POLITECNICO
MILANO 1863

Travlendar+ | Software Engineering II Project

Implementation and Test Deliverable

Authors

Antonio Frighetto

Leonardo Givoli

Hichame Yessou

Professor

Elisabetta Di Nitto

Academic year 2017/2018

Contents

Chapter 1

Introduction

This document has the purpose to explain what has been implemented and the underlying technologies adopted while coding the application Travlendar+.

1.1 Scope

This document aims at helping users - those who are alleged to test out the application - to prepare a basic environment to give the application a try.

Chapter 2

Tools, Frameworks and Functionalities

2.1 Tools

Here are the tools we relied on while developing the application:

- Visual Studio 2017 IDE version 15.5.2 (Community/Professional/Enterprise), with .NET Desktop Development and Mobile Development with .NET installed, both for Windows and macOS.
- Java JDK 1.8.
- Android SDK 8 - 26 With Platform Tools and Build Tools.
- Android emulator.
- iOS simulator.

2.2 Frameworks

The application has been implemented as a cross-platform mobile application using the *Xamarin.Forms* framework. As stated in the Overview of the Architectural Design, the programming language used is C#. With Xamarin.Forms we had the opportunity of using the .NET Framework which allows a much broader support to solve the problems that we faced. The architectural pattern follows the decision took in the Design Document, the Model-View-ViewModel (MVVM). Designing the majority of the application logic in the ViewModel helps to keep a neat distinction between the UI code.

As said in the DD, the client-server model is achieved with AWS Cognito, the main player of our data synchronization functionality by implementing the .NET AWS Client that consumes the service authenticating the users on the Identity Pool of Travlendar+. The login process to authenticate the user has been implemented with the Facebook client for Xamarin, consuming the wrapped APIs to retrieve the identity of the users.

For the basic user interface of the calendar we relied on a Telerik plugin, which provides us with some basic APIs to change view (e.g. switching from Months to Years), but after that, we've customized it by ourselves.

For the Maps, we used Xamarin.Forms.Maps APIs which wraps native maps APIs. For the navigation, we relied on Google Maps, which is where the user is redirected to (already set up with user settings) when navigation occurs.

2.3 Functionalities

Referring to the RASD and the DD documents, we have implemented the majority of the functionalities stated. First of all, a user must log in to his Facebook Account in order to access to the main interface of Travlendar+.

Main Interface

The main interface of Travlendar+ consists of a calendar that allows the user to manage the main activity that a calendar can offer and other custom functionality. In particular:

- The user can switch to different calendar views: Week, Month and Day. Additionally, if he should go back to today's date, a simple button in the toolbar has been implemented for this functionality.
- In the month and day view, every meeting that the user has added before are shown on the respective day cell so that he can visualize his appointments easily and quickly.
- Through the tool-bar the user can add new appointments to his calendar, visualize the settings of the behaviour of Travlendar+ logic, manage his transport tickets and logout from the application.

Add appointment interface

The Add appointment page is the main functionality of Travlendar+, that allows the user to create a new appointment by following the rule of Travlendar+ logic. The add appointment page lets the user adding, removing and modifying the following items:

- **Title:** the name of the appointment that must not be empty
- **Location:** the address of the appointment is set through a redirect to Maps where a user can write down the address manually or take his current position.
- **All day**: a simple toggle that allows the user to add an appointment in his calendar that covers all day.
- **Start and End**: these are the information of the event's beginning and the event's end.
- **Alert**: the toggle, if selected, will notify the user 10 minutes ahead of schedule that the appointment is going to start.
- **Calendar**: an event can have a category such that Work, Home, Birthday, Festivity.
- **Notes**: a section where the user can specify some additional information about the event.

The logic behind the Add appointment page offers the main functionality of Travlendar+. In particular:

- An event already added cannot be inserted into the calendar.
- Two events cannot be overlapped. An alert will be displayed in order to notify the user.
- Considering the settings information that the user can modify in the main page if the lunch break is selected, the application automatically reserve a time slot for the lunch break so that an appointment cannot be added during the lunchtime set by the user.

Once an event has been added, the user can easily see all the information by clicking on the event. In particular, the user can modify all the information added before, remove the event from his calendar and navigate to the event's location. If the user has set the application settings, before launching Google Maps the navigation functionality creates the correct navigation with the settings information, in particular by inserting the correct means of transport or the habits.

Settings page

In this section, the user can define various kind of user preferences. The settings support different travel means such that car, bike or public transport and the user can activate or deactivate each of these. Additionally, the user can also activate the carbon footprint constraint that automatically selects the combination of transportation means. At the end, the lunch break toggle allows the user to reserve slot every day for the lunch break. In particular, the user can define the start time and how many minutes long is the lunch break. The time slot must be between 11.30am and 14.00pm and it must be at least half an hour long. If the user has not respected this constraint for the lunch break, the settings cannot be saved and an alert notifies the user to select the correct start time.

