Table of contents

1 Introduction 1
2 Objectives and specifications 3
2.1 Objectives 3
2.2 Specifications 3
3 Bibliographic research 4
3.1 Technology and trends 4
3.1.1 Android OS 5
3.1.2 iOS 6
3.2 Developing applications 7
3.2.1 Native vs. web-based 7
3.2.2 Client and server side 7
3.2.3 Databases 7
3.2.4 Maps 7
3.3 Communication 7
3.3.1 OSI ISO 7
3.3.2 TCP/IP 7
3.3.3 HTTP and Sockets 7
3.3.4 Mobile communication 7
3.3.5 GPS and GLONASS 7
4 Analysis and design 8
4.1 Databases 9
4.1.1 Server side 10
4.1.2 Client side 11
4.2 Server 11
4.3 Mobile app 12
4.3.1 SDK 12
4.3.2 User interface 16
4.3.3 Activities 19
4.3.4 Services 22
4.3.5 Notifications 23
5 Implementation 24
5.1 Mobile app 24
5.1.1 User interface 24
5.1.2 Database 32
5.2 Server side 32

6 Testing 33
7 Conclusions 34
8 Bibliography 35
9 Dictionary 36
10 Acronyms 37
11 Annex 38

1 Introduction

Over the last century, technology has evolved massively and has integrated in every aspect of our daily lives. Humanity has gone from black and white image reproduction to accesing the biggest part of human knowledge in seconds. Gordon E. Moore, the cofounder of Intel Corporation, stated in 1965 that every two years the number of transistors in integrated circuits would double. So far this has been true, but the future gets more challenging due to decreasing nano sizes.

With this evolution maintaining its momentum, we are likely to reach the technological singularity in less than a century. This concept has been issued by Raymond Kurzweil, an american inventor and director of engineering at Google, and represents a moment in time when artificial intelligence will exceed all the human intelligence. Even though this sounds like something you would rather read in a book written by Asimov, it is projected that the brainpower equivalent of all human brains combined will be surpassed somewhere near 2045 (Figure 1.1)

In current days, most people carry in their pockets a smartphone that has more processing power than Apollo 11, the spaceflight used to land on the Moon. These devices offer increased connectivity and portability of processing power. The use of sensors, such as GPS, accelerometer, gyroscope or pressure sensors make the mobile experience a more enjoyable one. The mobile experience entered a new stage a few years ago, when tablets were released. Having larger screens and better performances, they are slowly entering a stage where they will rival laptops and desktops.

At the same time with market growth of mobile industry, the needs of consumers are growing. GPS localization provides regular users with a lot of real world orientation possibilities. Although there are many entertainment and practical apps that use positioning, hiking paths in Romania have little or no coverage. This paper follows the creation of a touristic orientation system that can be used on Android devices on hiking trails. Using GPS and caching, the user can see the map with nearby trails and his current location.

The app consists of a client side made in native Android with multiple activities and a server side that keeps a record of all trail locations, made using PHP. Data is stored on the client side in an SQLite database, while on the server side it is stored using MySQL. Here are retained all paths by storing a list of GPS coordinates. These are retrieved by the client and stored in his local database. The maps on the client side are displayed using OpenStreetMap API. Due to the fact that there is little or no signal and internet connection in many areas, the maps must be cached before usage.

Because of the incresed number of hiking trails all around Romania, the app is thought similar of an open source system. Multiple users can record tracks. In offline mode, the app has the option on the client side to add track details and then register GPS coordinates every at a fixed time interval. This data is stored in the local database, and can be uploaded on the server when internet becomes available.

The application and its location capabilities have been tested in Cluj-Napoca. Considering in open areas the GPS signal is stronger than in between buildings, it should work flawlessly on isolated hiking trails. More details can be found in the testing chapter.

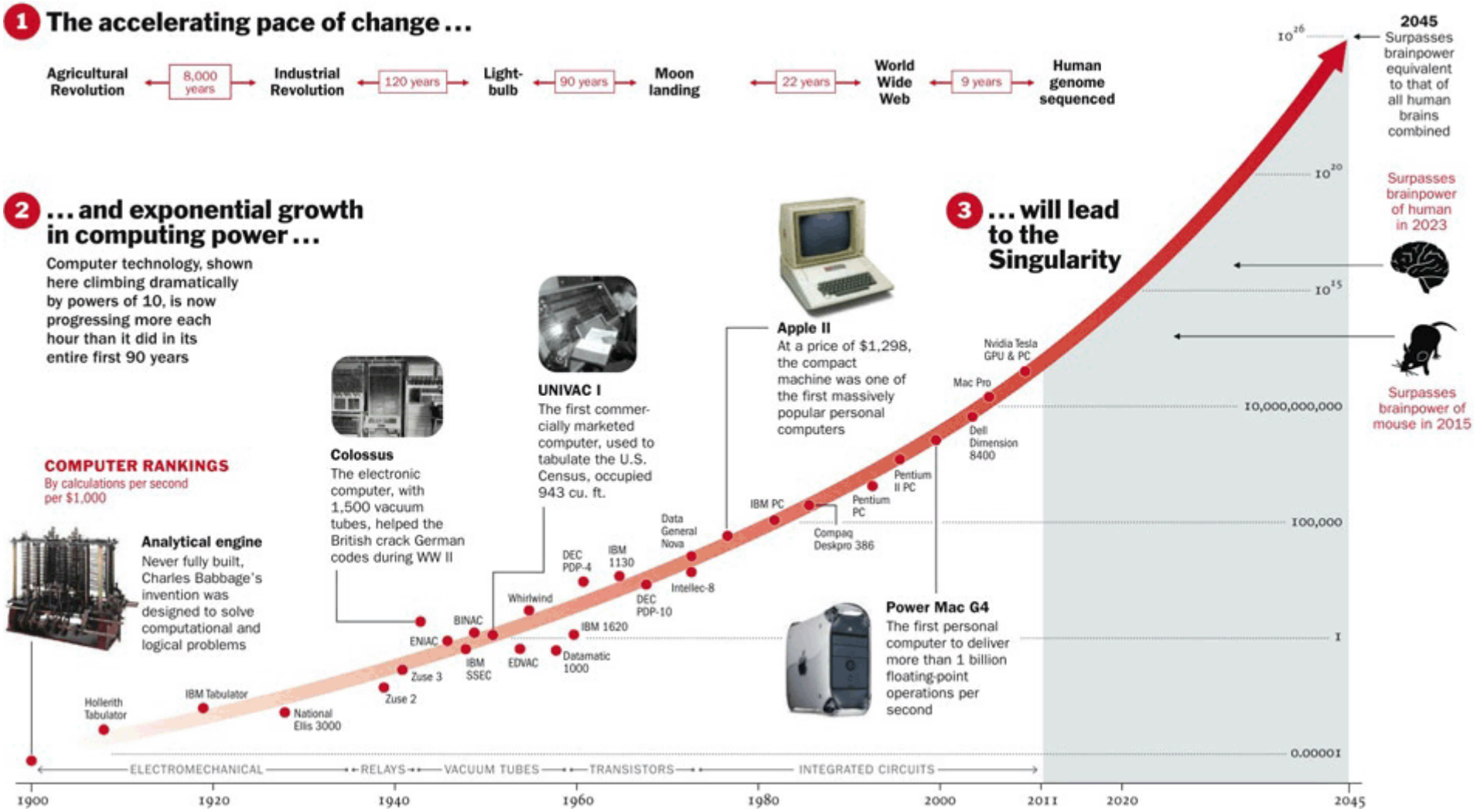

Figure 1.1 – Processing power evolution

2 Objectives and specifications

2.1 Objectives

As stated in the previous chapter, this paper follows the creation of a touristic orientation system that can be used on Android devices without an internet connection, to provide information about trails and location. The objectives are to:

- Provide an electronic portable solution for keeping information about touristic trails in Romania. This can be a more convenient way than carrying multiple physical maps.
- Provide accurate orientation capabilites, using GPS. Orientation on physical maps can be hard to master, especially when there are no relevant landmarks nearby or when the direction of North is unknown.
- Provide an open system for users to get and save information about hiking trails in Romania. This can be made by interacting with the server, which keeps its data in a database.
- Provide record capabilities for users, so they can acquire new information about trails. These records can be then shared with the rest of the users by putting the newly created maps on the server. For security reasons, only users that have an account can save maps to the server.

2.2 Specifications

For the system to achieve all these things, there must be implemented a software for clients and one for the server to keep and interact with data.

The client software represents an application for mobile devices that can be used on hiking trails, without internet connection, to display tracks and location information. This software must have the following features:

- ✓ Provide current location
- ✓ Provide track data
- ✓ Record new track data
- ✓ Get track data from the server (when internet connection is available)
- ✓ Put track data to the server (when internet connection is available)

The server software must keep track data and allow interaction with users. This software must have the following features:

- ✓ Keep track data in the database
- ✓ Keep user data in the database
- ✓ Allow users to login and save tracks
- ✓ Send track data when requested
- ✓ Save track data when requested

3 Bibliographic research

3.1 Technology and trends

Although smartphones have been on the market for more than 15 years, mass adoption happened after Apple released iPhone, the first smartphone with capacitive touchscreen. Google entered the mobile market one year later by acquiring and further developing the Android mobile operating system. The competition between these two companies increased over the years, both in marketshare and profit. The rapid rate of adoption led, in some countries, to a smartphone penetration of around 70%.

Similarly to many other brands competing for supremacy, iOS and Android are constantly developing and challenging one another. It is likely that no iOS user will use Android and vice-versa. There are differences between the two. For a regular user, using iOS is easier and more intuitive, while Android offers more customizable options. Also for developers, Android offers an open platform and more hardware devices, even though some of them might be of lower quality than Apple's.

According to TechRepublic, Apple makes about three to five times more revenue than Google, but Android owns a larger marketshare than iOS. This can be seen in the figure below.

