



TAMPERE UNIVERSITY OF TECHNOLOGY

DIEGO PINEDO ESCRIBANO
ANALYSIS OF THE DEVELOPMENT OF CROSS-PLATFORM
MOBILE APPLICATIONS

Master of Science Thesis

Examiner: Professor Tommi Mikkonen
Examiner and topic approved by the
Faculty Council of Computing and
Electrical Engineering on 9th May
2012

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Information Technology

PINEDO ESCRIBANO, DIEGO: Analysis of the development of cross-platform mobile phone applications

Master of Science Thesis, 46 pages, 33 Appendix pages

June 2012

Major subject: Software Engineering

Examiner: Professor Tommi Mikkonen

Keywords: mobile application, cross-platform, Android, iOS, Appcelerator Titanium, native application, Web based application

The development of mobile phone applications is a huge market nowadays. There are many companies investing lot of money to develop successful and profitable applications. The problem emerges when trying to develop an application to be used by every user independently of the platform they are using (Android, iOS, BlackBerry OS, Windows Phone, etc.). For this reason, on the last years many different technologies have appeared that making the development of cross-platform applications easier.

In this thesis one of these technologies, Appcelerator Titanium, will be analysed. During the analysis process we will explain what a mobile phone application is. Also will be explained what a cross-platform mobile application is and what their features are. Finally, Appcelerator Titanium is introduced, presenting the features this technology provides.

In order to show the results on a more visual way, a cross-platform mobile phone application that will be installed natively on Android and iOS platforms will be developed. It will contain the most important features that a mobile phone application should have.

The general conclusions extracted from this thesis and the application is that cross-platform mobile applications have some advantages compared with the regular mobile applications, but the development process depends on frameworks that are not mature enough.

To my parents, because they have always been when I needed them.

To my brother, who always supported me.

*To Martin, Carlos, Javi, Aitor and all my colleagues, which made me
become the person I am.*

*To my tutor Tommi Mikkonen, who gave me the opportunity to write
this thesis and always kept a positive and understanding attitude.*

To Javier Blas, who allowed me to write my thesis abroad.

*To the department of architecture at the TUT for giving me access to
their resources.*

*To Heidi Fordell and the whole team of the International Office, be-
cause they always helped me and they allowed me to extend my ex-
change period.*

To Hernan, who helped me with the corrections.

“If you think you cannot do it, don’t worry, you will not”

Carlos Antonio Rodriguez Mecha. Vancouver, 2010

TABLE OF CONTENTS

1.	Introduction	1
2.	Mobile Applications	2
2.1.	Types of mobile phone applications.....	2
2.2.	Components.....	3
2.2.1.	Graphical user interface	3
2.2.2.	Application logic and interfaces	5
2.3.	Cross-platform application.....	7
2.3.1.	Cross-platform development frameworks.....	7
3.	Appcelerator Titanium	9
3.1.	Introduction	10
3.1.1.	Implementing the application.....	11
3.1.2.	Compilation of the code.....	13
3.1.3.	Pack the application	14
3.2.	Elements	14
3.2.1.	Navigation.....	15
3.2.2.	Controls.....	18
3.2.3.	GPS and Maps.....	23
3.2.4.	Camera	27
3.2.5.	Audio.....	28
3.2.6.	Storage	29
4.	Analysis of a cross-platform mobile application	31
4.1.	Use Case: Travel Guides application	31
4.2.	Analysis of the requirements.....	32
4.3.	Application design.....	34
4.3.1.	UI module	35
4.3.2.	lib module	36
4.3.3.	app module.....	37
4.3.4.	Screenshots.....	37
4.4.	Evaluation of the framework.....	42
5.	Conclusions	45
	References	47
A.	Use case implementation.....	50

1. INTRODUCTION

The development of mobile phone applications is a huge market nowadays. There are many companies investing lot of money to develop successful and profitable applications. The problem rise when trying to develop an application to be used by every user independently of the platform they are using (Android, iOS, BlackBerry OS, Windows Phone, etc.). The traditional way to deal with this problem has always been to hire more experts and split the development process into different flows, each one taking care of one target. However these solutions may seem inefficient for a small company that cannot afford to hire an expert.

For this reason, in the last years many different solutions have appeared that make the development of cross-platform applications easier. These solutions are usually based on a framework with specific tools that allows the user to program under one middleware technology and as a result one can obtain an application that runs on different platforms (the ones the specific solution allows to develop for).

The goal of this thesis is to analyse cross-platform mobile applications and specifically, the ones produced using the framework Appcelerator Titanium. In order to achieve this goal Appcelerator Titanium will be analysed and other frameworks will be reviewed. During the analysis process we will explain what a mobile phone application is, establishing the difference between the two existing types of applications, native applications and Web applications. Also will be explained what a cross-platform mobile application is and what their features are. Finally, Appcelerator Titanium is introduced, presenting the features this technology provides, analysing the homogeneity of the applications produced by the framework, its problems and possible solutions. In order to show the results in more visual way, a cross-platform mobile phone application that will be installed natively on Android and iOS platforms will be developed. It will contain all the most important features that a mobile phone application should have.

This thesis is divided into four chapters. In Chapter 1 there is an introduction to the thesis. In Chapter 2 we will explain what a mobile application is, types of applications and after that we will focus on what is a cross-platform mobile application. In the Chapter 3 we will analyse the Appcelerator Titanium framework. The Chapter 4 will explain how to develop a cross-platform application using Appcelerator. An entire mobile application will be developed. Finally, Chapter 5 we will expose the conclusions of this thesis.

2. MOBILE APPLICATIONS

A mobile handheld application is software developed to be used on any kind of mobile devices such as smartphones or tablets. These kinds of applications could be stand-alone applications or they could be distributed applications spread around different devices, networks and/or servers. Nowadays almost all the applications are developed for smartphones and tablets that offer very good and powerful tools and frameworks to develop this software.

2.1. Types of mobile phone applications

There are two types of mobile phone applications: native applications and applications based on web technologies. A native application is the one that was designed to be installed on a specific operating system (iOS, Android, Symbian, etc.) and it is also depending of the device's firmware. In order to have an application running in different devices or models small or big changes will be needed. Web based applications are developed using Web technologies such as HTML or JavaScript, and should be interpreted by the Web browsers of the different operating systems.

There are several differences between native and web applications. Each of them has its own advantages and disadvantages. Based on the results obtained by the Global Intelligence Alliance paper [1] and the analysis made by Worklight [2] it is possible to extract the key points that make the difference between native applications and web applications. The first point is the user experience; this is one of the most important factors to consider when deciding what kind of application is going to be developed. The results extracted from the study shows that native applications have the ability to build a superior user interface, thus the experience of the user is better on the native applications than with web based applications. This improvement is reflected by the fact that "twice as many publishers saw higher user adoption and usage volume over native applications." In particular "30% respondents with both interfaces see over double usage volume over the native application". [1 p. 16]

The other important difference remarked on the research is that the access to the main device features is still better in native applications, but the evolution of the technologies, particularly HTML5, is balancing the scales with the Web applications. It is a matter of time that web technologies have access to the device camera, GPS, contacts, calendar, file system, accelerometer, and so forth.

The next feature remarked by Global Intelligence Alliance is that with Web applications the developers have full control over the application and they have direct control over own distribution, without need to seek 3rd-party vendor approval. In some cases this is an advantage because it is possible to obtain full profit from sales or advertisements within the application. On the other hand, 3rd-party vendors like the Apple App Store or the Google Android Market provide a powerful marketing tool to promote and distribute the applications.

And finally, the last feature that makes a difference is the ability to distribute the application on as many devices as possible. This time, the Web applications has an advantage over the native applications, because, as it is mentioned on the conclusions of the paper, “Web apps offer an architectural advantage when targeting a cross-device launch, where significantly less platform migration is required as compared to native applications” [1 p. 6]. It means that in the case of willing to develop for many devices it is a better option to use a Web application.

To summarise the information extracted from both studies, on the **Figure 2.1** there is a direct comparison between native and Web based applications.



Figure 2.1 Comparison between Native and Web apps from Worklight [2]

2.2. Components

A mobile application can be divided into graphical user interface (GUI), and application logic and interfaces to device’s peripherals.

2.2.1. Graphical user interface

The graphical user interface is the access point where the user will control the behaviour of the application. It includes all the graphical elements and the information that will be shown to the user. This is one of the most important elements of a mobile application

because it is the one that defines the user experience. How the user interacts with the application will be the difference between a successful application and the others.

Both Android and iOS have separated modules to create and customize the user interfaces. Android uses a hierarchical structure of elements (**Figure 2.2**) defined in a XML file [3]. In this file there are elements called *views* that are inserted within other elements called *viewGroups*. Different elements like buttons or text boxes could be added inside the views. Each element of the hierarchy will become a Java class where certain behaviour will be implemented. Moreover, *viewGroups* can be nested.

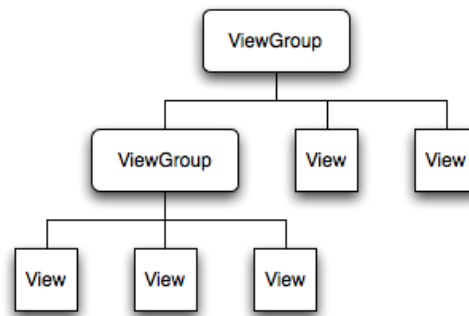


Figure 2.2 Hierarchical structure of an Android UI

On the other hand, iOS uses a service called Cocoa Touch [4], which is a framework that abstracts some functionalities of the operating system to provide the tools needed to develop the user interface.

Android and iOS provide many different elements to build the user interfaces. Many of those elements are the same, implemented with different technologies, but some of them are exclusive of the platforms. It is important to remark this fact, because when one is developing a cross-platform application it is necessary to know what could be implemented in every platform and what cannot. It is possible to find more information about the whole set of elements each platform provides on the developer pages of Android [5] and iOS [6]. The elements provide by both platforms can be categorised in two groups: Containers and content. In **Table 1** this division is displayed.

Table 1 Shared elements between Android and iOS

Containers		
Windows: <i>A window is an empty drawing surface or container. Windows can be opened and closed.</i>		
Tabs: <i>The tabs, usually grouped over a tab group, are one way to organise different windows with the same hierarchy.</i>		
Layouts: <i>These elements are used to specify how the content will be displayed on the container.</i>		
Content		
Buttons	Date and time pickers	List pickers
Switchers	Progress bars	Text labels and inputs
Alert dialogs	Table views	Search bars
Sliders	Images	Scrollers

2.2.2. Application logic and interfaces

Application logic is the part of the program that defines the behaviour of the application. It will determine what actions will be executed in response to different kind of stimulus like the user interaction, phone calls, internal schedulers, etc. This part will interact with the software installed in the operating system to be able to access different features like the camera, Internet, the agenda and every other feature that the operating system of the device allows to access.

For the scope of this thesis, just the following features that Android and iOS provide will be studied: Camera, Audio, GPS, Maps and Storage.

Camera. Both Android and iPhone provide interfaces for accessing the camera of the device. With this interface it is possible to take pictures and shoot videos, zoom in or out or change the configuration of the camera. Both platforms also provide the tools to build a specific camera application and access to the most basic and low level features, accessing to the raw information captured by the camera.

Audio. Every mobile phone has a microphone and a speaker and almost every mobile operating system provide access to them. With those tools Android and iPhone offers a way to reproduce (play, stop, and move) and record audio in several formats. More about the recording and reproducing process in those platforms is provided in [7], [8] and [9].

GPS and localization. Nowadays every mobile device is provided with a GPS. The operating systems are responsible for providing an access interface to this service to offer the possibility to build localization based services [10]. The GPS was originally thought to use just satellites signals. But the huge amount of devices equipped with this system may affect its speed and accuracy for mobile applications. For this reason many operating system like Android [11] or iPhone [12] complement this system with localization based on Wi-Fi and mobile networks (Figure 2.3). Those methods could be transparent for the application developer and for the final user but both of them have the chance to select which method to use. The most common data extracted from a GPS and that can be used by the application developers are longitude, latitude, altitude, speed and time. Those data have a precision depending on the quality of the GPS signal and the rest of techniques mentioned before.

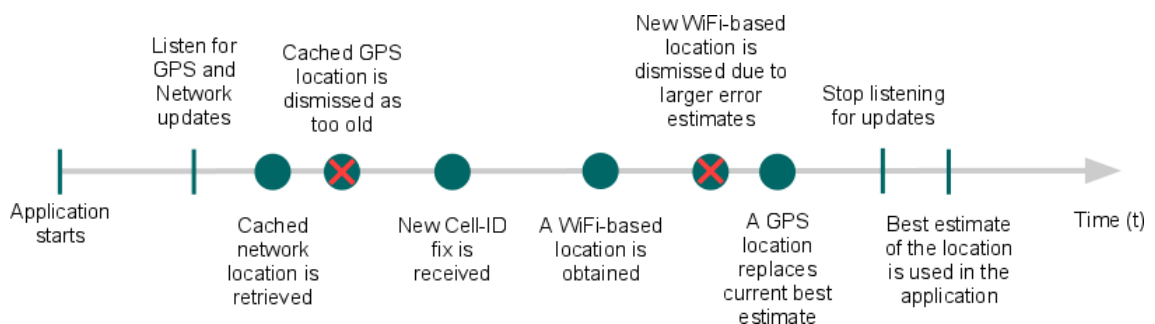


Figure 2.3 A timeline representing an Android application waiting for location updates.

Maps. There are several map systems that can be used by a mobile OS. The most famous are Google Maps [13] used by Android and iOS, Bing Maps [14] used by Windows Phone, Nokia Maps used by Symbian or OpenStreetMaps. All those systems are based on the existence of a map service stored on the company servers who provides an API to interact with it. The most common features those services provides are to visualize a map of an specific area, move the map, zoom in and zoom out, create pins and add them to the map to mark places, create routes between places or look for information (addresses, places, etc.) in the map.

Storage of information. The persistence of the application is needed in order to keep its own state and to store the information collected or generated during the life time of the application. There are several ways to store this information but can be split in two groups: the techniques that use the local resource to store the information and the others that use Internet to do it. The local storage could be used in many ways.

One way to do it is to store the information on the device's file system. It could be done over the memory of device or over external storage devices such as SD cards. Every operating system offers an API to access to the file system to create many types of files.

This API is very similar to the one that can be found in every PC operating system, so there are services to create, delete, move, and edit files and directories.

One specific type of file frequently used to store information is XML file format [15]. Mobile operating systems usually define a specific module within the API to manage this kind of files.

Another system to store information on the device is using data bases. Both Android and iOS offer the possibility to use the data base manager SQLite [16]. This is transactional SQL database engine that can work without any previous installation and configuration and this made this tool suitable to work in many kind of devices.

2.3. Cross-platform application

A cross-platform application is a software developed to run on different platforms without behaviour or visual changes. There are several levels of cross platform applications. In one side of the scale, there are the full cross-platform applications like a Java application where no changes are needed in the code to be able to run it over any platform like MacOS, Windows or Linux. In the middle, there are applications that need small modifications inside the code to be running in different platforms but almost all the codification is the same. One example could be a web application written in HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and Javascript. While these are standards by the World Wide Web Consortium [17], every different Web browser implements them in a different way. Thus a modification of the codification is needed in order to have a cross-platform (in this case, a cross-browser) application. On the opposite side of the scale, there are the applications that are designed once and programmed several times, one implementation per platform. With this technique an application is created, which is not truly cross-platform, but for the end user it looks like.

When we are dealing with cross-platform mobile applications, the most common type are the second and the third type. The second type, the one-time implementation plus modifications per platform, is mostly used by the web based applications. In the other hand, the third type is totally linked with the native applications.

2.3.1. Cross-platform development frameworks

As it was mentioned in the previous section, there are two kinds of mobile phone applications, natives and web applications. Because of this separation, the different frameworks already existing on the market will be divided. There are frameworks that produce native applications and there are frameworks that produce web based applications. In the first group we have frameworks like *Titanium Appcelerator* or *PhoneGap*. These frameworks are open source and all of them are using web technologies like *JavaScript* or *HTML* as the main programming language.

Titanium Appcelerator [18] is an open source framework that uses *JavaScript* syntax and his own libraries to create cross-platform native applications. With this platform *Android* and *iOS* applications can be developed so far. To develop for *Android*, *Windows XP*, *Windows 7*, *MacOS* or *Linux* is required. To develop for *iPhone* and *iPad* a *MacOS* computer is required due to the impossibility to obtain the *XCode* compiler for any other platform.

PhoneGap [19] is an open source platform that uses *JavaScript*, *HTML* and *CSS* to develop cross-platform native applications for *iOS*, *Android*, *Blackberry*, *Windows Phone*, *WebOS* and *Symbian*. Using the same logic than *Appcelerator*, to develop for *iPhone*, *PhoneGap* running over a *MacOS* computer is needed. To develop for *Android*, *WebOS* or *Symbian* every platform can be used, namely, *Windows*, *MacOS* and *Linux*. To develop for *Windows Phone 7*, *Windows 7* or *Windows Vista* is are required. Finally, to develop for *Blackberry*, *Linux* cannot be used.

Both platforms claim to allow the access to the main important features of a native application: the GPS, the camera, the audio, the gallery, the contacts, and so on. On the other hand, we have frameworks that produce applications thought to be executed by Web browsers, the Web applications. There are several frameworks to develop this kind of applications but the most important and used nowadays are *Sencha Touch* and *jQuery mobile*.

Sencha Touch [20] is a framework used to create HTML5 Web applications for iPhone, Android, and BlackBerry. It uses an extended version of JavaScript library called ExtJS. In order to access the different services the device provides (Camera, GPS, contacts, etc.), the application must be wrapped in a native shell.

jQuery mobile [21] is a framework that uses HyperText Markup Language 5 (HTML5), Cascading Style Sheets (CSS), and JavaScript to create Web application for a big amount of operating systems (iOS, Android, Windows Phone, Blackberry, Palm WebOS, Meego, Kindle and much more). jQuery is a tool to create web versions for mobile devices rather than mobile phone applications with complex functionality.

Appcelerator Titanium will be used because of the following rationale. The first reason is because it produces a full native application. It was already explained on a previous section why a native application is preferred. The other reason is that Titanium Studio, the framework used by Titanium Appcelerator, is based on the Eclipse framework which is commonly familiar among developers. And the last but not the least, Appcelerator has a big community of developers to provide support, which makes solving problems an easy task.

3. APPCELERATOR TITANIUM

Appcelerator Titanium [18] is an open source platform that provides the tools to develop cross-platform mobile device applications using JavaScript [22] as the core technology. Titanium allows users to program native and web based applications for iOS and Android platforms.

To interact with Titanium, the framework Appcelerator Studio [23] is required. This framework is based on the Eclipse IDE [24] so it allows the users to do the same things they can do with the original Eclipse project, namely coding, testing, deployment, and versioning. It also includes the possibility to install the application directly on the device to test it and debug it. This is particularly interesting because there are features like the camera, the audio recorder or the accelerometer that are not accessible with the emulators that XCode or Android SDK provides.

Titanium acts like a bridge between the different implementations of a mobile phone application components made by the different operating systems (Android and iOS) and the JavaScript programming code. The corresponding diagram is shown on the right side of **Figure 3.1**.

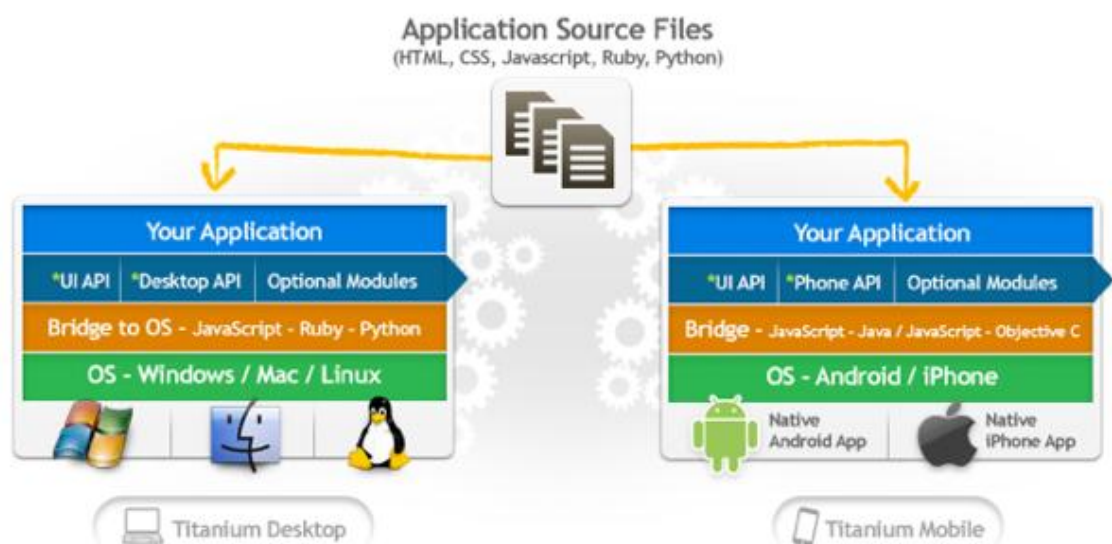


Figure 3.1 Titanium Mobile architecture

All the Appcelerator features that will be explained have been tested with an iMac with MacOS 10.X, Xcode 4.2.1, Android SDK v3.1, and Appcelerator Titanium 1.8.1 framework. The application has been tested on MOTOROLA Xoom tablet with an Android operating system. For iOS platform the application has been tested on the iPhone Emulator that XCode provides.

3.1. Introduction

The development of a mobile application using Titanium has several steps. The first thing that has to be done is to create a *Titanium Mobile Project* (**Figure 3.2**) and specify the name and the company name of the application (always starting with “com.”). In addition, the target platforms will also be listed.