Tickets page

This page allows the user to save their personal Tickets, providing a fast access to them as they are presented as a List, identifying each of them with a specific name. The user can either import the tickets from the media gallery of the device or take a picture of the physical ticket and keep it in Travlendar+. Tapping on a specific Ticket will be showed up in full-screen allowing the inspection. When taking or choosing a picture, the user will be previewed with the result before saving it. The tickets are saved on the device and specifically for the user, meaning that if a different user will login won't see the other user's tickets. Long pressing on the Ticket will allow the user to view the ticket or deleting it.

We wondered ourselves what we should have created for handling the tickets. Perhaps one of the simplest solutions was to redirect the user to an application like ATM or of that kind so that he could directly buy the tickets from there. However, this limited the user to use it around Milan, and as soon as one wishes to expand the geographic area, ATM would not have worked anymore. Then another application should have been added through which the user would have been redirected to. This would have created maintainability issues in the long term, and that is exactly what drove us to develop such ticket page!

Chapter 3

Code Structure

3.1 Structure

As said in the DD, to address the issue of maximizing the code sharing, our application project is based on a *Portable Class Library* (PCL), which means that the main solution consists of a Core PCL project, which contains all the pages (*Views*) and, mostly for each page, its counterpart of business logic (*ViewModels*) and two other platform-specific projects. The latter, specifically for iOS and Android, contain anything that must be written for a specific platform. This typically consists of renderers, a way that Xamarin.Forms provide to developers to customize the appearance and behaviour of Xamarin.Forms controls on each platform and handle native elements differently, and other code which is platform-dependent. So, the majority of the functionalities, wrapped in classes, are located in the PCL project, which shares the business logic across the implementation and the customization is left to the platform-specific projects.

The PCL consists of two folders, the *AppCore* and the *Framework* ones, which contains respectively the implementation of the core logic and the abstract classes or interfaces that are implemented by the PCL itself or the platform projects, and *App.cs*, the entry point of the application. Again, we respected the choices mentioned in the DD, which means that, having used the MVVM architectural pattern, not only we tried to maximize the code sharing implementing almost all the functionalities we were requested to - with a great focus on quality too -, but we achieved separation of concerns and used platform code through the dependency service as well.

Each of the two platform-related projects includes a *Dependencies* folder that contains the platform implementations of the platform functionality needed to be shared. An example could be the implementation of the Toast notification (a non-annoying pop-up which mainly informs users that something went wrong), natively present on Android but not on iOS (so it has been implemented on iOS too). The other folder is, as said, the so-called *Renderers*, which implements a native UI element changing the normal view of the Xamarin.Form element in something specifically customized for the platform. Here's an example too: the secondary toolbar, which is the navigation bar that includes all the main buttons such as *Today*, *Tickets* and so on, is perfectly rendered on Android through the pop-up button (the three dots button). On iOS, however, such bar is not rendered at all, and, at the time of writing, has not been implemented by Xamarin.Forms yet. So we created it by ourselves, directly with native iOS API. As can be seen from the renderers, the code has been highly customized to achieve also a friendly and pretty user interface. Figure 3.1 shows how the code has been laid out.

