Travlendar+ Design Document

Sinico Matteo, Taglia Andrea

Version 1.0

November 2017

Table of Contents

1. INTRODUCTION

A. Purpose

B. Scope

C. Definitions, Acronyms, Abbreviations

Application: by this term we refer both to the web-app and the overall system.

D. Revision history

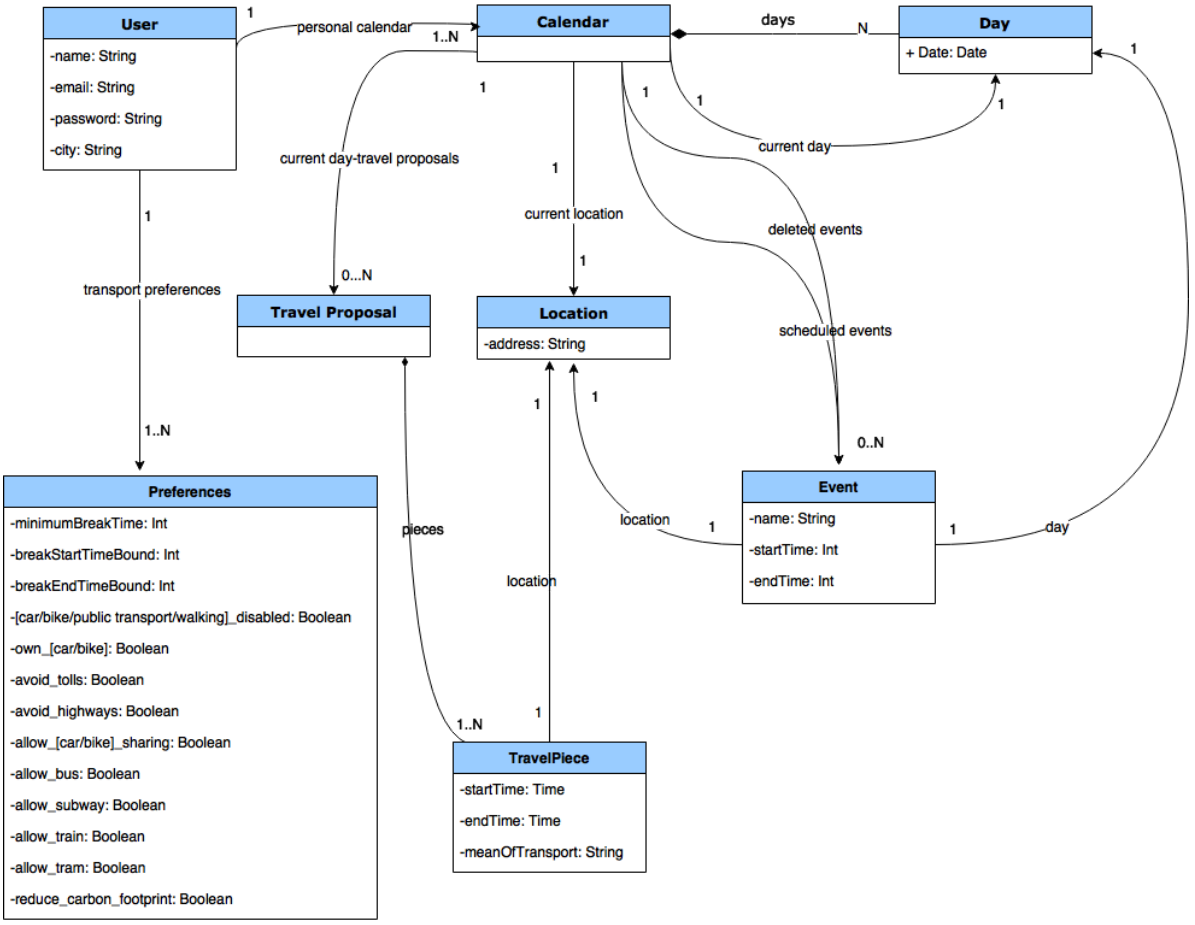
E. Reference Documents

F. Document Structure

2. ARCHITECTURAL DESIGN

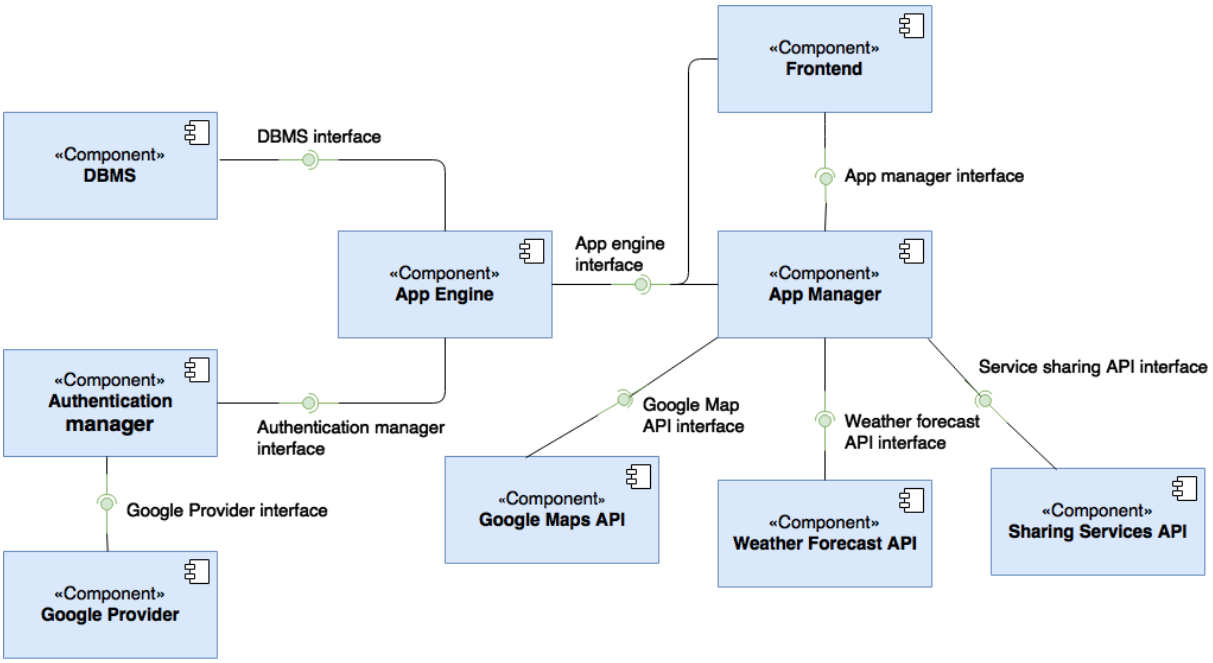
A. Overview:

In this section is addressed the architectural design of Travlendar+. In the following subsections are depicted some of the most relevant shades of the system-to-build. Therefore is given an insight over the software components of the system and the interfaces interleaved among them. Is also given a runtime view of the system to explain its behavior and is addressed the deployment of the software component to have a static view of what the system will looks like.
It also provided a more detailed class diagram with respect to the high level one provided in the RASD document:



B. High-level components and their interaction

In this section there is the component diagram of the application, that describes, at high level perspective the component of the application and the interactions between them.



B.1 DBMS

This is the data-base management system of the application that will manage all the data coming from the user and the application itself. These data in details are: user profile information (first name, last name, username, password, e-mail, sex, date of birth); user calendar (appointments) and user preferences (car, bike, public transport, walk, carboon footprint, break). The DBMS will provide an interface to allow the application to perform queries over the data.

B.2 Authentication Manager

