POLITECNICO
MILANO 1863

Software Engineering II

Travlendar+

Implementation & Testing
Document

Authors:

Edoardo D'Amico
Gabbolini Giovanni
Parroni Federico

1st December 2017

Indice

Chapter 1

Introduction

After the release of the DD document the application Travlendar+ has been made up following the guidelines explained in the previous two documents. The application is a prototype, these means that not all the functionalities described in the RASD and DD documents have been implemented but only the most important ones. By the way the application manages to cover the basic needs of a user. Tests and debug of the application have been carried out on a virtual android device and on a real android device. For the implementation of this prototype a month and half has been spent by our three-people team.

1.1 Front page

1.2 Purpose

1.3 Scope

In this paper there is an overview of the main topics concerning how the implementation of the prototype of Travlendar+ application has been made. First of all the functionalities that are currently implemented in the software will be presented. then the adopted development frameworks will be explained with its pros and cons. then it comes to the structure of the source code of the application. in the end the tests performed to test the right functioning of the application and their outcomes and how the various part of the application has been put together will be presented. In the last part of the document there is a brief guide on how to install the developed application.

1.4 Revision history

Chapter 2

Requirements and functionalities implemented

In this chapter it's reported a mapping between all the functionalities that were considered during the analysis part (i.e. the ones listed in section **RASD: 1.1** of the RASD, and then better described also in **RASD: 2.2** and **RASD: 3.2**. In these sections all the goals and requirements at which will be referred are listed) and the featuers that the proposed prototype actually has. Since functionalities and requirements are fully described by goals, here we will specify just which goals are actually implemented, explaining the reasons of the choices made. It's clear that, since the fullfilling of goals it's possible only when all the requirements associated are implemented, when it's said that a goal it's present in the prototype also all the requirements associated are implemented. However, a further description of requirements will be presented when needed.

2.0.1 Goal 1

The system should offer the possibility to create a new account

The functionality is fully implemented.

2.0.2 Goal 2

The system should be able to handle a login phase

The functionality is implemented but the requirement **RASD: R6** isn't: all the parts involving the online part of data synchronization are not implemented in the prototype. It has been chosen not to implement these features since they weren't considered to be basic, something not stricly needed for a prototype implementation. However, the data of the user are saved locally to the device in which the application it's installed: it won't be difficult to extend this client-side data management to a server-side one, once a fully implementation will be required.

2.0.3 Goal 3

The system should give to the signed user the possibility to recover his password

The functionality has not been implemented.

2.0.4 Goal 4

The system should allow the user to insert an appointment according to his necessities and his preferences

The functionality is fully implemented, but the appointments are saved (**RASD: R10**) just in the device and not online, as explained in 2.0.2.

2.0.5 Goal 5

The system should provide a way to modify an inserted appointment

The functionality is fully implemented, but the modified appointments are saved (**RASD: R12**) just in the device and not online, as explained in 2.0.2.

2.0.6 Goal 6

The system should provide a way to create a valid schedule of the user appointments when requested and display the scheduling result

The functionality is implemented, in particular all the various data are retrieved from the user and from external API (**RASD: R12** through **RASD: R16**), except for the informations about strike days and delays that are not yet considered, since it turned out that these data were available to be retrieved only by paing the various API services. So, since the application it's still a prototype and since these added details weren't bringing any basic features but just advanced ones, we decided to forget about them. Moreover, except for the described lacking data that are not considered, the **RASD: R17** it's fullfilled. Last, the created schedules are saved (**RASD: R18**) just in the device and not online, as explained in 2.0.2.

2.0.7 Goal 7

The system should let the user create valid multiple schedules and decide which one is chosen for the current day

This functionality is fully implemented.

2.0.8 Goal 8

The system should be able to book the travel means involved in the current schedule under user approval

This functionality is not fully implemented, our prototype presents just a draft of the final desired behaviour. Infact, a full implementation was too much effort-costy: it was needed to interface with the transit services and with the user's credit account, in a way that just a click was needed from the user side to buy the tickets for a schedule. So, since the purpose was to build a basic prototype, this feature was considered to be advanced, and so this functionality has being implemented as a simple redirecting to the website of the transit company. So **RASD: R20** it's not fullfilled.

2.0.9 Goal 9

The system should be able to display in real time user position and the directions to be followed in order to arrive to the next appointment on a

dinamically updated map

This functionality is implemented: when a schedule is running, the static directions that link all the appointments, according to the schedule that has beign computed, are displayed on the main page of the application, together with the user position. So, even if the directions are just static and not dynamic, the requirements **RASD: R21** through **RASD: R23** can be considered as fullfilled.

2.0.10 Goal 10

The system should be able to notify the user when a shared travel mean is available and it would optmize the current schedule

This functionality is not implemented, together with it's requirement. In particular the shared travel means are not considered at all in our prototype, since they can be thought as an extension of what it's actually implemented and don't add any relevant feature to our draft, apart from having more kind of travel means to choose. Moreover, the data-retrieving concerning the presence of neighbor shared means was available just for some kind of shared services. Anyway, the prototype it's prone to consider new travel services that can be added in the final version of the application without changing the structure of the code, as explained in **code structure section**

Chapter 3

Frameworks

In this chapter we show the main implementation details, in particular the choice we have made about frameworks, programming languages, tools, environment used to develop the entire application.

3.1 Frameworks and Programming languages

3.1.1 Database

Application data are stored in a MySQL database, located in a free remote host at 000.webhost.com. This service offers the possibility to have a completely free domain in which is present a MySQL database. We decided to choose MySQL because is a really reliable DBMS and allows to build quickly all the relational schemas thanks to his handy and comfortable web interface (php-admin). In addition, it is well known for its performance and flexibility.

3.1.2 Server side

The server side part has been developed using Slim Framework, a PHP set of libraries that facilitate the process to write a wide variety of web applications (`https://www.slimframework.com`). We chose this for its simplicity and rich documentation. PHP is the open source most popular server side language and it can run on both UNIX and Windows servers. In general, PHP is secure, fast, reliable and compatible with the majority of DBMS, so really suitable for developing web applications (and it is already installed in 000webhost hosts).

3.1.3 Client Side

The client side part consists in an Android application. It is written completely in Java, the most used object-oriented programming languages at all. The crucial advantage of Java is that it is platform independent: it can run on whichever machine, also in mobile devices. Android Studio, the IDE for developing android mobile application, is really integrated with Java and its packages-class structure. We chose to create an Android application because we already familiar in programming with Java and for its versatility. In addition, one can publish applications into the Google Play Store for free (for the Apple Store you have to pay the developer account fee). One disadvantage is the poor backward compatibility of Android, in fact lots of previous version of Android running on older devices do not support newest application. This because the Android framework is introducing more advanced features only nowadays. In conclusion, Android is quite good mobile environment, but it has some little drawbacks (battery usage, performances, anti-malware security...)

3.2 Middleware

The authorization mechanism we used is a middleware that allows to give some access control to the API offered by the server side service. OAuth libraries for PHP has been used in the API development (see chapter **RASD: 3.4.1**). This middleware handles communication between different levels of the API structure, providing a smart way to stratificate and separate the logic layers.

