

ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

**SCUOLA DI INGEGNERIA E
ARCHITETTURA**

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea in Reti di Calcolatori

**CROWDSENSING AND PROXIMITY SERVICES FOR
IMPAIRED MOBILITY**

Candidato:

Cortellazzi Jacopo

Relatore:

Chiar.mo Prof. Ing. Corradi Antonio

Correlatori:

Prof. Ing. Foschini Luca

Dott. Ing. Ianniello Raffaele

ANNO ACCADEMICO 2014/2015

Sessione III

INTRODUCTION	5
1. SMART CITIES SCENARIO	7
1.1 CROWDSENSING	8
1.2 SMART CITIES	9
1.2 ACTUAL SUPPORTING SYSTEMS	12
1.3 NOWADAYS NECESSITIES.....	13
2. TECHNOLOGIES BACKGROUND	16
2.1 ANDROID	17
2.1.1 ARCHITECTURE.....	19
2.1.2 APPLICATION LIFECYCLE.....	21
2.1.3 ACTIVITIES AND FRAGMENTS	23
2.1.4 INTENT	27
2.1.5 SERVICES.....	29
2.1.6 ASYNCTASK AND LOADERS	31
2.2 SPRING MVC.....	34
2.2.1 ARCHITECTURE	37
2.2.2 DI PRINCIPLE.....	39
2.2.3 SPRING BEAN.....	41
2.2.4 WEB	42
2.2.5 RESTFUL COMMUNICATION	44
2.3 OPEN STREET MAP (OSM).....	47
2.4 PERSISTENCY TOOLS.....	50
3. IMPAIRED MOBILITY APP	53
3.1 CLIENT/SERVER MODEL.....	56
3.1.1 CLIENT FUNCTIONS.....	58
3.1.2 SERVER FUNCTIONS.....	60
3.2 ENTITIES OF INTEREST.....	63
3.2.1 POI	63
3.2.2 BARRIERS	65
3.3 USER INTERFACES	66
3.3.1 ANDROID APPLICATION.....	67
3.3.2 WEB PAGE	68
3.4 LOCALIZATION.....	69

3.5 COMMUNICATION	69
3.6 PLAN A SAFE ROUTE	70
4. APP AND SERVER DESIGN	71
4.1 EMPLOYED TECHNOLOGIES AND TOOLS	71
4.1.1 SERVER SIDE.....	72
4.1.2 CLIENT SIDE.....	73
4.1.2.1 ANDROID CLIENT	73
4.1.2.2 WEBPAGE	77
4.2 DATA PERSISTENCY.....	80
4.3 DESIGN OF THE ENTITIES OF INTEREST	81
4.3.1 POI IMPLEMENTATION.....	81
4.3.2 BARRIERS IMPLEMENTATION.....	83
4.4 LOCALIZATION TOOLS	84
4.5 DESIGN OF THE CLIENT/SERVER LOGIC.....	86
4.6 PEDESTRIAN ROUTE DESIGN	87
5. IMPLEMENTATION	90
5.1 IMPLEMENTATION OF THE CLIENT/SERVER LOGIC	92
5.2 IMPLEMENTATION OF SAFE ROUTE FUNCTION	100
5.3 DATA EXCHANGE.....	103
5.3.1 PHOTO SETTINGS	104
5.3.2 TRANSMITTED ENTITIES	106
5.4 GEOPOSITION.....	111
5.4.1 GEOCODING.....	112
5.4.2 REVERSE GEOCODING.....	113
5.5 EXPERIMENTAL MEASUREMENTS.....	113
5.6 TECHNICAL CONCLUSIONS AND FUTURE IMPROVEMENTS.....	117
CONCLUSIONS AND ON-GOING WORKS:	120

Introduction

Nowadays, around 1 billion people live with disabilities, and the percentage is around 15% of the total planet population. A large part of them suffers from impaired mobility problems that can cause a large number of difficulties related to the 'normal life' movements, so heavily limiting their life full experience.

Because of that, the society has become more and more sensitive to this kind of problematic: a vast number of institutions have been established to help and sustain this category of people, including ONG associations and public organisations. Their main goal is to help impaired people, pushing toward their integration into everyday activities.

Despite all these efforts, a lot of impaired mobility people cannot live an ordinary lifestyle because a lot of buildings and infrastructures are not equipped properly for their specific needs, lacking environmental tools that could help them to move. This kind of constraints is also evident if you look around: how many times have we seen a bar with an assisted entrance? Personally I have not seen so many of them, I have just seen a lot of places with a step in the main entrance, instead.

If a person without problems can easily pass over a step, a person in a wheelchair cannot do it so easily. Indeed, she cannot overstep it without external help, causing a physical limitation to her activities. At least in Italy, we are really far away from having a properly assisted society: most of the private activities do not provide any tool to let in these people. There is also a lack of IT platforms that support dynamically this category of people, caring about their social integration and in general, their needs.

This thesis project aims to tackle all these issues and proposes a platform to support people with impaired mobility by helping them to map all the buildings and a large number of architectural obstacles on a map and to review all of them considering their level of accessibility. The final goal is to help the impaired community to communicate with each other, sharing their opinions to let other people in the community choosing the better way to get across or the better place to go in order to be assisted properly.

In order to be effective in its goal, the application must be easy to use and accessible to everyone at any moment and everywhere. For this reason, the software will be developed around two pillars that are the main base for today communication: the Internet and the smartphone environment.

The first two chapters of this thesis have an introduction role. Indeed, the first chapter will introduce the background environment that surround this thesis project, in particular the 'smart cities' principle, getting through the technologies used for sustain this ideology. The second chapter will explain all the technologies used in this project, giving a background knowledge that is necessary for understanding the whole development process.

The last three chapters explain at different levels the software development: the third chapter contains all the consideration done during the analysis phase, getting through the general idea about the application workflow. It will introduce the general project environment and the different entities that compose the structure. The fourth chapter talks about the design of the application, explaining the structure of the project and the functions needed by the entities composing the whole pack. It shows and motivates how the technologies have been used into the project and the main logic of its development. The last chapter shows the coding phase, getting into the details of coding. It will contain sections about the algorithms used and about the resource management, including some monitored data.

1. Smart Cities Scenario

This section describes the general environment that surrounds the development of this thesis project and its main principle and purposes. I have decided to put this section at the beginning of the document because it is important to understand the main scenario in which this type of platform is inserted into, in order to understand the reason of its development.

Nowadays the cities are trying to collect data from their different areas to understand the necessities of their citizens. This thesis project has the same goal because the main idea is to collect information regarding the adaptation levels of the various structure elements of the cities for improving the life experience of people with impaired mobility.

This necessity has been born by the necessity to build an infrastructure that supports people with mobility handicap after collaboration between the project ParticipAct of the University of Bologna and the EU CHEGO LÁ. The first one is developing an *Mobile CrowdSensing (MCS)* application which can create a dynamic and collaborative infrastructure composed by the crowd, while the second one wants to support impaired mobility people in their every-day life creating them an open infrastructure that reviews different kinds of places. This put as key-principle of the project the easy use of it and the high interactivity level required.

As first, the designed platform has the main purpose to inform these people to the current conditions of the places in the current city regarding the adaptation tools. Indeed, it has been developed with the final purpose to help people with mobility problems in order to avoid unpleasant experiences caused by not adapted passages or unequipped structures that can represent a significant inconvenience. For these reasons, the system must be up to date, in order to provide real-time information to the final user.

For satisfy this necessity, there is the need to open the system to the users, allowing them the possibility to upload their own opinion and suggestions for helping the other community users. Because of this, the entire system bases its roots on the 'crowdsourcing' principle, which indicates the process to obtain services, ideas or just information by a large group of volunteer people that are not organised between them.

The term '*crowdsourcing*' has been used for the first time in the 2005 from Jeff Howe and Mark Robinson and derives from the idea that the work through Internet was "outsourcing to the crowd". This allows obtaining a large amount of data, usually current information, by a self-organized organism composed by people from different parts of the world.

More specifically, the key-principle of this project is the *crowdsensing*, which is a branch of the crowdsourcing.

1.1 Crowdsensing

The term '*crowdsensing*' indicates the concept by which a large amount of mobile sensing devices share a considerable amount of data with the aim of measuring physical phenomena or to support a common interest purpose.

These sensing devices could cover a large spectrum of objects, such as smartphones, sensors embedded in videogame platforms, music-players or built-in car devices. Obtaining information from these such devices is possible because they contain a large number of sensors, as motion sensors, accelerometer, GPS, microphone, ... This makes possible the sharing of multiple type of information with an enormous amount of people.

Mainly, there are two different type of crowdsensing: the Participatory sensing and the Opportunistic sensing. The first requires a direct interaction with the user, while the second work in background, without

breaking the normal device activities. Crowdsensing applications represent a turning point regarding the data retrieving, because with this kind of approach is possible to create a self-organized system which can guarantee a complete set of data: it is self-organized because the entire data stored are inserted from the users and, moreover, the received data are obtained from an extended spectrum of people, which guarantees a 360° set of information.

Usually, these kinds of systems contain a real-time set of information because they are characterised by a high-speed updates frequency. This characteristic cannot be reached by closed system, because they do not allow the users to change the database content.

So, basing on these factors, the purpose of the crowdsensing project is to create an application that can draw a general review of a city considered from a specific point of view, registering and storing the weak and the strong points of it. Regarding this thesis project, the crowdsensing tool is used to collect the information needed to review the city structures equipment, in order to communicate to the final user a brief suggestion based on main criteria that can summarize the actual conditions of the selected entity.

1.2 Smart cities

The 'smart cities' expression indicates a city able to improve its own services and tries to enhance the personal experience of its own citizens through the use of the modern technologies. The concept is briefly explained by the Fig. 1, 2, which shows a possible smart city model, in which all the needed functionalities are improved and synchronized together by sensors, constantly monitoring a focused condition, which could be the pollution of the air, the traffic congestions, etc.

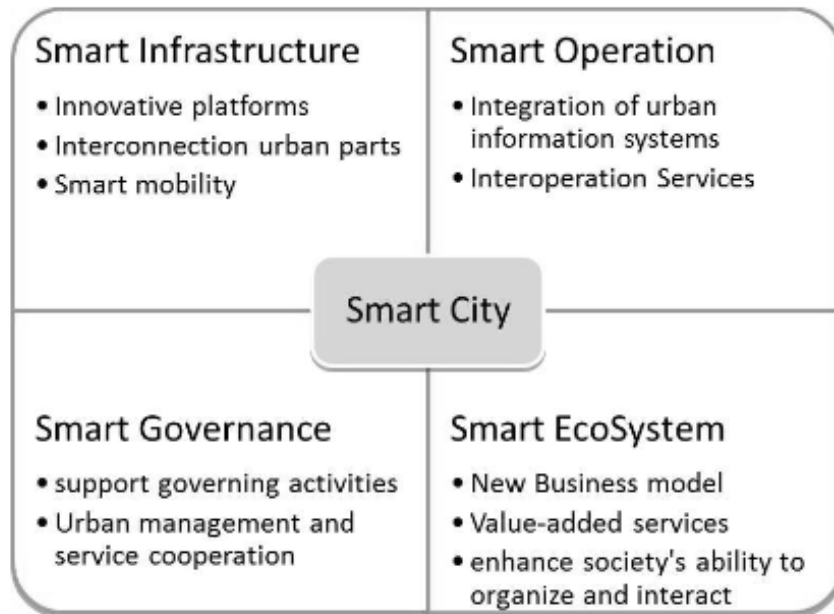


Figure 1. Smart city principles

Nowadays, the current way to obtain information about the city is through fixed sensing devices as video cameras, traffic lights and other types of fixed sensor. It is evident that this type of approach is limited, because this kind of sensors covers a very strict set of data. Indeed each single sensor can retrieve a specific type of information, as video for the camera, the number of current vehicles and their average speed for street sensors or just the current pollution level of the air.

This means that theoretically, in order to collect a correct draw of a city using these kinds of sensors the entire city should be covered by them because, as the name suggests, they can register the activities just in a little area. Moreover, it would imply a huge amount of money and a high-efficiency organisation that will merge all the received data and stored them in real-time.

At last, with the increasing number of people who lives in the cities, this method could be considered inefficient because it could not produce the amount of information needed, which is directly proportional with the number of the city citizens.

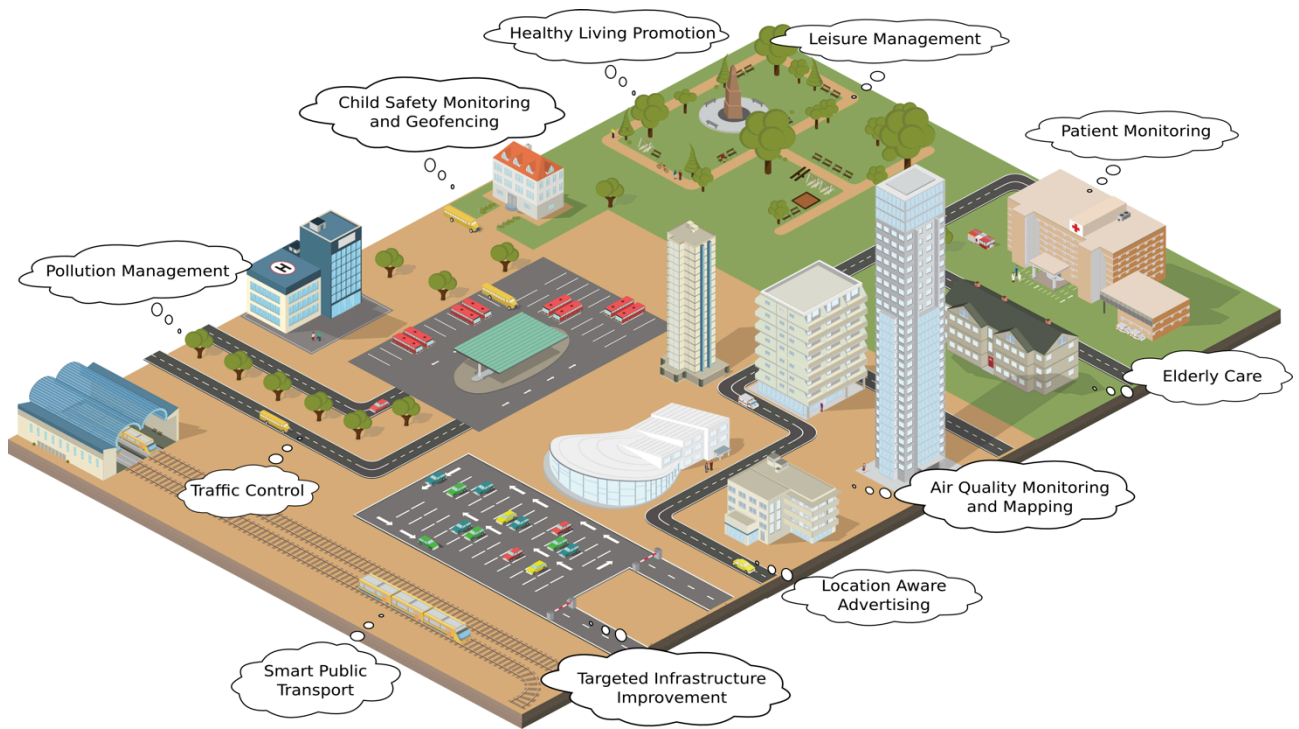


Figure 2. Possible functionalities implemented by an efficient smart city system

For this purpose, crowdsensing is a new paradigm that can be useful in order to handle the increased number of required data. By this resolution approach, the citizens will become active actors that can collect data around the all city area, constantly updating the data pool. Applying it does not imply any kind of investment because the smartphones or other mobile devices will represent the required sensors.

This can potentially create a self-organized system that ask to groups of citizens to complete some kinds of ‘Tasks’ and upload the data to a central system that computes the received data and collects information regarding the city. These information cover a large spectrum of possible options: for example using the smartphones is possible to measure the air pollution and noise pollution, it is possible to share a photo of an architectonic barriers that blocks a certain road and so forth. So, by these kind of sensors is possible to retrieve a various set of data regarding different aspects, a feature that is not provided by the fixed category. Moreover, these sensors are mobile because they get around with the person who carries them: it means that a single sensor could cover different areas in different times.

A possible drawback is that use the citizens as data miners does not guarantee a constant data flow: while using the fixed sensors pledge a 24/7 data computation, crowdsourcing application must let to the user the possibility of shutting down all the sensors and interrupting any type of data sourcing, allowing his/her own privacy.

1.2 Actual supporting systems

As briefly explained before, nowadays the main crowdsourcing sources are the 'sensor networks'. This term indicates a network composed by autonomous sensors spread into a chosen area for monitoring it and retrieves useful information. There are multiple types of sensor networks and one of the most common type is the one founded on Machine-to-Machine (M2M) principle.

This term indicates a large number of devices that communicate between the various entities through a direct link or satellites networks, without any human interaction. The sensors just collect data during all they lifetime and send them to the repository. The sensors could cover a wide area of different data type. The Fig. 3 summarize a simple M2M structure just for exemplify the concept: there are some sensors, in that case three different sensors, which collect the information from the environment and send them to the M2M AreaNetwork that will handle the received data for sending them to the M2M Server, which will be programmed for understanding the meaning of the received data and act basing on it. In the picture case, the server will send a notification to a device.

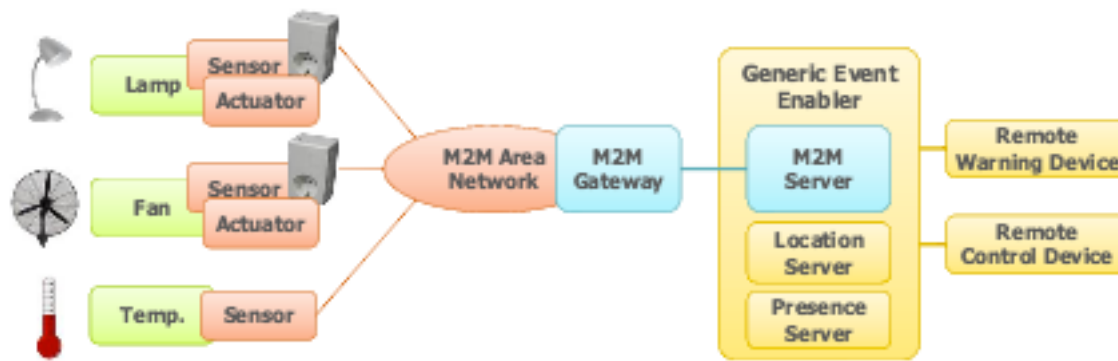


Figure 3. Example scheme of M2M

1.3 Nowadays necessities

The need to obtain day by day a larger amount of data from the sensors descends from the increasing number of the cities inhabitants. This can be easily afforded through the MCS applications because each person has its own smartphone, increasing exponentially the number of the available sensors. The MCS applications are platforms that use the mobile devices, as smartphone or tablets, as sensors. Indeed, today's mobile devices have more computing, storage and communication resources than a simple sensor; also usually they have the possibility to use various sensors in the same time, allowing storing a notable amount of data.

Secondly, people bring their smartphone whatever they do, making the smartphone a highly reliable crowdsourcing tool: it can potentially store data all day long, documenting our life day by day and merging the collected data into a single database.

An example can be the estimation of the traffic congestion. Nowadays the data for calculating it come from mote-class sensors that have been places and set up in the road. Instead of spend money and resources for setting up this kind of network, we could just use the localization sensor

of the drivers: the larger number of near positions the system receive, the higher is the probability of a congestion in that point of the road.

Another useful application would be the customization of a map, which is not possible by the traditional M2M sensor networks. Indeed, information as the actual condition of a place or the review of its activity is not obtainable through sensors, but just by the users sharing. This allows to create interactive map platforms that can help the citizens to map a certain area in the most useful way for them. A common example is the *Point of Interest (POI)* mapping, as explained in the Fig. 4.



Figure 4. A POI-oriented map

Unfortunately, these MCS application still presents several drawbacks: for example the data could be of a different quality type, because not all the mobile devices have the same type of sensors, computing resources or accuracy. For this reason, in order to implement a functional MCS application a device target must be identified, allowing having a minimum quality data.

Another aspect that must be considered is the privacy: of course humans concerns their own personal spaces and the user may not want to share data sensors that can reveal private information, as for example the current location. For this reason, there is the necessity to find a way to convince the user to use MCS applications and to share data, such as incentives. So, based on these assumptions, crowdsensing seems to be the most eligible way to collect data following the 'smart city' concept,

even considering its drawbacks. Indeed, while a sensor can just register and share physical data, MCS applications can also collect other kinds of data, such as social data.

Three different categories of MCS application can be considered:

- **Environmental:** these applications measure natural events, collecting physical data exactly as the mote-class sensors do
- **Infrastructure:** these applications measure large-scale phenomena that are related to the public infrastructure. This can also be implemented using sensors, implicating a waste of resources
- **Social:** these applications regard information sharing between the people of the same community. This feature, obviously, cannot be implemented by sensors and it represents one of the most useful functionalities for citizens.

2. Technologies background

This chapter overviews the main technologies employed in this project. I have decided to put this background material before other chapters because the main idea is to build background knowledge to understand the choices taken into the project.

Indeed, the world technologies are improving and changing day-by-day, introducing new kind of devices and communication methods. The main point of strength of a MCS application lies in the end-user device on which it has been developed: to ensure a frequent use by the user it should be developed on a platform that is at the base of the all-day life and that allows to share data by remote.

For this purpose this platform has been developed on two main environments that represent the routine of the modern human: the mobile and the web.

2.1 Android

Android has been initially developed by Android Inc. and later bought by Google in 2005, next unveiled in 2007. This alliance has the common goal to foster innovation on mobile devices and to give to the consumers a better user experience than what is available on today's mobile platforms. The main idea behind Android is to create an openness platform that allows the developers to work more collaboratively, increasing rapidly the services offered by the system itself.

A large number of industries build mobile devices based on Android *Operative System (OS)* as Samsung, Lenovo, Huawei, LG, Xiaomi, etc... which makes Android the most common installed mobile OS. Indeed nowadays, it covers, basing on 2015 statistics, the 82,8% of the mobile market, followed by the 13,9% of the iOS OS, as shown in Fig. 5.

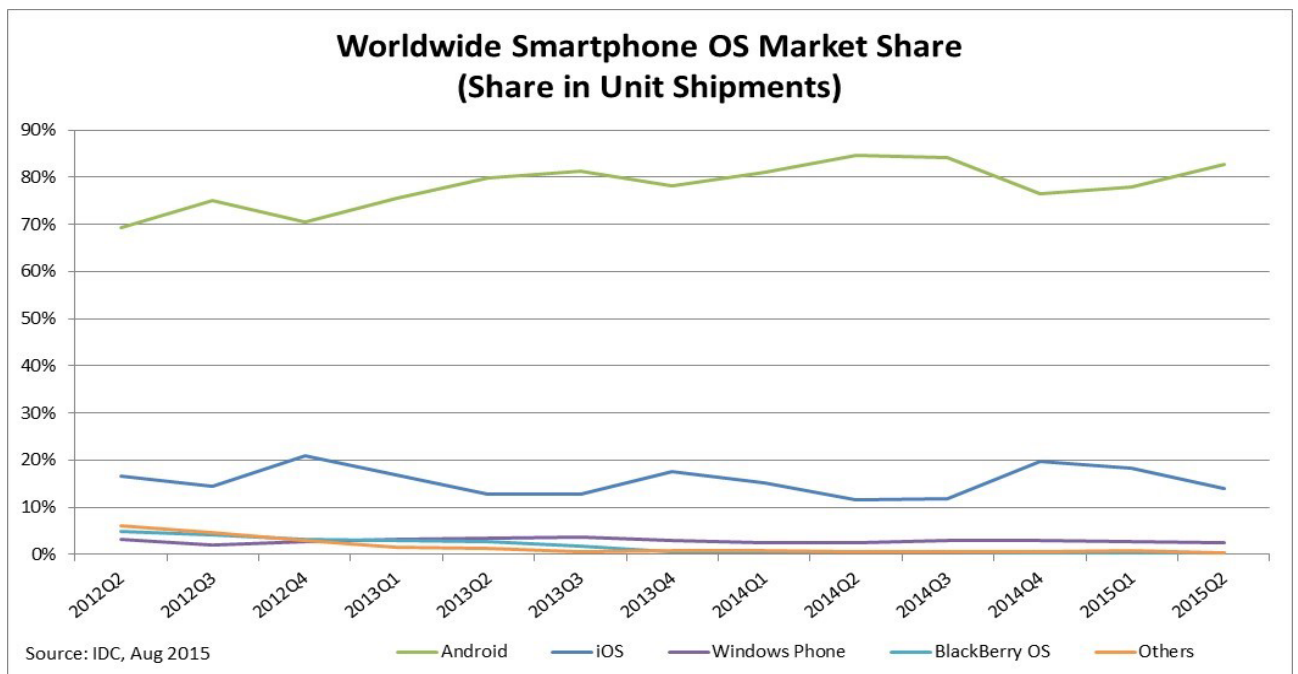


Figure 5. A graph representing the most used mobile OS

Android is a totally open source OS, with is characterised by one of the largest community, due to the reduced development costs and to the rich development environment. Another main aspect of Android is the constant version update: starting from the unveiled in the 2007 and until today, 11 versions have been published:

Code name	Version number	Initial release date
Cupcake	1.5	April 27, 2009
Donut	1.6	September 15, 2009
Eclair	2.0–2.1	October 26, 2009
Froyo	2.2–2.2.3	May 20, 2010
Gingerbread	2.3–2.3.7	December 6, 2010
Honeycomb ^[a]	3.0–3.2.6	February 22, 2011
Ice Cream Sandwich	4.0–4.0.4	October 18, 2011
Jelly Bean	4.1–4.3.1	July 9, 2012
KitKat	4.4–4.4.4, 4.4W–4.4W.2	October 31, 2013
Lollipop	5.0–5.1.1	November 12, 2014
Marshmallow	6.0–6.0.1	October 5, 2015

Passing to the hardware part, originally the main hardware platform is the ARM architecture, with the adding of x86 and MIPS architectures officially supported in later Android versions. Moving forward with the more recent versions of the OS the requirements have changed, reaching the 1824MB required on devices running Marshmallow version with a high screen definition.

Also, each Android smartphone is composed by a large number of sensors, as still or video camera, GPS, orientation sensors, accelerometers, gyroscopes, barometers, magnetometers, proximity sensors, thermometers and touchscreens. These hardware characteristics make the mobile devices category the most eligible sensors nowadays.

2.1.1 Architecture

Android is an OS that is characterised by a pretty complex architecture, shown in the Fig. 6.

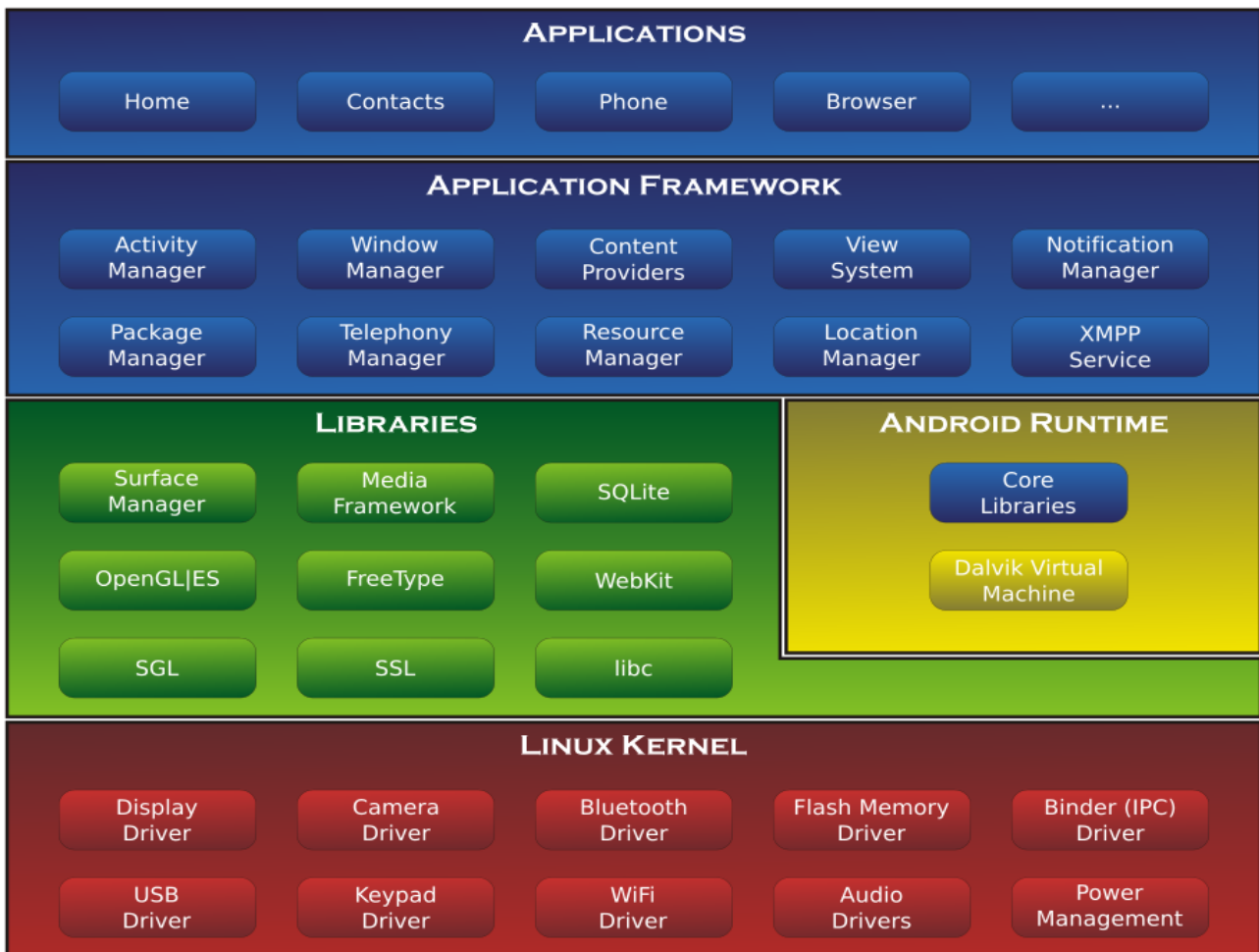


Figure 6. The Android architecture

The bottom layer (the red one in the picture) is composed by a Kernel, which is based on a Linux kernel (the original version was the 2.6, from android 4.0 it has been updated to version 3.x), which approximately 115 patch. This layer provides a level of abstraction from the device hardware and it contains all the essential hardware drivers, interfacing all the peripheral devices.

