Travlendar+

Design Document

Calzavara Filippo, Filaferro Giovanni, Benedetto Maria Nespoli

Version 1.0.0

Contents

1 Introduction

1.1 Purpose

The Design Document (DD) contains a functional description of "Travlendar+"'s System.

This document explains each component that will be inserted into the system, its architecture and the design patterns that will be implemented to ensure that all the requirements are satisfied.

Components will be described both at High Level and more in depth, illustrating and explaining all the subcomponents every component is made of.

The reader of this document will get a clear idea about its architecture (hardware and software) whether he wants to have a detailed description of the system or a more general one.

1.2 Scope

To learn about the Scope of the project look to the "1.2 Scope" section of the RASD document.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Definition	Explanation
Appointment	A period of time in which something take place at a certain time
Travlendar+	The name of the platform to develop
Event	A synonym for appointment
System	A synonym of Travlendar+
Up Coming Event	A particular event that is expected to occur soon
Up Next Event	A synonym of Up Coming
User	A potential utilizer of this project
Ride	A service performed by a ride-sharing company and Cabs
Mockup	A scale or full-size model of a design or device
RESTful API	API that follow the REST paradigm
Direction API	The name of Google Maps™ API service

1.3.2 Acronyms

Acronym	Explanation
GUI	Graphic User Interface
ETA	Estimated time of arrival
API	Application programming interface
RASD	Requirement Analysis and Specification Document
PNR	Passenger name record
QR	Quick Response Code
OS	Operative System
JSON	JavaScript Object Notation
REST	Representational State Transfer
APNS	Apple Push Notification Service
CRUD	Create, Read, Update, Delete
ER	Entity Relationship
TDD	Test Driven Development

1.3.3 Abbreviations

Abbreviation	Explanation
App	A synonym of Travlendar+
[Gn]	N-goal
[Dn]	N-domain assumption
[Rn]	N-functional requirement

1.4 Revision History

- 1.0.0 - Initial Version (13/11/2017)

1.5 Reference Documents

- RASD document previously delivered.

1.6 Document Structure

The paper includes eight areas. The first one, is composed by the introductory information provided in order to give an orientative view of what this document is about.

The second one provides an in-depth description of each Architectural Design aspect which reflects the decisions that the team made.

The third section is about the algorithms that are designed to make the full system work.

Then a "User Interface Design" section will go through the design decisions.

Section five provides informations about how the requirements defined in the RASD document map to the design elements defined in this document.

The sixth section will identify the order in which the subcomponents of the system will be implemented and tested.

Finally, a seventh and a eight section allows the reader to learn the effort spent on this project by each team member and the references made in the whole paper.

2 Architectural Design

Figure 1 represents Travlendar+'s architecture, based on a two physical tiers system with distributed application and data. In order to maintain higher standard in terms of offline reliability, the client application is provided of a presentation level, application logic and data management that merely synchronizes and fetch updates from the server through a non-faulty connection.

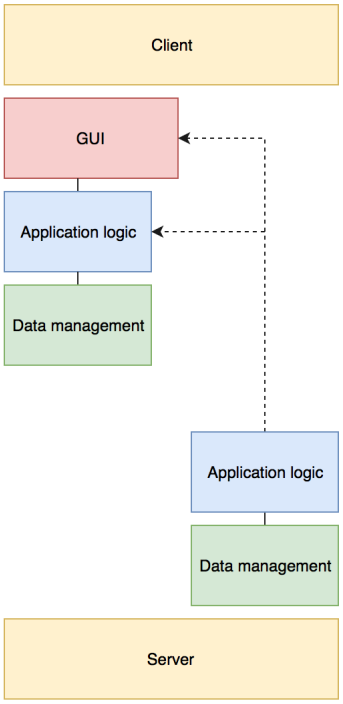


Figure 1: General tier architecture for Travlendar+. An application logic and data management layer is provided to the client application for offline reliability

Nevertheless, most of the computation burden such as the scheduling, is performed sever-side.

The following subsections are presented as follows:

1. The first section, *High-level components overview and their interaction*, contains details on how the communication works and how the server is structured
2. The second section describes the components of the application logic, including the external services called via API and the ER diagram
3. The *Deployment view* section contain a diagram with the interactions between several components
4. The *Runtime view* chapter includes more detailed sequence diagram of most important interaction with the system
5. *Component interfaces* section contains information regarding the interface to be built in order to make intelligible different system
6. Section *Selected architectural styles and patterns* contains the major settings of the implementation of the system
7. Finally, a chapter named *Other design decisions* describes further decision detail taken into account during the developing of this document

2.1 High-level components overview and their interaction

The mobile application located on the client's phone connects to the central server by sending REST Api calls in asynchronous mode with a separated thread. The main server replies back with the information requested using the JSON encoding, which is reinterpreted locally by the application. An example of this interaction is reported on Figure 2.

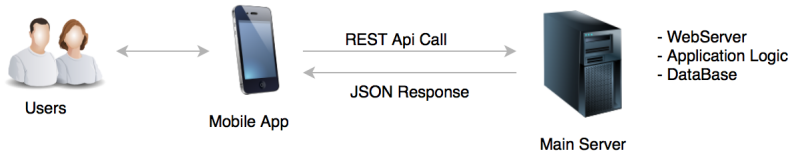


Figure 2: General architecture

The engineering team of this project has identified Heroku™ Cloud as the most suitable platform to host the main server due its performance and scalability features. As shown in Figure 3, a centralized server is composed by three entities: the web server, the application logic and the database. The first is in charge of dealing with all HTTP/HTTPS API requests in a consistent and systematic way. It also communicates with the application logic, which is the second layer, also called Mobile App Services as it contains the logic to run the computation. Finally, the last layer is the database which stores the dataset making it available to the application logic.

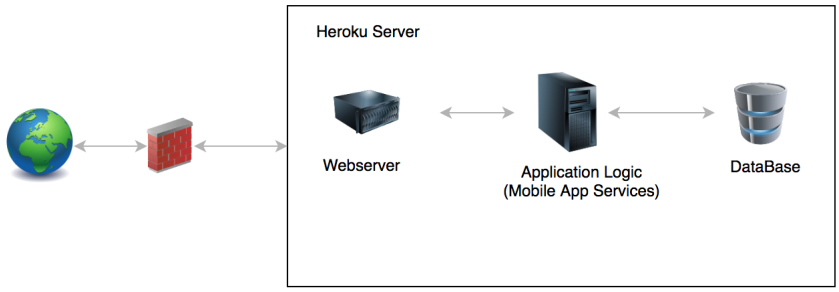


Figure 3: Central Server relies on Heroku Cloud. Its representation includes three layer connected to each other

All three active parts hereinbefore described are could be located into several machines: replication and load balancing issues are transparent services provided by the Heroku platform

2.2 Component view

A high level component view and related interfaces are represented at Figure 4. The system is composed of a major component, namely the *Mobile Application*, that interacts with two subsystems, *Scheduling Services* and *User Services*.