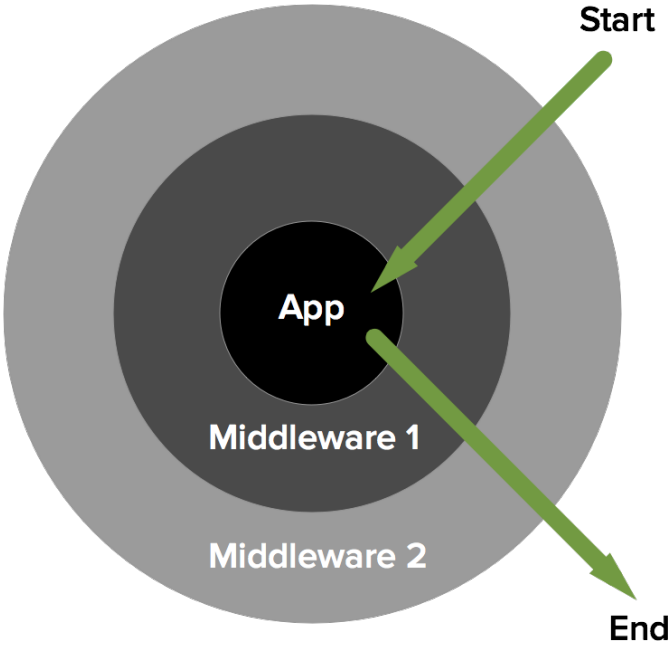

Figura 3.1: Middleware pattern

This is the core of the API in the index.php file of the server:

```
$app->map(['GET', 'POST'], Routes\Authorize::ROUTE, new
    Routes\Authorize($server, $renderer))->setName('authorize');
$app->post(Routes\Token::ROUTE, new Routes\Token($server))->setName('token');

$authorization = new Middleware\Authorization($server, $app->getContainer());

// ROUTES
$app->get('/hello/{name}', function (Request $request, Response $response) {
    $name = $request->getAttribute('name');
    $response->getBody()->write("Hello, $name");

    return $response;
});

$app->group('/api', function () use ($app) {
```

```
    $app->post('/register', function ($request, $response) {
        return \App\Controllers\AuthenticationController::Register($request,
            $response, $this->db);
    });


    $app->group('/user', function () use ($app) {

        $app->post('/profile', function ($request, $response) {
            return \App\Controllers\UserController::Profile($request,
                $response, $this->db);
        });

    });

})->add($authorization);


$app->run();
```

We can see how it's easy to define the APIs structure and relative callbacks.

3.3 API

In this first version of Travlendar, we have implemented only a few strictly necessary web api. The following table summaries the main available API:

3.3.1 Token: Request a bearer token to authorize the client application

POST travlendar/public/token

BODY:

grant_type: client_credentials

client_id: <id>

client_secret: <secret>

3.3.2 Token: Request a bearer token to authorize the user by username and password

POST travlendar/public/token

BODY:

grant_type: password

client_id: <id>

client_secret: <secret>

username: <username>

password: <password>

3.3.3 Registration: Register a new user

POST travlendar/public/api/register

HEADERS:

Authorization: Bearer <token>

BODY:

email: <email>

password: <password>

3.3.4 User profile: Request user profile information

POST travlendar/public/api/user/profile

HEADERS:

Authorization: Bearer <token>

BODY:

email: <email>

password: <password>

Chapter 4

Server code structure

The server code is organized in 3 main files:

index.php

This file contains the entry point of the web api. The central object of this part is $app, a Slim/App object that is responsible of all the API handling mechanism. The logical code flow is the following:

1. at first, it initializes a db connection reference (in order to use afterwards). This object (of PDO class) is useful to query the database and retrieve data in a high level manner.

2. it initializes the Authorization Server object and related Storage, providing the possibility to use the database as support to save and set authorization token to the client that will connect to the server. These classes offers a middleware layer used for authorization control.

3. it registers several routes. These allows to establish a mapping between an url request and a callback (that will handle that request and will send a response). For instance, when we go to <domain>/travlendar/public/api/user/profile, we handle that request with a callback (UserController::Profile) located in somewhere in the class structure. The majority of the routes has been set up with the authorization middleware, in order to be accessed only if the request contains a valid access token.

controllers/AuthenticationController.php

The class contained in this file manages the user authentication. This means that a user can register to the application and then be recognized through its credentials. The method Register is the callback for the /travlendar/public/api/register api. The method allows a user to register providing valid email and password.

controllers/UserController.php

The class contained in this file manages all the data related to a particular user, for example, profile, appointments, schedules... The method Profile is the callback for the /travlendar/public/api/user/profile api. The method allows to retrieve the information about a user profile (bike, car and public means pass).

Chapter 5

Client application code structure

In this chapter is shown the main structure of the application code. For the code structure, how said in the DD document, has been decided to use the MVC pattern. These means that there are mainly 3 category of class:

- **Model Class**, used for modelling the logical property of the objects represented in the class
- **View Class**, this class is used for modelling the design of a view of the applications
- **Controller Class**, these classes manage to handle the interaction beetwen the Model objects and the Views.

This is the project main structure:

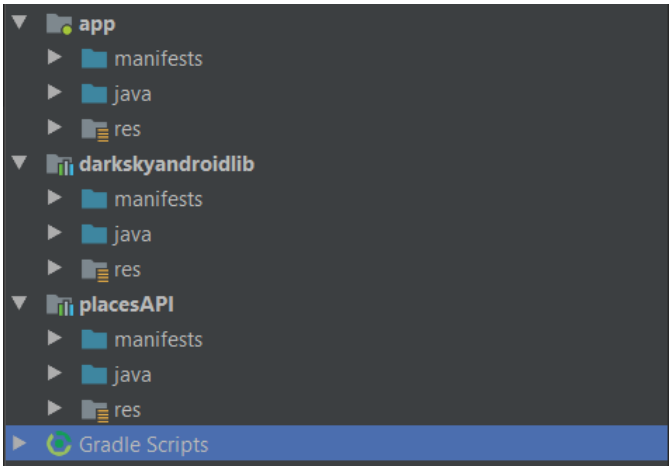

Figura 5.1: Global Structure of the code

The application is divided in three main projects:

- app project, containing the biggest part of the code
- darkskyandroidlib project, containing some classes used for managing the interaction between the application and the wheater forecast api

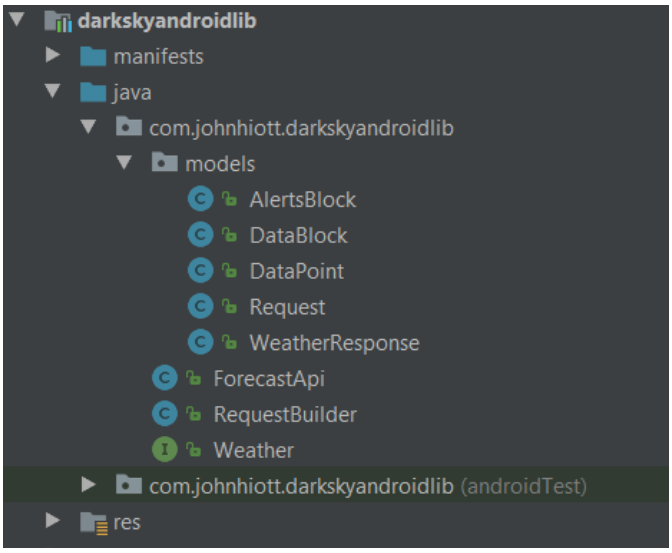

Figura 5.2: Wheather forecast api folder

- placesAPI project, containing the classes used for handle the interaction with the google map api