This is the component of the application responsible both of the procedure of signing-in and signing-up. All the basic features, like password recovery, secure authentication will be guaranteed by the use of “Google Sign in”, which allows the user to sign in with its google account, without creating a new unuseful account. In order to do so the authentication manager needs an interface with Google sign-in, and has to provide an interface to the app engine.

B.3 Google sign-in provider API

Google Maps API Component takes care of the HTTPS requests and responses needed to allow the user to log in with his google account. API documentation available at https://developers.google.com/identity/sign-in/web/devconsole-project

B.3 App Engine

This component is the backend of the application, it’s the unique point of access to the data stored in the data-base and to the authentication manager. Thus it takes the HTTPS requests from the application and it translates it either in queries to the DBMS or in authentication request to the authentication manager. Then it retrieve the results to the rest of the application. So it’s the middleware between the data layer and the application logic layer.

B.4 App Manager

This is the core of the application, hence it’s responsible of receiving, either dispatching to the right component (app engine, google maps API, weather forecast API, sharing services API), or resolving all the request coming from the frontend of the application which are: sign-in, sign-up, edit profile information, edit calendar, evaluate day-travel proposal, edit preferences, show travel, (check the RASD document section B.2, B.3, and B.4 to have a complete view of this functionality). In order to do so it needs interface with all the component stated before, and it has to provide an interface to the frontend of the application.