On top of the Linux kernel there is a set of libraries (in the picture the green part) that gives to Android the needed functionality for guaranteeing a high interactivity. For example we can notice WebKit, which is an open-source web browser engine, SQLite, which is a useful repository for storing data, and SSL, which is responsible for Internet security. In addition to these libraries, Android encompasses Java-based libraries that are specific to the Android development. These libraries facilitate for example the content access, providing some default services including messages, system services and inter-process communication.

On the same level of the libraries we find a section called Android Runtime that contains two main elements in the Android architecture. The first is the Dalvik Virtual Machine, which is a JVM that uses Linux core features like memory management and multi-threading. This VM allows to every Android application to run in its own Dalvik VM instance, without interfering with the others processes handled by the OS. The second is a set of core libraries that enables Android application developer to write Android applications through the Java programming language.

On top of these two second-level layers there is the Application Framework level that provides high-level services in form of Java classes that can be used and modified by the developers. One of the main services offered by this layer is the *Activity Manager*, which controls all aspects of the application lifecycle, interacting with the overall activities running in the system. As following there is the *Content Provider*, which allows the data sharing between different applications. This is particularly useful when a data has been obtained with one application, for example the camera, but it needs to be used into another application context, for example upload a comment with a photo into a second application. Thirdly, we find the *Resource Manager*, which provides access to non-code embedded resources, as layouts, images, pre-defined strings and other type of resource data. Then there is the *Notification Manager*, which handle all the notification system of the Android applications, allowing showing alerts or messages to the user. At last, the *View System* guarantee a large set of views used to create the GUIs.

As top layer there is the Application level, which contains all the installed application on a device.

2.1.2 Application lifecycle

An Android application is composed by building blocks called Components. Briefly the four main components of an Android application are the *Activities*, which handle the user interaction, the *Services*, which handle background processes, the *Broadcast Receivers*, which handle the communication between the OS and the applications and the *Content Providers*, which handle the data management. These entities are declared and coupled into the *AndroidManifest.xml* file that ratifies the interactions between them, other than the application permissions and all the libraries dependencies.

The application object is created whenever one of the declared components is started, creating a new process with a unique ID under a unique user. This object starts before any declared component and runs as long as another component of the application runs.

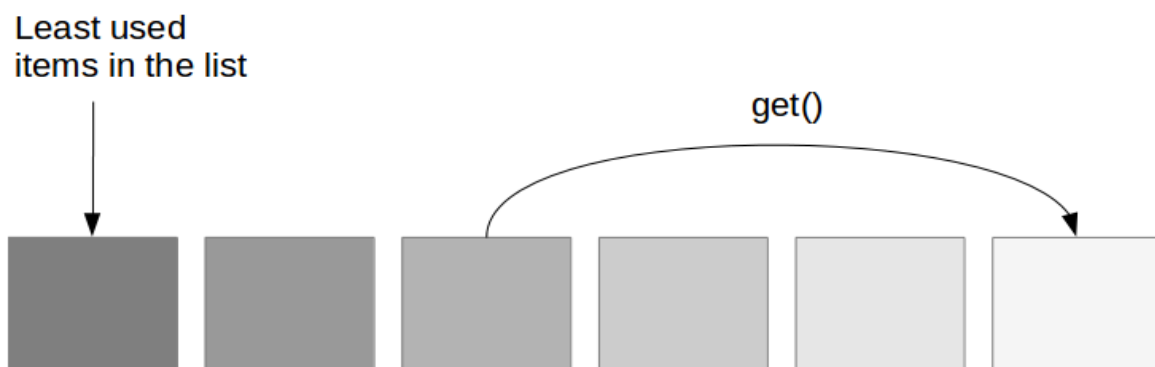
When an application is started the system checks if an instance of the selected application is still in the device memory, allowing a faster restart of the object. Of course, the application objects are removed from the stack if the system needs to free up resources. This operation follows a priority system explained below:

Process status	Description	Priority
Foreground	An application in which the user is interacting with an activity, or which has an service which is bound to such an activity.	1
Visible	User is not interacting with the activity, but the activity is still visible or the application has a service which is used by a inactive but visible activity.	2

Process status	Description	Priority
Service	Application with a running service which does not qualify for the precious priorities.	3
Background	Application with only stopped activities and without a service or executing receiver.	4
Empty	Application without any active components.	5

All processes with priority 4 or 5 are added to a *Least Recently Used list (LRU list)* and the processes that are at the beginning of it are killed if the system requires a free up of resources. The opposite option is the one in which the processes have been recalled by the user, obtaining a high priority. This workflow is illustrated in the Fig. 7, which explains the LRU Cache logic.

LRU Cache



Calling `get()` for an item, moves it to the top of the cache

Figure 7. The Least Recently Used list behaviour

2.1.3 Activities and Fragments

As a C, C++ or Java program start from a `main()` function, the Android application is initiated by an *Activity* through the `onCreate()` call-back method.

In particular an activity is an application component that provides a Graphical Interface, allowing the user to interact with the application. Multiple activities bounded together create an application object. Between all the activities there is an activity called 'main' that is the first activity called when the application is launched.

When another activity starts the previous is stopped and is put into the *BackStack*, which is a stack that store the activities sequence. For an efficient organisation of the activities, Android has implemented a lifecycle for this kind of entities.

As we can notice in the Fig. 8, this lifecycle is based on the activity state and allows creating a strong and flexible application if used in the right way. An activity can exist in three different states: Resumed, Paused and Stopped.

Resuming an activity means to recall the selected activity in foreground, 'running' the current activity. Instead, when an activity is in the Paused state, it means that another activity is in foreground but the previous activity is still visible (for example a half transparent activity that does not fully cover the screen size). When an activity is in the Stopped state, the activity is totally obscured by another activity. Both Paused and Stopped activities could be killed if the Android system needs to free up memory.

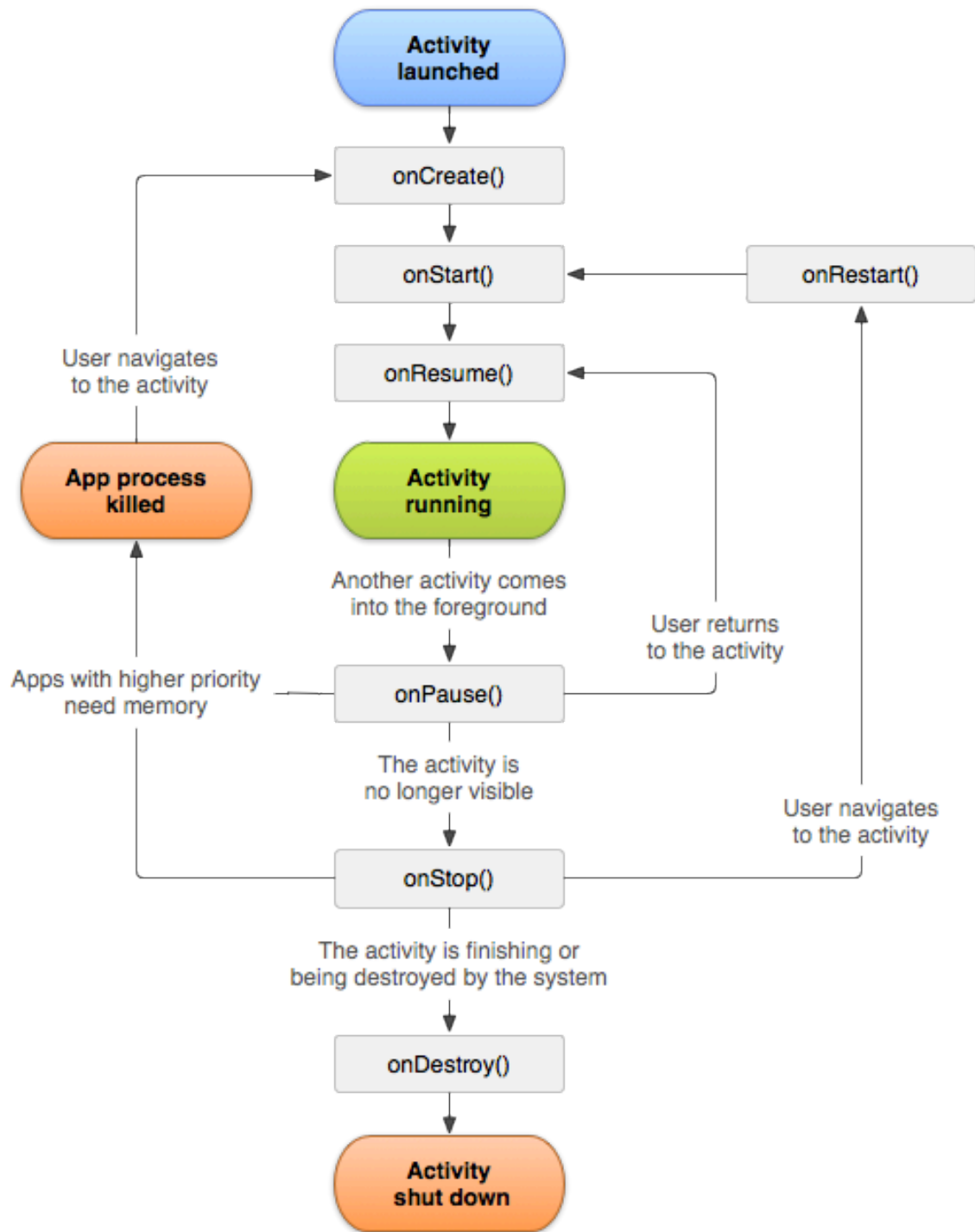


Figure 8. An activity lifecycle

As illustrated into the Fig. 8, each change of state launches a method (`onCreate ()` when the activity is created, `onStart ()` when it becomes visible to the user, `onResume ()` when the user starts interacting with the application, `onPause ()` when the current activity is being paused and another is being resumed, `onStop ()` when the activity is no longer visible, `onDestroy ()` when the activity is destroyed by the system, `onRestart ()`

when a stopped activity is being resumed) and each method can be customized replacing the default one.

Moreover, an activity can contain a various number of other types of component: the *Fragments*. A Fragment is a portion of the user interface into an Activity, which acts as 'Container of fragments'. Indeed a single activity can contains multiple fragments, or a new fragment can replace and old one.

This is possible because fragments are always embedded into an activity; for this reason the activity lifecycle heavily conditions the fragment lifecycle. For example if an activity is paused, all its referred Fragment entities are paused too. The philosophy behind this component is to support a more dynamic and flexible UI. Indeed, Fragments were not implemented at the Android born, but have been introduced in Android 3.0.

Fragment are characterised by their own lifecycle, which is displayed in the Fig. 9. As it is clear, the Fragment lifecycle follows the activity philosophy, launching methods when the state of the fragment element changes. Indeed, three different states exist that refers to a fragment: Resumed, Paused and Stopped.

The state-dependant methods are the following:

- `onAttach()` : called when a fragment had been associated with the activity
- `onCreateView()` : called to create the associated view hierarchy
- `onActivityCreated()` : called when the activity method `onCreate()` has returned
- `onDestroyView()` : called when the associated hierarchy is being destroyed
- `onDetach()` : called when the fragment is bing disassociated from the activity

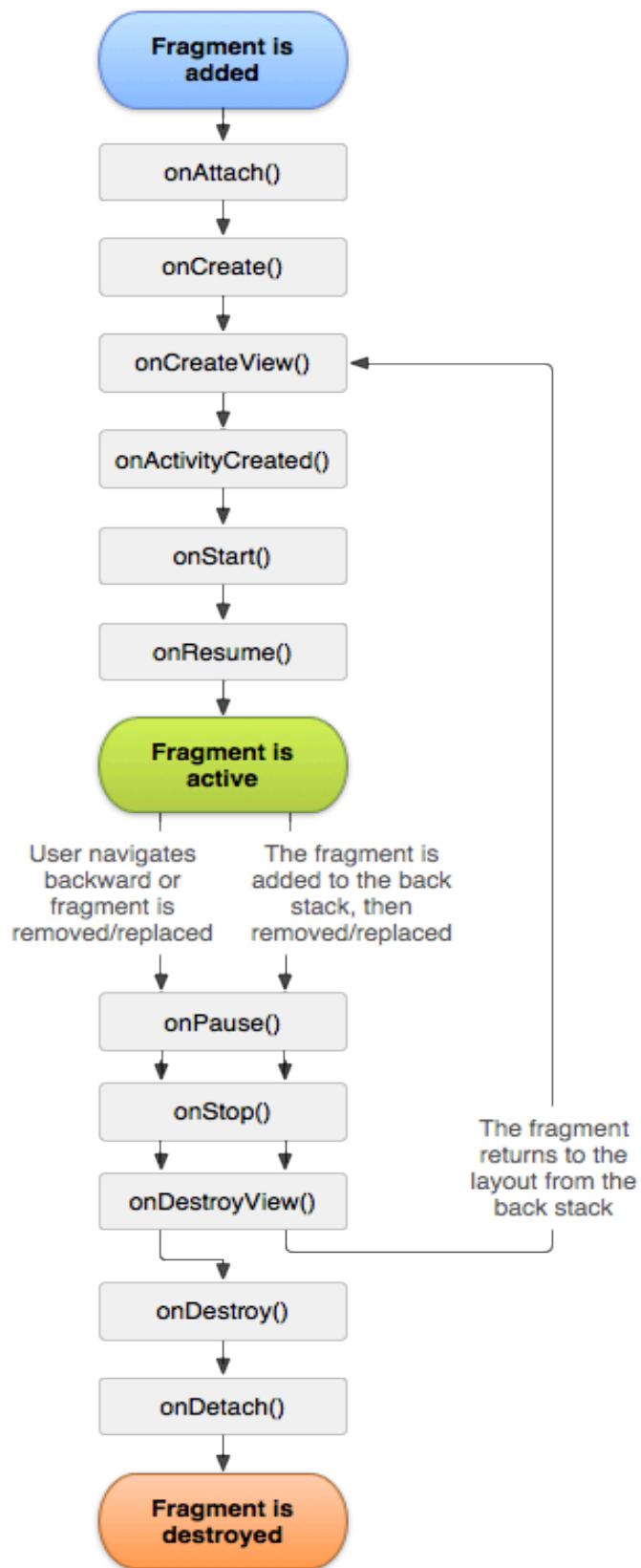


Figure 9. A Fragment lifecycle

One of the main differences between an activity and a fragment is that whenever a fragment is stopped it is not stacked into a collection by default as an activity does. Indeed, this feature is obtained by calling the *FragmentManager* method *addToBackStack*, adding in this way the fragment to a stack hosted by the activity.

2.1.4 Intent

An Android *Intent* is an abstract description of an operation to be performed and it could be used for launching an activity or for communicating with another component of the application, as a Service or BroadcastReceiver. More precisely, an Intent is a bundle of information that carries information used by Android OS to determine which component must be called, as showed in Fig. 10.

Furthermore, there are two different types of Intent types: the explicit Intents that contains the name of the component to start, and the implicit Intents that do not name a specific component, but letting the system handles the requested action.

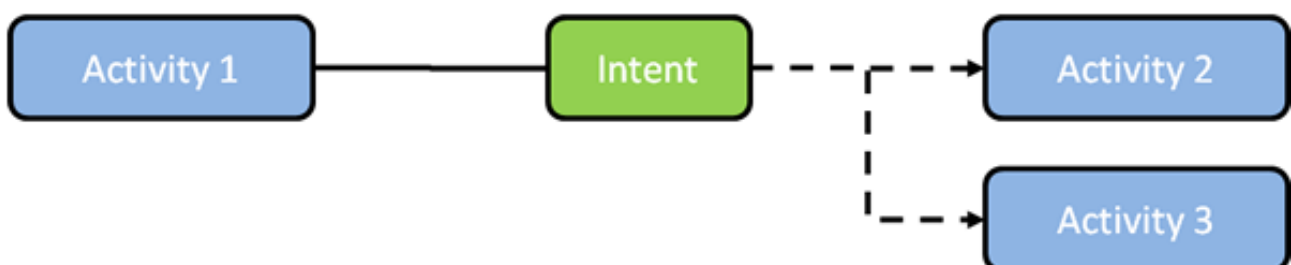


Figure 10. The intent role into Android

To handle the second type of Intent, the Android system checks the <intent-filter> tag into the AndroidManifest.xml file. Using this tag is possible to specify the type of intent that an entity can accept based on the intent <action>, <data> and <category>.

Intent

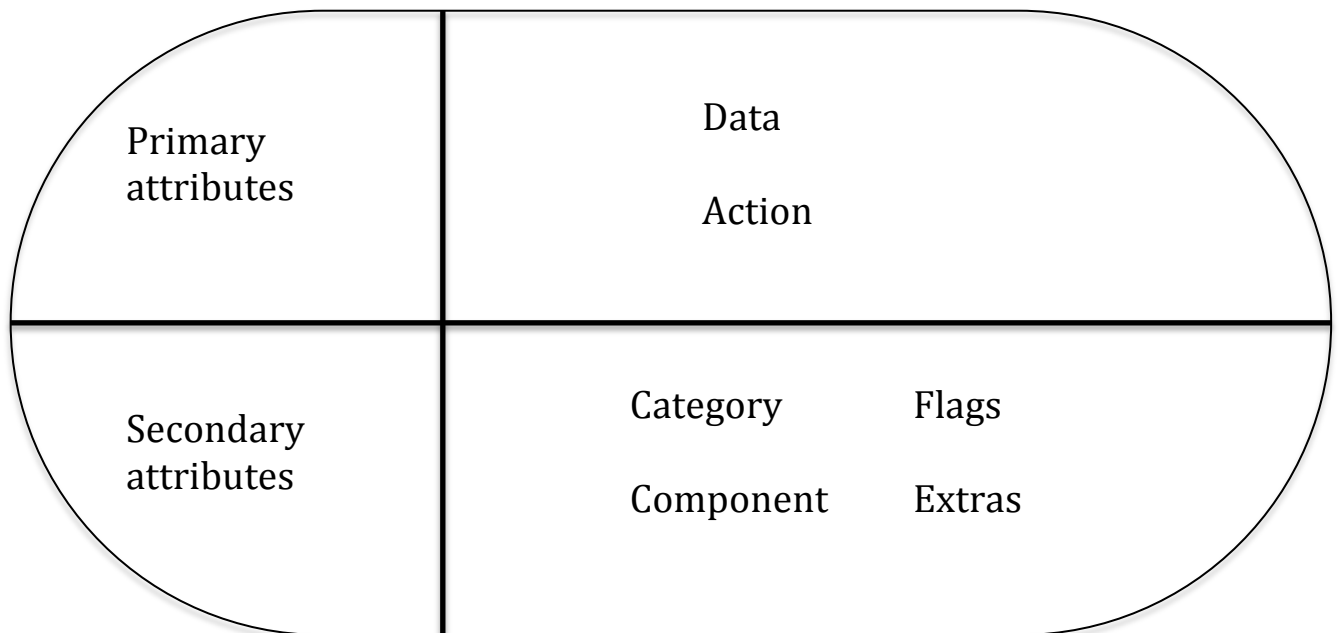


Figure 11 The Intent data entities

As showed in Fig. 11, an Intent is composed by six components:

- **Action:** this is a mandatory part of the intent and is a string indicating the action to be performed or reported.
- **Data:** this field adds a data specification to an intent filter, consisting in a data type or a URI or simply both
- **Category:** this field is an optional part of Intent object and it's composed by a string containing additional information about the kind of component that should handle the intent.
- **Extras:** this field contains key-value pairs containing additional information that must be delivered to the called component, as variables.

- **Flags:** This field contains instructions on how to launch an activity and how to treat it after it's launched.
- **ComponentName:** This optional field is a `ComponentName` object representing a component class. If this field is set, the `Intent` object is delivered to an instance of the addressed class.

2.1.5 Services

A *Service* is an android component that can perform long operations in the background. They do not provide any type of user interface and they are launched by other components and still run even if the component or the application itself has been stopped.

A *Service* is characterized by a state, which has two different setting option: the first is `Started`, which means that an application component starts it calling `startService ()`, the second is `Bound`, which indicates that an application component binds to the *Service* entity by calling `bindService ()`. A bound *Service* offers a client-server interface that allows the other application components to interact with *Service* obtaining results or sending requests across processes; furthermore, more components can bind to the same *Service*.

Another *Service* characteristic is that this component has a lifecycle and some call back methods. The Fig. 12 illustrates which is the lifecycle of a *Service* started with `startService ()` on the left side, while on the right side it explain the lifecycle of one started with `bindService ()`.

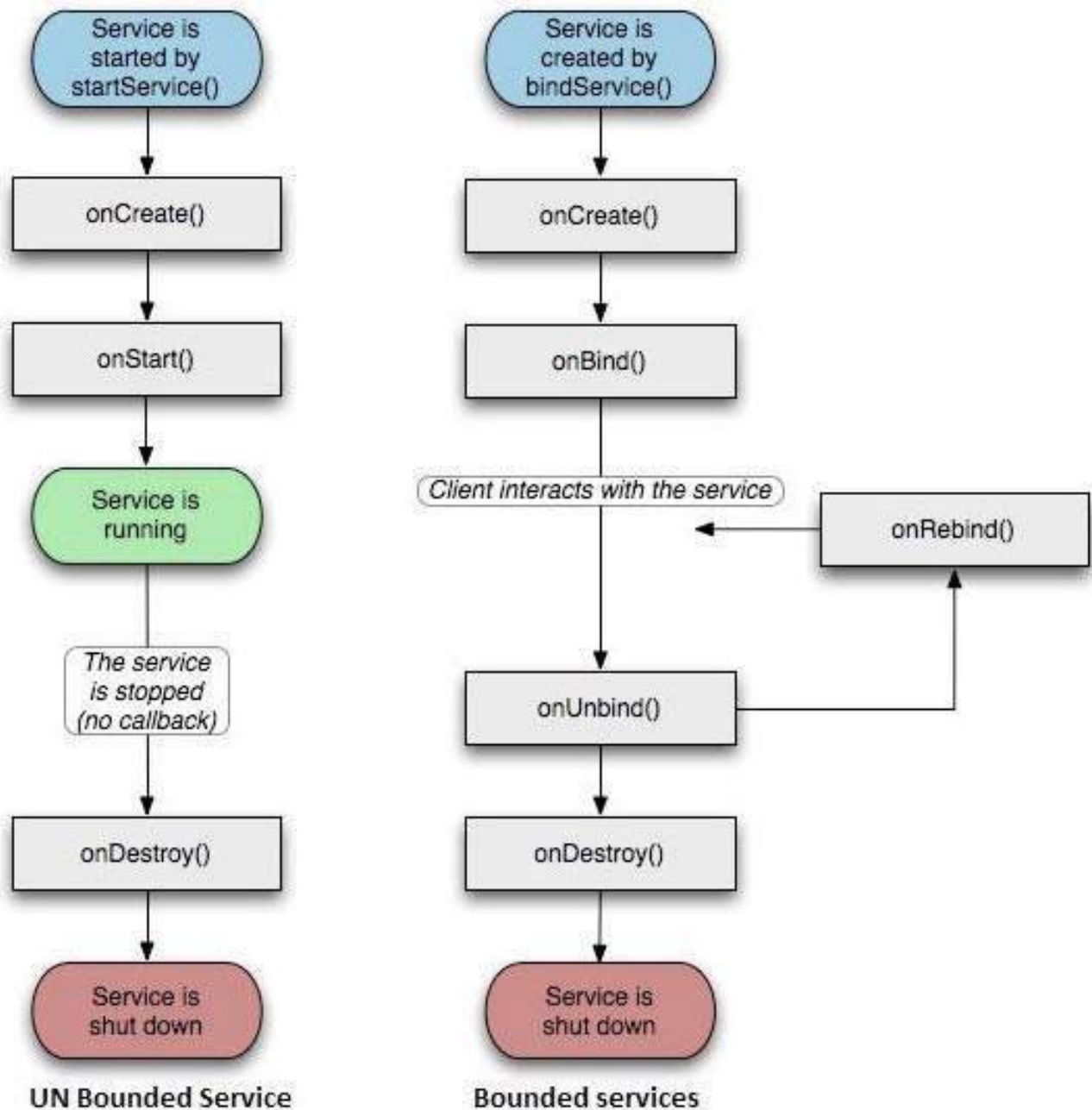


Figure 12. A Service lifecycle

The most important method that must be overwritten for create a Service are:

- `onStartCommand()`: this function is called when another component requests to start the service by calling `startService()`
- `onBind()`: this function is called when another component wants to bind to the service by calling `bindService()`

- `onCreate()`: this method is called when the service is created
- `onDestroy()`: this method is called when the service is not used and is being destroyed

These functions represent the code lifecycle of a Service component. Once a Service is started is a developer duty setting an event that calls the stop of the component, even if the application has been destroyed, or it could potentially go on until the entire device is shutting down.

2.1.6 AsyncTask and Loaders

By default, Android modifies the user interface and handles inputs event from one single user interface thread, called the main thread and If the programmer does not use any concurrency constructs, all code of an Android application runs in the main thread and every statement is executed after each other. If you perform a long lasting operation, for example accessing data from the Internet, the application blocks until the corresponding operation has finished. For this reason, Android as OS provide two main classes that can handle asynchronous requests: the AsyncTasks and the Loaders.

The AsyncTask class encapsulates the creation of a background process and the synchronization with the main thread. It also supports reporting progress of the running tasks. To use AsyncTask you must subclass it. Indeed, AsyncTask uses generics and varargs. The parameters are the following `AsyncTask <TypeOfVarArgParams , ProgressValue , ResultValue>` An AsyncTask is started via the `execute()` method, which calls the `doInBackground()` and the `onPostExecute()` method. `TypeOfVarArgParams` is passed into the `doInBackground()` method as input, `ProgressValue` is used for progress information and `ResultValue` must be returned from `doInBackground()` method and is passed to `onPostExecute()` as a parameter. The `doInBackground()` method contains the coding instruction which should be performed in a background thread. This method runs

automatically in a separate Thread. The `onPostExecute()` method synchronizes itself again with the user interface thread and allows it to be updated. This method is called by the framework once the `doInBackground()` method finishes. The whole lifecycle is summarized in the Fig. 13.

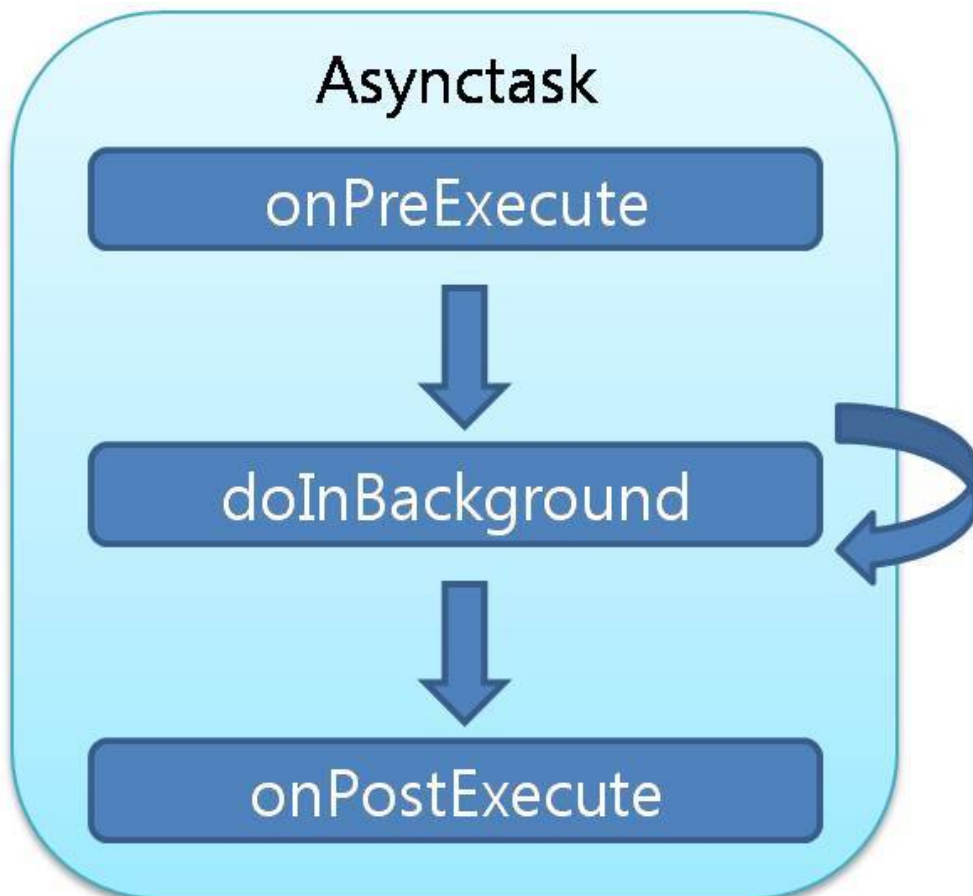


Figure 13. AsyncTask workflow

Because an AsyncTask has methods that run on the worker thread (`doInBackground()`) as well as methods that run on the UI (e.g. `onPostExecute()`), it has to keep a reference to its Activity as long as it's running. But if the Activity has already been destroyed, it will still keep this reference in memory. This is completely useless because the task has been cancelled anyway. Another problem is that we lose our results of the AsyncTask if our Activity has been recreated. For example when an orientation change occurs. The Activity will be destroyed and recreated, but our AsyncTask will now have an invalid reference to its Activity, so `onPostExecute()` will have no effect.

The second class that can implement asynchronous requests is the Loader class. This class allow to load data asynchronously in an activity or fragment. They can monitor the source of the data and deliver new results when the content changes. They also persist data between configuration changes. If the result is retrieved by the Loader after the object has been disconnected from its parent (activity or fragment), it can cache the data. Loaders have been introduced in Android 3.0 and are part of the compatibility layer for Android versions as of 1.6. This class is supported by a second entity, the LoaderManager, which keeps your Loaders in line with the lifecycle of your activities or fragments. If Android destroys your fragments or activities, the LoaderManager notifies the managed loaders to free up their resources. The LoaderManager is also responsible for retaining your data on configuration changes like a change of orientation and it calls the relevant callback methods when the data changes. Loaders are characterised by four different states:

- **Reset:** in this state, the loader gives up any data associated with it for garbage collection. Called by the LoaderManager when destroying the loader.
- **Started:** this is the started state achieved by a call to `startLoading()` that'll invoke the `onStartLoading()` callback. Monitoring for changes and performing new loads based on the changes will also be done here. This is the only state in which `onLoadFinished()` is called (generally in the UI thread).
- **Stopped:** in this data, no data can be delivered to the client (that can only happen in the Started state). It may observe/monitor for changes and load content in the background for the purpose of caching that can be used later if the loader is started again. From this state the loader can be started or reset.
- **Abandoned:** an intermediary state between *stopped* and *reset* where it holds the data until a new loader is connected to the data source, so that the data is available until the restart is completed.