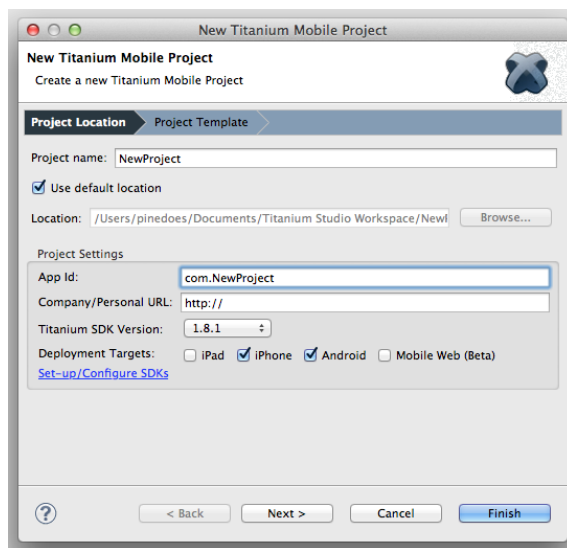


Figure 3.2 Creation of a new project

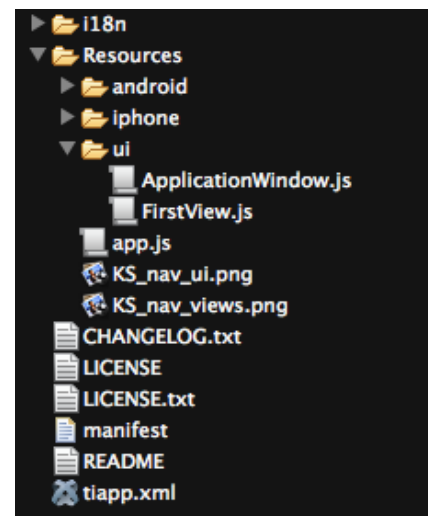


Figure 3.3 Structure of the project

Once the project is created, Titanium creates a structure of folders and files that helps to organize the project. **Figure 3.3** shows this structure. On the root folder we can find an XML file called *TiApp.xml*. This file contains the configuration of the application and information that it will be used to generate the native resource. Because each platform we are going to use (Android and iOS) need different parameters to configure their applications, we will include platform related information in this file. More information about this file can be found on the Wiki of the project [25]. In the root folder we can also found the *i18n* directory that will be used to store the string files needed for the internationalization of the app. We can also find some other files which content is not relevant for the development on an application.

The actual code of the application is localised into the *Resources* folder. Here we can find the main file that will be used as starting point of the app, the *app.js*. Under this folder one can also find particular folders like the *android* or the *iphone* folder. Inside

this one can find information related to each platform like images configured with the specific resolutions each platform demands or every resource that will be used by the specific operating system.

The last elements to analyse are the JavaScript files that define the logic of the application. The code of the application will be written on those files with extension *.jess* and they will use the API that Titanium SDK provides [26].

Once the code of the application is ready, the framework provides two options. The first one is to run the application within an emulator of any of the platforms or run it into a connected device. The second option is to pack the application for distributing it over the different platform channels. In both cases the source is compiled into binary code and the Android APK [27] or iPhone IPA [28] is generated.

3.1.1. Implementing the application

As was mentioned above, the coding is done using JavaScript syntax and an extensive API is provided in order to cover as much functionality as possible. In the documentation of the Titanium project [29] one can find all the list of features available to create a mobile application. The most important and the most relevant items for this thesis are the following:

- Support for all native platform UI controls.
- Support for geo-location.
- Support for camera: taking photos, playing and recording video.
- Support for streaming and recording audio using the microphone.
- Support for native maps.
- Support for file system (reading and writing files and folders).

JavaScript is a powerful tool that allows the programmer to use this language in many different ways, from a more object oriented paradigm to a procedural code. Despite this fact, Titanium Appcelerator suggests that, in order to design strong, reusable and fast applications, some techniques have to be used and obtain a high **modularity** degree. There are some tools that can be used to have a modular application. The most important are the *includes*, the *requires*, the *URL* property and the self-invoking functions.

Include: The first facility is using the *Titanium.include()*. This function extracts the code from the given JavaScript file and inserts it within another file. This way big code files can be splitted into several parts to be able to manage them efficiently. The problem of this technique is that the code within the included file will be part of the global

scope. It means that the variables and the name of the functions are part of the same namespace and there could be repeated variables.

The first step to use *includes* inside an application is divide the application. In this example the application will be divided in two parts, one will contains the definition of a window and the other will contain the content of the window.

windowDefinition.js:

```
...
var window = Titanium.UI.createWindow( );
Titanium.include('windowContent.js');
...
```

windowContent.js:

```
...
var label = Titnaium.UI.createLabel({
    text: 'I am included content'
});
window.add(label);
...
```

Finally, the compiler extracts the code from windowContent.js file and inserts it where the include function was called. The windowContent name space is inserted in the windowContent's one. Thus, the code within windowContent file can access to the variables of windowDefinition.

Require: The next facility Appcelerator provides in order to have a modular application is the use of the CommonJS [30] function *require()*. This function execute the code within the required file and generates an API with the functions defined within. This API is exported as a variable so whenever the user want to access the functionality of one particular file, a simple call over that variable is needed. The variables and functions within the required file belongs to its own name space so the global scope is not saturated. The following code shows how this function has to be used:

windowDefinition.js:

```
...
var window = Titanium.UI.createWindow( );
var content = require('windowContent');
var label = content.newLabel();
window.add(label);
...
```

windowContent.js:

```
...
exports.newLabel = function (){
    var label = Titnaium.UI.createLabel({
        text: 'I am a included content'
    });
    return label;
}
```


URL property: The last modularity facility is to use the URL property of a Titanium.UI.Window object. This property allows the programmer to define the content of a window object on a different file keeping the variables and the namespace locally.

Self-invoking functions: Another technique to be used in a modular JavaScript application is the *self-invoking* functions. The primary motivation behind self-invoking functions is to create their own scope. In JavaScript, only functions have scope. The variables defined outside of a function are dumped into the global object. The following example code explains the features mentioned above:

```
windowDefinition.js:
...
var window = Titanium.UI.createWindow({
    url: 'windowContent.js',
    mycolor: 'red'
});
...

windowContent.js:
...
var win = Titanium.UI.currentWindow;
var label = Titanium.UI.createLabel({
    text: 'I am a included content'
    color: win.mycolor;
});
win.add(label);
...
```

Using these four techniques (includes, requires, URLs and self-invoking functions) inside an Appcelerator applications the final results will be easier to debug, to maintain, and to re-use.

3.1.2. Compilation of the code

There are two types of code that we will need to compile, static and dynamic code. The **static source** is analysed to find references to Titanium modules and the localization strings included into the xml files, the metadata included into tiapp.xml and density specific images are generated following the platform specifications. After this phase, the process is forked depending on the platform:

Android. First of all, an Eclipse project for Android is generated and setting up with the metadata mentioned before. After that the JavaScript code is compiled to Java byte code using the JavaScript framework Rhino JSC compiler [31]. The final step is to use the Android SDK tools to create the final APK file. It is out of the scope of this thesis to

explain how the Android SDK creates the APK file from the Java byte. More information about this process is available on the Android developers web page [32].

iOS. The first step is the creation of an XCode project and as it was done with the Android project, Titanium will use the metadata to configure the XCode project. After that, the JavaScript code is transformed into base64 format and inlined [33] as a variable into a generated C file. Finally, the XCode use this file to generate the final binaries and the final IPA.

In order to interpret and execute **dynamic source** a JavaScript interpreter is needed. To solve this problem Titanium includes a reduced version of those frameworks with the rest of the files of the application. For the iOS application, a forked version of WebKit's JavaScriptCore [34] is used, and a snapshot of Rhino 1.7 R3 CVS [31] is used for Android. On the last version of the Titanium SDK, the 1.8.1, a new JavaScript interpreter was introduced, the V8 [35]. The performance of V8 is better than Rhino's [36] but in this research Rhino will be used because there was not enough documentation about how V8 is involved on the compilation process.

3.1.3. Pack the application

As mentioned above, Appcelerator Titanium generates binary codes compiled into .apk or .ipa files to be installed directly on the corresponding devices. In order to generate the .apk file for Android, we have to select the *Distribute > Android Market Place* menu option and specify a private key that will be used to sign the application. In order to obtain this key we just need to go to the *Android Developers* web page and look for the guide to create a private key [37]. Regarding iOS, if we want to generate the .ipa we have to select the *Distribute > Apple iTunes Store* menu. In this case we will need to have a registered iOS device, and an Apple WWDR Intermediate Certificate, Distribution certificate and a Distribution Provisioning Profile.

3.2. Elements

In this section all the elements one can use to create an application with Titanium Appcelerator will be explained. For every element, examples of the final result on both platforms (Android and iOS) will be given, and, if the example requires it, program code is given.

3.2.1. Navigation

In Titanium allows users to implement two ways to navigate between windows. The first one is based on **Tabs** and the second one is based on **Windows**. It is also possible to mix the two systems to have a richer user experience.

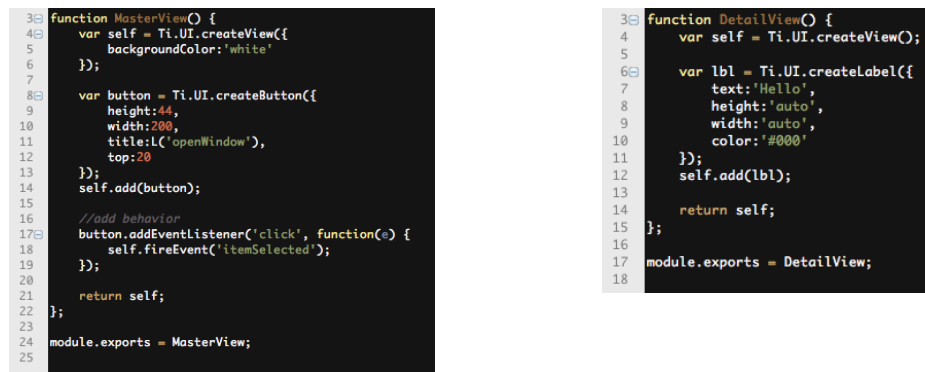
A **tab** structure represents a common tab structure where each tab represents an individual piece of information with the same hierarchical level than the rest of the tabs. In **Figure 3.4** the final result over the iPhone emulator and the Android device can be found.



Figure 3.4 Tab based app in iOS (left) and Android (right)

In order to achieve a tab based application, tabs and windows objects have to be created and the windows should be linked with the corresponding tab.

A **windows** system represents an application with hierarchical data or a stack of windows with different hierarchical level. In this case the coding is more complex than the tab based navigation because iOS and Android have different systems to manage the stack of windows in order to be able to come back to the previous windows. In this system there is a *Master view* that contains the components of the main window and a *Detail view* that contains the components for the second window on the hierarchy. The code needed to create this system is shown in **Figure 3.5**



```

38 function MasterView() {
39   var self = Ti.UI.createView({
40     backgroundColor: 'white'
41   });
42
43   var button = Ti.UI.createButton({
44     height: 44,
45     width: 200,
46     title: 'openWindow',
47     top: 20
48   });
49   self.add(button);
50
51   //add behavior
52   button.addEventListener('click', function(e) {
53     self.fireEvent('itemSelected');
54   });
55
56   return self;
57 }
58 module.exports = MasterView;

```

```

38 function DetailView() {
39   var self = Ti.UI.createView();
40
41   var lbl = Ti.UI.createLabel({
42     text: 'Hello',
43     height: 'auto',
44     width: 'auto',
45     color: '#000'
46   });
47   self.add(lbl);
48
49   return self;
50 };
51 module.exports = DetailView;

```

Figure 3.5 Master View and Detail View

In addition there is an *Application window* per platform we want to develop for, namely, one for Android and one for iOS (**Figure 3.6**). Inside this window object is where the platform specific behaviour will be implemented.



```

22 //determine platform
23 var osname = Ti.Platform.osname;
24
25 //require and open top level UI component
26 var Window;
27 if (osname === 'android') {
28   Window = require('ui/handheld/android/ApplicationWindow');
29 }
30 else {
31   Window = require('ui/handheld/ios/ApplicationWindow');
32 }
33
34 new Window().open();
35 }
36

```

Figure 3.6 Two application windows OS dependent

In **Android**, the code within this application window should define at least two windows: the actual *Application window* where we have to attach the *Master view* and another window that will contain the *Detail view* (**Figure 3.7**). When we want to navigate to the second window, we just need to add a button to the *Master view* with an action associated that will navigate to the *Detail view*. In this case the action is as simply as *open* the second window that contains the *Detail view*.

```

38 function ApplicationWindow() {
39   //declare module dependencies
40   var MasterView = require('ui/common/MasterView'),
41       DetailView = require('ui/common/DetailView');
42
43   //create object instance
44   var self = Ti.UI.createWindow({
45     title: 'Main Window',
46     navBarHidden: false,
47     backgroundColor: '#ffffff'
48   });
49
50   //construct UI
51   var masterView = new MasterView(),
52       detailView = new DetailView();
53
54   //create master view container
55   self.add(masterView);
56
57   //create detail view container
58   var detailContainerWindow = Ti.UI.createWindow({
59     title: 'Second Window',
60     navBarHidden: false,
61     backgroundColor: '#ffffff'
62   });
63   detailContainerWindow.add(detailView);
64
65   //add behavior for master view
66   masterView.addEventListener('itemSelected', function(e) {
67     detailView.fireEvent('itemSelected');
68     detailContainerWindow.open();
69   });
70
71   return self;
72 };
73
74 module.exports = ApplicationWindow;
75

```

Figure 3.7 Application Window for Android

In **iOS** the situation is a bit different because this operating system is designed to run over devices without a hardware button to go back to previous windows. Thus, a system to allow this behaviour should be implemented. For this reason we have to use the component *Navigation group* [38]. This element allows the device to know in which window we are and what was the previous window to allow the user navigate backwards. Hence a new window for the *Main view* has to be created and added to a navigation group previously created.

Following the same schema as Android does, a second window has to be created and linked with the *Detail view* (**Figure 3.8**). When we want to navigate to the second window we also have to add a button to the *Master view* with an action associated that will navigate to the *Detail view*, but in this case, we have to use the navigation group object in order to open the second window. As a result of this code, we end with these two applications shown in **Figure 3.9** Windows system for Android and **Figure 3.10** Windows system for iOS.

```

36 function ApplicationWindow() {
37   //declare module dependencies
38   var MasterView = require('ui/common/MasterView'),
39       DetailView = require('ui/common/DetailView');
40
41   //create object instance
42   var self = Ti.UI.createWindow({
43     backgroundColor: '#ffffff'
44   });
45
46   //construct UI
47   var masterView = new MasterView(),
48       detailView = new DetailView();
49
50   //create master view container
51   var masterContainerWindow = Ti.UI.createWindow({
52     title: 'Main Window'
53   });
54   masterContainerWindow.add(masterView);
55
56   //create detail view container
57   var detailContainerWindow = Ti.UI.createWindow({
58     title: 'Second Window'
59   });
60   detailContainerWindow.add(detailView);
61
62   //create iOS specific NavGroup UI
63   var navGroup = Ti.UI.iPhone.createNavigationGroup({
64     window: masterContainerWindow
65   });
66   self.add(navGroup);
67
68   //add behavior for master view
69   masterView.addEventListener('itemSelected', function(e) {
70     detailView.fireEvent('itemSelected');
71     navGroup.open(detailContainerWindow);
72   });
73
74   return self;
75 };
76 module.exports = ApplicationWindow;

```

Figure 3.8 Application Window for iOS



Figure 3.10 Windows system for iOS

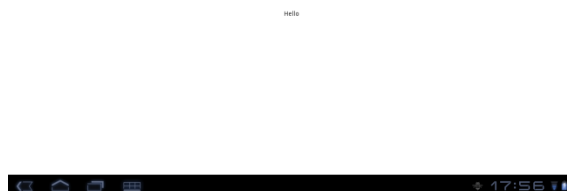


Figure 3.9 Windows system for Android

3.2.2. Controls

The graphical user interface controls that can be implemented for Android and iOS using Appcelerator Titanium include pickers, buttons, switchers, scrollers, table views, text labels, inputs, sliders, progress bars, image views, views and search bars.

There are three kind of **pickers** that can be implemented in Appcelerator Titanium for both Android and iOS platforms: date pickers, time pickers, and plain data pickers. All of them are implemented using the same *Picker* object, and depending on the attribute *type* the specific picker will be created. The results are shown in **Figure 3.11** for the iPhone emulator and in **Figure 3.12** for the Android device.



Figure 3.11 Pickers on iPhone emulator

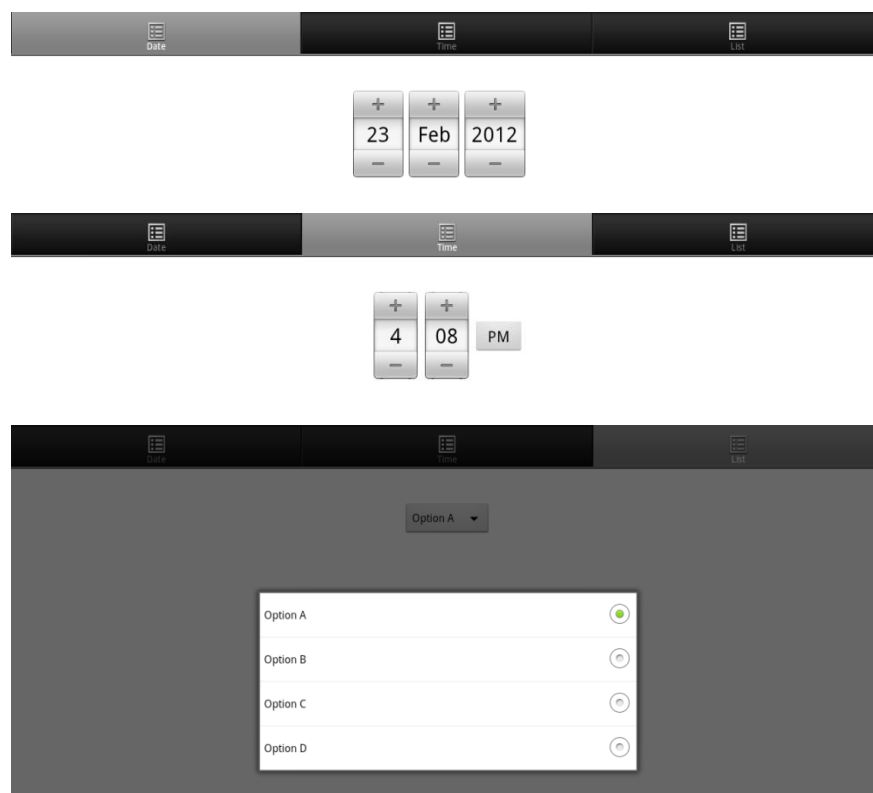


Figure 3.12 Pickers on Android

In addition, Titanium supports just two types of **buttons** that work for both Android and iOS platforms: the standard button and the switcher (with two states, on or off). In **Figure 3.13** we can see a standard button, following by another standard button with a background image, and finally a switcher with an *on* / *off* state.

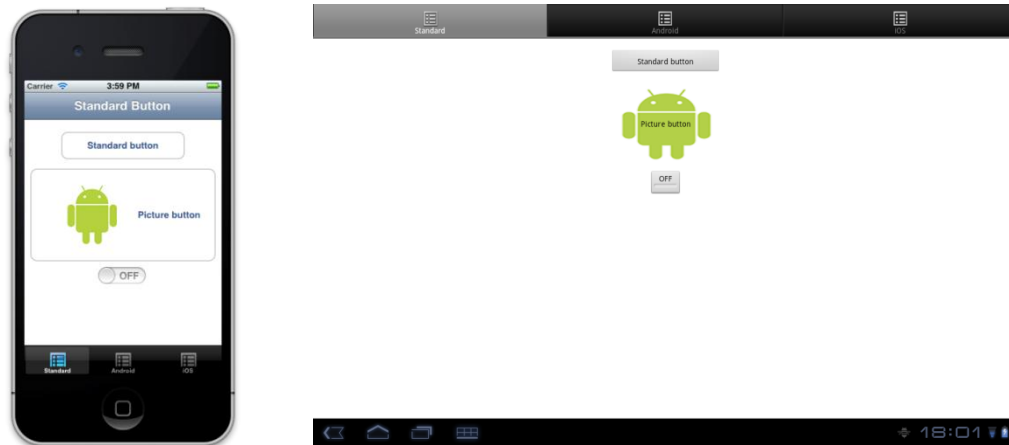


Figure 3.13 General Buttons in iOS (left) and Android (right)

In addition, there are individual features that can be implemented depending on the platform (**Figure 3.14**). For Android it is possible to modify the background of the button depending on the state, namely, selected, focused, or disabled. For iOS it is possible to change the style of the button using the attribute *style*. There are four possible values that will change the appearance of the button. Using a *BAR* style, we could integrate several buttons within a *button bar*. Using a *BORDERED* style, we can add a border to a standard button. If the *DONE* style is used, the button will include an icon that represents the end of a task. Finally, using a *PLAIN* style, our button will have no style and we will be able to add our own, like gradients, background colours or borders.