B.5 Google Maps API

Google Maps API Component takes care of the HTTPS requests and responses needed to compute the estimated time it would take to get from one place to another by different means of transportation. API documentation available at https://developers.google.com/maps/documentation/directions/start

B.6 Weather Forecast API

Weather Forecast API Component takes care of the HTTPS requests and responses to the open weather service (API documentation at https://openweathermap.org/api) for weather forecasts data.

B.7 Sharing Services API

Sharing services API Component takes care of the HTTPS requests and responses to a urban mobility aggregator service that wrap in it all the most important sharing service available at the moment. This component will be bought choosing among the most known providers of this kind of service, of course, taking into account the solution which best will fit in the implementation phase.

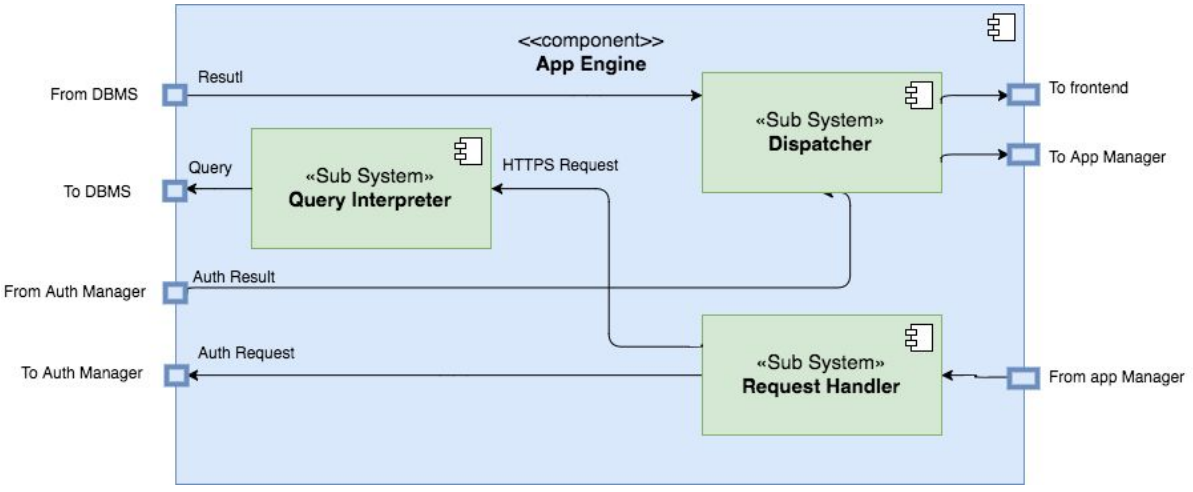
B.8 Frontend

This component it is the Graphic User Interface of the application. To see an example of it see the section 4 of this document. It requires interfaces both to the app manager and the app engine in order to perform request and to retrieve data and visualize it.

C. Component view

In this section is provided an insight of those components whose behavior must be refined to have a comprehensive view of the system to build.

C.1 App Engine



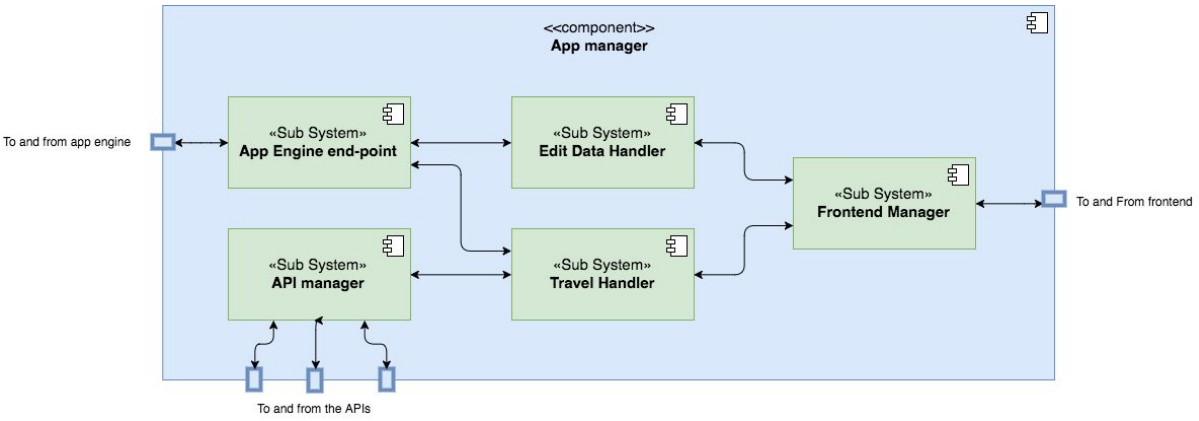
The app engine component is composed of 3 sub system:

- Request Handler
- Dispatcher
- Query Interpreter

that are three piece of software which accomplish different task.