2.2 Spring MVC

With the technology progress we are assisting to a radical change of human habits. Always more our life is strictly related with the technologies around us, and one of these technologies is Internet. As we can notice from the Fig. 14 the graph curve trend is almost exponential, indicating that the number of users is increasing very quickly.

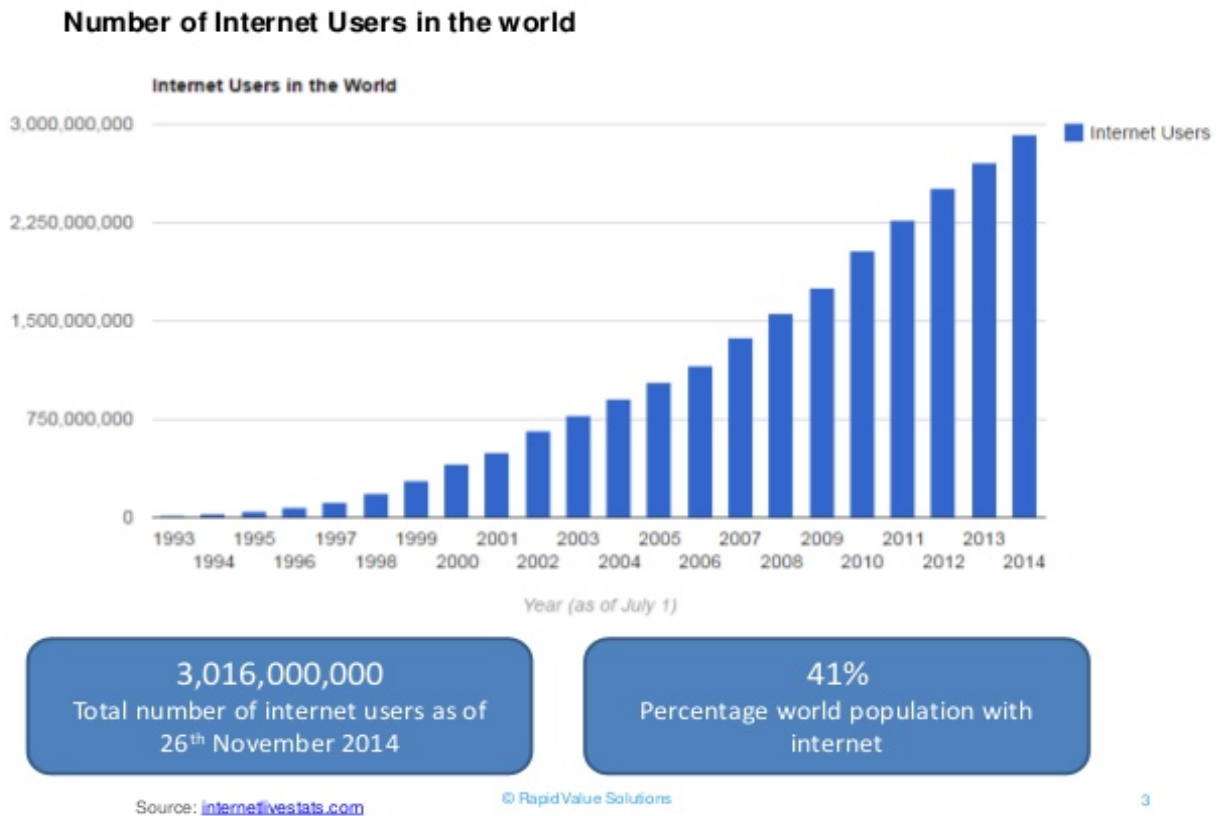


Figure 14. Internet users until 2014

Internet has brought an inner innovation in the communications environment, redefining the most common communications media, giving birth to a new kind of services as Internet telephony and Internet television. This has facilitated sharing ideas between people, creating new dynamic platforms as blogs or web feeds that allows to publish thoughts

and opinions on the web. Because of the freedom that Internet allows to its user, a lot of companies offer their services online, through the websites. This kind of approach is replacing totally the physical one.

Nowadays there are a lot of different websites that covers the most widely spectrum of functions, starting from the social networks and ending with the shop websites, passing through personal spaces in which the owner can publish whatever he wants to, as blogs. Anyway, even these kind of services need to be constantly updated.

Indeed, in the beginning all the websites were static, which means that all the content was written in HTML combined with CSS, guaranteeing a short development time and a faster website, because the content of the page can not change and remains every time the same.

The problem with this approach is the low interactivity of the website. Indeed, the website does not adapt itself to the current user and does not allow the development of any kind of personalized section. For these reasons nowadays the largest number of websites are dynamic, which are not just composed by HTML and CSS, but also contains some part of a web-scripting language, as JavaScript, PHP, Ruby, etc.... These scripting languages allow adding functionalities to webpages, such as recover data from the database, save data into the database or calling external APIs to get some information, as shown in the Fig. 15.

A platform that can handle this kind of complexity is Spring, which is one of the most popular application development framework for enterprise java allowing to create an high performing reusable code.Spring framework is an open source Java platform initially written by Rod Johnson and the first release was under Apache 2.0 in 2003. Using Spring brings various kind of benefits, first of all this platform enables developers to develop enterprise-class applications using POJOs.

POJO is an acronym, which stands for *Plain Old Java Object* and indicates a Java object not bound by any special restriction. Using POJOs allows Spring to do not need an EJB (Enterprise JavaBeans) container, but it needs just a robust servlet as Tomcat.

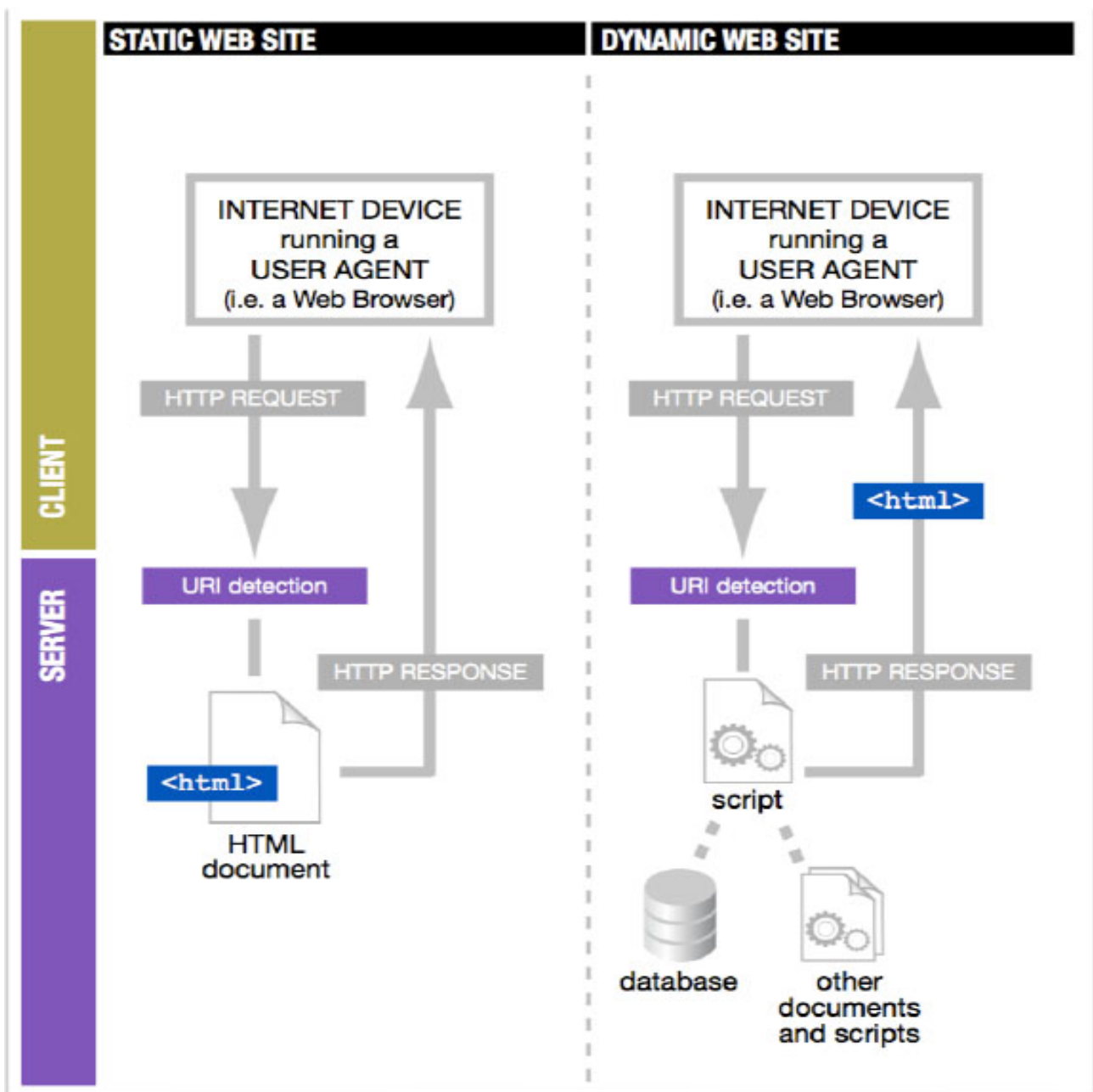


Figure 15. Differences between a static website and a dynamic one

Moreover, Spring is based on the *ModelViewController(MVC)* framework, providing a great alternative to other web frameworks, and it provides an API to translate specific exceptions of different technologies, such as JDBC, Hibernate or JDO. Furthermore, this framework is lightweight if considering size and transparency caused by the usage of an *Inversion of Control (IoC)* container, especially if compared to an EJB one. This brings a consider limit of the resource consumptions and CPU.

One of the key components of Spring is the *Aspect Oriented Programming (AOP)* Framework that provides the implementation of aspects and behaviours present across the application, dividing them from the application domain. At the base of this approach there is the concept of Crosscutting Concern, which represents functionalities that can be invoked in various points of an application, such as caching, logging or the authentication. To reduce the redundancy of the application the AOP approach tries to isolate these functionalities in a strict number of modules, called aspect that represent the modularity unit of the AOP, as the class represent the unity of modularity of OOP.

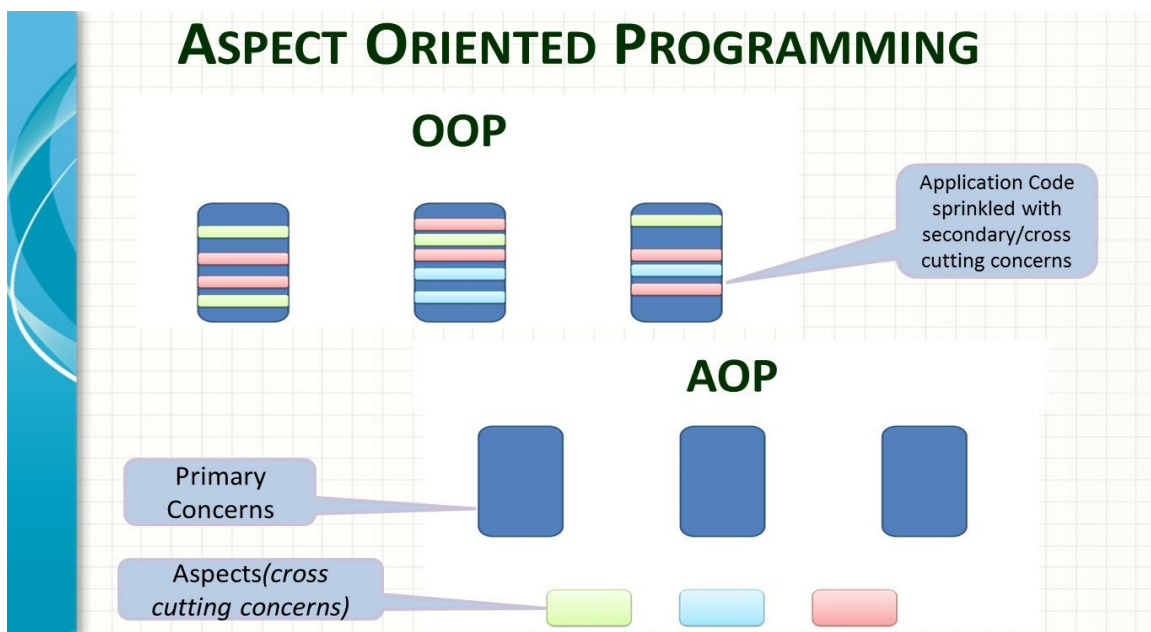


Figure 16. The difference between OOP and AOP

Another key component is the previously mentioned IoC Container, which is better explained in one of the following sections.

2.2.1 Architecture

One of the main properties of the Spring Framework is the modularity. This allows developers to pick and choose which modules are applicable to

project, without the necessity to bring in the rest of them. The framework is composed by 18 modules, illustrated in Fig. 17.

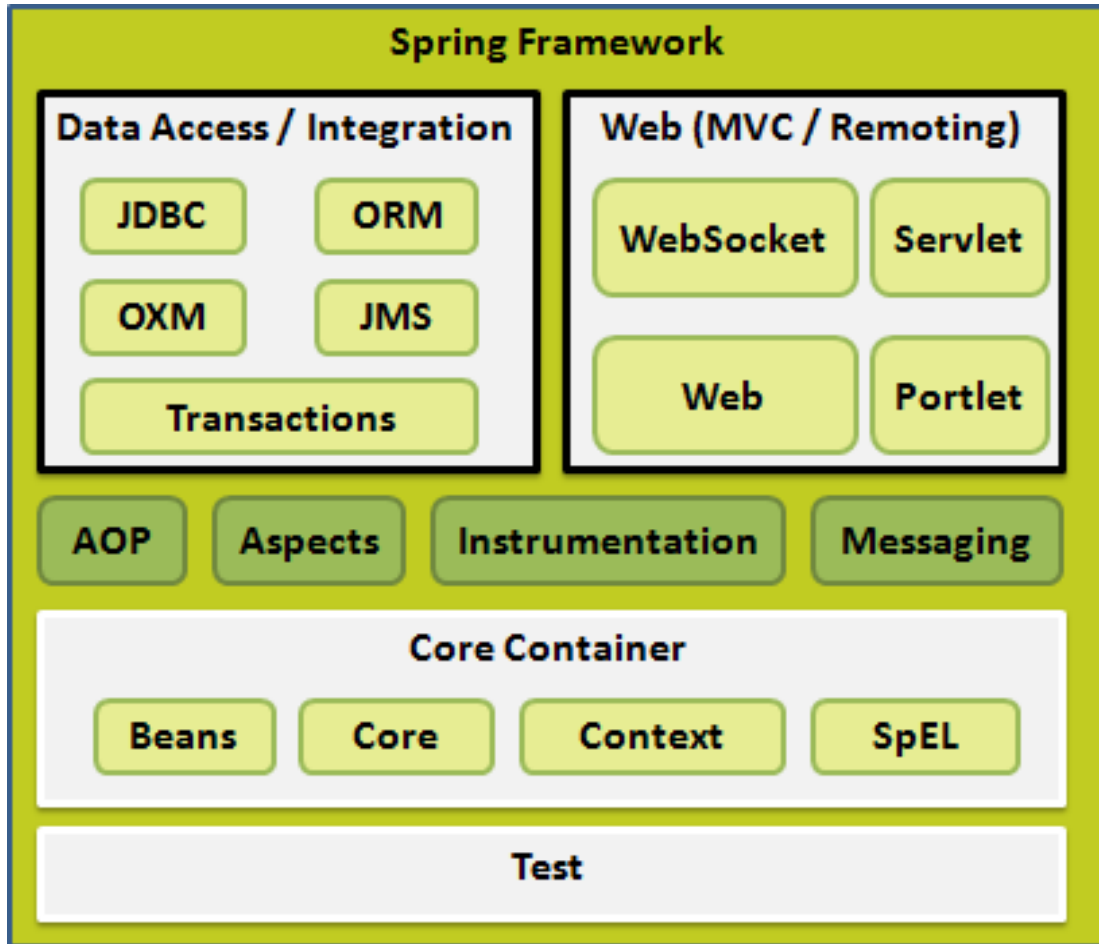


Figure 17. The Spring framework architecture

The heart of Spring Framework is the *Core* container that is the base for all the other modules, built on top of it. It provides the Dependency Injection feature and it contains the BeanFactory that creates and manages the lifecycle of the various application objects. Also, the Context module is built on the previous two and it is a middleware to access any defined object in the application.

The Spring DAO section contains the modules needed from the application to interact with its own database, creating abstractions over the low level tasks. It supports various types of components as JDBC, ORM (Object

Relational Mapping), OXM (that supports Object/XML mappings), JMS (Java Messaging Service) and the Transaction (that supports programmatic transactions for POJOs classes).

The web layer offers web oriented features to build a solid and flexible web application, such as multipart file-upload functionality and the implementation of the IoC container using servlet listeners. This module is built on the application context module that mainly provides enterprise level services; moreover, Spring offers a web implementation using the MVC approach, that will be explained in details in the following sections.

A key-module is the AOP module, that provides an Aspect-Oriented Programming implementation, helping in the implementation of the various crosscutting concerns in the application. These functionalities are decoupled from the application code and are injected into the point cuts through configuration file.

2.2.2 DI principle

The Spring container is the heart of the whole framework, because it creates objects, links them together, configures them and manages their complete lifecycle from creation until destruction. This component bases its own workflow on the *Dependency Injection (DI)* principle, which is a more specific application of the IoC. The IoC is an architectural principle born in the late 80s based on the will of inverting the traditional Control Flow.

In the traditional programming, the developer should define entirely this flow logic, whereas adopting the IoC this flow is set by a generic, reusable library. With this approach, differently from the traditional programming pattern, the reusable code calls into the custom and specific code, not the opposite.

This approach increases the modularity of the program, decoupling the execution of a task from the implementation, and it makes the entire code extensible, allowing adding new elements without changing the previous code. The Dependency Injection is a specific application of this principle by which the dependencies between classes are injected by an external entity (assembler) into the java class, the IoC container in the Spring case.

Inject a dependency into a class means to use one of the three following methods for links two classes:

- Constructor Injection : the dependency is injected through argument constructor
- Setter Injection : the dependency is injected through 'set' method
- Interface Injection : the dependency is injected through mapping between the interface and implementation

Spring framework uses the first two injection methods, indeed the container gets the needed information for instantiates, configures and assembles an object by reading the configuration metadata that can be represented either by an XML file, Java annotations or Java code. Adopting this principle allows to design a loosely coupled system, helping in gluing all the classes together and keeping them independent. The IoC Controller workflow is summarized in the Fig. 18.

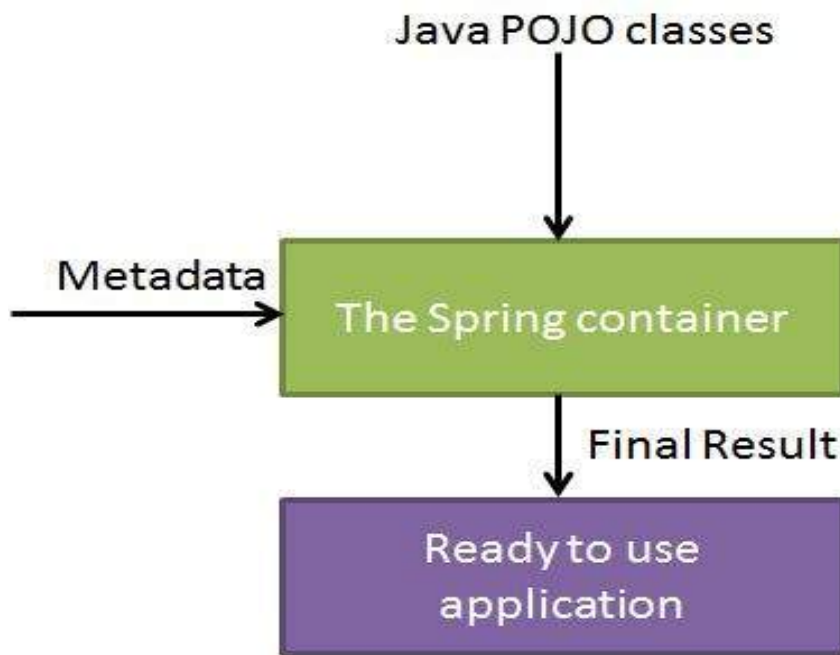


Figure 18. The IoC workflow

2.2.3 Spring Bean

The objects assembled by the IoC container that form the backbone of the Spring application are called beans, which are created by the pattern explained in the previous section. The configuration metadata used by the container contains the information used to build up the objects; especially the metadata declare how to create a bean, its lifecycle and its dependencies.

The most important field that is indeed mandatory is the 'class' field, which indicates what class must be used to create the bean.

2.2.4 Web

The Spring Web module provides a MVC architecture, allowing to create flexible web applications. This design architecture divides the model and the view, two entities that are generally merged in one in other architectures. This allows to adopt different develop techniques and to modify the code of one of the two components without impacting the other one. Theoretically, the Models are entities that allow accessing the necessities application data, while the Views are graphical interfaces that interact directly with the final user and show the data.

At last, the Controllers are entities that implement the web application logic, integrating the Models and the Views. A Controller receives the input from the user, manages the data searching and figures out which view is called by the user. In Spring, the Model elements are the POJO classes that represent the stored data, the View is represented by the HTML pages and the Controller are classes that have the responsibility of handling specific URL callings.

For implement this function, inside the controllers each declared function is linked to an URL and if that specific URL is called, the bounded function is called too.

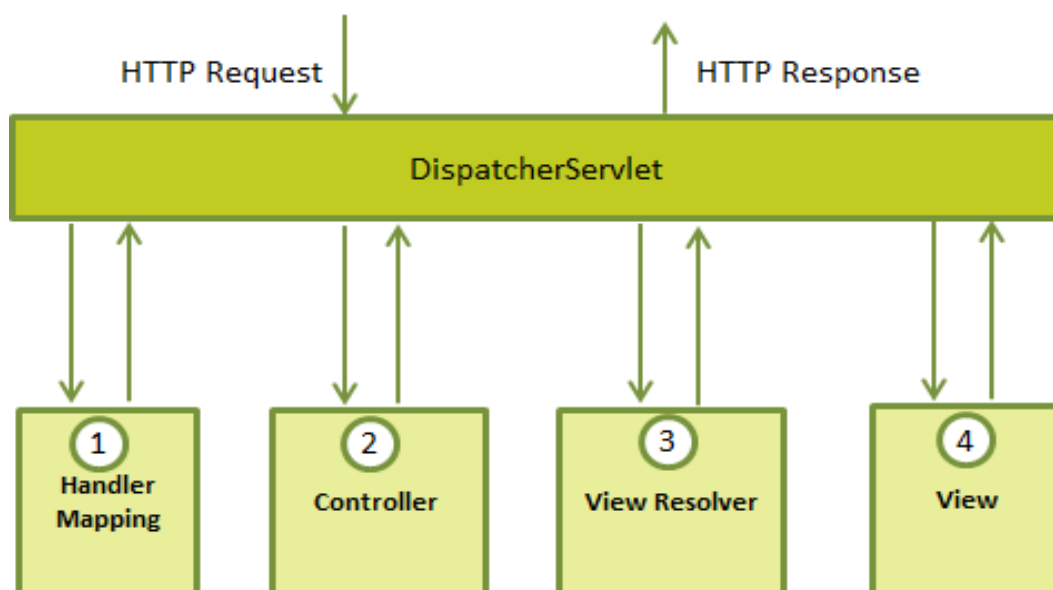


Figure 19 .The Spring MVC model

As it is showed in the Fig. 19 all the structure is designed around a DispatcherServlet that handles all the HTTP requests and responses. Indeed, after that the HTTP request is sent by the user and received by the Server, the DispatcherServlet consults the HandlerMapping, which is the component that is in charge of identifying the right controller to call for handling the incoming request.

When the right controller is called, it handles the request by calling one of the declared methods based on one HTTP method. The Controller will set the model data and it returns the view name to the DispatcherServlet, which get the appropriate view taking help from the View Resolver. Once view is finalized the DispatcherServlet passes the model data and the view to the user browser.

All of these components are parts of the WebApplicationContext module, an extension of the ApplicationContext one.

As showed in the Fig. 19, Spring MVC web module allows communicating with the various clients relying on the HTTP, the Hypertext Transfer Protocol. Through this protocol, Spring can map into the Controller methods that are called requesting a declared URL with a specific request method, that could be GET or POST.

In this way, the clients can call a method passing to the server as parameter the URL and the needed metadata to handle the request. The HTTP protocol offers a client-server communication on the port 80 and uses the TCP as transport protocol, opening a stream channel between the two parts. Getting into the details, the HTTP uses just one TCP connection: the client initiates an HTTP session by opening a TCP stream to the HTTP server. After this phase it sends the request messages to the server specifying the requested service.

At this point the server will answer to the request with a message containing the status of the server followed by the requested resources or an error. As shown in the Fig. 20, the HTTP communication protocol uses

just one TCP connection for send all the requested data: in a single stream multiple entities can be transmitted.

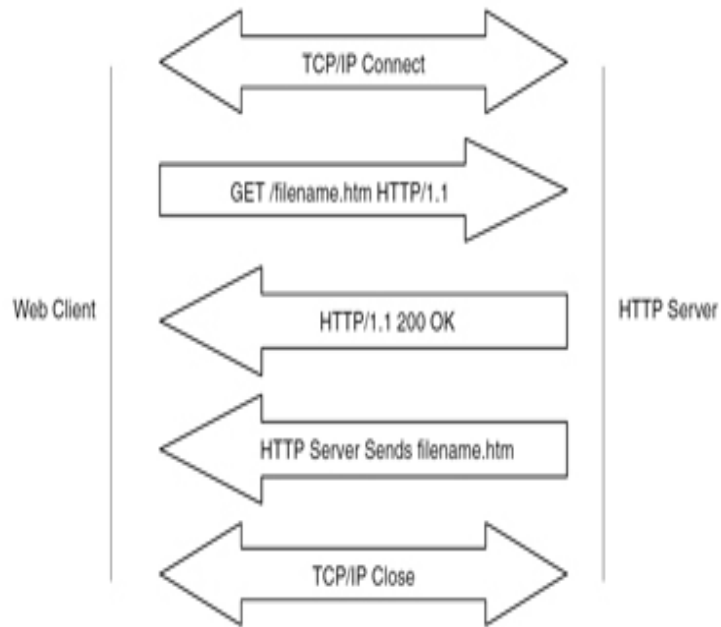


Figure 20. An HTTP connection model

2.2.5 RESTful communication

In this section I'm going to talk about the REST architectural style because in this thesis project I have used some of the key-principle of this development philosophy for communicate between the clients and the server. Representational State Transfer (REST) is the software architectural style of the World Wide Web introduced by Roy Fielding in the year 2000.

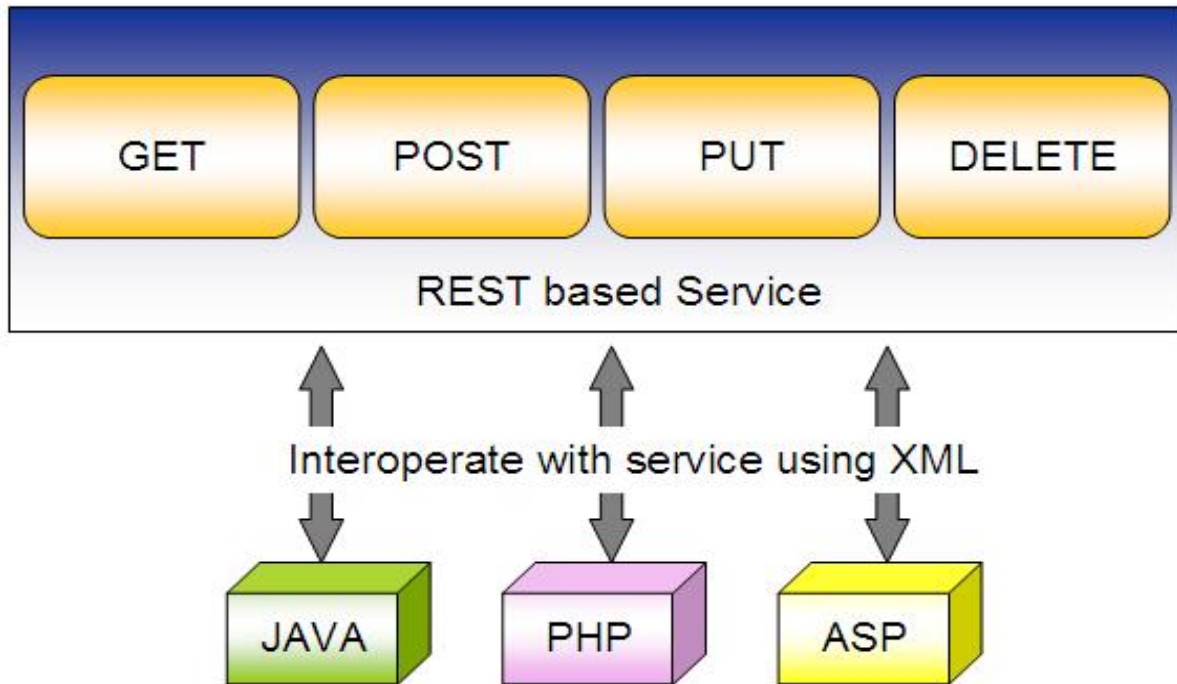


Figure 21. A simple description scheme that represent the REST communication

With the term RESTful we point at the systems that follows over the REST principles. communicate through the HTTP/HTTPS protocol and its methods.

The pillar principles of the REST philosophy are the following:

- Uniform interface
- Stateless interaction
- Cacheable
- Client-Server
- Layered System
- Code on demand (optional)

The main concept of this approach is the existence of resources that are accessible through an URI and the various components share the representation of these data. Usually this representation is done by language independent data formats, as JSON or XML. The differences between them are many. As first, XML is a mark-up language as the HTML, while JSON is born just to represent objects as strings. Moreover, JSON is lightweight and takes fewer characters to transmit the same information contained into an XML document. For example:

JSON (133 characters)

```
{
  "id": 32,
  "title": "Android Development",
  "author": "Paul Deitel",
  "published": {
    "by": "Pearson",
    "year": 2007  }
}
```

XML (189 characters)

```
<?xml version="1.0"?>
<book id="re">
  <title> Android Development </title>
  <author> Paul Deitel </author>
  <published>
    <by> Pearson </by>
    <year>2007</year>
  </published>
</book>
```

This characteristic makes the JSON transmission faster. Moreover, it has been discovered that JSON files are serialized and de-serialized faster and use a less amount of CPU resources. On the other hand, XML is not a data format, it's a language. Indeed, it is possible to add metadata into the tags as attributes, which is not so comfortable to develop using JSON. It is preferred from JSON when complex data structures come to play due to its tree structure.