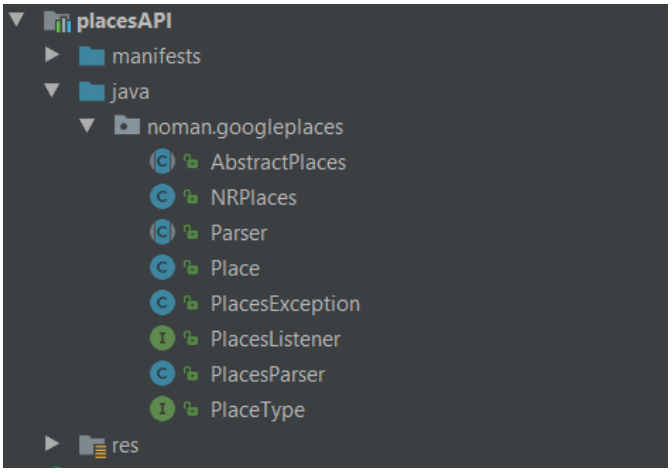

Figura 5.3: Places api folder

plus the Gradle Scripts an advanced build toolkit, to automate and manage the build process, while allowing you to define flexible custom build configurations. Each build configuration can define its own set of code and resources, while reusing the parts common to all versions of your app. The Android plugin for Gradle works with the build toolkit to provide processes and configurable settings that are specific to building and testing Android applications. One of the most important feature of the gradle is that it can generate the application apk.

In the following sections we explain in a more exhaustive way the **app project structure** going more deeply in it.

5.1 Manifests folder

Every application must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code.

Among other things, the manifest file does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application, which include the activities, services, broadcast receivers, and content providers that compose the application. It also names the classes that implement each of the components and publishes their capabilities, such as the Intent messages that they can handle. These declarations inform the Android system of the components and the conditions in which they can be launched.
- It determines the processes that host the application components.
- It declares the permissions that the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the application's components.

- It lists the Instrumentation classes that provide profiling and other information as the application runs. These declarations are present in the manifest only while the application is being developed and are removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

5.2 Res folder

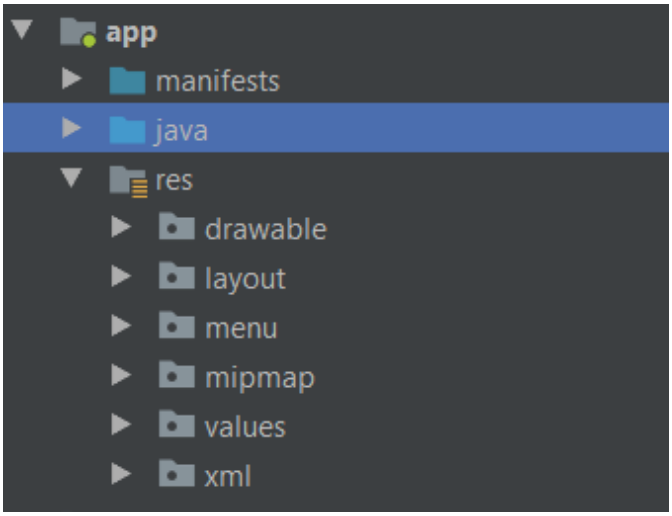

Figura 5.4: Res folder structure

This folder contain all the graphic elements such as the views of the application saved under xml format, or the images used for create the views, for example the button icon and stuff like that.

More precisely:

- **drawable** and **mipmap** folders, they contain all the images used for create the views.
- **layout** folder,it contains all the xml files representing the real view of the application such as the main page or the appointment page.
- **Menu** folder,there we can find other xml used for modelling the navigation bar (the bar located on the top of each view) of each single view.

- **values** folder, it contains some parameters for the view such as colors,dimensions,strings id and styles.

NOTA: LA CARTELLA XML NON CI DOVREBBE ESSERE IL FILE AL SUO INTERNO DOVREBBE ESSERE DENTRO LAYOUT !

5.3 Java folder

There are two folder one for the Model and the other for the Controller.

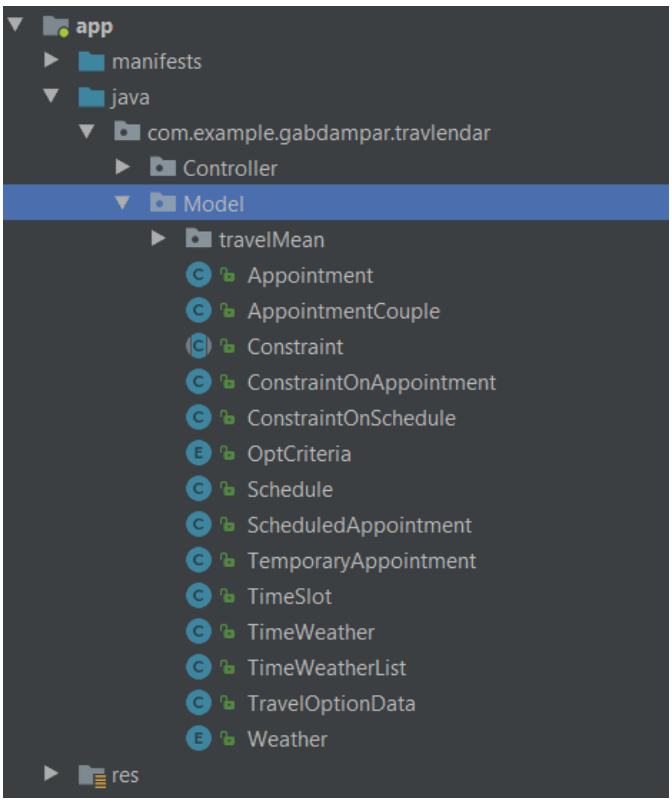

Figura 5.5: Model folder

In the model folder there all the classes used for the logic modeling, there is another folder used for group up all the travel mean since they have similar classes.