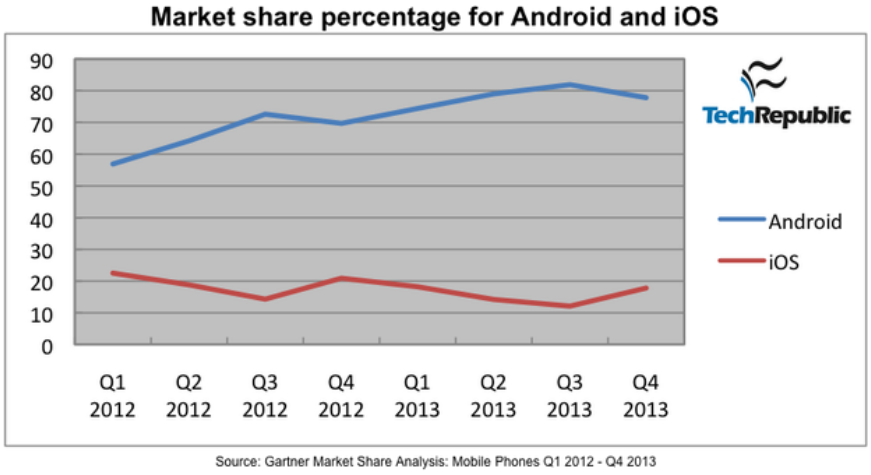

Figure 3.1 – Marketshare percentages

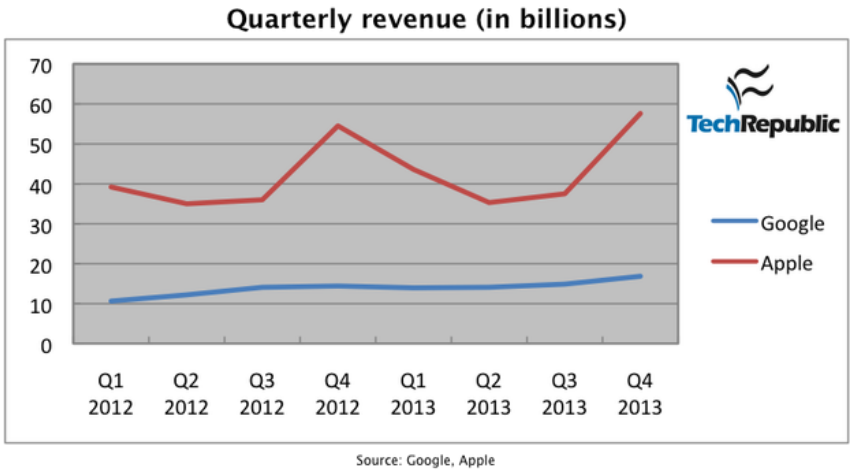

Figure 3.2 – Revenue for Google and Apple

3.1.1 Android OS

Android has been developed by Google since 2005. It is based on the Linux kernel and it is an open source operating system, anyone being able to use, distribute and modify the software. C, C++ and Java for the user interface were used to create this operating system. Since the launch of the first Android smartphone, the OS has evolved from version 1.0 (API 1) to 4.4 KitKat (API 19). According to the official site of Android, the most used version now is Android 4.1, 4.2 and 4.3 Jelly Bean (API 16, 17 and 18).

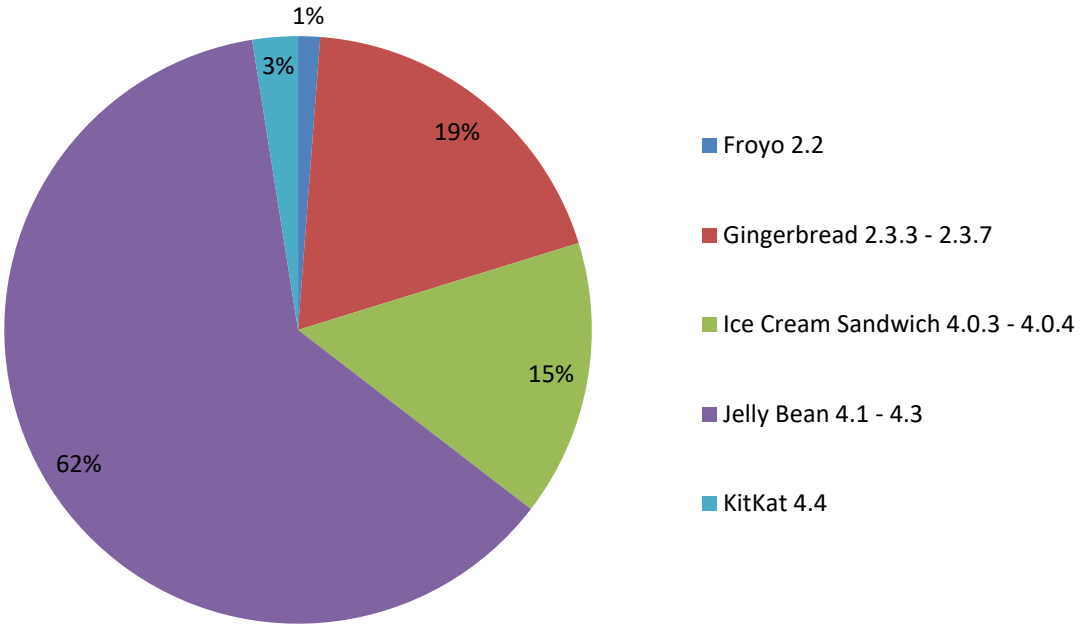

Figure 3.3 – Android version distribution

Android 1.0 (API 1) was firstly equipped on the HTC Dream smartphone. Many of the features it had were found in other phones, including Wi-Fi and Bluetooth support, camera or web browser. New elements were the Android Market, where the user could get apps and updates, synchronization with Google services and email servers or Google Maps for orientation. A smartphone running version 1.0 had a 500MHz processor, 192MB of RAM and 320x480 pixels screen size.

Android 2.2 Froyo (API 8), most used version in 2011, featured improvements like speed optimizations, support for car docks or Adobe Flash support. A smartphone running version 2.2 had a 1GHz processor, 512MB of RAM and 480x800 pixels screen size.

The newest versions of Android OS, 4.3 Jelly Bean (API 18) and 4.4 KitKat (API 19) evolved massively, together with the smartphones they run on. New features include 4K resolution support, a smart caller ID which searches unknown numbers online and wireless printing capability. A smartphone running version 4.4 has now a Quad-core 2.3GHz processor, 2GB of RAM and 1080x1920 pixels screen size.

3.1.2 iOS

iOS has been developed by Apple. It was first used on iPhone, but it expanded to other devices such as iPod or iPad. It is based on the Mac OS kernel and uses C/C++ and Objective-C compilers. Since the first iPhone, it has evolved from iOS 1 to iOS 7. According to the official website of Apple, the most used version is iOS 7.

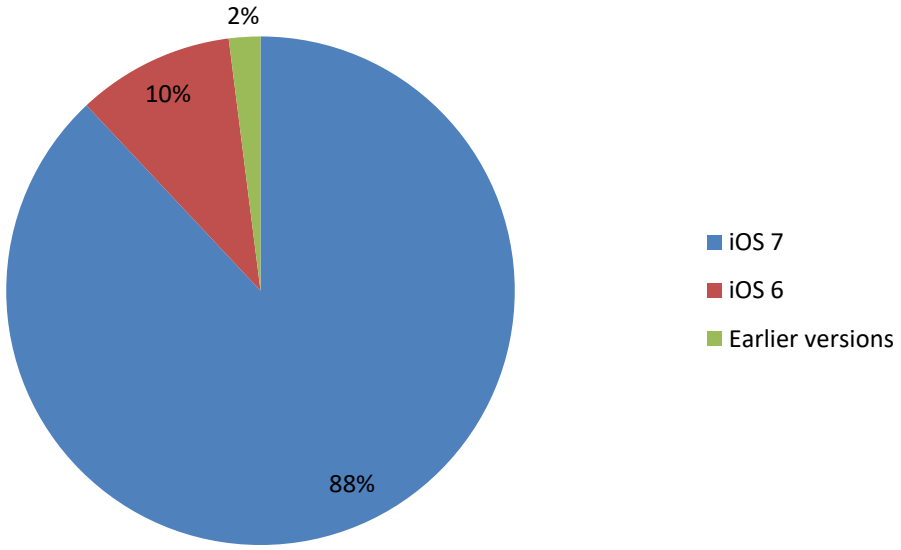

Figure 3.4 – iOS version distribution

iOS 1 was firstly equipped on iPhone 1. It distinguished itself by being the first phone with a capacitive touchscreen. Features of the OS included the Safari web browser, Google Maps or iTunes Sync. The smartphone had a 412MHz CPU and a screen resolution of 320x480 pixels.

iOS 4, equipped on iPhone 3G and iPhone 4, offered new features like multitasking, grouping applications into folders or creating mobile hotspots. The hardware included Retina Display, a 1 GHz processor, 512 MB of RAM and 640x960 pixels screen.

The latest version, iOS 7, evolved massively, including new features like Touch ID or Find My iPhone. It is equipped on latest devices like iPhone 5S, having a dual-core 1.3 GHz CPU, 1 GB of RAM and a 640x1136 pixels screen resolution.

3.2 Developing applications

3.2.1 Native vs. web-based

3.2.2 Client and server side

3.2.3 Databases

3.2.4 Maps

3.3 Communication

3.3.1 OSI ISO

3.3.2 TCP/IP

3.3.3 HTTP and Sockets

3.3.4 Mobile communication

3.3.5 GPS and GLONASS

4 Analysis and design

The specifications of the system impose a client-side application and a server-side application, both storing information about hiking trails and communicating between them. For the client side, I will develop the application in Android. On the server side, I will use XAMPP with MySQL and PHP.

XAMPP is a free web server solution, consisting of an HTTP server, database and interpreters. The acronym stands for:

- X for cross-platform
- A for Apache
- M for MySQL
- P for PHP
- P for Perl

At a first view, a sketch of the system would look like this:

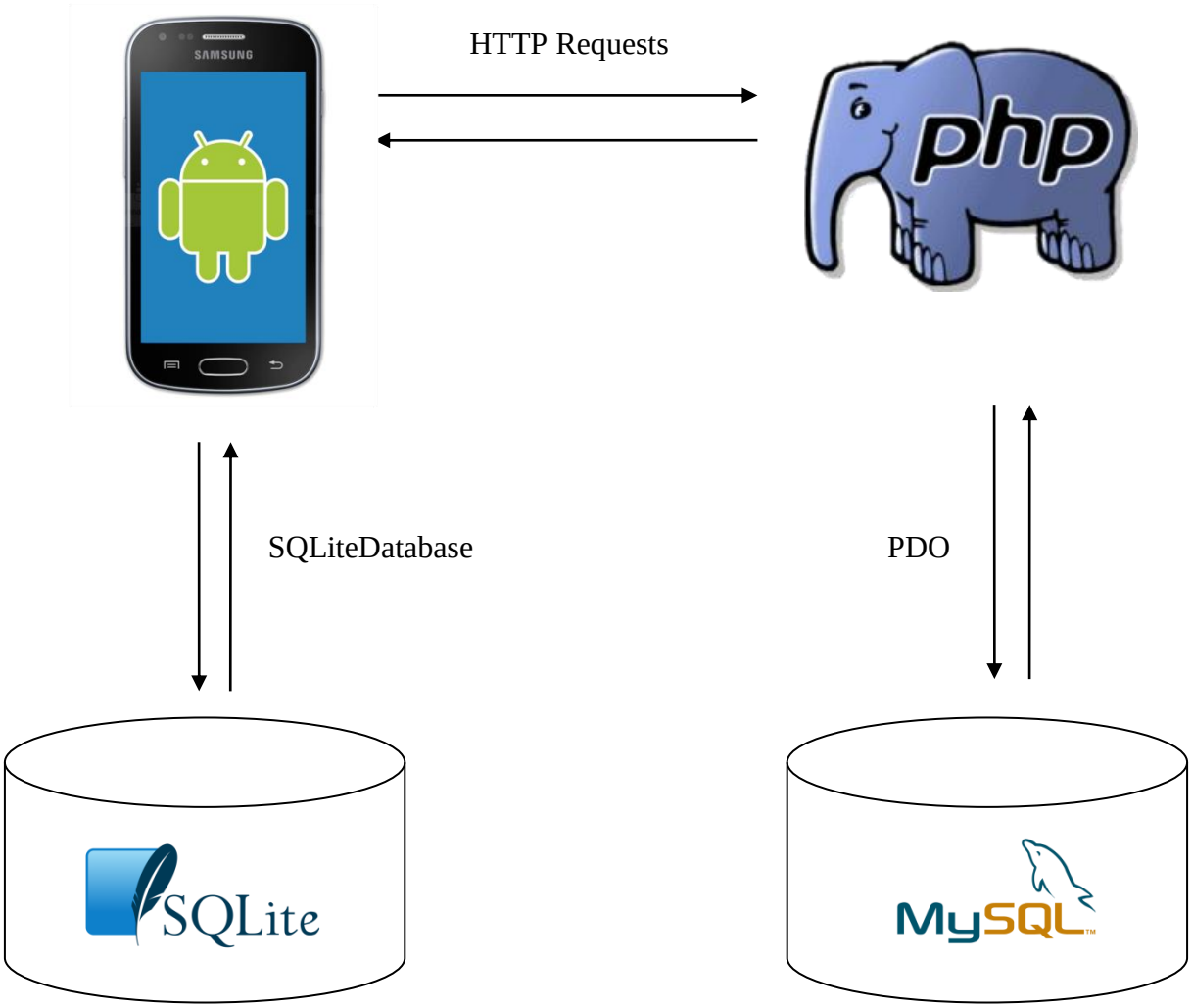

Figure 4.1 – Overall view of the system

4.1 Databases

Hiking trails in Romania have standard formats. The markers are usually painted on trees, rocks or special signs and link two or more touristic objectives. Their color is always red, yellow or blue and their size between 16 to 20 centimeters. Marker types are:

- Colored vertical line in a white square, representing a main track also called artery. The colored line must have 6 centimeters in width while the white ones 5 centimeters.
- Colored cross in a white square, representing a linking track. The widths are the same as the colored vertical line.
- Colored equal-sided triangle on white background, representing a secondary track. The colored side must have 10 centimeters and the width of the white part surrounding the triangle must be 3 centimeters.
- Colored dot on white background, representing also a secondary track or a circuit. The diameter must be 10 centimeters and the width of the white stripe, 3 centimeters.

The combination of 3 colors and 4 shapes gives 12 possible tracks for an area. Each track starts at one objective and ends at another.

Taking into consideration the rules stated before, the databases must contain the track start point, end point, shape of marker and color a marker. To better manage the tracks, we can also retain the track zone. Considering that the system is open for users to add tracks, it is also better to retain the user which added a certain track. For a track to be displayed on a map, a set of GPS coordinates must be retained.

If an admin wants to manage or check the tracks, it is useful to know which ones have been added recently. Thus, each track can have a flag that can signal this. Also on the client side, each recorded track should have a flag that indicates whether it has been already updated on the server on not. For better management when getting and putting tracks from the server, each zone can be assigned a version to be compared with the client version.

A diagram for the database would look like this:

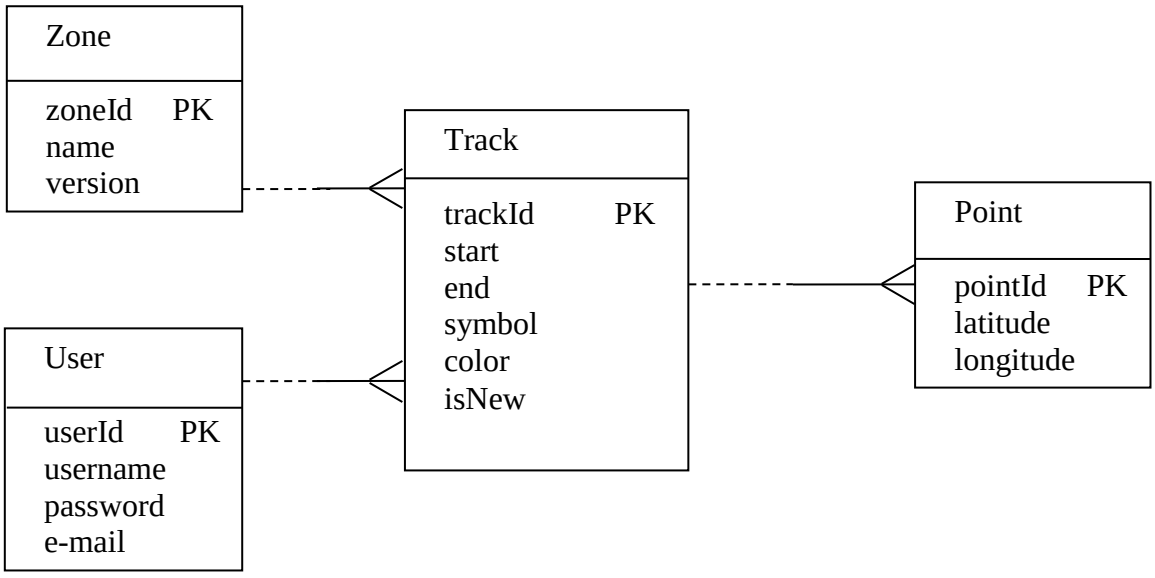

Figure 4.2 – Database ERD

4.1.1 Server side

MySQL is a suitable DBMS for the server-side, due to its ease of usage and reduced cost. The size of data is relatively low and it does not need to scale, as there is a fixed maximum number of hiking trails in our country.

The package includes the phpMyAdmin tool written in PHP for an easy manipulation and access to databases. This tool consists of a graphical user interface that allows a multitude of operations. Common operations that can be made include viewing, altering and deleting existing databases, adding, removing and updating tables, importing and exporting databases, performing SQL queries and more. PhpMyAdmin also offers a graphical view of the Entity Relationship Diagram, together with row types, primary keys and foreign keys.

PDO is a good solution for manipulating data through PHP. The acronym stands for PHP Data Object and supports multiple databases, including MySQL, Microsoft SQL Server, Oracle, PostreSQL or SQLite. Some of the most important methods are:

`PDO::beginTransaction`
- initiates a transaction

`PDO::commit`
- commits a transaction

`PDO::__contruct`
- creates an instance representing a connection to the database

`PDO::exec`
- executes a query and returns the number of affected rows

`PDO::lastInsertId`
- returns the last inserted id from the database

`PDO::prepare`
- prepares a query for execution and returns a PDOStatement object

`PDO::rollback`
- cancels a transaction

PDO can also make use of PDOStatements objects, which are prepared statements. After they are executed, they return an associated result set. Some of the most important methods are:

`PDOStatement::execute`
- executes the prepared PDOStatement

`PDOStatement::fetch`
- gets the next row in the result set

`PDOStatement::fetchAll`
- gets all rows in the result set and returns them as an array

`PDOStatement::rowCount`
- returns the number of rows affected by the query

4.1.2 Client side

Android platform contains an API for SQLite databases to create, delete, execute queries and perform any other common tasks. Databases are stored locally on the smartphone and are linked to the application that created them. Thus, their names must be unique only within the same application.

The Android OS does not have a standard feature for operations on local databases, but there are certain apps on Google Play that allow this, or this can be made remotely from the computer through ADB.

The class for the database is called *SQLiteDatabase*. Some of the most important methods of this class are:

```
delete(String table, String whereClause, String[] whereArgs)
```
- deletes rows from a table

```
insert(String table, String nullColumnHack, ContentValues values)
```
- inserts rows in a table

```
openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory
factory)
```
- opens the dabase if it exists, otherwise creates it

```
rawQuery(String sql, String[] selectionArgs)
```
- performs a query and returns a Cursor object over the data set

```
update(String table, ContentValues values, String whereClause,
String[] whereArgs)
```
- updates rows in a table

4.2 Server

The server must interact with the users and with the data. XAMPP offers an Apache server that support PHP and Perl scripts. As there is no need for a complex or scalable solution, I will implement the server side in PHP.

The server must support the following interactions with the user:

- Receive a request for tracks and respond with track data
- Receive a request to save a track together with track data
- Receive a request for login and respond with user data

This interaction between server and client can be made through HTTP requests. PHP offers access to the HTTP request through some arrays of variables. For example, when a POST request is made to the server, PHP can interpret data by accessing $_POST. The following variables are declared for HTTP requests:

$_GET – HTTP GET variables
$_POST – HTTP POST variables
$_FILES – HTTP File Upload variables
$_REQUEST – HTTP Request variables

4.3 Mobile app

4.3.1 SDK

The Android SDK from Google comes with a variety of tools that help build applications. The environment for development is Eclipse IDE with Android Developer Tools (ADT) plugin. This also integrates tools such as SDK manager or Virtual Devices Manager. The emulator comes handy for testing and debugging, but due to its relatively slow speed and difficult integration with maps, I will use an Android device for testing.

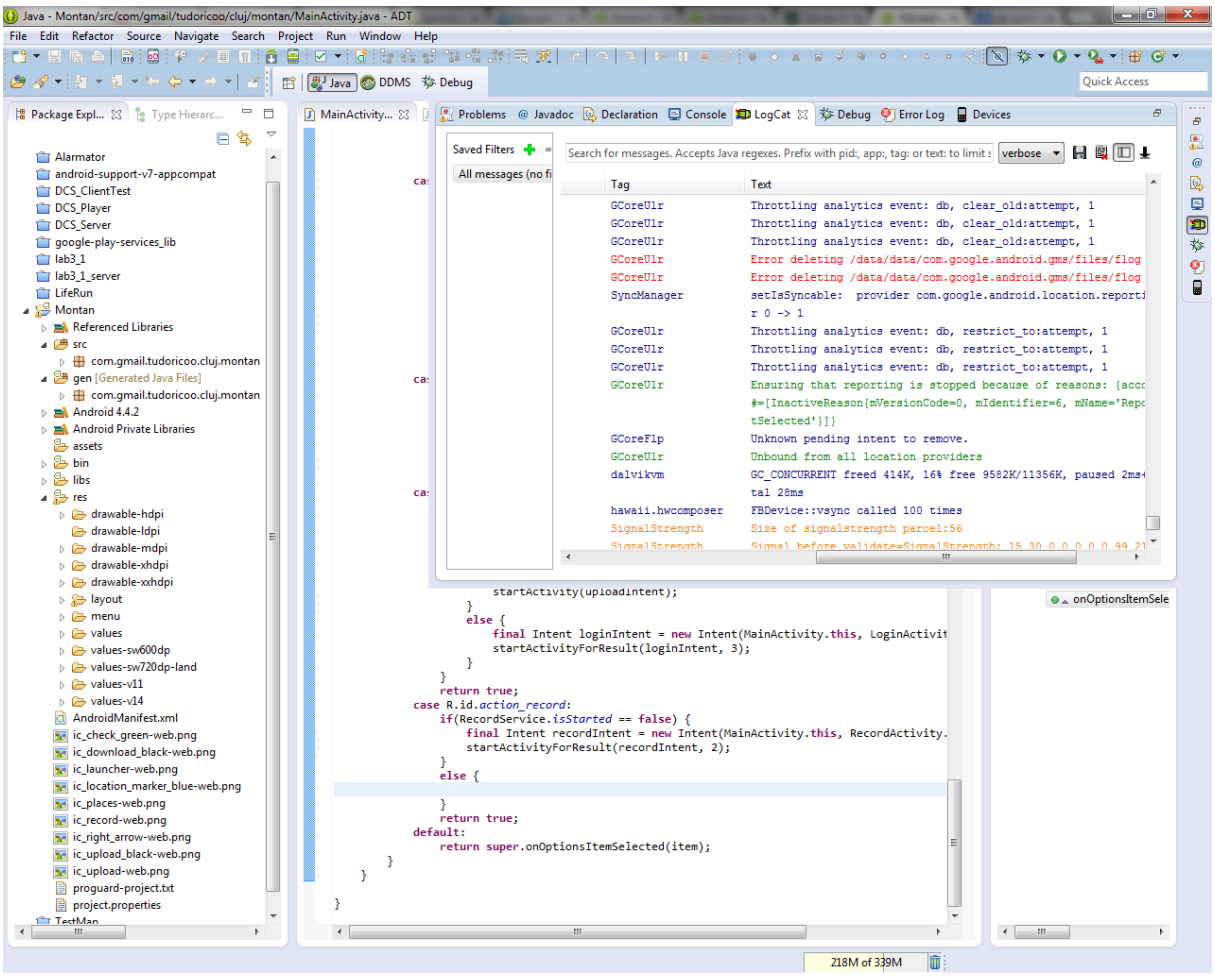

Figure 4.3 – Eclipse with ADT

The specifications of the development device can be seen below.