2.3 Open Street Map (OSM)

One of the most important decision regards the choice of the map provider, which conditions the choice of the libraries and the choice of the APIs to interrogate for the various requests. The platform chosen is *OpenStreetMap (OSM)*, which is a collaborative project with the goal to create a free editable map on the world. Indeed, the map data are collected by volunteers performing systematic ground surveys using tools such as GPS units, camera or notebooks, for then uploading the recovered data in the OSM database.

The principle behind this map provider is the same of my thesis project, the crowdsourcing, demonstrating that it is a winning approach. The OSM project recently has switched from a Creative Commons license to an Open Database Lince, which is a share-alike license, allowing OSM to be shared and used as long as all of the data uploaded from the members are made available to all the other users. Indeed, OSM was created in 2004 by Steve Coast and from an update of the 2013, the registered users nowadays are estimated as 1 million.

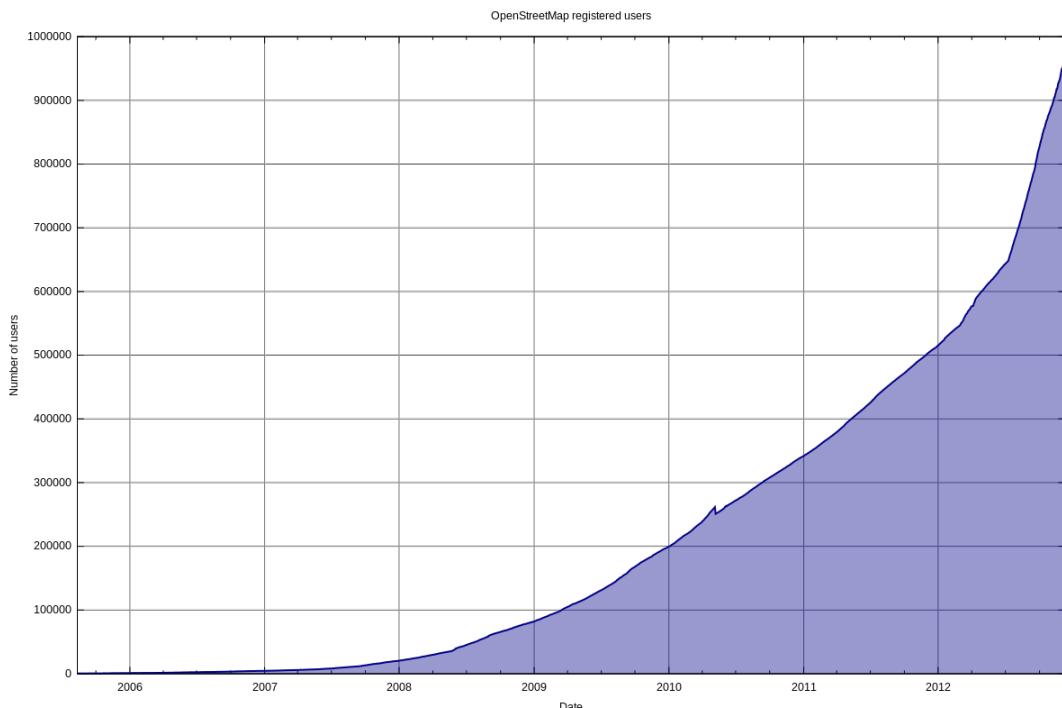


Figure 22. The amount of registered users to OSM from 2006

OSM uses a topological data structure composed by four main elements:

- Nodes: points with a geographic position and stored as coordinates, used to represent the OSM POI or elements without size
- Ways: lists of nodes that represent a polyline or a polygon used for represent streets, parks or areas in general
- Relations: lists of elements in which each member is linked to the others by a role. These components are used for representing existing relations between OSM elements.
- Tags: are key-value pairs of strings that are used to store metadata about the map objects, for example the belonging area, the kind of entity represented, etc...

Moreover, each node is characterized by its own ID number: this information will be useful for implementing the route calculation. With the correct usage of the described elements is possible to retrieve the needed data. For this map project, the main elements to focus on are the Nodes and the Tags.

Indeed, the POI saved in OSM are represented as Nodes with particular tags, so for retrieve the needed information its necessary a query that focuses on the common factor of all of them, which in the case is the location area. To access to the OSM database I choose to use the Overpass API , a read-only API that serves up custom selected parts of the OSM map data.

It is characterised by an intuitive scripting language for ask the data and an enormous pro of this API, in my opinion, is that you can test the queries in real-time on a specific website, called Overpass Turbo. For facing the localization issues, OSM relies on *Nominatim*, which is a tool to search OSM data, or through the geocoding or through the reverse geocoding.

This tool is accessible via an API, to retrieve information during the running time of the application. Also, OSM provides a large number of different tiles, offering a large number of variants for the developers: some tiles are specific for transports, other for streets, other are all in black and white, et

Making a comparison with Google maps, the main difference is that Google is a closed system that cannot follow the OSM rhythms, considering the updates. The Fig. 23 explains the collaborative concept that stands behind the OSM project.

Indeed, following the crowdsourcing principles, OSM is hugely more powerful than Google Maps, because it relies on the people contribution that is an enormous number if compared with the Google workers. Because for this project the main purpose is to have a dynamic system that tries to offer real time contents, the most eligible map provider is OSM.



Figure 23. A painting representing the OSM crowdfunding principle

2.4 Persistence tools

In order to adopt a high level approach for managing the database I have selected the Hibernate platform. Indeed, the development of the web applications and in general of the information technologies brings to development models that take advantage of the Javabeans, as for example Spring.

Hibernate is one of the most solid persistence middleware open source service for the development of Java applications, which is based on the relative framework. This platform offers an *Object-Relational Mapping (ORM)* service, allowing to map an object-oriented domain model to a relational database.

The main idea is to declare the association between the javabean class and the relative table through a tool as descriptors or XML and, based on this set of information, the infrastructure will set up dynamically the entire data model. Hibernate moreover allows to the developer to automate the *CRUD(Create,Read,Update,Delete)* procedures of the databases. The Fig. 24 shows clearly the collocation of Hibernate into a structured application.

This tool is totally compatible with the Spring framework, inserting itself into the ORM module, allowing to define for each table a corresponding class. In this project there is a last platform that is based on the ORM approach offered by Hibernate: the Java Persistence API(JPA).

The JPA is a collection of classes and methods to persistently store a vast amount of data into a database that allows an easy interaction with the database instance. Nowadays the current version is the JPA 2.1 and it has supported by a vast number of platforms, including Hibernate. The main functionalities of the JPA infrastructure are showed in the Fig. 25, that represents the different JPA layers.

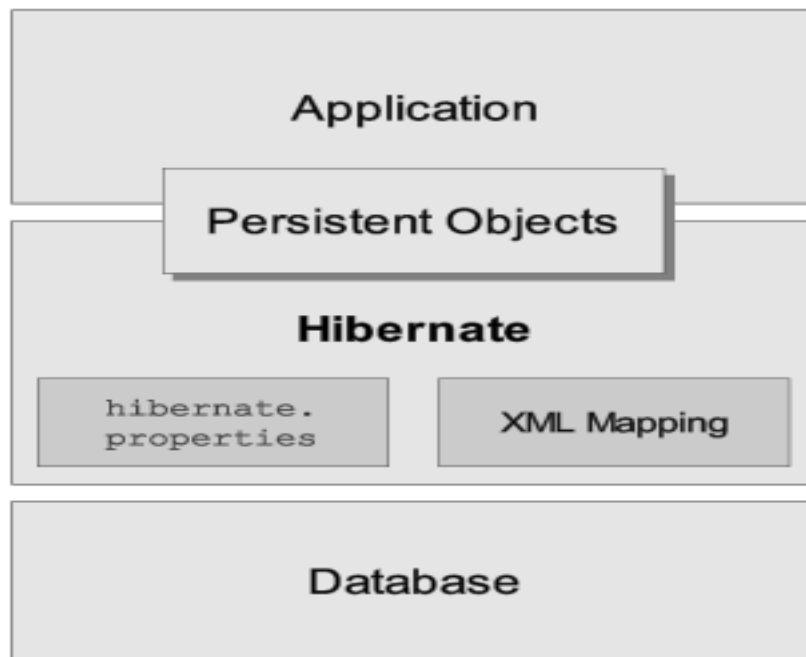


Figure 24. The Hibernate collocation inside a complex architecture

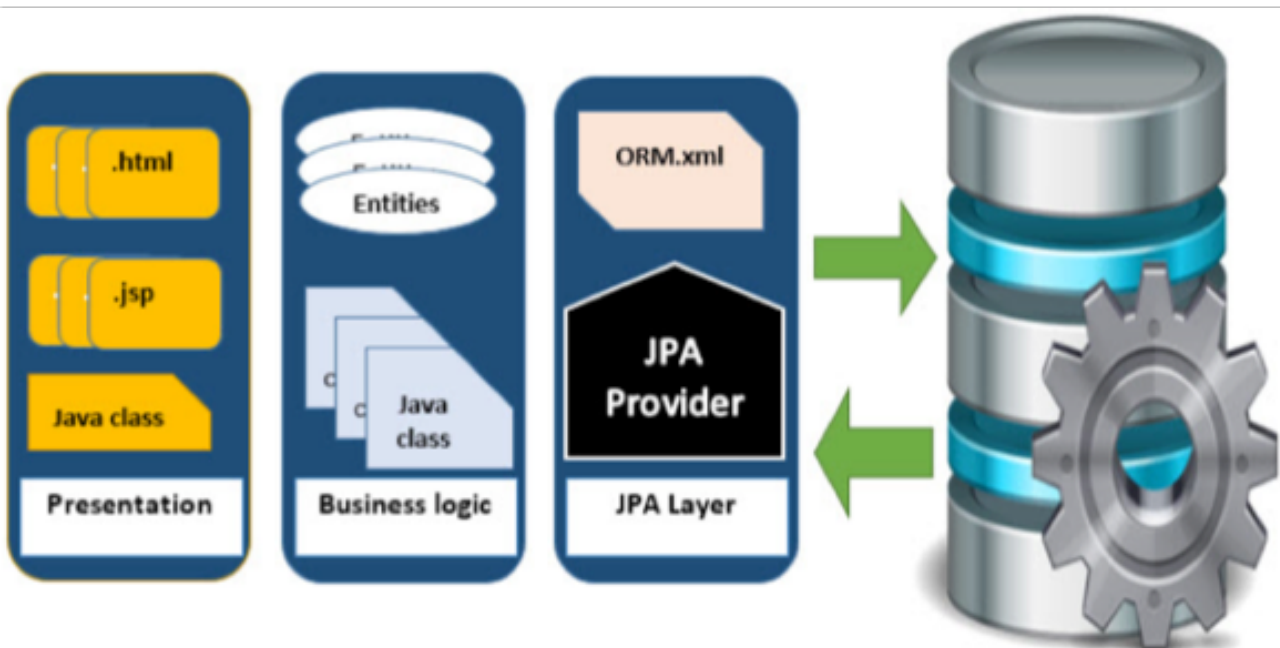


Figure 25. A scheme of a server based on JPA

The JPA architecture is based on a PersistenceContext, which can be identified by a cache, which is directly linked with the database. On the

cache is possible to modify the files and the flush operation of it propagates these changes to the persistence level. The entity that handles all the database operation is called *EntityManager* and it should be just one at project, avoiding lacks caused by a missing synchronization between the various entities.

Every entity instance of an object is characterized by a state and there are four different states, as shown in Fig. 26 :

- New or Transient : an entity has added to the Java memory
- Managed : an entity has become persistent
- Detached : when an entity is already stored in the database and is not already present in the cache
- Removed : when an entity is removed from the database

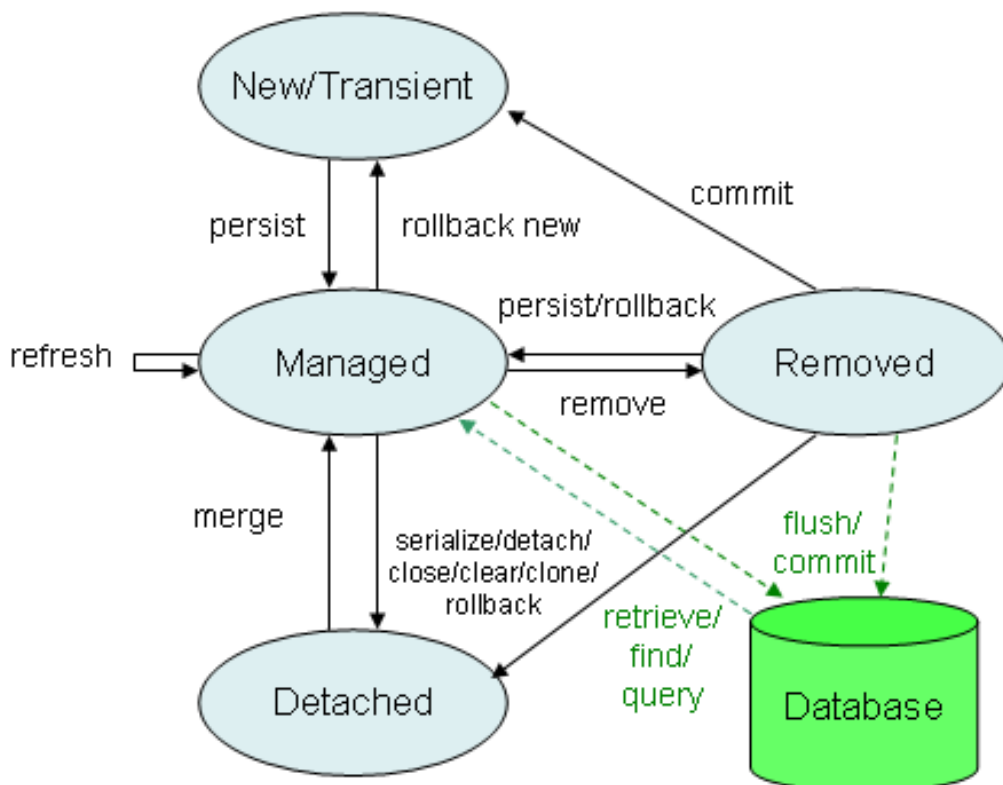


Figure 26. Entity states scheme in JPA

3. Impaired Mobility App

This section explains which are the functionalities needed to the project, which tools have been adopted and their role. Indeed, this communication system requires a lot of tools for sharing data frequently between the client parts and the server, which has to store all the received information into the database.

As it is showed in the Fig. 27, the entire system project contains three entities that must handle different type of communication, between them and to other kind of resources, as external APIs or internal tables. As we can notice there are two different devices that communicate with the server: a mobile device and a personal computer one. So, the server must implement two different listeners, one for each kind of client type.

Moreover, there are two main entities that will be the keys of the application: the POI, which could include all the private or public structures accessible to people, and the *Barrier* one, which contains all the architectonical obstacles that can represent a huge problem for a person with mobility problems. All the entities will be stored into a unique database and can be added from the users or from the system itself that will download them by an external call. In order to store them into the database there is the need to define the POJOs classes for both of them and the relative fields.

The system will allow to the users to add both type of entities into the database through specific functionalities, but, in order to avoid unnecessary work for the final users, it will at first download all the POI entities already stored into the map provider database. This will be used to hold the system updated too, checking at some time intervals if new entities have been added.

Both entities will contains various data fields, as text or images, in which some of them will be mandatory while others not. Both clients must send and receive both POI and Barriers objects and for doing that I will implement entities that can handle asynchronous transactions, both on the web client and on the mobile one.

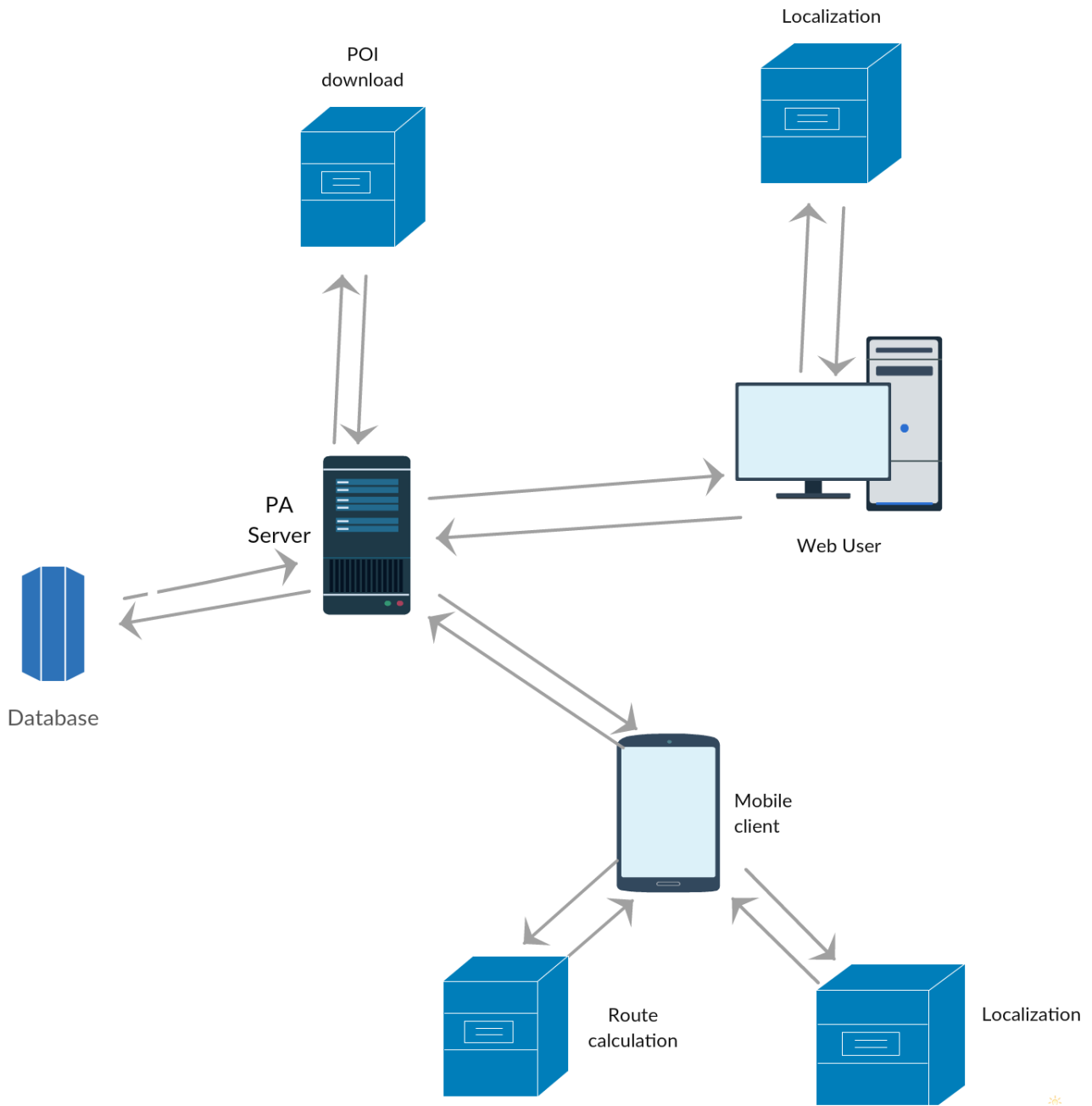


Figure 27. Thesis project diagram

Both clients need to access to the information stored into the database in real-time, allowing to the end-user their changes on act. They have also to implement location functionalities, in order to significantly decrease the amount of spent resource on the server machine, retrieving just the entities belonging to the current area region. Indeed, the database will contain entities belonging different cities and countries, so the system logic

must search just for a contained amount of them at every call, limiting the resources costs.

The Android application also needs to implement a routing calculation that track down a pedestrian route calculated trying to avoid the obstacles present between the starting point and the destination one. Considering the available resources on the mobile device, making this operation on-board could compromise the entire workflow. This means that the application will ask to an external API to calculate this route and so it must handle an external resource request.

Moreover, another key-point is the dynamicity of the final interface, which has to guarantee a dynamic interaction between the clients and the stored dataset, representing the contained information clearly and faithfully. In order to retrieve this goal, both clients must use solid and light libraries to build and customize the map view.

There is also the need to cover a large set of possible actions done by the users: for example if the user will zoom on an area that have not been searched before the system must retrieve the POI already present into the map provider database, avoiding to show an empty map.

3.1 Client/Server Model

The model that is at the base of this project is the Client-Server architecture. As the name suggests, there are two main subjects that interact among them: the server and the client, as represented in Fig. 28.

The server, also called daemon, is a device that offers services to other entities across a network. It is called daemon because it is a component that never stops working and it must be ready for answering each type of manageable request, whenever it is sent. For these reasons, usually as server we mean a computer with high available resources and high reliability and that is totally dedicated on offering services to the connected users.

The entity that sends the requests to the server is called client that is an external device connected to the same network of the server and that ask at least one of the services offered by the server. The client device can be represented by any type of device that can access to the network, such as a personal computer or a mobile device.

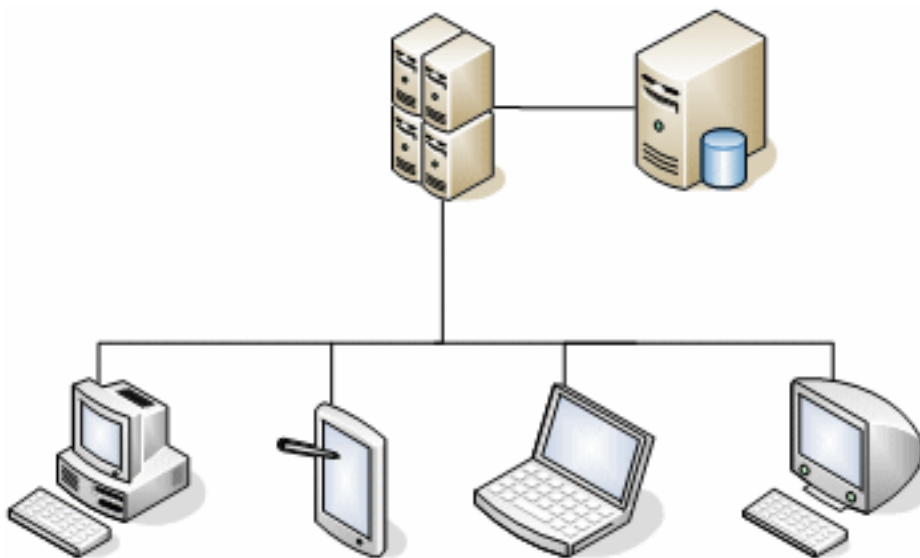


Figure 28. A brief representation of the C/S architecture

There are different types of Server possible usages:

- File Server: this kind of Servers allows to access to the files stored on the server machine as they are on the client device. This is possible if the server machine share with the users a portion of its hard disk space
- Database Server: this kind of Servers stores a large amount of data regarding a common topic or at least linked between them. It structures the manage of these data using a database infrastructure
- Web Server: a web server is a server that provides a web page as answer to a client request. The information sent by the web server are carried by the HTTP protocol and it is part of the Wold Wide Web.
- Application Server: this kind of Servers is used to run a program on the server-side and to share the functionalities with the clients.

The server developed in this project will cover two of the explained role: it will be a database server because it will contain all the data into a local database and it will implement a webpage, allowing to interact with it through the WWW service.

The whole communication logic between the various components of the application will be a complex net: indeed, the clients must communicate with the server every time they starts, in order to communicate their actual position and every time that they need to upload a set of information. These calls will be light and fast, indeed one of the main goals is to manage the needed data through algorithms that does not exceed the resources required.

In order to develop this feature must be considered the main point by which the web application cannot receive or send classes entities, but exchanges these object under a representative language as XML or JSON, while the Android application can send and receive entire classes because both sides are implemented through Java language.

3.1.1 Client functions

As first I would like to illustrate the required functionalities of the user interfaces because on these requirements is build the entire system logic. The user interface is composed by two parts, one running on an Android device and the other one running on a browser, implemented by the server. Both of them need to show a map, with a specific tile and information inside.

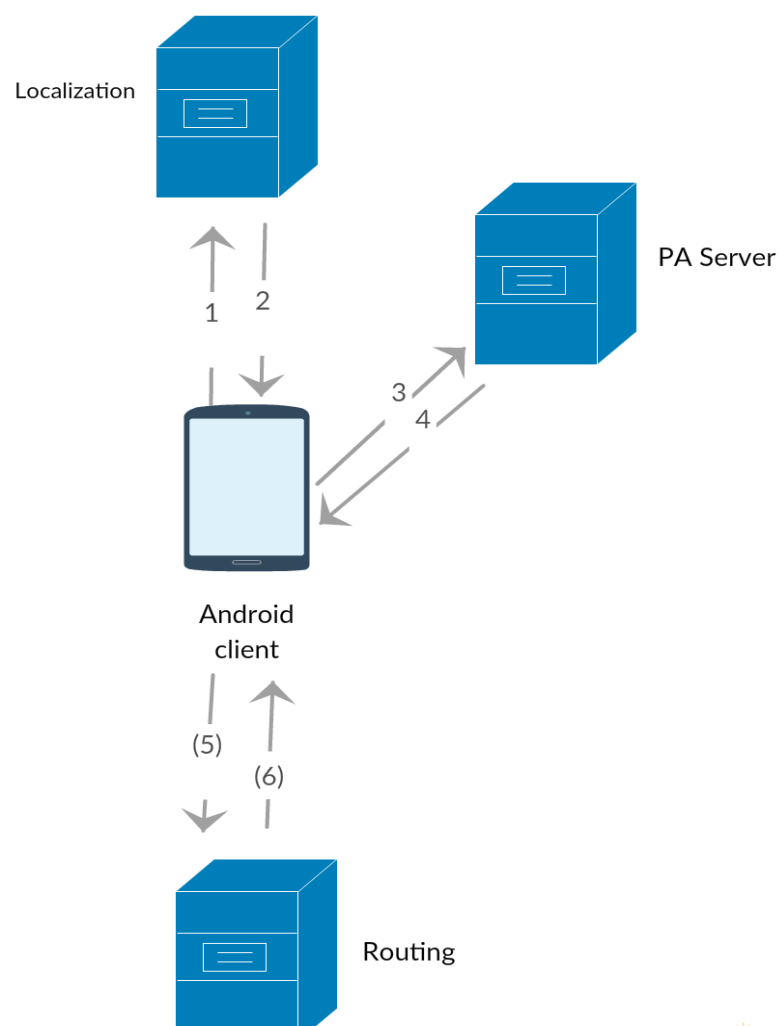


Figure 29. The mobile interactions diagram

Both Fig. 29, 30 shows briefly the clients logic. As it could be noticed, the client must as first locate the user, understanding which portion of data it must retrieve from the server. For each location the system has to store all the entities of interest, as public or private structures and architectonic

barriers, belonging to that specific area. For this purpose both the client parts have to find the actual city area, limiting the amount of exchanged data. Without adopting this precaution the system could send to the client all the stored data every time, causing a useless consumption of resources both on server and on client side.

Moreover, both clients have to reorganize the received data and the communication should be faster as possible for allowing to the user a fluent experience. Both of the interfaces need to communicate with the server for obtaining the needed data and the tools used will be different. The two parts have mainly differences: the web side will be implemented in a scripting language while the mobile side will be implemented in Android, a compiled programming language.

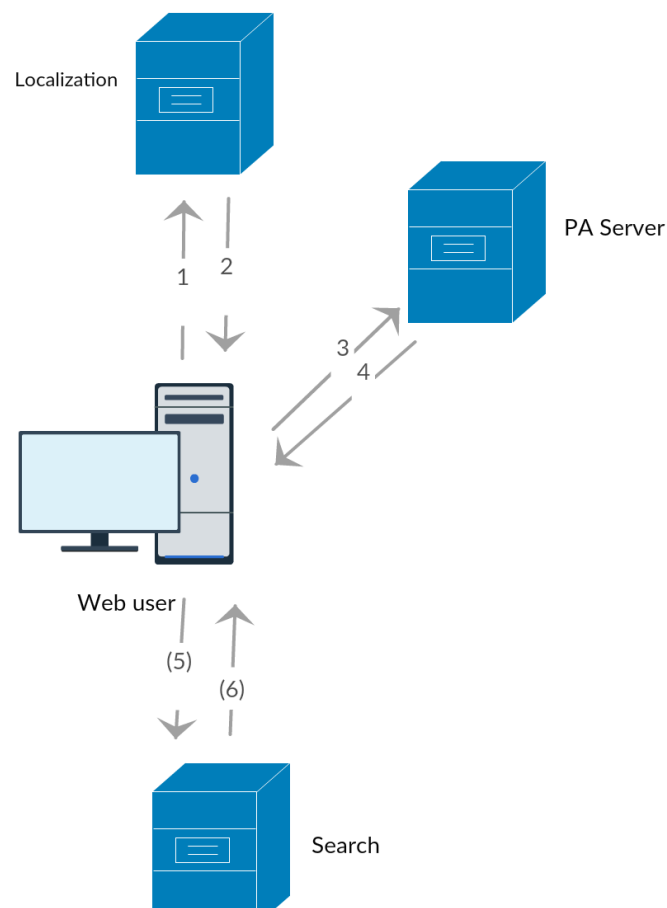


Figure 30. The web application interaction diagram

Furthermore, as I introduced before, the user experience should be as fluid as possible, considering that each entity is categorized and characterised

by a different icon and that each one will be probably complex, containing different kind of data. So both clients should handle the computing of at least 2000 elements that must be dynamically visualized on the map.

Moreover, the user interfaces need to implement a part that allows the consumer to upload a comment to an existing entity or inserting a new one. This feature will be developed by creating two different spaces, one for each type of object that will be edited, a POI or a Barrier.

At last, the Android client must implement one more service, consisting into a route planning that will bring the user to the declared destination avoiding as much architectural barriers as possible that will be implemented by an external call to an API. The client will then retrieve the output set of data and represent it on the visualized map.

For all of these purposes, is needed a dynamic and flexible platform that can handle all the necessities variants. The clients must send requests to external APIs and to the server, manipulating the received data for obtain the information needed and then prepare the context for the final user, offering a useful service.