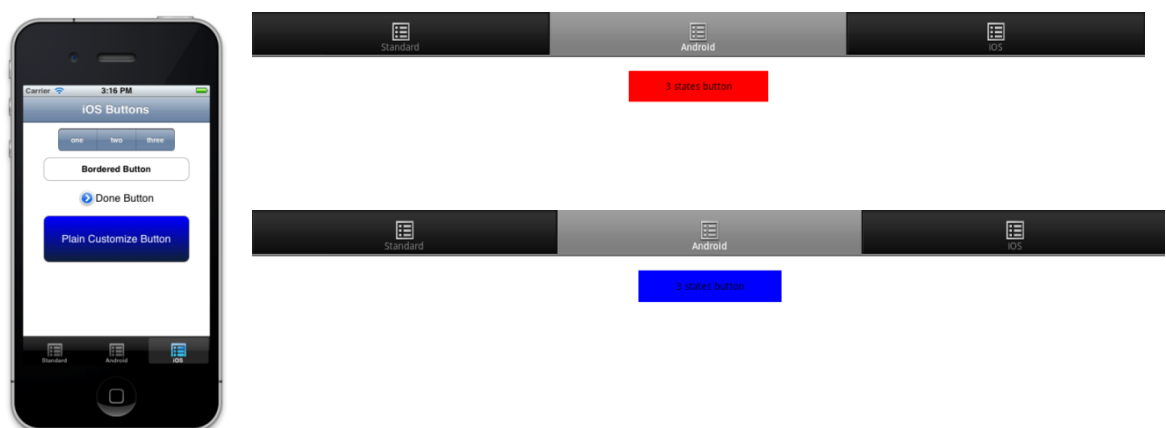


Figure 3.14 Special buttons in iOS (left) and Android (right)

Because the goal of a button is to initiate a certain action, we have to use a mechanism called *eventListener*. By adding this listener to the button, we are telling the operating

system that every time a specific event occurs over the button, the code within the listener should be executed.

Another GUI control is the **scroller**. Both Android and iOS allow scrollable views when the content exceeds the size of the container. There are two types of scroll that can be implemented for the two platforms: horizontal scrolling and vertical scrolling. In addition, there are two kinds of containers where scrolling is allowed: One can scroll over a *Titanium.UI.ScrollView* or over a *Titanium.UI.ScrollableView*. A *ScrollView* is a container that will store a normal view which content exceeds the dimensions of the container. This special view can use horizontal or vertical scrolling but only iOS allows both at the same time. If the *scrollType* property is not set, the scroll view attempts to deduce the scroll direction based on some of the other properties that have been set. Specifically, if *contentWidth* and *width* are both set and are equal to each other, or if they are both set and *showVerticalScrollIndicator* is set to true, then the scroll direction is set to "vertical". If *contentHeight* and *height* are both set and are equal, or if they are both set and *showHorizontalScrollIndicator* is set to true, then the scroll direction is set to "horizontal". If *scrollType* is set, it overrides the deduced setting. In case we want to scroll over different views, we have to use the *ScrollableView*. With this container, we can create an array of normal views, link them to the container and scroll horizontally over them. In order to have a better user experience, both platforms provide a system to inform the user that there is a scrollable area. To enable or disabled this feature the *showPaginControl* property should be set. In **Figure 3.15** there are examples of horizontal scrolling (left upper corner), vertical scrolling (right upper corner) and scrollable views (bottom). In this example **image** views are also used to show the direction of the scrolling.

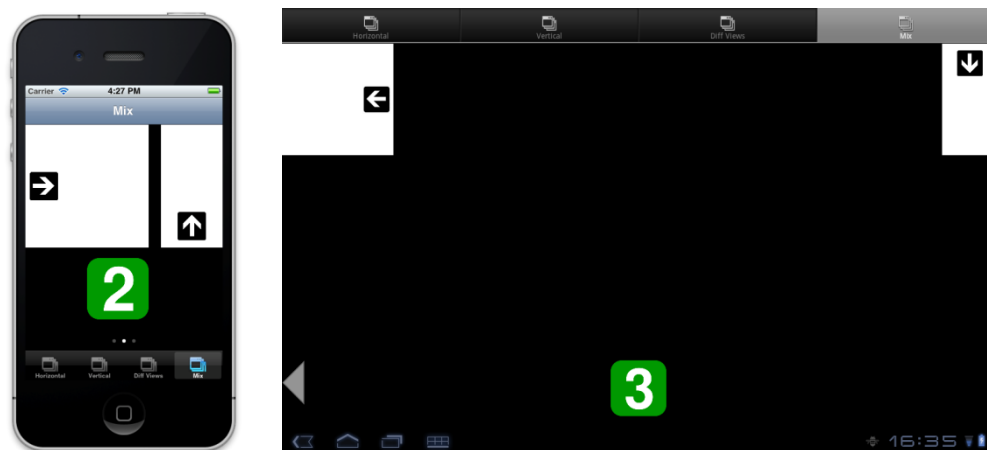


Figure 3.15 Scrolling in iOS (left) and Android (right)

One component that is used to organise the information inside a table is the **TableView**. This element consist on one *Titanium.UI.TableView* and within this table, there are *Titanium.UI.TableViewRow* objects. Within the row, any UI component can be added to

create from simple rows with just a title on them, to more complex ones with views, buttons, labels or more tables inside. By default, this table is scrollable vertically when the number of rows exceeds the size of the screen. In addition, it is possible to use a **search bar** to filter the table rows. In order to have this feature on our table a *Titanium.UI.SearchBar* has to be added.

There is a difference between the final result of the *Android* application and the *iOS* application. In *iOS* all the features of the search bar are working properly but in *Android* the *cancel* button that should clear the filter information is not working a manual removing of the characters on the filter is needed to restore the filtering. In **Figure 3.16** an example of the use of a *TableView* is shown.

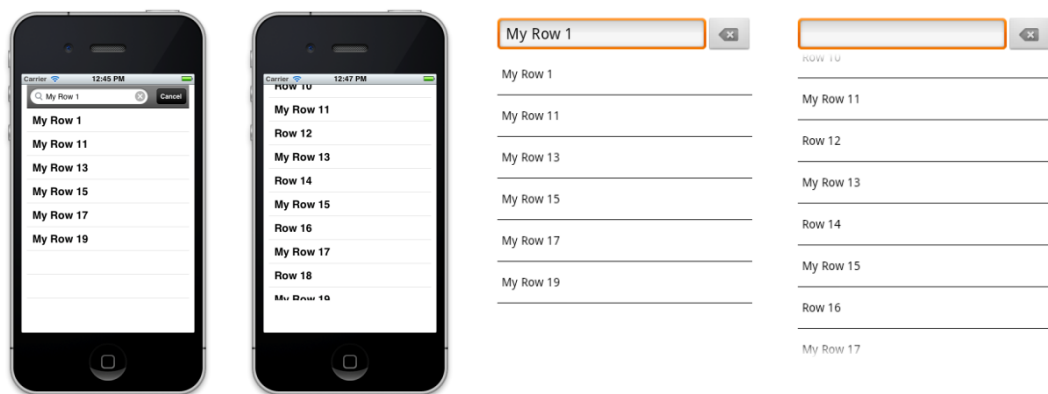


Figure 3.16 *TableView* in *iOS* (left) and *Android* (right)

The remaining GUI elements to be analysed are the **sliders**, the **progress bars**, and the **Alert dialogs**. The sliders are as simple as they are in Android or iOS and the result of the Appcelerator slider is the same in both platforms. To set up the minimum and the maximum values is the only important feature that should configure in order to use the slider into an application. The other element is the progress bar. A progress bar is a graphical element that shows a horizontal line that represents the progress of a defined action. To modify the value of the progress bar, the action linked to it should increase the value every unit of time. This unit of time is defined by the action itself. In the example used to explain this component, an *HTTP* connection is established and the monitor of the stream downloaded will be the one who will increase the value of the bar. The behaviour of the progress bar is the same in Android and iOS and the only difference is the visualization of the bar. Each operating system uses his own styles and Appcelerator does not provide any functionality to do a manual customize over the bar.

The last element used in this example is the alert dialog. These dialogs are small pop-up views with simple text and simple buttons that show the user some information. There are two ways to create an alert dialog. The first is using the function *JavaScript* function *alert()* including a message. This method will show the dialog with an OK button. The second method is using the object *Titanium.UI.AlertDialog* that can be customized by

the user adding titles or buttons with different functionalities. By default *Android* does not add any OK button to an empty alert dialog created by the Appcelerator method but *iOS* does. In **Figure 3.17** there is an example of the use of all these elements.



Figure 3.17 Sliders and Progress bars in iOS (up) and Android (down)

3.2.3. GPS and Maps

In order to implement the GPS system the first thing to be done is to give Android enough permissions [39] to access the data coming from the Internet connexion and the GPS. To achieve this, the following lines have to be added to the *TiApp.xml* file which will be partially used to create the Android manifest:

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

We need to remember that the signal is obtained asynchronously. Titanium provides two systems to access the data from the GPS: the function *Geolocation.getCurrentPosition()* (**Figure 3.18**) and the event *location*. The function *getCurrentPosition()* is an asynchronous function that is called once when the application is launched and the first connection with the GPS is done. Within the function, every feature the GPS provides is accessible through the *coords* object. The other option is the

asynchronous event *location*. This event triggers repeatedly on the location change. To manage this event, to add a listener to the Geo-location module is needed and the behaviour of this listener will be the same than the previous function accessing to the same *coords* object.

```

32 Titanium.Geolocation.getCurrentPosition(function(e){
33     var region = {
34         latitude: e.coords.latitude,
35         longitude: e.coords.longitude,
36         altitude: e.coords.altitude,
37         heading: e.coords.heading,
38         accuracy: e.coords.accuracy,
39         speed: e.coords.speed,
40         timestamp: e.coords.timestamp,
41         altitudeaccuracy: e.coords.altitudeAccuracy,
42     };
43 });

```

Figure 3.18 Asynchronous GPS data

The map systems in iOS and Android are based on Google Maps [13], and so Titanium's as well. The map is represented by a *Map.View* object, and it is used for embedding native mapping capabilities as a view. Over this view the mapping location, the type of map and the zoom level can be controlled and custom annotations can be added. An annotation is a geo positioned place represented with a pin over the map.

In order to use a map inside an application, a *Map.View* has to be created and parameterized (**Figure 3.19**). There are several attributes one can use to customize the map but the most important are three: the **map Type** attribute allows one to use satellite maps with real pictures, standard maps with roads and name streets, and a mix between both of them. **region Fit** is used to focus the view on the map's region. This region is defined calling method *setLocation()* and the information obtained with the functions/events mentioned above can be used to define the parameters of the region. The last one is **userLocation**; and it is used to show a special pin on the map that represents the current position of the device. In the current version of Titanium is used on this research, there is a bug over Android platforms that made this current location pin disappears whenever the view of the map is not showing [40].

```

6 var mapView = Titanium.Map.createView({
7     width: 300,
8     height: 300,
9     mapType: Ti.Map.STANDARD_TYPE,
10    regionFit: true,
11    userLocation: true,
12    visible: true,
13    touchEnabled: true,
14    focusable: false,
15 });

```

Figure 3.19 Map.View

```

32 var point = Ti.Map.createAnnotation({
33     latitude: lat,
34     longitude: lon,
35     title: name,
36     pinColor: Titanium.Map.ANNOTATION_GREEN,
37     subtitle: 'This is a random annotation',
38     animate: true,
39     myid: 1,
40 });
41 mapView.addAnnotation(point);

```

Figure 3.20 MapAnnotation

The last step is to explain how to add, modify and remove annotations to the map view. In order to **add an annotation**, a *Map.Annotation* object has to be created using the function *Map.createAnnotation()*. This called has to be customized with several param-

ters like the *latitude*, *longitude*, or *pincolor*. **Figure 3.20** shows how to create a simple annotation. One consideration we need to have when we are defining the colour of the pin is that there is a bug on Titanium [41]. This bug made the Android application crash when a click event is fired over an annotation which colour was not established as *Map.ANNOTATION_RED*, *Map.ANNOTATION_PURPLE* or *Map.ANNOTATION_GREEN*. If the colour is not defined, Titanium will use blue as predefined and the application will crash. Once the annotation is added to the map we can **edit** it or remove it from the view. To modify an annotation we need to access the object and use the set of function provided by this object like *setTitle*, *setPincolor*, etc. The changes will affect immediately to the map annotation in iOS but not in Android. It means that if we change the colour of the pin, in iOS we will see the new colour on the map annotation instantly but in Android we will not. To see the change over the map, we will need to force Titanium to refresh the annotations over the map adding new ones or deleting any of them.

To **remove an annotation** we have to access the *map view* and remove it from the list of annotations using method *removeAnnotation()*. Both editing and removing actions need to have the annotation object. Because Titanium does not allow to recover the annotations that were created using method *addAnnotation()* we need to maintain our own list of annotations object to be able to modify them or delete them.

In **Figure 3.21** and **Figure 3.22** we show the results of all the features commented above:



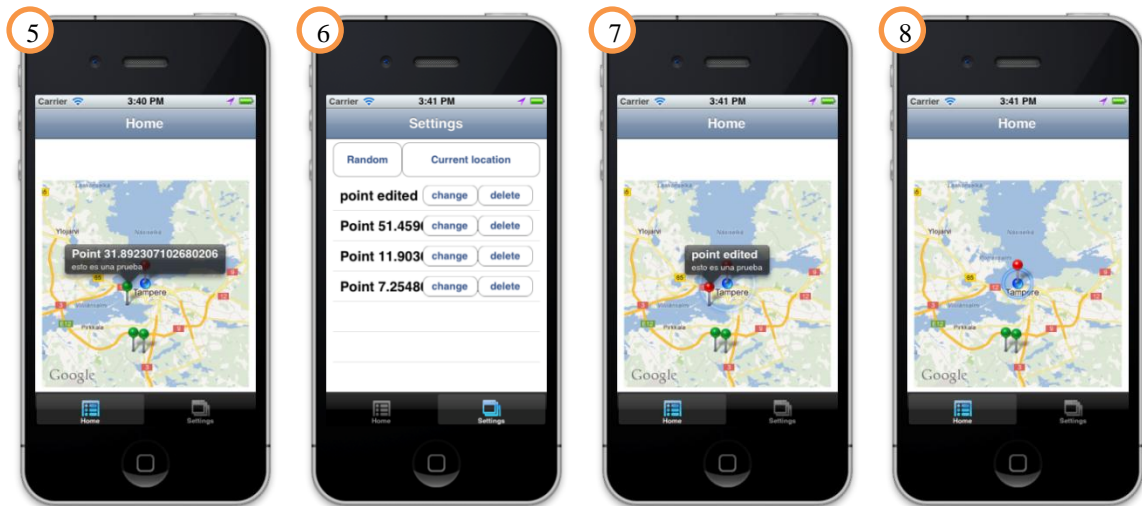


Figure 3.21 Map and GPS example in iOS

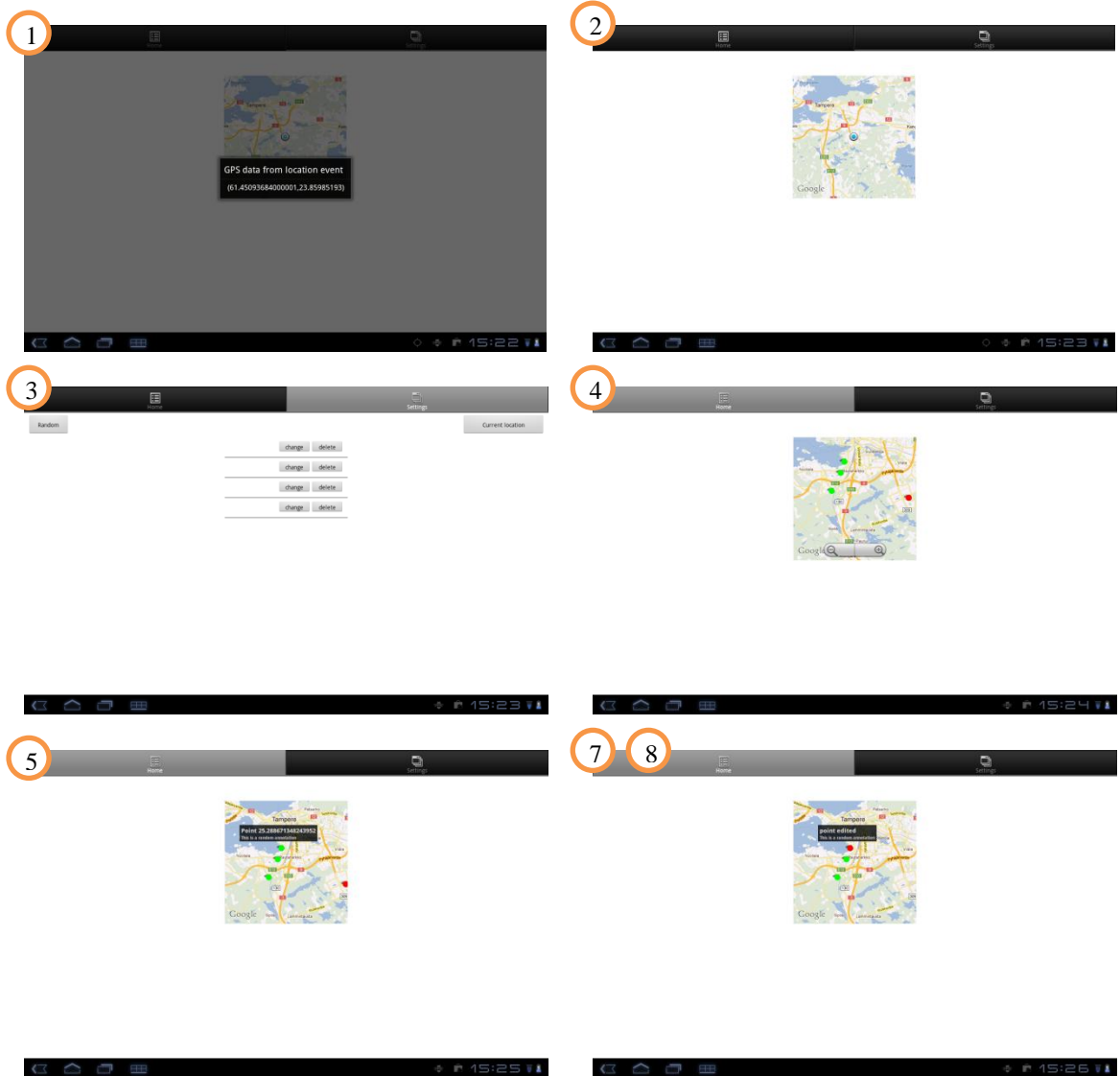


Figure 3.22 Map and GPS example in Android

1. Open the app and first detection of GPS signal.
2. Map with the current location of the user (blue pin).
3. Add three random pins and one pin on the current location (the last one).
4. Visualization of the pins added.
5. Checking the information of the first pin.
6. First pin edited. The name and the colour will change.
7. Visualization of the pin modified.
8. Visualization of the map after delete de modified pin.

3.2.4. Camera

In this case, we will focus only on Android due to the impossibility to access the camera inside the iPhone emulator. For this reason we will also not discuss about the features Appcelerator provides for use the camera in iPhone.

In order to access the camera functionalities Titanium provides several tools within module *Titanium.Media*. The first step to use the camera is, like on the GPS, is to set up the Android permissions customizing the *TiApp.xml* file:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

After that, function *Titanium.Media.showCamera()* has to be called and customized with several parameters and callback functions, namely, *success*, *cancel* and *error* that define the behaviour of the application when the event defined by the name appears.

As mentioned above, there are two options to use the phone camera: use the interface provided by the operating system to interact with it (**Figure 3.23**) or access directly to the raw image captured by the camera, and implement the rest of functions by ourselves like the button that takes the picture (**Figure 3.24**).

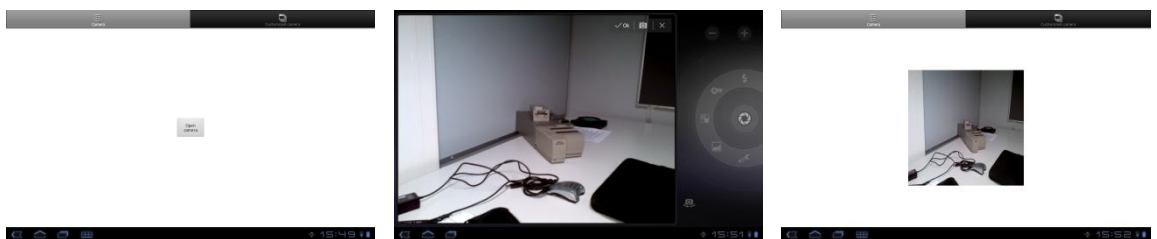


Figure 3.23 Predefined camera interface on Android

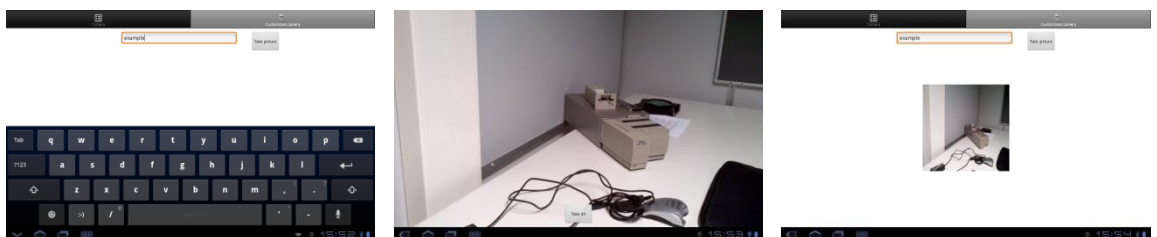


Figure 3.24 Customized camera interface on Android

In Appcelerator Titanium, over the Android platform, the pictures are stored by default on the picture album with a specific name Android defines, but it is easily modified to be stored in a different place with a different name.

3.2.5. Audio

There are two basic functions when Titanium Appcelerator is dealing with audio. The first one is to reproduce audio (from a local file or from Internet) and the second one is to record audio. The first feature we will analyse is how Appcelerator reproduces audio in iOS and Android platforms. Nevertheless the audio recording feature will be tested in Android platform only due to the impossibility to use an audio input on the iOS emulator.

When we want to **reproduce** a sound we have to take into account what kind of source we are using. In the case we are using a local source we have to use the object *Titanium.Media.Sound*. In the other hand, if we want to use an Internet file to reproduce it as a streaming, the object *Titanium.Media.AudioPlayer* should be used. Both solutions are based on create a stream object of bits (the audio information) and associate it with a state, namely, *playing* and *paused*, to control the flow of the playback. In **Figure 3.25**, there is an example of the features mentioned above.