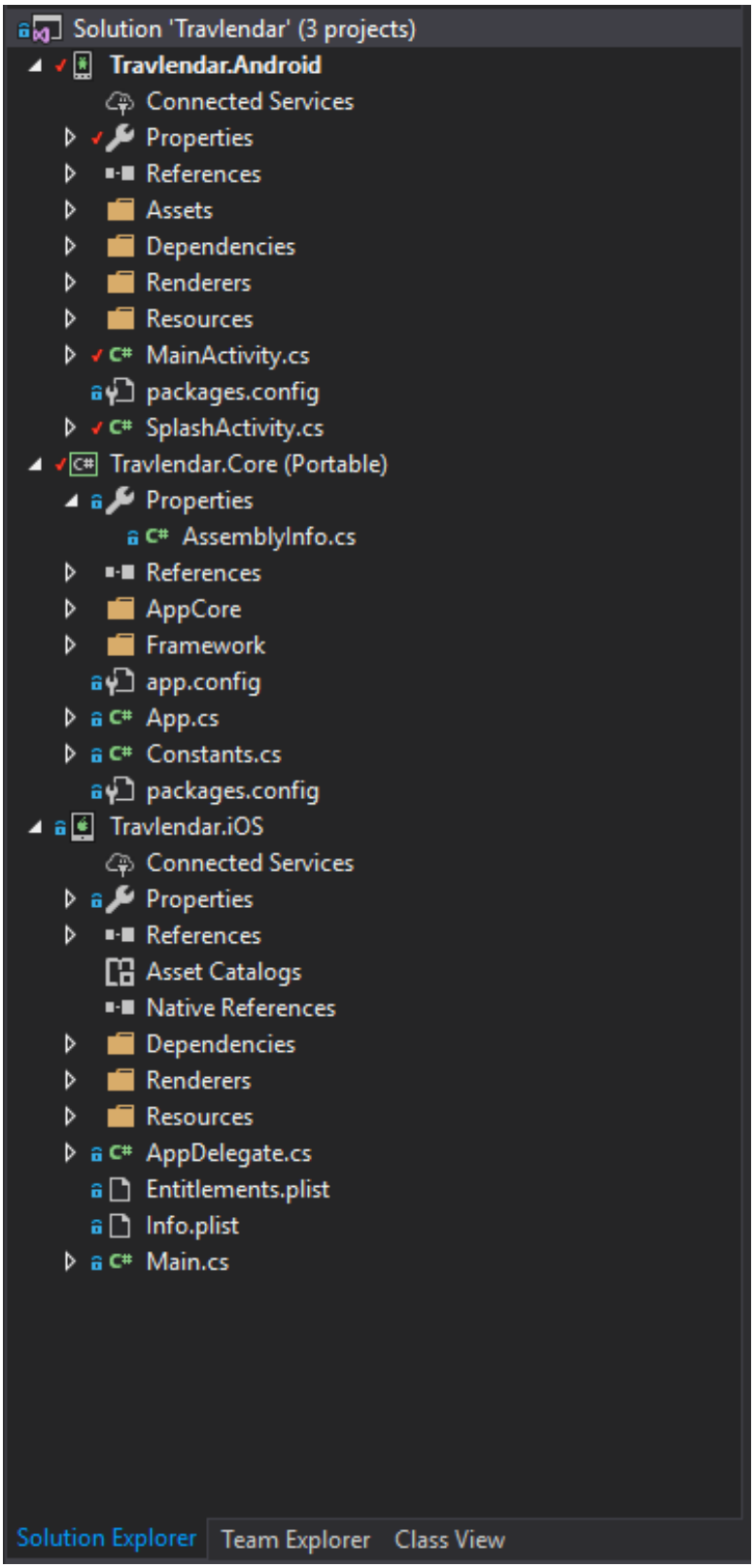

Figure 3.1: Code Structure.

Chapter 4

Installation and testing

4.1 Installation

Grab here the zip folder with the main solution and all the code inside. Unzip it, open it with Visual Studio and then compile it with the aforementioned tools. All the packages needed will be restored by Visual Studio automatically. In order to allow the Facebook Login API to verify the authenticity of the machine that builds the application, it's needed to generate and add your Development Key Hash inside the Facebook Developers portal.
Generating your Development Key Hash will require your Xamarin debug.keystore that usually is placed in:

```
C:\Users\USERNAME\AppData\Local\Xamarin\Mono for Android\debug.keystore
```

for Windows users, while for macOS users it should be in:

```
~/.local/share/Xamarin/Mono for Android/debug.keystore
```

Once located the keystore we will generate the unique Hash Key for your machine with the command (Windows and macOS)

```
keytool -exportcert -alias androiddebugkey -keystore debug.keystore |
openssl sha1 -binary | openssl base64
```

It should be a string in the format of base64 (e.g. tOv9E7+ETK8kjylFQBKlr5QErBk=) and you will need to send it to one member of our team as well as the Facebook account that will be used for the test.

This is needed due to Facebook strict rules on non-reviewed applications. Submitting the Facebook integration for an approval would have required more effort without knowing if it would have been accepted by the Facebook Developers team.
Inside the shared folder, you can find directly the binaries, namely the `.apk` (for testing the application on Android) and the `.ipa` (for testing the application on iOS). Of course, to have a better experience the application should be tested on both platforms. Remind that, - unless you're jailbroken - Apple strictly forbids you to run unsigned code. This means that an own provisioning self-signed profile must be created if you plan to test it on iOS too.
Regarding Android instead, it's just needed to check Unknown sources under the System/Security section of your device

4.2 Testing

Each Travlendar feature has been tested mainly according to the rules of the well-known *Test Driven Development* (TDD) technique. Specifically, localized tests were implemented in order to better study the behavior of the application and the resolution of errors or bugs. This implies that we continued focusing on the next feature only after that the software had passed the generated tests. As said in the DD, we have tested the performance of our application with Xamarin Profiler tool. The tool has allowed us to collect information about our mobile application, regarding time complexity, usage of particular method and the memory being allocated. Xamarin Profiler has enabled us to drill deep and analyze these metrics to pinpoint problem areas in code. Furthermore this tool has helped us to take care to understand where most of the time was spent in our application, and how much memory was used by our mobile application.

We decided to use Xamarin Profiler because, especially in mobile application, unoptimized code is much noticeable and the success of our application depends a lot on optimized code that runs efficiently. Additionally, just for the iOS application, we used TestFlight, a service for over-the-air installation and testing of mobile applications. We have distributed our application to external beta testers, who could subsequently send feedback about the application. This approach was very helpful for testing quickly every functionality of our application and for retrieving any consideration from external users. Again, as said in the DD, xUnit.net was used too.

