
```

Code.Snippet 7 - Server DB script

Next, an insight on the tables and their columns for the server part is included.

- User: This table stores the main element which allows a server to make distinctions between different users. Every entry is mapped to a single unique user inside the system.
    - userId: the integer identifier for each user entry and also the primary key of the table.

    - userProfileId: an integer value representing the type of profile chosen for this user element. It is mapped to the 'UserProfile' table.
    - rec: a boolean that indicates if the user has any pending recommendations to be claimed.
    - joinDate: this field holds a Date object, representing the instant at which the user created its account in the application and joined the system.
    - lastUpdated: another Date object is stored in this field. This time the value indicates the last time this user has executed a successful update command against the server, synchronizing its local database content.
    - visitedLocations: an integer value that counts the number of visited location by this user.
    - tokens: another integer value, containing the tokens for unlocking locations in possession of the user.
- UserProfile: The UserProfile table is a small static table containing all the available profiles from which a user can choose when creating its account.
    - userProfileId: this integer works as the primary key, also being the source of the foreign key of other tables.
    - userProfileString: contains a human-readable version of the profile associated to the above identifier.
- UserAccount: The account table contains the login and security information linked to entries of the User table.
    - userId: an identifier used both as primary key and as a nexus for the specific user from the User table for whom this information holds true.
    - userName: the username or nick a client has inputted when creating its account. It is used also for login purposes.
    - userPassword: the associated password for the above field. The pair userName:userPassword is the value:key used for authentication purposes.
- UserGroup: A simple table holding a mapping that indicates the role available for a user within the system. The primary key is a compound one, made by both fields.
    - userName: the same login information a user provides at account creation and the same value that is stored in the UserAccount table.
    - userGroup: a string field pointing the role the user with this account is allowed on the server.
- Location: The elements of this table represent the actual locations used in the application. A location is a single entity giving information and metadata about a specific place on the map.
    - locationId: an identifier used to distinguish and keep track of every different location; it is also the primary key of the table.
    - locationTypeId: an integer that is mapped in the LocationType table and gives information about the category where the place fits (Sightseeing, Gastronomical, Nightlife).
    - locationCityCode: a numerical field composed by the concatenation of the country's phone prefix and the local area phone code. Can be used to distinguish between

locations pertaining to different cities, and thus, different editions of the application.

  - locationName: the name of the location used when presented to the user on the server and on the mobile device.
  - locationInfo: textual information about each particular location.
  - locationImg: a BLOB field that can be used to store images and photos of the location. It can also point to an URL where the pictures are.
  - lastModified: stores the date of last modification for this entire entry. Useful for updating purpose.
  - sDelete: stands for 'soft Delete' and is a boolean that indicates if an entry of this table is active or not. Deactivated entries are treated as removed for a client and consequently deleted, so the device can save space while avoiding non-useful data. However, they are only marked in this way on the server, for statistical purposes and to be able to activate them later on, propagating the decision to the clients.

- GPS_Coordinate: A table for storing the positional coordinates for a location.
  - gpsId: primary key of this table, used to distinguish between entries.
  - locationId: references the field by the same name on the Location table. Used to relate coordinates to locations.
  - latitude: the latitude double value for this location.
  - longitude: the longitude double value for this location.
  - sDelete: another soft delete field, whose purpose is exactly the same as the one aforementioned.
- LocationType: A table with static information holding a mapping between the types a location can adopt and the integers used to include this information in other tables.
  - locationTypeId: the integer value that servers as primary key and to distinguish each category a location can pertain to.
  - locationTypeString: analogously to the 'userProfileString' field, this is the human-readable information defining a location type.
- Rate: The rate table contains rates: the single unit of transaction when a user submits a valoration for any place he has visited.
  - rateId: the primary key field, an integer identifying each rate entry as unique.
  - userId: the identifier of the user that has submitted this rate.
  - locationId: the identifier of the location this rate applies for.
  - userProfileId: the profile the rating user pertains to. Useful for also keeping track of a parallel scoring system segmented by profiles.
  - score: the integer value honoring its name; ranges from 0 to 10.
  - review: this text field contains the opinion the user has written for this location.
- Score: The score table entries keep track of the punctuation a location is given by a specific segment of the user base.

    - scoreId: primary key and integer identifier for each individual entry.
    - locationId: the identifier of the location this score applies to.
    - scoreCategoryId: the category of the score (explained inside the next table).
    - score: the proper mark of the location.
    - lastModified: another date field containing information about the last time the entry was modified.
    - sDelete: the soft delete field which allows to deactivate any entry.
- ScoreCategory: A table with static information related to the different categories under which a score can fall. Categories are just a way to refer to the 'Profiles' defined for user creation, plus a particular 'Global' category available only on this table. This allows the system to offer a global score (mix of scores provided by users with any profile), as well as keeping track of the current score issued by only a specific segment of the user base (anyone of the profiles).
    - scoreCategoryId: the primary key and identifier for the static information in this table.
    - scoreCategoryString: the correspondent human-readable version of the category.

3.2 SQLite

SQLite is the standard database implementation included out-of-the-shell in most devices with smartphone capabilities or tablets, this being the principal reason for its usage in this project. It is fast and lightweight, and offers excellent compatibility with the structure and language keywords used in other implementations.

The client database design and script can be seen next (Fig. 9). As already noted, this structure is smaller, some of the tables got only a single entry and there is a client specific table for keeping track of the progress, “ProgressTrack”.

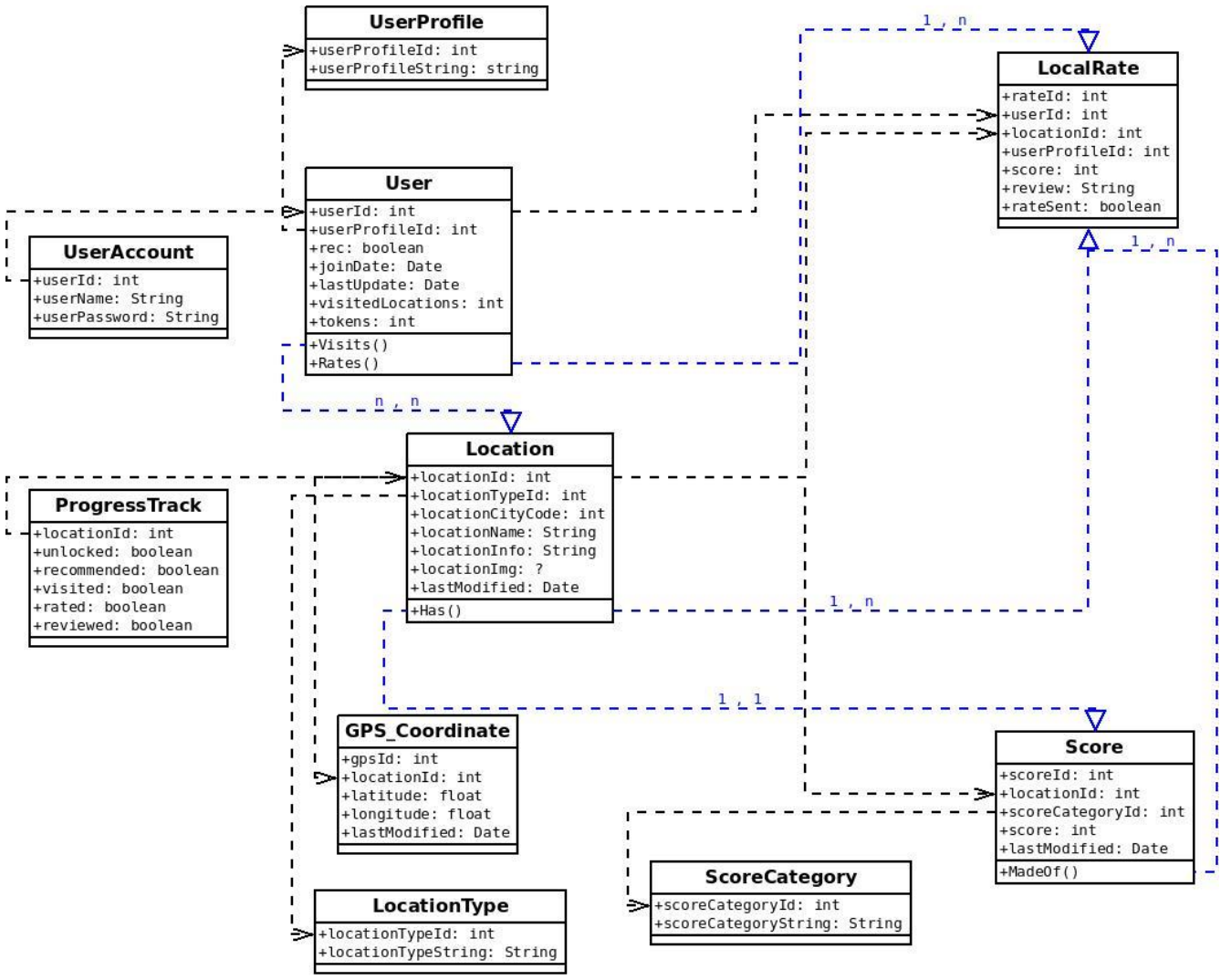


Figure 9 - Client DB

The client database script can be seen next (Code.Snippet 8).

```
CREATE TABLE "gps_coordinate" (
  "gpsId" int(10) NOT NULL,
  "locationId" int(10) NOT NULL,
  "latitude" float NOT NULL,
  "longitude" float NOT NULL,
  "lastModified" timestamp NOT NULL,
  PRIMARY KEY ("gpsId")
);
CREATE TABLE "location" (
  "locationId" int(10) NOT NULL,
  "locationTypeId" int(2) NOT NULL,
  "locationCityCode" int(7) NOT NULL,
  "locationName" varchar(40) NOT NULL,
  "locationInfo" text NOT NULL,
  "locationImg" longblob,
  "lastModified" timestamp NOT NULL,
  PRIMARY KEY ("locationId")
);
CREATE TABLE "locationType" (
  "locationTypeId" int(2) NOT NULL,
  "locationTypeString" varchar(20) NOT NULL,
  PRIMARY KEY ("locationTypeId")
);
CREATE TABLE "progressTrack" (
  "locationId" int(10) NOT NULL,
  "unlocked" tinyint(1) DEFAULT '0',
  "recommended" tinyint(1) DEFAULT '0',
  "visited" tinyint(1) DEFAULT '0',
  "rated" tinyint(1) DEFAULT '0',
  "reviewed" tinyint(1) DEFAULT '0',
  PRIMARY KEY ("locationId")
);
CREATE TABLE "rate" (
  "rateId" int(12) NOT NULL,
  "userId" int(10) NOT NULL,
  "locationId" int(10) NOT NULL,
  "userProfileId" int(2) NOT NULL,
  "score" int(1) NOT NULL,
  "review" text,
  PRIMARY KEY ("rateId")
);
CREATE TABLE "score" (
  "scoreId" int(10) NOT NULL,
  "locationId" int(10) NOT NULL,
  "scoreCategoryId" int(2) NOT NULL,
  "score" decimal(4,2) NOT NULL,
  "timesEval" int(10) NOT NULL,
  "lastModified" timestamp NOT NULL,
  PRIMARY KEY ("scoreId")
);
CREATE TABLE "scoreCategory" (
  "scoreCategoryId" int(2) NOT NULL,
  "scoreCategoryString" varchar(20) NOT NULL,
  PRIMARY KEY ("scoreCategoryId")
);
```

```
CREATE TABLE "user" (
  "userId" int(10) NOT NULL,
  "userProfileId" int(2) NOT NULL,
  "rec" tinyint(1) DEFAULT '0',
  "joinDate" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "lastUpdate" timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
  "visitedLocations" int(5) DEFAULT '0',
  "tokens" int(3) DEFAULT '0',
  PRIMARY KEY ("userId")
);
CREATE TABLE "userAccount" (
 "userId" int(10) NOT NULL,
 "userName" varchar(30) NOT NULL,
 "userPassword" varchar(30) NOT NULL,
 PRIMARY KEY ("userName")
);
CREATE TABLE "userProfile" (
  "userProfileId" int(2) NOT NULL,
  "userProfileString" varchar(20) DEFAULT NULL,
  PRIMARY KEY ("userProfileId")
);
CREATE INDEX "rate_ulx" ON "rate" ("userId","locationId");
```

Code.Snippet 8 - Client DB script

An insight on the tables and their columns for the client part would serve no purpose, as all the tables and fields present in both have the same purpose. As noted, the 'sDelete' fields are not present in the tables on the client, as on mobile devices there is no distinction between active or not; the entries are either deleted or inserted. One single table is worth mentioning, since it is present only on the client:

- ProgressTrack: This table keeps record of the progress a user has made within the application, specifically what milestones has reached and what marks are active for a certain location.
    - locationId: the identifier and primary key that indicates the location the next fields are valid for.
    - unlocked: indicates whether the location has or has not been unlocked inside the app, and thus, available for visiting.
    - recommended: indicates whether this location has been recommended, and thus, available for visiting. Also used for distinguishing recommended locations from unlocked ones.
    - visited: another boolean. This one notes if a location has already been visited or not.
    - rated: if a location has been visited, this boolean could be true or false, depending on whether the user has already rated it or not.
    - review: for this field, the same conditions as for the above one apply. The field type, however, is a text one.

This concludes the database chapter; next, the server is introduced.

4 Server

This chapter is a breakdown of the application server, one of the two main components of the system. The server is the direct responsible for interacting with the database, managing the connections to the resources and acting as a central point for client's requests, generating responses and delivering content when needed. Since the functions of the server are broad and there are multiple parts of it that are worth describing individually, the chapter is divided in sections.

First, the intrinsic role of the server is noted, explaining the different types of servers, how a server works and the protocols involved in the process. After that, the Model-View-Controller (MVC) development paradigm is explained and shown within the overall structure. The model is the first reviewed element, making special emphasis in its setup and configuration requirements. A bigger picture is then depicted in the next section, where the whole interaction process is detailed, noting the controlling elements in each phase and the points a full request-response cycle traverses. Going one level deeper in this same section, a description of the major request-response actions is provided, taking into account the expected parameters at each end and what is specifically done with them. In addition, the JSON notation structure between interactions is included. Two more sections close this chapter: the first of them deals with the recommendation system, responsible for correlating scoring data between users and proposing affine locations to them; the other section explains the security features bundled with the server and the necessary configuration in order to achieve them.

4.1 Server structure

A server is a piece of software that runs on a computer, whose task is to accept and process requests of other pieces of software, namely the clients, and eventually craft a response to send back to them. For this to be possible the server has to be running on a unique Uniform Resource Locator (URL) and, obviously, this URL has to be accessible from any other point of Internet. There are different kinds of servers: application servers, mail servers, web servers... some of them being able to run stand alone, while others needing to do so in a particular container. As noted in former chapters, in this project's scenario Glassfish application server is being used; this server is able to hold both application and web content. It is difficult to strictly label this project, so let's just state that an application server is used to host what is a web mostly web content with some enhancing like the management of database pooling and transactions and some application logic within the software.

LiveGuide’s server architecture is based off of a Servlet, a piece of action-processing code acting as a controller for the backend logic. In order to process any request of a client, the transmission of this information goes through several of the layers of the OSI model during the data unit exchange; it is important to notice that the programmatic management of these transmissions start by the acceptance of the connection by the server, specifically driven by the TCP/IP connection protocol (Fig. 10). Requests can be made to the server via both HTTP GET and POST messages, albeit at this point all the requests

are using the former, simply concatenating the needed parameters to the URL.

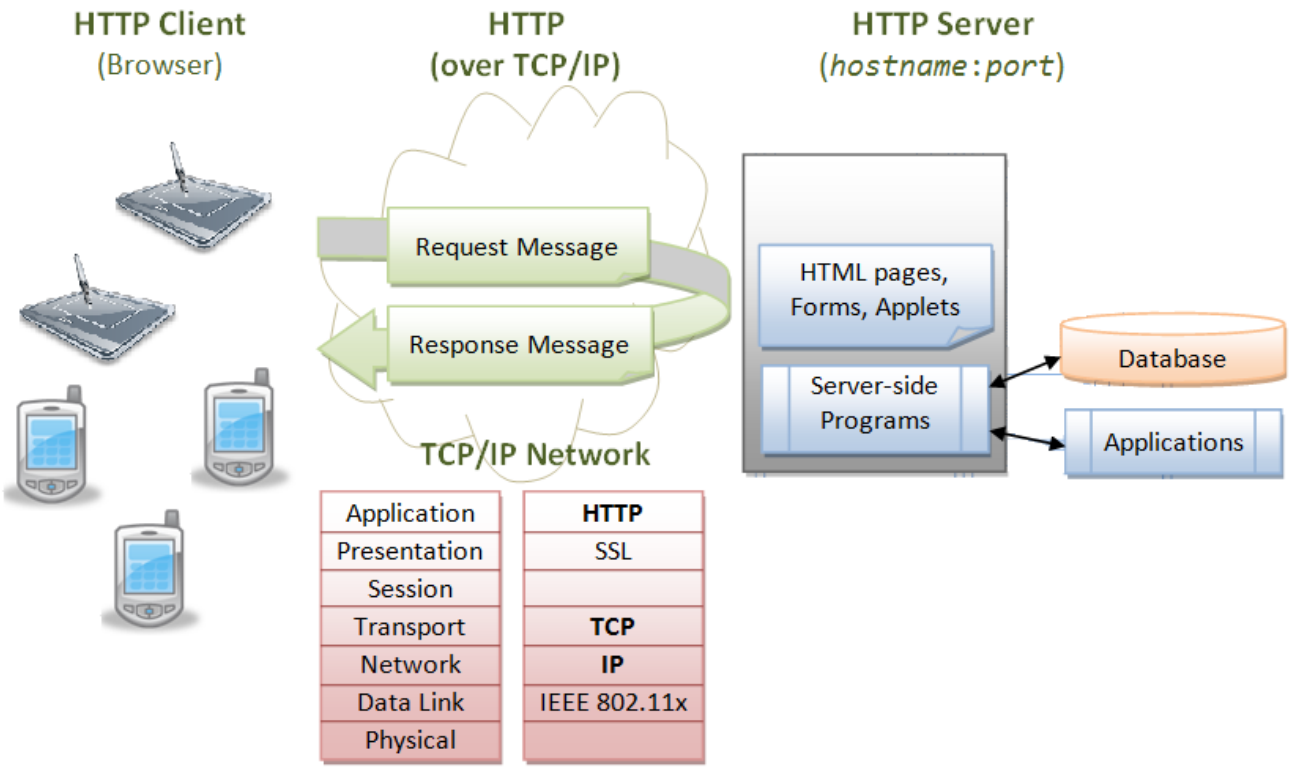


Figure 10 - Client-Server

4.2 Server MVC

For quite some time now there has been a generally accepted path to develop multiple facet software, a vision providing a documented modality for achieving particular development challenges; this is what is called a development paradigm. This project has been built taking into consideration the Model-View-Controller concepts and structured in three clearly defined categories (Fig. 11).

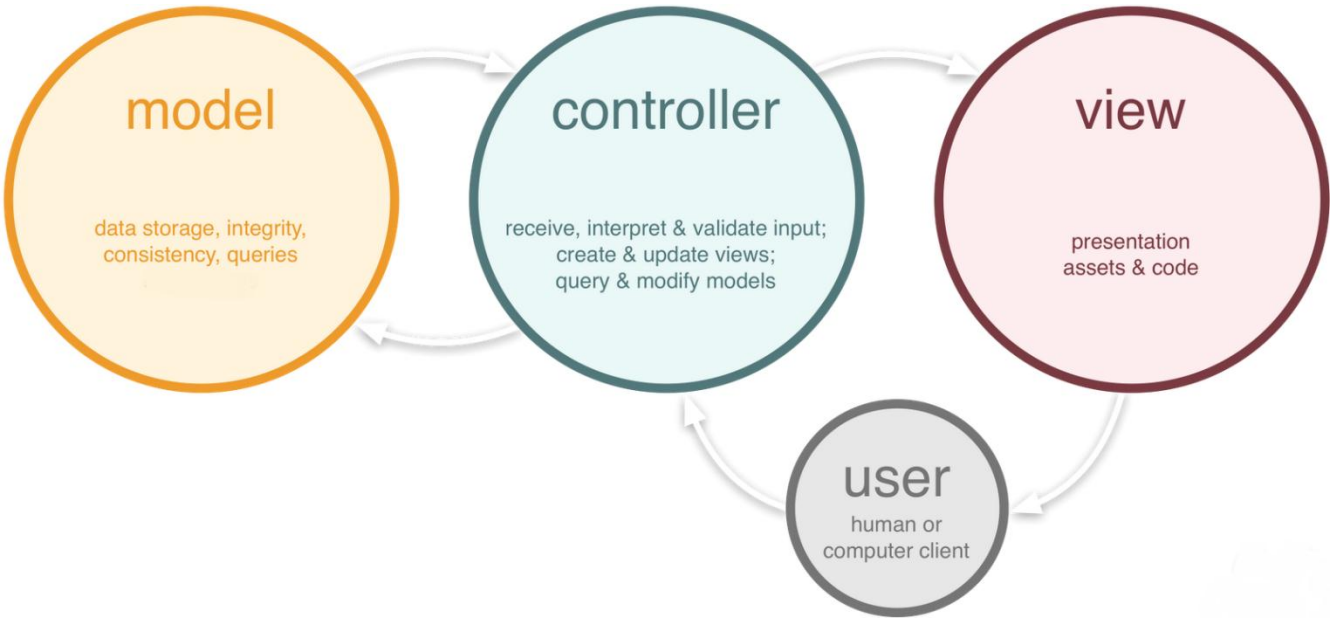


Figure 11 - MVC

- Model
  Some applications manage a considerable amount of data, be it created and manipulated at real time or just accessed in store alike containers. When this is the case, it can be decided to use a database to store the accessed information; this represents the model of the application. A database is a passive element, it contains information which is structured and classified throughout tables and that can be queried and modified at any moment through the controller element. In LiveGuide, as should happen with any other project implementing the model from this paradigm, this element can be safely relocated and its implementation changed, without it affecting the rest of the project's elements.
  LiveGuide uses MySQL as the database implementation, manages it through a pool of connections that offer the end resources and uses EclipseLink as a persistence provider (compliant to the JPA 2.0).

- View
  The view is in charge of the appearance of the data, which specifically includes the responses. How the information is presented to the user has a great impact on the perceived quality of the application; a responsive user interface, intuitive commands, a fancy design, solid frame... all intervene to achieve this goal. It is for this reasons that in most commercially available applications, the front-end design is assigned to a graphic designer. Being this a one-man project and not being the front-end a indispensable part of the project, there is nothing fancy about the end result. Albeit there is a web front-end on which a dump of the database can be seen and several ideas for appealing options of rankings, statistics and top locations to be placed on it, right now the only true view element is the Servlet response within a JSON structure.

- Controller
  The controller is probably the most important element of the whole system and is represented by the Servlet itself. It is in this core where all the possible actions are mapped, the tasks defined and the flow of the application controlled. The controller is in charge of starting and initializing every component, assuring that the database resources are available and reachable through the connections in the pool, verifying the binding to the URL and creating the context needed for the client requests to be correctly routed towards the wanted action. There are two

possible ways for interacting with the Servlet: a GET and a POST method. The difference between them is the location of the parameters, with the GET method they are concatenated to the requesting URL by the means of ampersand and equal signs, serially ordered after a question mark at the end of the mapped URL [?var1name=value1&var2name=value2...]; in a POST method they are located in the message's body. Traditionally, the POST method is used for submitting data to a server (for example data that comes from a form) and the GET method is used for querying or requesting information. Nevertheless, as mentioned before, GET method can also include parameters at the end of the URI as long as the final length of the URI is not above 255 bytes. This is the reason that, at this point, client's requests are made using GET methods; however it should be eventually changed for compliance.

Next, these components are described in detail.

4.3 Model

One of the elements with which the server has to communicate is the database on its side. There are several ways to do this: some involve a manual configuration of the set up and the communication, others benefit from wizards and tools for process automation while there could also be a heterogeneous solution. The resulting bridge is used for querying and managing the data stored in the database, as requested by the different actions executed by users.

4.3.1 Database set up

Once the schema and its tables are defined, a connection pool can be created with the admin console command ('asadmin') or from the Glassfish server's administrator console (localhost, port 4848). This console is just a graphic tool with the same capabilities than the server's admin console command: it allows configuring the server's options, scripts, security, domains, realms and several other things. In order to do so, a new pool has to be made inside 'Resources → JDBC → Connection Pools' with the following field contents:

- Name: liveguideserver
  The name of the new connection pool, matters only for internal purposes as when configuring the resource later.
- Resource type: javax.sql.DataSource
  The type of resource that will be managed through this connection pool.
- Database vendor: JavaDB
  Indicates the needed driver for the database vendor. The DataSource class name is automatically selected after this.
- Database name: liveguideserver
  The name of the schema that will be managed through this connection pool. Same as the name used to create the MySQL schemata.
- User: liveguideclient
  One of the users that has enough privileges to do the operations that will be requested from him.
- Password: ******
  The password for the specified user.
- ServerName: localhost
  Where the database is hosted. In this demonstrator's scenario, the database is hosted in the same machine as the server program.

After created, the database can be 'pinged' through this admin console to check whether the communication works and the information inside is consistent.

The second thing that has to be done is the creation of the resource. Again, in the Galssfish admin panel, create it within 'Resources → JDBC → JDBC resources'. Those are the used values for this project:

- JNDI Name: liveguide
  The Java Naming and Directory Interface (JNDI) allows the discovery and look up of data and objects via a name. It is going to be used later for the authentication system explained at section 4.6.3 – Authentication.
- Pool Name: liveguideserver
  The name of the pool that has been created in the last step.

With this set up and an existing database with the indicated names, it is possible to connect to it through the wizard in the used IDE (Netbeans in this case). The result is, besides the established communication, the creation of the glassfish-resources.xml' configuration file holding the following information (Code.Snippet 9):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 Resource Definitions//EN"
"http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
<resources>
    <jdbc-connection-pool allow-non-component-callers="false"
associate-with-thread="false" connection-creation-retry-attempts="0"
connection-creation-retry-interval-in-seconds="10" connection-leak-
reclaim="false" connection-leak-timeout-in-seconds="0" connection-
validation-method="auto-commit" datasource-
classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource" fail-all-
connections="false" idle-timeout-in-seconds="300" is-connection-
validation-required="false" is-isolation-level-guaranteed="true" lazy-
connection-association="false" lazy-connection-enlistment="false"
match-connections="false" max-connection-usage-count="0" max-pool-
size="32" max-wait-time-in-millis="60000"
name="mysql_liveguide_liveguidePool" non-transactional-
connections="false" pool-resize-quantity="2" res-
type="javax.sql.DataSource" statement-timeout-in-seconds="-1" steady-
pool-size="8" validate-atmost-once-period-in-seconds="0" wrap-jdbc-
objects="false">
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="3306"/>
        <property name="databaseName" value="liveguide"/>
        <property name="User" value="liveguide"/>
        <property name="Password" value="tIfc2_2M6-$fU|PNO&lt;?8"/>
        <property name="URL"
value="jdbc:mysql://localhost:3306/liveguide?zeroDateTimeBehavior=conv
ertToNull"/>
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="liveguide" object-
type="user" pool-name="mysql_liveguide_liveguidePool"/>
</resources>
```