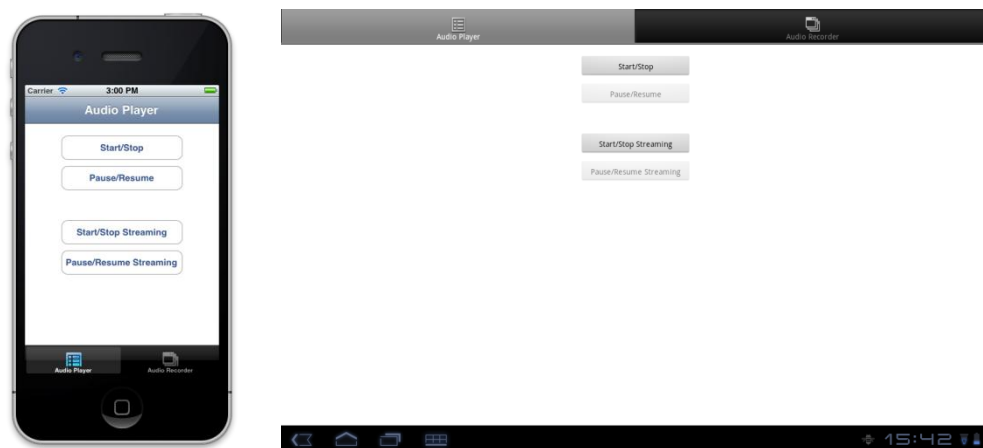


Figure 3.25 Audio Player in iOS (left) and Android (right)

The other possible feature one may include in an application is to **record** audio. Appcelerator provides a module called *Titanium.Media.AudioRecorder* to fulfil this aspect. Unfortunately this module is only available for iOS in this version of the Appcelerator SDK. For the same reason given on the camera paragraph, due to my impossibility to obtain an iPhone and because the absence of a microphone on the iOS emulator we will focus the development of an audio recorder with Appcelerator for Android platforms. There are not a module that implements the audio recording for Android so we have to use a third-party module that implements this functionality. In particular, we are going

to use the *AudioRecorder* module provided by *Codeboxed* [42]. This code is an external module precompiled and ready to use. This module was successfully tested with the 1.7.5 version of Titanium Appcelerator but with the upgrading to the 1.8.1 version this module is not compatible anymore. For this reason all the future mentions will be referring to the implementations done in the previous version.

The first step to use this module is to download the binary codes packed into a *.zip* and place it in the root folder of the Titanium project. After that we have to add an entry to the *TiApp.xml* file to indicate the framework where is the code we are going to use:

```
<module platform="android" version="2.0">com.codeboxed.audiorecorder</module>
```

When we proceed to compile and execute the application, Titanium unzips the module binaries and creates a files structure into the project starting on the folder *modules*. The final step is add a reference to the module using the *require* directive and the module can be used.

This module follows the same logic than the original Appcelerator *AudioRecorder* for iOS, allowing the user to pre-set the format, name, duration, compression, etc. of the audio. It also allows the features of a normal audio player like play the sound recorded and stops it.

3.2.6. Storage

In order to keep the information generated by an Appcelerator application we can use the three techniques mentioned above, namely, use the device file system to store the information into a plain file or into a XML file, or use an SQLite database. To access to the device's file system we have to use the *Titanium.Filesystem* module. It provides basic functions that can be used to create, open, write, read or remove files and directories. The use of this API is very similar to any other OS. File system so it will not be explained here. The only aspect worthy to mention and that affects directly to the matter, the cross-platform application, is where is allowed to store information within the device file system. For security reasons Android and iOS use a *sandbox* [43] and [44] design to protect the applications and the operating system. For this reason Appcelerator provides direct access to the following variables:

applicationDataDirectory: This is a writable directory that can be used to store applications-specific files. In iOS, this directory is specifically designated for user documents. In Android, files in this directory cannot be accessed by other applications, so it should not be used for files that must be used by another application.

applicationDirectory: Path to the iOS application directory.

applicationSupportDirectory: This is a writable directory used on iOS for application files that are not user documents.

externalStorageDirectory: This directory can be used for storing arbitrary data on removable storage, such as SD card. It is read/write and files in this directory can be passed to other applications.

resourcesDirectory: It contains the path to the application's resource directory. It is a read-only directory so in case of modifying any files in this directory, they must first be copied to another directory. It is important to note that when the application is running on the emulator, the resources directory may be writable, but it is not writable on device.

Once we know where we can store our directories and files we can pay attention to a very useful time of file, the XML. Using the *Titanium.XML* module we have access to the *JavaScript DOM Level 2* [45], and thus all the functions this standard defines are implemented in Titanium Appcelerator.

Finally, the last method to store data on the device is using the *SQLite* database. Appcelerator provides just the functionality to open the database and execute commands so notions on *MySQL* are needed in order to interact with the database. The three basic steps we need to when we want to use a database are the creation/opening the data base using function *Titanium.Database.open()*, the execution of any *MySQL* command using *Titanium.Database.execute()* function and at the end, the closing of the database using the function *Titanium.Database.close()*. Closing the database is recommended in order to save resources on the device.

4. ANALYSIS OF A CROSS-PLATFORM MOBILE APPLICATION

Once we know how to deal with the different features Appcelerator provides, we can create an application that uses all of them. In this chapter an example application will be created so we can analyse how easy it is to implement all the features in the same system. As it was mentioned on the previous chapter, the tools going to be used to develop the application are an iMac computer with MacOS 10.X, the Xcode 4.2.1, the Android SDK v3.1 and Appcelerator Titanium 1.8.1 framework. The application is made for be running on Android devices and iPhones being a MOTOROLA Xoom Tablet the main device for testing the Android application and the iPhone Emulator that XCode provides the virtual device for testing the iOS application.

At the end of this chapter the development of the whole application will be analysed, pointing out the difficulties and difference integrating the features mentioned on the previous chapter on the same system.

4.1. Use Case: Travel Guides application

This application is a geo positioning based application whose main goal is to allow users to create their own digital travel guides when they are walking on the streets. A digital travel guide is a simplified digital version of a physical travel guide book. When a user creates a travel guide, he/she writes a name and a description for the guide and choose the city where the guide belongs. Furthermore, the user can add places to the guide. When the user is located on a place she/he thinks is interesting, she/he can create a new place that will be linked with the geo location of the device. The place will be defined by a name and a description and the user can also add a picture taken by his camera or an audio recording describing that place. Based on this the user can:

1. Create a new travel guide. A travel guide is defined by a name and a city (mandatory fields) and a description.
2. Create a new place within the travel guide. A place is defined by a name (mandatory field), the coordinates (obtained automatically by the application) and a description. The user can also add a picture taken by the device camera or an audio recording.

3. Visualise the guides created. The application must show a list with all the guides and offers a system to open the guides.
4. Visualise the places created within a travel guide. The application must show a list with all the places belonging to a guide and offers a system to open the place.
5. Edit an already created travel guide. The user can modify the name, the city or the description. She/he can also remove, edit or crate new places.
6. Edit an already created place. The user can modify the name and the description but not the picture or the audio.
7. Eliminate a travel guide. Delete a travel guide from the list of guides and remove it from the interface.
8. Visualize every place from a guide into a map. For this feature the user will need Internet access at the moment of the visualization.

Besides, the application:

1. needs a device with GPS in order to obtain the location of the places.
2. will run on devices with the Android 3.1 version.
3. will run on iPhone devices with the iOS 4.X version.

4.2. Analysis of the requirements

Once we know the requirements of the application, we can analyse them to extract the information regarding the different features the application should provide and needed to build the system.

Create a new guide (**Figure 4.1**) consists of the creation of an object within the application that contains the basic information of the guide and pointers to the places that belongs to them.

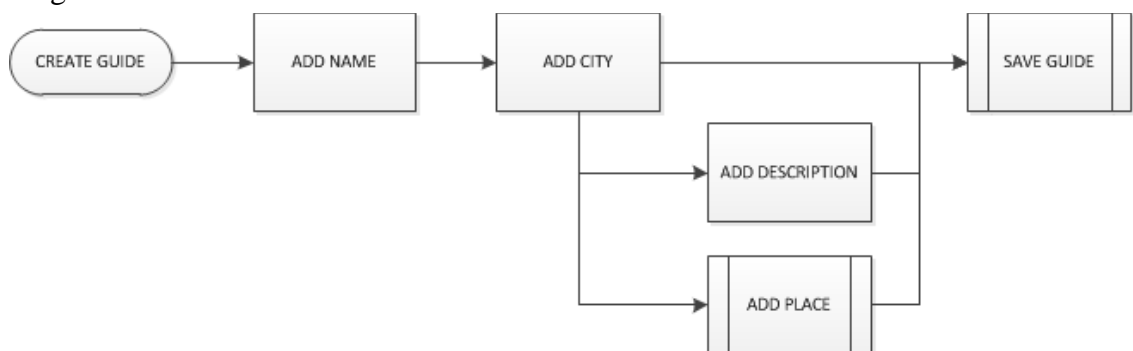


Figure 4.1 Create Guide

Create a new place (**Figure 4.2**) consist of create an object within the application that contains the basic information of the place. Also during this process the audio and pic-

ture files that could be taken will be renamed with the identification number of the particular place to be able to know what file corresponds to what place.

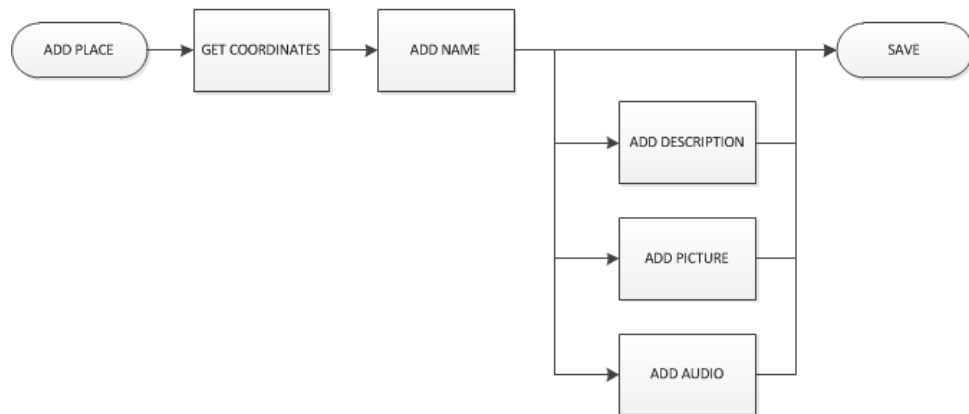


Figure 4.2 Create place

Save a guide (**Figure 4.3**) consists of the creation of a folder with an identification number for the guide as a name and within, create a XML file, *info.xml*, that contains the basic information of the guide: the identification number (given by the application), the name, the city and the description (optional). Within this folder, there will be as much folders as places the particular guide has. The name of those folders will follow the same criteria of the parent folder and they will use an identification number of each place. Each place folder will have inside an xml file, *info.xml* that contains the basic information of the place: the identification number (given by the application), the name, a description (optional), the latitude, the longitude and a reference to the places where the media files (pictures and audio) will be stored during the creation process and later. It is important to remark that in the case the media files are stored somewhere else outside the application directory, and we will have to move them to the final location.

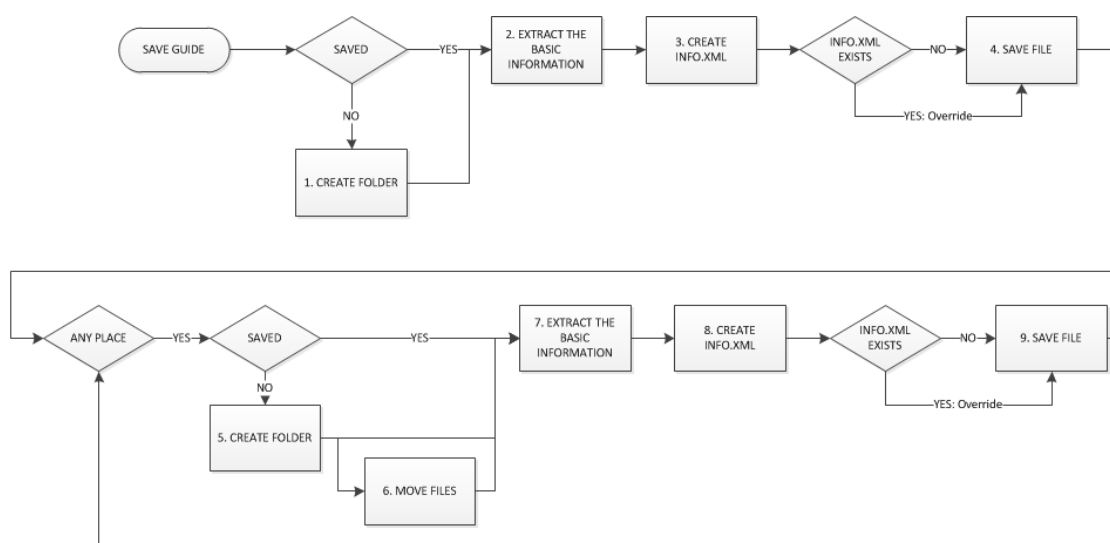


Figure 4.3 Save guide

4.3. Application design

Once we have defined the behaviour of the application we have to design the application and translate the diagrams into the JavaScript code. The application structure will be the one defined in **Figure 4.4**.

- The application directory contains a *Resources* folder with the application code.
- The *PLACE_AUX* directory contains folder structure needed to temporary host the media files (picture and audio) of the places of the guide that is being created by the user on that moment.
- The *MY_GUIDES* directory contains the folder structure needed to store all the information of all the guides created by the user. This folder is kept until the application is removed from the device.

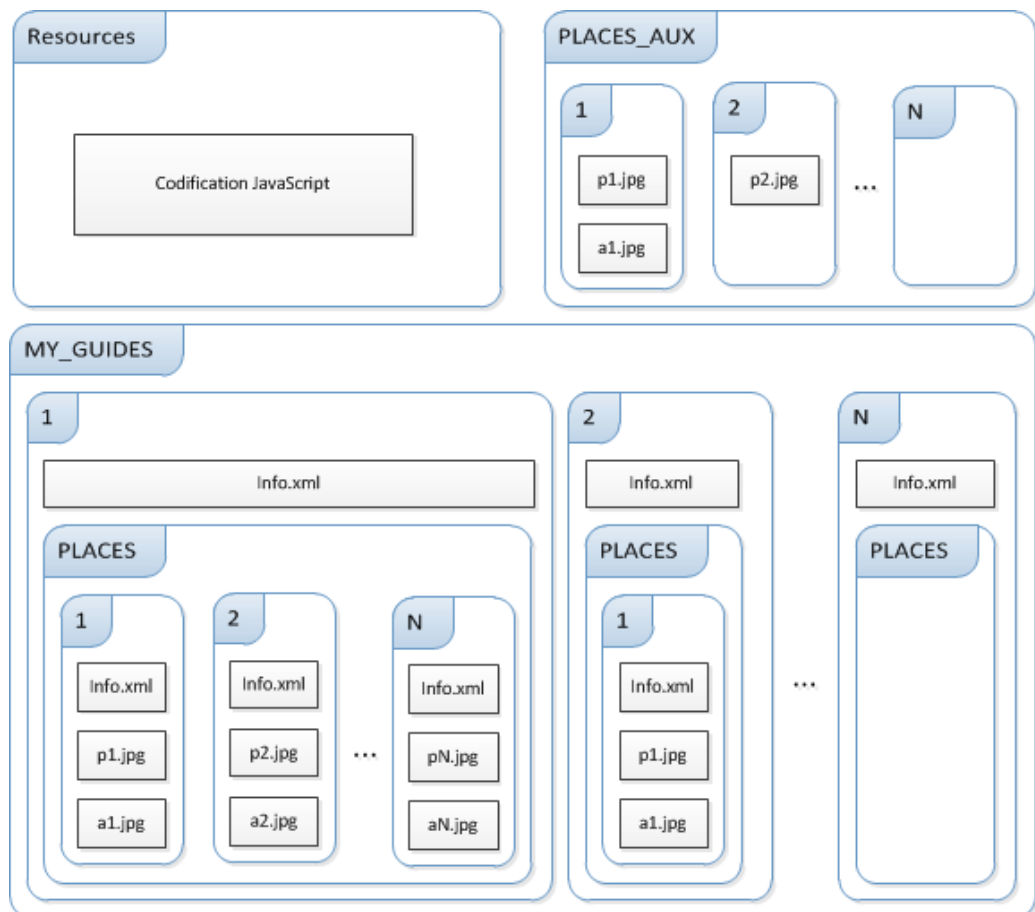


Figure 4.4 Application structure

The implementation of the applications follows the design shown in **Figure 4.5**.

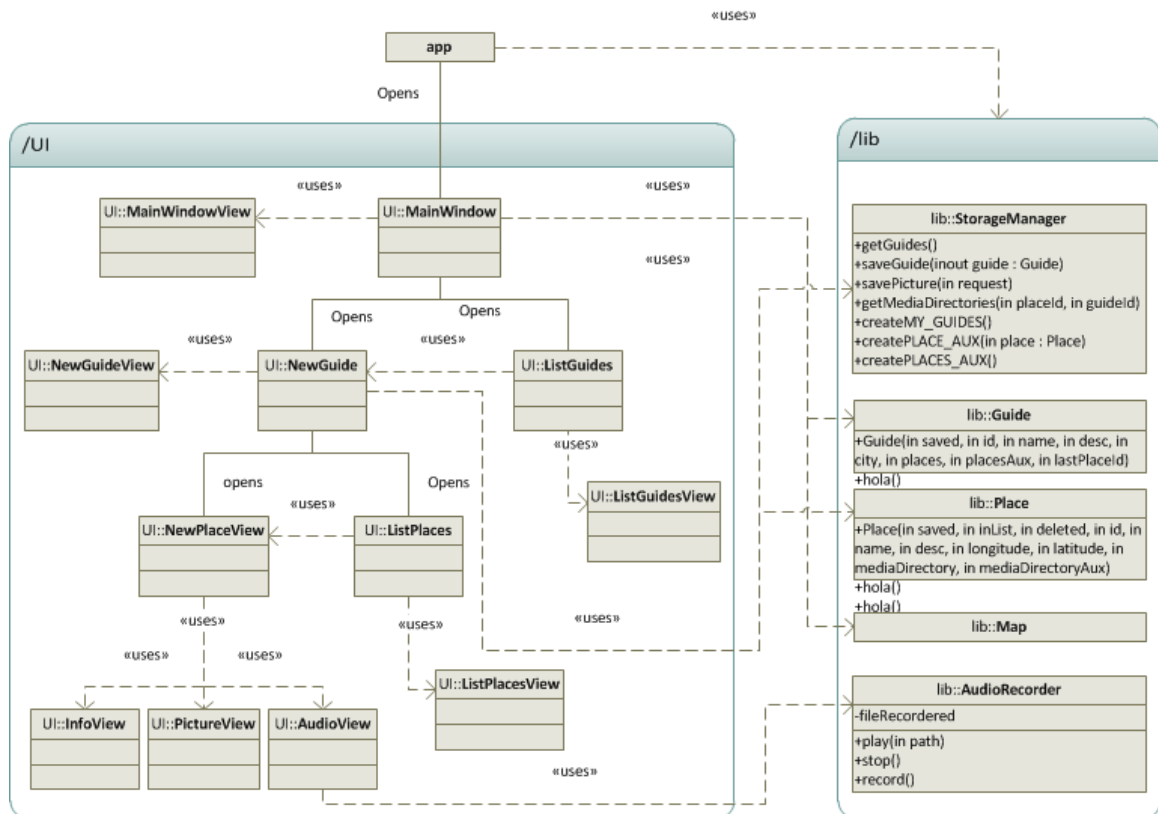


Figure 4.5 Application diagram: Class approach diagram of the application

The application is divided into three main modules: user interface module (UI module), library module (lib module) and application module (app module). These will be discussed in detail in the following.

4.3.1. UI module

The UI module shows the information of the application to the user. It allows the user interact with the application to create and visualise the guides. The graphical user interface is represented as a tree where the windows that are not leaves have their own window managers (*MainWindow*, *NewGuide*, *ListGuides* and *ListPlaces*) with the views belonging to them (*MainWindowView*, *NewGuideView*, *ListGuidesView*, *ListPlacesView*) and the windows that are leaves (*NewPlaceView*) are controlling by its parents, in this case, *NewGuide* and *ListPlaces*. The window managers are the responsible to define and control the navigation between its children windows. They handle every event arising in the sub windows. The views elements display the controls and the information of the application.

MainWindow: displays the *MainWindowView* that shows the two actions the user can select when they open the application: Create a new guide and show a list of the guides created by the user. This manager also control the navigation between the windows needed to fulfil these actions (*NewGuide* and *ListGuides*).

ListGuides: displays the *ListGuidesView* that shows the list of guides already created by the user. When the user does click in one of the guides of the list, this guide is shown using the manager *NewGuide*.

NewGuide: controls the information shown by the *NewGuideView* and needed to create a new guide. This view is divided into two views, one for show the basic information and other to show a map and a menu to add or list places. These views will be shown accordingly to the user actions. First the info view is shown and when the user does click on one button, the other places view is shown.

ListPlaces: follows the same logic than the *ListGuides* manager. It displays a view with the list of places (*ListPlacesView*) created for a specific guide and when the user do click in one of the places, the place is shown using the manager *NewPlace*. The view attached to the manager also offers the possibility to remove the place from the list and thus, from the map that represent the list of places.

NewPlaceView: This view is the ones that show the information needed to create a new place. It is divided into 3 more views, one for the basic information (*infoView*), one for the picture information (*pictureView*) and other for the audio information (*audioView*).

4.3.2. lib module

The lib module encapsulates the functionality that is external to the graphical user interface and the main logic of the application. There are five libraries in this module:

StorageManager: This library is used to have a unique point of interaction with the device file system.