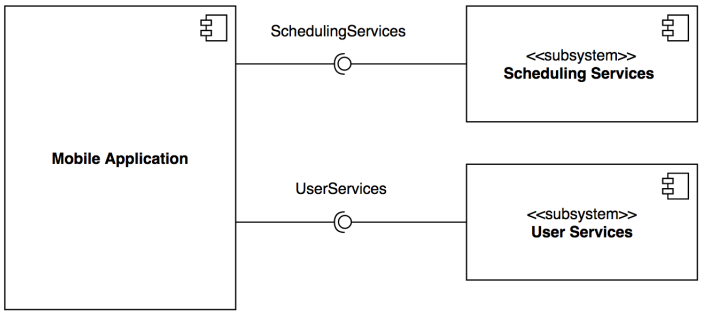


Figure 4: The high level component view of the system

Each module that compose the subsystems is interfaced with the DataBase (specifically the DBMS) in order to let each part work independently and simultaneously.

2.2.1 Component view of the Scheduling Services

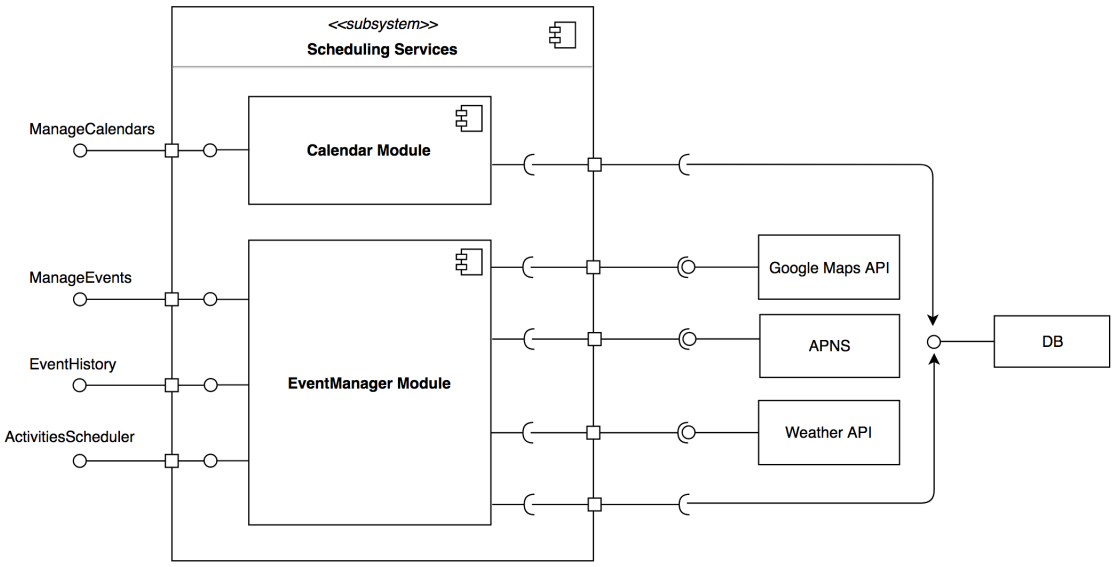


Figure 5: The Scheduling Services: includes details on

The *Calendar Module* works as calendar manager and offers the methods to edit, add and remove calendar. It is its responsibility to check calendar constrains such as duplicates. The *Event Manager* module allows the creation and managing of the events and handles the rescheduling when necessary (see section 3.1 for more details). It also offers the interface that allows to fetch information about previous events. This module is connected to Google Maps API, Weather API provided by OpenWeatherMap and to Apple Push Notification Server, that handles the notification system towards the mobile application.

2.2.2 Component view of the User Services

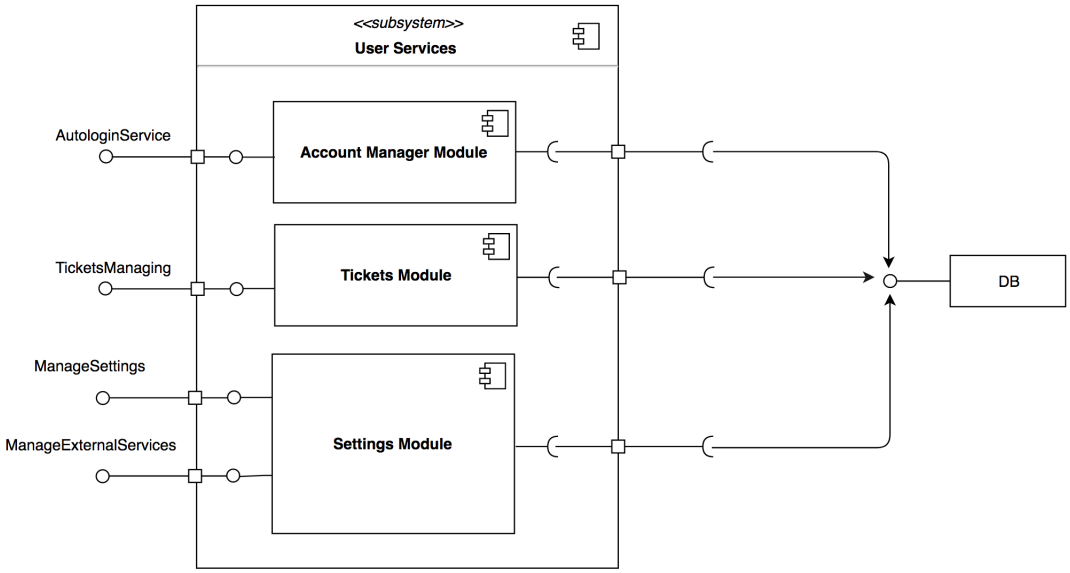


Figure 6: User Services in detail

User services contains an *Account Manager Module* that handles the login/auto-login functions (cf. Section 3.2.1 RASD document). The *Tickets module* provide tools to store the ticket provided by the local transit company into the database. Moreover, a setting module provides tools for setting managing such as the eco-friendly mode or the external services available.

2.2.3 Entity relationship diagram

The following diagram provides a graphical representation of the Database Schema that will be adopted by the system