Code.Snippet 99- Persistence information

4.3.2 Object-Relational Mapping

In order to use and manage a database from within an object-oriented programming language, a technique for transforming this data between incompatible system types is needed. This technique is called Object-Relational Mapping (ORM) and, in essence, it creates a virtual representation of the original database which can be used from the programming language.

To do so, there are several alternatives, depending on the languages used in the program. Since the server has been developed using Java (Oracle's version) and the Enterprise Edition already ships with it, TopLink[21] has been the used framework. In turn, TopLink is an implementation of the Java Persistance API (JPA)[22]; a framework for management of relational data, a work resulting from the JSR 317 specification. Together, they offer some interesting tools for coding.

As can be seen in the next figure (Fig. 12), the elements from the structured database can be represented by Entities within the programming language after applying the IDE's ORM wizard. These entities are managed by the EntityManager, which in turn is used to interact with the database itself through the JPAController classes.

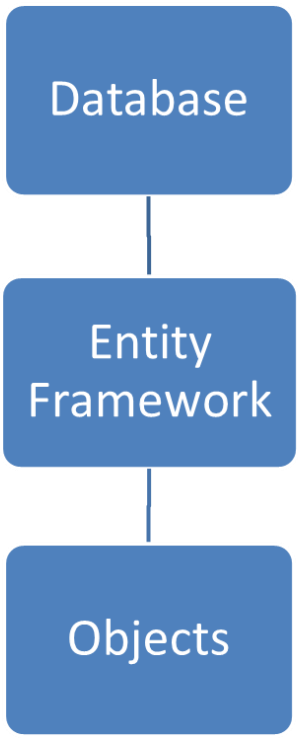


Figure 12- ORM Diagram

- ***Entity***

An entity is a lightweight persistence domain object. Each entity class references a table from the database, including all its columns, properties and characteristics, while instances of it represent a single entry in that table. Among the useful tools in the NetBeans IDE, there is a wizard for automatically creating the entity classes out of a connection to the target database.

- ***EntityManager***

Entities are managed through an EntityManager. Each EntityManager is associated with a persistence context; a persistence context defines the scope under which particular entity instances are created,

persisted and removed. This class contains the needed methods to interact with the specified table.

- ***JPAController***

These are helper classes that contain methods for creating, editing and deleting a single entry from the specified table, wrapped accordingly inside transactions (more about transactions in this same chapter) and using an EntityManager. In addition, every other method which interacts with entities and, in turn, with the database, should be build inside this controller class, following the same transaction guidelines than the packed methods.

- ***Persistence and Transactions***

The Persistence Context indicates the span of existence of a set of entities in a data store; i.e. the scope of the entities while preserving their persistence features. A transaction is a set of operations that either fail or succeed as a unit, consisting on a single (or a set) of queries that manipulate the data in the tables of the database. When used in JEE JPA provides integration with JTA (Java Transaction API).

- ***EntityManagerFactory and UserTransaction***

In the project's scenario a constant flow of communicating clients is expected, which means that special attention had to be put around concurrency. For JPA to be thread-consistent, it cannot share the same EntityManager instance across all threads; application-managed entity managers are needed, since their persistence context is not propagated to application components, enforcing the life cycle of each EntityManager instance to be managed by the application. In order to create them, an EntityManagerFactory has to be used; EntityManagers are created and destroyed for every transaction inside the controller class.

The EntityManagerFactory instance has to be first obtained by injecting it into the application component, in the following way:

```
@PersistenceUnit
EntityManagerFactory emf;
```

In addition, since the scenario is a concurrent one and application-managed entity managers have been used, the transactions have to be carried out manually. That is to say, specifically indicating the points at which transactions begin, commit or rollback in case of an unexpected event.

The UserTransaction instance has to be injected into the application and can be shared among different threads, since it locks on the needed resources while doing operations.

```
@UserTransaction
UserTransaction utx;
```

4.4 Controller

LiveGuide's server is basically a Servlet with a mapping to an URL for every possible request in the web.xml configuration file. The chosen technology over which the Servlet is deployed has been Java Enterprise Edition. A Servlet is nothing but a class that helps extend the capabilities of a server, can handle requests and respond to them and provide content dynamically with or without processing. The Servlet lifecycle can be seen at (Fig. 13). Servlets have more features that will not be needed in our scenario as the approach of the request-response patterns in this project does not need the maintenance of a session, the reason of which Enterprise Java Beans (EJB) have not been used. On one side, the