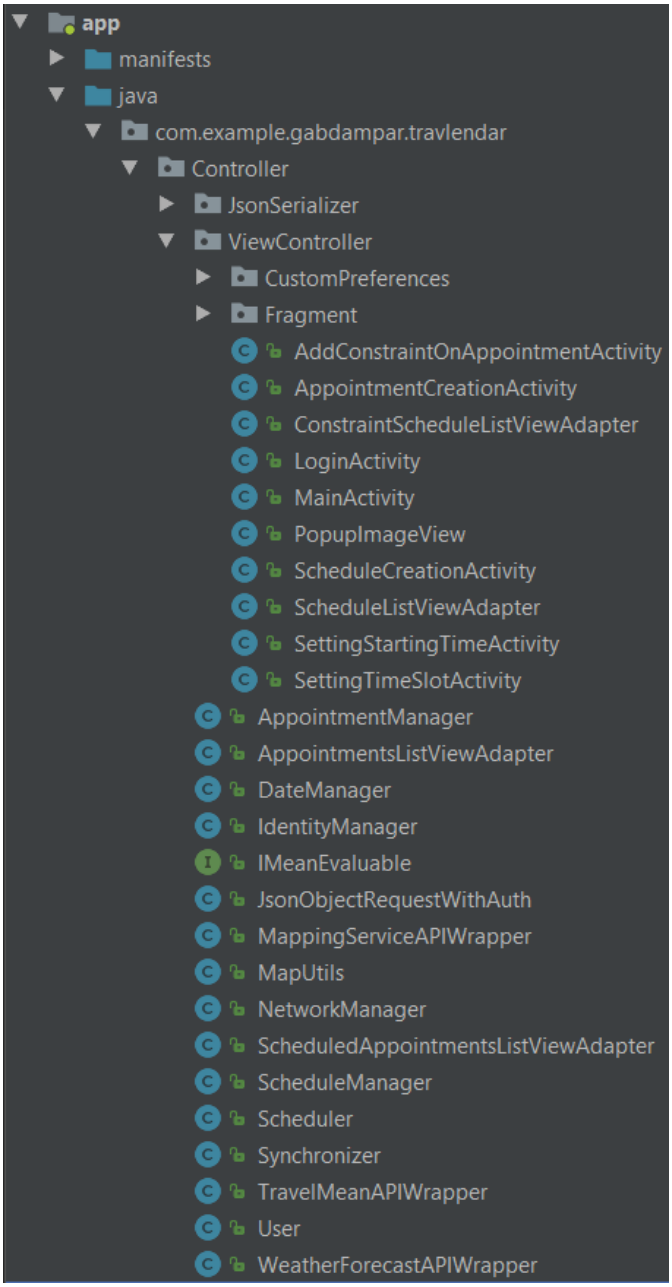

Figura 5.6: Controller folder

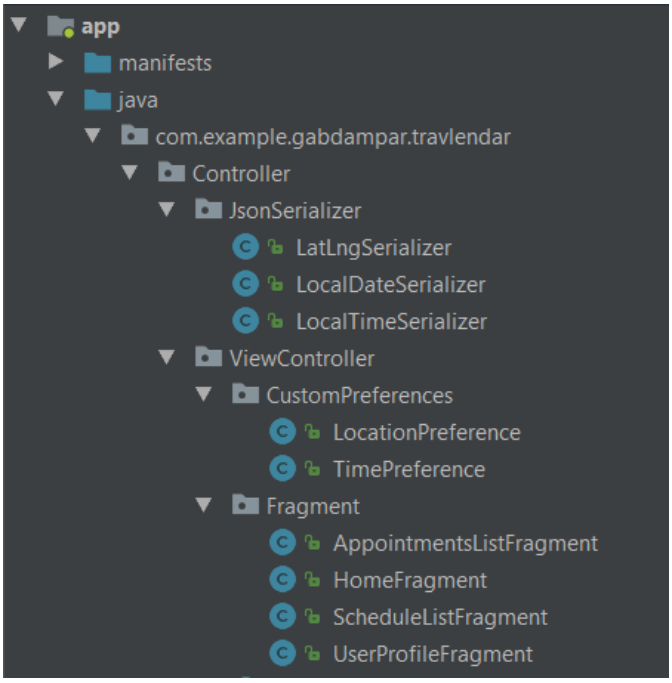

Figura 5.7: Other folders inside the Controller folder

The controller folder is divided in two main parts:

- Controller class
- View Controller Class, all the classes in the homonym package.

there is this division since android studio when a new view is created, generates automatically the controller of this view as a new class. We have added more controller in order to divide the tasks, that each controller have to perform, in a more logic way.

Must be mentioned the JsonSerializer folder, containing some classes used for serialize some model classes in order to save their object in the devices memory.

The folder CustomPreferences contains some classes used to specify a user preference such as the starting location of one of his schedule.

In the next part of the section the most important classes will be explained.

IdentityManager

This class manages all the necessary step allowing a user to be authenticated in the application. It is made up of 5 main methods:

1. TokenRequest: sends an asincronous request to the token api and return the obtained token through the specified callback
2. Login: basically call TokenRequest with the provided email and password and cache those credentials with the token in case of success
3. Register: call asincronously the register api passing the specified email and password
4. GetUserProfile: call asincronously the user profile api passing the specified email and password
5. Logout: de-cache the user credentials and token

LoginActivity

The activity that handle the interaction between the user and the Login view. The activity calls IdentityManager methods to perform login and registration processes.

Scheduler

This class is used to compute asincronously a new schedule. Inputs are an appointments list, a starting time, a starting location, some constraints and an optimization criteria. The only exposed method are IsConsistent and ComputeSchedule. The first one check if all the input parameters have been set correctly. The second one actually does the big work. It perform the following actions:

1. retrieving weather conditions information
2. computing predecessors and distances matrixes (see **DD: 3.2**) of all the appointments
3. recursively building arrangements (all the possible ways of ordering the appointments consistently with respect to the starting times)
4. trying to create a schedule from each arrangement. This consists of some substeps:
   (a) create temporary appointment with a wake-up dummy appointment. These appointments have additional information supporting next computations

(b) loop two by two the temporary appointments until the end and assign the best travel mean to the second. The best travel mean is chosen from the available and usable list of travel means. This brings to also assign the starting time and the cost (time, cost, carbon), calculated for that travel mean.

(c) check if there are a time conflict (starting time of the current appointment and ending time of the previous are overlapping) or a mean conflict (taking that mean causes to exceed the maximum distance imposed by some constraints).

(d) if confilicts are found, add a dummy constraint on a temporary appointment to avoid taking that mean. In case of time conflict, this dummy constraints allows to take a faster travel mean (penalizing at least the cost); in case of mean conflict, it allows to take a different mean (penalizing at least the time).

(e) repeat from point (b) until there are no more time/mean conflicts or no more dummy constraints can be added to any appointment

5. ordering all the created schedules

6. calling the mapping service api to get route information on the "best" schedule (with the least cost) and verify the real feasibility of that schedule (done by calling *getBestScheduleAsync*, see below for more details)

7. returning the schedule if feasible or null otherwise

getBestScheduleAsync(..) it's the point were the API calls are performed to get the real data for a schedule, and to decide if the schedules (ordered by the most convenient one) are actually feasible, infact all the computation was based on estimates so far. The method takes every schedule that was computed, starting from the most convenient one, and performs the API calls (*getTravelOptionData*(..) in particular) for every couple of scheduled appointments of the schedule, passing theire locations and starting times as parameters. If the results don't fit on the timings that were previously calculated, the schedule it's discarded and we try with the next one, until we conclude that there's a feasible schedule or not. In case a schedule it's accepted by this method, the various calls that were performed have set all the useful data for further computation on the **TravelOptionData** object, linking couples of scheduled appointments of the schedule. When the method terminates, the listener it's called, passing the computed schedule, if any, or null. Infact, since also the Scheduler deals with asynchronous calls (perfomed in

getBestScheduleAsync(..)) it has to provide an interface through which getting the results back. When the caller will receive the results, it will add it to the list of schedules.