Chapter 5

Screenshots

Here we put some screenshots of our Travlendar+ mobile application.
From iOS device:

- Login page screenshot fig. 5.1
- Calendar page screenshot fig. 5.2
- Calendar month view screenshot fig. 5.3
- Appointment creation page screenshot fig. 5.4
- Map page screenshot fig. 5.5
- List view screenshot fig. 5.6

From Android device:

- Calendar page screenshot fig. 5.7
- List view screenshot fig. 5.8
- Calendar month view screenshot fig. 5.9
- Settings page screenshot fig. 5.10

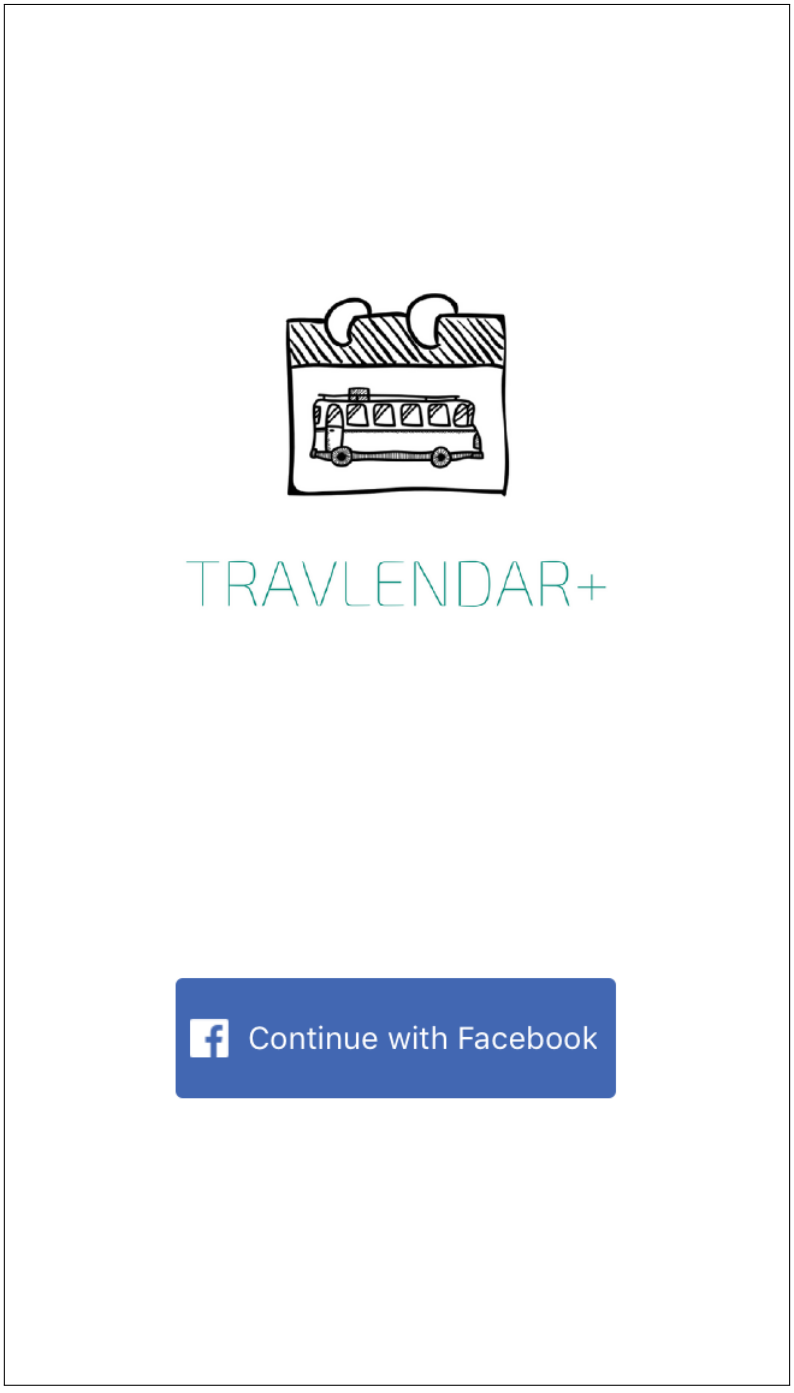

Figure 5.1: Login page screenshot.

Figure 5.2: Calendar page screenshot.

Figure 5.3: Calendar month view screenshot.