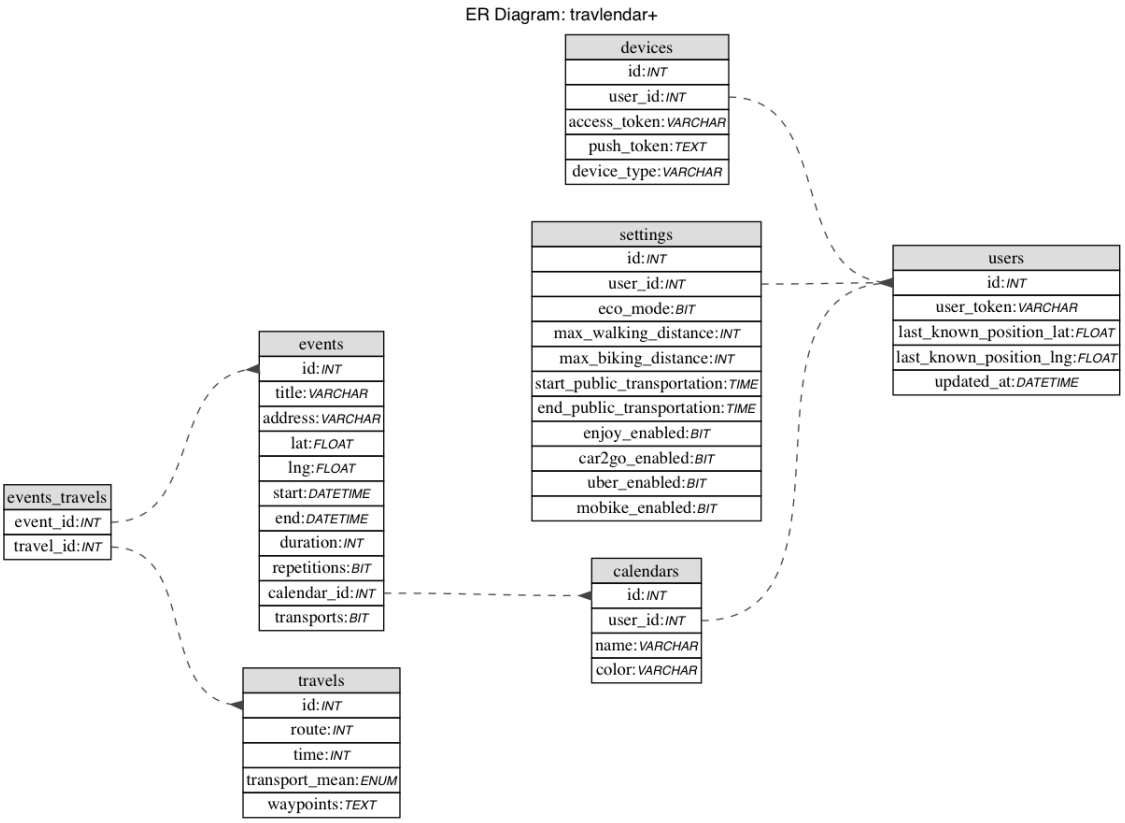


Figure 7: ER Diagram of the database

Further clarification regarding the schema:

- The column *transports* is of type BIT(5) for simplicity. If the bit is 1, the corresponding mean of transportation is enabled for that event. From the leftmost to the rightmost bit, there are: Walking, Biking, Public Transport, Sharing Services and Car.