Servlet interacts with the clients in what is known as the front-end. Inside its core starts the back-end with what is known as the application logic, closely connected through an application managed connection of pools to the resources mapped in the database, carrying out all transactions through this channel. The configuration and parameters for this deploy are as follow (Code.Snippet 10).It would be useful to initialize the database connection pool and resources before the client's requests start to arrive. What is needed is a context listener to hijack the Servlet initialization event and run some code to prepare a fully functional and responsible environment for the clients (Code.Snippet 11).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>ControllerServlet</servlet-name>
        <servlet-class>web.ControllerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ControllerServlet</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>0</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Code.Snippet 10 - Servlet mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <listener>
        <description>ServletContextListener,
HttpSessionListener</description>
        <listener-class>util.ContextListener</listener-class>
    </listener>
    <servlet>
        ...
</web-app>
```

Code.Snippet 11- ContextListener config

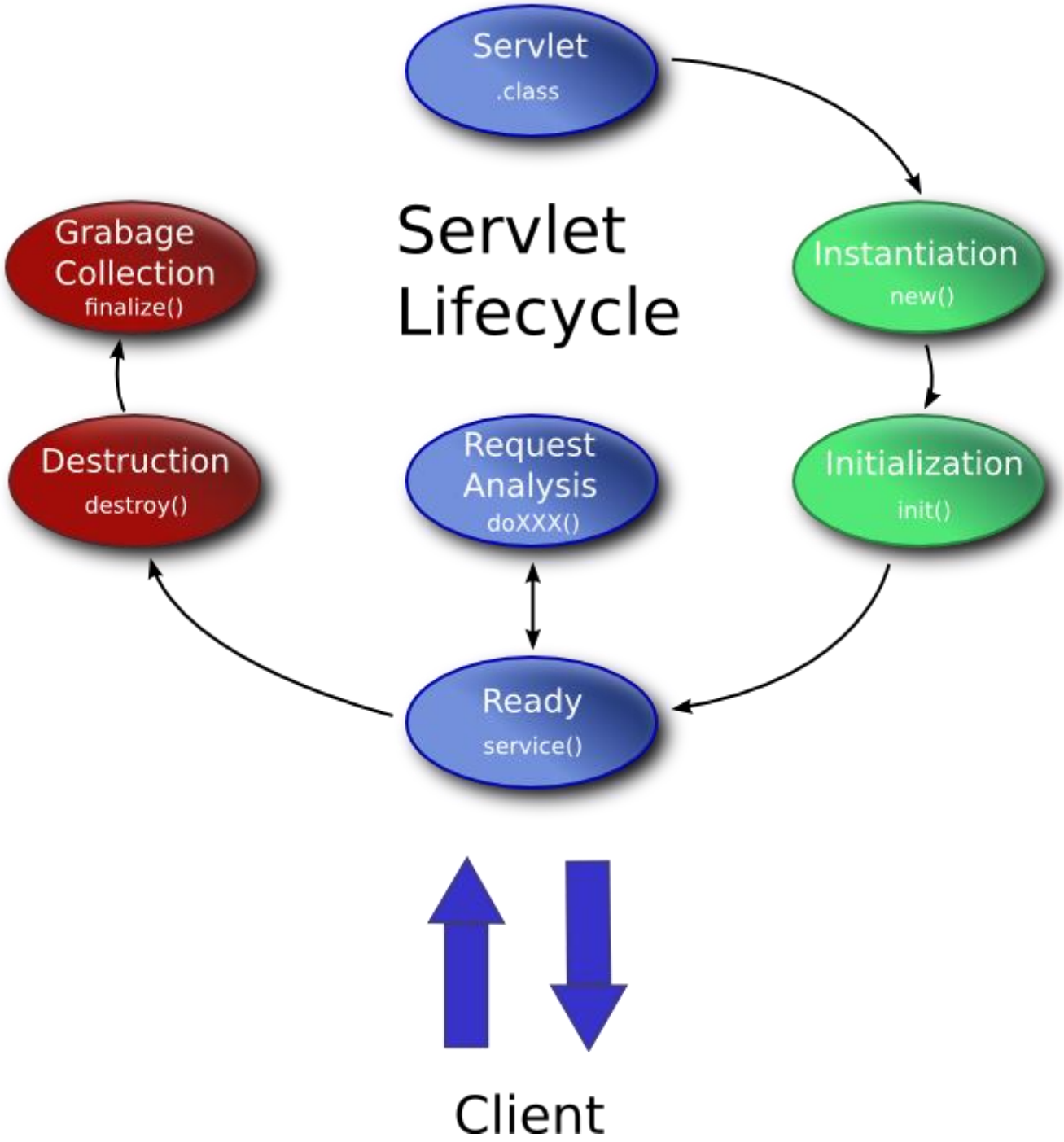


Figure 13- Servlet lifecycle

The whole server is a complex structure for which, at every client's request, intervene the different elements at specifically designed steps. In general lines, a client needs to do some action and in order to do so it has to interact with the server. The server has a set of discrete actions which it allows and is capable of processing and responding to, and has them mapped as URI in the configuration file. The client then proceeds to access the needed URI, attaching any compulsory parameters at its end. On turn, the server guides this request to the appropriate class processing the action and concludes by crafting the response. Usually, this response is packed with some results from the database and a view manager

presents them in a specific page and format, according to the results, as can be seen in the figure (Fig. 14).

Nonetheless, in LiveGuide the presentation occurs at the other end, on the client's terminal. It can be said that the view element is technically on the mobile device and that the decisions about how to present the information is there; a compatible approach to the matter is to say that technically part of the view is the JSON response message, however it is not shown on the server's front-end.

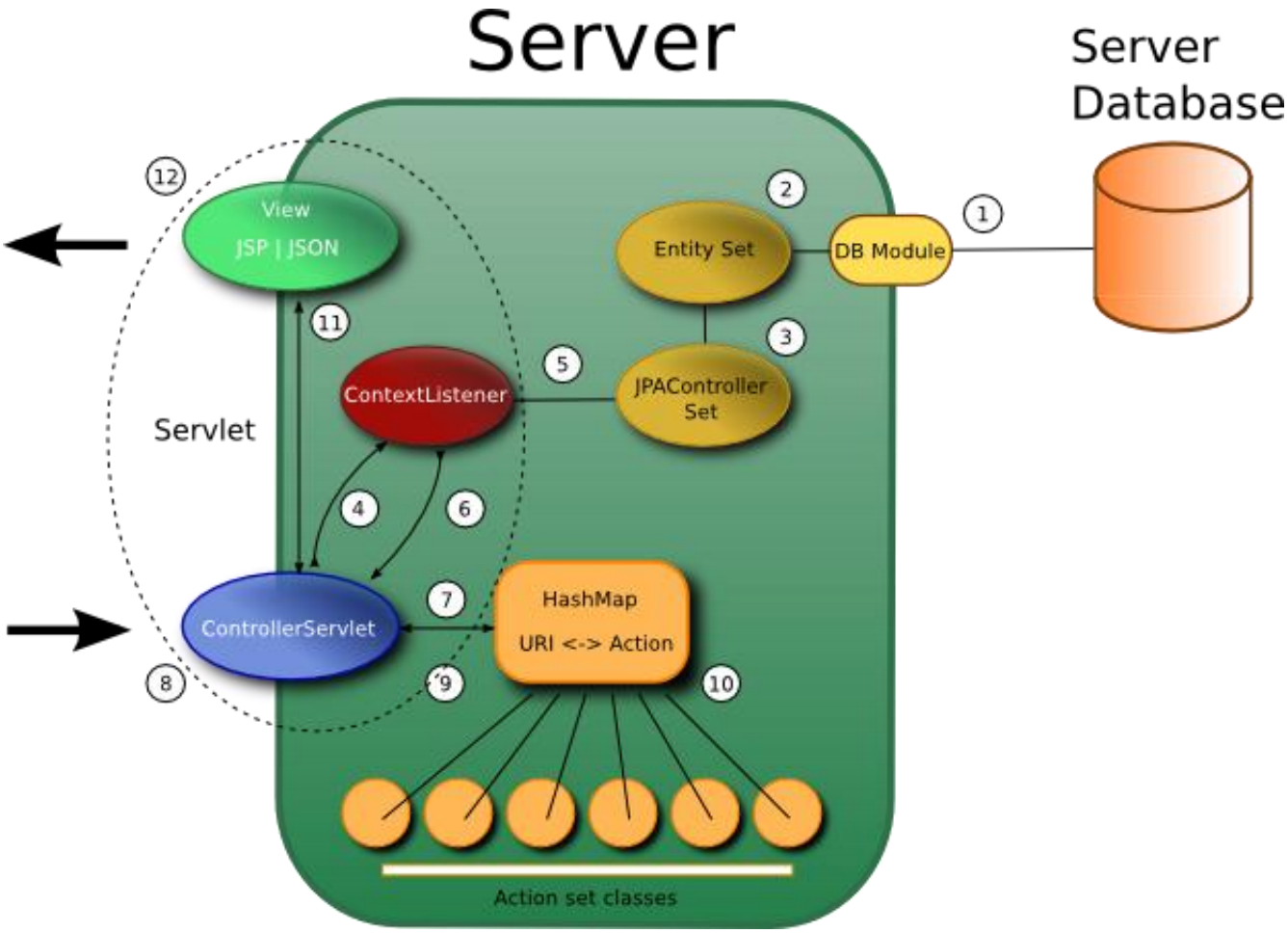


Figure 14 - Server interaction diagram

A closer look to the code in this project shows the following (Fig. 14):

1) A connection to the database resource is created with the parameters in the configuration files.

2) Entity classes are created to mirror the model at the server.

3) Controller classes are created from each entity. These classes control the transactions with the original database through the use of the UserTransaction resource and the EntityManagers obtained through the EntityManagerFactory persistence unit.

4) When the ControllerServlet attempts to init, the process is hijacked by the ServletContextListener which initializes one controller for every entity kind and bundles them in the Servlet's context, allowing a shared use between multiple clients (threads).

5) After that, the ContextListener releases the lock and allows the Servlet to init. This process consists in the creation of a HashMap where all the possible actions are stored. Every action's key is the end of the URI to which it is mapped (for example '/init.do'), while the value is a class derived from the

original abstract Action class, with the needed controllers extracted from the servlet context as arguments.

6) When a petition arrives, the controller servlet (in both its method GET and POST) extracts the requested action and calls it's perform method, passing along any parameters that were sent. If a non-existent page is requested, the ViewManager creates a JavaServerPage (JSP) with an error message.

7) The perform function of an Action class deals with the logic and processing of the client's action. Creating a new user, requesting a database update or uploading and introducing entries in the database are all done at this point.

8) Finally two outcomes are possible. Either the client has requested a valid action and there have been no problems during its processing, in which case a JSON response is returned; or the requested action was one of the available on the main page (such as a database dump, statistics..). Then, the information is presented in a JSP. The JSP (as well as the error pages) are supposed to only be accessed from a web browser and are no supported in the mobile terminals!

4.4.1 Actions

This segment of the report describes the different actions that are mapped in the servlet, noting when those are available from the mobile terminals and when they should only be used on site.

- Init [.../init.do] (on site)
  Not really an action, is the default page which is generated when accessing to the Servlet, without specifying any path. It serves as a check and, during the carried tests, it also served to populate the 'rate' and 'score' tables with some randomly generated data. Also, this initial page serves as point of contact with the project – some information about the system, how it works and links with statistics, highly rated locations and other features are planned to be placed here.
- Create User [.../createUser.do] (available on mobile client)
  When any application client wants to start using all the features of the project, the first thing he must do is to create an account. The compulsory parameters in order to do it are:
  - Username: the username the client wants to register with – it must be unique.
  - Password: the associated password for the chosen username – uniqueness is not compulsory.
  - LastUpdate: the last time a user has updated the information in his local database or, what is the same at the moment of user creation, the date of the most recent entry in his database. It could be the one which came bundled with the download or the one of the last manual update the user did, if he ever decides to create another user.
  - Profile: the profile identifier chosen by the client.

When this request starts to be processed (Fig. 15), the first thing done is to create a 'UserAccount' entity and perform a check and update over the pertinent table. If the chosen user name is unique, the pair gets inserted as an entry; doing both operations within the same controller method guarantees atomicity and avoids any possibility of interference between the two steps. Next, the 'lastUpdate' parameter is pulled and the date is parsed (it can adopt different formats depending on some outcomes). The 'profile' parameter is also pulled and a new 'User' entity is created and introduced in the database. The 'userId' of this entity is retrieved and used to update the 'UserAccount' entity which was initially created without this field. Finally, the convenient information is also introduced in the database in the form of a 'UserGroup' entity, formed by a unique key-value pair containing the user name and the group at which it pertains (hardcoded as the 'USER' for the clients using this action). After that, the database is queried

for a descendant ordering of all the locations based by their score – only taking into account the scores submitted by users who match the profile of the client creating the account. That is to say, every user will have different locations available at start and the newly unlocked ones will also follow this pattern, essentially allowing for a unique experience bound by the profile of the client. To conclude, all this information is parsed to JSON format, namely the whole 'User' entity and three arrays containing the ordered locations (one for every location category).

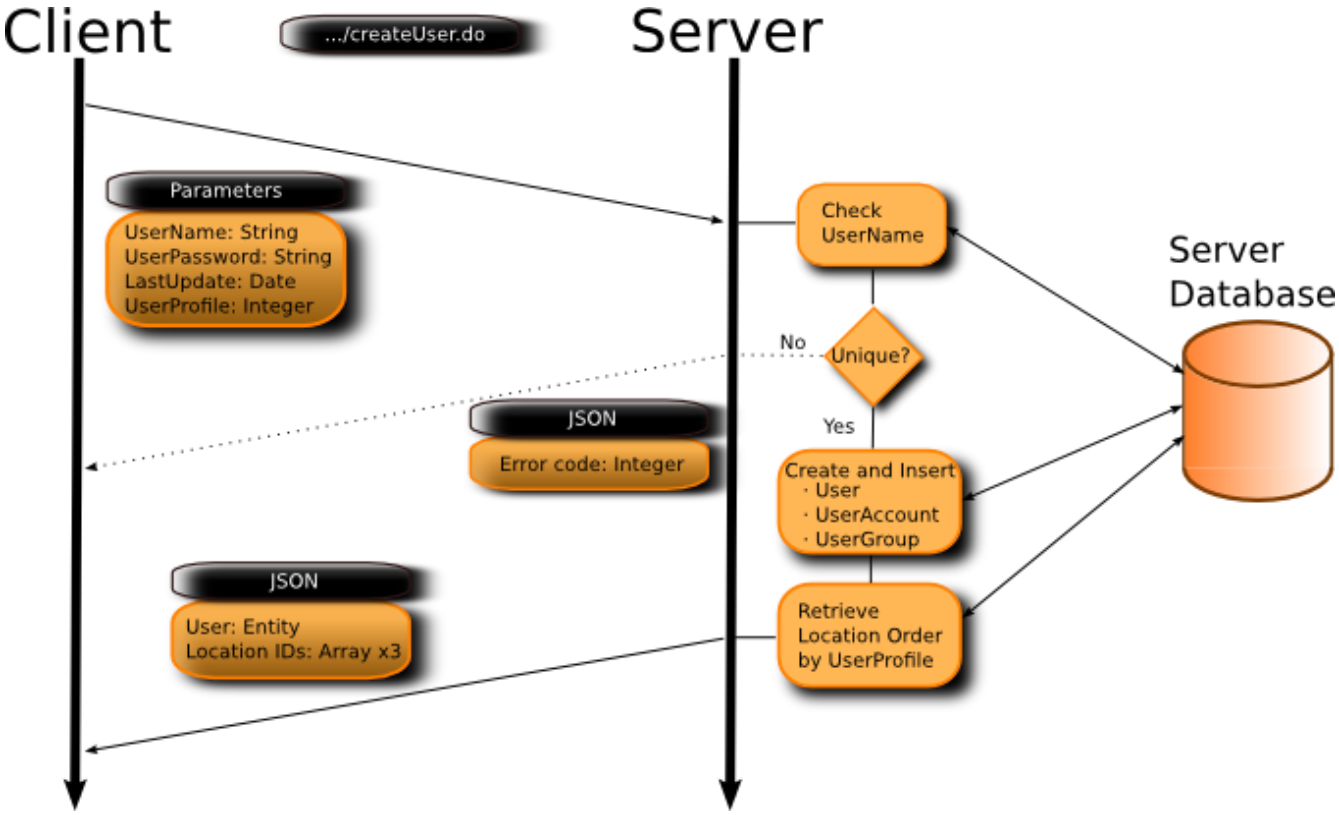


Figure 15 - CreateUser logic

- Update [.../update.do] (available on mobile client)
  Whenever any client wants to synchronize its local database with that on the server, he attempts to call this action through a request. The needed parameters are:
  - UserId: the identifier of the user activating this action.
  - LastUpdate: same parameters as described in the last action.
  - Visited: number of visited location this user has in his counter. It is used for statistical purpose only on the server side (yet to be implemented).
  - Tokens: the number of tokens in user possession at the moment of triggering the action. This is again only to keep track of usage, not intended as an anti-cheat system.

The first thing done when performing the action is the retrieval of the last update parameter. The date is then used to query the 'Location', 'Score' and 'GPS_Coordinate' tables of the database, returning every entry with a field 'lastModified' placed in time after the date submitted by the user. Those are the entries that need to be synchronized. The resulting arrays are then parsed to a JSON format and appended to a manually crafted header for the format. This header contains a keyword and a numerical value for each of the arrays; it is used to indicate the length of each of them to the client when parsing back, as there is no support to do that on the fly when different object arrays have been merged on the same string (Fig.

16).

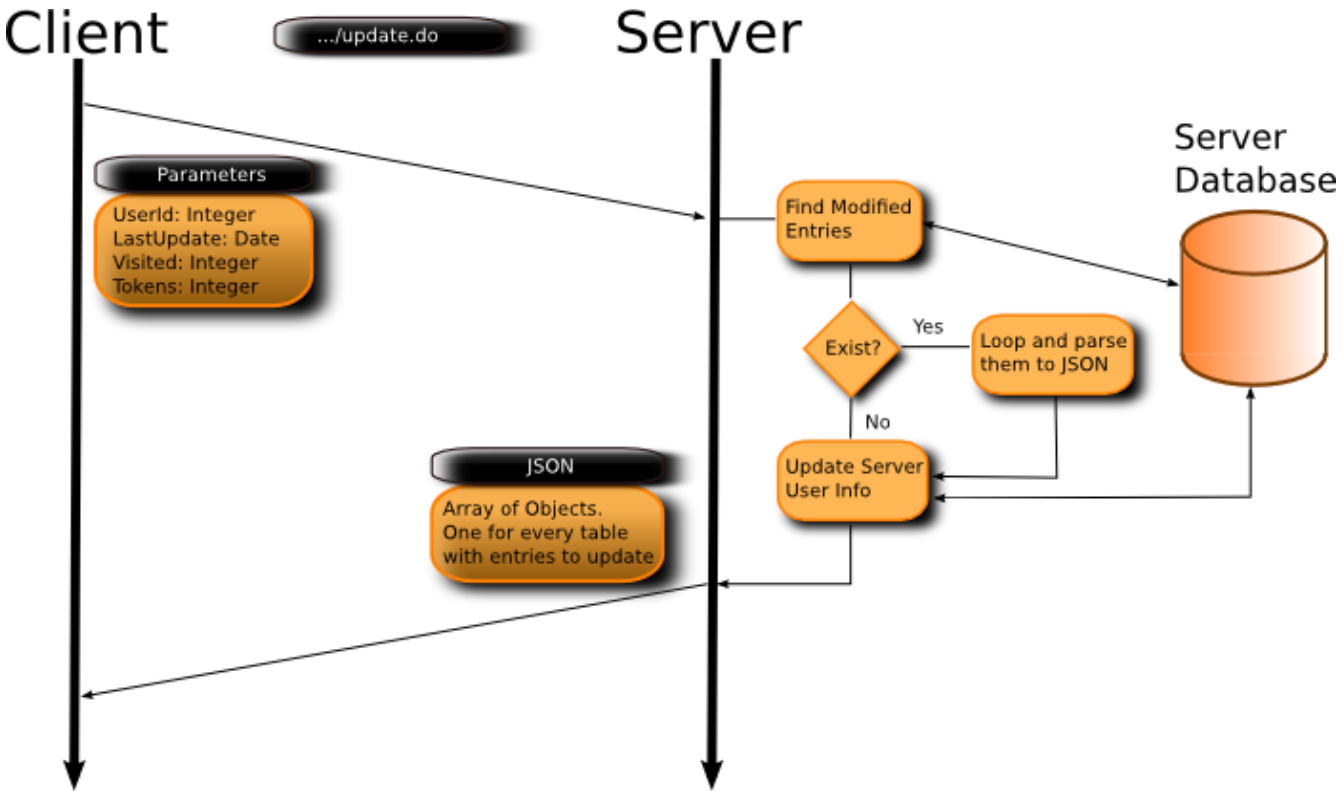


Figure 16 - Update logic

- Upload [.../upload.do] (available on mobile client)
  Every time a user rates and writes a short review about a location, this is submitted to the server through the upload action. The parameters needed for it to work are:
  - Rate: a whole rate entry is concatenated to the URI, including all its fields except for the rate identifier, which is created on the server.
  - Rec: an additional boolean variable is set, indicating to the server whether this user qualifies for a recommendation process or not.

The upload action starts by parsing all the parameters and creating a new instance of the 'Rate' entity class (Fig. 17). The method checks whether the pair 'userId' 'locationId' already exists in the database, in order to avoid duplicate entries and cheating and inserts it if it is not present. Right after that, it searches, modifies and edits the two 'Score' entries involved in the rating process (the global and the one referring the profile of the rating user). This also happens as a single process in the controller class to guarantee that the operations are done atomically. The scoring mechanism is rather simple: all it involves is counting the number of times it has received a valoration and calculating the average of the stat. In the case the user applies for a recommendation, the static method to find it out is executed next (this system is explained in detail in section 4.4]. The response is always an 'IdAffinityPair' class – an object containing two values, an integer and a float – whose content varies in one of the following:

- Integer = -3; Float = 0.
  When the user does not apply for a recommendation or when detected that this rate pair was already submitted.
- Integer = -2; Float = 0.
  The user applies for the recommendation; however the algorithm has not been able to

find an affine candidate.

- Integer = -1; Float = 0.
  The user applies for the recommendation and the algorithm has found one or more affine candidates, however none of the possible locations they could recommend surpassed a predefined threshold.
- Integer = the identifier of the recommended location; Float = the estimated score.
  The user applies for a recommendation and it has been found among the possible candidates and their proposed locations. The float calculates the estimated score that the requesting user would give this location, based on the correlation of information between users.

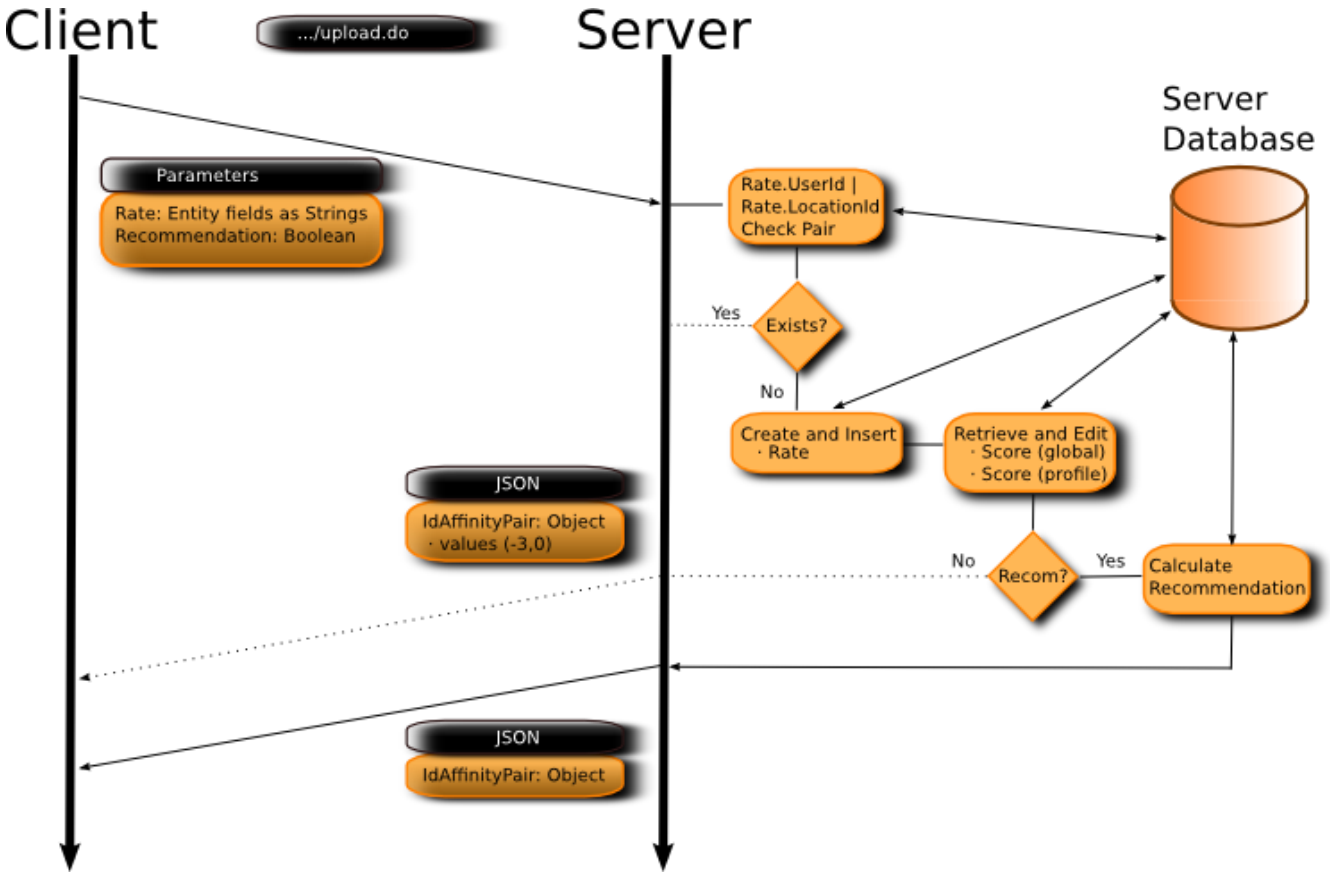


Figure 17 - Upload logic

- Dump [.../dump.do] (on site)
  This action is available from the main screen of the Servlet application. The only thing it does is a dump of a database to a table-cell format. Useful for checking the contents, but likely to be removed before the beta goes out.
- Statistics [.../statistics.do] (on site)
  Yet to be implemented. This action should provide the manager with some useful information about the usage of the application. In addition, it could serve as interesting info for current and future clients to see.

4.5 Recommendation system

One of the key features of this application is its capability to keep track of the user's rates and decisions

and propose recommended locations by correlating this information with that of other clients. For the implementation of such a system there were multiple different options, ranging from the possibility to use any of the already existing algorithms to the ideation of a new solution. Albeit this section plays a major role in the uniqueness of the whole system, it is not the goal of this project to develop a brand new algorithm to address this need; there are several contests with monetary prizes given by the big players for improvements of their existing systems. Instead, a quick review has been made to the generally accepted solutions and the following has been picked as suitable.

This recommendation process involves the next steps:

1. Identification of candidate users that have visited and rated a portion of locations similarly to the user requesting the recommendation.
2. The affinity (the degree of similitude between rates) among the candidates and the user is calculated with the cosine similarity.
3. Each of these candidates then recommends its best location (one that the user has not visited yet).
4. Normalize the rating locations in order to compare them on the same scale and then multiply it by the affinity coefficient calculated previously.
5. The highest value (over a set threshold) is the one of the location sent back.

Next some details of the cosine similarity and rating normalization are introduced, and this process is described in detail in section 4.5.3.

4.5.1 Cosine similarity

The cosine similarity[23] is a measure of similarity between two entities which are represented as vectors. Once those vectors have been projected over space, the cosine similarity is the cosine of the angle between both of them; thus, resulting in 1 for superimposed vectors (cosine 0º) and 0 for the perpendicularity drawn by two vectors in an angle of 90º. According to the cosine similarity, two vectors are similar if the angle they form is 0 – distance between the vectors is zero.

In mathematic theory there are infinite dimensions and the formula of the cosine similarity has been extended and prepared for such situations. This comes in handy, as in this application each of the scores given for a specific location is represented as one dimension, resulting in something impossible to draw (in most cases), but very easy to understand and apply if broken down with an example and calculated with the given formula (Eq. 1, 2).

Equation 1- Cosine similarity (1)

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$$

Equation 2- Cosine similarity (2)

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

In other words, a user who has rated locations A, B and C with a score of 5, 10 and 8 respectively would have his vector be [5, 10, 8]. Let's calculate the cosine similarity of the following table with user rates (Table 1):

Table 1 - Cosine similarity example

User \ Location	A	B	C	D	E	F
User1	7	9	5	10	?	?
User2	9	10	9	5	-	9
User3	2	8	6	10	8	-

Equation 3 - Cosine similarity calculation (1)

$$Similarity(1,2) = \cos(\theta) = \frac{(7*9 + 9*10 + 5*9 + 10*5)}{\sqrt[2]{(7^2 + 9^2 + 5^2 + 10^2)} + \sqrt[2]{(9^2 + 10^2 + 9^2 + 5^2)}} = 0.9167$$

Equation 4 - Cosine similarity calculation (2)

$$Similarity(1,3) = \cos(\theta) = \frac{(7*2 + 9*8 + 5*6 + 10*10)}{\sqrt[2]{(7^2 + 9^2 + 5^2 + 10^2)} + \sqrt[2]{(2^2 + 8^2 + 6^2 + 10^2)}} = 0.947$$

In this example, the calculation of the cosine similarity between users 1 and 3 (Eq. 3) is higher than that between users 1 and 2 (Eq. 4), after applying the above stated formula. Thus, if user 1 is trying to find which user is more affine to him for requesting a recommended location, in the above example user 3 would recommend location E. Location E has been rated as 8 by user 3, while the alternative (location F) has been rated higher (9) by the user with whom there was lower affinity. Would this recommended location be correct or their rates also should matter? More on this topic in the next section.

4.5.2 Normalization

However, as the reader has possibly already detected, a strange situation might happen when two users with the vectors [1, 1, 1] and [9, 9, 9] are compared (there are many more case like this one; this should serve only as an example). The output of the algorithm would indicate that the users are totally affine one to the other, while the given scores could be anything but similar. This is fine, both in mathematical terms and in the meaning they are given in this project. In the former, this situation indicates that the vectors have indeed the same direction on the plane; only the values of one of them are smaller in comparison to the other, thus, resulting in a smaller vector module. On the latter, one has to think in relative terms and abstract the results.

How is a full affinity between such distant users a good result? Well, one has to think that the cosine similarity is only one step of the recommendation process and that the final goal is to give the requesting user a location such that he will rate it the highest among all the other possible locations the system could choose. In that sense, a user whose low rates are always scored high by the requesting user could be useful. Even the lowest score could be a choice if it is always matched by the highest score by the user the recommendation is calculated for: this is what *relativity* stands for. Nevertheless, it is not as easy as just using the cosine similarity like this, as further questions arise: is it equally good a recommending user with a vector of [1, 1, 1] than a user with a vector of [5, 5, 5]? What if the requesting user has a vector of [9, 8, 7] and the possible choices are [5, 4, 3] and [5, 3, 1], is there an optimal candidate?

Logic indicates that any user rating with the same score has actually all the range of scores at that value and, a user scoring only in a fixed range is subject to the same clause. This is the *spread* of the scores; a wider spread allows for bigger precision when recommending, while a smaller one introduces more uncertainty about any future variation of the recommending user. In addition, the certainty increases

with a larger vector (more dimensions or rates made) and decreases when it is smaller. Logic is fine, but in this project it is needed to use a provable mathematical formula that encompasses all the former: this is where the normalization takes part.

The normalization process transforms every vector to a new spatial plane, where the module of all normalized vectors is 1, allowing for comparison between core-different vectors and with different lengths too. It consists in calculating the module (Eq. 5) of the vector and then dividing the specific score of the candidate location to be recommended by it. After that, the result is multiplied by the module of the vector of the user requesting the recommendation. The module of a vector is calculated by finding out the square root of the summation of every component of the vector up to the power of two.

Equation 5 – Unitary vector

$$|v^{\rightarrow}| = \sqrt[2]{\sum_{i=1}^{n} v_i^2}$$

If the example of the previous section is retrieved, this normalization process can be applied to find out a more adequate result or to justify the previous recommended location (Eq. 6, 7, 8).

Equation 6 – Unitary vector calculation (1)

$$|v^{\rightarrow}_1| = \sqrt[2]{7^2 + 9^2 + 5^2 + 10^2} = 15.96872$$

Equation 7 – Unitary vector calculation (2)

$$|v^{\rightarrow}_2| = \sqrt[2]{9^2 + 10^2 + 9^2 + 5^2 + 9^2} = 19.1833$$

Equation 8 – Unitary vector calculation (3)

$$|v^{\rightarrow}_3| = \sqrt[2]{2^2 + 8^2 + 6^2 + 10^2 + 8^2} = 16.3707$$

Note how all the location's rates are included for users 2 and 3 (even those of the potential recommendation) to achieve a true unitary vector for the normalization. After this, the candidate locations have to be multiplied by the module of the vector of user 1 and divided by the module of the vector of their own user (Eq. 9, 10).

Equation 9 – Normalized rate calculation (1)

$$NormalizedRate_2 = 9 * \frac{|v^{\rightarrow}_1|}{|v^{\rightarrow}_2|} = 7.4919$$

Equation 10 – Normalized rate calculation (2)

$$NormalizedRate_3 = 8 * \frac{|v^{\rightarrow}_1|}{|v^{\rightarrow}_3|} = 7.8036$$

The candidate score can then be normalized to reveal its true value and multiplied by the cosine similarity in order to estimate how the requesting user would rate the proposed location.

If this is done for the example here proposed (Eq. 11, 12):

Equation 11 – Estimated value calculation (1)

$$EstimatedValue_2 = 7.4919 * 0.9167 = 6.8678$$

Equation 12 – Estimated value calculation (2)

$$EstimatedValue_3 = 7.8036 * 0.947 = 7.39$$

The result is that the best recommended location would be the one proposed by the user 3. In this example, the result supports the initial approach of utilizing only the cosine similarity; however, this will not be always the case, so a full calculation as explained here has to be done.

4.5.3 Implementation

This section provides a detailed explanation on how this recommendation system has been implemented in the project: first on the client side and after that on the server one.

In order to qualify for a recommendation, there is a requirement that the user must met. To avoid calculating the cosine of one-dimensional vectors (when the user has only visited one location) and to help reduce potential cold-start issues, a number of locations have to be rated before being considered for a recommendation application. This number is now set on 3; however it should be clear that this is not the result of some analysis carried out: 1 rate was simply ruled out for obvious reasons and 2 rates were dismissed, trying to enroll the client a bit more before starting to offer recommendations. Starting from the third rating, the server tries to find a recommended best location every two user rates (on 3, 5, 7, 9...).

On the server side, there is a set of queries and calculations that has to be done in order to try to return a recommendation. The queries, on a much larger scale than the operations, are costly in the time domain as for most of them the access to the database has to be restricted (acquiring a lock) and the calculations over the result set of elements has to be done in a specific order. With this in mind, it has been tried to do these queries and operations in the less restricting and time consuming way, acquiring some of the result sets beforehand and doing the heavy work with the database unlocked. During the following explanation, for the sake of clarity, the user who has applied for a recommendation is going to be the 'requesting user', the user which could possibly make him a recommendation are going to be the candidate users and the best rated location for each of these users (not visited by the requesting user) is labeled the candidate location. The process can be followed in (Fig. 18).

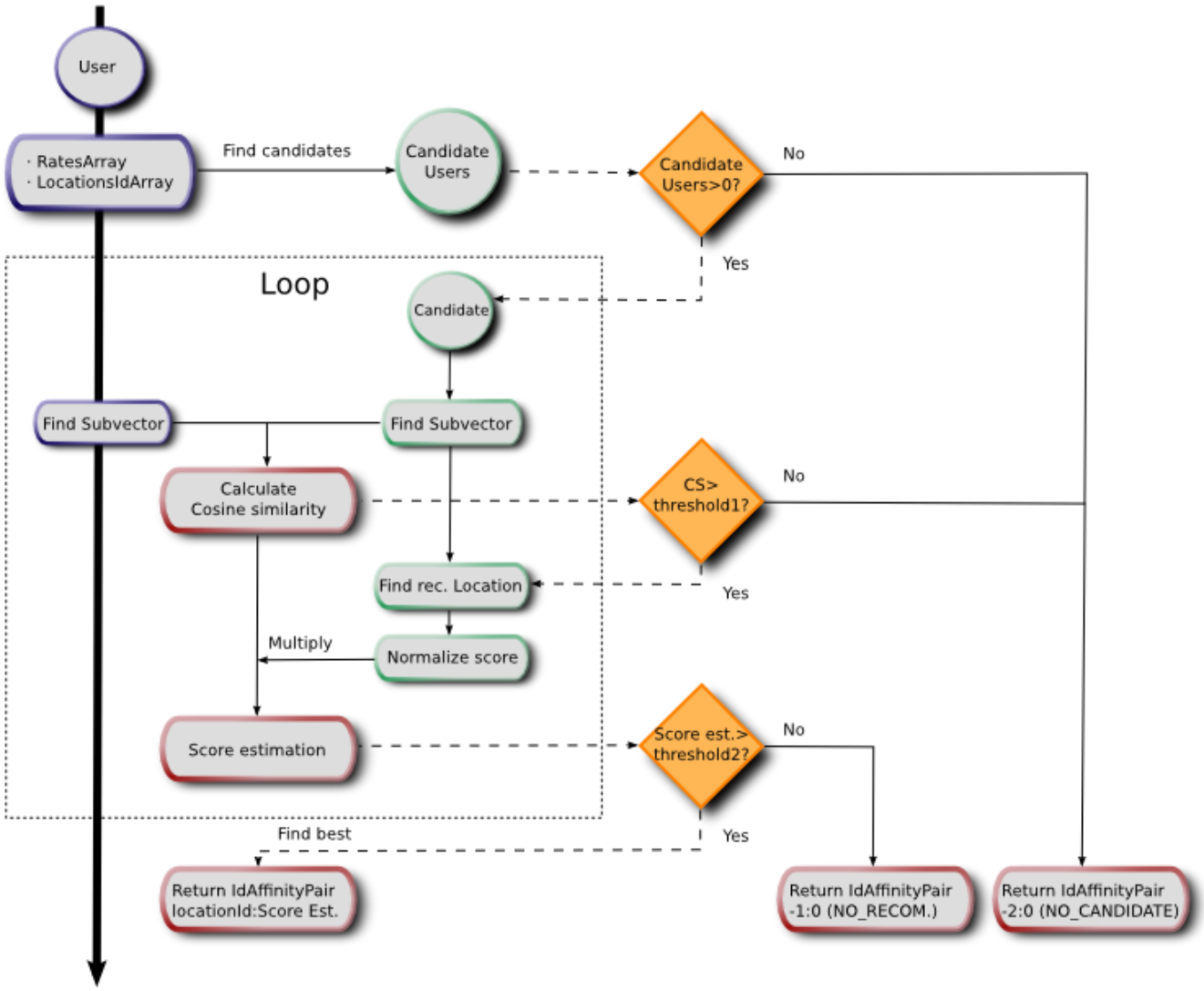


Figure 18 - Recommendation process

First of all, all the user rates are acquired from the database as entities and stored in an array like that, which allows an access to them and their processing during the rest of the phases. One of the first things that is done is strip those rate entities down to only the integer that indicates the rated location identifier; a recurring element that is stored in a new array. Originally it was thought that a filtering of the entire requesting user's location would benefit the server in reducing the cost of time – something along the lines of picking a bigger percentage of random 'newer' rates and a smaller percentage of 'older' rates. The question was however, would that really be a benefit? And would that benefit come at the price of introducing a weakness in the system? Since the answer to these questions is truly unknown, and cannot be known without carrying out some serious documented analysis, the decision was to use all the available data as a valid start point. If severe time-sinks were detected sample reduction and performance tests would have been carried out; thankfully this was not needed.

The next task is to find each and every user that could make a recommendation. The guide for this was simple yet efficient: find every user that has rated at least 2 of the locations that the requesting user has rated AND that has at least one additional rated location, not present in the requesting user's sample of rates. The query to find those users has been done in two parts, using the handy JOIN operator (Code.Snippet 12):