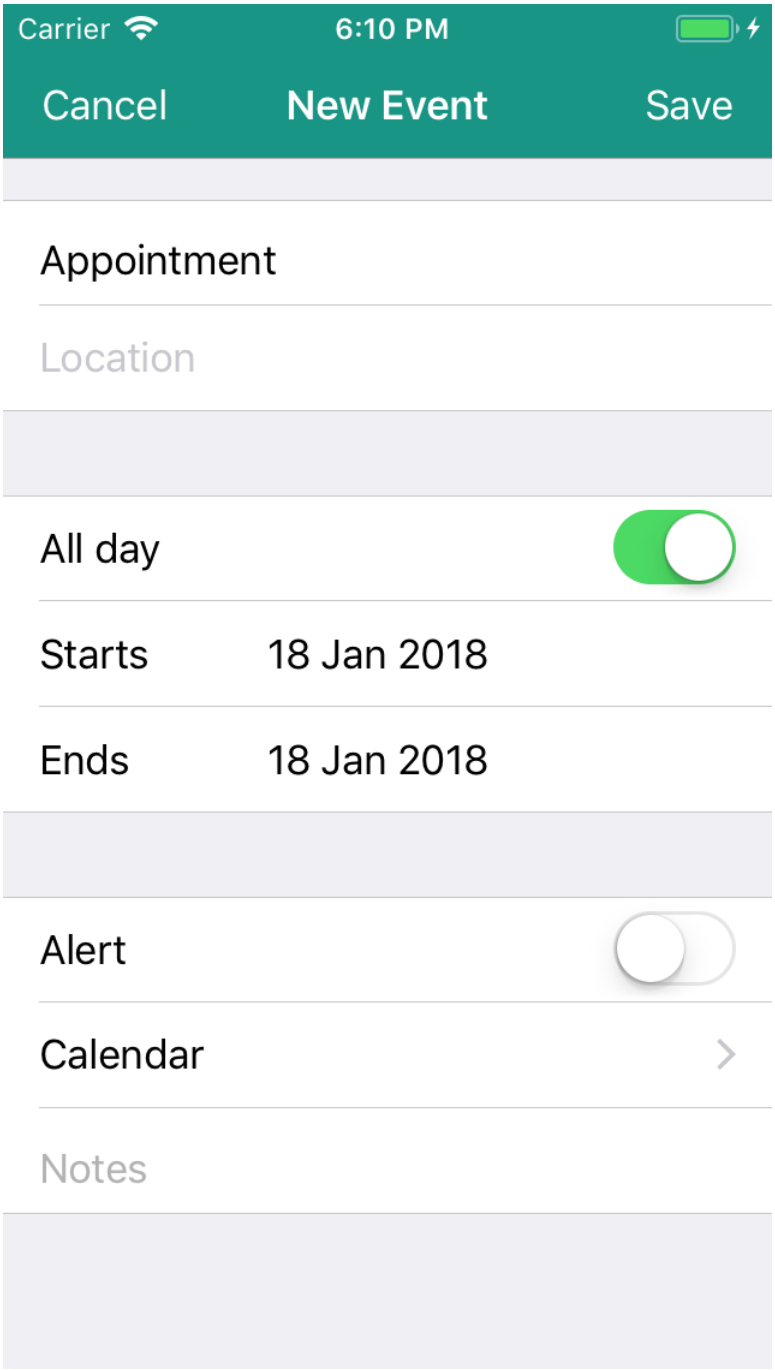

Figure 5.4: Appointment creation page screenshot.