ScheduleManager

It lists all the schedules that have been computed over time, other than an handle to the current schedule that it's under execution right now. Based on this variable, the **HomeFragment** can decide it's state. The class offers also the method *getDirectionsForRunningSchedule*(..) which retrieves, by means of the **TravelOptionData** objects contained in the schedule, a textual representation for the directions to give to the user. This class is made up as a Singleton, since just an instance of this object can be available.

AppointmentManager

It lists all the appointments that have been created over time. It also offers a method which, relying on the *getStopDistance*(..) method of **MappingServiceAPIWrapper**, sets the minimun distances to each kind of transit stop, from the selected appointment. This class is made up as a Singleton, since just an instance of this object can be available.

AppointmentCreationActivity

this class is responsible for the appointments creation and editing, before one of the two actions mentioned before is performed it checks if all the mandatory fields has been filled, if that has not been done it shows a warning message. This class handles the interactions with the minor classes AddConstrantOnAppointmentActivity, SettingStartingTimeActivity, SettingTimeSlotActivity, classes used for set some parameters during the creation of an appointment. These interaction is made mainly with the callback method onActivityResult called from one of the three classes when the data expected has been inserted by the user.

ScheduleCreationActivity

it manages the creation of the schedules grouping up all the data inserted by the user. If some mandatory fields are left empty it shows a warning message.

HomeFragment

It's the view that first appears to the user when he/she opens the application. It has, as background, an object of type **GoogleMap**, showing all the appointments of the actual day, spreaded across the region that it's considered. This is the view as it shows up with it's starting state, that is, without any schedule on running. The state of it changes when some schedule is ran: the map is resized and at the bottom appear the directions for the schedule that it's actually running. When a schedule it's stopped (by means of the Cross button on the view), the state of the view returns as initial.

UserProfileFragment

It's the view that lets the user change it's personal parameters, saved and managed by the built-in **SharedPreferences** environment in Android.

AppointmentListFragment

Interprets in a graphic way, by means of an **ArrayAdapter**(AppointmentListViewAdapter class), the list of the created appointments contained in the **AppointmentManager** as a **ListView**. Using the filter features granted by the ArrayAdapter class, it can show the appointments according to a date chosen by the user. this class handles even the event generated by the click and the long click of the user on one of the appointment, in the first case showing the main infos of the appointment and its position on a minimap, in the second giving to the user the possibility to delete the selected appointment.

ScheduleListFragment

Interprets in a graphic way, by means of an **ArrayAdapter**(ScheduleListViewAdapter class), the list of computed schedules contained in the **ScheduleManager** as a **ListView**. Using the filter features granted by the ArrayAdapter class, it separates the schedules in past schedules and current schedules (that are computed for the current day or in the future).this class handles even the event generated by the click and the long click of the user on one of the schedule, in the first case showing the main infos of the schedule clicked and the route that the user must follow to reach all the appointment composing the schedule on a minimap, and a list view showing the travel mean to take

to reach the next appointment. In the second giving to the user the possibility to delete the selected schedule.

NetworkManager

Class with one method used for check if the user is connected to internet.

MapUtils

Encapsulates a **GoogleMap** object, allowing to draw a schedule (meaning all the appointments as markers and the polylines linking the appointments as lines of different colors) or a list of appointments (meaning as a set of markers) on map. Note that this it's implemented as a static class instead of an extension of the **GoogleMap** class because **GoogleMap** it's declared as static.

5.3.1 API wrappers

As underlined in **DD: 2.2**, an indipendent set of components of our application is represented by the API wrappers. That is, they represents the way in which the system gather information from the external world. These informations, together with the internal data provided by the user, are the fundamental ingredients for the schedule computation. The use of the Adapter pattern (as introduced in **DD: 2.6**) is adopted for any class belonging to this subset of components, since we had these needs:

1. Retrieve data in a format that was requested by the other classes;
2. Separate the other classes from the specific external API service that was used;

The Adapter pattern fits perfectly these requests, leaving these part of the application totally open to modifications: if we will have the need to use another external source of data, will be enough to change the Adapter class, leaving the methods firm unchanged. Moreover, all these classes are Singletons: just one instance of a wrapper its needed. Last, since all the classes perform asynchronous calls, the results from these wrappers can be obtained only by extending the interfaces that they provide, and passing those objects to the methods that perform the calls (say, *getWeather*(..), *getTravelOptionData*(..), etc..), as suggested by asynchronous programming patters.

WeatherForecastAPIWrapper

This subcomponent relays on another wrapper which was already built for the DarkSky libraries, so, basically, we exploited the functionalities of this wrapper adapting the data for our purposes. In particular, by calling *getWeather*(..), we can retrieve the weather conditions for the next 48 in a certain location, starting on a certain date. This can lead to the possibility of caching the weather data, saving some API calls when not needed

MappingServiceAPIWrapper

This subcomponent relays on another 2 wrappers which were already built for the Google Places API and Google Directions API. Two features are offered:

1. *getStopDistance*(..): useful to retrieve all the transit stops that can be found in a certain range, with the relative distance from a specified location.

2. *getTravelOptionData*(..): useful to retrieve all the useful data about the travelling from a spot to another, specifying a deterministic starting time. These data are filled in a **TravelOptionData** object, which is a field available in every scheduled appointment.

TravelMeanAPIWrapper

This class it's not implemented because it deals with strike days and tickets, which are details that are not considered in our prototype, as explained in 2

Chapter 6

Testing

6.1 Server tests

6.1.1 Goal 1

Precondition and action tested: Registration with empty email
Expected behaviour: Incorrect email error response
Actual outcome: Incorrect email error response

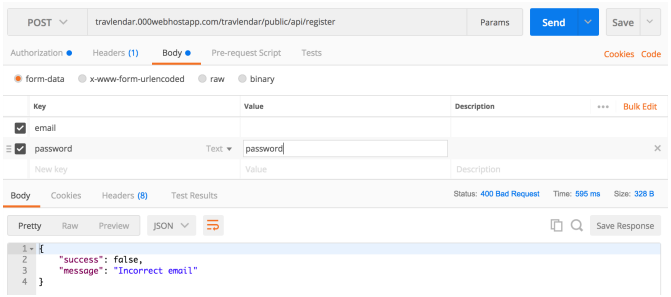

Figura 6.1: Graphical representation of the actual outcome

Precondition and action tested: Registration with empty password
Expected behaviour: Incorrect password error response
Actual outcome: Incorrect email error response