- The column *repetitions* is of type BIT(7) for simplicity. If the bit is 1, the event will repeat for the corresponding day of the week. The leftmost bit is for Monday, and the rightmost is for Sunday.
- The other BITs are BIT(1) meaning TRUE for 1 and FALSE for 0.

2.3 Deployment view

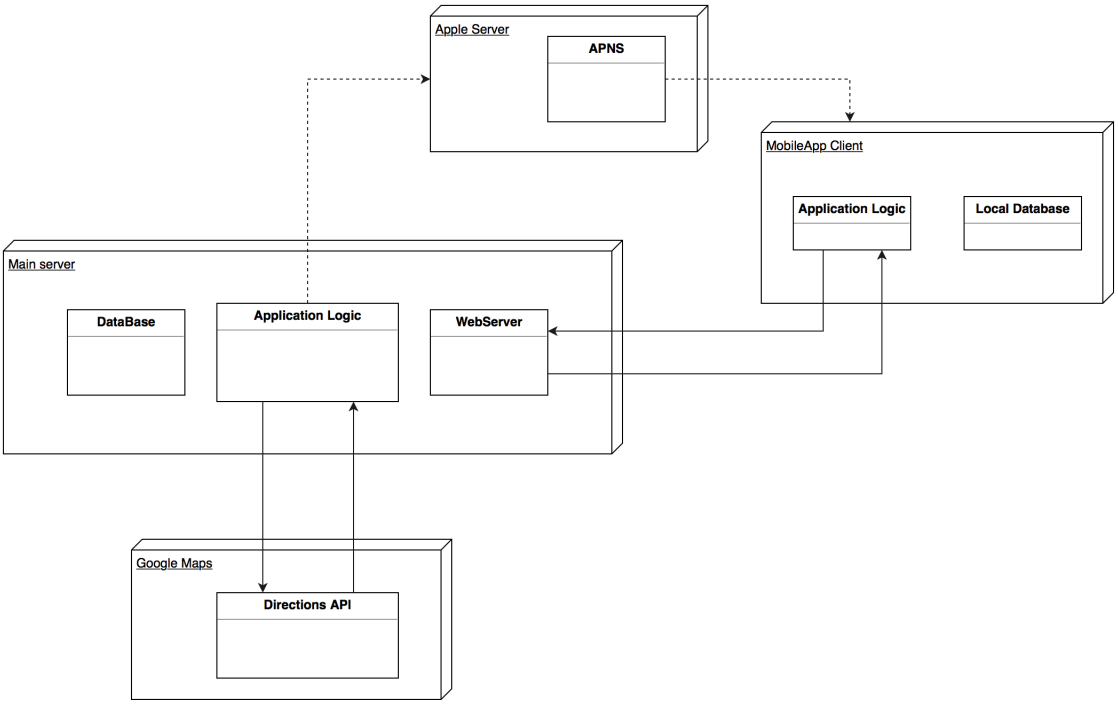
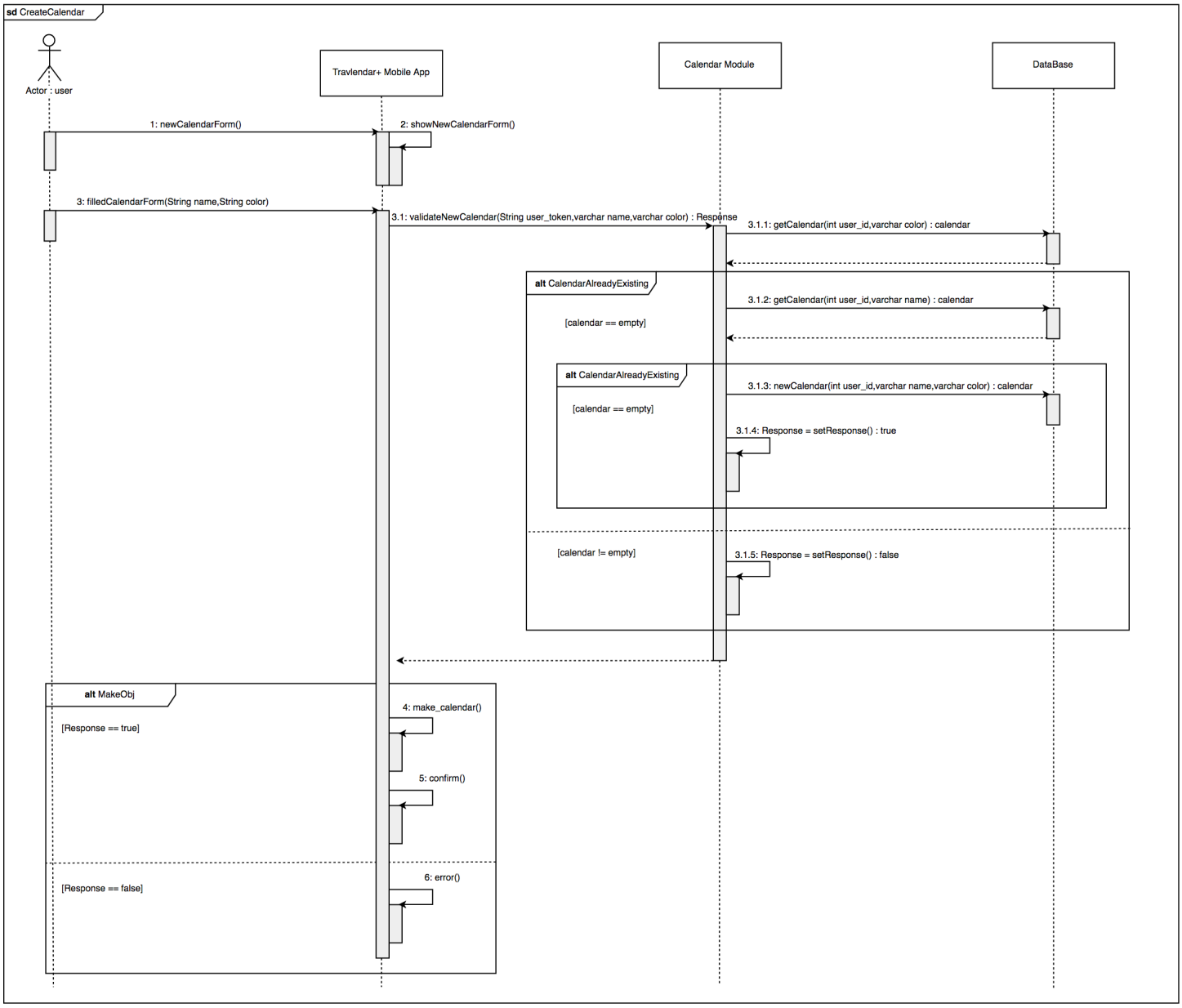
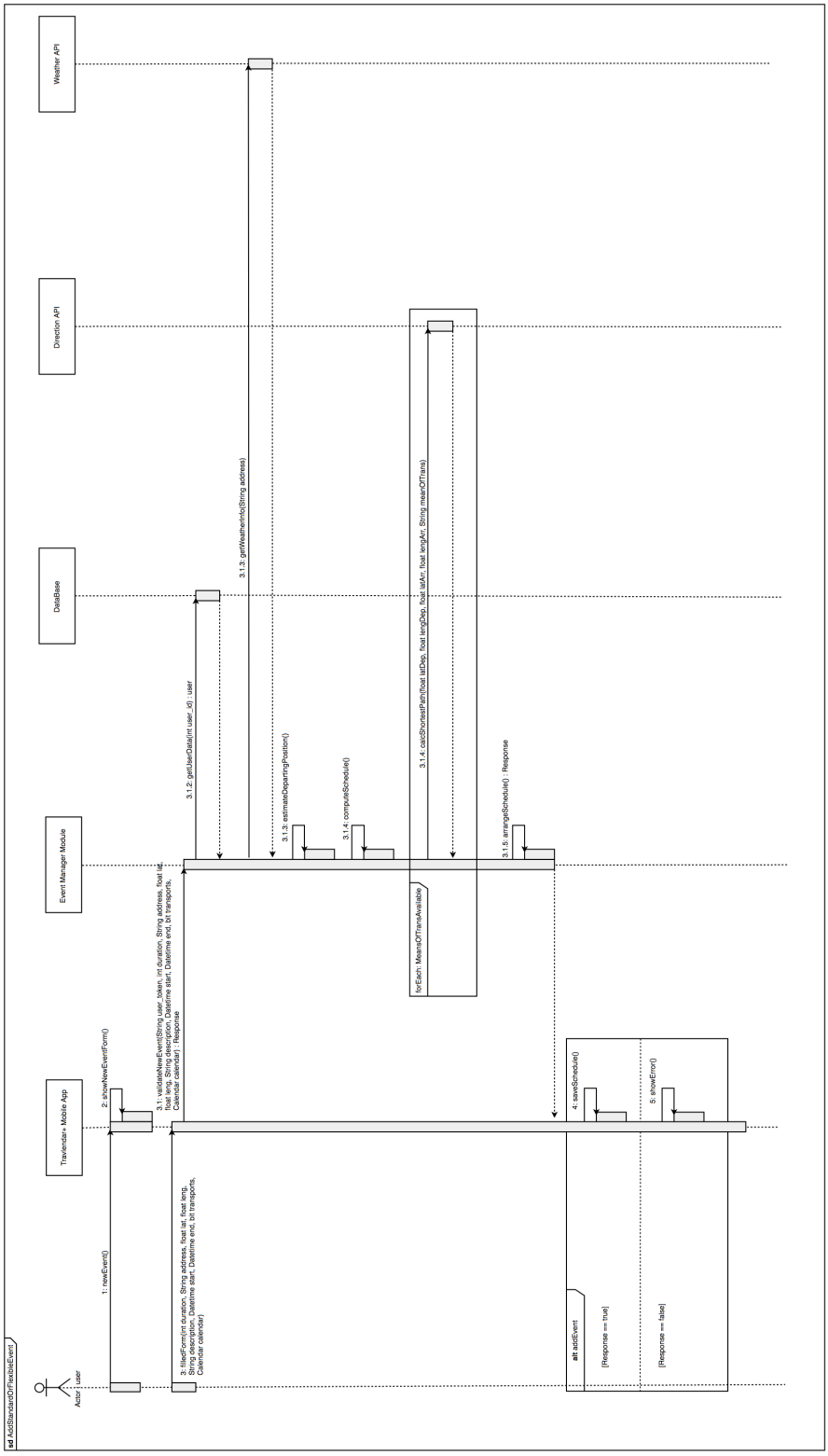