The **Request Handler** receives the HTTPS requests from the app manager and decides either to direct these requests to the query interpreter or to redirect them directly to the Authentication Manager, simply reading the header of the HTTPS requests.
The **Dispatcher**, the other way round, receives the results from the Authentication Manager and from the DBMS and then sends them either to the frontend or to the app manager, respectively if they are information that update the view of the application or if they are just requested by the app manager to compute the day travel proposal.
The **Query interpreter** receives the HTTPS requests from the Request Handler and then translate them into queries understandable by the DBMS.

C.2 App Manager



The App manager is composed of 5 sub system:

- FrontEnd Manager
- Edit Data Handler
- Travel Handler
- App Engine end-point
- API Manager

each of them is a different piece of software which accomplish a different task.
The **FrontEnd Manager** is responsible of receiving the requests from the frontend and dispatch them to the correct sub system: either the edit data handler if the request it's a query or a modification of the data, the travel handler if it's a request to compute the day travel proposal either default or not, or the App engine end-point if it's an authentication request.
The **Edit data handler** handles all the query and data modification request (edit calendar, edit profile, edit preferences, show travel)[1]. The sub-system is responsible of checking the feasibility of the data inserted or modified:

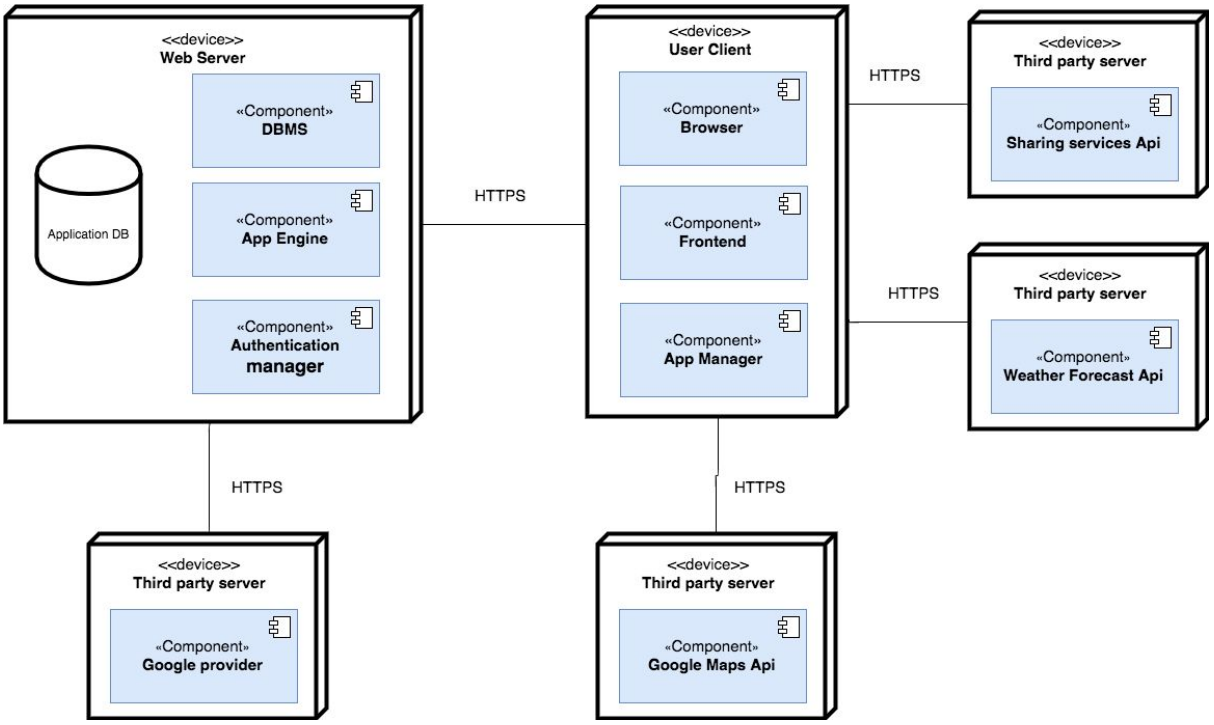
- edit calendar: the date, the starting and the ending hour of the appointment have to exists on the calendar; if the appointment day is the current day the start hour has to be greater than the current hour.
- edit profile: the date of birth has to be an existing day on the calendar.