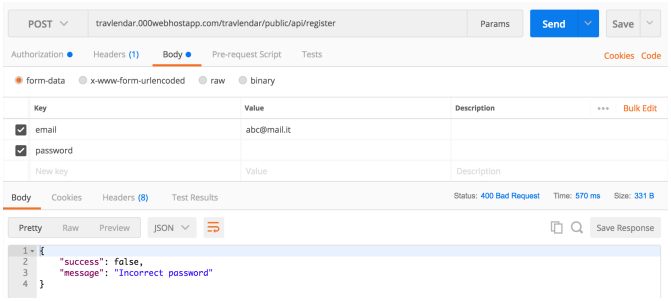

Figura 6.2: Graphical representation of the actual outcome

6.1.2 Goal 2

Precondition and action tested:	Login with wrong client credentials
Expected behaviour:	Wrong credentials error response
Actual outcome:	Wrong credentials error response

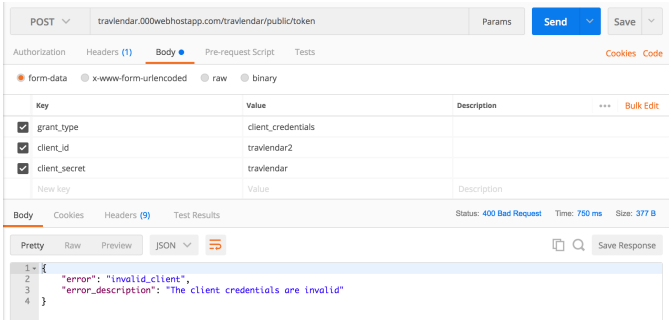

Figura 6.3: Graphical representation of the actual outcome

Precondition and action tested:	Login with wrong user credentials
Expected behaviour:	Wrong username or password error response
Actual outcome:	Wrong username or password response

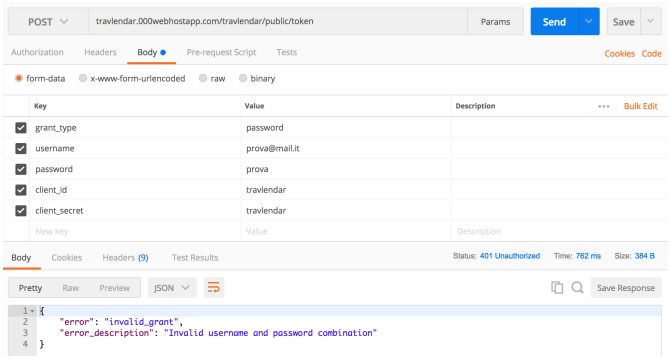

Figura 6.4: Graphical representation of the actual outcome

6.1.3 Other tests

Precondition and action tested: Access auth-protected api with correct access token
Expected behaviour: Response succesfully returns
Actual outcome: Response succesfully returns

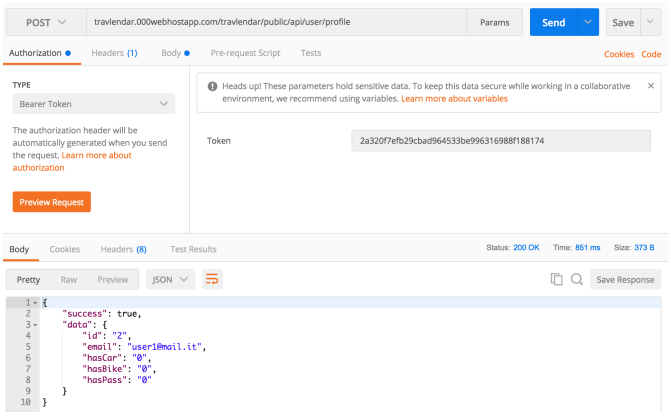

Figura 6.5: Graphical representation of the actual outcome

Precondition and action tested: Access auth-protected api with wrong access token
Expected behaviour: Wrong access token error response
Actual outcome: Wrong access token error response

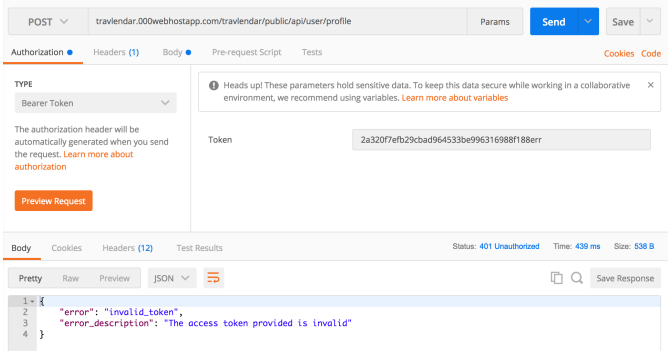

Figura 6.6: Graphical representation of the actual outcome

6.2 Client test

6.2.1 Goal 1

Precondition and action tested: Registration with invalid email
Expected behaviour: Incorrect email error message
Actual outcome: Incorrect email error message

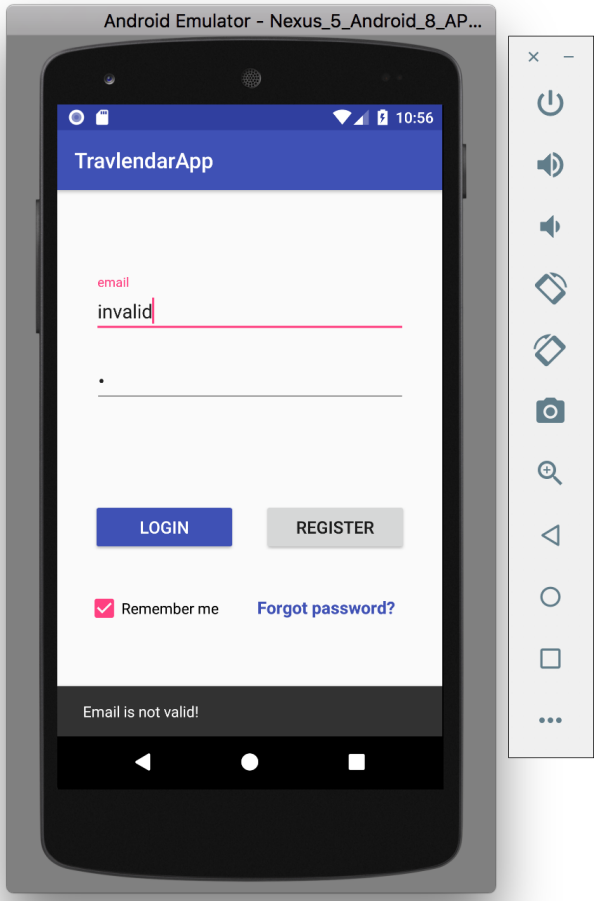

Figura 6.7: Graphical representation of the actual outcome

Precondition and action tested: Registration with non matching passwords confirmation
Expected behaviour: Non matching passwords error message
Actual outcome: Non matching passwords error message