Table 4.1 – Development device specifications

Device name and model	Samsung S7580
Standard battery	1500 mAh
Internal memory	4 GB
RAM memory	768 MB
OS	Android 4.2, Jelly Bean
Processor clock	1.2 GHz
Location capabilities	GPS, A-GPS, GLONASS

For the SDK to run, the system must have the following specifications:

- Be equipped with either Windows XP (32-bit), Windows Vista (32/64-bit), Windows 7 (32/64-bit), Mac OS X 10.5.8 or later (x86 only) or Linux capable of running 32-bit applications
- Have JDK 6 installed
- Have Apache Ant 1.8 or later installed

The package with the software can be downloaded from the official website of Android for developers. It is about 400MB large.

After install, the developer must manage his tools so he can start development. This can be done from the Android SDK Manager. As a minimum to develop applications, the following must be installed:

- Android SDK Tools
- Android SDK Platform-tools
- Android SDK Build-tools
- SDK Platform (preferably for the latest API)
- A system image if there is no physical device

For more complex applications, there are lots of APIs that can be downloaded and used. For example, Android Support Repository is required for Android Wear or Android TV, while the Google Play services APIs provide features such as User Authentication or Google Maps. All these can also be downloaded from the SDK manager. The interface of the Android SDK Manager can be seen in figure 4.2.

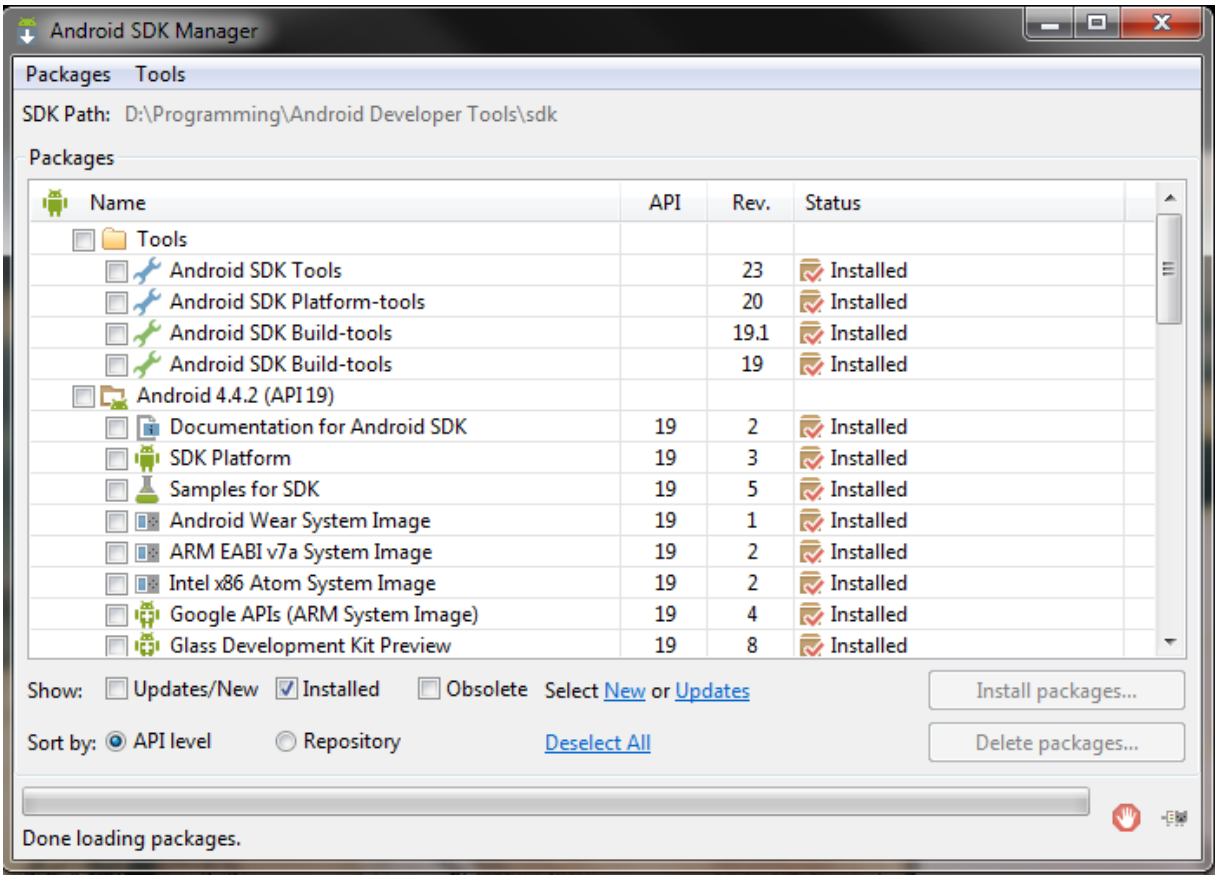

Figure 4.4 – Android SDK Manager

By default, on Android physical devices, the developer options are hidden by default for Android 4.2 and higher. To make them visible, the developer must go to *Settings-About phone* and tap the *Build number* seven times. Afterwards, the developer options will be visible in the settings menu.

Also, for the application to be installed and debugged via USB, the host computer running Windows must have drivers for the phone. For security reasons, on Android 4.2.2 and higher, at the first run of an application, the phone will display a message indicating whether to accept or not an RSA key for debugging. This ensures that any commands will be performed only after the device has been unlocked and the developer has acknowledged the dialog.

The developer options allow to:

- Launch debugging mode when USB is connected
- Keep the screen on while connected
- Allow mock locations for location testing
- Show touches and pointer location
- Show layout boundaries, CPU usage, etc

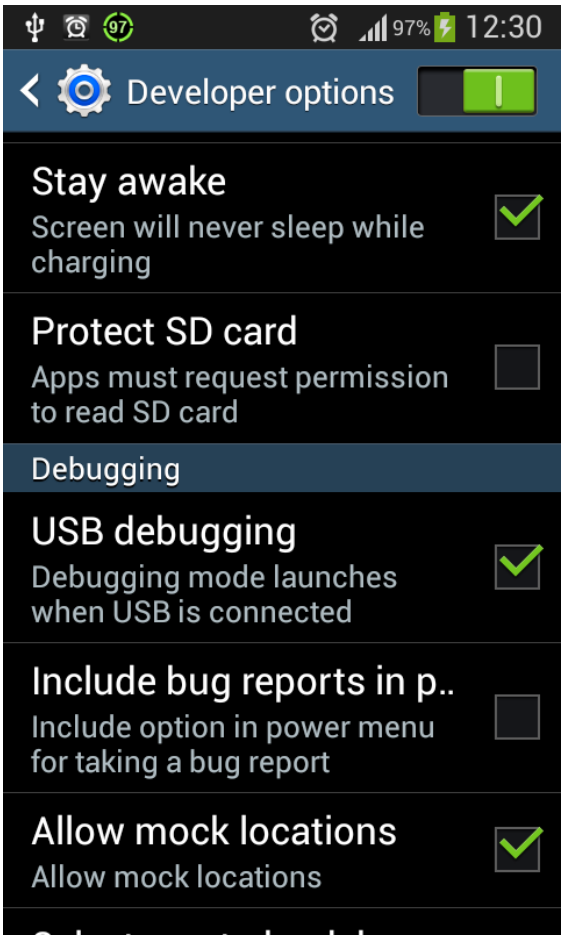

Figure 4.5 – Developer options on Android device

The project of an Android application will be built into an *.apk* file to be installed on the device. Each project has a standard structure, with specific subfolders and files. This can be seen in the figure below.

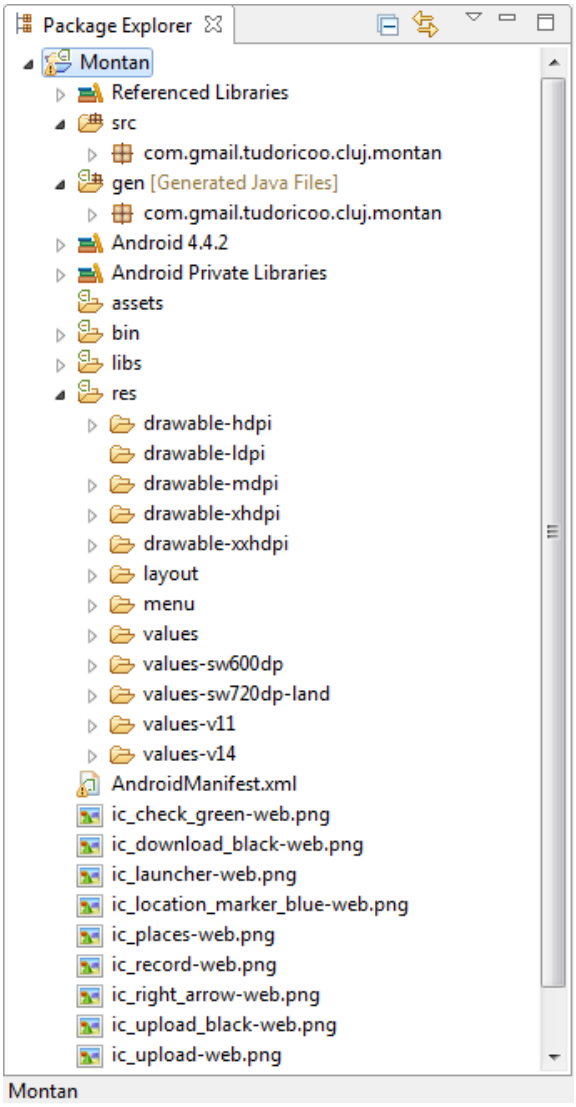

Figure 4.6 – Project structure

- */src* – contains all class files of Activities and any other custom classes created by the developer
- */bin* – output directory, where the *.apk* file will be located
- */gen* – contains files generated at build, such as *R.java*
- */res* – contains application resources. Subfolders include:
    - */anim* – XML files that are compiled into animation objects
    - */drawable* – bitmap and XML files that contain shapes or objects
    - */layout* – XML files that are compiled into layouts
    - */menu* – XML files that contain application menus
    - */values* – XML files that contain lists of colors, strings and other used in the application
- */libs* – contains other libraries

A very important file, found in the root of the file, is the AndroidManifest. This is a XML file that contains the definition of the application and its components. It contains permissions that are requested, a list of all the activities, services, intent receivers and content providers , what APIs are requested and supported and others. The only requested elements for an application to run are the tags *<manifest>* and *<application>*, but usually there are others that can contain other children. Same level elements do not need to be ordered

4.3.2 User interface

The Android operating system runs on a multitude of smartphones and tablets, with screen sizes raging from 3 inches to more than 10 inches. For an application to look similarly and offer the same user experience on all screen sizes, it is necesarry to design the user interface accordingly.