Map: This library is used to encapsulate the map view and the location events. This view will be just one view created because Android does not allow more than one map view per application.

AudioRecorder: This library is used to encapsulate the interaction with the external audio recording module.

Guide: this library is the definition of a guide object. It is the ones that knows how to create an object of that type returning a JavaScript Object Notation (JSON) object.

Place: this library is the definition of a place object. It is the ones that knows how to create an object of that type returning a JSON object.

4.3.3. app module

This module contains just one element, the *app.js* file. This object is the starting point of the application and it is the responsible to initialise the graphical user interface and listen to the possible petitions it may need (save guides, read guides, etc.).

4.3.4. Screenshots

Next pictures show the final result of the use case application. In **Figure 4.6** we can see the main window of the application where the user can create a new guide or access to the list of guides already created.

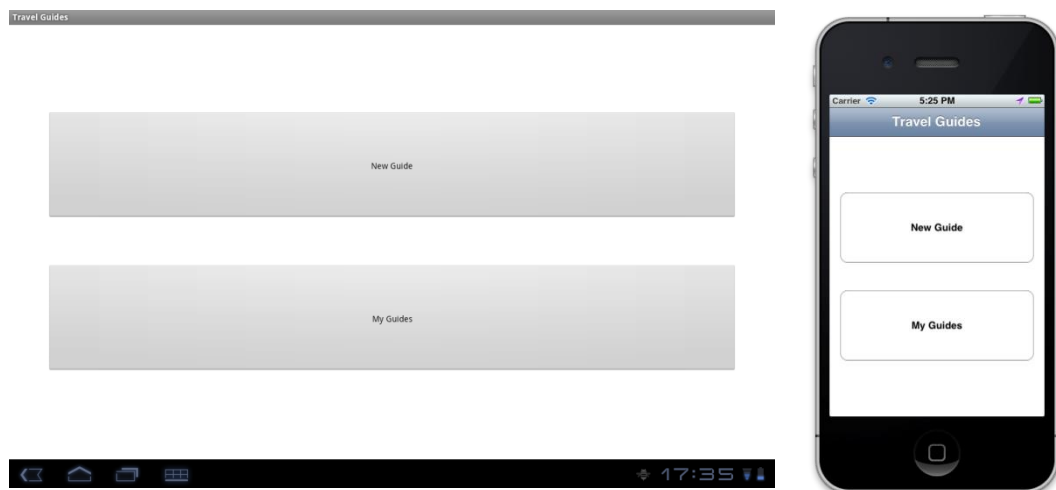


Figure 4.6 Main Window in Android (left) and iOS (right)

In **Figure 4.7** we can see the interface to define the basic information of a new guide. There are two input fields and one list picker. The input fields are used to define the name of the guide and a description. The list picker is used to select the city of the guide. The only appreciable difference between the Android and iOS interfaces is the list picker. In Android the list picker is smaller and the action needed to select one item is different than in iOS.

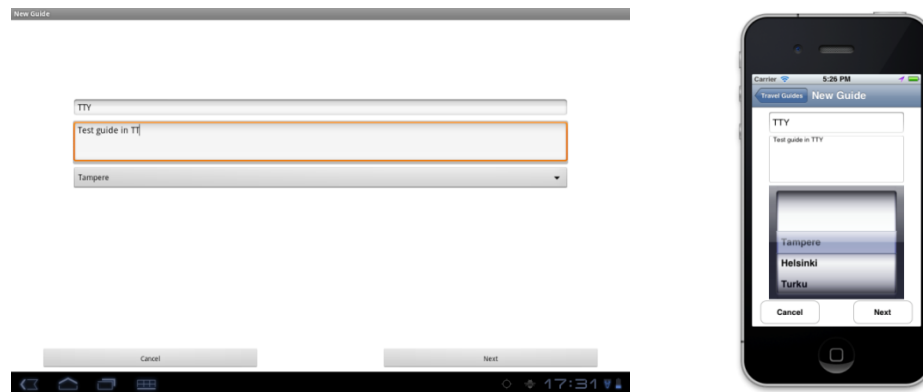


Figure 4.7 *New Guide: Basic Info in Android (left) and iOS (right)*

In **Figure 4.8** we can see the interface where the user can create new places within the new guide or show the list of places already created within a new guide. It also shows a map with the places created. This map is displayed using the map system of Android or iOS. Both platforms are using Google Maps so an Internet connection is needed in order to display this map.

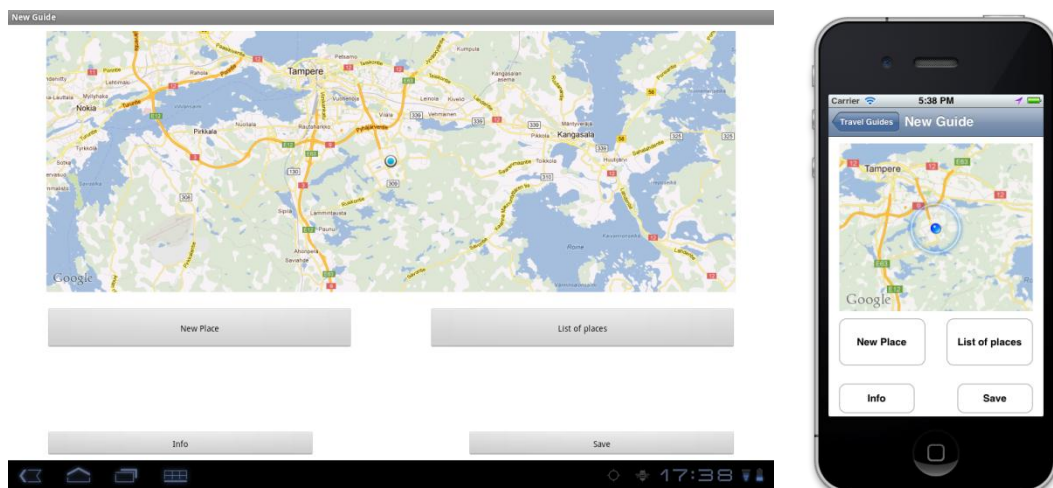


Figure 4.8 *New Guide: Places in Android (left) and iOS (right)*

In **Figure 4.9** we can see the basic information of a new place. This information is needed to create the place. In this case the interface contains two inputs to specify the name of the place and a description. This interface is the first out of a group of three. To change between interfaces a swip gesture is needed.

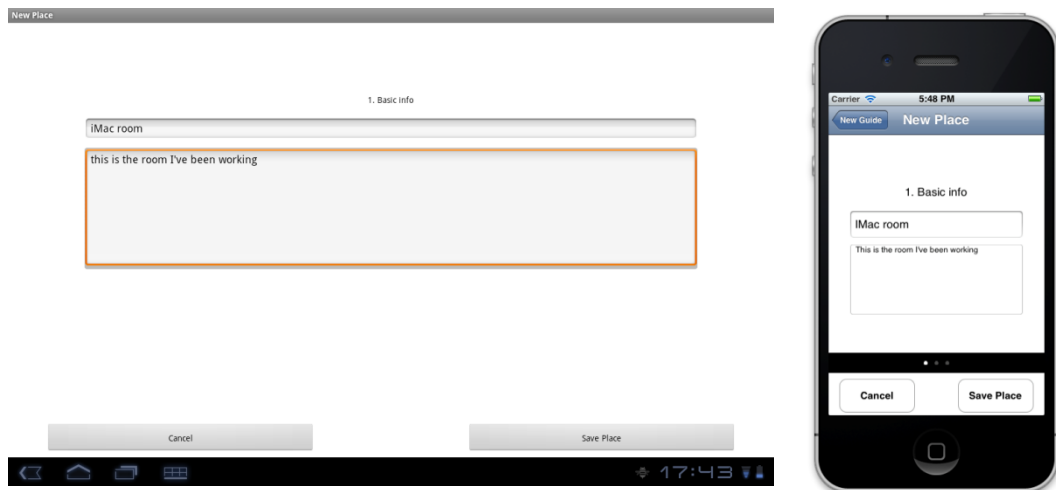


Figure 4.9 *New Place: Basic Info in Android (left) and iOS (right)*

The second interface of the group is shown in **Figure 4.10**. On the Android interface an image view is shown after the user take a picture with the camera. On the iOS version an alert dialog is displayed because the iOS emulator does not provide camera emulation.

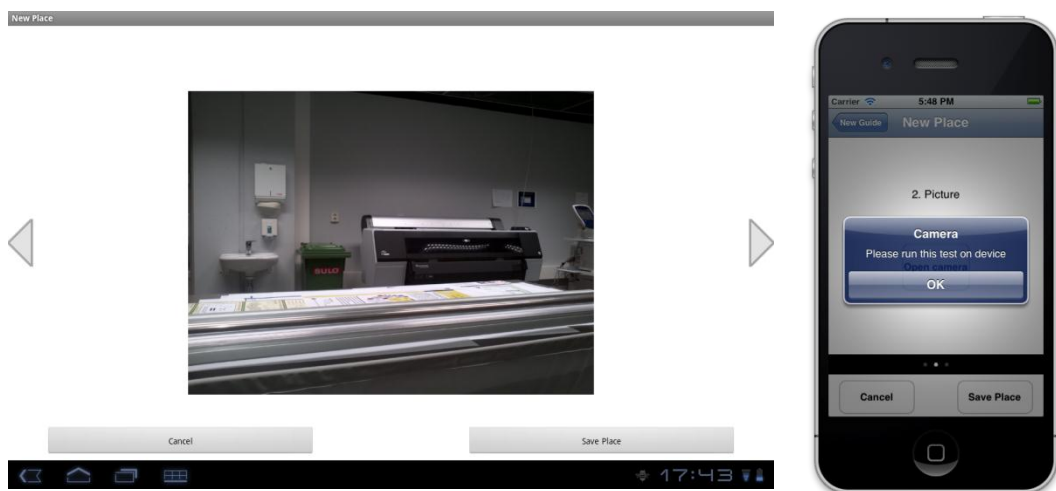


Figure 4.10 *New Place: Picture view after take the picture in Android (left) and iOS (right)*

The last interface of the group of three is shown in **Figure 4.11**. In this screen we can see the interface to add a new audio to the place. As it was mentioned before, the module to record audio is not working anymore so the interface does not allow the user to interact with the audio recorder.

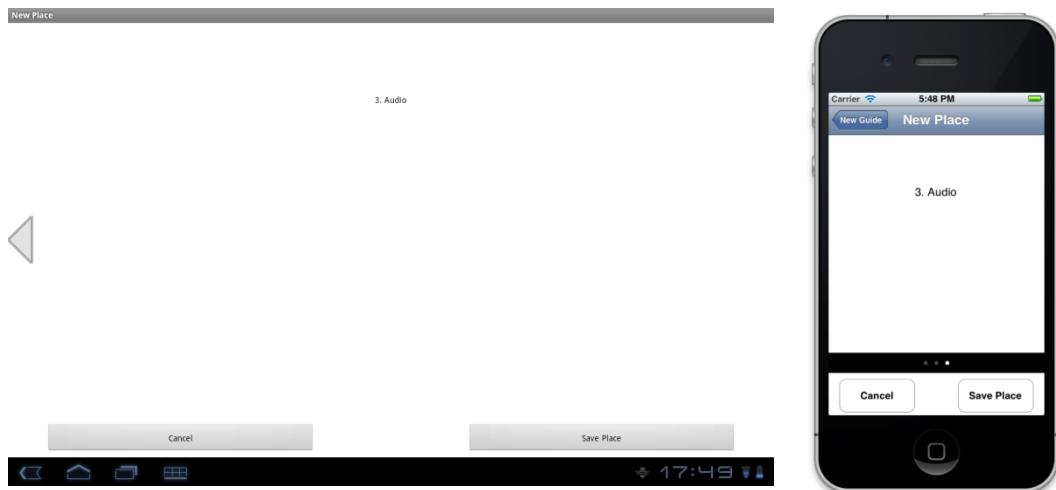


Figure 4.11 *New Place: Audio in Android (left) and iOS (right)*

In **Figure 4.12** we can see how the interface informs the user about a mandatory field. In this case is informing to the user that the input field needed to specify the name of the place is empty.

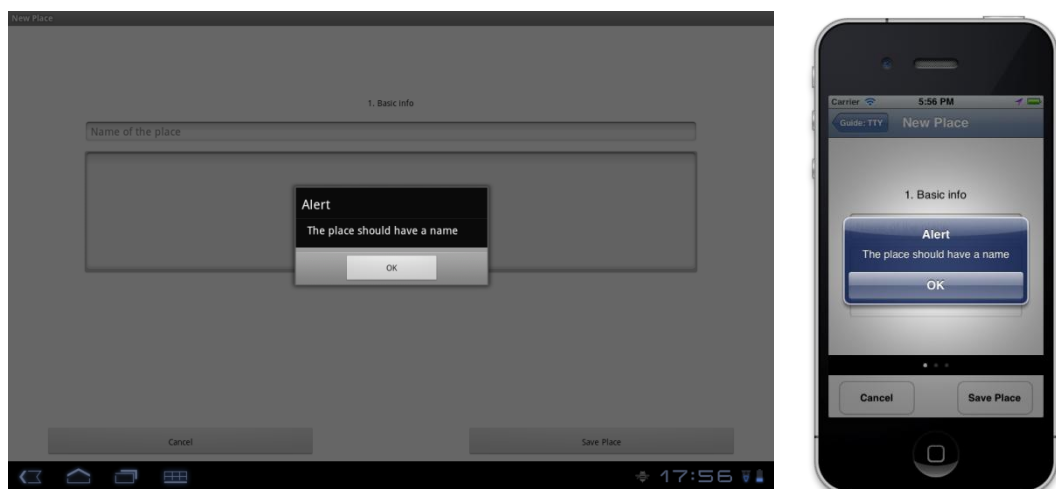


Figure 4.12 *New Place: No name alert in Android (left) and iOS (right)*

In **Figure 4.13** we can see the list of guides of the user. Every element of the list contains two elements: A text with the name of the guide and a button with the “x” character to delete the guide from the application. In order to open an specific guide, the user has to click in the specific row of the list.



Figure 4.13 List of guides in Android (left) and iOS (right)

In **Figure 4.14** we can see the map with the places belonging to one guide. As it was explained on a previous figure, the places are shown on the map. A green pin is used in order to point the places on the map. To visualize the basic information of the place the user has to click in any of the pins.

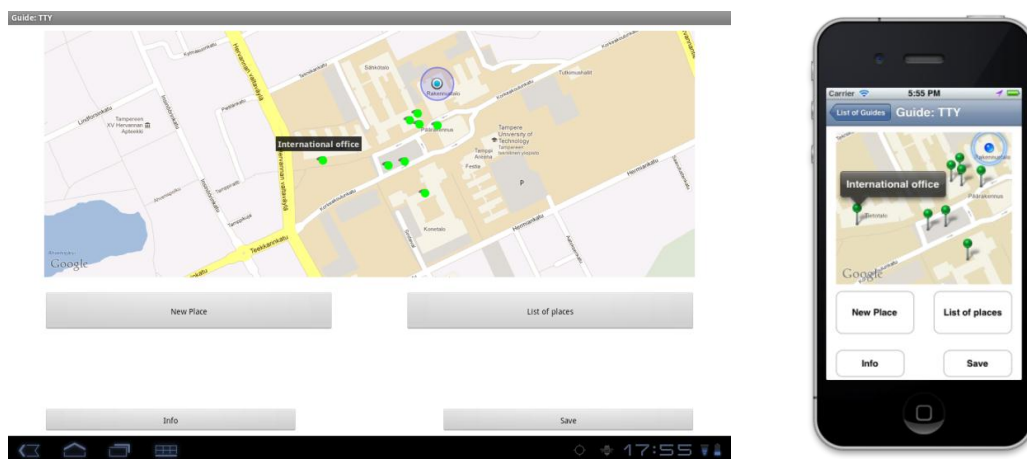


Figure 4.14 List of places: Map in Android (left) and iOS (right)

In **Figure 4.15** we can see the list of places belonging to one guide. Every element of the list contains two elements: A text with the name of the place and a button with the “x” character to delete the place from the guide. In order to open an specific place, the user has to click in the specific row of the list.



Figure 4.15 List of places: List in Android (left) and iOS (right)

4.4. Evaluation of the framework

The first point of the evaluation is the operating system where the framework can work. During the development of this thesis we used three kinds of environments: Linux, Windows and MacOS. All of them work perfectly with Titanium Appcelerator but there is one aspect that makes the MacOS environment best suited. While the Android SDK works in any of the three environments, the XCode, needed to compile the iOS application, is working only in MacOS environment. For this reason it is advisable to use Appcelerator only on systems (virtual or physical) with the Apple operating system.

The second aspect to be analysed is how is the interaction with the framework, including the programming part, the framework support and the external dependencies. Regarding the programming task, JavaScript is most commonly used in Web environment where the techniques applied are a bit different from the ones that have to be used in Appcelerator. For this reason, the users that know JavaScript because the use on web technologies will have to learn how to use properly the language in order to have a correct behaviour of the application. This fact made the learning process a bit slow in the beginning but once a proper knowledge about how to properly use the tools is acquired, the development process goes quite fast. To help this process it is available a debug mode (we have to remind that Appcelerator Titanium is a framework based on Eclipse IDE so it provides the same debugging interface than his parent). The debug mode is only working if we are using the virtual devices. This poses a problem because the development using the Android emulator is significantly slow that it is highly recommended use a real device. Not being able to debug the Android application in real time with the device is a problem that slows down the development process.

The small disadvantages commented above are not enough to determine if Appcelerator is a good option to develop a cross platform application or not. On the other hand, there is one that definitely could make a difference for many developers. Appcelerator does not provide a graphical tool to create the user interfaces. Despite Appcelerator encourage the developers to use a MVC model to separate the user interface from the application logic, the development of the interface is not as good as it is on the native applications development. When developers are using frameworks to develop native applications (Eclipse+ADT or XCode+Storyboard) they can use visual tools to create the user interfaces (**Figure 4.16**). This made the work easier and it could be considered a big shortcoming from Appcelerator.

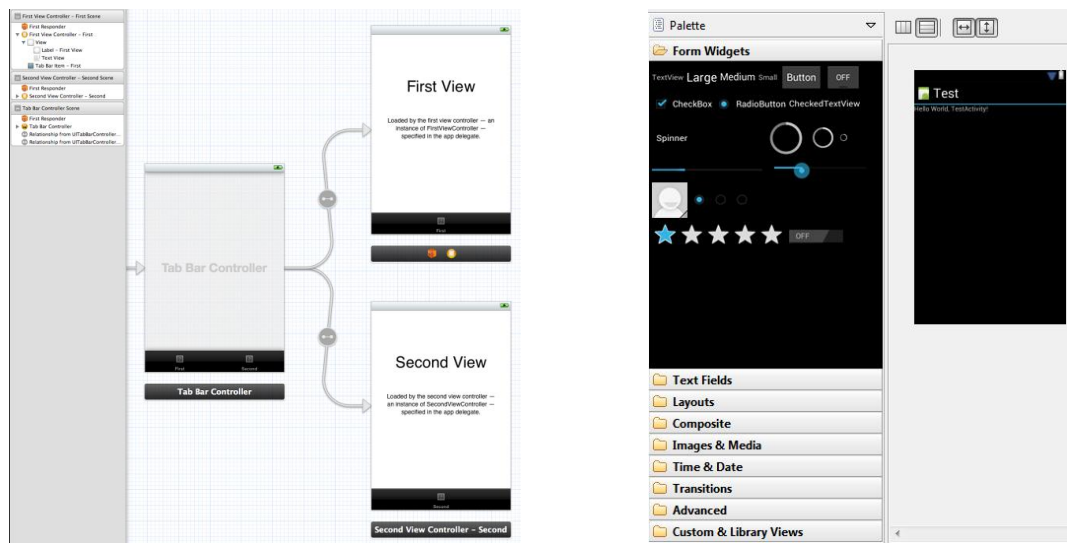


Figure 4.16 Graphical GUI development tool in XCode and Eclipse

It is very likely that any developer working with a new framework encounters a problem that she/he cannot solve. For this reason the companies that create these frameworks must provide good documentation and online tools to solve the problems of the users. In this aspect Appcelerator has a good support system on his web page. They have a good bug manager and there is a big community contributing to the “Questions and Answers” section that made easier to solve problems. On the other hand, Appcelerator has a large list of bugs unsolved that can make a developer regrets about his decision to use this framework.

Another point of view to take into account is the level of dependence between the framework and the native SDK (Android or iOS). There is a clear dependency between them but it is not affecting directly the development with this framework. When a mobile phone application is designed, the SDK versions are established and a new upgrade of the SDK made by the owner companies will not affect to the development of the application, because it is out of the scope of the project the possible modifications on the new SDK the company may do.

Once the application is developed, it is time to pack all the resources to be able to distribute it. In this phase, Appcelerator shows a clear lack of transparency and flexibility due to the fact that it does not provide the native code (Java or ObjectiveC). To modify the final solution in order to make improvements or future changes demanded by the native platform is not possible. Appcelerator packs everything into the final application file (.apk or .ipa) ready for distribution. It could be considered a shortcoming or a virtue. Some developers could consider this feature something to speed up the process of publishing the application on the online markets but others could see it is a negative aspect because the impossibility to know how the application is implemented. But definitely it is a feature to take into account before to start to develop an application with this platform.

The last aspect to take into consideration is the degree of similarity between the applications produced for Android and iOS and the level of parallel programming that has to be done in order to cover native aspects of the individual platforms. On this part Appcelerator is working pretty well. The level of parallel programming is not so high so most of the code need not be split in two to cover this aspect. During the development of the examples and the use case application just the different system to manage the windows hierarchy and navigation between Android and iOS had to be managed in parallel. The rest of the code is shared between platforms. It is true that some of the graphical elements that can be added to the user interfaces like buttons or pickers are slightly different but these differences are not so visible when we are dealing with small applications without very fancy user interfaces.

