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 is fully 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 it's 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 it's 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 it's 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 it's 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.

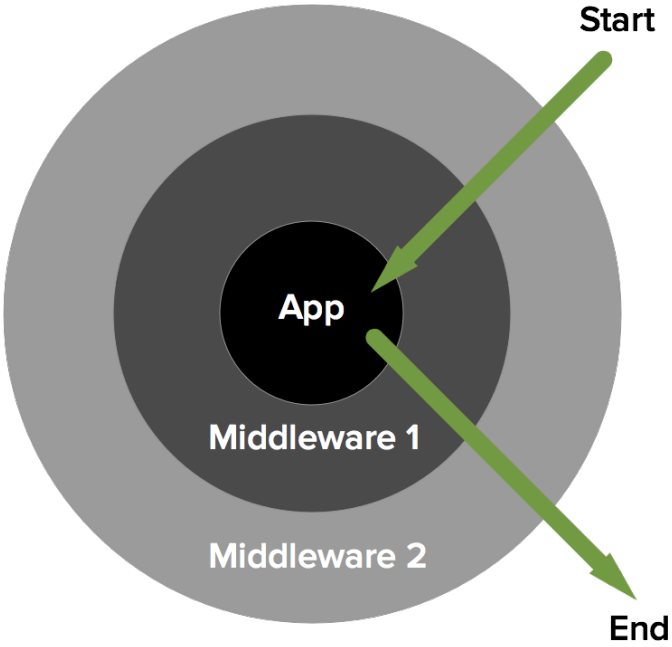

Figura 3.1: Middleware pattern

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

Code structure

4.1 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).

4.2 Client application code structure

Controller/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

Controller/ViewController/Login

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

Controller/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 temporaty 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

7. returning the schedule if feasible or null otherwise

Chapter 5

Testing

5.1 Goal 1

Precondition and action tested: bla
Expected behaviour: bla
Actual outcome: bla

5.2 Goal 2

5.3 Goal 3

5.4 Goal 4

5.5 Goal 5

5.6 Goal 6

5.7 Goal 7

5.8 Goal 8

5.9 Goal 9

5.10 Goal 10

Chapter 6

Installation instructions

6.1

Chapter 7

Effort Spent

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