Figure 8: High level components

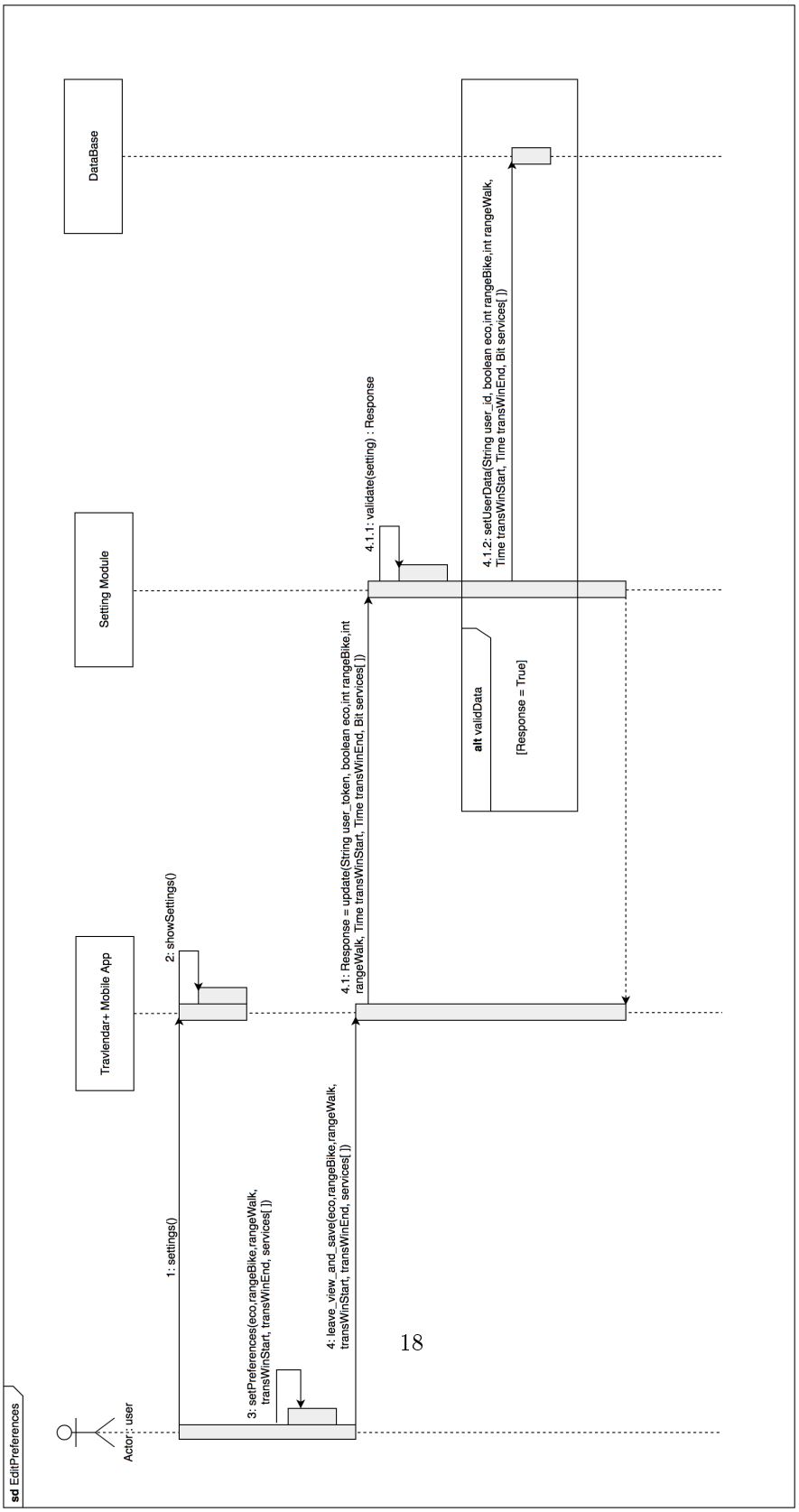
2.4 Runtime view

2.4.1 Create Calendar

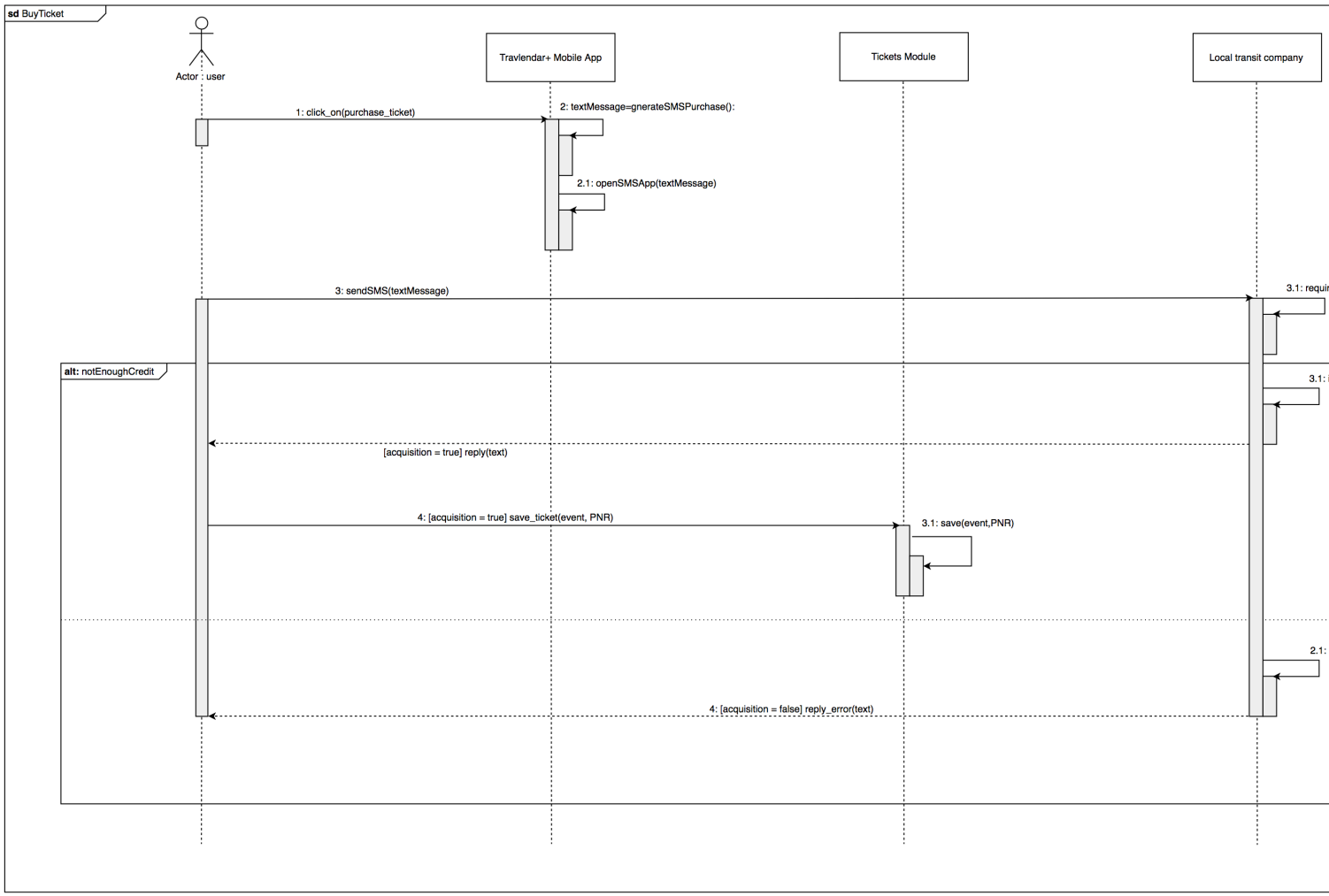



sd AddStandardOrFlexibleEvent
Actor : user
Travlendar+ Mobile App
Event Manager Module
DataBase
Direction API
Weather API
1: newEvent()
2: showNewEventForm()
3: filledForm(int duration, String address, float lat, float long, String description, Datetime start, Datetime end, bit transports, Calendar calendar)
3.1: validateNewEvent(String user_token, int duration, String address, float lat, float long, String description, Datetime start, Datetime end, bit transports, Calendar calendar) : Response
3.1.2: getUserData(int user_id) : user
3.1.3: getWeatherInfo(String address)
3.1.3: estimateDeparingPosition()
3.1.4: computeSchedule()
3.1.4: calcShortestPath(float latDep, float longDep, float latArr, float longArr, String meanOfTrans)
forEach: MeansOfTransAvailable
3.1.5: arrangeSchedule() : Response
alt addEvent
[Response == true]
4: saveSchedule()
[Response == false]
5: showError()

2.4.3 Edit Preferences



2.4.4 Buy Ticket



2.5 Component interfaces

In this section are described the interfaces that the Clients and the Server will use in order to communicate

2.5.1 RESTful API

Our tiers will be connected through a network that will use a JSON RESTful Application Programming Interface.