3.1.2 Server functions

The complementary part of the previous side is composed by the Server functions. All the webpage behaviour described in the previous section will be implemented on the server side, but will run on the client machine, once the webpage has been downloaded to the user's browser.

The Server must also handles different kind of communications, both with the connected clients and both with eventually external APIs. As shown in the Fig. 31 its main duty is to receive the requests by the clients and to compute them, recovering the requested data or from the database or from

the web, in some rare cases. It must implement all the methods needed at various abstraction layers for allowing to retrieve or modifying data stored in the database.

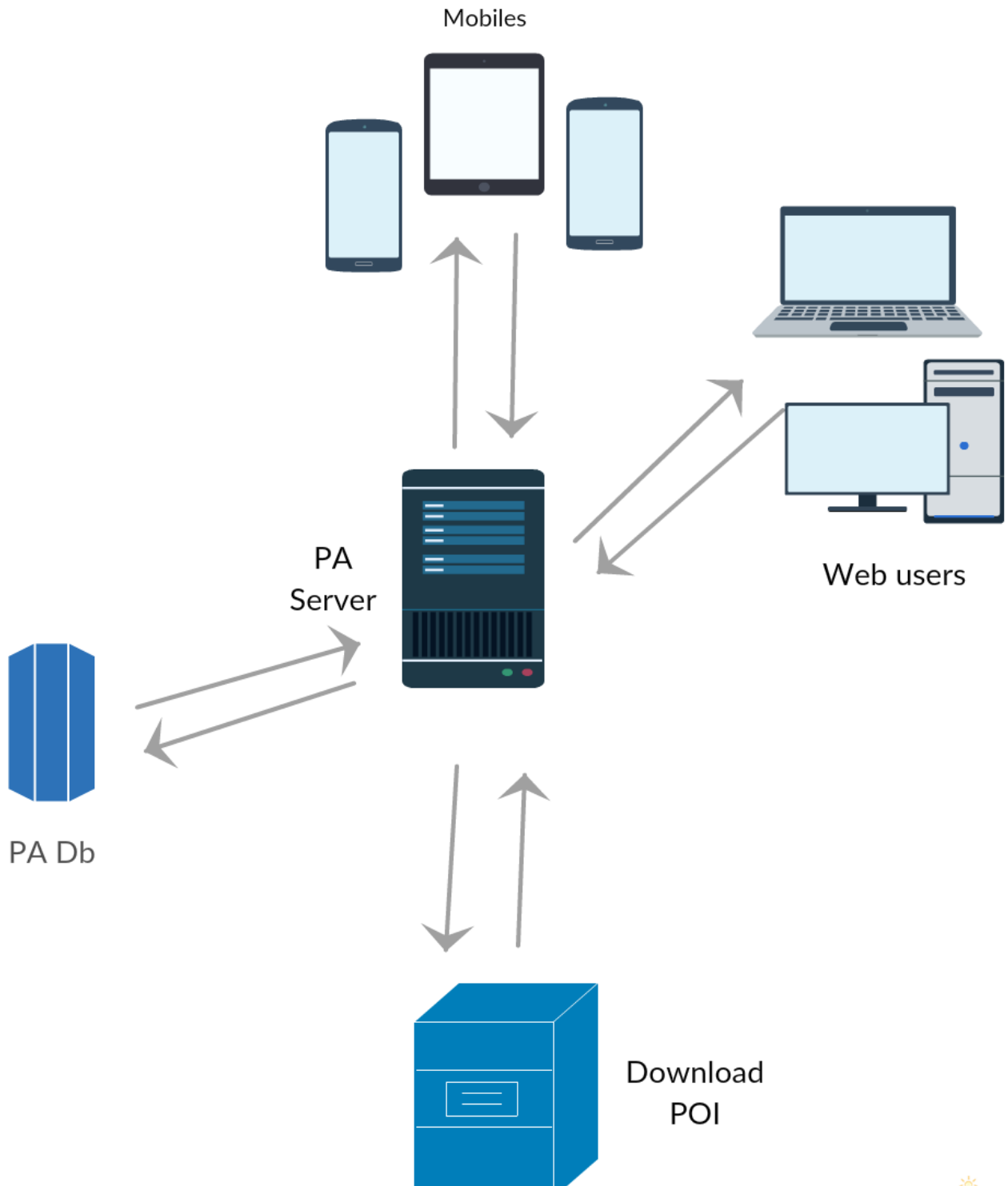


Figure 31. Server interactions schema

It will also need to handle an administrator section in which the admin can launch the search algorithm for update the database, because the consulted database receives updates very shortly. This section must allow the administrator of the website to constantly check the current condition of the database, displaying all the stored elements. There must be the possibility to delete them too.

In the case the user's browser visualize a location not inserted yet into the database, the client will send a request to the server asking to download the needed data to visualize the entities of interested of the focused location, avoiding the visualisation of a void map. Furthermore, the dynamic composition of the webpage has to guarantee some kind of synchrony, because if the user uploads a new data on the database it must be immediately visible, at least for that single consumer.

All the data displayed in the map must represent accurately the information stored into the database. Moreover, the server has to send the data to the user in the best way for it to process them, choosing the most eligible option. For this reason, the data organisation should be as simpler as possible but clear and efficient.

The last important issue to consider is the database organisation and how the data will be stored. For ensuring the needed flexibility a high level approach is needed, with the possibility to create data type classes, directly stored into the database. This allows to the developer to see the database as a 'box of classes', impossible if considering the data storing from a low abstraction level.

3.2 Entities of interest

The map must display correctly located entities using *markers*. Each marker should be different depending on the category of the entity to represent and on its average vote. The main purpose of the map is shows the city under the impaired mobility prospective, suggesting the best places to go in order to live a comfortable experience and which ones avoid.

All the information regarding the entities is contained into the server database and just a few of them are showed to the user. The most important thing is that the retrieved information is reliable, complete and compatible with the server software, which will re-elaborate and store them.




In this project there are two main entities at the base of the entire idea, which represent the main data traffic that have been introduced in the overview section of this chapter. Both entities can be directly created by the users or downloaded as first from the map platform API, reorganized and then stored. They represent the only entities shown on the map.

3.2.1 POI

A POI corresponds any public or private structure that has a particular role into the environment. For this project, this term points at all the structures that can host the presence, even if for a short time, of a person with impaired mobility. It could correspond to a shop, a public place, a museum or whatever kind of structure a person can think about: the Fig. 32 tries to explain the POI concept by a set of icons that symbolically represent various kind of different structures that have various roles into a social environment.



Figure 32. Some POI icons

Considered POI	New	Not adapted	Parcial adapted	Adapted
Adapted access?	★★★★	☆★★★	☆☆★★	☆☆☆☆
Adapted internal space?	★★★★	☆★★★	☆☆★★	☆☆☆☆
Adapted bathroom?	★★★★	☆★★★	☆☆★★	☆☆☆☆
Reserved parking space?	★★★	☆☆	N/A	☆☆
Color of the icon				

Each entity must be subject of a mass evaluation by the users considering some main criteria, basing on the crowdsourcing logic: you offer to the people a service and this will be evaluated by the crowd, which upload their own opinion and ideas to a central storage. The criteria are the ones contained into the table above, that summarise the outline of the entities showed into the map.

Basing on the average of the users' votes each POI has assigned an icon that indicates its membership. To allow this, every entity has its own history, composed by users' opinions permanently stored into the database as text or numbers. In particular every POI entity has a set of comments or images that belongs to it, over a location, a name and a specific category.

Indeed, all the structures must be contained in categories that subdivide all the POI in smaller groups containing places with similar characteristics, as one for bars and pubs, one for restaurants and other feed structures, etc. The number of these categories shouldn't be too high, in order to preserve simplicity and clearness

3.2.2 Barriers

The second entity of interest is the Barrier, which can regroup a large group of architectonic obstacles, as scaffolding, street blocks and every kind of physical object that can represent a mobility problem for people with impaired mobility.



Figure 33. A graphic explanation of the impaired mobility problem

The Fig. 33 represents a concrete application of the problem that this platform try to fix: in the illustrated situation a person with mobility problems can only try to find another route to reach his/her destination.

Using the routing calculation on the android application helps avoiding these kinds of sets back. Anyway, some kind of barriers could be adapted for the passage of people with impaired mobility: our system have to help identifying in which category a Barrier belongs to, based on its accessibility. A pillar of stairs as the one in above image represents an obstacle with the lowest accessibility because it does not help the person in any way to pass the obstacle. In the option by which on these stairs there is installed an elevator for people with impaired mobility, it would have represented a barrier with a positive accessibility score.

It must also be possible to upload a set of photos for each barrier, in order to allow the possibility for the users to personally consider the option to try to pass through or try to change route.

3.3 User interfaces

This section explains how the user interfaces are organized. This part is really important because software that does not provide an intuitive interface will be probably avoided from the consumers or at least cannot guarantee a positive experience.

Even if there are not instructions on how to use the interface, getting through the application functionalities must be an easy task for everyone who needs to use it.

3.3.1 Android application

The Android application needs to implement all the functionalities explained in the previous section. So, as first, it needs to implement a view representing the map of the current location of the user.

This map also need a 'my location' button that will set the map focus on the location of the current user. This client part is more focused on a local prospective, allowing the user to visualize by default the current city area. Anyway, he/she will have the possibility to change visualized location.

It also needs a filter space, to allow the user to select which category he/she wants to visualize on the map, offering a more flexible view. Moreover, the map view have to be the base for any route searching, that will draw the route line on the current map, changing so the focus point. For make this feature possible, there is the need to let to the user two blank spaces in which he/she can declare the starting route point and the destination point, typing them.

The map needs to hold fixed view simplicity, avoiding to show too many drawing elements that can potentially create confusion. There must be a view dedicated entirely to the insert of a new entity into the database because the Android screen is not so large, especially on phones. By this, put all these features in the same view will create a chaotic view.

So, because the entities that can be added are two different (POI and Barriers), the Android application will be composed by two different views, one for each type of data to insert/modify. For both, the user must have the possibility to choose the location of the new entity precisely and in a comfortable way for the final costumer. For resolve this issue, probably another map is the best solution, but there must be found a way to insert it without increase too much the resource consumption.

Another issue is the image resizing: there must be find a way to reduce as much as possible the file size without losing the image quality, in order to

limit the spent resource but maintaining a minimum detail quality allowing to show them both on mobile client both on the web part.

3.3.2 Web page

The web interface must support all the functions declared in the previous section and, as I said for the Android interface, this have to be as much intuitive as possible, allowing an easy interaction for the final costumer.

As first, the interface must be composed by a map, which needs to be fluent and dynamic for supporting all the possible changes. Moreover, this map has to allow the user to search for other areas, changing the view on other cities. For support this feature, there must be a space that allows the customer to write the searched area that will be next visualized by the map.

Also, despite the application, the webpage will list all the entities that belong to the selected area, dividing them in two lists: one for the Barriers and another for the POI. This feature brings the necessity of create two sections on the page in which these lists are shown dynamically.

As the Android application, the web interface need to implements a filter space allowing the user to select which category visualize despite others. The icons representing the various categories of entities remains the same of the Android application, in order to avoid confusion for users that use both the interfaces.

There is also the need of a 'my location' button, that will change the current focus of the map on the measured position. At last, there is the need of two sections for insert or update the entities, such as in the Android application. In order to avoid a chaotic interface, the inserts sections will be divided as in the Android project: one section will be used for the POIs

and the other for the Barriers . As in the mobile application, the images loaded by the users must be resized in order to save as much space as possible.

3.4 Localization

The localization covers a key- role in both the applications: on the localization data is based the entire logic for choosing the entities to retrieve from the server. Moreover, two different operations are needed: the geocoding and the reverse-geocoding.

The localization services will be different between the two clients: the android application has indeed a *Geocoder* class that is provided by the OS itself, while the browser has to lean on external geocoding services. These location services will be called to retrieve the needed information regarding a particular location, that will be declared using a text or a set of coordinates.

The geocoding operation uses the description of a location, such as the address, to find the geographic coordinates, while the reverse-geocoding uses the geographic coordinates to find a description of the location.

3.5 Communication

The communication has to be reliable and should not represent a big usage of resources, allowing a fluid and fast communication model. The communication tools have to support different type of transfer size,

because various amounts of data can be requested or uploaded by the clients and the system have to handle both situations.

Because the clients are composed by two different parts, the server have to differentiate the requests originated by the mobile devices by the ones sent from the client browsers, allowing an adapted communication logic. Furthermore, the communication model has to handle the transmission of different type of data, as text, numbers or photos.

3.6 Plan a safe route

On the Android client there is the possibility to calculate a 'safe route'. This operation will be done leaning on an external service accessible through API methods. Indeed, make this kind of calculus on board for a mobile device would represent an higher resource consumption that could not guarantee the fluent execution of the application. For this reason, the android client will ask to an external server to make this operation for then using the final result.

This service will allow calculating a pedestrian route that tries to avoid the Barriers entities that could be present on the path, planning a route that is accessible to people with impaired mobility. For implement this functionality is necessary to retrieve the correct location of the Barrier entity and ask to the API to do not calculate that nodes into the path calculation. This needs an high precision regarding the coordinates related to the entities.

The route will be drawn on the mobile map and will be composed by markers indicating the route directions. For avoiding an element overlap, the line route must be of different colours, from all the other elements, same for the relative markers.

4. App and Server Design

This section contains a precise introduction and explanation of all the tools used to satisfy the requirements displayed in the previous sections. Every choice made tried to respect the introduced goals, focusing on the flexibility of the code and allowing a fluid experience to the final customer, trying to alleviate the impact that a large amount of data can have on an application.

This section will not contain any piece of code or implementation that will be showed in the next chapter.

4.1 Employed technologies and tools

This thesis project is implemented directly into the ParticipAct platform, adding the explained functionalities to the previous project set of services. ParticipAct is a platform developed by the University of Bologna that collects different sensor data to store information in order to develop a crowdsourcing platform that will help to create smart services following the ‘smart cities’ principles.

This thesis project is built on the already present architecture that is showed in the Fig. 34. For this reason, a big part of the used technologies has been chosen in order to preserve the original structure of the project, getting through its logic.

Both the clients will base their map retrieves by the OSM infrastructure, chosen for its base principles and its characteristic to contain up-to-date information. This is totally coherent with the project principles.

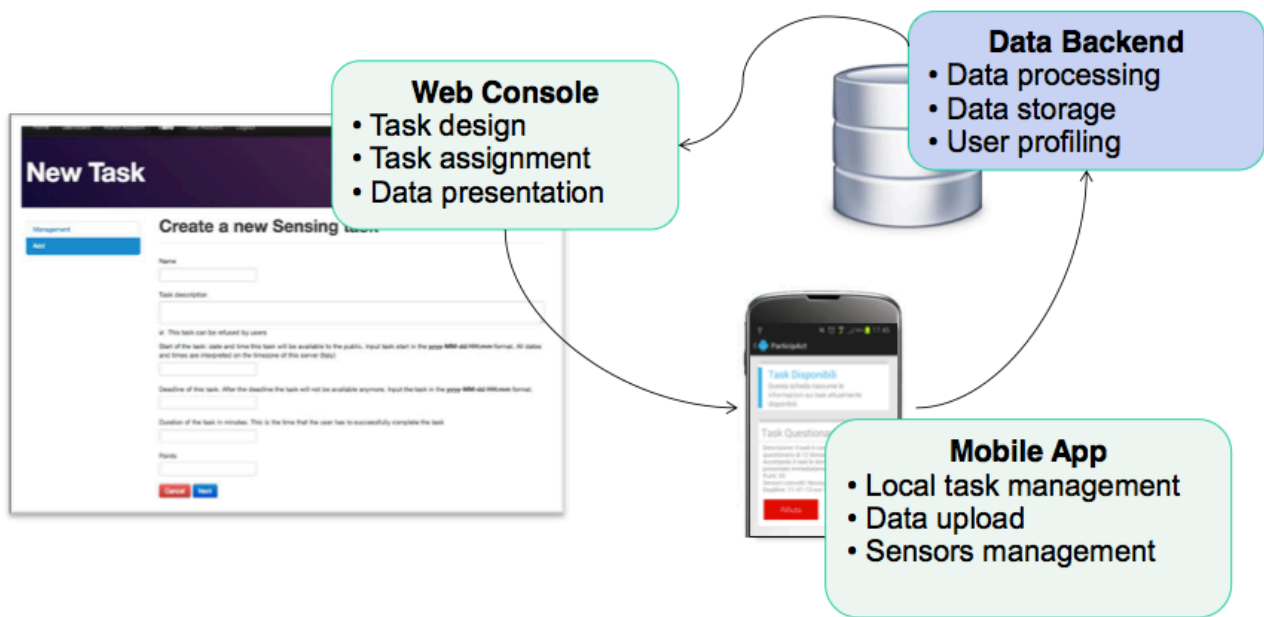


Figure 34. Participact environment

4.1.1 Server side

As first, I will introduce the design made on the Server side because on the following decisions depends the entire transmission model with the clients. For creating a web server, one of the most flexible platforms is Spring framework, explained in details in the previous section 3.2. This framework has been chosen for its flexibility and lightweight and had have been already in the previous ParticipAct version. Based on this consideration, all the server side has been developed on that framework.

Moreover, on the server side must be implemented the needed classes for access the database data, done by dividing the roles in three different levels: a *repository* level, which is the lowest and that directly implement the database access, a *service* level, which is in the middle between the repository level and the higher-abstraction level, and an *implementation* level, used by the developer for calling the service.

The server must implement the web interface that will not be run on the server machine, but on the client device, after the downloading. For this reason, the structure and the graphical part of the webpage will not influence the server resources, but the client ones.

The persistence of data on the server will be covered in the following sections.

4.1.2 Client side

Considering the server decision and the general decision of use OSM as map provider both client parts will be affected, adopting some libraries specific of OSM instead of others. Moreover, the communication will be based on the HTTP protocol, or the Server cannot reply properly to the clients.

Both clients have been built to be as much user-friendly as possible, reducing in the same time the waste of resources.

4.1.2.1 Android client

Consequently of the adopting choice of OSM, the Android client has been built on the osmdroid and osmbonuspack libraries.

Indeed, by default, Android uses Google services, because it is a Google product. Osmdroid is a full replacement for Android MapView class, which was v1 API and includes a modular tile provider system with support for a

large number of tile sources. It also provides a support with built-in overlays for plotting icons or shapes.

The `osmbonuspack` library offers an interaction with OSM services inside an Android application, leaving a huge personalization space for the developer. Indeed, the View used into the Android application is based on `osmdroid`, while all the added functionalities have been developed on the second library. It also includes a class which replace the android `Geocoder` class, calling the `Nominatim` API, for a full OSM extension. Moreover, `osmbonuspack` set default behaviour for all the map components, which can be freely changed by the developer following the application needs.

Regarding the communication between the Android device and the server, which must be *asynchronous*, the `Robospice` library has been chosen. Indeed, for executing an external request, Android needs to use an asynchronous tool, which by default is represented by the Android `Async Task` or the `Loaders`, but both of them have problems.

As first, `AsyncTasks` can create memory leaks or crashes because the call back methods declared will be executed even if the activity on which the `AsyncTask` has been declared has been destroyed. On the other hand, `Loaders` are too tied to the activity lifecycle, destroying the requests sent if the activity changes.

`Robospice` is a library that has been developed in contrast to these transmission methods that are considered by the developers as 'obsolete'. This because the other communication methods do not follow the activity lifecycle, losing the needed data while rotating the device or starting a new activity. In contrast, the `Robospice` library does. This feature has been implemented by starting an `Android Service` that works in the background of the application and will end its task even if a new activity is started and the previous one is closed. Moreover, it allows caching the results on the mobile device and limits the memory leaks, unlike `Async Tasks`. This allows a reliable communication between the mobile device and the server, avoiding data loss. The Fig. 35 shows in an ironic way what I have said the above sentences.

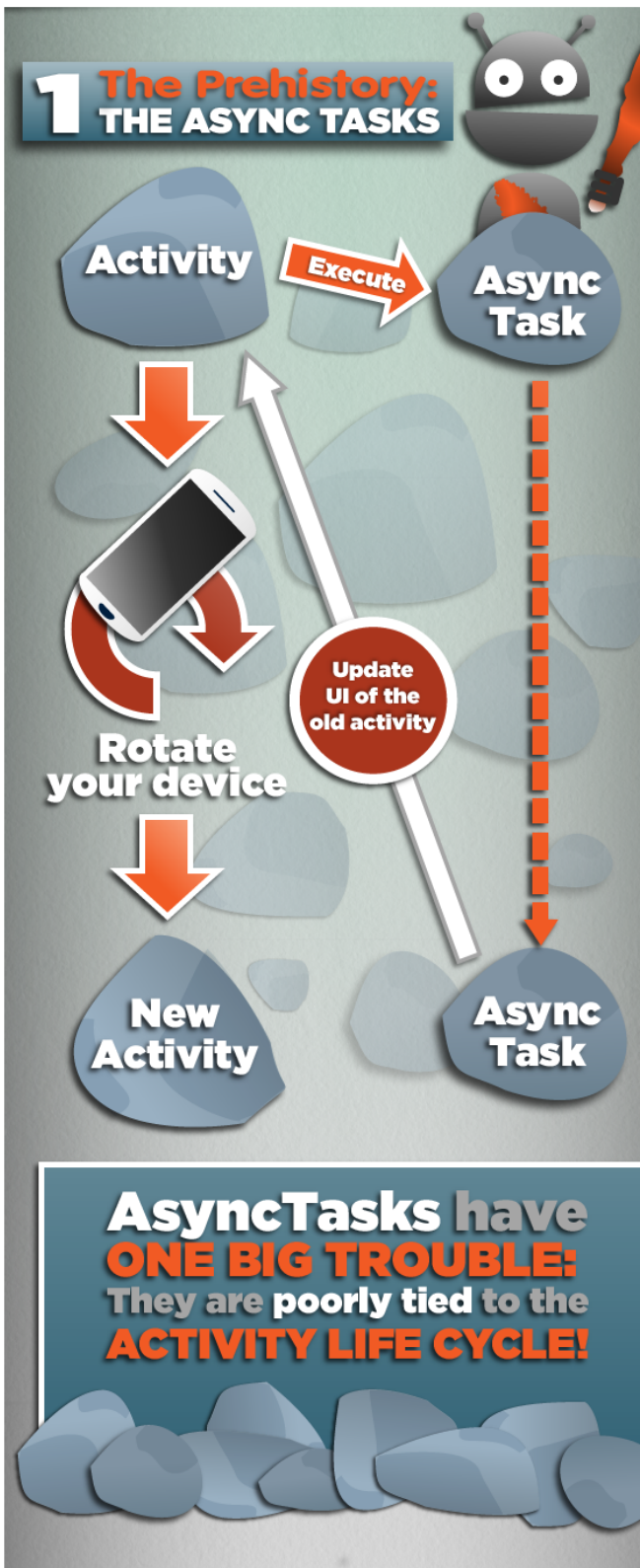
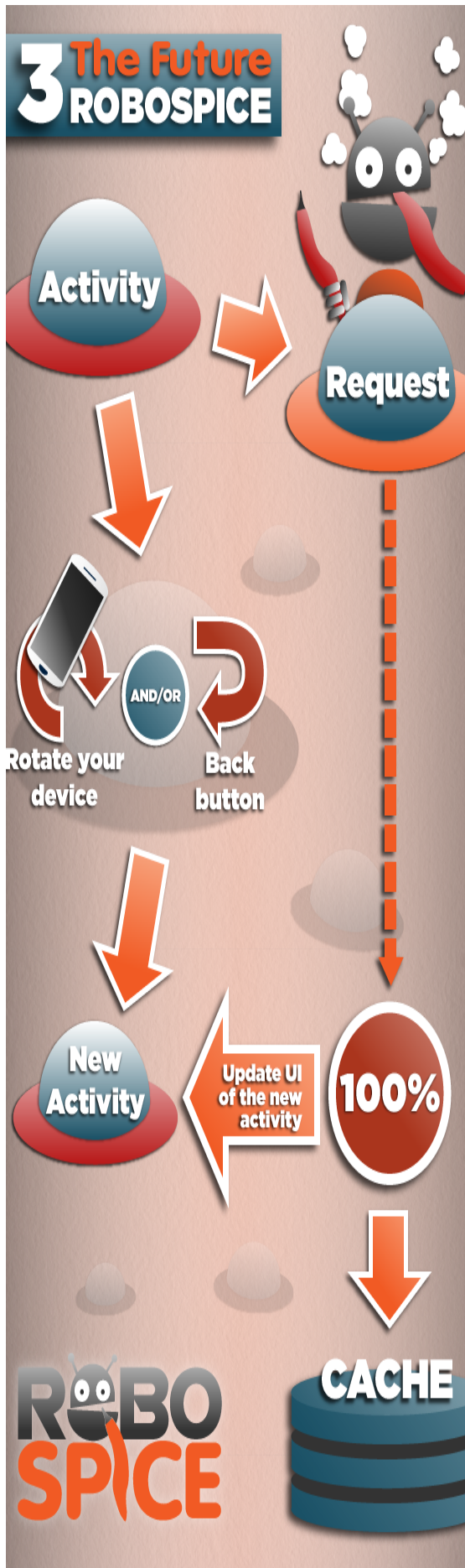


Figure 35. Robospice compared with obsolete solutions



Moreover, Robospice is strongly typed, so is possible to query web services using POJOs as parameter and the result can be a POJO too. This is particularly useful for this project, allowing to exchange with the server directly POIs or Barriers, staying at a high-level of abstraction.

For this reason, both for POI both for Barriers an extra class has been implemented that represent a list of entities, which is received from the server and that contains all the needed entities. The same POJO logic is used during the upload phase, in which the entity has been initiated on the client device and then sent to the server that will re-elaborate the received information for storing a new entity in the database.

So, the first screen of the application will show just a map with all the elements belonging to the current location using the osmdroid and osmbonuspack infrastructures. A button is provided for allowing the filter functionality, which will be implemented reorganizing the previous obtained data from the server.

There will be also two spaces designate for declaring the starting and destination point for the route calculation.

The two views for uploading, relative to Barriers and POIs, have been designed as two different pages. The more complex one is the POI interface: it must contain a text space for each field, such as name and description, and four different star-rating systems for each criterion. Moreover, it must have a space for a photo preview and the possibility to shoot. The same content must be provided in the Barrier section, that is less complex because the criteria is just one instead of four.

The last functionality needed from both the interfaces is the possibility to select precisely the location of the new entity or select which entity the user wants to update. For this purpose the best solution is to open a new screen showing a map with the already saved entities: if the user clicks one of the showed entities, the system understand that the user wants to update an already stored resource, holding its coordinates.

On the other hand, pressing on a free point on the map, the system understand that the user wants to insert a new value and it will retrieve the coordinates from the clicked location.

4.1.2.2 Webpage

In order to offer a dynamic map interface, the library chosen for implement the map view is *Leaflet*. This is a JavaScript library based on three main principles: simplicity, performance and usability. Indeed, it weighs just 33KB and offers all mapping features needed.

Leaflet provides a list of plugins developed by various Leaflet users for increasing the library potentiality, and in this project a pairs of them are needed. For example, the various functionalities of the web application, in

order to create a flexible and simple interface, can be contained into a map sidebar, that is an open source leaflet plugin.

The sidebar can be fully customized and the idea is to divide it in four parts, two containing information on the POIs and the other two on the Barriers. Getting into the details, one of the two parts contains the list of all the loaded entities, while the other allows to insert or update an entity. The main difference between the two subgroups is that the filter function must be inserted in the POI, allowing the user to display only a category of POI, exactly as in the android application. The filter selection beyond filtering the displayed markers, filter the list elements too, visualizing just the ones that belongs to the selected category.

Moreover, there must be the possibility for the customer to change the view of the area, because as default the map loads the area in which the user has been located. This feature can be implemented by geocoding the name of the desired city and then set the received coordinates as centre of the map.

The last main feature of the map is to find the current location of the user, which is mainly used to identify the city area in which he/she is located. This feature can be called just clicking a button, which have the function to centre the map in the registered location.

All the webpage behaviour will be implemented using JavaScript due to its extreme flexibility and potentially. Indeed, JS is one of the most versatile and effective languages used to extend website functionalities. As first, JavaScript is executed on the client side, saving bandwidth and resources on the web server and is relatively fast: indeed, the time difference between the download of the code and the results visualization on the webpage is completed almost instantly, of course depending on the task. Secondly, it can be used for a in a huge variety of applications and it can even support complex algorithms.

So, for sending requests to the server the AJAX technique has been adopted. *AJAX*, which stands for *Asynchronous JavaScript and XML*, is an approach that allows to create dynamic webpage, changing the content of the webpage dynamically, exchanging data with the server behind the

scenes. Moreover, the presence of XML in the name does not indicate the necessity to use the XML. This is based on JavaScript for representing the data and on XMLHttpRequest to retrieve or send data to the server. In the Fig. 36 there is showed the current web interface workflow. All the showed communications are implemented using AJAX and exchanging XMLHttpRequest objects.