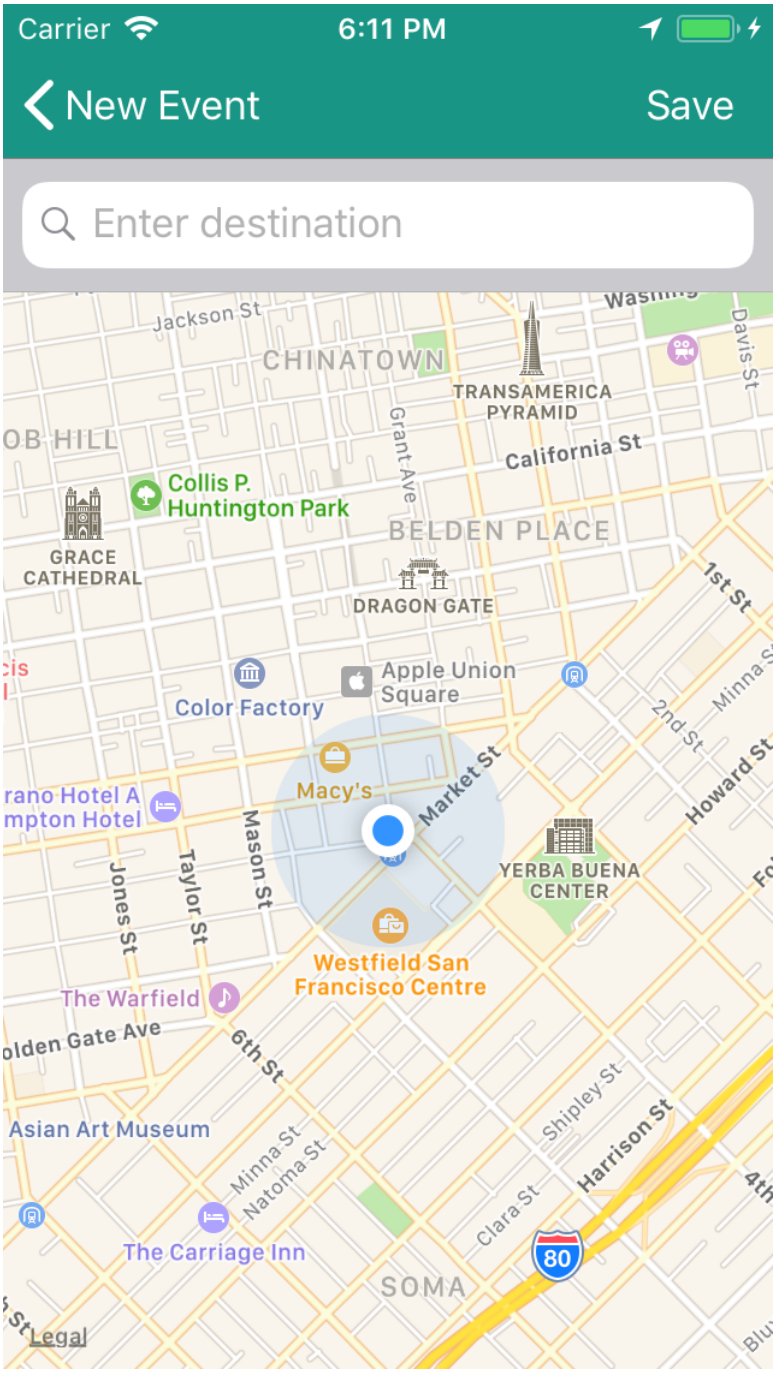

Figure 5.5: Map page used for fill the location field in appointment creation.

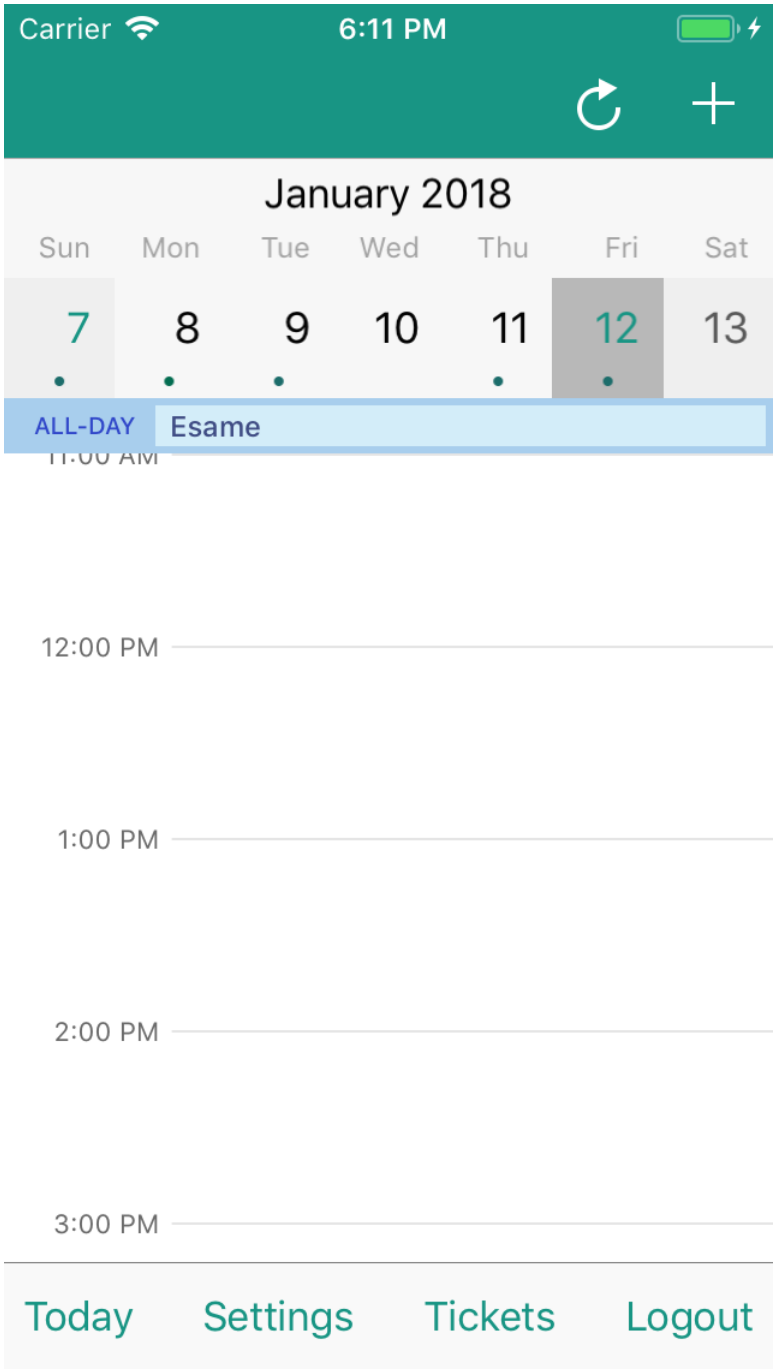

Figure 5.6: List view with an all-day appointment added.