```
"SELECT * FROM (SELECT `userId` "
                    + "FROM `rate` "
                    + "WHERE (`locationId` IN" +
  + "requesting_user_rated_locationIds) GROUP BY `userId` HAVING
COUNT(*) >= 2) AS A "
                    + "NATURAL JOIN "
                    + "(SELECT `userId` "
                    + "FROM `rate` "
                    + "WHERE `locationId` NOT IN ("
                + "requesting_user_rated_locationIds)"
  + ") GROUP BY `userId` "
                    + "HAVING COUNT(*) >= 1) AS B";
```

Code.Snippet 12 - Candidate searching query

Once all the candidate users are gathered it is time to calculate how the requesting user would score each of their recommendations. Using the stripped identifiers two sub-vectors are found (one for the requesting user and one for the candidate user), each of them containing the rates that the users have in common. Using these vectors it is straightforward to find out the affinity between the users, calculating the cosine similarity. This is done with every pair of users, filtering out all those candidates whose affinity coefficient does not reach a predefined threshold, as there is no sense in wasting computational power and time in users whose recommendation could hardly be good enough for the requesting user. The threshold has been hardcoded as 0.65, a value not based on any previous tests or studies, just the feeling that if any candidate user's rates are not related in at least two thirds with the requesting user, there is little left to be done.

In the second phase, every single candidate user is subject to further calculations. To start off, a full vector containing all its rates is requested to the database and used to calculate its own module. There was some uncertainty whether the whole vector should be used, rather than only the sub-vector or the sub-vector and the candidate location. Aiming for a more accurate application of the normalization technique, the whole vector was finally used, albeit that also meant a reduction in the estimated scores of the requesting user (as the spread was wider). Following, the candidate user's highest score for an unvisited location (for the requesting user) is queried; this score is normalized dividing by the same user's vector modulus and multiplied by the requesting user's vector modulus. To obtain the final estimated score that is supposedly going to be given by the requesting user, the former result is multiplied by the affinity coefficient calculated in the first phase. At the end of this second phase there is another hardcoded boundary set at 7, under which final estimated scores are discarded. Both thresholds are backed up by no theory other than some empirical tests; they can be modified or removed should it be needed. To conclude, driven by the caution of time costs, the initial version contained another filtering function for the selection of candidate users which consisted in picking them in small groups of 5 users, skipping to the next group of candidates only if none of the estimated scores of the previous group improved both thresholds. This was proved tremendously wrong as it could produce errors such as returning a poorly rated recommendation (even if it overcame the boundaries) only because of a highly affine user – even inside the same batch of candidates. Since the end decision of the recommendation goal was to return the best candidate location among everything in the database, this initial version was rapidly discarded. In addition, the final costs of all these operations are low enough to not suppose a threat to the well-functioning of the server.

Let's remember that this recommendation process is a part of the upload mechanism through which a user submits a rate for a given location. The result of which is, thus, dependent on several factors. However, should it be successful, the returned data is a key-value pair containing the recommended

location and the estimated score the requesting user would give to it, based on the affinity and the normalization processes.

4.6 Security

Nowadays security has turned out a major topic for any service connected and open to the Internet, even more than before and major issues as vulnerabilities and attacks have not ceased to increase. There are different types and degrees of security and the wisest thing is to apply them in several layers, not trusting a single protection system, but also to grant these on a per-need basis. In other words, do not introduce security procedures and features if they are not strictly necessary to protect a particular segment of the system; that would not only over-complicate the system without introducing benefits, but also result in the introduction of new points of attack and failure.

In that sense, the project here described does not have very complex security needs. In fact, besides some basic common-sense security measures that are generally applied to most projects, there is only one point that needs to be covered to offer a good user experience: the avoidance of user implantation as that would deter the normal progress and satisfaction the legit clients get from the application usage. Still, some other minor precautions have been made while developing this project and they are explained in the next sections.

4.6.1 Database

The database is one of the key points that have to be protected from unwanted access and misuse. It is important to keep in mind that every project should have a separate dedicated database, never risk your information merging everything in the same schema! This rule has been preserved while designing and deploying the LiveGuide server database.

Another thing that has been done to secure the database is to create a specific user for any transactions that involve the LiveGuide server schema. This approach guarantees that if any exploit or weakness is found in the server code, the attacker would only have access to the schemas allowed to the application's database user. Root should never be used for this purpose.

Finally, an additional safety measure is to limit the number of actions the database user can perform on the schema. If, during the normal execution of the program, the user's tasks are only INSERTs, UPDATEs and DELETEs, there is no need to grant him permissions for executing other operations such as DROP or CREATE which, in case of a security breach, could cause a severe loss of information.

4.6.2 Queries

One of the most prominent attacks, albeit it is quite old and well documented, is the SQL injection. This attack consists in the submission of additional SQL code actions inside requested parameters for operations in the database; code that is executed if not parsed adequately. For example, if in a field expecting a user name parameter the following is introduced:

```
ROBERT '); DROP TABLE users;
```

It would turn out in two different consecutive queries being executed if the input is not sanitized adequately. In this case ROBERT would complete the first already defined query inside the code (which could be successful or not after being ended with the '); characters) and the rest would execute as the second query, erasing the table 'users' (if present) with all its content.

Numerous web applications fail to provide an adequate protection against it and are victims of an attack that does not require any tools or skills – anyone could carry it. When using the JPA there are a number

of ways through which a developer can secure the content of the executed queries against the database, but there is also the possibility to use native SQL, decision that could bring an insecure situation. Among the secure ways of managing queries with the persistence API are:

- Positional parameter in JPQL
```
Query jpqlQuery = entityManager.createQuery("Select order from Orders order where order.id = ?1");
List results = jpqlQuery.setParameter(1, "123-ADB-567-QTWYTFDL").getResultList();
```

- Named parameter in JPQL
```
Query jpqlQuery = entityManager.createQuery("Select emp from Employees emp where emp.incentive > :incentive");
List results = jpqlQuery.setParameter("incentive", new Long(10000)).getResultList();
```

- Named query in JPQL
```
"myCart" = "Select c from Cart c where c.itemId = :itemId"
Query jpqlQuery = entityManager.createNamedQuery("myCart");
List results = jpqlQuery.setParameter("itemId", "item-id-0001").getResultList();
```

- Native SQL
```
Query sqlQuery = entityManager.createNativeQuery("Select * from Books where author = ?", Book.class);
List results = sqlQuery.setParameter(1, "Charles Dickens").getResultList();
```

All the above approaches use parameterized inputs bound to the data and the JDBC driver will escape this data adequately before the query is executed. An example of a vulnerable query usage is:

-
```
List results = entityManager.createNativeQuery("Select * from Books where author = " + author).getResultList();
```

In the case that the data used in the last query has not been escaped or sanitized it could contain malicious database code (payload) that would get executed, leaving the server vulnerable to SQL injection attacks.

4.6.3 Authentication

The last measure to add to the project is a authentication or use validation mechanism which would prevent any impersonation – a user submitting rates in behalf of another user. A very simple way to achieve a secure transmission and authentication would be to inject the @WebServlet, @ServletSecurity and @HttpConstraint key parameters; however that would not allow the server to make distinction between different users. A more complex but effective solution is to use the JDBC Realm for authentication.

An authentication realm is nothing but a domain scope of security policies. The realm itself consists of users and groups or roles to which they may or may not be related; for example, LDAP is a realm. The interesting thing of the JDBC realm is that it is bound to the database, which provides a lot of benefits besides not needing an administrator to manually add, delete or modify users, as these processes can be automated or left up to the users. To configure and set up a JDBC realm the following is needed:

- A table in the database which contains the login unique information for every user along the

password for that login. In LiveGuide server, this table is the 'UserAccount' as noted in chapter 3.

- Another table for mapping between the user login and the role or privileges it has inside the system. In LiveGuide server, this table is the 'UserGroup' table.

In Glassfish server, several configuration options are available through the administrator console, available on the port 4848 of the machine where it is located. Through one of the wizards on that console, it is possible to create and configure a JDBC realm, linking it to the pertinent database tables and to the deployed Servlet. To do so, a new realm must be created in Configuration → Security → Realm; the configuration parameters for LiveGuide server are as follow:

- Name: LiveGuideRealm
  The name of the newly created realm. It can be referenced with it from the Servlet.
- Class Name: the one ending in 'jdbc'.
  This one is pretty straightforward; among the different types of available realms this is the class of the JDBC one.
- JAAS Context: jdbcRealm
  Ensures the authenticated caller has the access control rights (permissions) required to do subsequent security-sensitive operations.
- JNDI: liveguide
  This field contains the name of the JNDI resource, which should be the same as the one used when creating it from the connection pool for the database (as explained in chapter 3).
- User Table: userAccount
  This field asks for the name of the table in the database which will contain the user login-password pairs.
- User Name Column: userName
  The column from the user table which references the login information. This column contents should be unique (for example, by using it as a primary key and controlling the input).
- User Password Column: userPassword
  The column from the user table which references the password link to the user login information.
- Group Table: userGroup
  This field asks for the name of the table in the database which will contain the user login-role mapping, used when deciding if a user has or has not permissions to execute or access certain content or action.
- Group Name Column: userName
  The column from the group table which references the user login information.
- Group Role Column: userGroup
  The column from the group table which specifies the role of the user it is linked to.
- Digest Algorithm: none
  It is possible to use a digest algorithm to store the passwords. For simplicity reasons it has been left out for this demonstrator. Nevertheless, it should be a priority when launching the beta or for any software that is in the “production” phase.

Once the realm is created, it is time to link it from the Servlet for it to work. The first thing is to make

aware the software of the possible existing roles in the database; to do that they have to be declared in the sun-web.xml configuration file. It might happen for this file not to be initially present; that is the case of many Web Application projects. It is safe to add it manually, taking into considerations that under the Netbeans environment (the one used in this project) there is an equivalent configuration file that should be preferably used, the glassfish-web.xml. This is how a role is mapped (Code.Snippet 13):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 Servlet 3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
  <security-role-mapping>
    <role-name>ADMIN</role-name>
    <group-name>ADMIN</group-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>USER</role-name>
    <group-name>USER</group-name>
  </security-role-mapping>
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>Keep a copy of the generated servlet class' java
code.</description>
    </property>
  </jsp-config>
</glassfish-web-app>
```

Code.Snippet 13 - Role mapping in glassfish-web.xml

More configurations involve specifying the login type and related realm in the web.xml deployment descriptor (Code.Snippet 14).

```
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>LiveGuideRealm</realm-name>
</login-config>
```

Code.Snippet 14 - Real and authentication type specification

In this same file, the security roles that were defined in sun-web.xml or glassfish-web.xml have to be declared as available for use during a login (Code.Snippet 15).

```
<security-role>
        <description/>
        <role-name>ADMIN</role-name>
</security-role>
<security-role>
        <description/>
        <role-name>USER</role-name>
</security-role>
```

Code.Snippet 15 - Security roles declaration

Finally, particular constraints can be put in place virtually for any part, action or path of the application (Code.Snippet 16). The 'update' and 'upload' actions should only be available to existing users, so a constraint is added specifying that any URL pattern containing *.do falls under this condition.

```
<security-constraint>
        <display-name>Main</display-name>
        <web-resource-collection>
            <web-resource-name>LGS</web-resource-name>
            <description/>
            <url-pattern>*.do</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name>ADMIN</role-name>
            <role-name>USER</role-name>
        </auth-constraint>
        <user-data-constraint>
            <description/>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
<security-constraint>
        <display-name>Account</display-name>
        <web-resource-collection>
            <web-resource-name>CreateUser</web-resource-name>
            <description/>
            <url-pattern>/createUser.do</url-pattern>
        </web-resource-collection>
</security-constraint>
```

Code.Snippet 16 - Security constraint declaration

As it can be seen, the user roles to which this constraint applies are also set at this step. Let's remember too that the information is sent in plain (basic login type). This would leave user account information vulnerable to interception attacks; as this has to be avoided, an additional tag with transport guarantee is included in the configuration. A 'confidential' value indicates that the transmission should occur only after a secure encrypted channel has been set up; in the project's scenario that would mean HTTP connections would turn HTTPS.