The API will lie on a Web server and will be called on a HTTP channel with a classic TLS encryption layer

All the methods will require an authentication, except for the login one. The exposed methods are:

- **/api/v1/login**
  - POST: Allow the user to get an authentication token. If the user is not present on the server database, he/she will be registered
    - Parameters: **user_token** A user token that is generated client side by the application
    - Return: **device_token** A token that will be used for future authentication on that device with other APIs
- **/api/v1/pushNotification**
  - POST: Update the current push notification
- **/api/v1/settings**
  - GET: Retrieve the user settings
  - POST: Save some settings
- **/api/v1/calendars**
  - GET: Retrieve the user calendars
  - POST: Save some calendars
- **/api/v1/events**
  - GET: Retrieve the user events
  - POST: Save some events
- **/api/v1/schedule**
  - GET: Retrieve the user schedule
- **/api/v1/position**
  - POST: Update the current user position

2.6 Selected architectural styles and patterns

2.6.1 Overall Architecture

Our system will be composed of two tiers:

1. A client executable, that will run on an iOS device
2. A server part, that will be served on a scalable Heroku instance

Each part will need a database: the server will store any piece of information about users, calendars and events on its database, while the client's database will have a cached copy of some events and calendars.

2.6.2 Data Definition Language

The following DDL in a SQL-like language is proposed for the database

```sql
CREATE TABLE `calendars` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned NOT NULL,
  `name` varchar(255) NOT NULL DEFAULT '',
  `color` varchar(6) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  CONSTRAINT `calendars_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `devices` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned NOT NULL,
  `access_token` varchar(32) NOT NULL DEFAULT '',
  `push_token` text,
  `device_type` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  CONSTRAINT `devices_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `events` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `title` varchar(255) NOT NULL DEFAULT '',
  `address` varchar(511) DEFAULT NULL,
  `lat` float(9,7) DEFAULT NULL,
  `lng` float(10,7) DEFAULT NULL,
  `start` datetime NOT NULL,
  `end` datetime NOT NULL,
  `duration` int(11) unsigned DEFAULT NULL,
  `repetitions` bit(7) NOT NULL DEFAULT b'0',
  `calendar_id` int(11) unsigned NOT NULL,
  `transports` bit(5) NOT NULL DEFAULT b'11111',
  PRIMARY KEY (`id`),
  KEY `calendar_id` (`calendar_id`),
  CONSTRAINT `events_ibfk_1` FOREIGN KEY (`calendar_id`) REFERENCES `calendars` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
CREATE TABLE `travels` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `route` int(11) unsigned NOT NULL,
  `time` int(11) unsigned NOT NULL,
  `transport_mean` enum('WALKING','BIKING','PUBLIC','SHARING','CAR') DEFAULT NULL,
  `waypoints` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_token` varchar(24) NOT NULL DEFAULT '',
  `last_known_position_lat` float(9,7) DEFAULT NULL,
  `last_known_position_lng` float(10,7) DEFAULT NULL,
  `updated_at` datetime NOT NULL ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_id` (`user_token`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `events_travels` (
  `event_id` int(11) unsigned NOT NULL,
  `travel_id` int(11) unsigned NOT NULL,
  PRIMARY KEY (`event_id`,`travel_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `settings` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned NOT NULL,
  `eco_mode` bit(1) NOT NULL DEFAULT b'0',
  `max_walking_distance` int(11) unsigned NOT NULL DEFAULT '2000',
  `max_biking_distance` int(11) unsigned NOT NULL DEFAULT '4000',
  `start_public_transportation` time NOT NULL DEFAULT '07:00:00',
  `end_public_transportation` time NOT NULL DEFAULT '22:00:00',
  `enjoy_enabled` bit(1) NOT NULL DEFAULT b'0',
  `car2go_enabled` bit(1) NOT NULL DEFAULT b'0',
  `uber_enabled` bit(1) NOT NULL DEFAULT b'0',
  `mobike_enabled` bit(1) NOT NULL DEFAULT b'0',
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  CONSTRAINT `settings_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Figure 9: DDL

2.6.3 Design Pattern

- **Client-Server**: This is the most useful and common pattern, that allow us to process the data acquisition and elaboration separately from the client logic. Moreover, using this pattern we can create simpler clients, and make the system more scalable: in fact, in order to scale the entire system, we can simply add more server, and keep the client the same.
- **Singleton**: Server-side the server app will be a Singleton, while client side there will be a API Manager as Singleton
- **Publish/Subscribe**: Used for the push notifications between the clients and the server

2.7 Other design decisions

We'll also use an external map service and a Weather API service, in order to find the best travel route, and to show to the user the events in a map view according to the weather forecast. To achieve this aim, we'll use the Google Maps Directions API and OpenWeatherMap.

3 Algorithm Design

In this section will be presented a description and an example code of the main algorithms for the Travlendar+ platform.

Code provided in these subsections will be written in C++ or pseudo-code that uses its syntax in order to better understand it without making the reading too complex.

Functions that will be called and not specified in each Subsection can be found in Subsection 3.4.

3.1 Event Schedule Algorithm

3.1.1 Description

This algorithm is the one who takes all users's events and tries to schedule them whether possible according to user preferences and distance from places. This will run Server Side.

Such function prepares a schedule following these steps:

1. Builds an array of events for each calendar of the given user;
2. Filters those events and divided them in two arrays of fixed and flexible events as described in Subsection 3.2;
3. It tries to fit them in the schedule:
   (a) If it succeeds (they are fittable and reachable) it adds them to the fixed array and considers them as fixed;
   (b) If it fails, it sends a notification to the user;
4. Then it tries to check if the other fixed events are reachable using the reachability function described in Subsection 3.3;
5. It terminates returning the created schedule.

```
Schedule* schedule(
        User *u,
        unsigned long date1,
        unsigned long date2
);
```

This is the main one that takes in:

- **u**: user reference;
- **date1**: start date;
- **date2**: ending date.

Overloaded Functions:

Overload 1: Schedule function that checks just the 12 hours before and after the event as parameter that has been created, modified or deleted.

```
Schedule* schedule(
        User *u,
        Event *e
);
```

This is the first one overloaded that takes in:

- **u**: user reference;
- **e**: event needing reschedule;

Overload 2: Schedule function that checks the next hours, this can be called when user uploads his current position or the system calls it every 2 hours;

```
Schedule* schedule(User *u);
```

This is the first one overloaded that takes in:

- **u**: user reference;

3.1.2 Example of Implementation

Overloads:

```
// Schedule function that checks the next hours, this can be called when user uploads his current position or the system calls it every 2 hours
Schedule* schedule(User *u) {
    // Schedule from now to tomorrow
    return schedule(u, now(), start_of_today() + 86400);
}

// Schedule function that checks just the 12 hours before and after the event as parameter that has been created, modified or deleted
Schedule* schedule(User *u, Event *e) {
    // Schedule 12 hours before and after the event is created
    if (e.repetitions & 0x0111111) {
        Schedule *scheduled = new Schedule();

        for each(unsigned long day in daysNeedingReschedule(u, e)) {
            Schedule *e = schedule(u, day, day + 86400);
            appendSchedule(scheduled, e);
        }

        return scheduled;
    }
    else {
        return schedule(u, e.start - 43200, e.start + 43200);
    }
}
```

Main Function:

```
// Schedule an entire date interval, this can be called from the other schedule
 functions
Schedule* schedule(User *u, unsigned long date1, unsigned long date2) {
    Schedule *s = new Schedule();
    s->update_time = now();
    vector<Event> *scheduled = new vector<Event>();
    s->schedule = scheduled;


    // Check for each user's calendar overlapping and reachability
    for each(Calendar *c in u.calendars) {