Figure 5.7: Calendar page screenshot

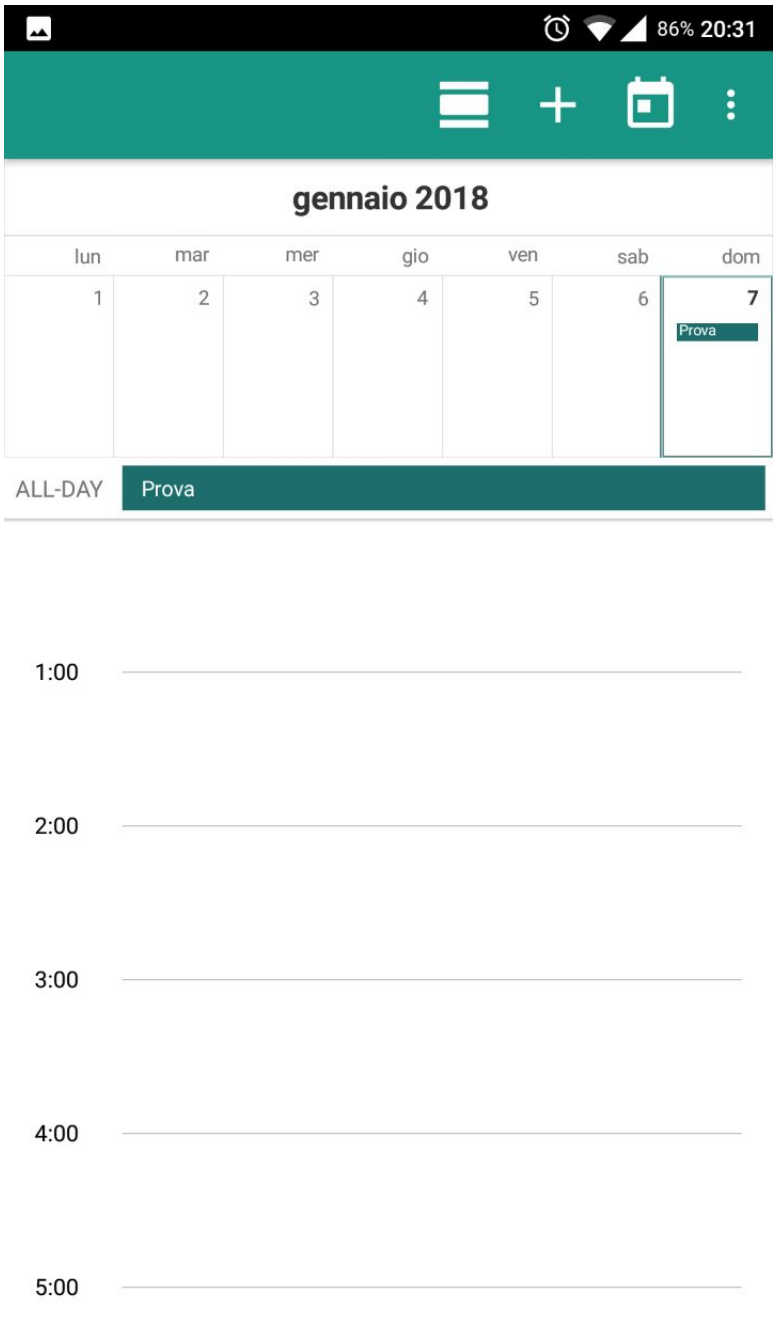

Figure 5.8: List view with an all-day appointment added.