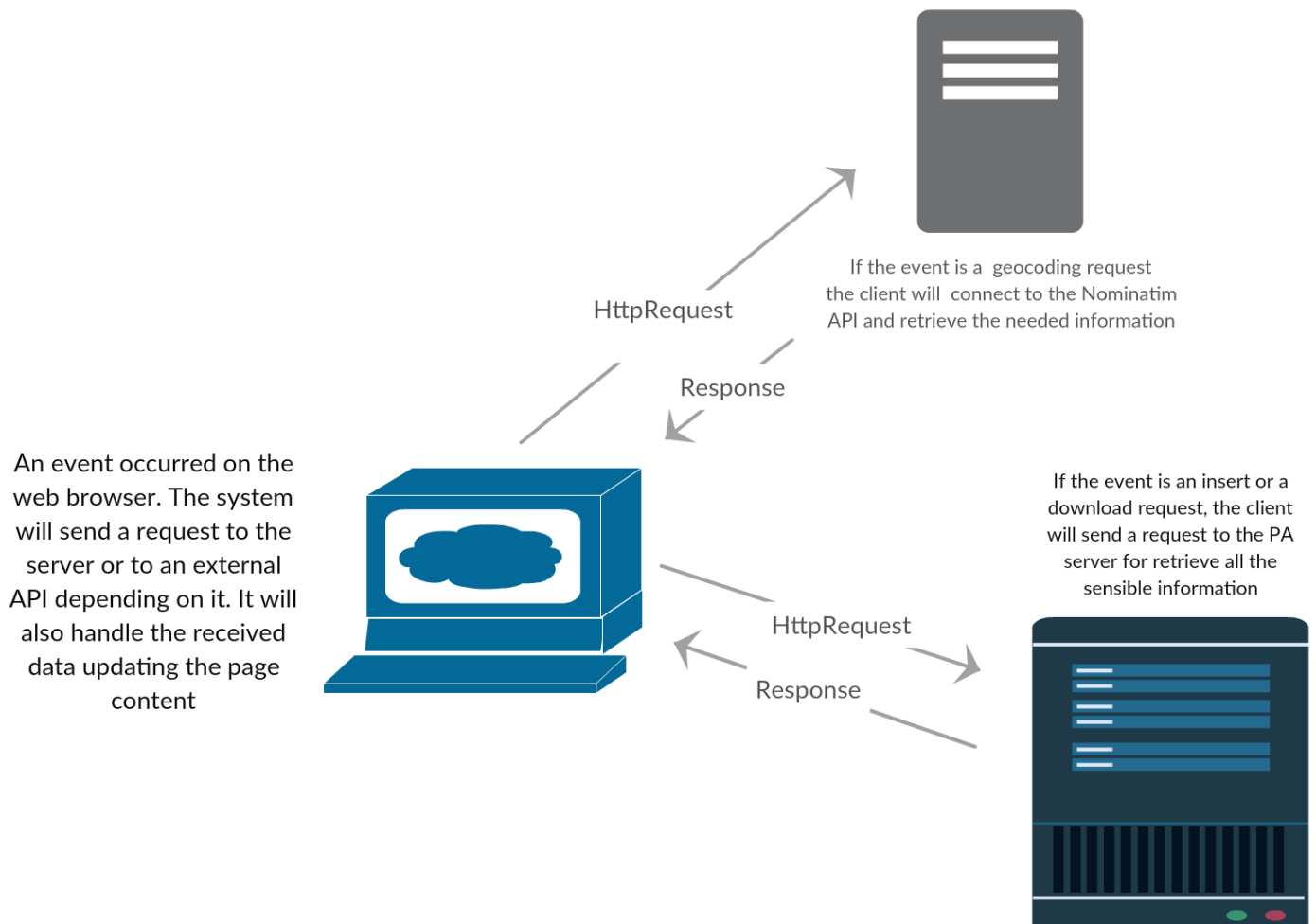


Figure 36. Web application workflow using AJAX

4.2 Data Persistency

The persistence of data can be implemented in various ways, even if using Spring as framework. As first, the database used is *PostgreSQL* a complete object-relational database management system (ORDBMS) that allows to store data securely.

It is cross-platform and runs on many operating systems because one of its main goals is being standards-compliant and extensible. It is particularly adapted for preserving data integrity, guaranteeing a reliable service, exactly what is needed by this project. Furthermore, it is particularly indicated when must be handled complex database designs: the saved entities will be probably the union of two or more different classes, creating a really complex database design. PostgreSQL seems to be the best choice.

Other adopted tools in order to allow a high-level interaction with the database are Hibernate and JPA, allowing to approach to the database entities as POJO classes. This is possible by the features offered by those two tools that provide an ORM approach. Basing on this, JPA provide a mapping infrastructure. The JPA mapping, as the Hibernate one, can be done by annotations or by XML, even if the XML approach could create confusion because all the information are contained in an external file that risks to begin too large and too complex if the project is vast. For this reason this project will use the annotations approach.

Based on these architecture principles, I will create two POJO classes for defining the needed entities for the project purposes, explained in the following two sections, using the declaring rules of JPA. Both entities need also to implement the Serializable interface in order to send them across the network.

4.3 Design of the entities of interest

This section illustrates the design process of the main entities of this thesis project. The main purpose is to have a complete set of information that can help to study and to understand the user opinions, figuring out what are the real needs of each city in order to improve the life experience of people with impaired mobility. For this reason, the set of stored information must consider as much details as possible.

Both the following entities depend on another entity, which helps to limit wastes of resources: the City entity. This entity store the cities that have already been populated and it counts the number of current POI and Barriers that belongs to each specific city area. This allows to create a link between the single element and the belonging area, allowing to narrow the loading phase too.

4.3.1 POI implementation

The first entity to declare is the POI, already explained in the section 4.2.1. For satisfying all the required characterization and for allowing a flexible usage of the resources, I have decided to create a POJO that is the join of other two classes: the Description class and the Votes class.

I have chosen this approach because in this way there is a clear role distinction between objects and because a vast number of Votes and Descriptions could be linked to the same POI entity. The classes are so linked by the One-to-Many relation, which indicates that for one entity can correspond many objects and vice versa.

The Votes class contains all the needed criteria to evaluate correctly a POI entity, such as a field for voting the parking service, another for reviewing

the bathroom adaptation, another for the adapted access and a last for the internal space adaptation. Moreover, it has a field called 'vote' in which there is stored the final vote, which is the output between the average of the previous fields. The Description entity is simple composed by a description field and an image field, representing the user comment.

Both classes depend by a single POI entity, which is referred in the structure declaration. Anyway, a POI entity must be composed by the following fields:

- Id: this allows to differentiating the single entities into the database
- Name: this field contains the name of the physical structure represented on the map
- Descriptions: a collection of Description objects that indicates the users comments regarding the specific POI
- Vote: a set of Votes objects that indicates the users feedbacks regarding the specific POI
- Type & Specific_type: these fields indicate the category identity of the specific object. The type, which represents the category of the objects, can be one of thirteen possible, which are: Food(1), Entertainment(2), Bar&Nightlife(3), Accomodation(4), Toilet(5),Shop(6),Diplomatic(7),Education(8),Sport(9), Tourism(10),Health(11),Transport(12),Supermarket(13).
- Latitude & Longitude: these fields provide the detailed location of the specific POI
- City: this field specifies the city in which the POI is located
- New POI: this allows to identify if a POI has been commented or not.

- User POI: this allows to understand if a POI has been added from an user or if it has been downloaded from the OSM database.

4.3.2 Barriers implementation

The Barrier entity, which is described in the section 4.2.2, follows the POI approach being composed by the union of more entities. The main difference is that the only vote submitted from an user can regards the Barrier accessibility and for this reason the Description class contains, in addition to fields for the description and for the image, a vote field in which each user can vote the accessibility of the architectonic obstacle.

The Barrier fields are the following:

- Id: this allows to differentiating the single entities into the database
- Name: this fields contains the name of the physical obstacle represented on the map
- Start date: the date in which it has been added to the database
- End date: the supposed date in which this obstacle will be removed
- Descriptions: a collection of Descriptions objects that indicates the user opinions on the obstacle
- Latitude & Longitude : these fields provide the detailed location of the specific Barrier

- City: this field specifies the city in which the POI is located

As in the POI case, the Description is referred to the Barrier using a One-to-Many relation.

4.4 Localization tools

As introduced in the previous sections, one of the main features in the localization, in order to organize the entire behaviour of the applications. The localization tool changes with the platform: the web interface will use a different tool respect the Android application.

Indeed, the Leaflet library provides the possibility to locate the user, calling a specific function. This is the `locate()` method, which uses the W3C Geolocation API to retrieve the current location of the device. Indeed, W3C Geolocation API is a platform provided by the World wide Web Consortium to standardize an interface to retrieve the geographical location information for a client-side device. The returned location is characterized by a specific accuracy based on the best location source available because the location can be based on various set of sources, such as IP address, MAC address, Wi-fi, RFID and GPS.

On the other hand, on the Android application the current location can be retrieved by three ways: using or the GPS provider or the Network provider or the passive provider. The GPS provider determines the location using satellites, while the network one determines the user location based on availability of cell tower and Wifi access points. The passive provider can be used to passively receive location updates when other applications or services request them without actually requesting the locations. This provider will return locations generated by other providers.

The main difference between these providers is that the GPS will return an high accuracy location, almost 20ft, but it is the slowest provider and it brings an high level of resource consumption. Also it needs a sight line to the satellites pain the missed sent request. The network provider guarantees a 200ft accuracy level, but it does not need all the precautions of the GPS provider. Indeed, it doesn't need a sight line to the satellites and it implicate a low resource consumption. The last provider, the passive one, has a really low accuracy level: 5300ft.

Having considered all the previous information, my decision is to adopt only the first two providers, because the system needs a minimum accuracy to work. An accuracy of 5300ft corresponds to 1.6km, which can create a wrong call to the server if the application is launched near a border area. The information gathered by the providers are then elaborated from a Location Manager, an entity in charge of provide the access to the location services.

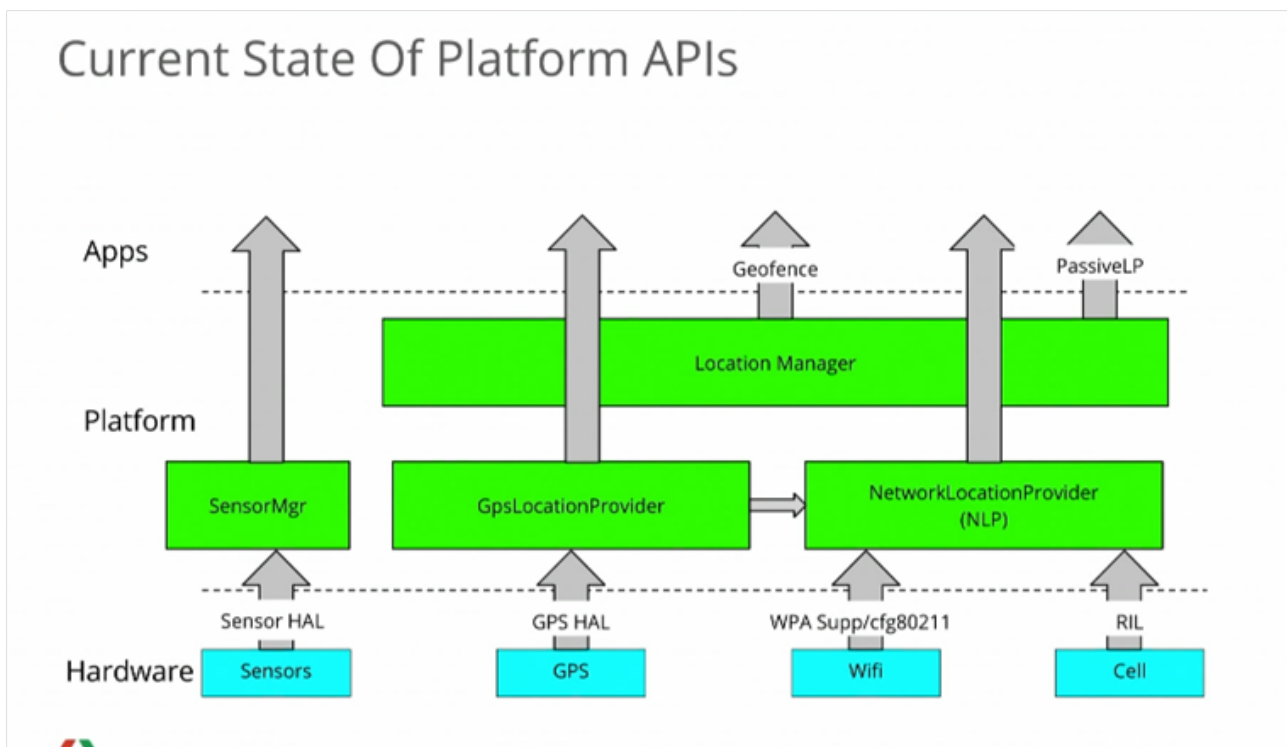


Figure 37. The adopted location architecture

Nowadays, Android provides a new kind of provider called Fused location provider but it is based on the Google services. For being coherent with the choice of using just OSM based services, I will not adopt that kind of provider, holding the architecture showed in Fig. 37.

4.5 Design of the Client/Server Logic

In this section I will in more details the logic that link the two clients to the server side, which is shown in Fig.38. As first, as introduced in the previous sections, the client must identify the current location of the user. The more accurate it is, the better it is, but the main accuracy level regards the location area, so both the Network provider and the GPS provider are a good choice, considering the Android application.

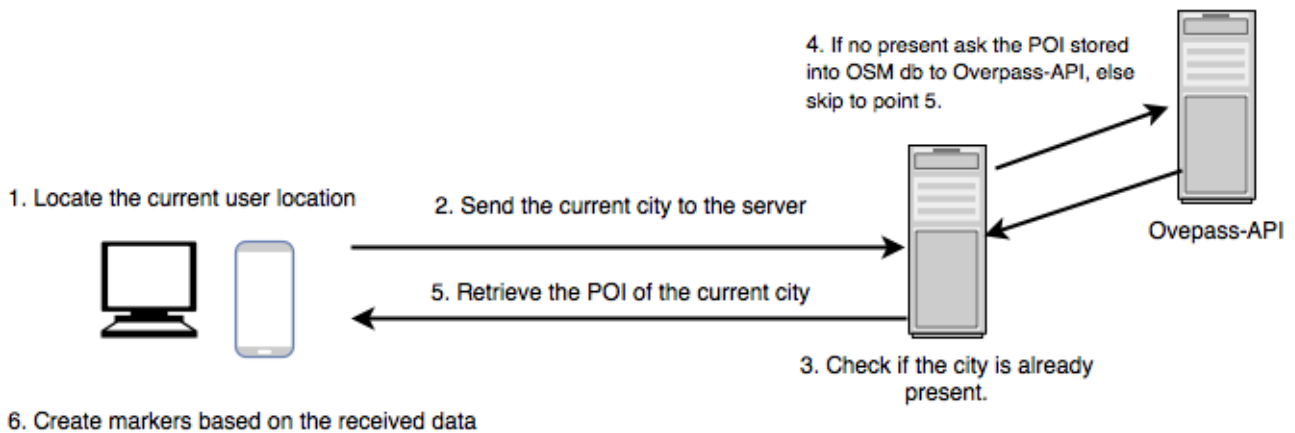


Figure 38. Data retrieval

Once that the device has identified the belonging area, it retrieves the corresponding city checking if it is already present into the database. If it has already been searched, the client just retrieve the belonging entities from the server, while if the city area has not already been searched, the server will make a request to the Overpass-API, downloading from the OSM database all the stored POI, forwarding them to the client.

After that the client has received all the data, it will reorganize them and create a marker entity for each of them, after having retrieved the precise category of each one of them, assigning the current icon. Each marker window must contain the user comments and the basic characterization information.

The second part of the communication, showed in Fig.39, consists into the upload of the entities: the user must be able to upload the needed information to the server, which has to store them into the database. Once the upload has been done, the map should being updated, allowing to the user to concretely see his/her contribute to the platform.

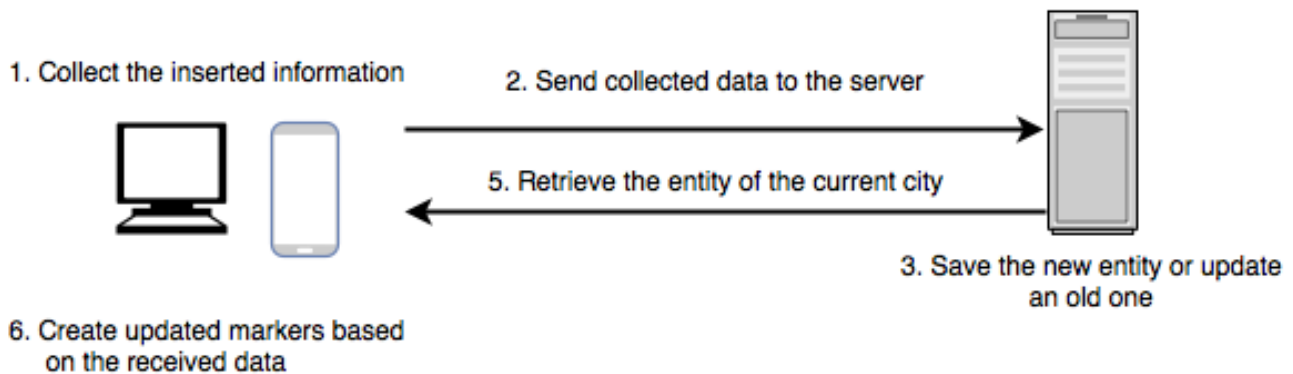


Figure 39. Data upload

Sending another request to the server and asking the new set of data stored, does this. For doing this, on the Android side has been adopted the Robospice library, exchanging between the client and the server directly the POJO, allowing an high-abstraction level communication even if it is heavier than a communication based on JSON, for example. This because the Android OS, as illustrated, allows to focus all the device resources totally on the current activity, freeing up memory if necessary.

On the web side, instead, the data are transmitted using the AJAX technique flanked with the JSON representation method.

4.6 Pedestrian route design

The extra function present into the Android client is the planning of a pedestrian route that avoids obstacles for impaired mobility people.

This service is offered using the osmbonuspack library, which allows to send requests to various kind of route APIs, adding various parameters to them. The selected API for this project is MapQuest, which is an American free online web mapping service. Nowadays, MapQuest has been integrated with OpenStreetMap, allowing to use OSM data to handle many of the features found in the MapQuest Platform Web Services. So, through the library the request is sent to the API and retrieved by them phone as a Road entity.

This entity is like the way entity of OSM and it is composed by nodes. Once that the nodes has been retrieved, they must be checked in order to understand if the route really avoid that obstacles, as summarized in Fig. 40 . If the route contains the Barrier coordinates into the path, the client will try to find an alternative route and show it on the map, linked by a line.

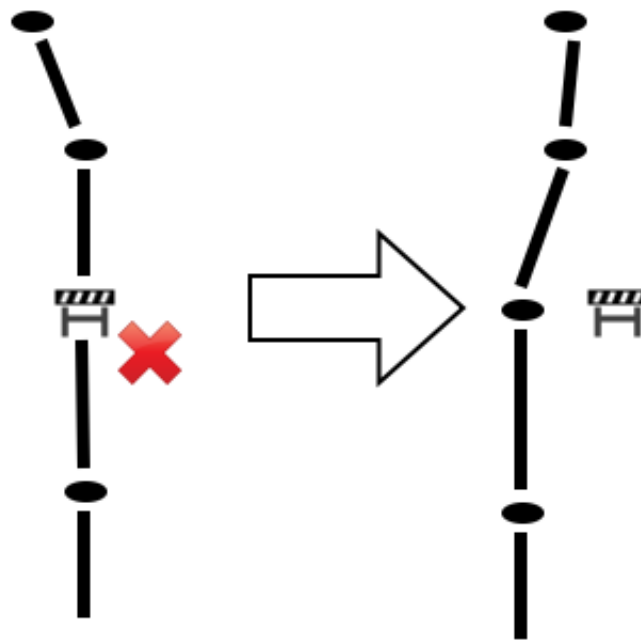


Figure 40. Route checking process

The key-part of this function is to identify the right ID of the nodes that contains an obstacle, in order to not consider them while sending the request. For this purpose, is possible to use the Nominatim Geocoder class provided in the osmbonuspack library that returns all the OSM considerable information that the default Android Geocoder can not get.

For limiting the resources consumption, the reverse-geocoding is run just on the Barriers elements that are characterized by Accessibility vote that is under the high-level threshold. The ID nodes that are sent into the request are not considered in the route calculation of the API.

5. Implementation

In this chapter I will explain all the passages and the choices taken during the code development. The whole work has been divided in two main phases: the theoretical development and the debugging. In the first portion I have developed the entire code just following the abstract schema of the project, while in the second part I have tried the efficiency of the code.

By these considerations, the entire code has not developed linearly, but it is the product of multiple considerations that have been done in different moments of the development and I will try to explain the consideration that I have done during the different phases. The base principle beyond the code production has been to offer the most fluid interface, considering the large amount of data that the client device has to manage. Indeed, both clients have to communicate with the server by sending single entities or large collections of them that must be displayed on the current map. This means to rebuild the original objects from a representation for the browser and to deserialize the received POJO for the mobile application.

Moreover, all the information contained into the received object have to be extracted and used to create map markers, which are divided in different categories, depending the specific entity. Indeed, POI entities are represented by icon numbers, in which the number changes with the category of the object while the Barrier entities are characterized by a different kind of icon, as showed in Fig. 41. This feature is obtainable by mapping each object with a specific category is characterised by a specific icon.

I have also spent a lot of efforts on prediction, trying to imagine all the possible user interactions, avoiding random crashes for unexpected actions. The all implementation can be divided in three main parts:

- The first part regards the persistence and the methods to access and modify the stored data

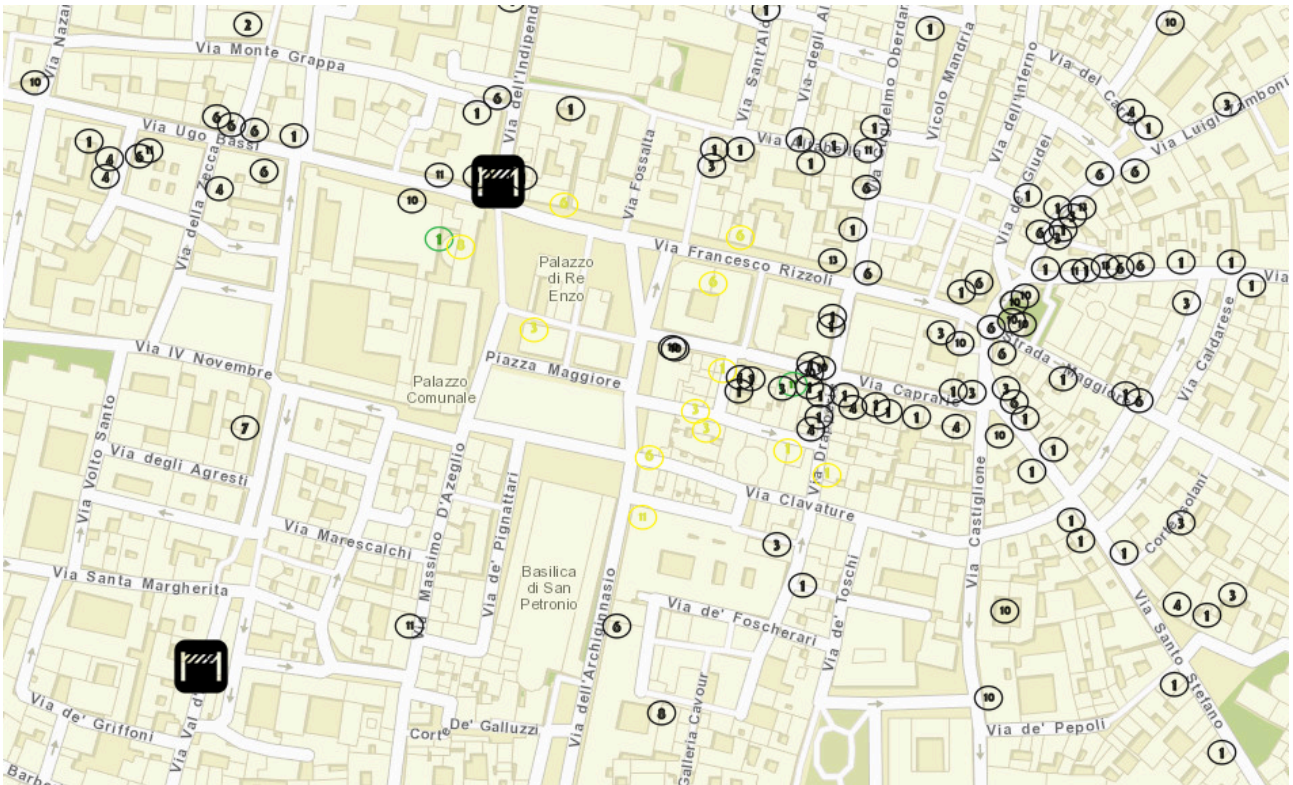


Figure 41. The browser map

- The second part regards the server interface and the called methods both for user and administrator panels
- The third part regards the mobile application and the methods called by it on the server

All the administration sections are just present on the web side and are just accessible by logging with root credentials. The system recognizes the authentication permissions by the `@PreAuthorize` notation. Indeed, this annotation allows Spring to associate each logged user to a specific set of contents, as functions or webpages. The maps and checks the current user authentication permissions, deciding which elements are accessible or not to the current logged user.

Moreover, both client maps does not allow zooming out excessively, because for now the need is to focus on city areas. For this reason, I have set a range of zoom level that allows the user to have both an entire vision of the current urban area both a precise view of a particular street. The

expansion over this zoom range will be probably considered for next implementations, allow the user to freely scroll around countries.

In both client implementations a particular attention has been spent on the photo resizing and the free of memory resources, avoiding an overweight of memory usage, particularly on the mobile application because one goal is to render the application runnable on old smartphones too. The GUIs have been designed very simple, trying to render the interface as intuitive as possible.

Another key issue is the coordinates accuracy: indeed, a vast number of operations need to reveal the presence or the absence of some entities and one tool is the coordinates pair. But if different tools are designed for have different accuracy levels, this can fail the entire identity algorithm. For these reason must be developed a flexible set of comparing algorithms.

5.1 Implementation of the Client/Server Logic

The first step for implementing the communication logic that characterises this project is to set up the tables that are going to contain the data regarding the needed entities. The two main entities, Barriers and POI, are stored in two different tables and are linked to their descriptions entities though foreign keys.

Each Description object is characterized by a mandatory text message and an optional image. The only difference between the description of a POI entity and a Barrier entity is that the Barrier one has another field, the vote. Indeed while the POI entity is provided by a support table that contains the current vote, the Barrier entity vote is calculated by the total average of all the descriptions referred to it. The entire referred descriptions entities link to the parent object through the OnetoMany JPA

relation, which allows associating to a single entity a set of others child objects. All these collections of objects are contained in Set<> structures, because one advantage of using Sets is that it forces you to define a proper equals/hashCode strategy, which should always include the entity's business key. A business key is a field combination that's unique, or unique among a parent's children. Moreover, each child entity contains an image field, which is referred to a single comment of an user. The approach adopted in this project for transferring images data between the various devices is through the base64 format. Indeed, it allows to encode various kind of images in Strings, sending them through the same HTTP connection that link the client and the server, without opening others.

This approach has been adopted because all the data transmissions transfer a low amount of data: indeed, on both clients, each image is resized with a set of algorithms show in the next sections, proportionally with its characteristics. Moreover, the user can upload comments sequentially, so the entities are sent individually, not in groups. This allows using the only advantage of base64 transmissions. I have decided to use that format because the maximum image weight is around 250KB, so lightweight image files.

All these descriptions entities are then displayed into the marker popup, showed in in both clients. These are indeed showed as lists into the marker window allowing the user to read both text and see the related image. On the browser application this is done by creating a specific HTML code for each object, which includes the name, the vote, a preview image of the place and a list of descriptions, as showed in Fig. 42. This code is built reading and understanding all the information contained into the POI and Barrier classes, creating so a faithfully graphical representation. While on the browser map is only possible to zoom up a specific photo from a comment, in the Android application clicking on a specific comment will open another fragment in which is displayed the photo of the current comment above the full comment list. The rest of the marker pop is the same of the browser one.

The main difference between the two markers is the background logic: indeed, on the browser, all the comments are loaded during the initiation phase, while in the mobile application all the comments are loaded when the user clicks on a specific marker. Indeed, the onOpen() method of the

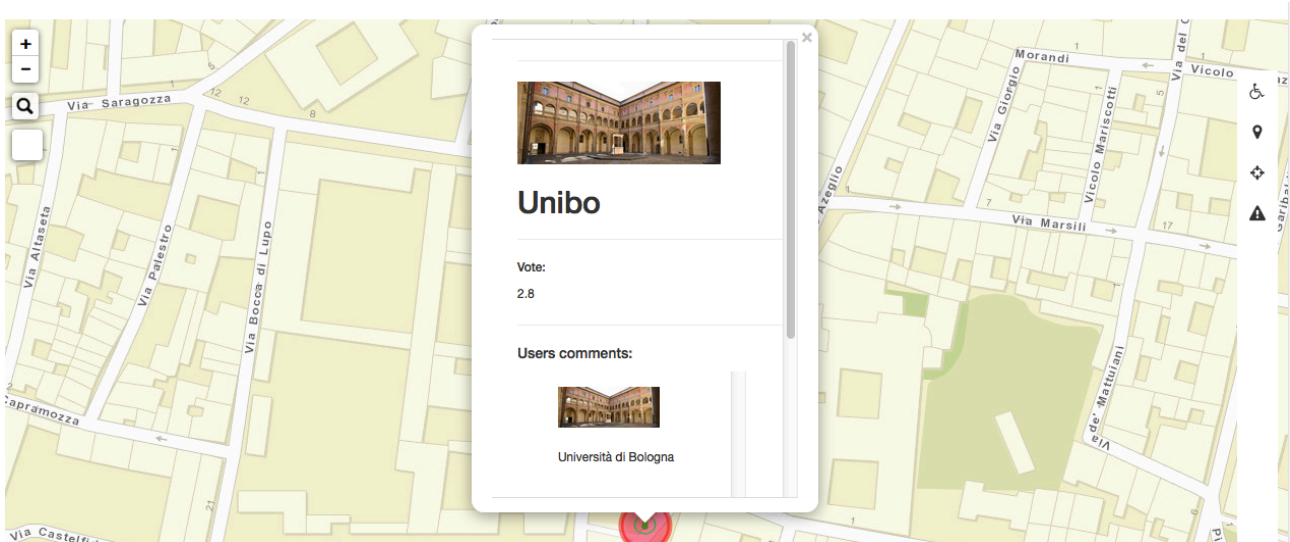


Figure 42. Web application marker cloud

marker call an asynchronous method that retrieves all the comments referred to the clicked entity, reducing the memory allocated during the initial phase; this logic has been adopted for both Barriers and POI. Moreover, both windows have a button that allows the user to directly open the comment view, in order to update the selected entity.