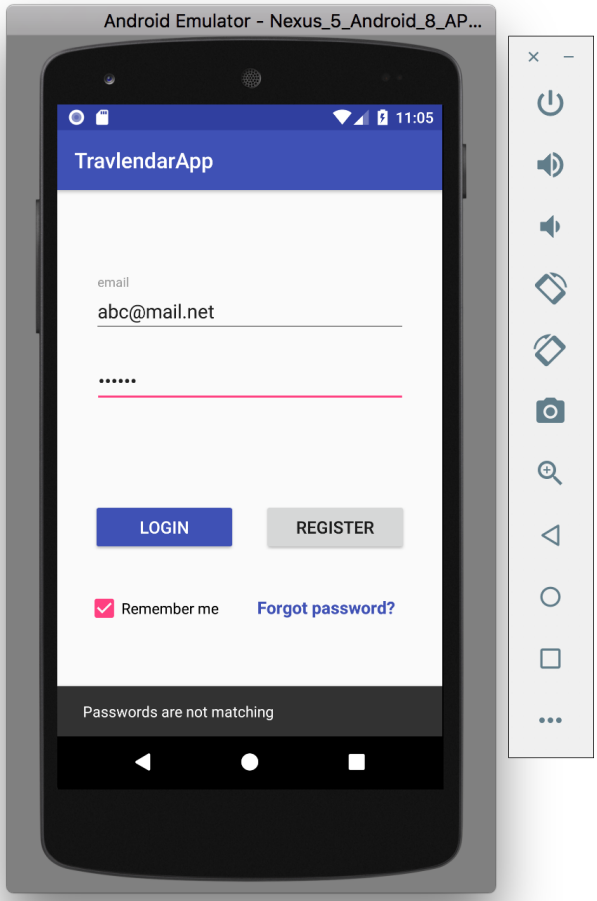

Figura 6.8: Graphical representation of the actual outcome

Precondition and action tested: Registration with correct user credentials and password confirmation
Expected behaviour: Registration succesfully done
Actual outcome: Registration done message

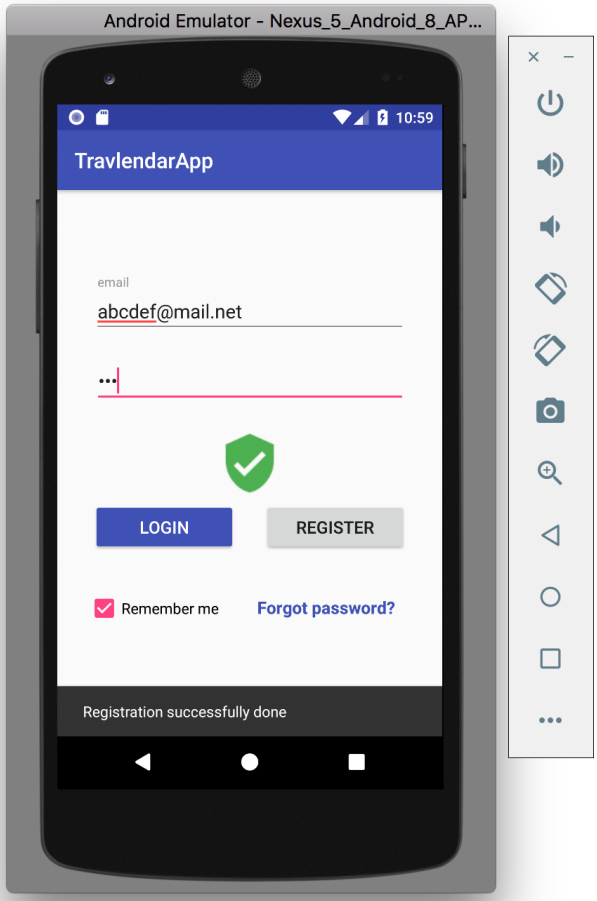

Figura 6.9: Graphical representation of the actual outcome

6.2.2 Goal 2

Precondition and action tested: Login with wrong user credentials
Expected behaviour: Wrong username or password error message
Actual outcome: Wrong username or password error message

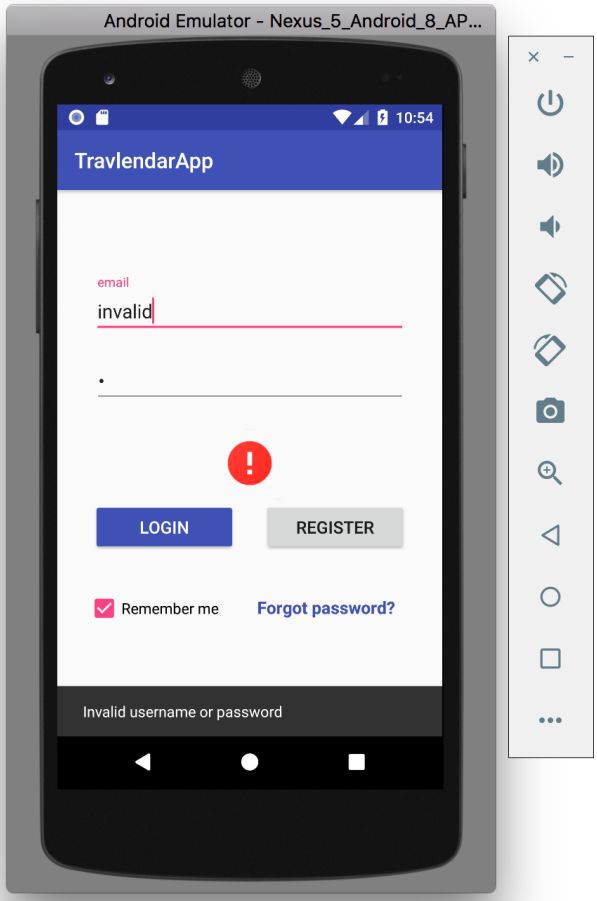

Figura 6.10: Graphical representation of the actual outcome

6.2.3 Goal 6

The system S.P.W. to create a valid schedule (1.3.6) of the user appointments when requested and display the scheduling result (1.3.9);

Precondition:	user has no computed schedule in his schedule list.
Action tested:	Creation of a new schedule.

Expected behaviour:

1. a click on the add schedule button is performed and the user is redirected to the schedule creation view.
2. the user selects the date in which he/she wants compute his/her schedule.
3. the empty fields are filled by the user.
4. the button for computing the schedule is clicked and the progress bar is shown.
5. the user is redirected to the schedule list and there, is added the new computed schedule.
6. with one click on the created schedule the schedule results are shown.

Actual outcome: The outcome is equal to the expected behaviour.