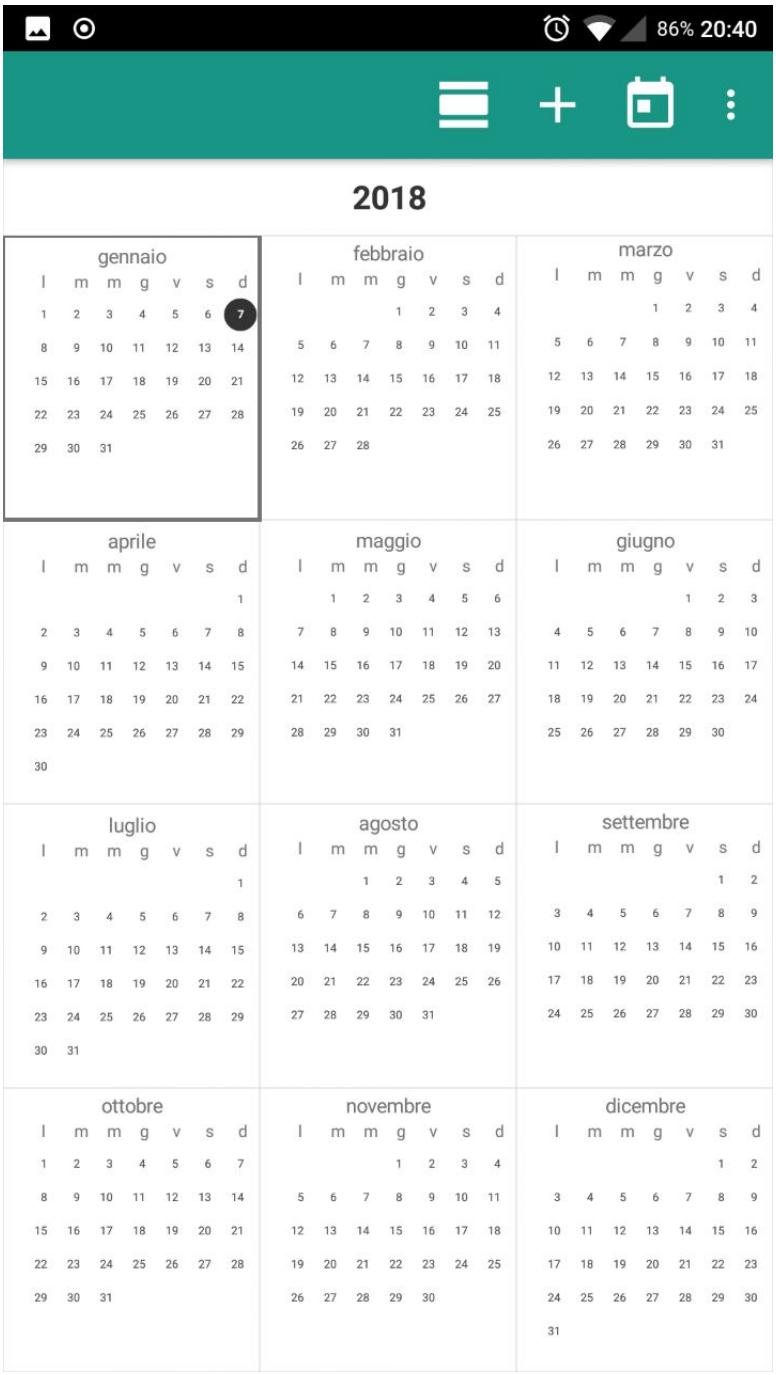

Figure 5.9: Calendar month view screenshot.

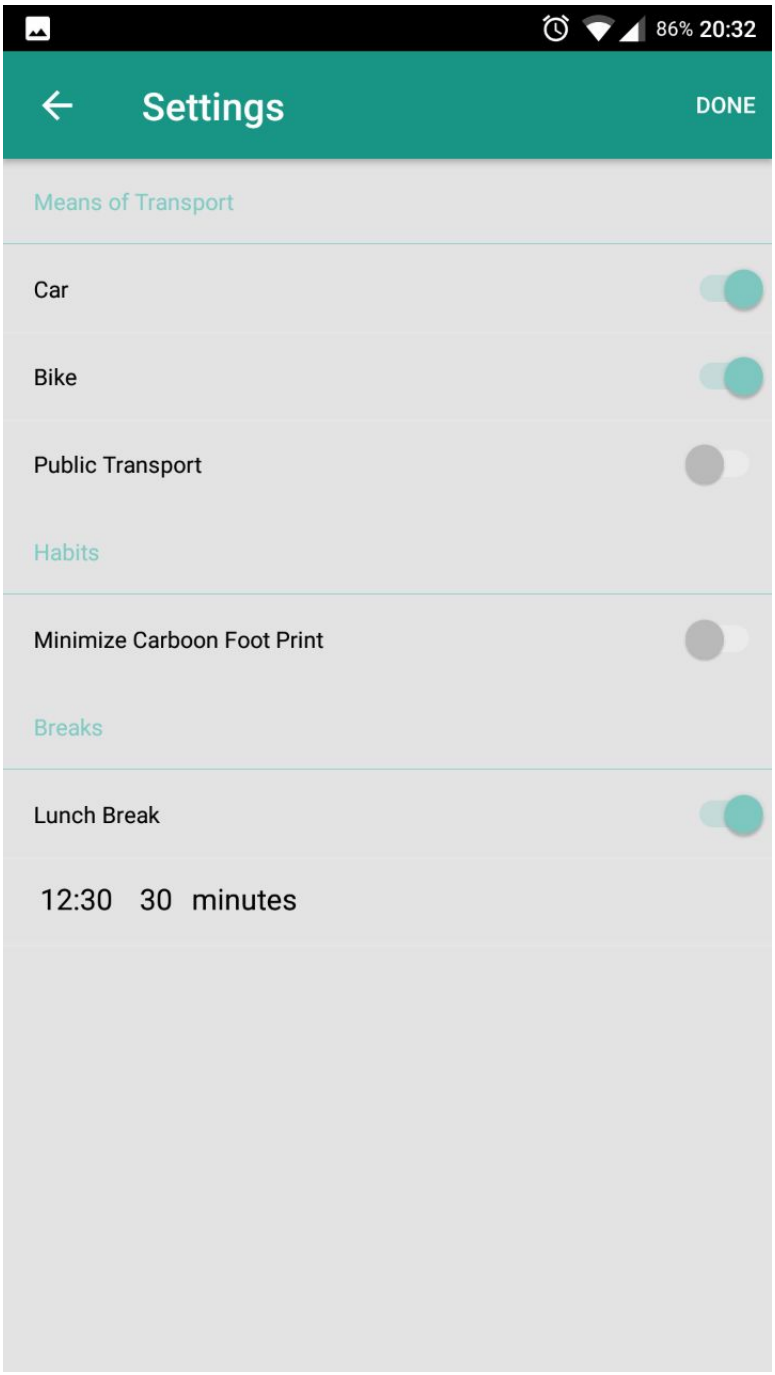

Figure 5.10: Settings page screenshot.

Appendix

Hours of work

All statistics about commits and code contribution are available on our repository on GitHub. Please, remind that all commits on our repository have been reviewed together. Also the whole work has been equally divided and everyone helped each other.

- Antonio Frighetto: 100+ hours.
- Leonardo Givoli: 100+ hours.
- Hichame Yessou: 100+ hours.