In order to obtain a fluid scroll interface for the map view, the two clients do not download all the entities regarding a specific city area all in one time. Indeed, downloading around 2000 elements, which one composed by images and descriptions, could cause an overload of the system, heavily reducing the system performance. For this reason, once the client focus on a first-screen, it will take the boundaries converted in coordinates of the current screen and calculate a larger area, as 1.5 times bigger than the original one and ask to the server to send all the entities referred to that area. This operation is launched in `onCreateView()` and will so load not just the entities that are showed on the current screen, but also others that are near. In this way, the system will call the server for a second download only if the user moves away from the current loaded area, overtaking a fixed threshold. Another precaution is to transfer a fixed number of items through all the zoom levels. In this way, the system will load a reduced set of entities, showing new ones when the user will zoom into a certain area. This has been done for two main reasons: at first, to reduce the resource consumption and to obtain a faster communication and in second to avoid

a confused map. Indeed, showing more than 200 elements on a map could cause the overlap of a large number of markers, bringing to an unclear map. Because a mobile phone has a reduced set of resources, I have adopted a different approach to download the data. Indeed, the client receives all the element information except the set of descriptions, which are downloaded later only if required, as explained in the previous paragraph. Thank to this strategy, the memory usage is heavily reduced, because the client do not have to download any file .

```

case 15:
    if(latDiff>=9000 || lonDiff>=9000){
        scroll_launched=1;
        IGeoPoint mapTopLeft = map.getProjection().fromPixels(0, 0);
        TopLatitude = (double) (mapTopLeft.getLatitudeE6()) / 1000000;
        LeftLongitude = (double) (mapTopLeft.getLongitudeE6()) / 1000000;
        IGeoPoint mapBottomRight = map.getProjection().fromPixels(map.getWidth(), map.getHeight());
        BotLatitude = (double) (mapBottomRight.getLatitudeE6()) / 1000000;
        RightLongitude = (double) (mapBottomRight.getLongitudeE6()) / 1000000;

        markers_visualization();
        if(new_center!=null) {
            map_center = new_center;
        }
    }
    break;
case 16:
    if(latDiff>=5000 || lonDiff>=5000){
        scroll_launched=1;
        IGeoPoint mapTopLeft = map.getProjection().fromPixels(0, 0);
        TopLatitude = (double) (mapTopLeft.getLatitudeE6()) / 1000000;
        LeftLongitude = (double) (mapTopLeft.getLongitudeE6()) / 1000000;
        IGeoPoint mapBottomRight = map.getProjection().fromPixels(map.getWidth(), map.getHeight());
        BotLatitude = (double) (mapBottomRight.getLatitudeE6()) / 1000000;
        RightLongitude = (double) (mapBottomRight.getLongitudeE6()) / 1000000;

        markers_visualization();
        if(new_center!=null) {
            map_center = new_center;
        }
    }
}

```

Figure 43. Scroll handler

The threshold can be calculated as shown in the Fig. 43: the system holds a variable that stores the current coordinates of the centre of the map. When the user scrolls the map, the application calculates the difference between the new map centre and the previous one. If the difference between the saved map-centre and the current centre is more than a fixed threshold, the call will be invoked. This method has to be called both on the scroll of the map both for the zooming operation, in order to update the current collection. Moreover, the zoom factor is a key variable In order to handle correctly the Scroll operation. At various zoom levels the delta changes radically: the more the zoom is high, the more the difference is little. For this reason, as shown in Fig. 42, I have implemented a switch statement that change the difference threshold based on the current zoom value. The

same operation is done on the browser client. This event is handled in two different ways: on the Android application this is done by initiating a `MapListener`, which is an entity that call functions when the map is dragged or zoomed, while on the browser the map is responsive to specific events, as the 'dragend' or the 'zoomend'.

The request sent to the server contains the latitudes and longitudes that describe the boundaries of the searched area, filtering the elements by their position. Indeed, the code is divided in two main parts: the first one obtains the data from the server, and the second shows the downloaded information. Both clients provide a set of functions that understand the information stored in the POI and Barriers collection and display them on the map as markers. The elements added to these collections are new entities or updated ones. Secondly, in order to calculate the boundaries of the search area, the clients calculate the difference between the top and bot sides and left and right sides. Then these differences are multiplied for a fixed constant in order to decide the shape of the future area and then added or subtracted to the current boundaries, creating a wider search area. Currently, the clients load from the database all the entities that are into an area that in 1.5 times the original one, as showed in Fig. 44. Thirdly, the map view will create the markers entities from these local collections, verifying which are the elements that are contained between the current screen boundaries. This allows a faster performance in case that all the downloaded entities are already stored in the local data set. This approach has been adopted for both Barriers and POI entities on both client types.

```
IGeoPoint mapTopLeft = map.getProjection().fromPixels(0, 0);
TopLatitude = (double) (mapTopLeft.getLatitudeE6()) / 1000000;
LeftLongitude = (double) (mapTopLeft.getLongitudeE6()) / 1000000;
IGeoPoint mapBottomRight = map.getProjection().fromPixels(map.getWidth(), map.getHeight());
BotLatitude = (double) (mapBottomRight.getLatitudeE6()) / 1000000;
RightLongitude = (double) (mapBottomRight.getLongitudeE6()) / 1000000;
diff_lat=Math.abs((TopLatitude-BotLatitude))*0.50000;
diff_lon=Math.abs((LeftLongitude-RightLongitude))*0.50000;
TopLatitude+=diff_lat;
BotLatitude-=diff_lat;
LeftLongitude-=diff_lon;
RightLongitude+=diff_lon;
```

Figure 44. Android implementatio of the resizing area

Another functionality that both clients must handle with is the situation in which the user changes the city area to visualize. This situation can be reached in two different ways: or the user directly searches for another

city in the search field or he gets there by scrolling the map. In both situations, the clients reset the current entity collections and verify the presence of the new city area in the City table. If the city is already present, the client just have to ask to the server the stored entities belonging to the new city area, while if the city is not present into the table, it must communicate to the server to search a new city area. In this case, the server will send the query to Overpass and download all the POI referred to the current area from the OSM database, saving them to the database and forwarding them to the client. In this case, the system will send the whole set of POI because they do not include any type of heavy data, due to their current initialization. This allows the end user to freely focus on desired locations, even if they are not saved into the database.

Furthermore, for the POI entity the vote is saved into another table, referenced to the single entity by another foreign key. This table contains one entry for each POI entity and the value is updated each time that a POI entity is reviewed by a user. The reason to adopt this kind of approach is that the POI vote depends on four different variables that must be stored in order to hold significant data regarding the entity of interest, while the vote that refers to the Barrier entities is composed by a single parameter. Indeed, the average number of POIs into a city area is around 1500 elements, which is higher than the estimated Barrier entities. For this reason I have decided to occupy a single row for each POI entity, saving memory and resources. Indeed, holding a row for each vote uploaded by a user would have been traduced into an enormous number of votes for each POI, summed to the resources needed to cycle into it for calculate the average vote.

The POI and Barrier tables are strictly linked to another set of data that contains all the searched cities: each row is just composed by the name of the city and the number of POIs assigned to it and the number of Barriers. This table has the function of 'chronology' of the system. Indeed, once that a city has been searched and so saved into this table it can not be searched again, at least for the users: the administrators can indeed update the current data saved by logging into the administration panel and run the adding algorithm. Indeed, from this section, this limit has been bypassed in order to allow to the administrators to update the information stored. This necessity born because, as said in the previous chapters, the OSM database get updates very frequently for its crowdsourcing nature.

Furthermore, for each entity there are mandatory fields and optional fields, as said previously: these aspects reflect themselves on the database structure and on the future control logic of the algorithms. Using the ORM approach, allowed to Hibernate and JPA, I have declared the relative entities using java classes with annotations, as suggested. The various entities are created through a *Builder* entity, unique for each class. This entity allows to build a specific class, setting all the parameters by set methods. After that all the needed parameters have been set, the Builder will make the entity persistent, adding it to the JPA cache.

Getting through, the administrator POI panel is composed by a single table that contains all the POI into the database with the needed information, as shown in the Fig. 45. From this panel is also possible to add a specific city area to the database or update the stored information with the new set of data contained into the OSM database. This operation will not override entries with the same name and position, it just add entities that are not present into the database yet. This allows checking the current state of the POIs stored into the database, including the number of descriptions referred to that entity. Instead, the Barrier administration section is composed by a table that shows all the Barrier entries. In both sections is possible to directly delete a selected entity.

Insert the name of the city to search:

Run

POIs Contained into the database

Name	Type	City	Specific type	Latitude	Longitude	Vote	Description number	
Michelangelo	health	Bologna	pharmacy	44.3483235	11.7152433	0	0	Delete
Polo Territoriale Città di Imola (Ospedale Vecchio)	health	Bologna	hospital	44.3550335	11.7048449	0	0	Delete
Vigili Urbani	governmentservices	Bologna	police	44.349289	11.6993626	0	0	Delete
Biblioteca Comunale	education	Bologna	library	44.3547159	11.711278	0	0	Delete
Cinema Cristallo	entertainment	Bologna	cinema	44.3557051	11.7156435	0	0	Delete
McDonald's	food	Bologna	fast_food	44.368268	11.7284024	0	0	Delete
Scuole Medie Sante Zennaro	education	Bologna	school	44.3490541	11.6998196	0	0	Delete

Figure 45 .POI administrator panel

On the other hand, the user panel (shown in the Fig. 46) is just composed by a mapview that contains a set of extra tools that implement the interactive functions of the map, as locate a specific POI, filter the showed content, add an entity of interest, etc. These blocks gain interactivity to the map, rendering it flexible and updated. For example, is possible to insert an entity of interest into the database or update an existent one and instantly visualize the inserted data in real time into the map, without accessing another section of the website.

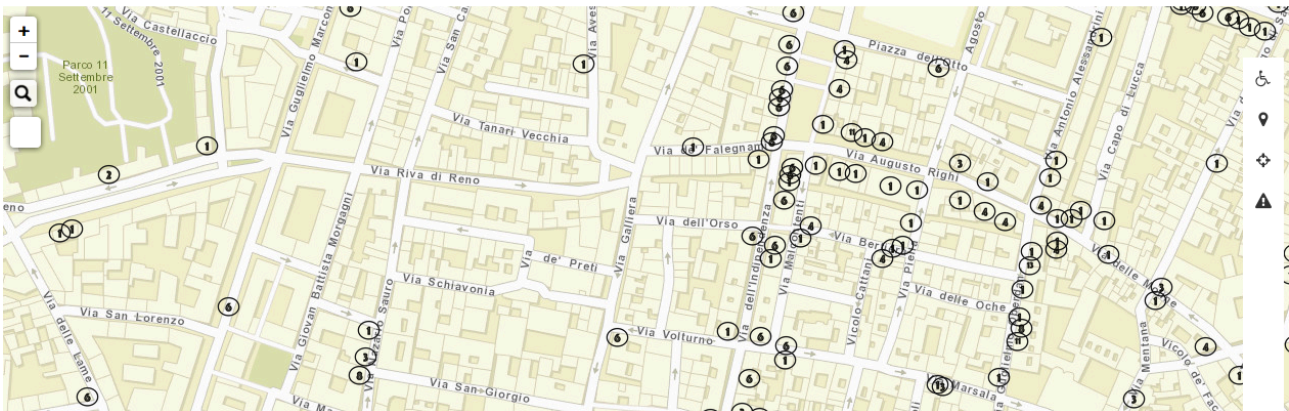


Figure 46. User map

These blocks are all Leaflet plugins, customized for the current project. I have modified the Turbo87/sidebar-v2 project in order to obtain a personalized sidebar, containing four different sections: the first section contains a list with all the downloaded Barriers; the second section contains the filter functionality and the list of the downloaded POIs. Both lists allows to directly locate the selected entity on the map. The last two sections allows to insert information into the database, creating a new entity or updating a new one. Moreover, I used the stefanocudini/leaflet-search project allowing the search function to the map.

The main logic of this thesis project is to hold a frequent communication with the server, in order to charge a reduced number of entities at time, with the goal to reduce the used resource both on server and on client side. The alternative would have been to send all the stored entities to the client just once in the beginning. But after some tests I have seen that it uses a large amount of the resources of the client for 20s at least, freezing the application. For this reason I have decided to adopt this approach, limiting the queried area to 1.5 times the current screen of the user. In this way the

size of the exchanged data is heavily reduced, by sending a little portion of the total amount of data. Another strategy adopted in order to reduce the resource consumption in Android is to pass through fragments information as the current city area by Intents, avoiding to call Geocoding requests in all the application screens.

5.2 Implementation of safe route function

Various steps compose the calculation of the pedestrian route implemented in the Android client, which as introduced before, is not directly implemented on the device itself. Indeed, the calculation algorithm to calculate the route would have implied the full amount of resources, probably causing a crash of the application or at least a long freezing of it. Indeed the most used algorithm for calculate the fastest route between two different points is the *Dijkstra's algorithm* which implies a lot of calculus resources. Moreover, this path is specific for pedestrian users and have the characteristic to avoid the Barrier entities that are located between the route. So, the algorithm would get heavier, because there are also some conditions to respect. Because of that, the client send a request to OpenMapQuest through the library osmbonuspack, though a class declared in the library, the RoadManager.

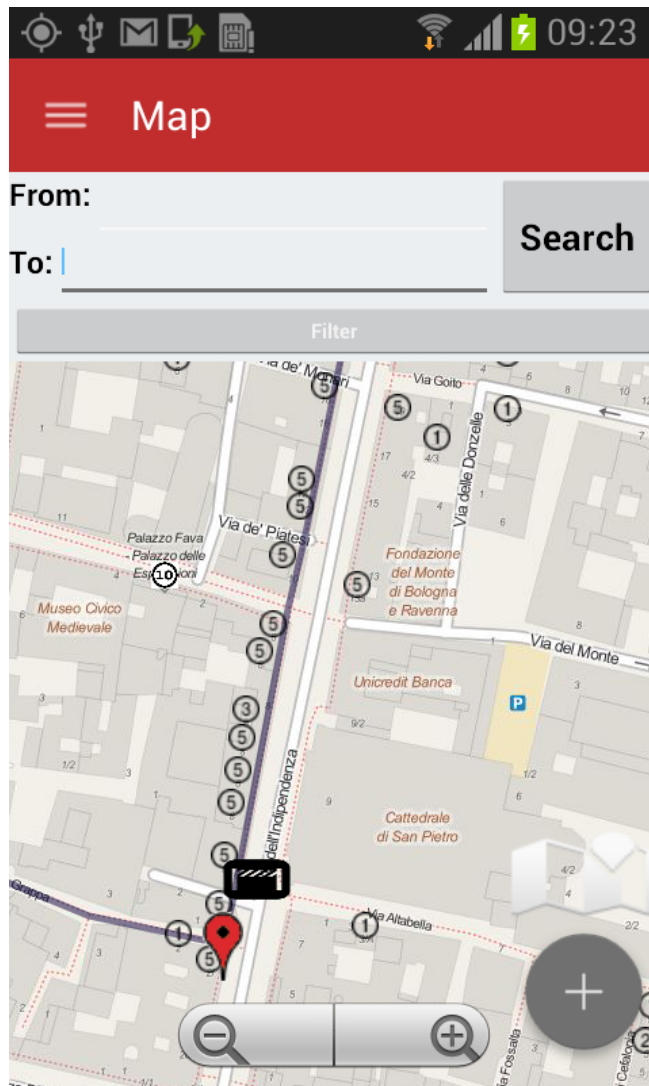


Figure 47. Avoid a Barrier

OpenMapQuest provides two ways to avoid specific options to insert into the request in order to avoid certain parts of a route: `tryToAvoidLinkId`, which tries to calculate a road that does not cross some parts of the streets, and `mustAvoidLinkId`, which calculate an effective route that avoid some places. The only difference between these two calls is that the second one can cause a crash of the routing calculation, while the other one is always safe. Moreover, the arguments of these two options are the IDs of the nodes or streets that must be avoided. For these reasons, the algorithm provide some checks and it could send two requests in order to try to guarantee an efficient routing.

Getting through the algorithm, at first it identifies the Barrier elements that are characterised by a low adaptation score and calls a reverse geocoding

operation by Nominatim, obtaining the OSM ID of that elements. Indeed, OSM categorizes each entity present into an area with a node, and each single street is composed by an array of nodes. For this reason, the algorithm executes another request for identifying the ways in which is present that particular node. After having retrieved this information, the client will call the external API and receive the route calculated by the `trytoAvoidLink` option. After this, it will check the received route, understanding if it really does not cross any Barriers entity. If the result is positive, it will load that route on the map, but if the result is negative, it will send another request using the `mustAvoidLinkId`. I have decided to call this method as second because after some testing I have verified that the route calculated by the first method is in general shorter and that this algorithm just works if there are some near nodes around the Barrier, while the first one calculate an effective alternative route. Because of that, the application will verify if this calls cause the crash of the API, controlling the Status response. If the status reports the presence of a route, this will be loaded into the map, while is the status reports an error, the previous route will be loaded.

```

this.roadmanager.addRequestOption("tryAvoidLinkIds=" + to_avoid);
temp_result = roadmanager.getRoads(waypoints);
boolean redo = false;
if (temp_result.length > 0) {
    if (temp_result[0].mStatus == Road.STATUS_OK) {
        for (GeoPoint n : temp_result[0].mRouteHigh) {
            for (Barriers b : barriers.getList()) {
                double latitude__ = Math.round(b.getLatitude() * 100000.0) / 100000.0;
                double longitude__ = Math.round(b.getLongitude() * 100000.0) / 100000.0;
                double latitude_ = Math.round(n.getLatitude() * 100000.0) / 100000.0;
                double longitude_ = Math.round(n.getLongitude() * 100000.0) / 100000.0;
                if (equals_double(latitude_, latitude__, 0.0003) && equals_double(longitude_, longitude__, 0.0003)) {
                    redo = true;
                }
            }
        }
    }
}

this.roadmanager.addRequestOption("mustAvoidLinkIds=" + to_avoid);
temp_ = roadmanager.getRoads(waypoints);
if(temp_result.length>0 && temp_result[0].mStatus==Road.STATUS_OK){
    temp_result=temp_;
}

```

Figure 48. The routing algorithm checkings

Each node of the showed route, if clicked, will pop the directions to get through the path, including the distance left and the name of the streets to get through.

5.3 Data exchange

Through the entire project, clients and server exchange a large amount of data, in order to guarantee a flexible platform. Each client is characterised by its own way to send data and to manage them. The logic that stays behind it is really similar, but the implementation is really different. Indeed, the browser client communicates with the server by JSON files, while the Android one through POJOs. The first guarantee high-speed transmissions representing the shared objects as Strings. On the other hand this implies a low level interaction and a full rebuilding of the original element from the JSON. Instead, the second communication method is characterised by lower speed transactions, but it guarantees a high-level interaction because both devices just work with class entities. Moreover, the webpage has been implemented through a scripting language, while the application by a compiled one.

Anyway, they also share the almost the same functionalities: they download and upload POI and Barrier entities, in various ways. Both clients must implement the functionalities to create a new entity from scratch, through specific forms. These forms have been implemented as sidebar sections in the browser and as other fragments in the application. These input sections must collect various kind of data type, as text, numbers and images. In order to guarantee a reduced consumption of resources on both machines, all the images are resized, reducing their size. Indeed, there is not the necessity to store high quality images, because they have to adapt to pictures with smaller sizes.

5.3.1 Photo settings

As introduced, both clients must provide a resizing algorithm to reduce the current size of the image file. This feature is necessary in particular on the mobile client because it allow the option to take a picture and to set it as image of the entity to upload. Indeed, this operation produces a high resolution image, which weigh 1 or 2mb at least, causing multiple issues if manipulated without a correct resizing strategy. Indeed, the Android OS will throw an OutofMemory exception if the size of the bitmap is not modified properly while obtaining the image from the device memory. This is because mobile devices have constrained system resources; in particular Android devices can have 16MB of memory available for a single application. Making some calculus and testing with a Galaxy S3 Mini, the photos taken have a resolution of 2560*1920, for a weigh of 2.30MB average. Mostly the photos have ARGB_8888 configuration, carrying to a resource consumption of 19.6MB, exhausting all the application resources.

Because of that, the resizing algorithm in the Android application has been implemented by the following steps. At first, the system retrieves the URI of the image from the system. This operation depends on the user choice to load the image from the gallery or by taking a picture. Anyway, after this step, using the file URI is possible to obtain information about the current size of the image and orientation, in order to represent the correct image information in the preview ImageView. In order to obtain the orientation information, the URI must be converted in a filepath, and the algorithm to do this changes with the current OS version. In order to adapt the application to various OS versions, I have created a module, called UriSupport, which implement all the different variants. Depending on the obtained information and on the desired output, the class ImageResizer calculates the optimal scale factor of the current image by setting the inSimpleSize value, as shown in Fig. 49. This factor requests the decoder to subsample of the original image, saving memory. After this operation, this class receive as input the orientation of the image, understanding if it must be rotated or not. This must be necessary in the case that the image is portrait. The image is then processed and added as image of the entity that must be uploaded.


```

bitmap=null;
AssetFileDescriptor fileDescriptor =null;
fileDescriptor =getActivity().getContentResolver().openAssetFileDescriptor( mImageURI, "r");
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeFileDescriptor(fileDescriptor.getFileDescriptor(), null, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
ImageResizer resizer=new ImageResizer();
String[] projection = {MediaStore.Images.ImageColumns.ORIENTATION};
Cursor cursor = getActivity().getContentResolver().query(mImageURI, projection, null, null, null);
int orientation = -1;
if (cursor != null && cursor.moveToFirst()) {
    orientation = cursor.getInt(0);
    cursor.close();
}
bitmap=resizer.scaled_bitmap(fileDescriptor.getFileDescriptor(), imageHeight, imageWidth, orientation);
poiImage.setImageBitmap(bitmap);

while ((current_w / inSampleSize) > MAX_WIDTH
        && (current_h / inSampleSize) > MAX_HEIGHT) {
    inSampleSize *= 2; //calculate the scaling factor
}
options.inSampleSize=inSampleSize;
options.inJustDecodeBounds = false;
bitmap=BitmapFactory.decodeFileDescriptor(fd, null, options);

```

Figure 49. the scaling factor calculation

Instead, in the browser client the image is loaded from the local disk of the user computer through a file input field. The image is then processed by a JavaScript entity, the FileReader. This is an entity that allows to read file in JavaScript using asynchronous methods. This entity, after having read the file, loads it into the selected image field as preview. Until this point the image has not been resized. The resize method, shown in Fig. 50, is called when the user tries to upload the new entity in the database, sending to the server the resized image. This is done by the canvas entity, which is mainly used to load or draw graphic on webpages. The function will receive as input the image field and the two-dimension threshold. It calculates the proportion of the image and calculates the scaling factor for resize the file.

Both algorithms allows to choose the correct resolution of the image proportionally with the setting of the limit size. Increasing these thresholds increase the image size, but increase the resolution.

```

function imageToDataUri(img, max_width,max_height) {
    var canvas = document.createElement('canvas'),
        ctx = canvas.getContext('2d'); // create a 2D canvas
    var height=img.naturalHeight;
    var width=img.naturalWidth;
    var ratio=height/width;
    var scale;
    var correct_height;
    var count;
    var correct_width;
    var count;
    //adapt the algorithm to the ratio for maintaining the proportions
    if(ratio<=1 ){
        correct_height=(max_width*ratio);
        correct_width=max_width;
        count=height/correct_height;
        scale=1/count;//calculate the scale variable
    }else{
        correct_height=max_height;
        count=height/correct_height;
        scale=1/count;//calculate the scale variable
    }
    var iwScaled=width*(scale);
    var ihScaled=height*(scale);
    canvas.height=ihScaled;// calculate the new height
    canvas.width=iwScaled;// calculate the new width
    ctx.drawImage(img,0,0,iwScaled,ihScaled);
    // produce a jpeg format image
    return canvas.toDataURL("image/jpeg");
}

```

Figure 50. The image resizing method in Javascript

5.3.2 Transmitted entities

All the entities are stored into a common database and this project is mainly based on three tables: the Barriers, the POI and the Cities. While the last one is just a support table, the first two tables contain all the key information that are necessary for make this software works. Thanks of Hibernate and JPA, all the methods that access the database are managed as Objects, as explained before.

Indeed, each table correspond to a Java class that is characterized by the `@Entity` annotation, indicating that the class is a POJO. By this, the system creates a table in which each field is a variable of the POJO class. All the fields are characterised by a single or a set of annotations that declare the properties of that particular field. Using these tools, I have created collections of element referred to a single entity, as sets of descriptions linked to a single POI or Barrier. These entities are all linked together through the `@OneToMany` annotation. This is not the only annotation with which is possible to create collections in Hibernate. Indeed there is also the annotation `@ElementCollection` but it has one main drawback: it does not allow to update or to delete an item stored into the database because Hibernate does no create a primary key for the collection tables so the system can not identify which item has been changed. On the other hand, a `OneToMany` relationship in Java is where the source object has an attribute that stores a collection of target objects and if those target objects had the inverse relationship back to the source object it would be a `ManyToOne` relationship. All relationships in Java and JPA are unidirectional, in that if a source object references targets object there is no guarantee that the target object also has a relationship to the source object. Moreover, all the mandatory fields are characterised by the not null annotation.

```

@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@Table(name="poi")
public class POI implements Serializable{
    private static final long serialVersionUID = -6503864251024996729L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotEmpty
    private String name;
    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL, orphanRemoval=true)
    @JsonManagedReference
    @Column(name="description")
    private Set<Description> description= new HashSet<Description>();
    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL, orphanRemoval=true)
    @Column(name="voting")
    @JsonManagedReference
    private Set<Votes> voting= new HashSet<Votes>();
    @NotEmpty
    private String type;
    @NotNull
    private double latitude;
    @NotNull
    private double longitude;
}

```

Figure 51. Declaration of the POI entity

For each class of the project, two support classes implement the access to the database: one repository class and one service class. These two classes, combined together, manage the database interactions regarding a specific class, as storing or modifying objects. In order to add or modify City,

Barrier and POI entities I added six different classes, two for each entity: Repository-Service class for POI, Repository-Service class for Barriers and Repository-Service class for City. Repository classes work at the persistence layer and acts as database repository. Indeed, this class implements the direct communication functions between the client and the database, querying that using the HQL or the EntityManager. The Service class is an abstraction level above, using repository methods to elaborate more complex algorithms and working on the service layer. These two modules together provide a large set of functions recalled in all the server code as getting a list with all the entities stored, retrieve a single entity from the database or update an existent entity. In this way a high abstraction prospective is maintained, because the single requests handled by the Hibernate Controller do not access directly to the entityManager object, while they access to it through other entities that are lower level.

The communication between the web client and the server is based totally on JSON representation. Indeed, the client uses AJAX post requests to send the data to the server, while the server uses the Jackson library to convert the entire list of elements retrieved from the Service. This can be done through the ObjectMapper entity, which can parse a JSON from a string, stream or file, and create an object graph representing the parsed JSON. Through this approach, the original classes must be rebuilt once they have been received: for example the client receive an JSONArray of elements representing all the stored POI and in order to rebuild the original entity it has to iterate across that array and to recreate all the objects locally, initiating new POI objects on the client machine and caching them on a variable. A characteristic of the web client is that it provides two full list that show all the downloaded entities of interest. This has been done by associating to each downloaded entity an HTML block of the list. In order to create classes in JavaScript I used the function approach, creating a function object for each entity of interest. The next step is the marker creation, showed in the Fig. 52. It is indeed done by iterating through the saved collection and categorizing the contained entities basing on their category and their vote. The upload of an entity of interest is possible by retrieving all the inserted value from the input fields of the HTML page and sending them to the server as JSON.

```

if(POI.type.toLowerCase()=='food'){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:food_icon_yellow]).bindPopup(ult_html,{minWidth:280,maxWi
}else if(POI.type.toLowerCase()=='accomodation'){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:accomodation_icon_yellow]).bindPopup(ult_html,{minWidth:2
}else if(POI.type.toLowerCase()=="entertainment"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:entertainment_icon_yellow]).bindPopup(ult_html,{minWidth:
}else if(POI.type.toLowerCase()=="bar&nightlife"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:bar_icon_yellow]).bindPopup(ult_html,{minWidth:280,maxWi
}else if(POI.type.toLowerCase()=="toilets" ){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:toilets_icon_yellow]).bindPopup(ult_html,{minWidth:280,m
}else if(POI.type.toLowerCase()=="supermarket"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:supermarket_icon_yellow]).bindPopup(ult_html,{minWidth:28
}else if(POI.type.toLowerCase()=="education"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:education_icon_yellow]).bindPopup(ult_html,{minWidth:280
}else if(POI.type.toLowerCase()=="governmentservices"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:diplomatic_icon_yellow]).bindPopup(ult_html,{minWidth:280
}else if(POI.type.toLowerCase()=="sport"){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:sport_icon_yellow]).bindPopup(ult_html,{minWidth:280,maxW
}else if(POI.type.toLowerCase()=="tourism" ){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:tourism_icon_yellow]).bindPopup(ult_html,{minWidth:280,m
}else if(POI.type.toLowerCase()=="transport" ){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:transport_icon_yellow]).bindPopup(ult_html,{minWidth:280
}else if(POI.type.toLowerCase()=="health" ){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:health_icon_yellow]).bindPopup(ult_html,{minWidth:280,max
}else if(POI.type.toLowerCase()=="shop" ){
    var marker = L.marker([POI.latitude,POI.longitude],[icon:shop_icon_yellow]).bindPopup(ult_html,{minWidth:280,maxWi
}
}

```

Figure 52. Marker creation

On the Android client, instead, I used the Robospice module to retrieve the information needed. As explained before, it is a library that allows making asynchronous requests that can be cached as JSON and that guarantee reliability, because the network requests are implemented using a Service, guaranteeing a proper integration with the Activity lifecycle. Using this approach, the entities transmitted are not JSON representation, but directly POJO classes, maintaining a high level approach.