5 Client

This chapter provides detailed information about the LiveGuide client software. Unlike the server which is usually unique or, in some cases distributed or mirrored, there are multiple instances of a client, generally one for every user of the service. Clients are also software that runs as a single (or sometimes multiple) batch of processes, allowing users to perform a predefined set of actions locally or with communication to the server. Among much other possible segmentation, clients can have a graphic user interface (GUI) or not, being the latter the typical approach for commercially targeted applications and for those wanting to reach a wider user spread.

LiveGuide client is a piece of software devised for portable devices with a graphical interface that is used for information storage and examination purposes and for periodically reporting the progress of using that information to the LiveGuide server. The client works by combining different layers of software and regulating the communication between them; detailing this is the topic of the first section of this chapter. In the next section, the full application usage is reviewed, linking a more technical view with the functional analysis explained in chapter 2. To conclude the chapter, a breakdown of the client interaction with the server is included, making an emphasis on the interchanged requests and responses.

5.1 Client structure

Nowadays, there is a broad choice of mobile devices capable of running third party software in its core; these devices range from smartphone terminals to tablets of different sizes and hardware specifications. One of the first decisions which face software developers is that all those devices are bundled with their own operating system, making programming for them core-different and forcing them to choose one language/platform and stick to it. Albeit there are some major players like Android OS and iOS, let's not forget that, even with marginal percentages in some cases, there are still millions of users of different operating systems such as BlackBerry OS, Windows Phone, Symbian, WebOS... (Fig. 19) Another challenge that developers find is that even under the same operating system, there are several different manufacturer brands (some of which have particularities) and a huge variation in system specifications such as screen sizes, processing power or memory. On top of that, relating the two issues above, a third one arises: different hardware specifications lead to the support of specific versions of an operating system, which could leave out important features unsupported for certain applications.

Figure 19 - OS and device brands

With these difficulties in mind, the decision was to build an application such that it could be used by almost any operating system and supported by hardware aged at least 4 years. One of the tools that allow to do this is the Apache Cordova framework (formerly known as PhoneGap)[4], as it was noted in chapter 2. The magic in PhoneGap happens by invoking a container in the native language of system which is capable of processing and viewing web page programming languages such as HTML5, CSS and JavaScript. For example, in Android, this is done creating a WebView and carrying all the action and user input on this same Activity screen.

Although the initial intention was to develop the whole client using this framework, there was a specific functionality point that could not be addressed: the permanent geolocation. The geolocation is a feature present in PhoneGap and the framework has done a frankly good work integrating the sensor functionality into their code, however it can only be used when the application is in the foreground. Portable devices applications work different than the applications people are used to in their PC. On them, multiple processes are capable of running concurrently and, even if the user switches screens or opens new software, he can be sure that all the other processes will still be executing in the background. With smartphones and tablets an application will only "run" when it is on screen. If this stops being true the application is suspended and none of its code will run. This is where PhoneGap lacks: it only allows creating Activities (again using the Android case as an example) which are fine 99% of the time – application features can be used as long as they are on the screen. But when one of the main features of the application needs to keep track of the client's position and compare it periodically with the locations that can be visited, when this has to be done even if the application is paused in the background, even if it is closed or has not been opened after a system restart... then, PhoneGap's geolocation management is not enough. In fact, it is not the geolocation implementation per se; it is the lack of access of PhoneGap to the creation of background services. That issue left two possible approaches for the client: literally ask the clients to open the app and wait for it to detect the match when on one of the possible location sites or find a workaround. The first approach has nothing wrong, a lot of applications use it constantly and users are used to it when publishing a place they are eating out at, a newly discovered shop... at most social networking apps. However, the second approach presents an alternative way to do it, enhancing the user experience; a chance to play around with the interaction between layers and to enhance the value of the whole project, trying to build it in a new way. Thus, a PhoneGap plugin with native code has been used to achieve the creation of a background service with the aforementioned features, explained in detail in the next sections of this chapter (Fig. 20).

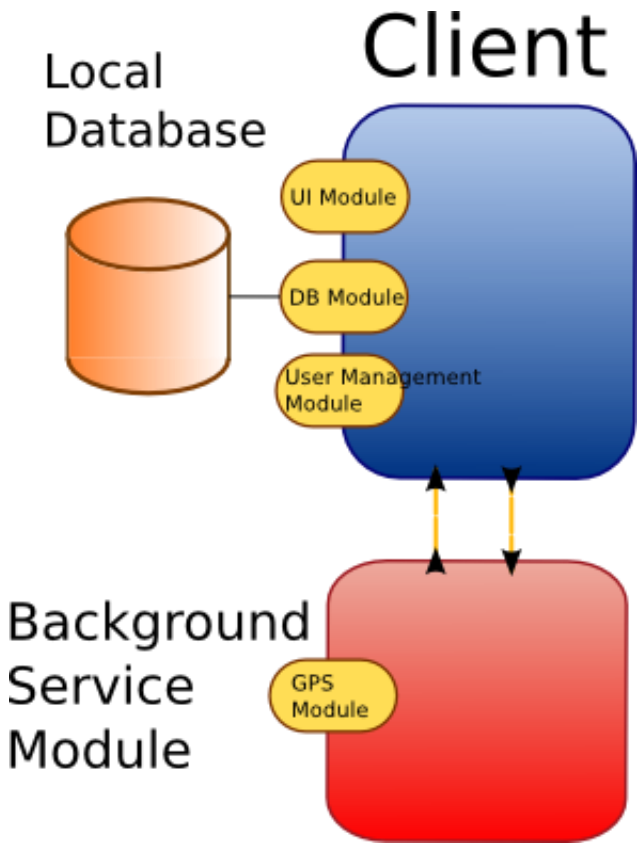


Figure 20 - Client structure

5.2 Application

PhoneGap applications run inside a web browser-alike viewer, as it has been noted. As most web pages, the components of these applications are HTML files, holding the structure of the document, its tags, identifiers and classes; CSS files managing the appearance, positioning and styles of the different elements; JavaScript files in charge of the navigation, flow, events and execution of all the complex needed functionalities.

The usage of the jQuery and jQuery Mobile libraries in this project allow for a simple structured and intuitive navigation through what has been noted as 'page' elements. With the usage of this concept in the libraries it has been enough to have a single HTML file containing all the available pages in the application and navigate through them AJAX-style with the use of identifier tags preceded by the sharp character '#'. This is what the main menu code looks like in the client software (Code.Snippet 17):

```
<div data-role="page" id="populate">
                <div data-role="header">
                        <h1>Header</h1>
                        <a href="#" data-rel="back">Back</a>
                </div>
                <div data-role="content">
                        <p>Content</p>
                </div>
</div>
```

Code.Snippet 17 - Page structure with back button

The mobile library version of jQuery already comes with predefined CSS rules which have been included in the project without modifications, except for two cases: the adaptation of the sliding lateral plane minimum size to a more suitable value and the inclusion of a custom centered image for the 'unlock' button in the list of elements used in the 'Discover' screen. Everything else has been used as it is.

A web page with static content and simple navigation through it is from two decades ago and will not be enough to cover the requirements of this project. What is needed is to dynamically enhance and modify the content, keep track of the user's progress, bind elements to events on the fly and offer full compatibility with the platform sensors and communication technologies. All these events and logic is handled in the LiveGuide client by JavaScript. The next sections are dedicated to explain some bits of the JavaScript code that handles events and some of the main functions involved in the lifecycle of the app, breakdown the plugin background service, detailing how does it communicate with the main piece of software and give an overview about how the geolocation works. In addition, a small section about the html5sql libraries and how they are implemented for local database storage is included.

5.2.1 Lifecycle

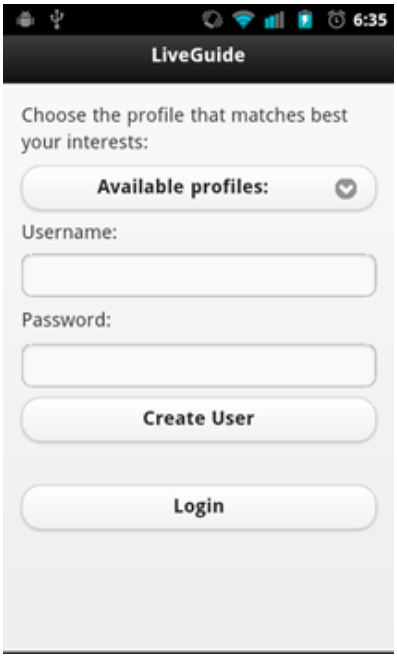
The application consists of 3 major phases: the initial phase, on which the databases are created or loaded and the data structures are set up, allowing a user log in or the creation of a new account; the main phase, at which all the features of the application are opened (updates, uploads...) and where the discovery and progress stages occur; and the dormant phase, at which the application is either closed or in background, but the plugin's service is active checking the location coordinates and waiting for a match to awake the main app. The first two phases are discussed here, while the third one is on the next section.

It is quite common to require the execution of certain tasks and to set up specific structures at the initialization of the software, before the user interaction has started. In web pages, traditional jQuery allowed to do so right after the DOM tree was initialized, binding operations to the $(document).ready() event. However, when using jQuery Mobile one has to keep in mind that the whole tree (for every single page in the web app) is initialized in the same way at start and that event cannot be used except for the very first execution of the software. As detailed in jQuery Mobile's documentation and noted in the second chapter of this report, this library is able to work on a per-page basis (being a page every content surrounded by those tags in the loaded HTML document) and can act separately on a number of different predefined events. The event used to set up when one of these pages is about to be loaded and shown has the 'pageinit' name and is called in this way (Code.Snippet 18):

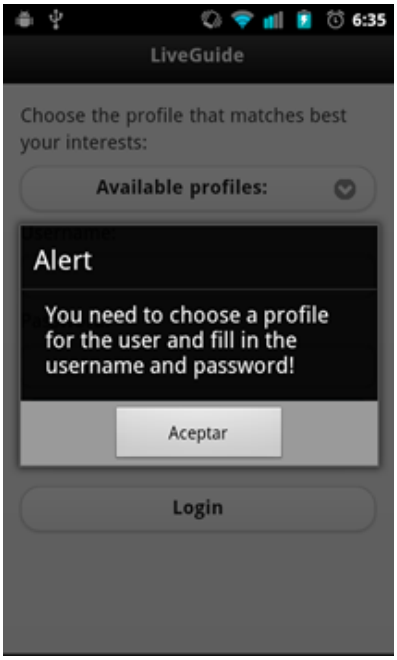
```
$(document).on("pageinit", "#locations", function() {
    $('#locations').on('pageshow', function() {
        refreshEntries(1);
        refreshTokens();
    });
    $('#navbarLocations').on("click", "li", function(event){
          refreshEntries($(this).find('a').attr('id').charAt(1));
        });
});
```

Code.Snippet 18 - Event binding: 'pageinit'

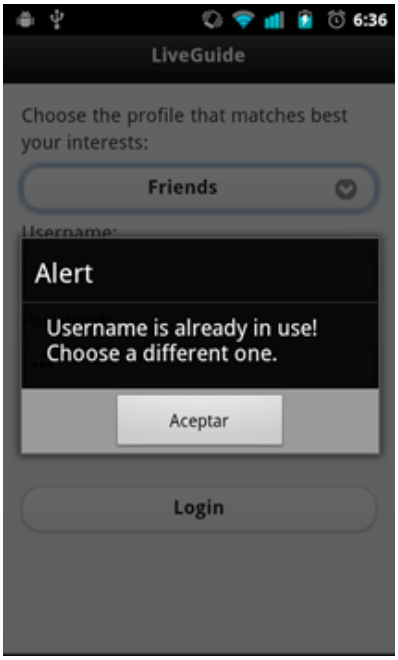
The first thing that the software tries to do when it is started is to create and populate the database with the bundled files in the package. At doing so, a boolean variable is set in the local storage of the WebView, allowing to skip these steps at the next start. Right after that, the 'User' table is checked and the user information is loaded in a global variable if present, changing the text of the 'Login' button as can be seen on (Capture 5). If the user information is not found, the standard home screen suggests the creation of a new account as shown on (Capture 1, 2, 3, 4).



Capture 1

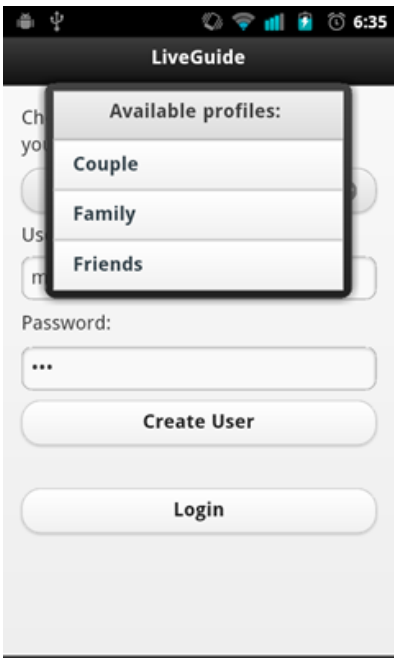


Capture 2

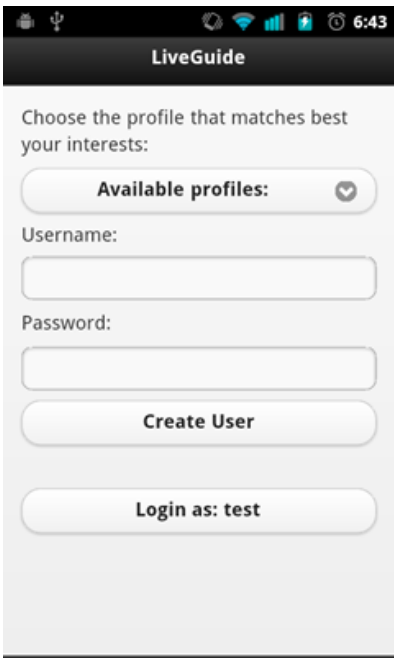


Capture 3

The flow diagram of this initial phase can be seen in chapter 2: 'Functional Analysis' section.



Capture 4



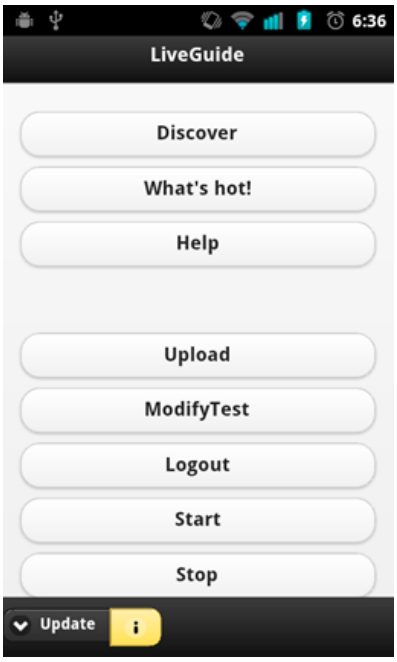
Capture 5

The second phase is actually the functionality complex offered to a logged user. On the main screen, several buttons offer navigation paths which are bound to the click event through the use of dynamic binding (Code.Snippet 19).

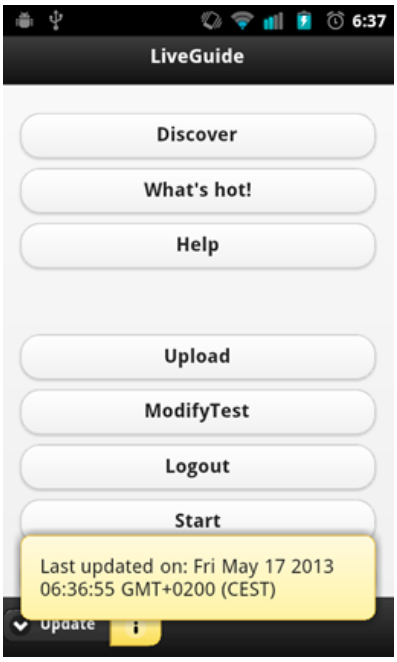
```
$('a[href$=#update]').on("click", update);
```

Code.Snippet 19 - Binding events

The update button calls to a function that executes an action on the server; if it has been successful, the value of the small informative pop-up besides it is updated with the adequate date value (Capture 6, 7).



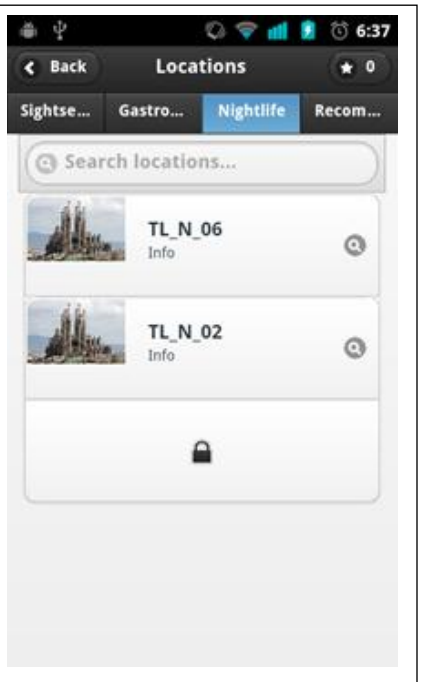
Capture 6



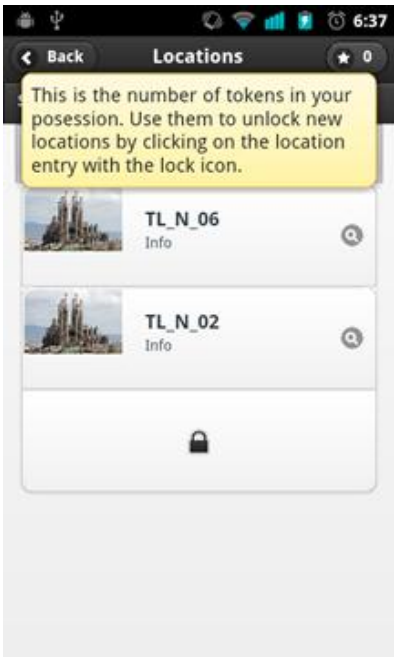
Capture 7

The main functionality is provided behind the page triggered by clicking on the 'Discover' button – the starting point for every city-discovery journey (Capture 8). The information placed here is given in the form of location elements, conveniently structured in categories through which the user can navigate thanks to the 'Navigating Panel' right below the page's header. The available categories have been described in the second chapter of this report, while the locations that appear under every category depend on the chosen profile (as noted in the previous chapter) as they are directly returned in the 'CreateUser' action's response. The first time the page is created and initialized two events are bound to it: the refresh functions for the entries and the user tokens trigger every time it is shown (on the 'pageshow' stage) and the refresh entries once again, attached to the click event on any tab of the 'Navigating Panel'. That is to say, every time the page is shown (be it from the result of navigating out and into it or just switching the active tab) the 'Location' and 'ProgressTrack' tables are queried and the information joint, resulting in the unlocked locations that can be seen under each category. These locations are presented in a 'ListView' container, with a neat format including the location name, a small picture and some information about it. The first element in each of these containers has an "entry" class name (just like every other location element), nonetheless the display style is set to "none" - essentially making it invisible. The use of this invisible element is as a template; it can easily be cloned and filled with data extracted from the database and appended to the list, while removing the style attached to it, avoiding unnecessary HTML code in the JavaScript methods. The last element is always the 'unlock' button, with a special class for a correct positioning inside this structured list. This button does what its name implies: if the user has enough tokens and there are any locations marked as locked inside the 'ProgressTrack' table for that category, the next location according to the profile order is unlocked and the token value is decreased in one unit. The next images show how this looks like in the app (Capture 9, 10, 11, 12, 13).

For the last tab, 'Recommended', there is no 'unlock' button as locations shown in this pseudo-category cannot be unlocked. In order to list them, a function crosses the resulting information from querying the 'Location' and 'ProgressTrack' tables, this time searching for locations marked as 'recommended'.