The **Travel Handler** is the real core of the application because is the component responsible of the computation of the day travel proposal. Thus it has to be able to access both the DBMS and the APIs and also to the frontend to retrieve the results. A detailed explanation of

[1] please refer to the RASD document (sections 3.B.2 and 3.B.3) to have a detailed view of all the functionalities offered by the application.

the algorithm used to compute the day travel proposal is shown in the section 3 of this document.
The **App engine end-point** both wraps the requests from the app manager components in an HTTPS request ready to be sent to the app engine and, the other way round, unwraps the HTTPS responses in an understandable format for the app manager components.
The **APIs manager** redirects to the correct API the requests from the travel handler.

D. Deployment view



The criterion which has lead the deployment is dictated by the choice of adopting a classical two tier client-server implementation[2].
All the APIs are in a stand-alone block because we see them as black-box to which sending requests and from which receiving responses to utilize in the other component of the application, all the requests and the responses are sent and received by HTTPS requests.
The core of the application ,and obviously the frontend, are deployed in the user client, while the data management and the security features are kept together in a unique tier in order to reduce the cost and to improve the performance of the application. The two tier communicate via standard HTTPS protocol.

E. Runtime view:

You can use sequence diagrams to describe the way components interact to accomplish specific tasks typically related to your use cases

[2] please refer to section 2.G of this document to have a detailed explanation on the advantages offered by a two tier architecture.

F. Component interfaces

G. Selected architectural styles and patterns:

Please explain which styles/patterns you
used, why, and how

H. Other design decisions

3. ALGORITHM DESIGN:

The most relevant algorithms are described below in terms of input necessary for the algorithm to be executed, output that it should return, logic steps it should follow during execution. These high level descriptions would then become the algorithms specification for the developers team.

A. Compute Travel Algorithm

The actual core of the whole application, the algorithm looks for the optimal route to get the user from one appointment to another taking into account his personal preferences.

INPUT :

- List of user personal Events from current day, each one including:
  - event location
  - start time
  - end time
  - name
- User's break time preferences (all integer values):
  - minimum time
  - lower start time bound
  - upper end time bound
- User's travel preferences (all binary values):
  - [car/bike/public transport/walking]_disabled
  - own_[car/bike]
  - avoid_tolls
  - avoid_highways
  - allow_[car/bike]_sharing
  - allow_[public transport - bus]
  - allow_[public transport - subway]
  - allow_[public transport - train]
  - allow_[public transport - tram]
  - reduce_carbon_footprint

EXECUTION :

1. prepare output list of size Events + 1
2. n = 0

3. for each couple of Events i,j do //note that first and last event location will be home
    a. if (weather forecasts expect rain for that time) disable walking and cycling
    b. if (not car_disabled && (own_car || allow_car_sharing)) API requests with origin = i.location, destination = j.location, arrival_time = j.startTime, mode = driving, avoid preferences => create TravelPiece with info received and if time to start travel higher than current output_list[n], substitute TravelPiece
    c. same for bike. if (reduce carbon footprint = true) give priority
    d. same for public transport -> append transit options based on preferences
    e. same for walking. if (reduce carbon footprint = true) give priority
    f. n++

OUTPUT :

- List of Travel Piece which will be interleaved with Events to make a full day schedule. Each Travel Piece is composed of:
    - mean of transport
    - starting location
    - time to start travelling

4. USER INTERFACE DESIGN:

Provide an overview on how the user interface(s) of your system will look like; if you have included this part in the RASD, you can simply refer to what you have already done, possibly, providing here some extensions if applicable.

5. REQUIREMENTS TRACEABILITY:

Explain how the requirements you have defined in the RASD
map to the design elements that you have defined in this document.

6. IMPLEMENTATION, INTEGRATION AND TEST PLAN:

Identify here the order in which you plan
to implement the subcomponents of your system and the order in which you plan to integrate such subcomponents and test the integration.

7. EFFORT SPENT:

8. REFERENCES

A. Bibliography

- IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, which can be retrieved on the beep page of the course.