The terms that represent the basic elements when considering the design are:

- *orientation*, the physical rotation of the device so it is either viewed as portrait or as landscape. The same application can have different positions for UI elements on the two orientations.
- *screen size*, the physical dimension of the screen, usually measured in inches. Android groups screen sizes into small, normal, large and extra large screens.
- *screen density*, the number of pixels per physical length, usually dpi. Android groups screen densities into low, medium, high and extra high. These are translated into the application as **ldpi**, **mdpi**, **hdpi**, **xhdpi** and **xxhdpi**.
- *resolution*, the number of physical pixels on a screen. This is not important when designing is made for multiple screens.
- *density-independent pixel* or dp, a virtual unit that should be used for defining layout regardless of screen density. The following formula is used to convert dp units to pixels:

$$px = dp * (dpi / 160)$$

- *scale-independent pixel* or sp, the same as dp units, but relative to the user's font size preference. This is used for fonts.

When dp units are used properly, an application will look identical on two devices having the same screen size and different screen densities. When pixel units are used instead, the result will differ.

To achieve support for all desired screen resolutions, the manifest file of the project should contain all screen sizes supported by the application, using the *<supports-screen>* tag. Different layout XML must be created for each screen size. Each one must be placed in a folder having the name *layout-sw<N>dp*, where *<N>* is the smallest width required. Also, different screen densities should contain different drawables. Each drawable must be placed in a folder having the name *drawable-<size>*, where *<size>* can be replaced either with *ldpi*, *mdpi*, *hdpi*, *xhdpi* or *xxhdpi*. The scaling ratio for different sizes should be 3:4:6:8.

The central element of the user interface is the activity, which represents an instance of a screen. For an easier navigation, I will choose a main activity that allows access to the others through the menu. A splash screen also improves the user experience, so it will be the starting point of the application. Considering the system requirements, there will be an activity for each of the following:

- list available tracks
- record new track
- get tracks from the server
- login
- save tracks to server
- splash screen

The flow of the screens can be seen in figure 4.2. Each screen represents an activity.

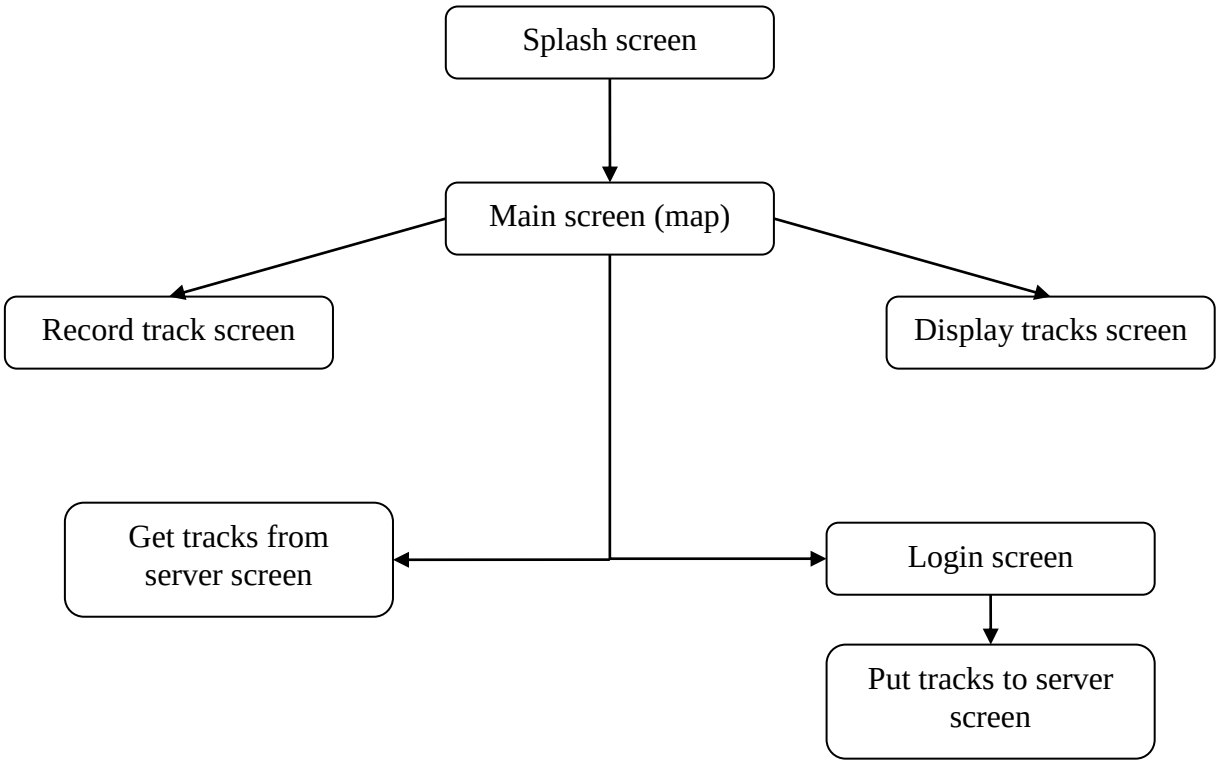

Figure 4.7 – Flow of screens

An activity's layout is usually defined through an XML file, but can also contain objects that are created programatically. The framework gives flexibility to use either one of the two possibilities or both. There can be a base layout that will get objects created only at runtime. For a better separation, it is advisable to declare UI in XML files.

A layout file must have only one root element of the type View or ViewGroup that can contain other objects as childs. Each element support a variety of XML attributes. The most important are:

- id, to uniquely identify the object in the tree. Using this id, the object can be later accesed and modified programatically using the function *findViewById(R.id.myLayoutObject),* where R is an automatically generated class file
- layout parameters, to define sizing. Each element must contain the attributes *layout_width* and *layout_height* to define their dimensions. Usually the dimensions are expressed to *wrap_content* or *fill_parent*, and more rarely, in a fixed dimension. Each element can contain paddings, which add dimensions inside the borders, and margins, which add dimensions outside the borders.
- layout parameters, to define positioning. An element can be centered in its parent using *layout_centerHorizontal*, *layout_centerVertical* or *layout_centerInParent* to center both horizontally and vertically. For some layouts, there are options to position the element in relation with other elements or its parent (*alignTop, alignParentTop,* etc)
- text to contain and font parameters. The text it is recommended to be a string resource added to the *strings.xml* file. Text sizes are set through the attribute *textSize* and are usually expressed in *sp*.

The most common layout types are:

- Linear Layout, a layout that contains only one vertical or horizontal row of elements. If the length of the children exceeds the length of the screen, a scrollbar is created automatically
- Relative Layout, a layout where each element can be positioned relative to other elements. For example, child X is at the right of child Y and child Z is below child X.
- List View, used for dynamic content that is in the form of a list. It displays a single column of elements. These elements are populated programatically, using an *Adapter*
- Grid View, similar to List View, but a layout which displays a scrolling grid of columns and rows.

The ADT plugin for Eclipse offers for layout files a graphical representation, together with options to add or view existing elements. This can be seen in figure 4.4.

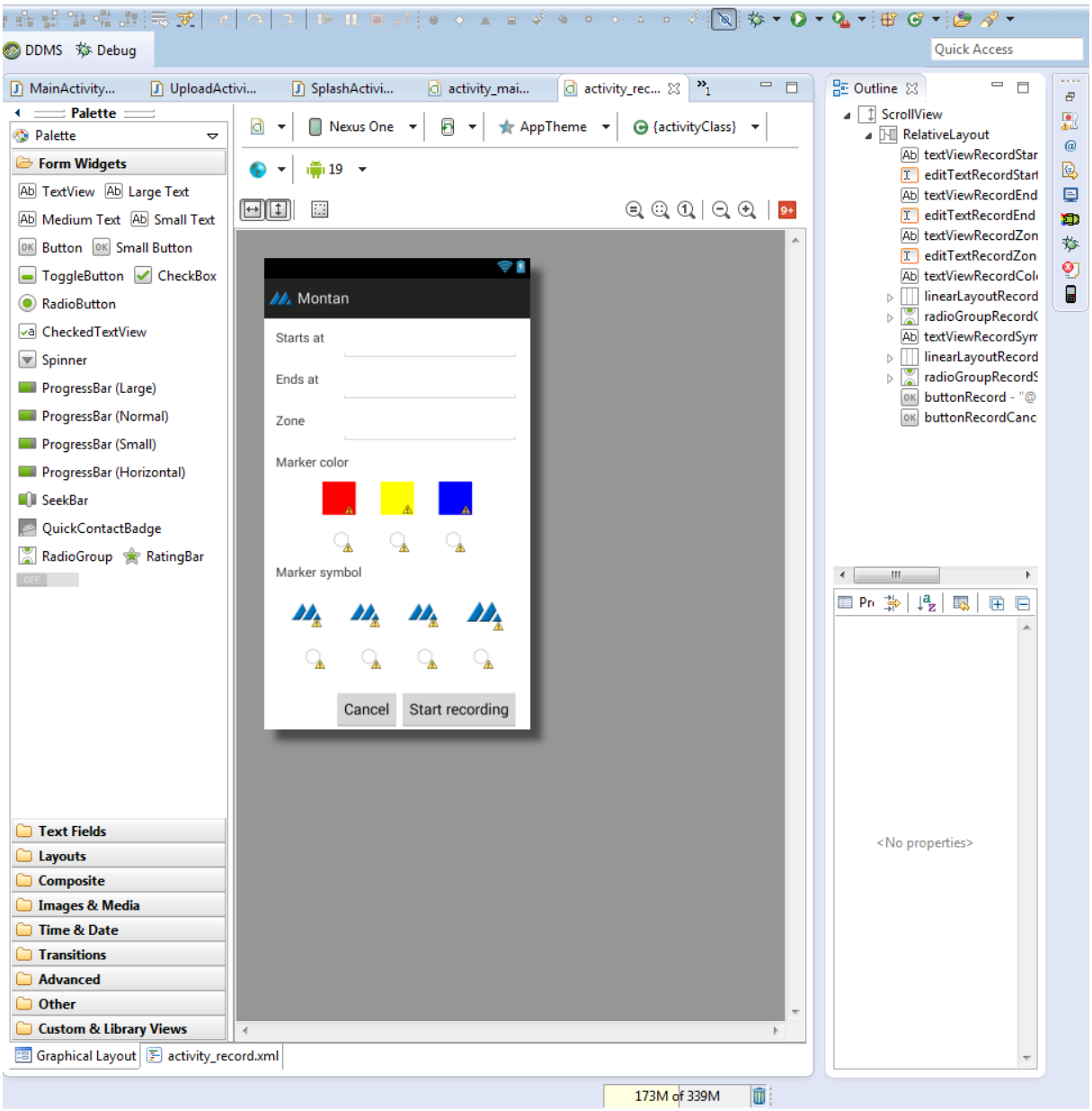

Figure 4.8 – Eclipse support for designing layouts

4.3.3 Activities

An activity is one of the most important components of an Android application. It displays a screen, generated from layout elements, that users can interact with. An application usually has multiple activities that can interact between them. Any activity can start another. When a new activity is started, the previous is stopped and put in a stack („backstack“). This stack is a LIFO mechanism.