Capture 8

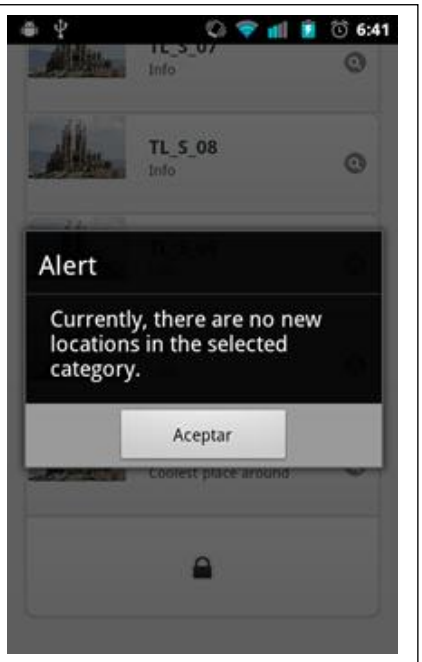


Capture 9

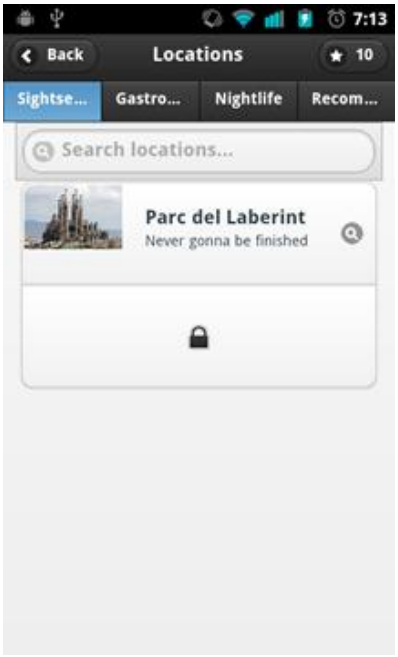
Capture 10

Finally, there is a useful widget right under the 'Navigating Panel' and over the head of the 'ListView': a 'Search Box'. When available locations start to grow in number and things get complicated, this search

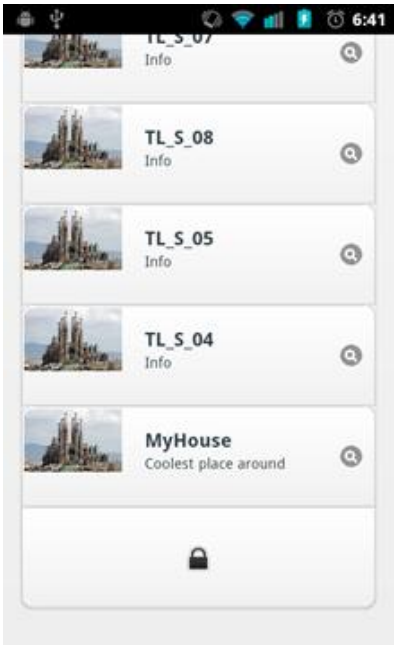
tool does a wonderful job at filtering the locations under every category (Capture 14).



Capture 11



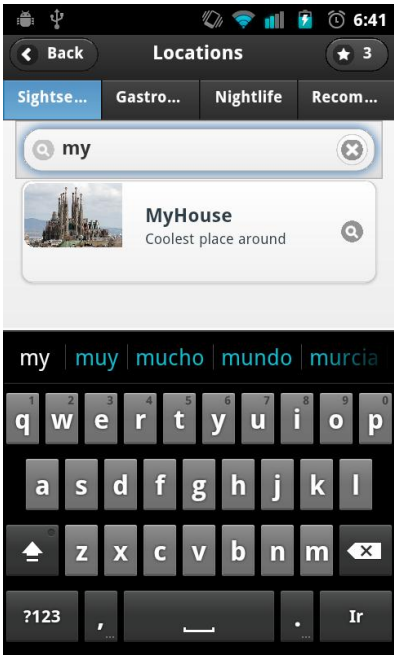
Capture 12



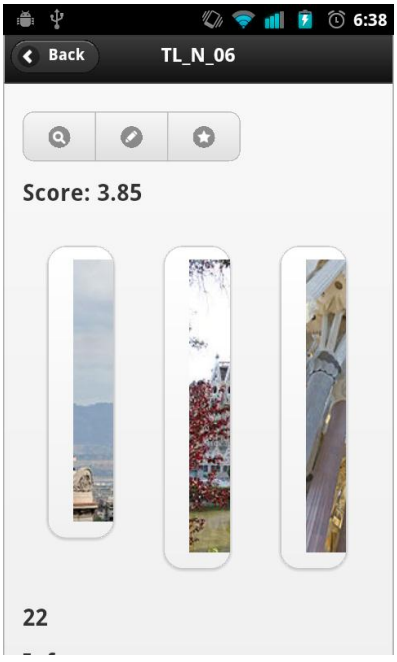
Capture 13

While this list of locations serves as an index, clicking on any of them will take the user to the 'Location Screen' – a new template page that is filled with all the available information from the database about the chosen location. All list elements with the class type “entry” are dynamically bound to click events for navigating to this new page, while the unlock button is bound to the execution of the check and unlock functions. The 'Location' page looks as shown in the next capture (Capture 15, 16).

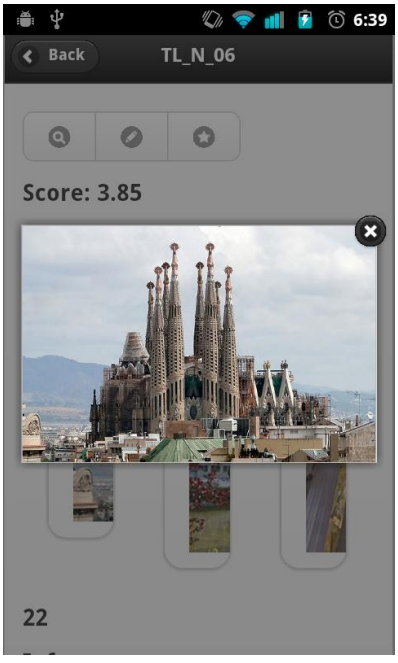
The header of this page is filled with the location's name. In the content area appears the global score,



Capture 14



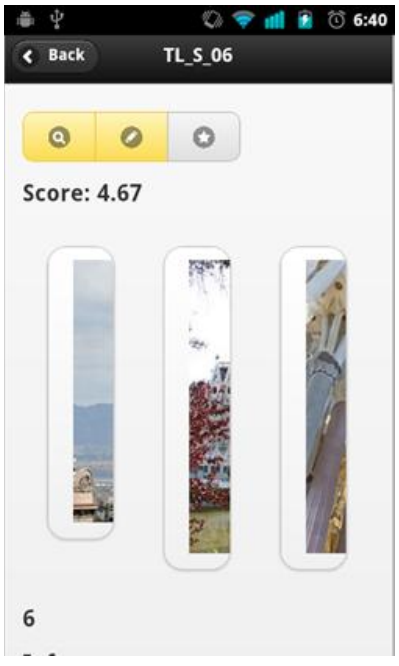
Capture 15



Capture 16

some pictures and, right under them, the all the available location information along with its identifier. The pictures use the pop-up and fill screen concept that has been introduced in chapter 2, however a small thumbnail should also be provided for at least 2-3 of the different screen sizes (something that has not been done yet in this project) for a more natural and unclipped appearance. At the bottom of the page, two buttons labeled 'Map' and 'Reviews' are sitting. The first one opens a dialog screen with a static map of the location's position, while the second shows a tuned sliding panel with the latest reviews available for the chosen location. On the top of the content area, a small item-set widget with 3 icons is shown. The icons represent, from left to right, the visited, rated/reviewed and recommended status of the opened location. When the user clicks on any of these a small informative pop-up is shown or a bigger dialog with rating options is placed hovering the screen (Capture 17, 18). The action depends on the actual status, for which a hint is given depending on the color of those icons: the white/grayish color indicates a negative or an unmet condition – it can be shown on any of the three icons; the blue color indicates that the rate/review option is available and can be accessed by clicking the icon – only applies for the rate/review action; the yellow/golden color indicates that a condition has been met or an action fulfilled – can be applied to all icons. Albeit the color of the icons is fixed at creation, the library allows for an easy dynamic modification should it be required. To change the color of one of these elements (Code.Snippet 20):

```
buttonR.removeClass('ui-btn-hover-c').removeClass('ui-btn-hover-
e').addClass('ui-btn-hover-b').removeClass('ui-btn-up-
c').removeClass('ui-btn-up-e').addClass('ui-btn-up-b').attr('data-
theme', 'b');
buttonR.attr(
                    {
                                "href": "#popupLocationRatedYes",
                                "data-position-to": "window"
                    });
```



Capture 17

Capture 18

Code.Snippet 20 - Style modification

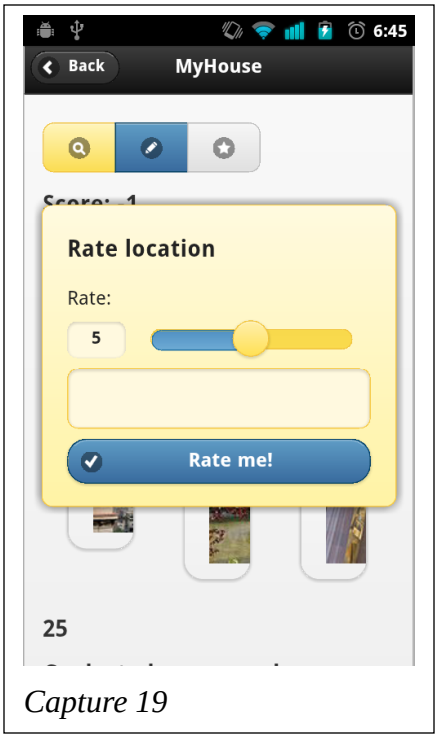
The pop-up balloon with the information or the rate/review dialog when it applies has been created beforehand for every possible scenario (Capture 19). The change is made by dynamically linking to one or other element when the requirements have been fulfilled in this way (Code.Snippet 21):

```
$('#popupLocationVisited p').html('<p>You have not visited this
location yet.</p>');
```

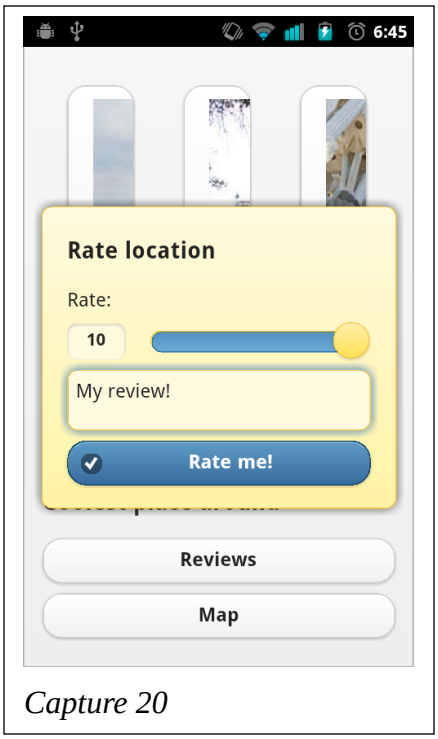
Code.Snippet 21 - Pop-up switching

The rate/review dialog (Capture 20) is a simple form with a slider to set the score and a text field for the user to input his review. The submit button triggers an event which attempts to send the rate through an XML HTTP Request, by accessing to a specifically dedicated URI at the server.

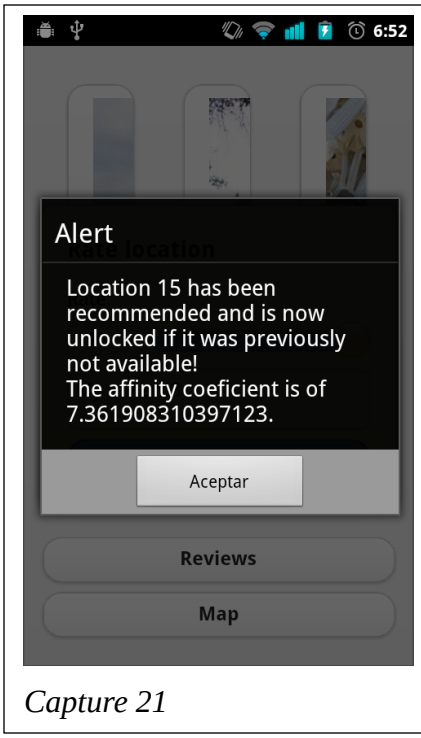
The last screens show the outcome of a positive recommendation (Capture 21, 22, 23).



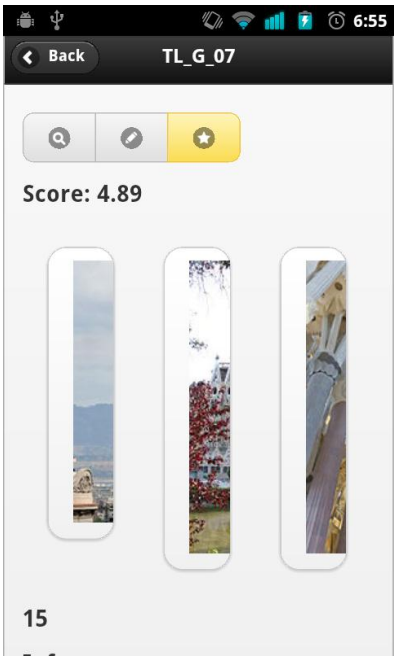
Capture 19



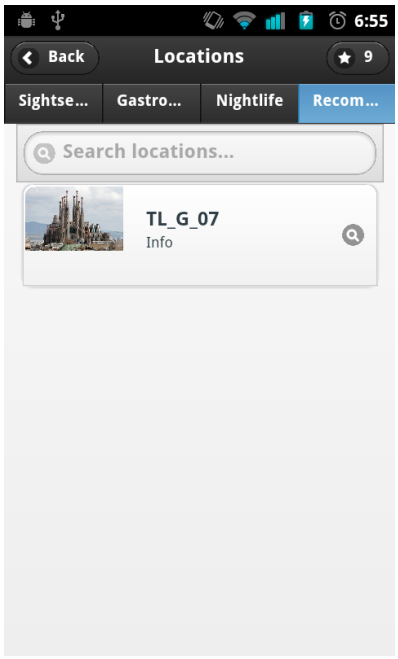
Capture 20



Capture 21



Capture 22



Capture 23

The lifecycle of the application reviewed to this point is resumed in the following diagram (Fig. 21).

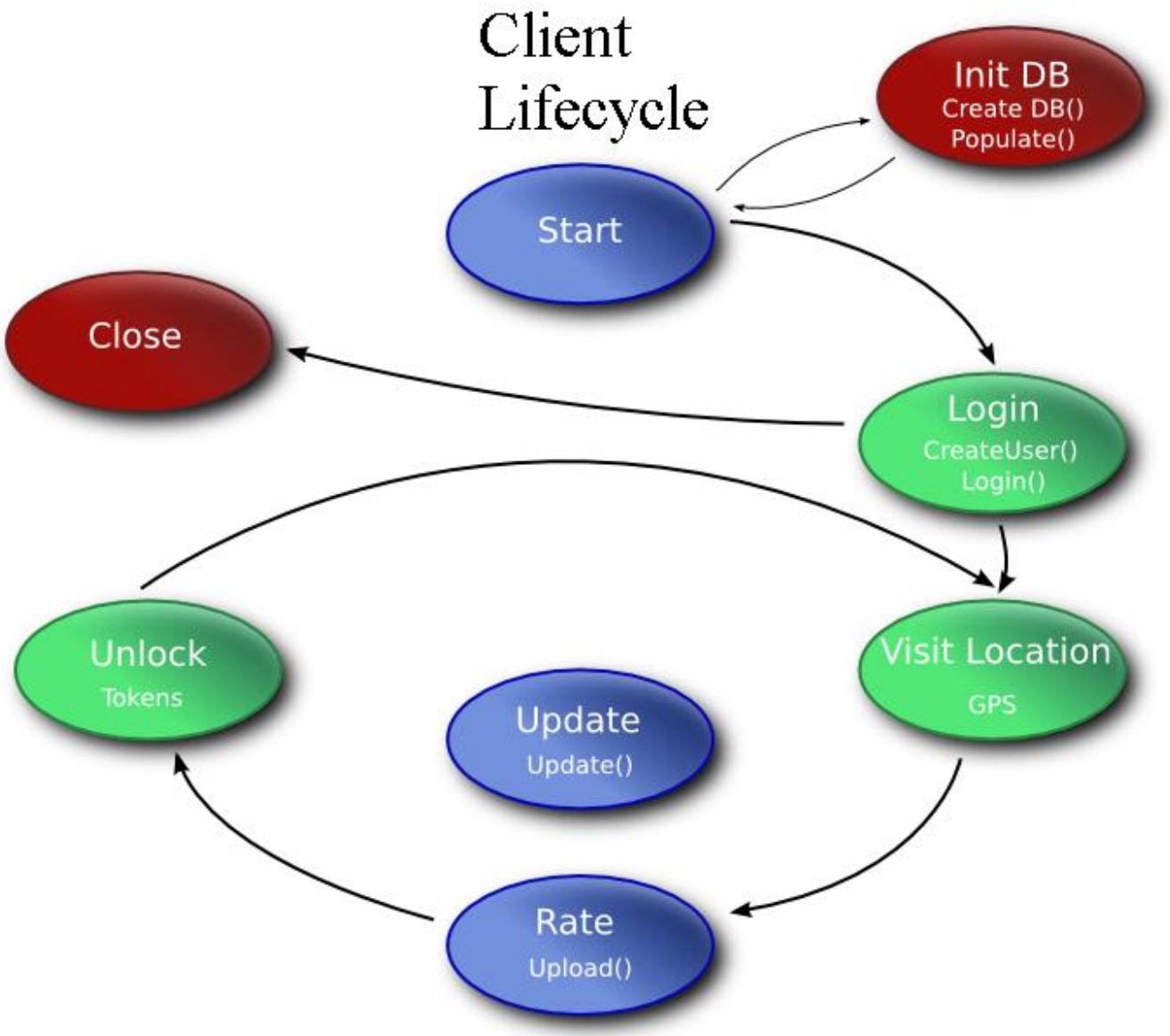


Figure 21- Client lifecycle

5.2.2 Background service

5.2.2.1 How a native plugin is made

With PhoneGap, if one wants to execute native code on any project a plugin has to be programmed. In order to create a plugin it is needed to create a JavaScript file with the definition of its methods (which serves as a pseudo-interface), the proper JS file with the usage of those methods and a native class containing the implementation of the former.

The JS definitions that serve as a bridge between the native and the 'WebView' programming are wrapped with a call to 'cordova.define()'. The first argument of the method is the identifier of the plugin (it can be a String too) and the second is the factory method that creates the plugin. The first thing that should be done inside this factory method is to use the 'require()' function to define a local variable called 'exec', pulling it from the 'cordova/exec' identifier. This is really important as the 'exec' method is the magic part of PhoneGap; it is the bit that handles the communication between the native and

JavaScript layer. Next, the declaration of the prototype methods takes place; every method should include a success and a failure callback mechanism, as they are directly passed as arguments to the main call inside it: the exec method. In addition, the exec function accept three more arguments: the name of the plugin (a String), the native name of the method to be executed (again a String) and an array of parameters for the native method. Finally, the last interesting bit of this code file is where the plugin class is created and exported with the following syntax (Code.Snippet 22):

```
function CreateBackgroundService(serviceName, require, exports,
module) {
    var exec = require("cordova/exec");

    var BackgroundService = function(serviceName) {
        var ServiceName = serviceName
        this.getServiceName = function() {
            return ServiceName;
        };
    };

    var BackgroundServiceError = function(code, message) {
        this.code = code || null;
        this.message = message || null;
    };
var backgroundService = new BackgroundService(serviceName);
    module.exports = backgroundService;
```

Code.Snippet 22 - Background service exportation

The JS file with the call to the recently defined methods has nothing particularly special in it; the usage just has to comply to the prepared functions and their arguments, so it will not be covered here. The files are attached to the Annex of this report for further details. The next important thing is the native class itself; this project will cover only the Android variant among all the possibilities. Android native programming is done in Java and, as such, the created classes can opt to all the language's features. When writing the class all that has to be assured is that it extends the 'org.apache.cordova.api.Plugin' class and that it implements the 'exec' method. Inside 'exec' the method that has been called has to be checked and then execute the proper native code to fulfill the request. The response is wrapped into a 'org.apache.cordova.api.PluginResult' object instance that also has to be imported in the file.

Finally, the PhoneGap application has to be aware of the created plugin, so it is required to include it in the configuration file. Inside 'res/xml/config.xml' the following line has to be added, containing the plugin's name and the value linked to it, which is the full name (including namespace). Additional modifications might be required, depending if the plugin uses some specific sensors or features of the operating system that would normally require permissions in a native app (such as connectivity or phone access).

5.2.2.2 What has been implemented

Now, it has been explained what has to be done in order to build and attach a plugin to a PhoneGap application. However the particular needs in the case of this project were somewhat more complex: the plugin should start a background service which should always be available, start after a system halt, awake periodically for some operations and awake the main app if it is needed. Since the purpose of this project is not to reinvent the wheel, the public Git repository of PhoneGap plugins was checked for a similar occurrence. A suitable (a bit outdated) plugin was found, able to start and stop a background service for Android operating systems and awake itself with a timer. Albeit the original author himself

says that “it is better described as a helper class or base code rather than a plugin” it has been of great help in the development of this project. With a bit of tuning and the addition of some extra features and lines of code, the desired requirements were fulfilled. The pack of files used to implement and attach this plugin is shown on the next diagram (Figure 22).