        // Get all the events on the calendar between that dates making sure they
         are all now or in the future
        vector<Event> *events = getCalendarEventsInRange(c, d1, d2);
        events = nextEvents(events);
        resetEvents(events);

        // Ensure events does not overlap
        if (overlap(events)) {
            vector<Event> *overlapping = overlappingEvents(events);
            notifyOverlapping(u, overlapping);
            break;
        }


        // Just filter events ordering them based on filterEvent() rules
        vector<Event> *fixed;
        vector<Event> *flexible;
        filterEvents(events, fixed, flexible);

        // Try to fit each flexible event
        for each(Event *e in flexible) {
            // Get available time slots from fixed event array relative to this
             event
            vector<TimeSlot> *timeSlots = timeSlots(fixed, e);

            // For each time slot I try to fit the events
            for (int j = 0; j<timeSlots->size(); j++) {

                long eDuration = (long)e->duration;                         //
                 Event Duration
                long sDuration = timeSlots[j]->end - timeSlots[j]->start;   //
                 Time slot duration
```

```
if (eDuration > sDuration) {
    // Can't fit so try another one
    continue;
}

// We can try to know if it is reachable
if (eventIsToday(e)) {
    // Consider user latest reliable position
    Event *prev = siblingTimeSlotEvent(c, timeSlots[j]);
    if (prev == null) {
        // no previous event, try user location
        Coordinates *userPos = reliableUserPosition(u);
        if (userPos == null) {
            // No user position, try another time slot
            continue;
        }
        else {
            if (eventIsReachable(userPos, e)) {
                // REACHABLE
                insert(fixed, e);
                remove(flexible, e);
                break;
            }
            else {
                // Try another time slot
                continue;
            }
        }

    }
    else {
        if (eventIsReachable(prev, e)) {
            // REACHABLE
            insert(fixed, e);
            remove(flexible, e);
            break;
        }
        else {
            // Try another time slot
            continue;
        }
    }
}
else {
    // Don't consider last reliable user position

    Event *prev = siblingTimeSlotEvent(c, timeSlots[j]);
    if (prev == null) {
        // no previous event try another time slot
        continue;
    }
    else {
        if (eventIsReachable(prev, e)) {
            // REACHABLE
            insert(fixed, e);
            remove(flexible, e);
            break;
```

```
                }
                else {
                    // Try another time slot
                    continue;
                }
            }
        }

    }


}

if (flexible->size() > 0) {
    notifyUnreachable(u, flexible);
    return null;
}

// Now we need to make sure that unscheduled and unfitted events are
 reachable

for each(int i = 0; i<events->size(); i++) {
    if (i == events->size()-1) {
        // last one so exit
        break;
    }

    if (events[i]->routes == null || events[i]->suggested_start == null ||
     events[i]->suggested_end == null) {
        // Unscheduled

        if (eventIsFirstOfDay(events[i], c)) {
            if (eventIsToday(events[i])) {
                // Try with user location

                Coordinates *userPos = reliableUserPosition(u);
                if (userPos == null) {
                    // No user position, unreachable then
                    notifyUnreachable(u, events[i]);
                    return null;
                }
                else {
                    if (eventIsReachable(userPos, events[i])) {
                        // REACHABLE
                        scheduled->push_back(event[i]);
                        continue;
                    }
                    else {
                        // Not Reachable
                        notifyUnreachable(u, events[i]);
                        return null;
                    }
                }
            }
            else {
                // Always reachable so go next
                scheduled->push_back(event[i]);
```

```
                        continue;
                    }
                }
                else {
                    // Check if next event is reachable from this one

                    if (eventIsReachable(events[i+1], events[i])) {
                        // REACHABLE
                        scheduled->push_back(event[i+1]);
                        continue;
                    }
                    else {
                        // Not Reachable
                        notifyUnreachable(u, events[i+1]);
                        return null;
                    }
                }


            }
            else {
                // Already scheduled
                scheduled->push_back(e);
            }

        }



    }

    return s;

}
```

3.2 Flexible Event Fitting Order Algorithm

3.2.1 Description

This is the algorithm that will be triggered when section 3.1's functions calls:

```
void filterEvents(
        vector<Event> *main,
        vector<Event> *fixed,
        vector<Event> *flexible
);
```

Such function will do the following:

1. Put fixed events contained in **main** to **fixed** vector and sort them by happening date;
2. Put flexible events contained in **main** to **flexible** vector;
3. Sort **flexible** vector using the fitness function described above.

$$fittability(e) = 1 - \frac{duration(e)}{end(e) - start(e) - occupiedRange(c)}$$

Figure 10: Fittability Function

This mathematical function takes into:

- **duration**(e): the duration of the flexible event;
- **end**(e): preferred end time interval of flexible event;
- **start**(e): preferred start time interval of flexible event;
- **occupiedRange**(c): number of hours already occupied in that range by other events in the event's calendar.

3.2.2 Example of Implementation

```
void filterEvents(vector<Event> *main, vector<Event> *fixed, vector<Event> *flexible) {
    for (int i = 0; i < main->size(); i++) {
        Event *e = main[i];
        if (e->duration == 0) {
            // It is fixed so add to fixed array
            fixed->push_back(e);
        }
        else {
            // It is flexible so add to flexible array
            flexible->push_back(e);
        }

    }

    // ** Now sort them

    // Sort fixed ones by start time
    sort(fixed.begin(), fixed.end(), [](const Event &lhs, const Event &rhs) {
       return lhs.start < rhs.start;
    });

    // Sort flexible ones by their fittability index
    sort(flexible.begin(), flexible.end(), [](const Event &lhs, const Event &rhs) {
        return fitness(lhs) < fitness(rhs);
     });
}
```

This piece of code uses also this utility function in order to get the fitness value of an event:

```
double fitness(Event *e) {
    return 1 - ((double)e->duration / ((double)e->end - (double)e->start - occupiedTime(e->calendar, e));
}
```

And another utility that returns occupied time by other events in a calendar prototyped above:

```
double occupiedTime(Calendar *c, Event *e);
```

3.3 Reachability Function Algorithm

This algorithm is the one which takes an event and verifies if it is reachable from another event or from a specified position.

```
bool eventIsReachable(Event *e1, Event *e2);

bool eventIsReachable(Coordinates coords, Event *e);
```

In order to check reachability this function does:

1. Takes user preferences taking into account maximum walk distance, maximum distance by bike, public transport timings;
2. Retrieves weather data about the target event place;
3. Queries Google Directions API to find the best routes paying attention to the target event's preferre travel means (according to weather data);
   (a) If the time to get to the place is compatible with the start of the event then it is reachable and so routes fields and suggested start/end time will be populated;
   (b) Otherwise it fails quits and returns false;

3.4 Other Used Functions, Algorithms and Data Structures

In this section can be found utility functions used by the main algorithms to simplify the development.

Images include function prototypes well commented so that they can be self-explanatory.

3.4.1 Data Structures

```
// ****** Utility

struct TimeSlot {
    unsigned long start;
    unsigned long end;
};

enum TransportMean {
    walk,
    bike,
    public_transportation,
    ride,
    car
};
```

```
// ****** DATA CLASSES

class Coordinates {

    float lat;
    float lng;

};

class Transportation {

    int time; // In minutes
    TransportMean transport;
    vector<Coordinates> *waypoints;

};

class Route {

    vector<Transportation> *transportations;