5. CONCLUSIONS

This thesis analysed the development of cross-platform mobile applications to find out the virtues and shortcomings of these kinds of applications. For this purpose the framework Titanium Appcelerator was chosen. Using Appcelerator a mobile application for Android and iOS was created. The development of this application was used to analyse the framework and the whole development process.

The general conclusions extracted from the development of this thesis and the application is that cross-platform mobile applications have many advantages comparing with the regular mobile applications but the development process depends on frameworks that are not mature enough.

There are several solutions on the market that help in creating cross-platform mobile applications. Each of them use different techniques and they lead to different results, but the common feature of all of them is that they use web technologies to create the applications.

Titanium Appcelerator is an easy to use framework that has a big community of developers. The solutions created with this framework keep a high level of similarity. Small changes in the code are needed in order to develop applications that have the same behaviour and appearance in Android and iOS devices.

The Appcelerator learning process is slow in the beginning. One may find many problems while starting to use the framework but once know how to use it, the development of a cross-platform application time is reduced.

Even though Appcelerator seem to have features that made it suitable to develop cross-platform mobile applications, it still has three important disadvantages. The first one is the stability. During the development of this thesis Appcelerator upgraded four times the Application Programming Interface (API). Some of the upgrades did not affect to the development process but others did. The next disadvantage is the development of the graphical user interfaces. The development of graphical user interfaces using native resources is made graphically. Most of the frameworks like Eclipse or XCode provide tools to simplify this process. With Appcelerator this process is made writing the code that represents those interfaces. This fact made the development slower and imprecise.

The last disadvantage is the big amount of bugs that are still open. To find one of these bugs is something that every developer prefers to avoid.

Finally, from the point of view of the developers and companies, to create cross-platform applications is something desirable because they can reach more possible buyers but nowadays is still more profitable to create different applications over the same development process rather than use any of these new technologies.

To summarize everything, here on **Table 2** the results extracted from this thesis are shown. Those results are related with the specific framework Titanium Appcelerator.

Table 2 *Titanium Appcelerator analysis results*

Interface creation	2	Poor interface creation and modification
Ease of programming	4	Programming techniques are not so difficult to learn
Learning process	3	A bit slow in the beginning but fast after that
Community support	4	Big community of developers behind

1: *Bad.*

2: *Is not so bad but it needs more improvements.*

3: *Good.*

4: *Very good but still not as good as native development.*

5: *As good as native development.*

REFERENCES

1. *Global Intelligence Alliance*. [Online] April 2010. [Cited: March 27, 2012.]
http://www.globalintelligence.com/insights-analysis/white-papers/native-or-web-application-how-best-to-deliver-cont/GIA%20Industry%20White%20Paper%202_2010_Native%20or%20Web%20App_How%20Best%20to%20De.
2. Native, Web or Hybrid MobileApp Development? *Worklight*. [Online] March 2011. [Cited: March 27, 2012.] <http://www.scribd.com/doc/50805466/Native-Web-or-Hybrid-Mobile-App-Development>.
3. *Android for developers*. [Online] [Cited: January 28, 2012.]
<http://developer.android.com/guide/topics/ui/index.html>.
4. *Cocoa Touch*. [Online] [Cited: February 16, 2012.]
<https://developer.apple.com/technologies/ios/cocoa-touch.html>.
5. *Android UI elements*. [Online] [Cited: February 16, 2012.]
<http://developer.android.com/reference/android/widget/package-summary.html>.
6. *iOS UI elements*. [Online] [Cited: February 16, 2012.]
<https://developer.apple.com/library/IOs/#documentation/UserExperience/Conceptual/MobileHIG/UIElementGuidelines/UIElementGuidelines.html>.
7. *Play audio Android*. [Online] [Cited: March 2012, 2012.]
<http://developer.android.com/guide/topics/media/mediaplayer.html>.
8. *Recording audio Android*. [Online] [Cited: March 5, 2012.]
<http://developer.android.com/guide/topics/media/audio-capture.html>.
9. *Audio iOS*. [Online] [Cited: March 5, 2012.]
https://developer.apple.com/library/ios/#documentation/AudioVideo/Conceptual/MultimediaPG/UsingAudio/UsingAudio.html#//apple_ref/doc/uid/TP40009767-CH2-SW6.
10. **Jochen H. Schiller, Agnès Voisard**. *Location-Based Services*. s.l. : Elsevier, 2004.
11. *Localization system Android*. [Online] [Cited: March 05, 2012.]
<http://developer.android.com/guide/topics/location/obtaining-user-location.html>.
12. *Localization system iOS*. [Online] [Cited: March 5, 2012.]
<http://support.apple.com/kb/HT4995>.
13. *Google Maps*. [Online] [Cited: February 2012, 2012.] <http://maps.google.com/>.

14. *Bing Maps*. [Online] [Cited: March 5, 2012.] <http://www.bing.com/maps/>.
15. *XML*. [Online] [Cited: March 5, 2012.] <http://www.w3.org/TR/REC-xml/>.
16. *SQLite*. [Online] [Cited: March 5, 2012.] <http://www.sqlite.org/>.
17. *W3C*. [Online] [Cited: March 15, 2012.] <http://www.w3.org/>.
18. *Appcelerator Titanium*. [Online] [Cited: February 16, 2012.] <http://www.appcelerator.com/>.
19. *PhoneGap framework*. [Online] [Cited: March 26, 2012.] <http://phonegap.com/>.
20. *Sencha Touch*. [Online] [Cited: March 26, 2012.] <http://www.sencha.com/>.
21. *jQuery Mobile*. [Online] [Cited: March 26, 2012.] <http://jquerymobile.com/>.
22. *Javascript*. [Online] [Cited: February 16, 2012.] https://developer.mozilla.org/en/About_JavaScript.
23. *Appcelerator Studio*. [Online] [Cited: February 18, 2012.] <http://www.appcelerator.com/products/titanium-studio/>.
24. *Eclipse IDE*. [Online] [Cited: February 18, 2012.] <http://www.eclipse.org>.
25. *TiApp.xml file*. [Online] [Cited: February 21, 2012.] <https://wiki.appcelerator.org/display/guides/tiapp.xml+and+timodule.xml+Reference>.
26. *Appcelerator Titanium Mobile 1.8.1 API*. [Online] [Cited: February 21, 2012.] <http://developer.appcelerator.com/apidoc/mobile/>.
27. *Android APK*. [Online] [Cited: February 20, 2012.] <http://developer.android.com/guide/appendix/glossary.html>.
28. *iOS App Store Package*. [Online] [Cited: February 21, 2012.] https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/ApplicationDevelopment/RM_DevelopingForAppStore/DevelopingForAppStore/DevelopingForAppStore.html.
29. *Github of the Appcelerator Titanium project*. [Online] [Cited: February 16, 2012.] https://github.com/appcelerator/titanium_mobile.
30. *CommonJS*. [Online] [Cited: March 28, 2012.] <http://www.commonjs.org/>.
31. *Rhino*. [Online] [Cited: February 20, 2012.] <http://www.mozilla.org/rhino/>.
32. *Android developers*. [Online] [Cited: February 21, 2012.] <http://developer.android.com/index.html>.
33. *Inlined process*. [Online] [Cited: February 20, 2012.] <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>.
34. *WebKit's JavaScriptCore*. [Online] [Cited: February 20, 2012.] <http://www.webkit.org/projects/javascript/index.html>.

35. *V8 JavaScript interpreter*. [Online] [Cited: February 22, 2012.]
<http://code.google.com/p/v8/>.
36. *Comparison between V8 and Rhino*. [Online] [Cited: February 22, 2012.]
<https://wiki.appcelerator.org/display/guides/V8+Performance+in+1.8.0.1>.
37. Developers Guide: Obtain a suitable private key. *Android developers*. [Online]
 [Cited: April 8, 2012.] <http://developer.android.com/guide/publishing/app-signing.html#cert>.
38. *Navigation group*. [Online] [Cited: February 22, 2012.]
<http://developer.appcelerator.com/apidoc/mobile/latest/Titanium.UI.iPhone.NavigationGroup-object.html>.
39. *Android Manifest permissions*. [Online] [Cited: March 12, 2012.]
<http://developer.android.com/reference/android/Manifest.permission.html>.
40. *Current Position pin bug*. [Online] [Cited: February 29, 2012.]
<https://jira.appcelerator.org/browse/TIMOB-7667>.
41. *Bug on Annotation pin color*. [Online] [Cited: March 1, 2012.]
<https://jira.appcelerator.org/browse/TIMOB-4453>.
42. *Codeboxed AudioRecorder*. [Online] [Cited: March 19, 2012.]
<http://www.codeboxed.com/2011/08/titanium-module-for-android-audio-recording/>.
43. *Android sandbox*. [Online] [Cited: March 21, 2012.]
<http://developer.android.com/guide/topics/security/security.html>.
44. *iOS Sandbox*. [Online] [Cited: March 21, 2012.]
<http://developer.apple.com/library/ios/#DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheiOSEnvironment/TheiOSEnvironment.html>.
45. *JavaScript DOM level 2*. [Online] [Cited: March 21, 2012.]
<http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#i-Document>.
46. **C.J.ROGERS**. *Music source used in Audio example*. [Online] [Cited: March 13, 2012.] <http://www.jamendo.com/en/track/910075>.
47. *Audio source used in Audio streaming example*. [Online] [Cited: March 13, 2012.]
<http://www.w3schools.com/html5/song.mp3>.
48. *Codeboxed Audio recorder module*. [Online] [Cited: November 14, 2011.]
<http://www.codeboxed.com/2011/08/titanium-module-for-android-audio-recording/>.

A. USE CASE IMPLEMENTATION

tiapp.xml

```
...
<android xmlns:android="http://schemas.android.com/apk/res/android">
  <manifest>
    <application/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.INTERNET"/>
  </manifest>
</android>
...
```

app.js

```
// Storage Manager
var storageManager = require('lib/StorageManager');
// List of guides
var guides_;
setUp();
var lastGuideID = getLastGuideId();
openGUI();
// Save the guide into the file system
Ti.App.addEventListener('saveGuideAndStore', function(guide) {
  storageManager.saveGuide(guide)
  if(!guide.saved){
    guide.listIndex = guides_.length;
    guides_.push(guide);
  }
  guide.saved = true;
});
// Delete a guide
Ti.App.addEventListener('deleteGuide', function(guideAux) {
  var guide = findGuide(guideAux.id, guides_);
  storageManager.removeGuide(guide.id);
  guides_.splice(guide.listIndex, 1);
});
// Create a temporary folder for a place
Ti.App.addEventListener('createPlaceAux', function(place) {
  storageManager.createPLACE_AUX(place);
});
// Save a picture
Ti.App.addEventListener('savePicture', function(request) {
  storageManager.savePicture(request);
});
// Open the Graphical user interface
function openGUI(){
//require and open top level UI component
var Window;
if (Ti.Platform.osname === 'android') {
  Window = require('ui/handheld/android/MainWindow');
}
else {
  Window = require('ui/handheld/ios/MainWindow');
}
new Window(guides_, lastGuideID).open();
}
// Configure the application in the beginning
function setUp(){
```

```

    // 1. Create MY_GUIDES directory in APP directory.
    storageManager.createMY_GUIDES();
    // 2. Read guides and places
    guides_ = storageManager.getGuides();
    // 3. Create PLACES directory in EXTERNAL STORAGE
    storageManager.createPLACES_AUX();
}
// Obtain the id of the last guide created
function getLastGuideId(){
    if(guides_.length == 0){
        return 0;
    }
    else {
        return guides_[guides_.length - 1].id;
    }
}
// Look for a Guide object within a list of guides
function findGuide(guideId, guides){
    var i = 0;
    var found = false;
    while( i < guides.length && !found){
        if(guides[i].id == guideId){
            found = true;
            return guides[i];
        }
        i++;
    }
    return null;
}

```

iOS/MainWindow.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * MAIN WINDOW / iOS
 *
 * Handler that manage the behavior of the MainWindow
 */
/*
 * Constructor of the module
 *
 * @param guides List of guides
 * @param lastGuideID Id of the last guide created
 * @return Window created
 */
function MainWindow(guides, lastGuideID) {
    //declare module dependencies
    var MainWindowView = require('ui/common/MainWindowView');
    //create object instance
    var self = Ti.UI.createWindow({
        backgroundColor: '#ffffff'
    });
    //construct UI
    var mainWindowView = new MainWindowView();
    //create master view container
    var mainWindowContainerWindow = Ti.UI.createWindow({
        title: 'Travel Guides'
    });
    mainWindowContainerWindow.add(mainWindowView);
    // Create sub-windows dependencies
    var listGuidesContainerWindow = require('ui/handheld/iOS/ListGuides');
    var newGuideContainerWindow = require('ui/handheld/iOS/NewGuide');
    //createiOS specific NavGroup UI

```

```

var navGroup = Ti.UI.iPhone.createNavigationGroup({
    window:mainWindowContainerWindow
});
self.add(navGroup);
// Map View
var Map = require('lib/Map');
//construct UI
var mapView = new Map();
// Open the window to create a new guide
mainWindowView.addEventListener('showNewGuide', function(e) {
    // Create the Guide Object needed.
    var Guide = require('lib/Guide');
    var guide = new Guide(false,++lastGuideID,"","",[],[],0);
    // Open the window
    navGroup.open(newGuideContainerWindow(navGroup,mapView,guide));
});
// Open the window to show the guides created
mainWindowView.addEventListener('showListGuides', function(e) {
    // Open the window
    navGroup.open(listGuidesContainerWindow(navGroup,mapView,guides));
});
return self;
};
module.exports = MainWindow;

```

android/MainWindow.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * MAIN WINDOW / ANDROID
 * Handler that manage the behavior of the MainWindow
 */
/*
 * Constructor of the module
 *
 * @param guides List of guides
 * @param lastGuideID Id of the last guide created
 * @return Window created
 */
function MainWindow(guides,lastGuideID) {
    //declare module dependencies
    var MainWindowView = require('ui/common/MainWindowView');
    //create object instance
    var mainWindowContainerWindow = Ti.UI.createWindow({
        title:'Travel Guides',
        exitOnClose:true,
        navBarHidden:false,
        backgroundColor:'#ffffff'
    });
    //construct UI
    var mainWindowView = new MainWindowView();
    mainWindowContainerWindow.add(mainWindowView);
    // Create sub-windows dependencies
    var listGuidesContainerWindow = require('ui/handheld/android/ListGuides'),
        newGuideContainerWindow = require('ui/handheld/android/NewGuide');
    // Map View
    var Map = require('lib/Map');
    //construct UI
    var mapView = new Map();
    // Open the window to create a new guide
    mainWindowView.addEventListener('showNewGuide', function(e) {
        // Create the Guide Object needed.
        var Guide = require('lib/Guide');

```



```

        var id = ++lastGuideID + "";
        var guide = new Guide(false,id,"","","",[],[],0);
        // Open the window
        newGuideContainerWindow(mapView,guide).open();
    });
    // Open the window to show the guides created
    mainWindowView.addEventListener('showListGuides', function(e) {
        // Open the window
        listGuidesContainerWindow(mapView,guides).open();
    });
    // Close the application
    mainWindowContainerWindow.addEventListener('close',function(){
        Ti.App.fireEvent('closingApp');
    });
    return mainWindowContainerWindow;
};
module.exports = MainWindow;

```

MainWindowView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * MAIN WINDOW VIEW
 * View with the main components of the Main Window
 */

/*
 * Constructor of the module
 */
function MainWindowView() {
    var self = Ti.UI.createView({
        backgroundColor:'white'
    });
    // Button to create a new guide
    var buttonNewGuide = Ti.UI.createButton({
        height:'25%',
        width:'90%',
        color:'black',
        title:'New Guide',
        top:'20%'
    });
    self.add(buttonNewGuide);
    //add behavior
    buttonNewGuide.addEventListener('click', function(e) {
        // Fire the event to the Main Window Manager (ui/handheld/XX/MainWindow.js)
        self.fireEvent('showNewGuide');
    });
    // Button to show the list of guides
    var buttonListGuides = Ti.UI.createButton({
        height:'25%',
        width:'90%',
        color:'black',
        title:'My Guides',
        bottom:'20%'
    });
    self.add(buttonListGuides);
    //add behavior
    buttonListGuides.addEventListener('click', function(e) {
        // Fire the event to the Main Window Manager (ui/handheld/XX/MainWindow.js)
        self.fireEvent('showListGuides');
    });
    return self;
};
module.exports = MainWindowView;

```

iOS/NewGuide.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW GUIDE / iOS
 *
 * Handler that manage the behavior of the NewGuide window
 */
/*
 * Constructor of the module
 *
 * @param navGroup Navigation group
 * @param mapView View of the map
 * @param guide Guide Object to be created
 * @return Window created
 */
function NewGuide(navGroup,mapView,guide) {
    //declare module dependencies
    var NewGuideView = require('ui/common/NewGuide/NewGuideView');
    //construct UI
    var newGuideView = new NewGuideView(mapView,guide);
    //create master view container
    var newGuideContainerWindow = Ti.UI.createWindow({
    });
    newGuideContainerWindow.add(newGuideView);
    if(guide.saved){
        newGuideContainerWindow.title = "Guide: " + guide.name;
    }
    else{
        newGuideContainerWindow.title = "New Guide";
    }
    // Dependencies of the sub-window
    var listPlacesContainerWindow = require('ui/handheld/iOS/ListPlaces');
    // Create a new place
    newGuideView.addEventListener('newPlace', function(e) {
        // Create the Place Object to be used
        var Place = require('lib/Place');
        var newId = ++guide.lastPlaceId;
        var mediaDirectories = require('lib/StorageManager').getMediaDirectories(newId,
guide.id);
        var place = new Place(false,false,false,newId,"","",0,0,mediaDirectories);
        //Create the Temporary place folder
        Ti.App.fireEvent('createPlaceAux',place);
        // Create the window that will show the menu to create a new place
        var NewPlaceView = require('ui/common/NewPlace/NewPlaceView');
        var newPlaceView = new NewPlaceView(place);

        var newPlaceContainerWindow = Ti.UI.createWindow({
            title:'New Place',
            exitOnClose:false,
            navBarHidden:false,
            backgroundColor:'#ffffff'
        });
        newPlaceContainerWindow.add(newPlaceView);
        // Cancel the place
        newPlaceView.addEventListener('cancelPlace',function(){
            navGroup.close(newPlaceContainerWindow);
        });
        // Save the place
        newPlaceView.addEventListener('savePlace',function(){
            // Take the cordinates
            place.latitude = mapView.myLatitude;

```

```

        place.longitude = mapView.myLongitude;
        // If the place is NOT already on the list
        if(!place.inList) {
            // Add it to the temporary places list
            guide.placesAux.push(place);
            place.inList = true;
            // Create the annotation and add it to the map view
            var point = Ti.Map.createAnnotation({
                latitude: place.latitude,
                longitude: place.longitude,
                title: place.name,
                pincolor: Titanium.Map.ANNOTATION_GREEN,
                subtitle: place.desc,
                animate:true,
                myid:place.id,
            });
            mapView.addAnnotation(point);
            place.annotation = point;
        }
        // Close the window
        navGroup.close(newPlaceContainerWindow);
    });
    // Open the window
    navGroup.open(newPlaceContainerWindow);
});
// Open the list of places
newGuideView.addEventListener('listPlaces', function(e) {
    navGroup.open(listPlacesContainerWindow(navGroup,guide.placesAux,mapView));
});
// Cancel the guide
newGuideView.addEventListener('cancelGuide',function(e){
    navGroup.close(newGuideContainerWindow);
});
// Save the guide
newGuideView.addEventListener('saveGuide',function(e){
    // Extract the basic information
    var name = newGuideView.infoView.inputName.getValue();
    var desc = newGuideView.infoView.inputDescription.getValue();
    var city = newGuideView.infoView.listPicker.getSelectedRow(0).id;
    guide.name = name;
    guide.desc = desc;
    guide.city = city;
    // Copy the auxiliar list of places to the real list of places
    guide.places = guide.placesAux;
    // Store the guide into the device file system
    Ti.App.fireEvent('saveGuideAndStore',guide);
    // Close the window
    navGroup.close(newGuideContainerWindow);
});
return newGuideContainerWindow;
};
module.exports = NewGuide;

```

android /NewGuide.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW GUIDE / ANDROID
 * Handler that manage the behavior of the NewGuide window
 */
/*
 * Constructor of the module
 *
 * @param mapView View of the map
 * @param guide Guide Object to be created *
 * @return Window created
 */
function NewGuide(mapView,guide) {
  //declare module dependencies
  var NewGuideView = require('ui/common/NewGuide/NewGuideView');
  // construct the UI
  var newGuideView = new NewGuideView(mapView,guide);
  //create object instance
  var newGuideContainerWindow = Ti.UI.createWindow({
    exitOnClose:false,
    navBarHidden:false,
    backgroundColor:'#ffffff'
  });
  newGuideContainerWindow.add(newGuideView);
  if(guide.saved){
    newGuideContainerWindow.title = "Guide: " + guide.name;
  }
  else{
    newGuideContainerWindow.title = "New Guide";
  }
  // Dependencies of the sub-window
  var listPlacesContainerWindow = require('ui/handheld/android/ListPlaces');

  // Create a new place
  newGuideView.addEventListener('newPlace', function(e) {
    // Create the Place Object to be used
    var Place = require('lib/Place');
    var newId = ++guide.lastPlaceId;
    var mediaDirectories = require('lib/StorageManager').getMediaDirectories(newId,
guide.id);
    var place = new Place(false,false,false,newId,"","",0,0,mediaDirectories);
    //Create the Temporary place folder
    Ti.App.fireEvent('createPlaceAux',place);
    var NewPlaceView= require('ui/common/NewPlace/NewPlaceView');
    var newPlaceView = new NewPlaceView(place);
    // Create the window that will show the menu to create a new place
    var newPlaceContainerWindow = Ti.UI.createWindow({
      title:'New Place',
      exitOnClose:false,
      navBarHidden:false,
      backgroundColor:'#ffffff'
    });
    newPlaceContainerWindow.add(newPlaceView);
    // Cancel the place

    newPlaceView.addEventListener('cancelPlace',function(e){
      newPlaceContainerWindow.close();
    });
    // Save the place
    newPlaceView.addEventListener('savePlace',function(e){
      // Take the coordinates
      place.latitude = mapView.myLatitude;

```