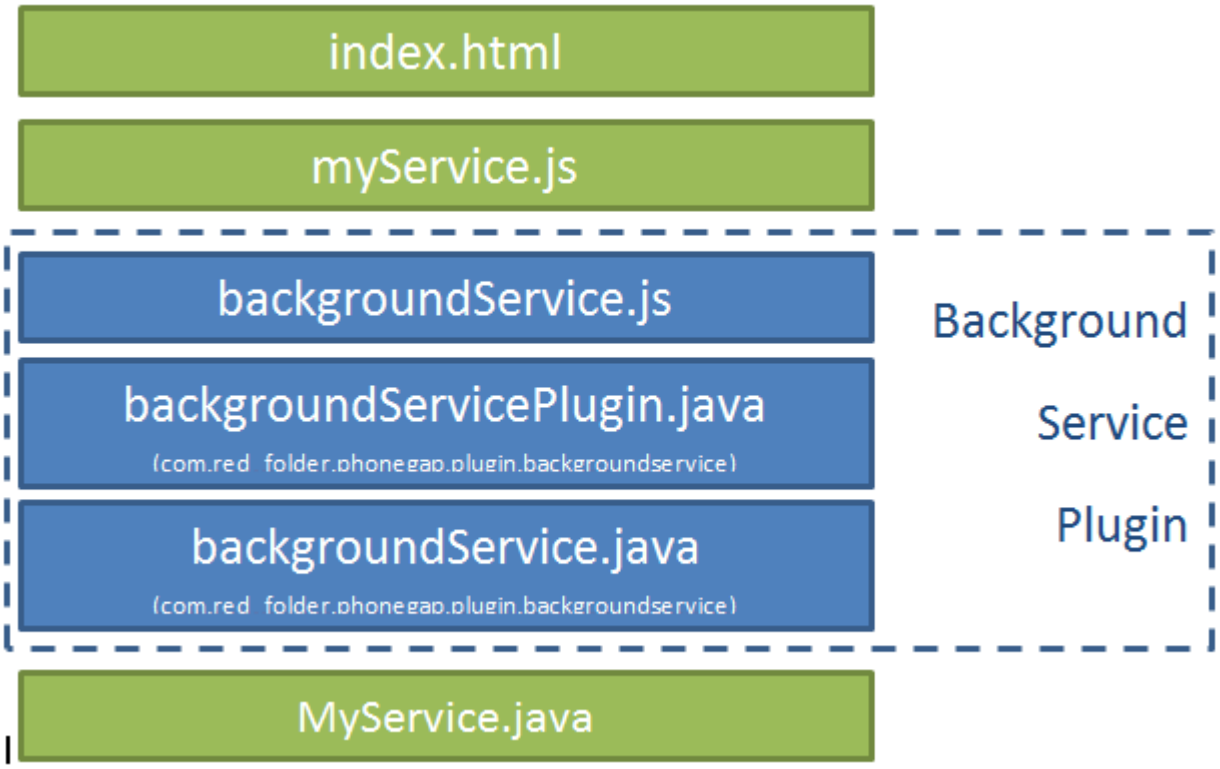


Figure 22 - Native plugin architecture

As noted in the previous section, a native class that extends 'Plugin' has to be created to execute the native code. However this has already be done; the segment of native code that implements the 'exec' class, checks the called method and contains the logic and code for generating the background service is contained in the 'backgroundserviceplugin.jar' library. This library already comes with enough functions to play with it: among other methods, there is a 'startProcess' and a 'stopProcess' call, methods to initialize and stop the timer and a useful 'setConfig' function through which parameters can be passed to the native part of the plugin. In order to use the library, the BackgroundService class (which in turns extends 'Plugin') has to be extended and six methods have to be overridden. Those are: 'doWork', 'get/setConfig', 'initialiseLatestResult' and 'onTimerEnabled/Disabled'.

- DoWork: is the method that gets called every time the timer executes. The method returns a JSON object which can be made available to the PhoneGap application through the plugin.
- Get/SetConfig: these functions can be used to share data and configuration between both layers of the application.
- OnTimerEnabled/Disabled: these methods allow attaching specific code for when the timer is started and stopped.
- InitialiseLatestResult: useful for passing information to the PhoneGap application, however it has not been used as the project's needs require something more elaborate.

Let's remember that the JS plugin has to be also defined, wrapped by a 'define' call in a separate file.

This is done in 'myService.js' and 'backgroundService.js' (the former containing the required wrapper and the latter a specific 'Service' plugin, to abstract the creation process). In addition, a file with all the JavaScript calls to the defined method has been created under the name of 'plugin.js' (also available at the Annex). With this frame and set up the plugin has been extended in the way described next.

Before going any further, the background service should be started using the 'startProcess' method and the timer should be set and enabled for the 'doWork' native method to be executed at the chosen time interval. The first stage requires the PhoneGap plugin to make available the location's GPS coordinates for it to be able to check them periodically in the background. The 'setConfig' method has been used for this purpose, as hinted by the plugin's original author – this method uses a JSON object as an argument. All the unlocked and not-visited locations from the 'Location' table have been queried and joint with the 'GPS_coordinate' table, the result was then parsed to JSON with the native 'stringify' implementation and set as the argument of the method.

On the native class handling this method, the JSON Object should be extracted and parsed, however it was not possible to do so in the way it was expected and documented in the Java API. Albeit the value of the referenced key could be extracted in a raw format by simply using the 'get()' method (the content being verified as valid JSON and identical to the one sent), any attempt to extract it as a JSONArray ended in an exception deep from the guts of the plugin. It was not really clear what caused the issue, because the class containing the logic always crashed when trying to decompile it and it was not possible to include a modified version in the project to print a different stack trace. Given this situation, a newly coded JSON parser seemed like the only solution. Once again, this fell out of the scope of the project, so a less elegant solution was programmed using the following regular expression to split and capture one or more digits, followed by a decimal dot, possibly followed or not by one or more digits (Code.Snippet 23).

```
Pattern p = Pattern.compile("[1-9]\\d*(\\.\\d+)?");
                Matcher m = p.matcher(str);
```

Code.Snippet 23 - Regular expression for parsing doubles

This proved completely functional and solved the parsing issue. The extracted location information (identifier, latitude and longitude) is inserted in a global array list and can now be accessed during the 'doWork' execution, every time the timer awakes it to run.

In essence, the 'doWork' method checks whether the current position of the user matches one of those in the array. Nevertheless, the positioning information is actually given with a certain error resulting in a circular area around a (best-guess) spot – this error varies depending on the used positioning system (GPS, mobile operator network) and can improve or get worse with external conditions such as weather, concrete, indoor locations, objects blocking the path or network availability. For this reason, to check whether the user's location matches one of the available sites for visiting, the application tries to find out if the user's position is inside a safety boundary around any of the locations. As it is explained in the next section, this approach also helps with conserving the battery life of the device.

In the case of a positive match, the system would like to alert the user and notify him about the newly detected event; this is the task of the 'Notification' class in Android. This class can create those notifications that appear on the Android notification bar, along with a personalized icon, a value (title) and a configurable message (text). Once attained the 'NOTIFICATION_SERVICE' context in Android, the resulting 'NotificationManager' class is able to push the crafted 'Notification' instance to the task bar (Capture 24, 25). In addition, it is possible to indicate the 'Intent' that should be executed or brought to the foreground should the user tap on the notification and, on top of that, it is possible to include extra

parameters that will be made available upon the execution. The LiveGuide client background service includes in this way the identifier of the newly visited location (Code.Snippet 24).

```
public void showNotification( String contentTitle, String contentText
) {
        int icon = R.drawable.icon;
        long when = System.currentTimeMillis();

        Notification notification = new Notification(icon,
contentTitle, when);

        notification.flags |= Notification.FLAG_AUTO_CANCEL;
        Intent notificationIntent = new Intent(this,
LiveGuideClient.class);
        notificationIntent.putExtra("locationId", contentText);

        PendingIntent contentIntent = PendingIntent.getActivity(this,
0, notificationIntent, 0);
        notification.setLatestEventInfo(this, contentTitle,
contentText, contentIntent);

        NotificationManager nm =
(NotificationManager)this.getSystemService(Context.NOTIFICATION_SERVIC
E);
        nm.notify(1, notification);
    }
```

Code.Snippet 24 - Notification procedure

Back on the on the main 'Activity' which is started by the 'Intent', and the 'WebView' responsible for the PhoneGap application. If, before the 'loadURL' method execution, one tries to retrieve the current 'Intent' instance, the previously included variables in the background service can be pulled out. Doing so, makes the previously set location identifier almost available to the PhoneGap app again, the only thing remaining to do is how to pass it to the 'WebView'. Luckily enough, among the Java features of Android programming, there is a mechanism to inject JavaScript code inside a 'WebView'. The injection of such code happens with the use of the following method (Code.Snippet 25):

```
Intent intent = this.getIntent();
        if(intent.getStringExtra("locationId") != null &&
intent.getStringExtra("locationId") != "") {
            String locationId = intent.getStringExtra("locationId");
            this.visitedLocation = Integer.parseInt(locationId);
        }
@Override
    public Object onMessage(String id, Object obj) {
        if (id.equals("onPageStarted")) {
            this.sendJavascript("javascript:test=" +
this.visitedLocation + ";");
        }

        return super.onMessage(id, obj);
    }
```

Code.Snippet 25 - Retrieval and JS injection of parameters in WebView

Nonetheless, even if not mentioned specifically in any official documentation, it cannot be used freely anywhere. Some users ($REF) have discovered that in order to successfully inject code into the PhoneGap application, the action has to be done only after the first PhoneGap internal message has triggered the parent class' 'onMessage' method. Attempts to inject code before that seem to be ignored by the application. What is done in this project is override this method and check for a message with identifier 'onPageStarted' (as suggested in the forums) before manipulating the value of an inner global variable; after that, the parent's code is executed. That closes the full cycle of communication PhoneGap – Native – Notification – Activity – PhoneGap and concludes with the lifecycle of the software (Capture 26). The next figure tries to synthetize everything mentioned for the interaction between them (Fig. 23).

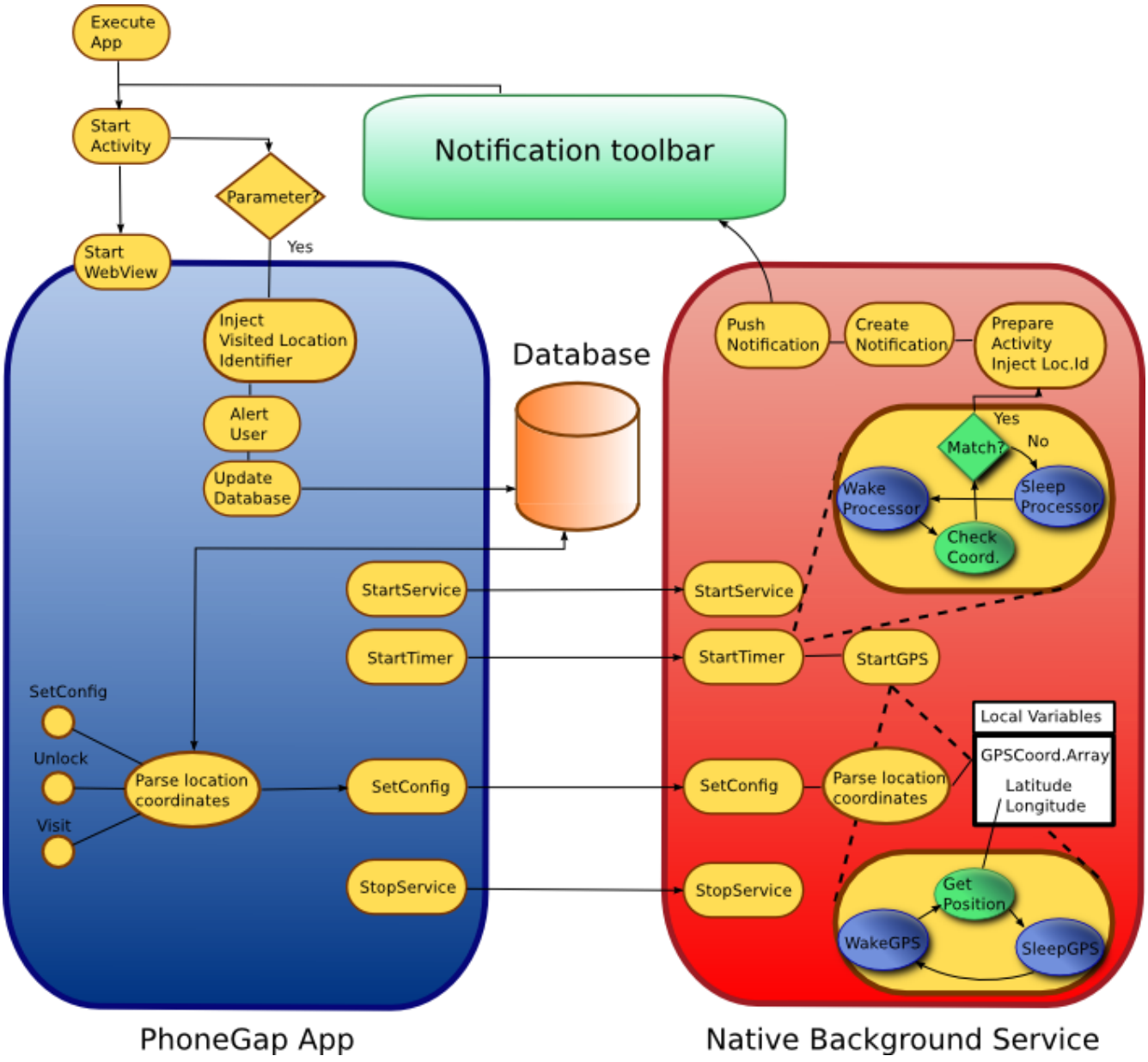


Figure 233 - PhoneGap - Native plugin - Notification interaction

To conclude, the JAR file with the background service plugin library has to be added as a resource to the project and the plugin included in the 'res/xml/config.xml' file (Code.Snippet 26).

```
<cordova>
    …
      <plugins>
          <plugin name="BackgroundServicePlugin"
      value="com.red_folder.phonegap.plugin.backgroundservice.Backgrou
      ndServicePlugin"/>
      </plugins>
</cordova>
```

Code.Snippet 26 - Plugin resource addition

Inside 'AndroidManifest.xml' permissions have to be set to allow the service to restart on device boot (Code.Snippet 27).

```
<uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
      <receiver
android:name="com.red_folder.phonegap.plugin.backgroundservice.BootRec
eiver">
            <intent-filter>
                <action
android:name="android.intent.action.BOOT_COMPLETED">
                </action>
            </intent-filter>
      </receiver>
```

Code.Snippet 27 - Service boot start permission

And the service declared in that same file (Code.Snippet 28).

```
<service android:name="edu.mvm.liveguide.MyService">
            <intent-filter>
                <action android:name="edu.mvm.liveguide.MyService"/>
            </intent-filter>
</service>
```

Code.Snippet 28 - Service declaration

5.2.3 Geolocation

For many systems, like LiveGuide client, it is of great usefulness to find out the user's position on the map; this is called geolocation. A position is said to be established when both coordinates, latitude and longitude, are known. In Android, this can happen through two different location providers: the Global Positioning Satellites (GPS) or the mobile provider's cell network (with the help of data connectivity), both having advantages and disadvantages around power consumptions, accuracy, Time to First Fix (TTFF)...

- GPS
    - Power consumption: very high. It uses hardware directly.
    - Accuracy range: very precise. Official documentation states an accuracy of less than 100 meters. Empirical tests show that in most cases this estimation is very conservative as it is quite common to get an accuracy of around 2 to 20 meters.

    - TTFF: very high. Can go up to several minutes, since at least 6-7 satellites are needed to consider a device synchronized.
    - Reliability: Average. Does not work inside buildings or around tall structures which could block the required line of sight. Not affected by weather (clouds, rain, snow...).

- Network Location Provider
    - Power consumption: very low. The position is calculated checking the surrounding cells and access points against a database were the Google “magic” happens.
    - Accuracy range: Theoretically is between 100 and 500 meters (when near Wi-Fi spots) and over 500 meters only on cell support. Real-world tests have shown that most of the retrieved positions' error range in the area of 50 meters.
    - TTFF: low. The data is compared and a position is returned almost instantly.
    - Reliability: Average. Most of the time it works indoors; however the required Internet connectivity has to be included at this point.

Now, several factors have to be taken into account for deciding which location provider to use. On one hand the GPS accuracy is much higher, but the time required for it to work, combined with the high power consumption would definitely be a handicap if: the GPS has to be turned on and waited several minutes to return a position; it is left activated, as it would be needed active constantly. On the other hand, albeit with a lower accuracy, the network location provider positioning seems reliable... if not for the connectivity need which, the author expects, will not be freely available for a great part of the tourists using the app (especially those coming from abroad).

One way in which this could be done is to keep the GPS deactivated and ask the user to turn it on when he starts going around the city. That would still drain a considerable amount of battery, but less than having it turned the whole day or specifically ask the user to activate it and wait when on spot of one of the locations. In fact, this would be the same as the initial option of avoiding the need of a background service plugin in first place. For the demonstrator however, and since we will not have any troubles with finding a Wi-Fi spot, the network location provider has been used for a very simple reason: the demonstrator is going to take place inside a building, cutting all access to GPS satellites.

To be able to use a location provider's services, a 'LocationManager' has to be retrieved from the application's context, the provider has to be chosen and the time period and 'LocationListener' have to be set. The provider will be invoked approximately at every defined period (most of the time it takes a bit longer) and it will check for a change in the user position (Code.Snippet 29).

```
locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);
        provider =
locationManager.getProvider(LocationManager.NETWORK_PROVIDER);

this.locationManager.requestLocationUpdates(LocationManager.NETWORK_PR
OVIDER,
                30000,          // 30-second interval.
                0,             // 0 meters.
                this.listener);
```

Code.Snippet 29 - Location provider invocation

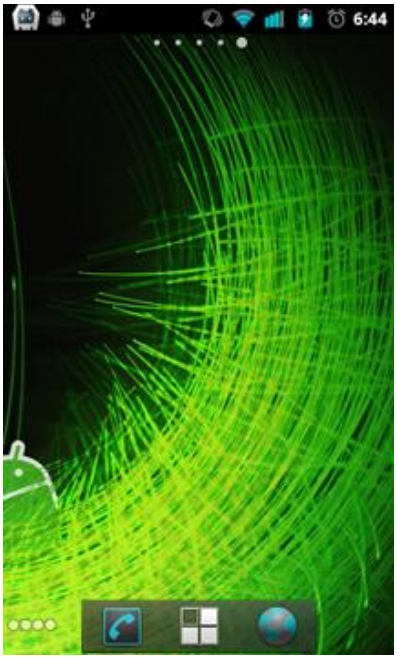
The 'LocationListener' can be defined inside the same class, as only some of its methods have to be

overridden. Its task is to detect changes involving the positioning, the provider or its status. For LiveGuide client only the 'onLocationChanged' method has been implemented and, in essence, all it does is capture the new latitude and longitude values and copy them to the globally defined variables (Code.Snippet 30).

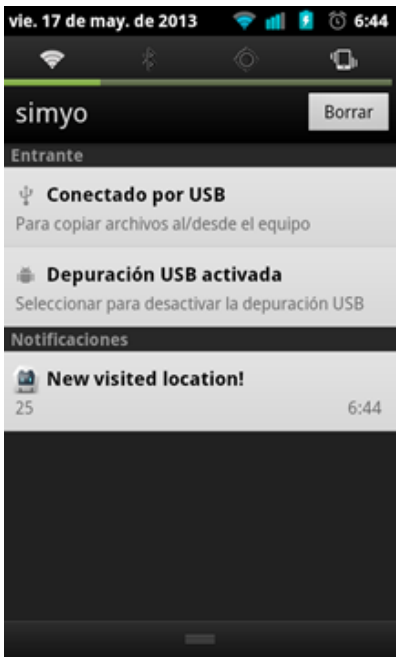
```
private final LocationListener listener = new LocationListener() {
        @Override
        public void onLocationChanged(android.location.Location
location) {
            latitude = location.getLatitude();
            longitude = location.getLongitude();
        }
};
```

Code.Snippet 30 - Location listener method

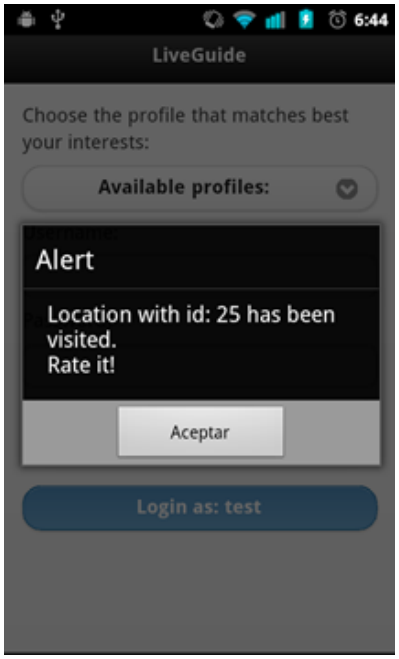
That is all the needed set up and configuration for being able to use the positioning of a location provider.