An activity consists of a class that extends *Activity* or a subclass of it. It can implement methods to be called when the activity transitions between its lifecycle stages. The lifecycle of an activity together with the methods can be seen below. The states are represented by boxes and the transitions with the corresponding methods by arrows.

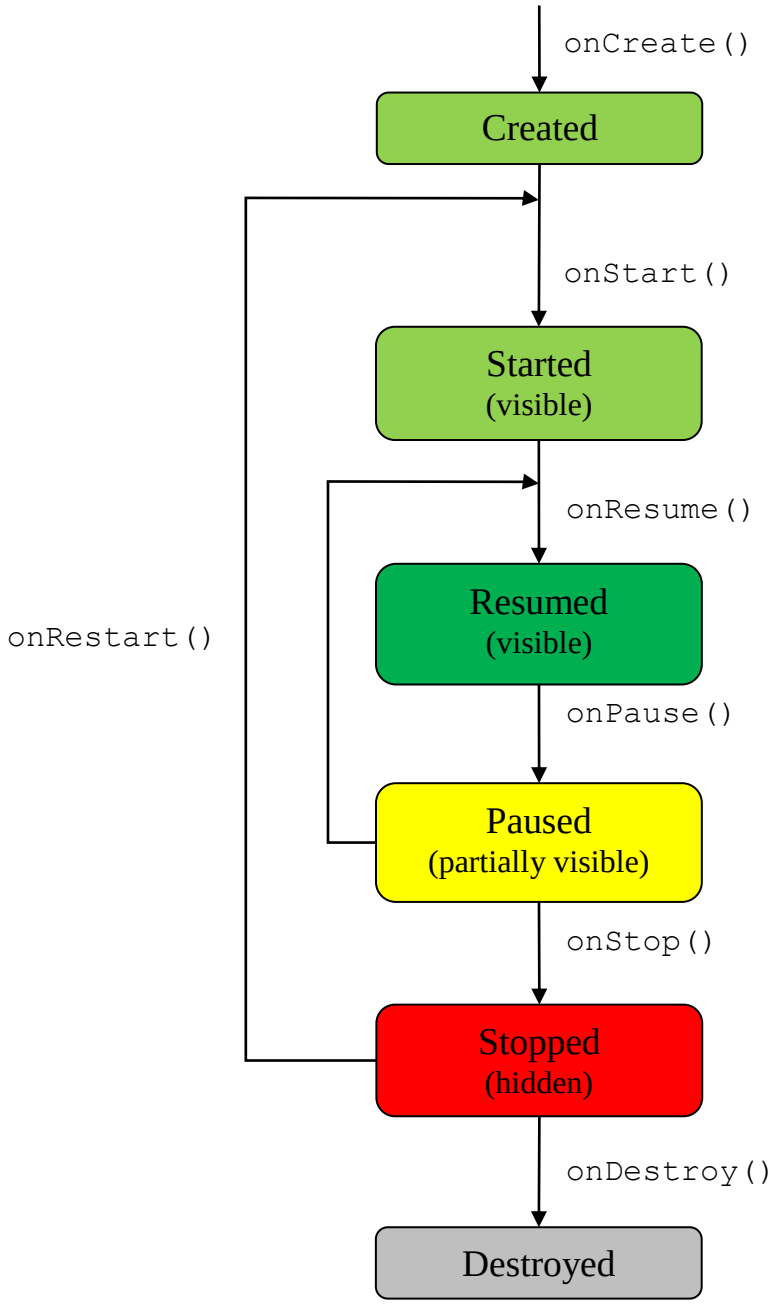

Figure 4.9 – Activity lifecycle

In each transition, a callback method is called. The methods are:

Table 4.2 – Activity lifecycle callback methods

`onCreate()`	Called when the activity is created. It should contain view creation and data binding.
`onStart()`	Called before the activity becomes visible on the screen
`onResume()`	Called before the activity allows interaction with the user
`onPause()`	Called when the activity loses focus of the screen. It is useful to commit unsaved changes to persistent data here and to stop intensive tasks, as it is probable that the system will resume another activity.
`onStop()`	Called when the activity is no longer visible on the screen
`onRestart()`	Called when activity is returning from the stopped state
`onDestroy()`	Called before the activity is destroyed. This can be caused either because the activity is finishing or because the system is killing it to save memory.

The activity is first created by another activity or by starting the application, if it is declared as implicit. The method *onCreate()* is called at this step. This is the only callback method that must be implemented explicitly. It should contain the initialization of essential components in the activity. The method *setContentView()* must be called here to define the layout to use for the user interface. The layout is a XML file defined in the *res/layout/* directory. A simple example of creating an activity woud be:

```
public class MainActivity extends Activity {
	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		this.setContentView(R.layout.activity_main);
	}
}
```

The method *onStart()* is called after *onCreate()* has completed. This is called before the UI becomes visible, but the user can only interact with it in the resumed state. If an alert or other similar object will take focus of the screen, the activity will enter the paused state by calling *onPause()*. Here, the activity is visibile in the background, but it does not have focus. In this point there are two possibilities: either the activity takes focus of the screen again, case in which *onResume()* is called, or it disappears off the screen and enters the stopped state by calling *onStop()*.

A stopped activity is alive and put in the backstack, but is completely obscured. Both in paused and in stopped states, there is a risk, if the system needs more memory elsewhere, that the activity will be killed. When the activity is recreated, it will pass all the states again.

When in stopped state, the activity usually resumes focus through back navigation. In this situation, *onRestart()* is called. The layout must be rendered again on the screen and allow interaction with user, thus the system will recall *onStart()* and *onResume()*.

Each activity must also declared in the manifest of the application, by adding an *<activity>* tag as a child of the *<application>* tag. An example can be seen below.

```
<manifest ... >
	<application ... >
		<activity android:name=".MainActivity" />
		...
	</application>
</manifest>
```

In the manifest, each activity can contain intent filters to declare how the system should handle them from outside of the application. The starting activity of the application should contain an *<action>* element and a *<category>* element in its intent-filter to specifiy it's the „main" entry point in the application and it is the application's launcher activity. If the application does not allow starting activities from outside of it, then no other intent-filters are required.

```
<activity android:name=".MainActivity" android:icon="@drawable/app_icon">
	<intent-filter>
		<action android:name="android.intent.action.MAIN" />
		<category android:name="android.intent.category.LAUNCHER" />
	</intent-filter>
</activity>
```

An activity can be started programatically by calling the method *startActivity()* and passing it an intent object. The intent can either specify the activity to start or describe a type of action to perform. The intent object can also store some data to pass between activities.

Starting an activity without passing it any data can be done like this:

```
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);
```

Starting an activity by passing it some data can be done like this:

```
Intent intent = new Intent(this, SecondActivity.class);
Intent.putExtra("key","value");
startActivity(intent);
```

Sometimes it is necesarry for the new started activity to send back some data to the activity that started it. For example, a login activity could send back to the main activity if the login has been successful or not. In order to do this, an activity must be started by calling *startActivityForResult()* rather than *startActivity()* and by setting an integer value to represent the request.

```
final Intent intent = new Intent(this, SecondActivity.class);
startActivityForResult(intent, 1);
```

The newly started activity should send back an intent object containing the results desired. The method *setResult()* assigns the intent to be sent and sets a flag to indicate the result of the operation.

```
Intent resultIntent = new Intent();
resultIntent.putExtra("key", "value");
setResult(Activity.RESULT_OK, resultIntent);
finish();
```

To read the result in the parent activity, the method *onActivityResult()* should be implemented.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
	super.onActivityResult(requestCode, resultCode, data);
	if(requestCode == 1 && resultCode == RESULT_OK && data != null) {
		...
	}
```

4.3.4 *Services*

Another important component in Android is the service. This can perform tasks in background, without providing a user interface. They remain alive even if the user switches to another application. One example of an application that uses services is a music player which can play music in the background while the user is browsing the internet. A service would be useful in our application for when the user wants to record a new track. He can initiate the command and stop it over several hours when he has reached his destination. Meanwhile, he can use other applications on the smartphone, or just keep it in his pocket.

There are two main types of services: *started* services and *bound* services. Bound services allow to send requests and get results in the component that binded it. For the track recording part of the application, a started service could be used. It should be started when the user initiates a record command, and stopped when the user has reached his destination.

Similarly to an activity, a service has some methods that handle its lifecycle. These methods can be seen below.

Table 4.3 – Service lifecycle callback methods

`onCreate()`	Called when the service is created. If the service is running, this method is not called
`onStartCommand()`	Called when another component requests starting the service by calling the method *startService()*
`onBind()`	Called when another component requests binding with the service by calling the method *bindService()*
`onDestroy()`	Called before the service is destroyed. This can be done either from the service, by calling *stopSelf()*, or from another component, by calling *stopService()*

The services must also be declared in the manifest file of the application, by adding a <*service*> tag as a child of the <*application*> tag. An example can be seen below.

```
<manifest ... >
      <application ... >
            <service android:name=".ExampleService" />
      </application>
</manifest>
```

A started service can be created by extending two classes:

- Service, which is the base class for services. By default, a service runs in the same thread as its hosting process and does not create his own thread. Services that do intensive work should be programmed to start in a separate thread.
- IntentService, a subclass of Service. This class creates his own worker thread and is useful to handle multiple requests simultaneously. Also, after it has handled all start requests, it stops itself automatically. There is no need to call *stopSelf()*.

A service can be started programatically by passint an Intent object to the *startService()* method. To avoid wasting system resources and battery power, it is important to stop services using *stopSelf()* or *stopService()* if called from another object.

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

The structure of a service implementation can be seen below.