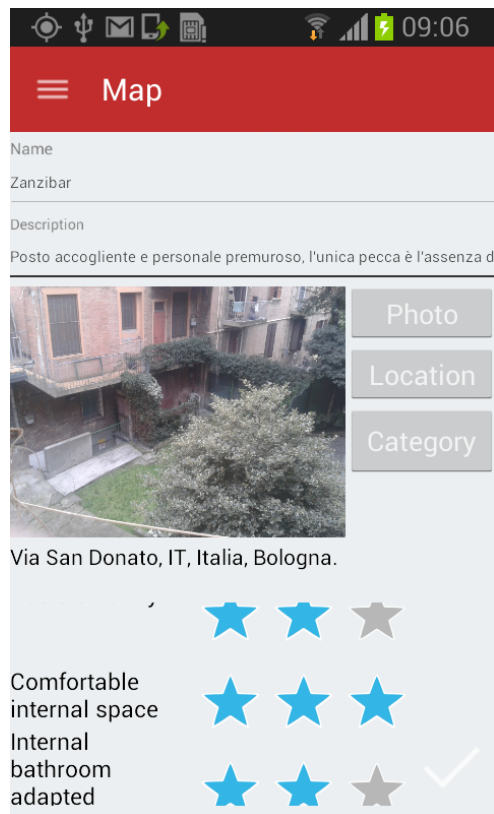


Figure 53. Mobile implementation of the upload screen

For this reason, all the entities sent extends the Serializable Interface, allowing to rebuilt a transmitted entity on the destination device using its unique Serialization ID. Using Robospice, I have created one request class for each request that the client must send to the server. The common characteristic of all these requests is that is possible to send a unique input value, declared as HttpEntity and inserted into the Http request. Moreover, all the request get cached into the device memory because the cacheExpiryDuration field, which declares the duration in milliseconds after which the content of the cache will be considered to be expired, is set as DurationInMillis.ALWAYS : this means data in cache is always returned if it exists, while DurationInMillis.NEVER would have meant that data in cache is never returned. In this way, the system reduce its memory consumption, sending a request only if the data are not present into the cache. So, both for upload or download a set of information, client and server just exchange a single variable, in which for the downloading phase correspond to a full list of the needed elements, while for the uploading phase correspond to a single entity. Indeed, I created two different fragments for implement the creation of a new entity of interest, one for the POI entity, showed in the Fig. 54, and the second for the Barrier. These layouts allow the user to insert all the sensible data that characterise an entity and verifies if all the fields are properly filled warning the user if some mandatory fields are not filled. If all the fields are properly set, it merges all the information extracted creating a new entity and sending it to the server, which will manage the received resource to merge it into the database.

```

super(R.layout.custom_marker, mapView);
btn = (Button)(mView.findViewById(R.id.bubble_moreinfo));
list=(ListView)(mView.findViewById(R.id.bubble_list));
image= (ImageView)(mView.findViewById(R.id.my_image));
vote=(TextView)mView.findViewById(R.id.bubble_description);

btn.setOnClickListener((view) -> {
    // adding comment, open the POI ACTION FRAGMENT
    Toast.makeText(view.getContext(), "Button clicked"+mSelectedPoi.getName(), Toast.LENGTH_LONG).show();
    FragmentTransaction Transaction = getActivity().getSupportFragmentManager().beginTransaction();
    MapActionFragment aDifferentDetailsFrag = new MapActionFragment();
    aDifferentDetailsFrag.setPoint(actual_position);
    aDifferentDetailsFrag.setTo_review(mSelectedPoi);
    aDifferentDetailsFrag.setMy_pois(my_pois);
    Transaction.replace(R.id.content_frame, aDifferentDetailsFrag);
    Transaction.show(aDifferentDetailsFrag);
    Transaction.addToBackStack(null);
    root.setVisibility(View.GONE);
    Transaction.commit();
});
}

```

Figure 54. Custom marker view

5.4 Geoposition

One of the key aspects of this project is the localization of the user. On this operation indeed depends all the software workflow. Indeed, if the system would not retrieve the current user location, all the system does not work correctly, because it cannot identify the city area to query. In order to guarantee a correct operation of the whole system, two main geolocalization operations are needed: the geocoding and the reverse geocoding.

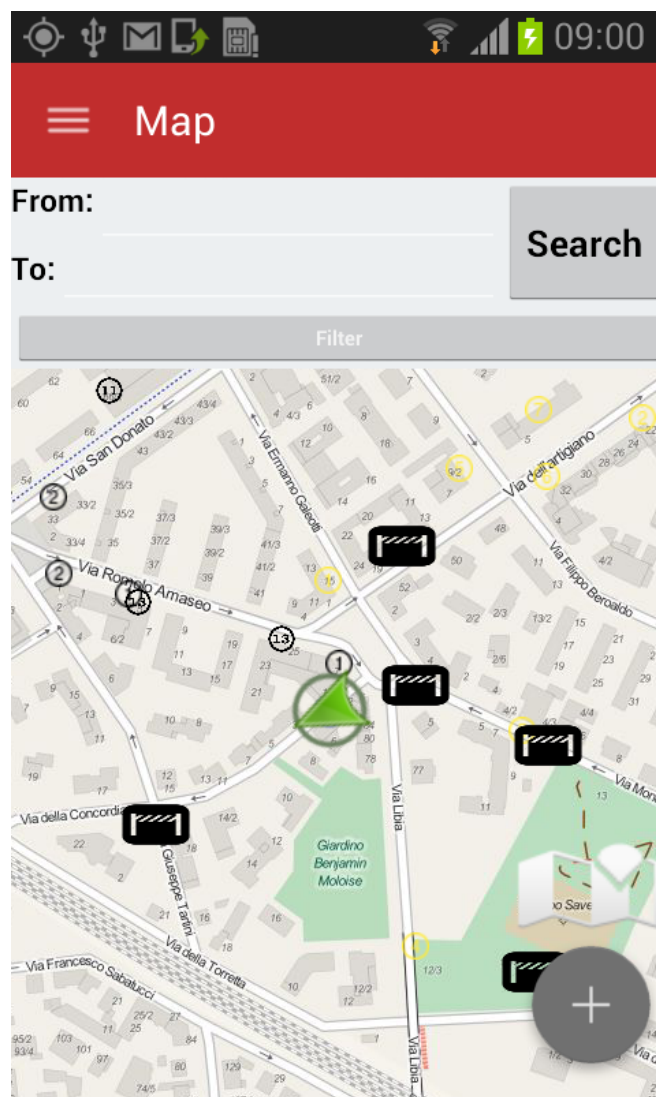


Figure 55. Mobile map implementation, in which is notable the user position

Both functionalities are implemented on both clients, using different tools. On the Android phones I have initialized a LocationManager, which allows

to check the actual system status regarding location services. Because this project needs a high precision location, the preferred location method is based on the GPS provider, even if it brings a higher resource consumption. If this provider is not enabled or not available, the application will rely on the NETWORK provider, but it does not use the PASSIVE provider, due to its low accuracy level. Moreover, in order to follow the movement of the user, I have implemented a `LocationListener` that updates the current location of the user if there is a difference of 20 m each 20s. In order to have the same effect on the map view, I have used an `osmbonuspack` library element called `MyLocationOverlay`, centering the map on the current location of the user.

On the browser client, the Leaflet library provides a location method that works as a set of calls: indeed, it keeps calling the location method until it does not receive the command to stop the location searching. Indeed, in order to get an accurate location, it tries to locate the user as a loop. For this reason, once that the system has retrieved the user is important to interrupt this request flow by the `stopLocate()` method, preserving resource. Moreover, I have set the high accuracy on, because as default the library turns it off.

All the calls needed to retrieve the needed information, as geocoding or reverse geocoding, are executed by sending a request to an external API, which could be Nominatim or the Android implemented one, and then managed to the current client, setting the required fields.

5.4.1 Geocoding

As explained in previous sections, the geocoding operation retrieves the coordinates of a particular place declared as text. This operation is used in both clients, especially for the search function. Indeed, the system receives as input the name of the place to display on the map, so the system must use a geocoding service to obtain the coordinates in which it has to focus on. In this project the most used provider is Nominatim, which is called both from the browser client both from the mobile one.

It is particularly useful in the Android application for the routing function. Indeed, this feature is fully based on geocoding, because it receives the starting point and the destination point as text and in order to execute the road calculation request, they must be converted into GeoPoint, done by coordinates. Moreover, the Nominatim Geocoder allows to retrieve the OSM ID of a particular node or street and this is necessary for inserting the correct fields into the request to MapQuestApi.

5.4.2 Reverse Geocoding

The reverse geocoding does the opposite operation, it indeed finds the description of a location receiving as input the coordinates. This service is used in both clients in the most localization services. Indeed, it is used in the main location function to retrieve the name of the current city area, because the localization services both on Android and on the browser retrieve the user location as coordinates. This operation is necessary in order to select the POI to ask to the server, because they are classified from belonging areas.

It is also mainly used into both clients when the user chooses the location for a new entity of interest: indeed, both mapview retrieve the location information as coordinates and in order to implement an intuitive interface, the view will display that coordinates set as the name of the selected street, by calling the Nominatim API and obtaining the referred name.

5.5 Experimental measurements

This project has been tested on two main platforms: a browser and an Android phone. The phone used is a Samsung SIII mini, which is provided by a Dual core processor with 1.0GHz with 1GB of RAM. At first, I have monitored the Memory spent during the initial phase, while the application

has to download all the needed element of interests from the server. As it is shown in the Fig. 56, it has a pick in the point in which the client has to manage all the received information, creating a different marker for each one of them. The first picture shows the memory usage, while the second one shows the CPU. The peak uses around 50% of the CPU, while the memory pick is about 24MB. Indeed, in the blue graph, the darker part of the graph represent the memory used, while the light blue one represent the memory hold free by the system.

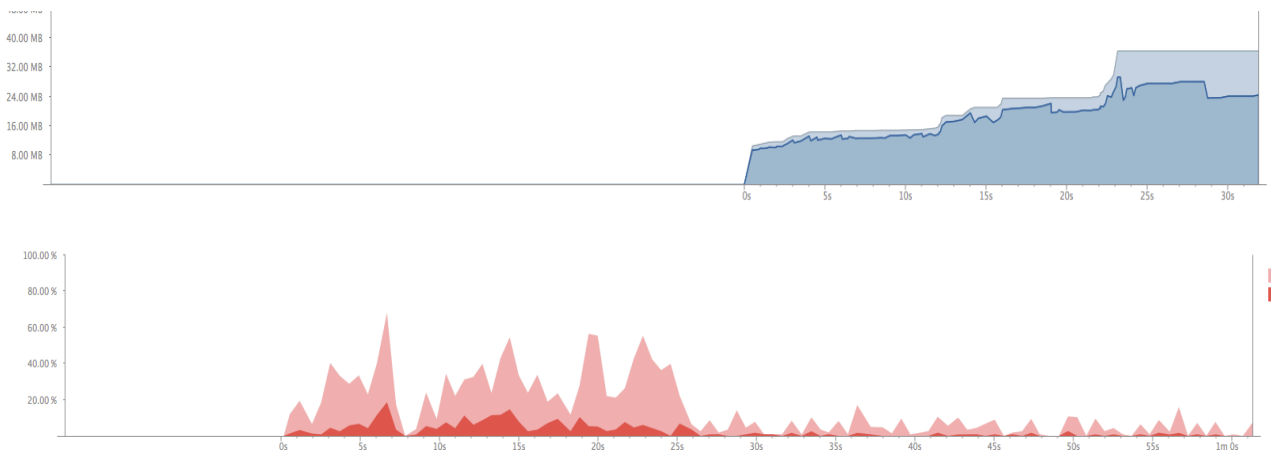


Figure 56. Initial phase

Anyway, the mobile application focuses at the first launch on the current position of the user that in my case was an area with a middle number of elements of interest. Focusing on the area with most elements I have registered the graphs in Fig. 57.

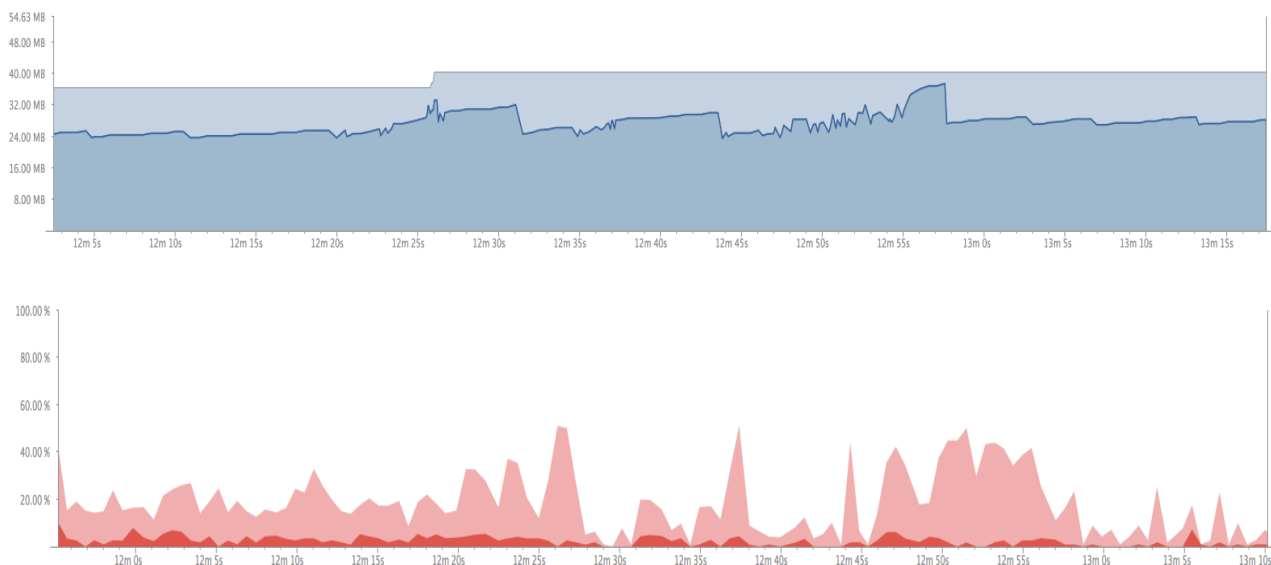


Figure 57. Resource consumption

As we can notice, the memory consumption is the same of the Fig. 56, while the CPU usage is higher. The main difference between the two areas is the number of visualized entities and their composition. Another monitored operation is the photo load, which is represented in the Fig. 58.

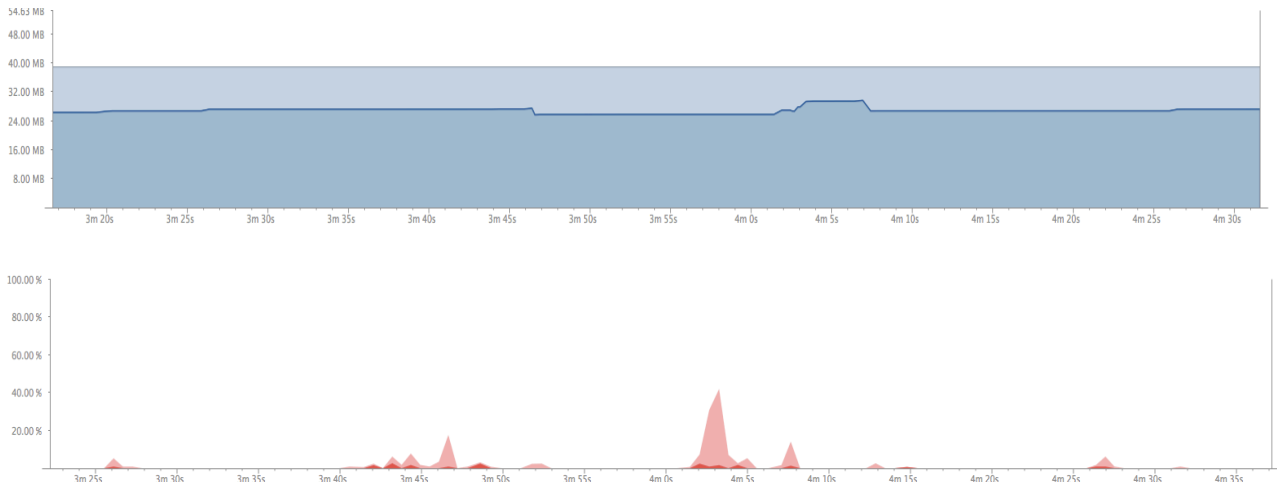


Figure 58. Photo loading

Indeed, loading Bitmap could cause a crash of the application due to the extreme memory consumption. As we can notice from the Fig. 58, this application can handle correctly the Bitmap loading, which correspond to the operation done between 4m and 4m 5s.

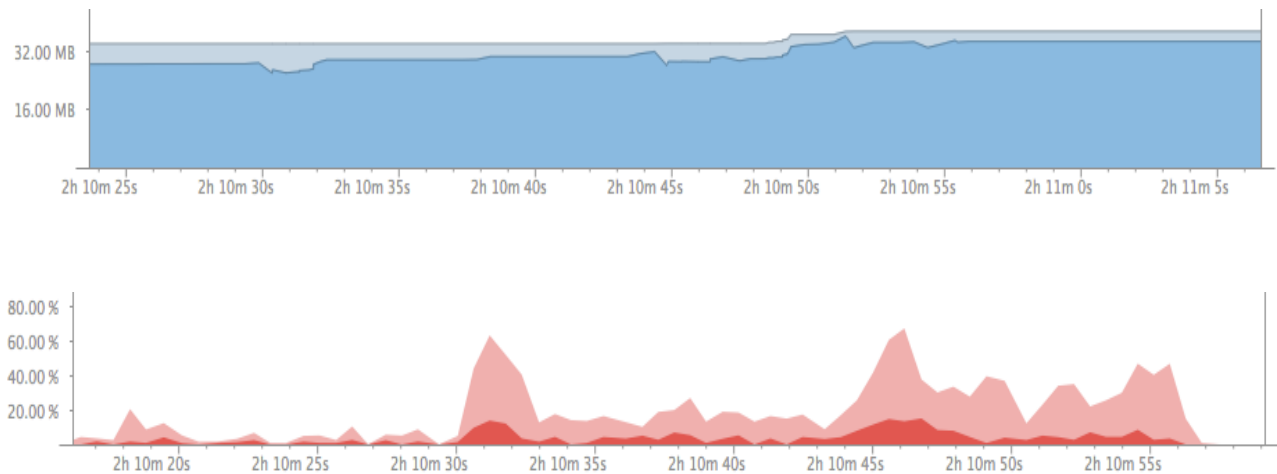


Figure 59. POI upload

The Fig .59, instead, represents the resource consumption while uploading a POI entity that contains an image. As we can notice, this operation brings the highest resource consumption, because after the upload the application will retrieve the new elements to display on the map. Because of that I have also monitored the network resources sent during this operation, which

embrace the two most significant network operations. The results are showed in the Fig. 60, showing a maximum upload speed of 1.1Mb/s and a download speed of 2.1 Mb/s.



Figure 60. Network resources

Instead, getting through the web client, it has been tested on two different browsers, Mozilla Firefox and Google Chrome. Monitoring the start phase in both browsers, I have noticed that Mozilla Firefox is faster than Chrome, as shown in the Fig. 61.

▲ 304	GET	bootstrap.js	localhost:8080	js	60,43 kB	60,43 kB	→ 36 ms
▲ 304	GET	bootstrap.min.css	netdna.bootstrapcdn.com	css	24,74 kB	119,67 kB	→ 57 ms
▲ 304	GET	leaflet.js	cdn.leafletjs.com	js	122,76 kB	122,76 kB	→ 211 ms
▲ 304	GET	bootstrap.css	localhost:8080	css	124,43 kB	124,43 kB	→ 58 ms
● 200	GET	leafletmap	localhost:8080	html	148,83 kB	148,83 kB	→ 1216 ms
▲ 304	GET	jquery-1.9.0.js	localhost:8080	js	261,05 kB	261,05 kB	→ 13 ms
● 200	POST	data_pos	localhost:8080	json	546,09 kB	546,04 kB	→ 1678 ms

<input type="checkbox"/> addcity	POST	200	xhr	jquery-1.9.0.js:8475	149 B	16 ms	
<input type="checkbox"/> data_pos	POST	200	xhr	jquery-1.9.0.js:8475	547 KB	1.97 s	<div style="width: 100%; height: 10px; background-color: green;"></div>
<input type="checkbox"/> data_barrier	POST	200	xhr	jquery-1.9.0.js:8475	2.7 KB	48 ms	<div style="width: 10%; height: 10px; background-color: green;"></div>
<input checked="" type="checkbox"/> barriers_icon.png	GET	200	png	Other	(from cache)	2 ms	<div style="width: 10%; height: 10px; background-color: blue;"></div>
<input checked="" type="checkbox"/> data:image/jpeg:bas...	GET	200	jpeg	jquery-1.9.0.js:6040	(from cache)	1 ms	<div style="width: 10%; height: 10px; background-color: blue;"></div>

Figure 61. Comparison between Mozilla Firefox and Chrome

This difference in average is 0.3s, which is particularly significant. Indeed, by the method data_pos, the clients retrieve the needed entities and it influences all the performance of the map. Despite this, the Chrome map interaction is more fluid than the Mozilla one. Due to the different development of the browsers, the webpage brings to different memory consumptions. On Mozilla I have used the add-on Tab Memory Usage 0.2.2 in order to get the memory consumption and the average consumption is about 30mb, with some peaks during the calls of 60mb. On Chrome,

instead, the JS Heap picks to 36.4 MB as shown in Fig. 62, using the developer tools built into the browser.

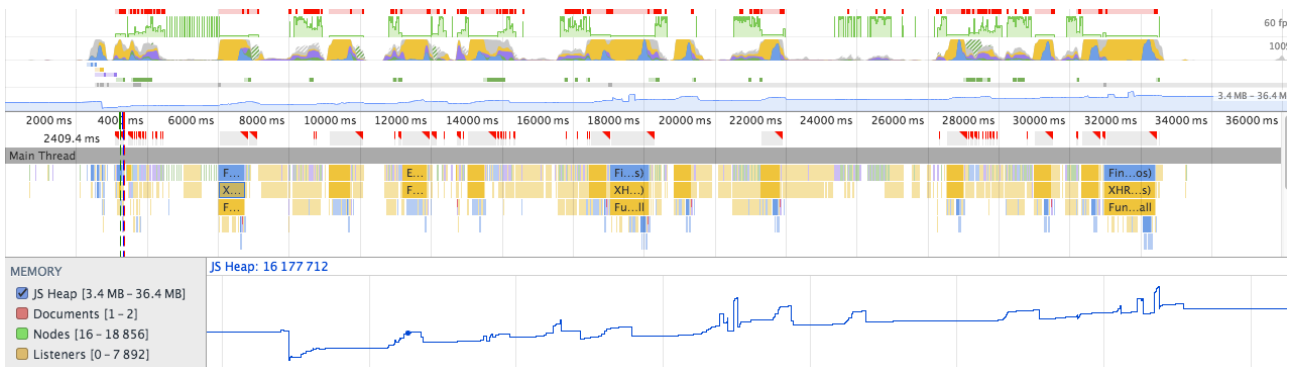


Figure 62. Memory measurements on Chrome

On the other hand, because Chrome subdivides each tab into a different process, using the Task Manager of Chrome, I have monitored a memory usage of 250MB in average, with some peaks at 300MB.

Moreover, the most heavy method is the one by which is possible to download POI from OSM and to save into the database. Monitoring it on both browser I have discovered an average time of 1 minute, depending on the city searched: more POI the city has, more time it gets.

5.6 Technical conclusions and future improvements

As it has been showed, the platform works correctly and a reasonable time has been spent on the debug part, testing the applications behaviour. Both clients are sufficiently fluid to allow a friendly user experience and the waiting phases are relatively short.

The memory consumption of the mobile application remains almost constant around 30mb, reaching the 35mb as maximum. Instead, the CPU usage is alternated by moments in which it is almost none and others in

which it covers around the 60% of the CPU. These are short periods of time that happens during the uploading phase and the downloading of the new data set, so this does not affect significantly the application performances. Anyway, asynchronous calls do not affect the user interface, allowing a smooth scrolling.

Analysing instead the performance of the web application has been noticed that the scripting part covers the largest part of the CPU activities, as shown in Fig. 63.

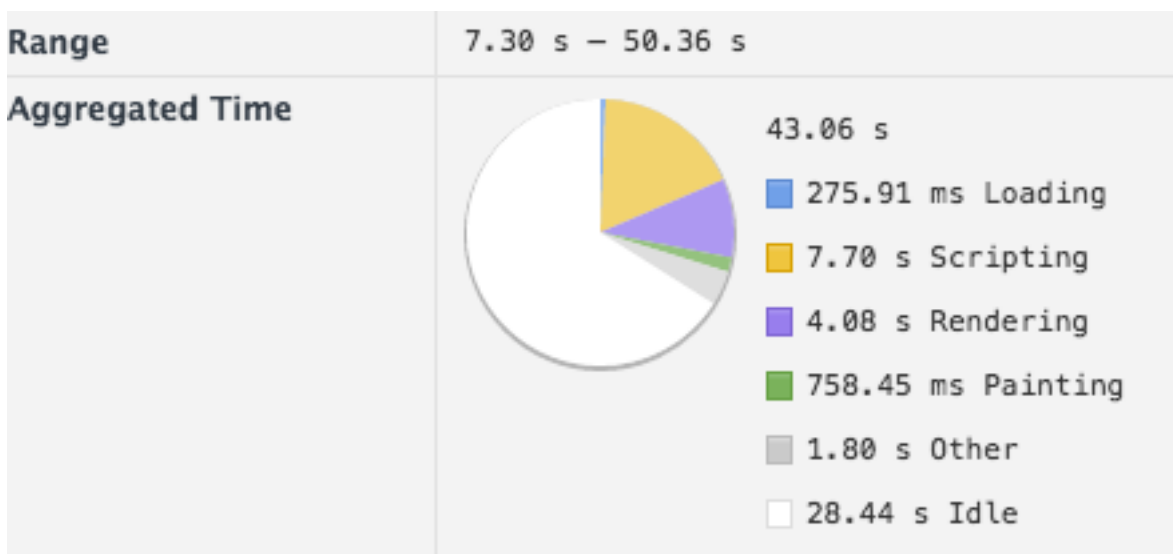


Figure 63. CPU usage summary

Moreover, the JavaScript Heap average size is around 20 MB as showed in Fig. 64, which is limited memory consumption, considered the set of data loaded. Indeed, on both browsers (Mozilla Firefox and Google Chrome), the map interactions are fluid and allow a dynamic representation system. Indeed, as shown in Fig. 65, the data exchanged between the clients and the server are low weigh, the showed method is the heavier and it transfer a data set around 170KB in average, and a lot of the needed files are retrieved automatically from the cache.

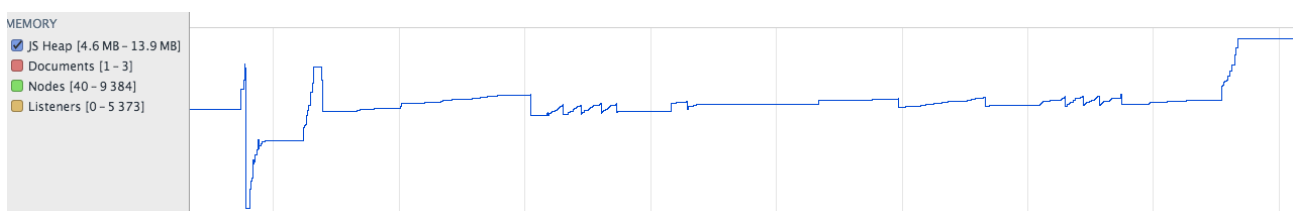


Figure 64. JS heap

data_pos	POST	200	xhr	jquery-1.9.0.js:8475	157 KB	1.52 s	
data_pos	POST	200	xhr	jquery-1.9.0.js:8475	158 KB	1.24 s	
data_pos	POST	200	xhr	jquery-1.9.0.js:8475	158 KB	1.80 s	
data_pos	POST	200	xhr	jquery-1.9.0.js:8475	165 KB	1.95 s	

265 requests | 1.0 MB transferred | Finish: 1.7 min | DOMContentLoaded: 377 ms | Load: 377 ms

Figure 65. Network heavier data transfers

Anyway, the platform is limited to a single city area and does not support the visualization of a whole country. This feature can be considered one of the main aspects as future deployment in order to obtain a flexible map that holds different zoom-level prospects. Moreover, in order to save space into the database, a way should be compressing the base64 images, reducing their size. This operation needs the implementation of compression and a decompression algorithm, the first for saving the file and the second one for retrieving it from the database. Finally, both applications should be more tested in order to find precisely the best threshold values both for the map scroll and for the maximum number of entities that the server can send.

Conclusions and Ongoing Work

The diffusion and development of mobile devices have experienced significant evolutions in the last years, both in application complexity and accuracy. These factors open new opportunities for the 'mobile sensing' area that becomes a common part of the people life. This thesis project focuses on this area and proposes an extension of the ParticipAct project, which has been designed around the goal of optimizing the smart cities life by adopting a crowdsourcing system. We had the goal of design a new experience for impaired people, integrating them into the city life.

In particular, our thesis makes possible for mobility impaired people to share experiences and data with other people, using our platform a map that links together all stored information. By using the new function, it is indeed possible to upload new entities and to comment existent ones, uploading comments and images. Both operations have been widely tested both on browser and on mobile client deployments.

Indeed, the crowdsourcing potential is wide, as explained in this document, granting more and more complex interactions between the crowd and any single application. That will help people in communication and in life, allowing an improvement and global feedback of the surrounding environment and of the knowledge of people. The concrete example of the smart cities, built on the interaction between the users and the city itself, shows the idea of retrieving the citizens needs and hints.

Unfortunately, our current platform is limited by focusing on a single city area, specifically the province. Indeed, the system cannot manage the load of an entire country, as for example the whole Italy. This will be probably part of the future development of the application, guarantying a flexible view that is not limited to the current city area. Another function that will be implemented in the future is the possibility to upload multiple images for a single comment. Indeed, by now, each user can upload only a single comment that is composed by a text and one image at most.

These features will amplify the application platform, guaranteeing a more flexible interaction with the end-users.

Bibliography:

- [1] Paul Deitel, Harvey Deitel, Abbey Deitel, Michael Morgano, “Developing Android application”, Pearson.
- [2] G. Cardone, A. Cirri, A. Corradi, L. Foschini, R. Ianniello, R. Montanari, “Crowdsensing in Urban Areas for City-Scale Mass Gathering Management: Geofencing and Activity Recognition”, IEEE Sensors Journal, vol. 14, no. 12, 2014.
- [3] G. Cardone, L. Foschini, P. Bellavista, A. Corradi, C. Borcea, M. Talasila, R. Curtmola, “Fostering ParticipAction in Smart Cities: A Geo-Social Crowdsensing Platform”, IEEE Communications Magazine, vol. 51, no. 6, 2013.
- [4] G. Cardone, A. Cirri, A. Corradi, L. Foschini, “The ParticipAct Mobile Crowd Sensing Living Lab: The Testbed for Smart Cities”, IEEE Communications Magazine, vol. 52, no. 10, 2014.
- [5] David Flanagan, “JavaScript guide”, O’Reilly, Apogeo.
- [6] Craig Walls, “Spring in action”, Manning.
- [7] Raghu K. Ganti, Fan Ye, Hui Lei, “Mobile Crowdsensing: Current State and Future Challenges”, IBM T. J. Watson Research Center, Hawthorne, NY.
- [8] Vogella, <http://www.vogella.com/>.
- [9] Android, <http://developer.android.com/>.
- [10] XML vs JSON, http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html.
- [11] PostgreSQL Global Development Group, PostgreSQL, , <http://www.postgresql.org/> .
- [12] Oracle, Java Persistence API, , <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html> .
- [13] Hibernate, , <http://hibernate.org/> .