```

        place.longitude = mapView.myLongitude;
        // If the place is NOT already on the list
        if(!place.inList) {
            // Add it to the temporary list
            guide.placesAux.push(place);
            place.inList = true;
            //Create the annotation and add it to the map view
            var point = Ti.Map.createAnnotation({
                latitude: place.latitude,
                longitude: place.longitude,
                title: place.name,
                pincolor: Titanium.Map.ANNOTATION_GREEN,
                subtitle: place.desc,
                animate:true,
                myid:place.id,
            });
            mapView.addAnnotation(point);
            place.annotation = point;
        }
        // Close the window
        newPlaceContainerWindow.close();
    });
    // Open the window
    newPlaceContainerWindow.open();
});
// Open the list of places
newGuideView.addEventListener('listPlaces', function(e) {
    listPlacesContainerWindow(guide.placesAux,mapView).open();
});
// cancel the guide
newGuideView.addEventListener('cancelGuide', function(e) {
    newGuideContainerWindow.close();
});
// Save the guide
newGuideView.addEventListener('saveGuide',function(e){
    //Extract the basic information
    var name = newGuideView.infoView.inputName.value;
    var desc = newGuideView.infoView.inputDescription.value;
    var city = newGuideView.infoView.listPicker.getSelectedRow(0).id;
    guide.name = name;
    guide.desc = desc;
    guide.city = city;
    // copy the auxiliar list of places to the real list of places
    guide.places = guide.placesAux;
    // Store the guide into the device file system
    Ti.App.fireEvent('saveGuideAndStore',guide);
    // Close the window
    newGuideContainerWindow.close();
});
return newGuideContainerWindow;
};
module.exports = NewGuide;

```

NewGuideView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW GUIDE VIEW
 * View with the components needed to create a new guide

```

```

*/
/*
 * Constructor of the module
 */
function NewGuideView(mapView,guide) {
    // Main view
    var self = Ti.UI.createView({
        backgroundColor:'white'
    });
    // Views for the Info and Places menus
    var newGuideInfoView = createInfoView(self,guide);
    var newGuidePlacesView = createPlacesView(self,mapView,guide.placesAux);
    // Listening the actions to navigate on the New Guide View
    self.addEventListener('goPlaces',function(e){
        newGuideInfoView.setVisible(false);
        newGuidePlacesView.setVisible(true);
    });
    // Listening the actions to navigate on the New Guide View
    self.addEventListener('goInfo',function(e){
        newGuidePlacesView.setVisible(false);
        newGuideInfoView.setVisible(true);
    });
    return self;
};
/*
 * Create the view for the info menu
 *
 * @param self Parent view
 */
function createInfoView(self,guide){
    // Info view
    var infoView = Ti.UI.createView({
        backgroundColor:'white',
        visible: false,
        width: '100%',
    });
    // View to center the form on the screen
    var centeredView = Ti.UI.createView({
        width: '100%',
        height: 400,
    });
    // Input for the name of the guide
    var inputName = Ti.UI.createTextField({
        width: '80%',
        height: 40,
        top: 0,
        editable: true,
        borderStyle: Ti.UI.INPUT_BORDERSTYLE_ROUNDED,
        hintText: 'Name of the guide',
        value: guide.name,
    });
    centeredView.add(inputName);
    // Input for the description of the guide
    var inputDescription = Ti.UI.createTextArea({
        width: '80%',
        height: 90,
        top: 45,
        editable: true,
        borderWidth:1,
        borderColor: '#bbb',
        borderRadius: 4,
        value: guide.desc,
    });
    centeredView.add(inputDescription);
};

```

```

// List picker to choose the city of the guide
var listPicker = Ti.UI.createPicker({
    top:140,
    width: '80%',
    font: {fontSize:10},
    selectionIndicator: true,
});
// Data for the list picker.
// The attribute "id" has to be the same than the position within the array.
// This attribute is used to translate numbers to city names.
var data = [];
data[0] = Ti.UI.createPickerRow({title: 'Tampere', id: 0});
data[1] = Ti.UI.createPickerRow({title: 'Helsinki', id: 1});
data[2] = Ti.UI.createPickerRow({title: 'Turku', id: 2});
listPicker.add(data);
centeredView.add(listPicker);
// Select the corresponding city when a created guide is picked from the list
Ti.App.addEventListener('guideWindowOpened',function(){
    listPicker.setSelectedRow(0,parseInt(guide.city),true);
});
// Add the inputs to the view to extract the information once we want to save the
guide.
infoView.inputName = inputName;
infoView.inputDescription = inputDescription;
infoView.listPicker = listPicker;
infoView.add(centeredView);
// Add the info and places view to the parent view
self.add(infoView);
self.infoView = infoView;
infoView.setVisible(true);
// Button to cancel the guide
var buttonCancel = Ti.UI.createButton({
    height:44,
    width:'35%',
    title:'Cancel',
    bottom:5,
    left: '5%',
    color:'black',
});
infoView.add(buttonCancel);
//add behavior
buttonCancel.addEventListener('click', function(e) {
    // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
    self.fireEvent('cancelGuide');
});
// Button to show the next view: the Places View
var buttonNext = Ti.UI.createButton({
    height:44,
    width:'35%',
    title:'Next',
    bottom:5,
    right: '5%',
    color:'black',
});
infoView.add(buttonNext);

//add behavior
buttonNext.addEventListener('click', function(e) {
    // Fire the event to the New Guide View (ui/common/NewGuide/NewGuideView.js)
    self.fireEvent('goPlaces');
});
return infoView;
}
/*

```

```

* Create the view for the places menu
*
* @param self Parent view
* @param mapView Map of the application
* @param places List of places of the guide
*/
function createPlacesView(self,mapView,places){
  // Places view
  var placesView = Ti.UI.createView({
    backgroundColor:'white',
    visible: false,
  });
  self.add(placesView);
  // Map view
  mapView.setTop(10);
  mapView.width = '90%';
  mapView.height = '60%';
  placesView.add(mapView);
  // Add the annotations to the map.
  for(var p = 0; p < places.length; p++){
    var point = Ti.Map.createAnnotation({
      latitude: places[p].latitude,
      longitude: places[p].longitude,
      title: places[p].name,
      pincolor: Titanium.Map.ANNOTATION_GREEN,
      subtitle: places[p].desc,
      animate:true,
      myid:places[p].id,
    });
    mapView.addAnnotation(point);
    // Link the annotation to the Place object to be able to delete the annotation
    later
    places[p].annotation = point;
  }
  // Button to create a new place
  var buttonNewPlace = Ti.UI.createButton({
    height:70,
    width:'40%',
    title:'New Place',
    top:'65%',
    left: '5%',
    color:'black',
  });
  placesView.add(buttonNewPlace);
  //add behavior
  buttonNewPlace.addEventListener('click', function(e) {
    // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
    self.fireEvent('newPlace');
  });
  // Button to show the list of places
  var buttonListPlaces = Ti.UI.createButton({
    height:70,
    width:'40%',
    title:'List of places',
    top:'65%',
    right: '5%',
    color:'black',
  });
  placesView.add(buttonListPlaces);
  //add behavior
  buttonListPlaces.addEventListener('click', function(e) {
    // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
    self.fireEvent('listPlaces');
  });
}

```



```

// Button to go back to the info view
var buttonBack = Ti.UI.createButton({
    height:44,
    width:'35%',
    title:'Info',
    bottom:5,
    left: '5%',
    color:'black',
});
placesView.add(buttonBack);
//add behavior
buttonBack.addEventListener('click', function(e) {
    // Fire the event to the New Guide View (ui/common/NewGuide/NewGuideView.js)
    self.fireEvent('goInfo');
});
// Button to save the guide
var buttonSave = Ti.UI.createButton({
    height:44,
    width:'35%',
    title:'Save',
    bottom:5,
    right: '5%',
    color:'black',
});
placesView.add(buttonSave);
//add behavior
buttonSave.addEventListener('click', function(e) {
    if(self.infoView.inputName.value != ""){
        // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
        self.fireEvent('saveGuide');
    }
    else {
        alert('The guide should have a name');
    }
});
return placesView;
}
module.exports = NewGuideView;

```

iOS/ListGuides.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST GUIDES / iOS
 *
 * Handler that manage the behavior of the Window that lists the guides created
 */
/*
 * Constructor of the module
 *
 * @param navGroup Navigation group
 * @param mapView View of the map
 * @param guides List of Guide objects
 * @return It returns the window object
 */
function ListGuides(navGroup,mapView,guides) {
    //declare module dependencies
    var ListGuidesView = require('ui/common/ListGuidesView');

    //construct UI
    var listGuidesView = new ListGuidesView(guides);
    //create master view container
    var listGuidesContainerWindow = Ti.UI.createWindow({

```

```

        title:'List of Guides'
    });
    listGuidesContainerWindow.add(listGuidesView);
    //declare sub-window dependencies
    var newGuideContainerWindow = require('ui/handheld/iOS/NewGuide');
    // Open a Guide
    listGuidesView.addEventListener('openGuide', function(guideAux) {
        // Obtain the real Guide object. the guideAux object is a JSON object that con-
        tains the information of the guide but is not the real object
        var guide = findGuide(guideAux.id,guides);
        // Open the window with the guide selected
        navGroup.open(newGuideContainerWindow(navGroup,mapView,guide));
        Ti.App.fireEvent('guideWindowOpened');
    });
    return listGuidesContainerWindow;
};
/*
 * Look for a Guide object within a list of guides
 *
 * @param guideId Id of the guide object to look for
 * @param guides List of guides
 */
function findGuide(guideId,guides){
    var i = 0;
    var found = false;
    while( i < guides.length && !found){
        if(guides[i].id == guideId){
            found = true;
            return guides[i];
        }
        i++;
    }
    return null;
}
module.exports = ListGuides;

```

android /ListGuides.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST GUIDES / Android
 * Handler that manage the behavior of the Window that lists the guides created
 */
/*
 * Constructor of the module
 *
 * @param mapView View of the map
 * @param guides List of Guide objects *
 * @return It returns the window object
 */
function ListGuides(mapView,guides) {
    //declare module dependencies
    var ListGuidesView = require('ui/common/ListGuidesView');
    //construct UI
    var listGuidesView = new ListGuidesView(guides);
    //create master view container
    var listGuidesContainerWindow = Ti.UI.createWindow({
        title:'List of Guides',
        exitOnClose:false,
        navBarHidden:false,
        backgroundColor:'#ffffff'
    });
    listGuidesContainerWindow.add(listGuidesView);
}

```

```

//declare sub-window dependencies
var newGuideContainerWindow = require('ui/handheld/android/NewGuide');
// Open a Guide
listGuidesView.addEventListener('openGuide', function(guideAux) {
    // Obtain the real Guide object. the guideAux object is a JSON object that contains the information of the guide but is not the real object
    var guide = findGuide(guideAux.id, guides);
    // Open the window with the guide selected
    newGuideContainerWindow(mapView, guide).open();
    Ti.App.fireEvent('guideWindowOpened');
});
return listGuidesContainerWindow;
};
/*
 * Look for a Guide object within a list of guides
 */
 * @param guideId Id of the guide object to look for
 * @param guides List of guides
 */
function findGuide(guideId, guides){
    var i = 0;
    var found = false;
    while( i < guides.length && !found){
        if(guides[i].id == guideId){
            found = true;
            return guides[i];
        }
        i++;
    }
    return null;
}
module.exports = ListGuides;

```

ListGuidesView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST GUIDES VIEW
 * View with the list of guides
 */

/*
 * Constructor of the module
 * @param guideList array with the guide objects
 */
function ListGuidesView(guidesList) {
    // Main view

    var self = Ti.UI.createView({
        backgroundColor:'white'
    });
    // TableView with the list of created guides
    var listGuidesTable = Ti.UI.createTableView({
        top:10,
        width:'100%',
        height:'100%',
    });
    self.add(listGuidesTable);
    // Add the guides to the table
    for(var g = 0; g < guidesList.length; g++){
        // Create row on the table for this guide
        var row = Ti.UI.createTableViewRow({
            guide: guidesList[g],

```

```

        color:'black',
        height: 50,
    });
    // Create a label to show the name.
    // This has to be done instead to use the attribute "title" because in Android,
when you add another element to the TableViewRow (in our case, a button),
    // the title is not shown.
    var titleLabel = Ti.UI.createLabel({
        text: guidesList[g].name,
        left: 5,
        width: '80%',
        color:'black',
    });
    row.add(titleLabel);
    // Button to delete the place
    var buttonDelete = Ti.UI.createButton({
        right:10,
        height:30,
        width:30,
        title: 'X',
    });
    row.add(buttonDelete);
    // Link the button to the row to be able to manage the events of the row.
    row.deleteButton = buttonDelete;
    // Add behavior
    row.addEventListener('click',function(roww){
        // If we do click on the delete button
        if(roww.source === roww.row.deleteButton){
            // Fire the event to the List Places Manager (ui/handheld/XX/ListPlaces.js)
            Ti.App.fireEvent('deleteGuide',roww.row.guide);
            listGuidesTable.deleteRow(roww.index);
        }
        // If we do click in other part of the row, open the place
        else {
            // Fire the event to the Main Window Manager (ui/handheld/XX/MainWindow.js)
            self.fireEvent('openGuide',roww.row.guide);
        }
    });
    listGuidesTable.appendRow(row);
}
// // Add behavior
// listGuidesTable.addEventListener('click',function(table){
// // // Fire the event to the Main Window Manager (ui/handheld/XX/MainWindow.js)
// // self.fireEvent('openGuide',table.row.guide);
// // });
return self;
};
module.exports = ListGuidesView;

```

iOS/ListPlaces.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST PLACES / iOS
 * Handler that manage the behavior of the Window that lists the places created
 */
/*
 * Constructor of the module
 *
 * @param navGroup Navigation group
 * @param places list of places
 * @param mapView View of the map *
 * @return It returns the window object

```

```

*/
function ListPlaces(navGroup,places,mapView) {
    //declare module dependencies
    var ListPlacesView = require('ui/common/NewGuide/ListPlacesView');
    //construct UI
    var listPlacesView = new ListPlacesView(places);
    //create master view container
    var listPlacesContainerWindow = Ti.UI.createWindow({
        title:'List of Places'
    });
    listPlacesContainerWindow.add(listPlacesView);
    // Open a place
    listPlacesView.addEventListener('openPlace', function(placeAux) {
        // Create the Place View
        var NewPlaceView = require('ui/common/NewPlace/NewPlaceView');
        var place = findPlace(placeAux.id,places);
        var newPlaceView = new NewPlaceView(place);
        var newPlaceContainerWindow = Ti.UI.createWindow({
            title:'Place: ' + place.name,
        });
        newPlaceContainerWindow.add(newPlaceView);
        // Cancel a Palce
        newPlaceView.addEventListener('cancelPlace',function(e){
            navGroup.close(newPlaceContainerWindow);
        });
        // Save a place
        newPlaceContainerWindow.addEventListener('savePlace',function(){
            // If the place is NOT in the list
            if(!place.inList) {
                places.push(place);
                place.inList = true;
                var storageManager = require('lib/StorageManager');
                place.mediaDirectory = storageManager.getPictureName(place);
            }
            else{
                listPlacesView.fireEvent('refreshPlace',place);
            }
            navGroup.close(newPlaceContainerWindow);
        });
        navGroup.open(newPlaceContainerWindow);
    });
    // Delete the place
    listPlacesView.addEventListener('deletePlace',function(placeAux){
        // Obtain the real Place object. the placeAux object is a JSON object that contains
        // the information of the place but is not the real object
        var place = findPlace(placeAux.id,places);
        place.deleted = true;

        // Remove the annotation from the map
        mapView.removeAnnotation(place.annotation);
    });
    return listPlacesContainerWindow;
};
/*
 * Look for a Place object within a list of guides
 *
 * @param placeId Id of the place object to look for
 * @param places List of places
 */
function findPlace(placeId,places){
    var i = 0;
    var found = false;
    while( i < places.length && !found){
        if(places[i].id == placeId){

```

```

        found = true;
        return places[i];
    }
    i++;
}
return null;
}
module.exports = ListPlaces;

```

android /ListPlaces.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST PLACES / ANDROID
 * Handler that manage the behavior of the Window that lists the places created
 */
/*
 * Constructor of the module
 *
 * @param places List of places
 * @param mapView View of the map *
 * @return It returns the window object
 */
function ListPlaces(places,mapView) {
    //declare module dependencies
    var ListPlacesView = require('ui/common/NewGuide/ListPlacesView');
    //construct UI
    var listPlacesView = new ListPlacesView(places);
    //create master view container
    var listPlacesContainerWindow = Ti.UI.createWindow({
        title:'List of Places',
        exitOnClose:false,
        navBarHidden:false,
        backgroundColor:'#ffffff'
    });
    listPlacesContainerWindow.add(listPlacesView);
    // Open a place
    listPlacesView.addEventListener('openPlace', function(placeAux) {
        // Create the place view
        var NewPlaceView = require('ui/common/NewPlace/NewPlaceView');
        // Obtain the real Place object. the placeAux object is a JSON object that contains
        // the information of the place but is not the real object
        var place = findPlace(placeAux.id,places);
        var newPlaceView = new NewPlaceView(place);

        //create detail view container
        var newPlaceContainerWindow = Ti.UI.createWindow({
            title:'Place: ' + place.name,
            exitOnClose:false,
            navBarHidden:false,
            backgroundColor:'#ffffff'
        });
        newPlaceContainerWindow.add(newPlaceView);
        // Cancel a place
        newPlaceView.addEventListener('cancelPlace',function(e){
            newPlaceContainerWindow.close();
        });
        // Save a place
        newPlaceView.addEventListener('savePlace',function(){
            // If the place is NOT in the list
            if(!place.inList) {
                places.push(place);
                place.inList = true;
            }
        });
    });
}

```

```

        var storageManager = require('lib/StorageManager');
        place.mediaDirectory = storageManager.getPictureName(place);
    }
    else{
        listPlacesView.fireEvent('refreshPlace',place);
    }
    newPlaceContainerWindow.close();
});
newPlaceContainerWindow.open();
});
// Delete the place
listPlacesView.addEventListener('deletePlace',function(placeAux){
    // Obtain the real Place object. the placeAux object is a JSON object that contains
    // the information of the place but is not the real object
    var place = findPlace(placeAux.id,places);
    place.deleted = true;
    // Remove the annotation from the map
    mapView.removeAnnotation(place.annotation);
});
return listPlacesContainerWindow;
};
/*
 * Look for a Place object within a list of guides
 *
 * @param placeId Id of the place object to look for
 * @param places List of places
 */
function findPlace(placeId,places){
    var i = 0;
    var found = false;
    while( i < places.length && !found){
        if(places[i].id == placeId){
            found = true;
            return places[i];
        }
        i++;
    }
    return null;
}
module.exports = ListPlaces;

```

ListPlacesView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * LIST OF PLACES VIEW
 * View with the places of the guide
 */
/*
 * Constructor of the module
 * @param placesList List of places to show
 */
function ListPlacesView(placesList) {
    // Main view
    var self = Ti.UI.createView({
        backgroundColor:'white'
    });
    // TableView with the places os the guide

```

```

var listPlacesTable = Ti.UI.createTableView({
    top:10,
    width:'100%',
    height:'100%',
});
self.add(listPlacesTable);
// Add the places to the list
for(var p = 0; p < placesList.length; p++){
    if(!placesList[p].deleted){
        // Create row on the table for this guide
        var row = Ti.UI.createTableViewRow({
            place: placesList[p],
            height: 50,
        });
        // Create a label to show the name.
        // This has to be done instead to use the attribute "title" because in Android,
when you add another element to the TableViewRow (in our case, a button),
        // the title is not shown.
        var titleLabel = Ti.UI.createLabel({
            text: placesList[p].name,
            left: 5,
            width: '80%',
            color:'black',
        });
        row.add(titleLabel);
        // Button to delete the place
        var buttonDelete = Ti.UI.createButton({
            right:10,
            height:30,
            width:30,
            title: 'X',
        });
        row.add(buttonDelete);
        // Link the button to the row to be able to manage the events of the row.
        row.deleteButton = buttonDelete;
        listPlacesTable.appendRow(row);
        // Add behavior
        row.addEventListener('click',function(roww){
            // If we do click on the delete button
            if(roww.source === roww.row.deleteButton){
                // Fire the event to the List Places Manager
(ui/handheld/XX/ListPlaces.js)
                self.fireEvent('deletePlace',roww.row.place);
                listPlacesTable.deleteRow(roww.index);
            }
            // If we do click in other part of the row, open the place
            else {
                // Fire the event to the List Places Manager
(ui/handheld/XX/ListPlaces.js)
                self.fireEvent('openPlace',roww.row.place);
            }
        });
    }
}
// Actualise the information of the table in case any of the places has been modified
self.addEventListener('refreshPlace',function(place){
    var rows = listPlacesTable.data[0].rows;
    var i = 0;
    var found = false;
    while( i < rows.length && !found){
        if(rows[i].place.id == place.id){
            rows[i].place = place;
            rows[i].title = place.name;
            found = true;

```



```

        }
        i++;
    }
    });
    return self;
};
module.exports = ListPlacesView;

