Politecnico di Milano
AA 2017-2018

Software Engineering Project

TRAVLENDAR+

ITD

Implementation and Testing Document

Version 1.0

Marco Prosdocimi - 898395
Enrico Ruggiano - 900040
Giacomo Vercesi - 899928

Contents

Table of Contents

depth 2

Introduction

This document present all the informations about the first implementation of Travlendar+.

Requirements and Functionalities

With reference to the RASD and DD documents we are implemented some functionalities that we consider essential for the first version of Travlendat +.

Implemented funzionalities

Calendar Functions

We implemented a complete calendar system in which you can submit personal events and the user can access to his personal agenda. The events can overlap and we are implemented a partucular "flexible event" that can rearrange itself(see RASD fom more information). Everything is simple and intuitive just like mockups presented on the RASD.

Map Functions

We implemented a first version of Best Route algorithm which calculates the best route to reach the meeting based on the user preferences. The optimal path route in this first version works only in Milan and works with:

- By foot
- Car
- Bike
- BikeMy (bike sharing).

In future release may be available other veichles and root path calculation (example: assemble different gpx route with different vehicle)

The meeting and the start locations need to be specified in the map on every personal event submission. The map is much user friendly as possible and it presents drag and drop markers to help the user to select the appointment's positions. The map can geolocalize the User, depending on the browser configuration and User choice to be geolocalized.

User Profile Functions

We implemented a complete user profile as we specify in the RASD. This will used to collect data like personal preferences on types of vehicle which will be used for calculating route path.

Alarm System

The first version of the alarm system send a notify in the browser when the user need to start travel for reach the meeting just as info message.

Future Features

Feature not implemented (some of them may be under comments on the code)

Calendar Functions

- Adding support to different timezone and locale time setup.
- Adding festivity and weather information using exteranl Api, which could give extra information to user and to optimal path server.

Map Functions

- Adding street names support on the meeting and starting location with external database or API.

- Adding support to entire region or support other cities (ex: Rome). (it Needs LOT OF RAM)
- Using different API for geolocation so to be more precise.
- The User can specify the start positions as the meeting position of the last event.

Alarm System

- Adding Email warning message support if allowed by the user. (Need some module on the EE server);
- Adding Telephone warning message support with sms. (Same)
- Adding different type of warning message and more User Friendly. (See RASD for different types of warning)
- **New Alarm System trigger events:**
  - The Alarm system notify the user when: he submit an event that overlaps with another.
  - The Alarm system notify the user when he try to submit an event than he can't reach in time.

Registration

- The registration verify user with email authentication sending message with an url/password/hash.

Adopted development frameworks

Frontend programming languages

Travlendar was projected to be easy, simple, UI friendly, client-server application and cross-platforming. So why not using Web Programmation and Framework?

Frontend is based on ModelViewControl and EventDispatching (see DD) and javascript libraries like JQuery are perfect for this scope. Also we didn't wanted to reinvent the wheel so we needed to have OpenSource, Full Documented and Maintened Libraries for the project. Also Javascript,html and CSS are perfect for client-server application and are used for Web Programming.

But the key is that we needed a cross-platforming language whose code is the same for a Broswe
thanks to OpenSource libraries like Electron and Cordova you can reuse your javascript to build a complete android/ios/mobile application without changing a line of code.

Other key reason is that Javascript is a simple and concise language with a vast library and API support which can let you create a full working web application with ease.

Also Html5 is great to build UI layout, and CSS helps you to have a layout much responsive as possible.

Libraries used:

- FullCalendar API. https://fullcalendar.io/
- OpenLayers v3 API. https://openlayers.org/

- Moment http://momentjs.com/
- Toast https://codeseven.github.io/toastr/
- Material Design lite: https://getmdl.io/
- JQuery https://jquery.com/

Adopted programming languages

Python

Python is a dynamically typed programming language that is used worldwide in a variety of applications. We chose this language over others for the following proprerties:

- dynamically typed: allows the construction of complex datatypes with ease and removes the burden of memory managment thanks to its garbage collector
- construction of plugin-frameworks such as those needed for the optimal route and scraper modules are trivial to implement
- vast and production-ready collection of libraries, such as those needed to deploy web frameworks and produce http requests
- interpreted: it allows for rapid prototyping of the application, does not need to be recompiled with every version and the container build process is very streamlined
- can be trivially scaled for heavier workloads with appropiate libraries

On the other hand it presents the following shortcomings:

- Due to the dynamically typed nature run-time errors are prone to happen and thus the codebase requires stricter coverage