Capture 24



Capture 252



Capture 26

5.3 Client interaction

The interaction with the client occurs at two different sides: the client with the SQLite database and the transmissions with the server through XHR and JSON. Requests and responses have already been thoroughly detailed in the server's chapter and little emphasis is going to be given here, besides displaying how are filled some of the fields for the request and what logic is followed when receiving a response.

5.3.1 Actions

Although the application has enough embedded information to be useful as standalone software, the true potential is unleashed when combined with the server. Besides for creating the initial account, progressing in the mobile terminal means that a user can unlock new locations and be up to date with the latest data available on the server side. Next, a review of the available actions is stated.

- CreateUser
  This action is executed from the home screen and is allowed only when the user has filled both, the username and password fields, and chosen a profile type from the dropdown select menu. An XML Http Request is formed and sent to the server's hardcoded URI path, where the data input by the user is evaluated. If the specified username is unique, a positive response is sent back; otherwise, an error code is returned. In this last situation, the client detects the message's content and avoids any further screen progress, informing the user that the chosen username is not unique. On the other hand, a positive response includes a full 'User' table entry (just as created on the server side) which the client stores locally and three arrays of ordered location identifiers (one for every location category) which are stored in the 'ProgressTrack' table, conserving the order.
- Update
  The update can be triggered by the user whenever he is on the main screen. An XHR message containing the user identifier and the last update date for it are sent to the server. The response can include zero or more concatenated arrays of objects; the possible array element types are 'Location', 'Score', 'GPS_Coordinate' and, in the near future, 'Review' and 'Top' – as long as in any of these tables there are new or modified entries comparing to the last time the requesting user has updated. Back on the client side: since this is a JSON concatenation of arrays of different elements, a group of key-value pairs has been attached at the beginning, informing the number of elements contained for update for each of the stated tables. The arrays are run through and the elements are modified or deleted, depending on the received data.
- Upload
  Whenever a client submits a rate and a review for any of the visited locations, an XHR message containing the whole 'Rate' entry is sent to the server, along one last boolean variable that indicates if the client applies for a recommendation. If the rate is unique, it is stored in the database. The response to the client has already been explained in this same section of the server's chapter. To resume, a successful recommendation process returns the identifier of the recommended location and the estimated score the server has calculated the client would give it. If the location identifier is negative, the rate was already submitted, the user does not apply for a recommendation or it could not be found. In the first case a pop-up would inform the user of the affinity and the unlocked location, while in the second case nothing really happens, besides the marking of the location as rated if it was not so before.

6 Results and future work

After reviewing the last details of the system it is time to test it and share the results here. This chapter is divided in two sections: first, the goals are revisited; after that, the chapter concludes explaining some of the future actions that should be taken. The first section has three subsections dedicated to explain goals that need time and field work to be achieved, the usability and the performance of the application itself. The second section addresses the port to native and the future work to be done.

6.1 Achieved goals

Once having developed the whole system it is time to look back and evaluate the results against the initially proposed goals and requirements. Questions that immediately arise such as "have the goals been fulfilled?" or "is the end application as useful as the expected?" are addressed here. They do not really have a single answer; some concepts have been achieved, while for others changes had to be made to get the system to work at all.

- ✓ Portable touristic guide.
  The resulting application has been designed for and tested on mobile platforms. It is free of cost for the users and its information structure has been based on a touristic guide format.
- Compatible with most mobile operating systems.
  One of the major imposed goals' results, the multi-platform availability, has not been so bright. Even if a great part of the mobile application coding for different platforms has been avoided by using the PhoneGap framework, the fact that a background service (a mechanism only available in native coding) is needed for it to completely work hinder the completeness of this objective. Thus the assumption: PhoneGap works; it is useful and can be utilized for such goal, however not for every imaginable scenario.
- ✓ Dynamic and updated information.
  The available location information can be easily managed from within the central server. Locations and their coordinates can be added, modified or removed in order for them to be up to date and reflect the real world information.
- ✓ Gaming-like features.
  This project is not exactly a game. Nonetheless, there is such *déjà vu* after having used it, as it requires the users to move to specific places and execute a number of actions for them to access the full features of the app.
- ✓ Geolocation for finding out user position.
  Another achieved goal is the implementation of the geolocation system along the application. The coordinate providers are successfully started and updated at timed intervals; this information is afterwards stored for usage in the next step.

- ✓ Progress is based on user actions:
    - Visiting a location requires to be physically present at it.
      In order to consider any of the available locations as visited, the user has to move next to its coordinates. They are compared against those stored from the position provider and the location is checked in case of a match.
    - Users are asked to rate and review a visited location.
      This is the main role of the users inside the app, visited locations have to be rated and reviewed and the information sent to the server.
    - The above process gives 'tokens' for unlocking more locations.
      Progressing in the application follows this game-like constraint; after the user visits and rates a location he receives tokens which allow him to unlock more locations.
- ✓ Location scores are based on user marks.
  Achieved. The scores of the locations are solely dependent on the quantity and mark of the submitted rates by the user base.
- ✓ Personalized experience:
    - Recommendation system offers recommended locations at some times, produced by correlating rate data against other users and calculating their affinity.
      Also in this achieved category falls the great recommendation system, which is capable of successfully correlating the data of hundreds of users and provide to them personalized locations.
    - Different user profiles (affect unlock order using specific profile sub-segment scores).
      User profiles have been implemented, affecting the order in which locations are unlocked in every category. Location scores by user profile also made it within the system.
    - User can choose the location category from which he wants to unlock more locations.
      As it happens with non-linear games, in the client software users can choose which path to follow and in what order, by deciding in which category to spend their tokens and unlock new locations.
- ✓ Location and score information on mobile devices can be synchronized against the server.
  If there is connectivity, local databases can request an update to the server. This has been achieved using pseudo-synchronization, by using a last modification date field in the tables. That is to say, only the minimum required entries are transferred and bandwidth is saved.
- ✓ A minimum level of security to avoid user impersonation and ciphered communication.
  User accounts in the system allow making a distinction between users when any of them submits an action. In addition, the configuration in the Servlet requests the communication to be confidential – ciphering the data channel between client and server.
- Get to substitute traditional paper guides (and digital where possible).
- Offer less known places, often only known by locals and difficult to find elsewhere.
- Usability aspects: smooth use, attractive design, minimum connectivity, GPS availability.
- Performance: small memory footprint, low battery drain.

These last four points are going to be reviewed in their own sections.

6.1.1 Replacement and enhancement of traditional guides

The first two objectives out of the last pack of four cannot be really evaluated at this point, whether it is because further coding is needed to deliver them or because they require some field work and an adequate marketing outline, are commented following. Getting to substitute paper or electronic traditional guides is a sensible task. First of all, LiveGuide could never completely substitute one of them as it is focused in a slightly different area of application and it also targets a specific segment of the user base. The content in it has to be unlocked gradually and for this reason it will never equal in quantity those of the traditional guide (without considering this as a good or bad thing at this point, though the author made clear his point of view in the second chapter). The candidate user, on the other hand, is someone who is ready to experience a new way of sightseeing and tourism, a more dynamic person for whom it will not be the first contact with a game – of course the app is not restricted to any age, but the expected generic user is sub 30 years – facts that will modify the user base when compared to traditional guides. Finally, it will definitely require a lot of field work to gather a significantly big and heterogeneous location database, keeping in mind the goal of offering slightly different places for visiting, less known, recommended by locals and stride away from the typical tourist sites.

Before closing this section, it should be noted that extra functionality has been planned for them system, in order to make it more appealing to potential users and offer additional control information for the managers.

6.1.2 Usability

Usability is a very objective topic as specific guidelines exist for its evaluation, such as Web Accessibility Initiative (WAI) and Responsive Web Design (RWD). On one hand, the WAI rules have not been followed when designing and developing the app, so they are not going to be used for evaluation. The RWD, on the other side, has been directly implemented within the jQuery Mobile framework. There are also independent usability insights, as for example those resulting in the perception a user has of the general quality of the application, comparing it to other known software he has used. However, these results drastically change over time, as new hardware and technologies emerge constantly making obsolete the former ones, and are also heavily dependent on the user, thus, not very practical for this evaluation. Nevertheless, let's try to do some objective comparison for the results in the actual technological market situation.

- **UI**

The user interface is one of the most important topics that should be addressed, it is the eyes of the user inside the system and a direct metric for evaluation depending on how it is perceived. PhoneGap has allowed to literally transform the application in a web page and programming with HTML, CSS and JavaScript, while including jQuery and its mobile counterpart, have made the rest of the user interface and its interaction. On one hand it can be said that the resulting UI is not basic and has been adorned with graphics, stripped-down to minimum, but at least intuitive. It could be easily further enhanced to reach a higher status, however that is not the problem with it. The biggest issue is that jQuery Mobile's widgets should adapt to different device and platforms; nonetheless, some situations have been detected where this clearly is not the case, mainly because the Responsive Web Design paradigm has not been followed to an end (even if the developers have stated so and this can be proven true for 99% of the library), producing some ugly and strange visual behaviors that should have been avoided. On a more general note, the overall performance of a 'web-browser application' cannot be compared to those written natively. On some points the hogs are noticeable (not impeding usability), the event reaction time is also a bit slower and, sometimes, flashes with 'artifacts' appear for instants during page navigation. Although the event reaction time issue is also present in native apps, simply because it

works in this way for touch devices (a minimum time is needed for processing different touch types and distinguish between them), the processing simply takes longer when it has to traverse several layers.The main mobile device where the app has been tested is 3 years old; on newer devices these issues are almost imperceptible, but still, the extra abstraction layer exist. Also, JavaScript and jQuery Mobile usually offer a poor speed and performance, given the environments where they are generally used do not require it. Still, some improvements can be achieved if most of the transitions are deactivated and page caching and pre-loading is activated. To conclude this section, it is also responsibility of the author to apply a bit of RWD and provide detection for different screen sizes, along with the resized versions of the photos for the locations; otherwise, the result can be very disturbing as shown in the demo.

- **Connectivity need**

A topic worth addressing is the fact that Internet connectivity is required during certain points for the correct working of the application. This truly can be an annoyance for most of the users, as not every city visitor will arrive with a paid data plan and free Wi-Fi hot spots are not as abundant as in other cities. This was the reason that initially the project was designed towards working without connectivity, however it soon was clear that such approach could not really happen. During the following events in the current demo, connectivity is required – still, kept to a minimum.

- Download of the application.
- User creation. Technically this probably happens during the same interval as the above.
- Local database update.
- Upload of a rate and a review.

- **GPS precision**

As it has already been commented during chapter five, the precision of the positioning depends on the system that has been chosen for that purpose. Hopefully, both alternatives offer enough accuracy to make the application behave like it should. Still, the workaround of checking within a boundary of a block the size of the 'Sagrada Familia' has been done in order to allow the positioning mechanism to shutdown itself and save battery life, albeit introducing a small probability of failing to detect or having a false positive.

6.1.3 Performance

The performance can greatly vary between different devices and available hardware components. Thus, only the next reduced set of points is evaluated.

- **Application size**

The space that the application is going to need in the storage device of the mobile terminal is important indeed. While many of them have expansion storage slots, not every device has such feature and, with the constant arrival of new useful applications in this growing market, a user can quickly run out of space on its terminal. The app's size is around 3MB, although it can be reduced to 2,5MB if the compressed versions of the libraries are used and even possibly to 2MB straight if stripped down to only the essential files. The contents of the database are the other important factor, calculations prove that approximately every 40 locations information (along with their position coordinates) require an additional megabyte of storage. Pictures have not been taken into account for the equation as their quality and size can make a significant difference in the final size. At a rate of 3 images per location, that could mean an additional megabyte for every two locations, unless a different approach is taken

(for example, storing the images online, however that would require permanent connectivity).

- **Battery drain**

LiveGuide incurs in a significant change in battery usage at three points: whenever the screen is lit (this applies for every application out there, as it is an electronic device characteristic), when communicating over the network and when using the positioning system. Communication has been held down to a minimum and there is little to do for the screen usage. For the positioning however, the workaround explained during the fifth chapter (background plugin section) has significantly reduced the battery drain.

6.2 Future work

After the processes of development and evaluation, it is time to define the future path this project should take. Next, the immediate work to be done is detailed, along the addition of some features and improvements that could make LiveGuide truly stand out against similar applications.

6.2.1 Port to native

The first thing that was understood is that applications such the one presented here should be done natively for every platform, even if that means programming it multiple times and doing so in several languages. There is nothing worse than an application failing not because it is not useful or does not work as expected, but because the perceived quality and smoothness for the interface and the user interactions has failed to be delivered. Certainly, it is possible to do interesting software with PhoneGap and, to a great extent, the additional libraries have helped in doing so, but as a general word of advice, those applications should be kept simple – this is not the way sophisticated software should be done. In addition, some months a well-known app for a well-known company (Facebook) decided to cease its attempts to deliver a single multi-platform product and started to develop specific version for each and every operating system, commenting on the matter that their former approach had been wrong and that Responsive Web Design is truly the future paradigm for this kind of software. For these two reasons and the ones stated before (event response time, flow pace of the app, smooth navigation...), the first thing that should be done if LiveGuide really wants to get to the clients and offer an appealing alternative, is to rewrite the app for native platforms, starting by Android OS and iOS. This should not be understood in the wrong way: the application as a utility is useful, however a sluggish user interface can easily drive users away to similar apps or make them believe that the switch from traditional guides is not worth it.

6.2.2 Finishing touches

Besides the port to native, some features were planned for the application that really did not make it on time for this demonstrator. However, that does not make them useless at all; here are some of the planned features and some new that were detected during the development – be aware that some of them are not compatible.

- ***Top visited***

It would be useful to populate the respective page on the mobile application with some lists containing the top locations. The 5 locations with the highest score could be provided (be it global score or segmented by user profiles) and also the same could be done with the most visited places.

- ***Server statistics***

Server statistics can serve the application developer(s) and manager(s) useful information about the user base. Number of active users, average locations visited per user, profile bias, average time spent visiting the city... are all statistics that can be used in a way or another to help the system grow and improve. For example, with the average visit time smaller location lists could be given to every user, reducing storage size; the mean amount of visited locations per user could be used to motivate clients to keep using the app.

- ***Rankings***

In the social era of the Internet, rankings can always be fun, appropriate and worth the time. An effective way of increasing the usage is to provide the clients a way to compare themselves, effectively promoting competition.

- ***Monetize LiveGuide***

It was never a primary goal to monetize the app, but a passive income for delivering a service and fulfilling a social need should never be passed by. The application will always remain free, but open to location advertisements in the following way. The owner of a local restaurant would like to promote his business through marketing, by getting to a large user base. With a small fee, and always checking the quality of the location, the place could be added to the system's database. With a bigger fee, it could also get promoted and appear in the first 2 locations any user gets when starting to use the app.

- ***Social contribution***

To truly deliver a complete 2.0 experience, rates and review can sometimes not be enough. A form on the web page, attached to an additional action on the server, could be used by the clients to inform about new locations that are not yet in the database and ask for their inclusion.

Moreover, the structure of the database is simple and well documented – an even bigger asset would be to allow any person in the world to create a local database for his own city and offer it online for usage with the LiveGuide system, effectively helping to promote his city and its locations. The actual location table already has a field to distinguish between cities by using the concatenation of the country's and the local area's phone prefix!

- ***Even more user personalization***

Right now a couple of things would be really desirable by any client and, with a little time and effort they could be implemented within the system. First, users should be able to mark any recommended location as unwanted, which would inform the server and avoid it for future recommendations. The second one involves the usability directly, there has not been found an easy way to mark the locations as visited on the main list the client sees. There *should* be a way, but it would involve modifying the CSS more heavily than what has already been tried. An easier solution would be to remove the visited locations from that list and create a new page containing only those places the user has already rated and reviewed.

7 Conclusion

When programming software and applications there is no single way set in stone to do it. Every project is different and its grade of complexity and sophistication can deliver original challenges during its design and developing processes. LiveGuide has been no different. It all started as a small idea to create a new prototype of tourist guide, mixing it with gaming experience and geolocation, and quickly grew with several other options and personalization features.

From the technical point of view, the imposed end results have been achieved and there is a fully functional prototype for testing purposes. The system has been divided into server and client; the server being able to deliver service for multiple client instances, while each of those elements has their own local storage device in the form of a database. Clients have been done with a framework that allows them to be ported without further coding in at least 90% of their functionality, while the resting of it had to be done natively, coding and attaching a plugin to the framework. Geolocation has been included as planned and the accuracy of both positioning providers has proven enough for the application's needs. The client has been predominantly written with web development languages and libraries (HTML, CSS, JS, jQuery, jQueryMobile, html5sql) due to the usage of the framework, thus, communication between the parts occur with XMLHttpRequests over an already established connection. The server, on the other hand, is a Servlet whose structure follows the MVC paradigm, while benefiting from the close and powerful relation which exists between servlets and Glassfish application server for interaction and security purposes. Lastly, a stable and trustful recommendation system has been attained which is able to correlate user information and calculate affinity between clients; on top of that, personalization also exist at the profile level, allowing for a unique experience for every user.

From the personal point of view, as well as that of the theoretical users, there are two main factors that have left the author unsatisfied. The first and most important is the “failure” to deliver a fully multi-platform application. That has been demonstrated as impossible with the current technologies and, even if the main part of the application can be spared of porting, it kind of “defeats” the purpose of using such framework in the first place. While this on its own is not something dramatically game-changing for a user, it comes along with a user interface which slightly underperforms in most cases, delivering a poor performance under certain circumstances, even when implementing the Responsive Web Design paradigm. Basically, the benefits of the adopted design have not surpassed the introduced shortcomings.

All in all, this project has been a very enjoyable experience and the end result should in no way be underestimated. Whether achieving the established goals ends up being a success or a failure, the generated knowledge in doing so is what matters most. There will always be margin for improvement and, given some of the personal reasons that drove the design and development of this project, sometimes it could mean to try a completely different approach. The fully functional prototype

delivered by this project offers a nice starting point; with some time and enhancements it can become the prized eye-catching, smooth and fluid application that it needs to, in order to be considered as a true alternative for today's city-guides.

8 Bibliography

A. Oracle Technology Network (Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Devika Gollapudi, Kim Haase, William Markito, Chinmayee Srivathsa). The Java EE 6 Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/ January 2013.
This is the official JavaEE manual containing, documentation and examples, made by the Oracle corporation staff.

B. Stark, Jonathan. Building Android Apps with HTML, CSS, and JavaScript. O'Reilly media, September 2010.
A book in the Open Feedback Publishing System (OFPS) for developing Android applications using web-browser technologies and PhoneGap.

C. http://simonmacdonald.blogspot.com/. MacDonald, Simon (PhoneGap contributor). 2010 – 2013.
Blog of one of the main contributors of the PhoneGap framework. Contains useful guides, solution to known issues and examples.

D. http://red-folder.blogspot.com/. Taylor, Mark (PhoneGap contributor). 2012 – 2013.
Blog of the original author of the Background Service Plugin used and extended here.

E. http://stackoverflow.com/. Various contributors. Various topics. 2011 – 2013.
Online, free, collaboratively edited question and answer site for programmers.

F. http://www.codecademy.com/. Learn JavaScript, HTML, CSS. 2012.
Online, free, interactive platform offering programming courses in various languages.

9 References

[1]. http://www.oracle.com/technetwork/java/javaee/overview/index.html

[2]. https://netbeans.org/

[3]. https://glassfish.java.net/

[4]. http://phonegap.com/

[5]. http://www.w3.org/TR/html51/

[6]. http://www.w3.org/TR/CSS/

[7]. https://developer.mozilla.org/en-US/docs/JavaScript

[8]. http://www.w3.org/TR/XMLHttpRequest/

[9]. http://tools.ietf.org/html/rfc6455

[10]. http://www.android.com/

[11]. http://jquerymobile.com/

[12]. https://developer.mozilla.org/en/docs/AJAX

[13]. http://html5sql.com/

[14]. http://www.w3.org/TR/webdatabase/

[15]. http://www.mysql.com/

[16]. http://www.sqlite.org/

[17]. http://dev.mysql.com/doc/refman/5.1/en/mysqldump.html

[18]. https://gist.github.com/esperlu/943776

[19]. http://www.json.org/

[20]. http://code.google.com/p/google-gson/

[21]. http://www.oracle.com/technetwork/middleware/toplink/overview/index.html

[22]. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[23]. http://en.wikipedia.org/wiki/Cosine_similarity