```
public class MyService extends Service{

	@Override
	public void onCreate() {
		super.onCreate();
	}

	@Override
	public int onStartCommand(Intent intent, int flags, int startId) {
		return super.onStartCommand(intent, flags, startId);
	}

	@Override
	public IBinder onBind(Intent intent) {
		return null;
	}

	@Override
	public void onDestroy() {
		super.onDestroy();
	}
}
```

Services can notify the user of different events using Toast Notifications or Status Bar Notifications. More information about these can be found in a following subchapter.

4.3.5 Notifications

Notifications are used in the Android system to inform the user about different events or data. Toasts are short text messages that appear on the screen for small amounts of time. These are useful only if the user is looking at the screen in that moment. Instead, status bar notifications can be seen by expanding the notification drawer for a longer period of time, until the user or the application removes them.

To create a notification, the object *NotificationCompat.Builder* must be used. A Notification object must contain a small icon, set by calling the method *setSmallIcon()*, a title, set by calling the method *setContentTitle()* and a detail text, set by calling *setContentText()*.

For our recording function, to inform the user that the process is running, we could use a status bar notification. The user should not be allowed to delete the notification until he has reached his destination, otherwise the process could be left running for too much time. Notifications that cannot be deleted by the user by swiping them are called ongoing notifications.

An example of an ongoing notification can be seen below.

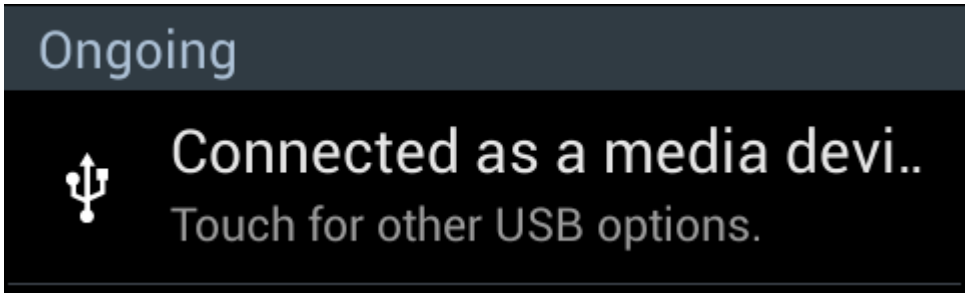

Figure 4.10 – Ongoing notification with icon, title and text

5 Implementation

5.1 Mobile app

When building Android applications, one of the rules that must be followed is that of naming. The package name declared in the manifest serves as a unique identifier for the application. Changing this when storing apps on Google Play results in two different applications. To avoid conflicts between apps, reverse domain ownership is used or, for independent developers, reverse e-mail address with location and application name. The package name I have used for my application is *com.gmail.tudoricoo.cluj.montan*.

5.1.1 User interface

For the user interface, I have used several resources and layouts. Each screen defined in the previous chapter has been designed in a layout file. The sizes of the application icon representing the logo, as well as other icons used, have been computed using the scaling ratio. The SDK also offers the option to import icon sets, either from the predefined list or from own images. The application logo with its all sizes can be seen in figure 5.1

Figure 5.1 – Application icon for ldpi (32x32 px), mdpi (48x48 px), hdpi (72x72 px), xhdpi (96x96 px) and xxhdpi (144x144 px) screens

The menu of the application consists of links to the other activities and a button to get the current location of the user on the map. The menu items are listed in the file *main.xml* from the folder */res/menu*. There are five menu items:

- Places, which takes the user to an activity listing all the tracks
- My place, which gets the location of the user, if possible, and shows it on the map
- Get tracks, where the user is taken to an activity where he can download zones data that are not the same version with data on the phone. This requires internet connection
- Put tracks, where the user is taken to an activity to login, if there is an internet connection. After he enters his credential, he is taken to an activity where he can see newly recorded tracks and select them to be sent to the server.
- Record track, which takes the user to an activity in which he can enter new track data. After he confirms it, he is taken back to the main activity where he can see progress of his recording.

A sample of the XML code implementing the layout of the menu can be seen below.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
       <item
          android:id="@+id/action_places"
          android:icon="@drawable/ic_places"
          android:showAsAction="ifRoom|withText"
          android:title="@string/action_places"/>
       <item
          android:id="@+id/action_my_location"
          android:icon="@drawable/ic_location_marker_white"
          android:showAsAction="ifRoom|withText"
          android:title="@string/action_my_location"/>
       <... />
</menu>
```

It is worth noting that each item has a *showAsAction* attribute set to *ifRoom/withText*. This implies that the menu items (i.e. the icon with the text) will be displayed in the action bar only if there is space, and if there is enough space, the text will also be displayed. This difference can be seen when rotating the screen.

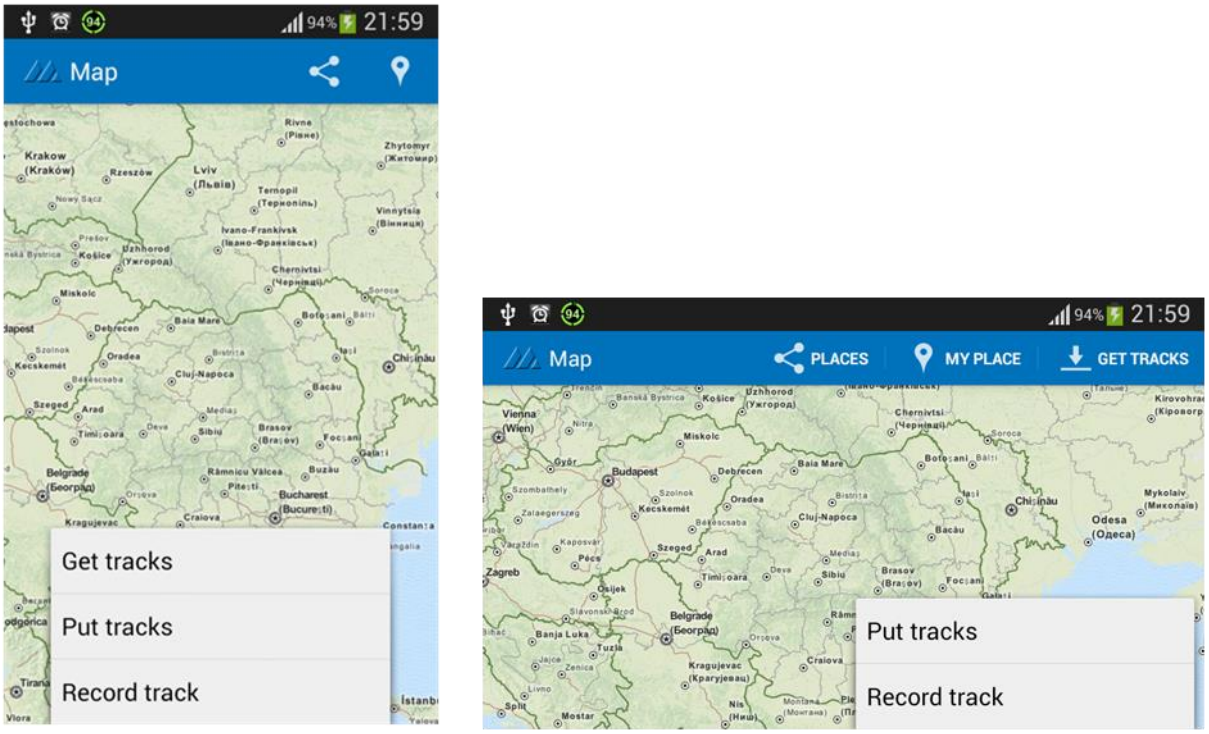

Figure 5.2 – Menu display differences between portrait and landscape orientations

The following activities are defined:

DownloadActivity.java
LoginActivity.java
MainActivity.java
PlacesActivity.java
RecordActivity.java
SplashActivity.java
UploadActivity.java

Each of their corresponding layouts can be seen in the next figures.

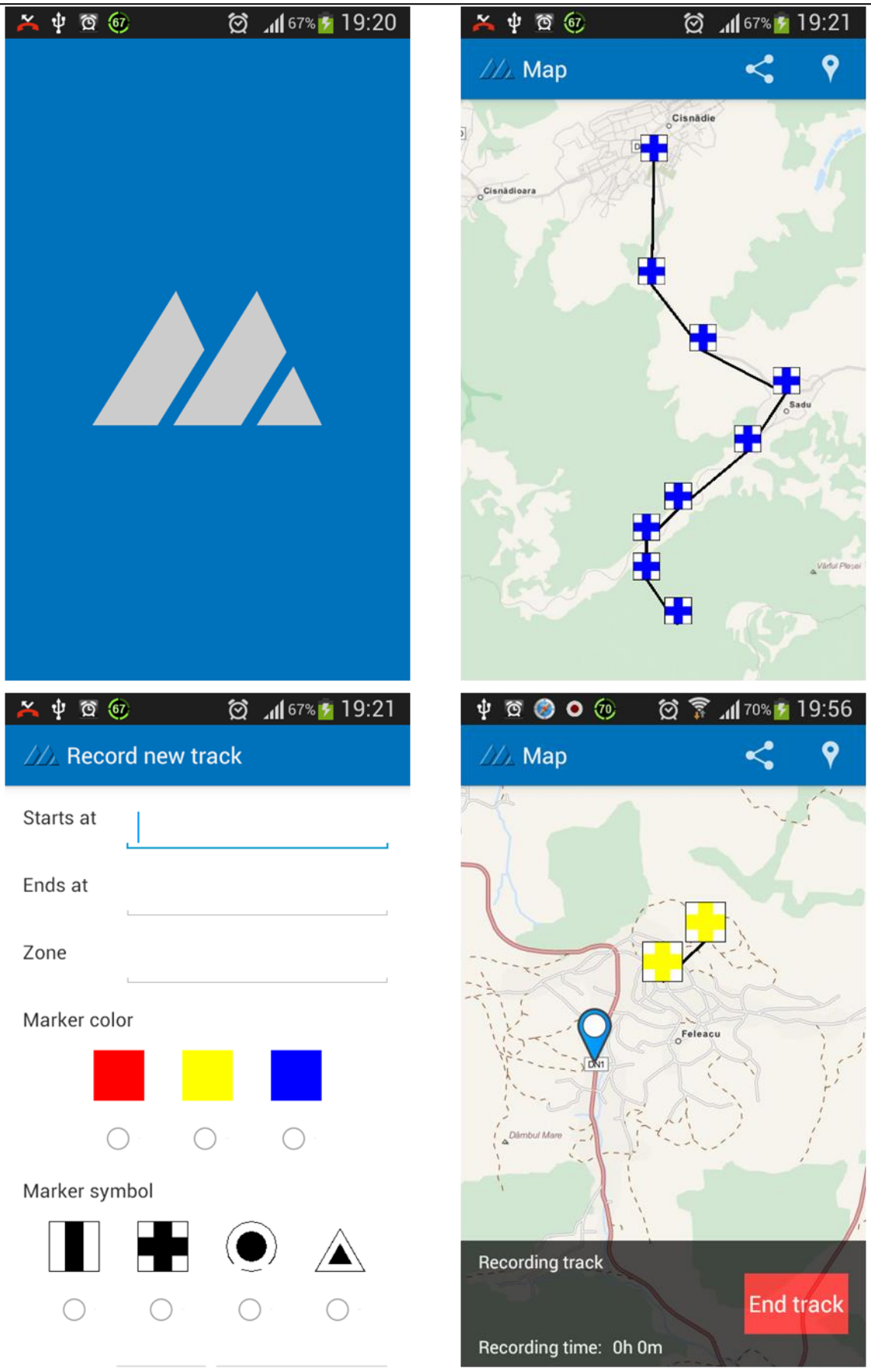

Figure 5.3 – Splash activity, Main activity, Record Activity And Main Activity when recording