Back End Framework

1) **unittest: Python Framework for tests, is very similar to other test libraty like jUnit.**
   https://docs.python.org/3/library/unittest.html.
2) **flask: Is a micro web framework written in Python and based on the Werkzeug toolkit**
   http://flask.pocoo.org.
3) **flask_cors: A Flask extension for handling Cross Origin Resource Sharing (CORS), ma**
   https://pypi.python.org/pypi/Flask-Cors
4) **rethinkdb**: RethinkDB is open-source, scalable JSON database built from the ground up for the realtime web(see below for more information).
5) **jsonschema: JSON Schema is a vocabulary that allows you to annotate and validate JS**
   https://pypi.python.org/pypi/jsonschema.
6) **schedule**:library that allows to program the running of a function at a specified time, used in the scraping module to help modules that need to scan a certain informations source everyonce in a while https://github.com/dbader/schedule
7) **BeautifulSoup**: python library to manipulate html pages, provided in the scraping module to aid the modules in data acquisition. https://www.crummy.com/software/BeautifulSoup/
8) **requests**: python library that semplifies making even the most complex HTTP requests. http://docs.python-requests.org/en/master/

9) **haversine**: python library to calculate the distance between 2 gps coordinates based on the haversine formula. https://github.com/mapado/haversine
10) **gpxpy**: python library that allows the manipulation of gpx files https://github.com/tkrajina/gpxpy

RethinkDB

The choice for using RethinkDB as the backend database was based on various factors. First and foremost the fact that with both the user and data database were eterogeneous enough that a NoSQL database seemed more suited for the job, moreover since we were settled on the use of json throughout the codebase it seemed natural to find a database software that natively supported json. Finally RethinkDB supports the streaming of changesets to the clients, avoiding the use of polling.

Thre main disadvantage to using RethinkDB and the underlying json model is that the information stored in unstructured and thus it is easier to have malformed data stored on the database. Also due to the NoSQL nature of the server performance is slightly inferior compared to a SQL database, however due to rethinkdb's support for sharding the leverages this deficency quite well.

Structure of the source code

The code is structured in the following way (starting inside the **implementation** directory):

- `*.Dockerfile` docker build files
- `build_valhalla.sh` and `build.sh` helper scripts to build the docker images
- `docker-compose.yml` describes the container order and network linking
- `endpoint` contains the code regarding the HTTP endpoint
- `web_interface` contains the html, javascript and css files that present the website and interact with the endpoint server
- `optimalroute` contains code pertaining to the optimal route server that is used by the endpoint when asked for routes
- `optimalroute/route_plugins` contains the plugins that provide the various pathing algorithms
- `scraper` contains code for the scraper module which gathers data from the web to be used by the optimalroute server
- `scraper/modules` has the individual modules that gather data for individual services
- `valhalla_server` contains the dockerfile to assemble the valhalla server together with the `milan_map_full.pbf` which is a PBF-encoded map of the Milan metropolitan area

Testing

WebInterface Test

Testing it was not simple. After trying to use QUnit library which let you write unit test to javascript code we decided to use other apporoach. This is because we used a lot of JQuery code which dispatches on user events (like button clicks, writing input field...) which is intestable with standard unit test. So decided to use Test Automation technique, which is great for web application. After using for a while Phantomjs without a good result we adopted Selenium Test Automation to reach our goals.

Thanks to Selenium is possible to create a custom user 'Marionette' navigation behaviour to test WebInterface functionality (like if the html div are correctly updated after a http post). Everything is written in Java with Selenium java API integrated with Junit library. Also when the test is finished it returns a clear html log file in which there are the test results.

Tools:

- Selenium API: http://docs.seleniumhq.org/
- see /webInterfaceTest/README.md to setup the test environment.

Back End test

we test the most important functions of the back end using the python framework unittest. These unit tests check the right work of the back end functions and their interaction with the database, so the tests need a rethink db to work.

instruction for test executions: start a rethinkdb session and execute with pyton 3.6 the file TestSuite.py in the endpoint folder.

System test

We use jmeter for testing all API endoint. In these case we ipotize the right work of the back end guaranteed by the "Back End tests". We test the API doing some Post and Get request and make some assert on the response.

1) In the Thread Group: Registration,UserProfile,Event we test all API endpoint by do the correct Posts and make some assertion on the response.
2) In the Thread Group: Post_missing_information we do some malformed post with some missing essential information. We expect a Bad Request response.
3) In the Thread Group: Wrong_post we test a possible post that try to modify random event that not belong to the user. so in this case we expect an Illegal Accession response.
4) In the Thread Group: Illegal_token we test an incorrect login and some post with incorrect token. In tthat case when the Server see that the token is incorrect stop the computantion and send an error message

instruction for test executions: start the docker environment and open with Jmeter the file API_test.jmx in the system_test folder.

Installation instructions

(note: these installation instructions are for linux-based operating systems, use in other OSes might require slight variation of the commands used)

The installation instructions are as follows:

1) Install **docker** as explained on https://docs.docker.com/engine/installation/
2) Install **docker-compose** as explained on https://docs.docker.com/compose/install/
3) Start the docker service using (might depend on the system used)

   ```
   sudo systemctl start docker
   ```

1) make sure you are in the **implementation** directory and run:

   ```
   sudo ./make_valhalla.sh
   ```

   this will build the valhalla server needed to get routing information

5) then run

   ```
   sudo ./build.sh
   ```

   this needs to be run every time there is a modification to the codebase

6) Finally run `sudo docker-compose up`

   this will bring up the entire application with all its services in one command.
   The program is browsable by going at `http://localhost`