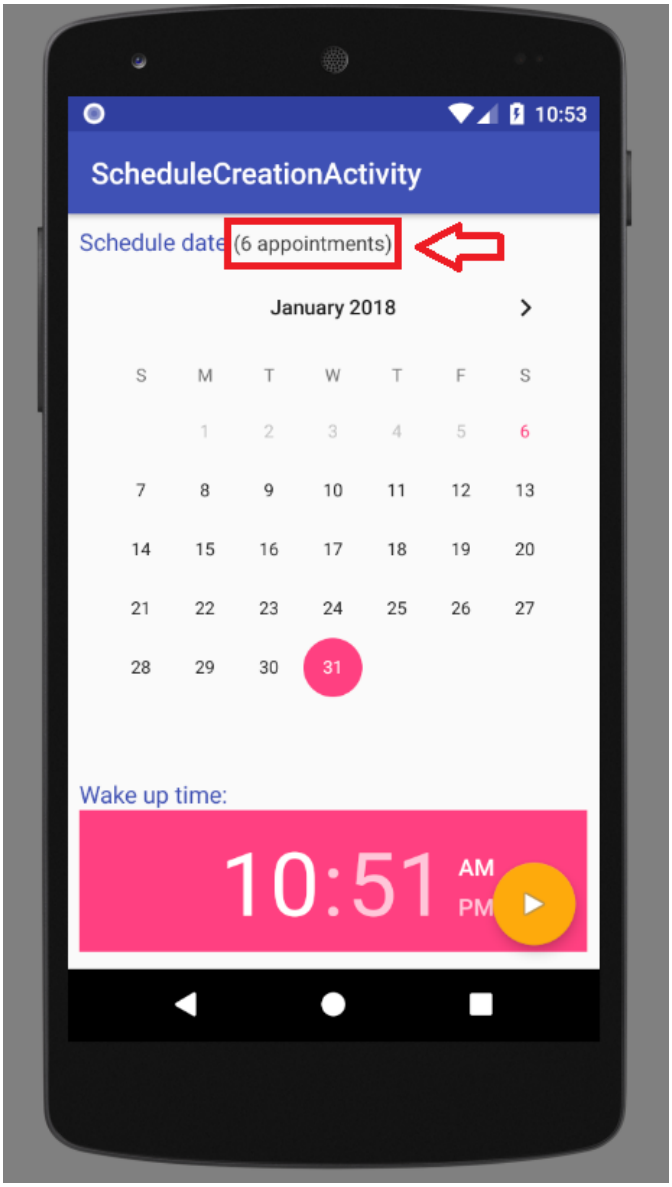

Figura 6.11: execution of point 2

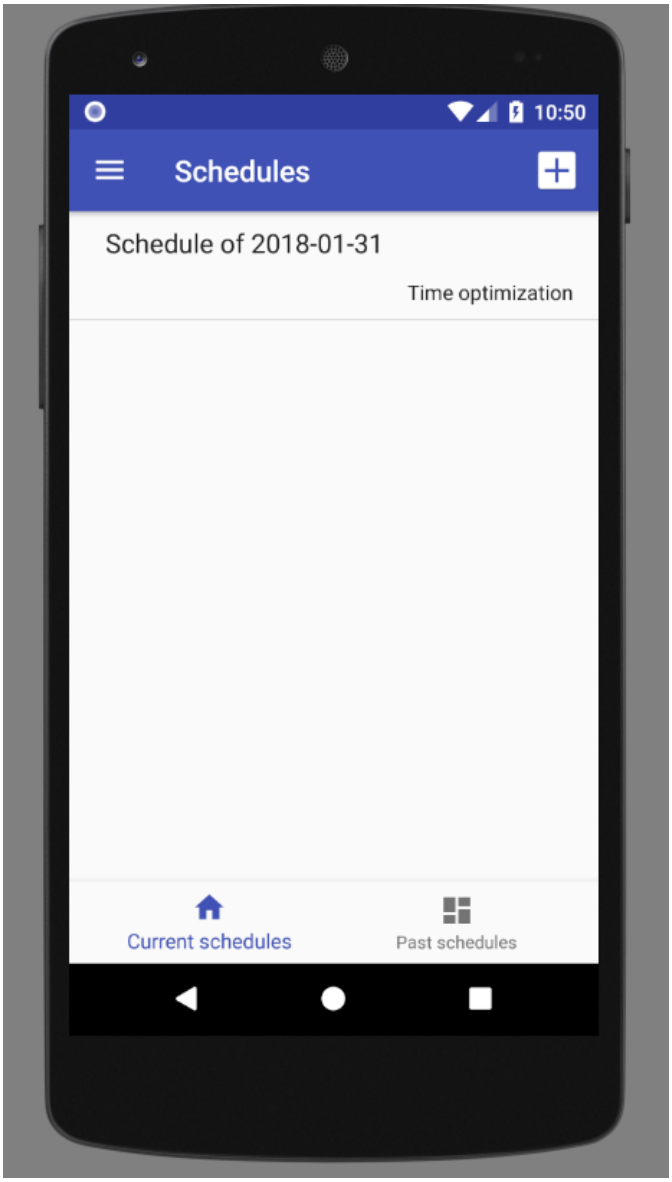

Figura 6.12: execution of point 5

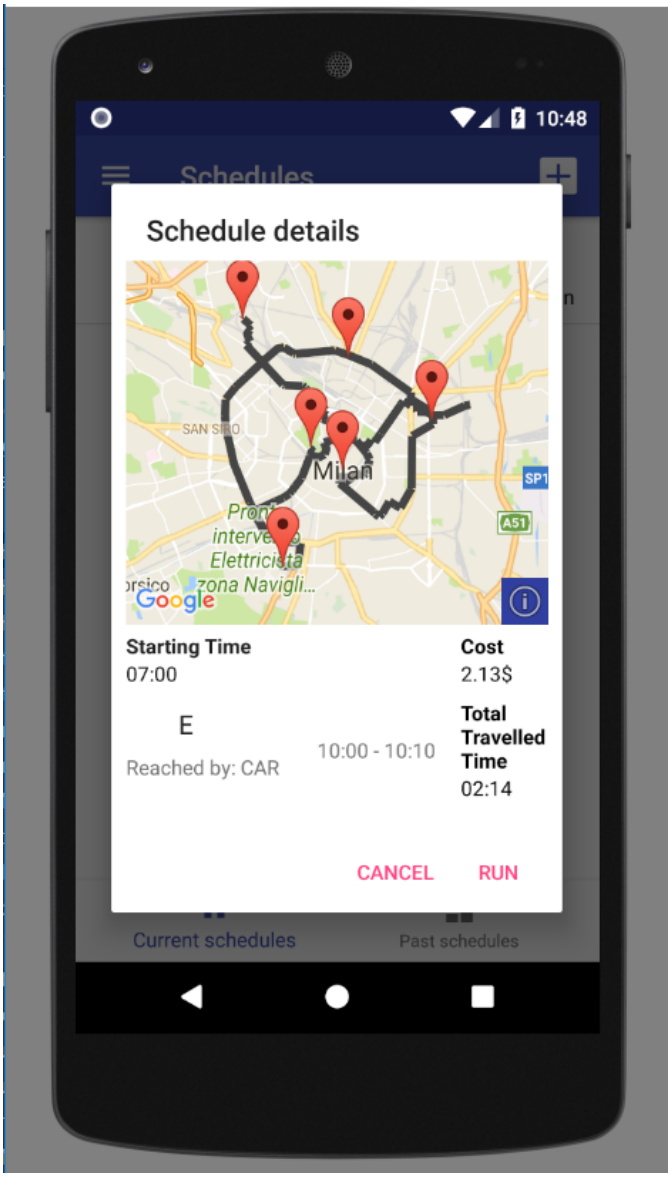

Figura 6.13: execution of point 6

Chapter 7

Installation instructions

7.1

Chapter 8

Effort Spent

- Federico Parroni: **hours**;
- Edoardo D'Amico: **hours**;
- Giovanni Gabbolini: **hours**.