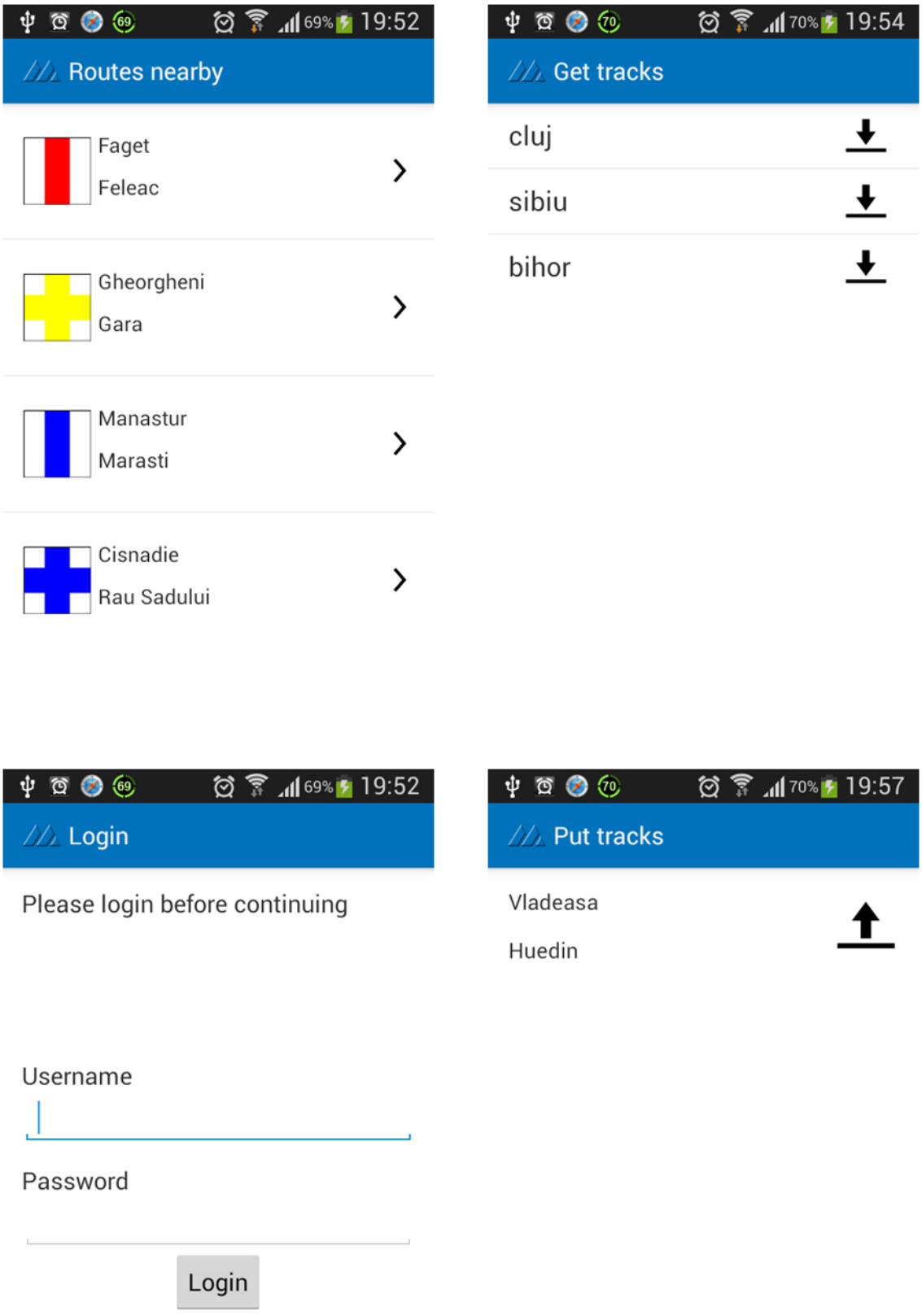

Figure 5.4 – Places activity, Download activity, Login activity and Upload activity

The activity with the most items in its layout is the *RecordActivity*. The elements are placed in a *RelativeLayout*, but because this is not scrollable by default, the *RelativeLayout* is wrapped inside a *ScrollView* item. The *RelativeLayout* contains several *TextView* and *EditText* objects for reading and writing text. Two objects of the type *LinearLayout*, oriented horizontally, are used to group the *ImageViews* displaying marker colors and symbols in a row. Two *Buttons* at the bottom of the screen allow the user to cancel the operation or to start recording.

The *MainActivity* consists of a *RelativeLayout* that wraps a *MapView* object from the osmdroid library. A second *RelativeLayout* is displayed only when the recording service is active. A *Button* here allows the user to end recording.

PlacesActivity, *DownloadActivity* and *UploadActivity* all consist of a *RelativeLayout* with a *TextView* and a *ListView*. The *ListView* is used to display data dynamically using a list adapter. Each of the three *ListViews* have defined a layout for children, in the files

- *activity_download_listview_item.xml*
- *activity_places_listview_item.xml*
- *activity_upload_listview_item.xml*

from the *res/layout/* directory in the project tree. The most complex is the *ListAdapter* of the *PlacesActivity*, due to the fact that it has to display a bitmap resource. A simple *ListAdapter* cannot achieve this, thus the base class must be inherited and extended.

The class *CustomPlacesAdapter* inherits the *BaseAdapter* class and implements the required methods *getCount(), getItem(), getItemId()* and *getView()*. The constructor receives the data about the tracks and a Context object, which points to the PlacesActivity object that created the adapter. A subclass called *ViewHolder* is used to define the layout of the list item, having the same components as the XML file: a *TextView* for track id, one for track start point, a *TextView* for track end point and an *ImageView* to display the track marker.

The most important method is the *getView()* method, which is called automatically when assigning the adapter to a *ListView*. It displays the data from the *routesList* ArrayList at the specified position in the data set. The implementation of the *CustomPlacesAdapter* can be seen on the following page.

The track markers are drawn on the screen using a static method from the class *PathMarker*. It receives as arguments the Context object where to draw, the symbol of the marker as a String, the color of the marker as a String and the size in dp as float. The size is transformed from dp to pixels for the current device running the application, due to the fact that Bitmap drawable objects can only be drawn in pixels.

The method interprets the color and the symbol and creates a Bitmap objects of the required size which will be later returned. The colors can be either *red, yellow, blue* or *black*, and the symbols *line, cross, triangle* and *circle*. A temporary Canvas object is created that is assigned the Bitmap object. On the canvas, there are drawn rectangles using *drawRect(...)*, paths using *drawPath(...)* and circles using *drawCircle(...)*.

The following two pages contain the implementation of *CustomPlacesAdapter* class and *PathMarker* containing the drawing function.

```
public class CustomPlacesAdapter extends BaseAdapter {
	private ArrayList<HashMap<String, Object>> routesList;
	private LayoutInflater inflater;
	private Context context;

	public CustomPlacesAdapter(ArrayList<HashMap<String, Object>> routesList,
				Context context) {
		this.routesList = routesList;
		inflater = LayoutInflater.from(context);
		this.context = context;
	}

	@Override
	public int getCount() {
		return routesList.size();
	}
	@Override
	public Object getItem(int position) {
		return position;
	}
	@Override
	public long getItemId(int position) {
		return 0;
	}

	@Override
	public View getView(int position, View convertView, ViewGroup parent) {
		ViewHolder holder;
		if(convertView == null) {
			holder = new ViewHolder();
			convertView = inflater.inflate(
				R.layout.activity_places_listview_item, null);
			holder.marker = (ImageView) convertView.findViewById(
				R.id.imageViewRouteMarker);
			holder.tvId = (TextView) convertView.findViewById(
				R.id.textViewRouteId);
			holder.tvStart = (TextView) convertView.findViewById(
				R.id.textViewRouteStart);
			holder.tvEnd = (TextView) convertView.findViewById(
				R.id.textViewRouteEnd);
			convertView.setTag(holder);
		}
		else {
			holder = (ViewHolder)convertView.getTag();
		}
		HashMap<String, Object> map = new HashMap<String, Object>();
		map = routesList.get(position);
		holder.tvId.setText(map.get("id").toString());
		holder.tvStart.setText(map.get("start").toString());
		holder.tvEnd.setText(map.get("end").toString());
		holder.marker.setImageBitmap(PathMarker.draw(context,
		    map.get("symbol").toString(), map.get("color").toString(), 100));
		return convertView;
	}

	public class ViewHolder {
		public TextView tvId, tvStart, tvEnd;
		public ImageView marker;
	}
}
```

```
public class PathMarker {

	public static Bitmap draw(Context context, String shape, String color,
				float sizeDP) {
		Resources r = context.getResources();
		float sizePx = TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
					sizeDP, r.getDisplayMetrics());
		float size = sizePx;

		float borderWidth;
		if(size < 100) borderWidth = 1;
		else borderWidth = size/100;

		Paint whitePaint = new Paint();
		Paint colorPaint = new Paint();
		Paint blackPaint = new Paint();
		if(color.equals("red")) colorPaint.setColor(Color.RED);
		if(color.equals("yellow")) colorPaint.setColor(Color.YELLOW);
		if(color.equals("blue")) colorPaint.setColor(Color.BLUE);
		if(color.equals("black")) colorPaint.setColor(Color.BLACK);
		whitePaint.setColor(Color.WHITE);
		blackPaint.setColor(Color.BLACK);

		Bitmap tempBitmap = Bitmap.createBitmap((int)size, (int)size,
					Bitmap.Config.ARGB_8888);
		Canvas tempCanvas = new Canvas(tempBitmap);
		tempCanvas.drawColor(0, Mode.CLEAR);
		tempCanvas.drawBitmap(tempBitmap, 0, 0, null);

		if(shape.equals("line"))
		{
			tempCanvas.drawRect(0, 0, size, size, blackPaint);
			...
		}

		if(shape.equals("cross"))
		{
			tempCanvas.drawRect(0, 0, size, size, blackPaint);
			...
		}

		if(shape.equals("triangle"))
		{
			Path blackPath = new Path();
			Path whitePath = new Path();
			Path colorPath = new Path();
			...
		}

		if(shape.equals("circle"))
		{
			tempCanvas.drawCircle(size/2, size/2, size/2, blackPaint);
			...
		}

		return tempBitmap;
	}
}
```

5.1.2 Database

5.2 Server side

6 Testing

7 Conclusions

Improvements to be made:

Routings

Time estimations

Create limits for the zones and draw them (for example, hash with green Parcul National Retezat)

Accuracy of tracks (more points)

Objectives on map

Transactions on SQL

Server track management (admins)

Client track management (edit names/delete recorded tracks)

Keep user logged in

Map maximum zoom out

8 Bibliography

1. Figure 1.1 - http://www.manhattandigest.com/2013/12/09/singularity-dont-know-start-paying-attention/
2. Figure 3.1 - http://developer.android.com/about/dashboards/index.html
3. http://developer.android.com/guide/practices/screens_support.html
4. http://www.techrepublic.com/article/apple-v-google-the-goliath-deathmatch-by-the-numbers-in-2014/
5.

9 Dictionary

3:4:6:8 scaling ratio = a ratio used to calculate sizes for bitmap drawables regarding screen density.

Drawable = A graphic item that can be drawn to the screen. This includes bitmap drawables - *.png*, *.jpg* or *.gif* files.

Manifest file = The file named „AndroidManifest.xml", found in the root of the Android project. It contains all the necesarry permissions and settings for the application.

RSA = a public-key cryptosystem, used for secure data transmission

10 Acronyms

ADB – Android Debug Bridge
ADT – Android Development Tools
API – Application Programming Interface
dp – Density-independent pixel unit
DBMS – database management system
DPI – Dots per inch
ERD – Entity relationship diagram
GB – Gigabyte
GHz – Gigahertz
GSM – Global System for Mobile Communications
HTTP – Hyper Text Transfer Protocol
IDE – Integrated development environment
JDK – Java Development Kit
LIFO – Last in, first out
mAh – Milliampere hour
MB – Megabyte
MHz – Megahertz
OS – Operating system
PDO – PHP Data Object
PHP – Originally Personal Home Page, nowadays PHP: Hypertext Preprocessor
RAM – Random Access Memory
SDK – Software Development Kit
px – Pixel unit
UI – User interface
USB – Universal Serial Bus
XAMPP – Cross-platform, Apache, MySQL, PHP, Perl
XML – Extensible Markup Language

11 Annex