```

NewPlaceView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW PLACE VIEW
 * View with the components needed to create a new place
 */
/*
 * Constructor of the module
 */
function NewPlaceView(place) {
    // Main view
    var self = Ti.UI.createView();
    // Button to cancel the place
    var buttonCancel = Ti.UI.createButton({
        color: 'black',
        title: 'Cancel',
        bottom: '1%',
        left: '5%',
        width: '35%',
        height: 50,
    });
    self.add(buttonCancel);
    // add behavior
    buttonCancel.addEventListener('click', function(e) {
        // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
        self.fireEvent('cancelPlace');
    });
    // Button to save a place
    var buttonSave = Ti.UI.createButton({
        color: 'black',
        title: 'Save Place',
        bottom: '1%',
        right: '5%',
        width: '35%',
        height: 50,
    });
    self.add(buttonSave);
    //add behavior
    buttonSave.addEventListener('click', function(e) {
        if(infoView.inputName.value != ""){
            place.name = infoView.inputName.value;
            place.description = infoView.inputDescription.value;
            // Fire the event to the New Guide Manager (ui/handheld/XX/NewGuide.js)
            self.fireEvent('savePlace');
        }
        else {
            alert('The place should have a name');
        }
    });
    // Dependencies to be able to create the interface
    var InfoView = require('ui/common/NewPlace/InfoView'),
        PictureView = require('ui/common/NewPlace/PictureView'),
        AudioView = require('ui/common/NewPlace/AudioView');
    //construct UI

```

```

var infoView = new InfoView(place.name,place.description),
    pictureView = new PictureView(place),
    audioView = new AudioView(place);
// Scrollable view to store the 3 views needed to create a place:
// INFO + PICTURE + AUDIO
var scrollableView = Titanium.UI.createScrollView({
    views:[infoView,pictureView,audioView],
    showPagingControl:true,
    pagingControlHeight:30,
    height: '70%'
});
self.add(scrollableView);
return self;
};
module.exports = NewPlaceView;

```

InfoView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW PLACE VIEW: INFO VIEW
 * View with the components needed to add the basic information of a place
 */
/*
 * Constructor of the module
 * @param name Name of the place
 * @param description Description of the place
 */
function InfoView(name, description) {
    var self = Ti.UI.createView();
    // Label to indicate the window we are visualizing
    var title = Ti.UI.createLabel({
        top:10,
        height:'auto',
        width:'auto',
        color: 'black',
        text: '1. Basic info',
    });
    self.add(title);

    // Input to write the name of the place
    var inputName = Ti.UI.createTextField({
        width: '80%',
        height: 40,
        top: 50,
        editable: true,
        borderStyle: Ti.UI.INPUT_BORDERSTYLE_ROUNDED,
        hintText: 'Name of the place',
        value: name,
    });
    self.add(inputName);
    self.inputName = inputName;
    // Input to add a description of the place
    var inputDescription = Ti.UI.createTextArea({
        width: '80%',
        height: '40%',
        top: 100,
        editable: true,
        borderWidth:1,
        borderColor: '#bbb',
        borderRadius: 4,
        value: description,
    });
};

```

```

        self.add(inputDescription);
        self.inputDescription = inputDescription;
        return self;
    };
    module.exports = InfoView;

```

PictureView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW PLACE VIEW: PICTURE VIEW
 * View with the components needed to add a picture to the place
 */
/*
 * Constructor of the module
 * @place Place object
 */
function PictureView(place) {
    var self = Ti.UI.createView();
    // Label to indicate the window we are visualizing
    var title = Ti.UI.createLabel({
        top:10,
        height:'auto',
        width:'auto',
        color: 'black',
        text: '2. Picture',
    });
    self.add(title);
    // Image view to show the picture taken
    var imageView = Ti.UI.createImageView({
        width: '80%',
    });
    // Open Camera button
    var button = Ti.UI.createButton({
        title: 'Open camera',
        width: 100,
        height: 70,
    });
    // Define the image of the image view depending on the status of the place (if it
    saved or not)
    var picFile;
    if(place.saved){
        picFile = place.mediaDirectory + Ti.Filesystem.separator + "p" + place.id +
        ".jpg";
        button.visible = false;
    }
    else {
        picFile = place.mediaDirectoryAux + Ti.Filesystem.separator + "p" + place.id +
        ".jpg";
    }
    var file = Ti.Filesystem.getFile(picFile);
    if(file.exists()){
        imageView.image = file;
        self.add(imageView);
    }
    // Add behavior
    button.addEventListener('click', function(){
        Ti.Media.showCamera({
            saveToPhotoGallery: false,
            // function to be called when the operation finish successfully
            success: function(event){
                if(event.mediaType == Ti.Media.MEDIA_TYPE_PHOTO){
                    imageView.setImage(event.media);
                }
            }
        });
    });
}

```

```

        self.add(imageView);
        var request = {name: place.id, data: event.media};
        // Fire the event to the Main App (/app.js)
        Ti.App.fireEvent('savePicture',request);
    }
},
// function to be called when the operation is cancelled by the user
cancel: function({}),
// function to be called when the operation fails
error: function(error){
    var a = Titanium.UI.createAlertDialog({title:'Camera'});
    if(error.code == Titanium.Media.NO_CAMERA){
        a.setMessage('Please run this test on device');
    }
    else {
        a.setMessage('Unexpected error: ' + error.code);
    }
    a.show();
},
});
});
self.add(button);
return self;
};
module.exports = PictureBox;

```

AudioView.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * NEW PLACE VIEW: AUDIO VIEW
 * View with the components needed to add an audio to the place
 */
/*
 * Constructor of the module
 */
function AudioView(place) {
    var self = Ti.UI.createView();
    // Label to indicate the window we are visualizing
    var title = Ti.UI.createLabel({
        top:10,height:'auto',width:'auto', color: 'black',text: '3. Audio',});
    self.add(title);
    // // Audio recorder dependencies
    // var audioRecorder = require('lib/AudioRecorder');
    // // Button to start the recording of audio
    // var buttonRecord = Ti.UI.createButton({
    //     // width: 200, height: 80, top: 20, title: 'Record',
    // });
    // if(!place.saved) {
    //     self.add(buttonRecord);
    // }
    // // Add behavior
    // buttonRecord.addEventListener('click', function(e){
    //     // var nameFile = "a" + place.id;
    //     // var path = place.mediaDirectories[1] + Ti.Filesystem.separator + nameFile;
    //     // audioRecorder.record(path);
    // });
    // // Button to stop the recording of audio
    // var buttonStop = Ti.UI.createButton({
    //     // width: 200, height: 80, top: 120, title: 'Stop',});
    // self.add(buttonStop);
    // // Add behavior
    // buttonStop.addEventListener('click', function(e){
    //     // audioRecorder.stop();
    // });
}

```

```

// // Button to play the recording of audio
// var buttonPlay = Ti.UI.createButton({
//   width: 200, height: 80, top: 220, title: 'Play',});
// self.add(buttonPlay);
// // Add behavior
// buttonPlay.addEventListener('click', function(e){
//   audioRecorder.play();
// });
return self;
};
module.exports = AudioView;

```

StorageManager.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * STORAGE MANAGER
 * Module that manage the storage of the guides and everything related with the file
system.
 */
// PLACES AUXILIAR DIRECTORY WHERE THE PICTURES AND AUDIO FROM THE DIFFERENT PLACES WILL
BE STORAGE TEMPORARY
var PLACES_AUX_DIRECTORY = getPLACES_AUX_DIRECTORY();
var MY_GUIDES_DIRECTORY = getMY_GUIDES_DIRECTORY();
/*
 * Create the PLACES directory where the pictures and audio from the different places
created will be storage temporary
 */
exports.createPLACES_AUX = function(){
  var places_aux = Ti.Filesystem.getFile(PLACES_AUX_DIRECTORY);
  if(places_aux.exists()){
    // In the case the application crashes and the directory could not be removed
    places_aux.deleteDirectory(true);
  }
  places_aux.createDirectory();
};
/*
 * Create the PLACES directory where the pictures and audio from the different places
created will be storage temporary
 *
 * @param place Place object
 */
exports.createPLACE_AUX = function(place){
  var place_aux = Ti.Filesystem.getFile(PLACES_AUX_DIRECTORY,place.id+"");
  if(place_aux.exists()){
    // In the case the application crashes and the directory could not be removed
    place_aux.deleteDirectory(true);
  }
  place_aux.createDirectory();
};
/*
 * Create the MY_GUIDES directory where the guides created by the user will be stored
 */
exports.createMY_GUIDES = function(){
  var my_guides = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY);
  if(!my_guides.exists()){
    my_guides.createDirectory();
  }
};
/*
 * Extract the guides information stored on the device
 */

```

```

* @return array of GUIDES objects defined with a json object
*/
exports.getGuides = function(){
  // 1. Create list of guides
  var guideList = [];
  // 2. Iterate over guides folder
  var guidesFolder = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY);
  var guidesFolderContent = guidesFolder.getDirectoryListing();
  for(var i = 0; i < guidesFolderContent.length; i++){
    var guideFolderName = guidesFolderContent[i];
    // 3. Recover information from info.xml
    var infoFile = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY +
Ti.Filesystem.separator + guideFolderName,"info.xml");
    var guideInfoXMLtext = infoFile.read();
    var guideInfoXML = Ti.XML.parseString(guideInfoXMLtext+"");
    var guide = guideInfoXML.getElementsByTagName("guide");
    var guideNode = guide.item(0);
    var guideID = guideNode.childNodes.item(0).text;
    var guideName = guideNode.childNodes.item(1).text;
    var guideCity = guideNode.childNodes.item(2).text;
    var guideDescription = guideNode.childNodes.item(3).text;
    var lastPlaceId = guideNode.childNodes.item(4).text;
    // 4. Create places list
    var placesList = [];
    var placesListAux = [];
    // 5. Iterate over places folder
    var placesFolder = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY +
Ti.Filesystem.separator + guideFolderName,"PLACES");
    var placesFolderContent = placesFolder.getDirectoryListing();
    for(var j = 0; j < placesFolderContent.length; ++j){
      var placeFolderName = placesFolderContent[j];
      // 6. Recover information from info.xml
      var infoFile = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY +
Ti.Filesystem.separator + guideFolderName + Ti.Filesystem.separator + "PLACES" +
Ti.Filesystem.separator + placeFolderName,"info.xml");
      var placeInfoXMLtext = infoFile.read();
      Ti.API.info(placeInfoXMLtext);
      var placeInfoXML = Ti.XML.parseString(placeInfoXMLtext+"");
      var place = placeInfoXML.getElementsByTagName("place");
      var placeNode = place.item(0);
      var placeID = placeNode.childNodes.item(0).text;
      var placeName = placeNode.childNodes.item(1).text;
      var placeDescription = placeNode.childNodes.item(2).text;
      var placeLongitude = placeNode.childNodes.item(3).text;
      var placeLatitude = placeNode.childNodes.item(4).text;
      var placeMediaDirectories = [ MY_GUIDES_DIRECTORY + Ti.Filesystem.separator
+ guideFolderName + Ti.Filesystem.separator + "PLACES" + Ti.Filesystem.separator +
placeID , PLACES_AUX_DIRECTORY = Ti.Filesystem.applicationDataDirectory + "PLACES_AUX"
+ Ti.Filesystem.separator + placeID];
      // 7. Create the place and add it to the list
      var Place = require('lib/Place');
      var place = new
Place(true,true,false,placeID,placeName,placeDescription,placeLongitude,placeLatitude,pl
aceMediaDirectories);
      var placeAux = new
Place(true,true,false,placeID,placeName,placeDescription,placeLongitude,placeLatitude,pl
aceMediaDirectories);
      placesList.push(place);
      placesListAux.push(placeAux);
    }
    // 8. Create the Guide and add it to the list
    var Guide = require('lib/Guide');

```

```

        var guide = new
Guide(true,guideID,guideName,guideDescription,guideCity,placesList,placesListAux,lastPla
ceId);
        guideList.push(guide);
    }
    return guideList;
};
/*
 * Save a guide into the file system
 *
 * @param guide Guide object to save
 */
exports.saveGuide = function (guide){
    // 1. Create the folder for the guide within MY_GUIDES
    //    ( IF EXISTS: nothing happens )
    var newGuide = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY,guide.id);
    if(!newGuide.exists()){
        newGuide.createDirectory();
    }
    // 2. Create the XML info file within the new GUIDE folder
    //    ( IF EXISTS: overwrite info.xml in case a possible modification )
    var infoFile = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY + Ti.Filesystem.separator +
guide.id + Ti.Filesystem.separator + "info.xml");
    infoFile.write('<guide>');
    infoFile.write('<id>' + guide.id + '</id>',true);
    infoFile.write('<name>' + guide.name + '</name>',true);
    infoFile.write('<city>' + guide.city + '</city>',true);
    infoFile.write('<description>' + guide.desc + '</description>',true);
    infoFile.write('<lastPlaceId>' + guide.lastPlaceId + '</lastPlaceId>',true);
    infoFile.write('</guide>',true);
    infoFile.write('\n',true);
    // 3. Create the PLACES FILE within the new GUIDE folder
    //    ( IF EXISTS: nothing happens )
    var places = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY + Ti.Filesystem.separator +
guide.id, "PLACES");
    if(!places.exists()){
        places.createDirectory();
    }

    // 4. Iterate over guide.places
    for(var p = 0; p < guide.places.length; p++){
        var id = guide.places[p].id + "";
        var placeFolder;
        var finalPlacesFolderName = MY_GUIDES_DIRECTORY + Ti.Filesystem.separator +
guide.id + Ti.Filesystem.separator + "PLACES";
        // If the place is already saved:
        if(guide.places[p].saved){
            placeFolder = Ti.Filesystem.getFile(finalPlacesFolderName, id);
        }
        // If the place is NOT already saved:
        else {
            placeFolder = Ti.Filesystem.getFile(PLACES_AUX_DIRECOTRY, id);
        }
        // If the place is mark to delete
        if(guide.places[p].deleted){
            placeFolder.deleteDirectory(true);
        }
        else{
            // If the place is not saved, we have to move the content from the temporary
            folder to the new folder
            if(!guide.places[p].saved){
                moveDirectory(placeFolder,places);
                placeFolder = Ti.Filesystem.getFile(finalPlacesFolderName, id);
            }
        }
    }
}

```

```

    }
    // Create the info.xml file
    // 2. Create the XML info file within the new GUIDE folder
    // ( IF EXISTS: overwrite info.xml in case a possible modification )
    var placeInfoFile = Ti.Filesystem.getFile(finalPlacesFolderName +
Ti.Filesystem.separator + id,"info.xml");
    placeInfoFile.write('<place>');
    placeInfoFile.write('<id>' + guide.places[p].id + '</id>',true);
    placeInfoFile.write('<name>' + guide.places[p].name + '</name>',true);
    placeInfoFile.write('<description>' + guide.places[p].desc +
'</description>',true);
    placeInfoFile.write('<longitude>' + guide.places[p].longitude +
'</longitude>',true);
    placeInfoFile.write('<latitude>' + guide.places[p].latitude +
'</latitude>',true);
    placeInfoFile.write('</place>',true);
    guide.places[p].saved = true;
    guide.placesAux[p].saved = true;
    }
}
};
/*
 * Remove a Guide from the file system. It will remove all the sub folders
 *
 * @param guideId Id of the guide we want to remove
 */
exports.removeGuide = function(guideId){
    var guideFolder = Ti.Filesystem.getFile(MY_GUIDES_DIRECTORY + Ti.Filesystem.separator
+ guideId);
    guideFolder.deleteDirectory(true);
};
/*
 * Save a picture into the temporary folder.
 *
 * @param request: { Picture Name + Picture data }
 */
exports.savePicture = function(request){
    var file = Ti.Filesystem.getFile(PLACES_AUX_DIRECOTRY + Ti.Filesystem.separator +
request.name,"p"+request.name+".jpg");
    file.write(request.data);
};
/*
 * Get the directories name where the media information (pictures and audio) will be
stored
 *
 * @param placeId: Id of the place
 * @param guideId: Id of the guide that contains the place
 *
 * @return array with the Media directories that contains the media files. If the guides
is not saved, the media directory will be within the temporary folder.
 * If the guide is saved, the media direcotry will be within the guide directory.
 */
exports.getMediaDirectories = function(placeId, guideId){
    var media = [
        MY_GUIDES_DIRECTORY + Ti.Filesystem.separator +guideId +
Ti.Filesystem.separator + "PLACES" + Ti.Filesystem.separator + placeId,
        PLACES_AUX_DIRECOTRY + Ti.Filesystem.separator + placeId +
Ti.Filesystem.separator
    ];
    return media;
};
};

/*****

```



```

* AUXILIAR FUNCTIONS
*****/
/*
* Get the name of the PLACES_AUX directory depending the OS
*
* @return String with the PLACE_AUX directory
*/
function getPLACES_AUX_DIRECTORY(){
    if (Ti.Platform.osname === 'android') {
        return PLACES_AUX_DIRECOTRY = Ti.Filesystem.externalStorageDirectory + "PLAC-
ES_AUX";
    }
    else {
        return PLACES_AUX_DIRECOTRY = Ti.Filesystem.applicationDataDirectory + "PLAC-
ES_AUX";
    }
}
/*
* Get the name of the MY_GUIDES directory depending the OS
*
* @return String with the MY_GUIDES directory
*/
function getMY_GUIDES_DIRECTORY(){
    return Ti.Filesystem.applicationDataDirectory + "MY_GUIDES";
}
/*
* Move a temporary place directory to the corresponding guide directory. It will move
also the content inside (picture and audio)
*
* @param from Temporary directory of the place
* @param to Final destination within the GUIDE/PLACES directory
*/
function moveDirectory(from,to){
    var name = from.name;
    // Create a new directory within the "to" folder with the same name than the ones we
want to move
    var newFolder = Ti.Filesystem.getFile(to.nativePath,name);
    newFolder.createDirectory();
    // Move the content within the "from" directory
    var folderContent = from.getDirectoryListing();
    for(var f = 0; f < folderContent.length; f++){
        var file = Ti.Filesystem.getFile(from.nativePath,folderContent[f]);
        file.move(newFolder.nativePath + Ti.Filesystem.separator + folderContent[f]);
    }
}

```

Guide.js

```

/*
* USE CASE: TRAVEL GUIDES APP
* GUIDE
* Module that define the content of a Guide Object
*/
/*
* Constructor of the module
*
* @param saved Boolean that indicates if the guide is already saved or not
* @param id Identification number of the guide
* @param name Name of the guide
* @param desc Description of the guide
* @param city City of the guide
* @param places List of places of the guide

```

```

    * @param placesAux Auxiliar list of places of the guide. It will be used to restore the
    changes in case the user cancel any modification.
    * @param lastPlaceId Id of the last places created within the guide *
    * @return JSON object with the guide information
    */
function Guide(saved,id,name,desc,city,places,placesAux,lastPlaceId) {
    var json = {
        saved: saved, id: id,name: name,desc: desc,city: city,places: places,placesAux:
placesAux, lastPlaceId: lastPlaceId,
    };
    return json;
};
module.exports = Guide;

```

Place.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * PLACE
 * Module that define the content of a Place Object
 */
/*
 * Constructor of the module
 *
 * @param saved Boolean that indicates if the place is already saved or not
 * @param inList Boolean that indicates if the place is already on the list of places or
not
 * @param id Identification number of the place
 * @param name Name of the place
 * @param desc Description of the place
 * @param longitude Longitude of the place
 * @param latitude Latitude of the place
 * @param mediaDirectories Array with the directories name where the media content of
the place is stored. It is an array with two positions.
 *     mediaDirectories[0] = Directory where the media content is stored when the
guide is saved
 *     mediaDirectories[1] = Directory where the media content is stored when the
guide is NOT saved
 * @return JSON object with the place information
 */
function Place(saved,inList,deleted,id,name,desc,longitude,latitude,mediaDirectories) {
    var json = {saved: saved,inList: inList,deleted: deleted,id: id,name: name,desc:
desc,longitude: longitude,latitude: latitude,mediaDirectory: mediaDirectories[0], medi-
aDirectoryAux: mediaDirectories[1],
    };
    return json;
};
module.exports = Place;

```

AudioRecorder.js

```

var audioRecorder = require('com.codeboxed.audiorecorder');
var fileRecorded;
audioRecorder.setAudioFormat("DEFAULT");
audioRecorder.setAudioEncoder("DEFAULT");
audioRecorder.setMaxDuration(60000); // 60 seconds
audioRecorder.setMaxFileSize(1000000);
exports.record = function(name){ var path = name; audioRecorder.setFileName(path,"3gp");
    audioRecorder.start();
};
exports.stop = function(){ fileRecorded = audioRecorder.stop();};
exports.play = function(){
    audioRecorder.playAudio(fileRecorded);
};

```

Map.js

```

/*
 * USE CASE: TRAVEL GUIDES APP
 * MAP VIEW
 * View with the map of the application.
 * In Android just one instance of this object can be created.
 */
/*
 * Constructor of the module
 */
function Map() {
    // Configuration of the map module
    Titanium.Geolocation.purpose = 'Show a map';
    Titanium.Geolocation.distanceFilter = 10;
    // Map View
    var mapView = Titanium.Map.createView({
        mapType: Ti.Map.STANDARD_TYPE,
        regionFit: true,
        userLocation: true,
        visible: true,
        touchEnabled: true,
        focusable: false,
        myLatitude: 0,
        myLongitude: 0,
    });
    // Callback function when position changes
    Titanium.Geolocation.getCurrentPosition(function(e) {
        setPosition(e, "getCurrentPosition");
    });
    // Callback function when position changes
    Titanium.Geolocation.addEventListener('location', function(e) {
        setPosition(e, "location event");
    });
    // Set the map location with the information given by the 'source' on the parameter
    'e'
    function setPosition(e, source) {
        // If there is an error obtaining the position
        if(e.error){}
        else{
            // Create a region with the data obtained
            var region = {
                latitude: e.coords.latitude,
                longitude: e.coords.longitude,
                animate:true,
                latitudeDelta:0.1,
                longitudeDelta:0.1,
            };
            // Set up the map
            mapView.setLocation(region);
            mapView.myLatitude = region.latitude;
            mapView.myLongitude = region.longitude;
        }
    }
    return mapView;
};
module.exports = Map;

```