};

class User {

    int id;
    string user_id;

    vector<Calendar> calendars;
    vector<Device> devices;

    Coordinates last_known_position;

    Setting settings;

};



class Settings {

    int id;
    bool eco_mode;

    int max_walking_distance;
    int max_biking_distance;

    unsigned long start_public_transportation;
    unsigned long end_public_transportation;

    User *user;

};

class Calendar {

    int id;
    string name;
    string color;

    vector<Event> events;

    User *user;

};
```

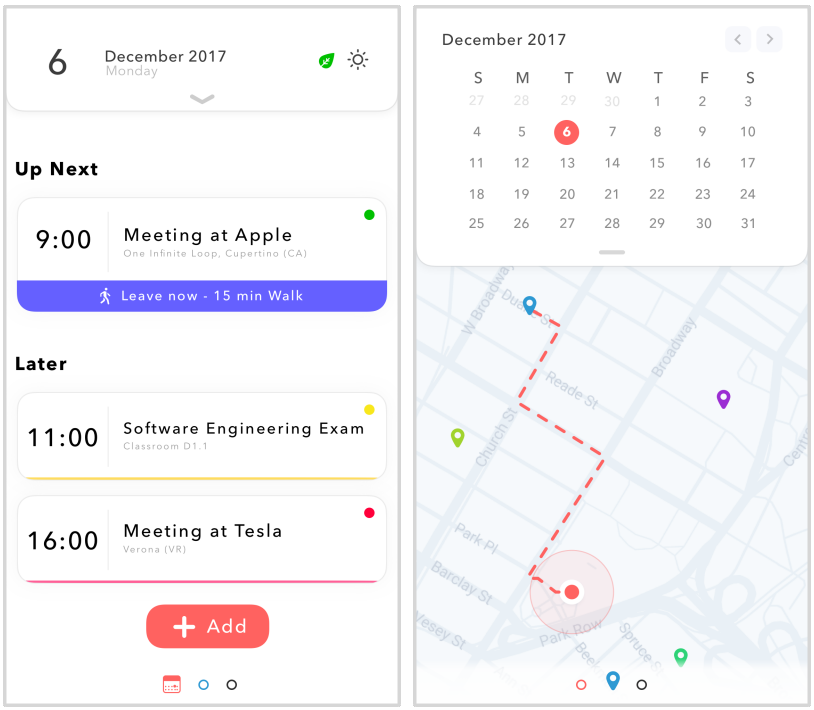
```
class Event {

    int id;
    string title;
    string address;

    Coordinates coords;

    // time
    unsigned long start;
    unsigned long end;
    int duration;                       // 0 if fixed event
    unsigned long suggested_start;      // not present in server database, just sent to client
    unsigned long suggested_end;        // not present in server database, just sent to client

    uint8_t repetitions;                // MSB 0, the others should be used as booleans for each weekday
    bool transports[5];

    vector<Route> *routes;              // not present in server database, just sent to client

    Calendar *calendar;

};


class Device {

    int id;
    string access_token;
    string push_token;
    string device_type;

    User *user;

};


class Schedule {

    unsigned long update_time;
    vector<Event>* schedule;

};
```

3.4.2 Utility Functions

```
// ******** Utilities

// Returns current timestamp
unsigned long now();

// Returns today's start timestamp
unsigned long start_of_today();

// Returns timestamp of the start of the day of a given timestamp
unsigned long start_of_day(unsigned long time);

// Foreach day in which events that are not e exist returns an array of dates in which schedule function should be called
vector<unsigned long> daysNeedingReschedule(User *u, Event *e);
```

```
// ******** User

// Returns last reliable user position
Coordinates* reliableUserPosition(User *u);



// ******** Events

// Given a set of events returns whether these overlaps
bool overlap(vector<Event> *events);

// Given a set of events returns an array of overlapping events if they exist
vector<Event>* overlappingEvents(vector<Event> *events);

// Returns a set of Calendar's events between some dates ordered by startdate.
vector<Event>* getCalendarEventsInRange(Calendar *calendar, unsigned long d1, unsigned long d2);

// Returns Present / future events from a set of events
vector<Event>* nextEvents(vector<Event>*);

// Removes travel data in an array of events, removing previously calculated routes and suggestions
void resetEvents(vector<Event> events);

// Returns true if the event is today
bool eventIsToday(Event *e);

// Returns true if this event is scheduled as first of the day
bool eventIsFirstOfDay(Event *e, Calendar *c);



// ******** TimeSlot

// Returns available time slots from a set of events relative to a single flexible event ordered from the smallest to the tallest
vector<TimeSlot> timeSlots(vector<Event> *events , Event *relative);

// Returns the event prior to a time slot
Event* siblingTimeSlotEvent(Calendar *c, TimeSlot *t);


// ******** Routes

// Returns time to reach event which is the mean value between all routes time
long routeTime(vector<Route> *routes);


// ******** Schedule
// Appends a schedule to an existing one
Schedule* appendSchedule(Schedule *s1, Schedule *s2);


// ******** Vector help
void insert(vector<Event> *e, Event *ev);
void remove(vector<Event> *e, Event *ev);



// ********* Notifications
// Sends back a notifications
void notifyOverlapping(User *u, vector<Event> *events);
void notifyUnreachable(User *u, vector<Event> *events);
void notifyUnreachable(User *u, Event* event);
```

4 User Interface Design



User Interface Design is very important to get a clear idea on how the final result of the application will appear to people.

This translated into creating Mockups of the application which can be found at chapter 3.1.1 "User Interfaces" of the RASD document previously delivered.

5 Requirements Traceability

The choices presented in this document have the aim to achieve the goals described on the *Requirements Analysis and Specification Document*. Follow a list of goals with each component and

- [G0] Allow people to use the app without a login function
  - [R1] User Services: Account Manager Module through the *Auto-login Service* interface
- [G1] Allow people to view the daily schedule with coming up events at the top
  - [R2][R4][R5] Scheduling Services: Event Manager Module through *EventHistory* interface
  - [R3] Scheduling Services: Event Manager Module through *ActivitiesScheduler* interface
- [G2] Allow people to view previous events
  - [R6] Scheduling Services: Calendar Module through *ManageCalendars* interface
  - [R7] Scheduling Services: Event Manager Module through *EventHistory* interface
  - [R8] Scheduling Services: Event Manager Module through *ManageEvents* interface
- [G3] Allow people to view the detail of each event
  - [R9][R11] Scheduling Services: Event Manager Module through *EventHistory* interface
  - [R10][R12][R13] User Services: Tickets Module through *TicketsManaging* interface
  - [R9][R14] Scheduling Services: Event Manager Module through *ActivitiesScheduler* interface
- [G4] Allow the users to create an event
  - [R15][R16][R17][R18][R19][R20] Scheduling Services: Event Manager Module through *ManageEvents* interface
- [G5] Allow people to see daily events on a map
  - [R21] Scheduling Services: Event Manager Module through *EventHistory* interface
  - [R22] Scheduling Services: Event Manager Module through *ActivitiesScheduler* interface

- [G6] Allow the user to set preferences
  - [R22][R23][R24][R26] User Services: Setting Module through *ManageSetting* interface
  - [R25] Scheduling Services: Calendar Module through *ManageCalendars* interface
- [G7] Allow the users to create calendars
  - [R27][R28] Scheduling Services: Calendar Module through *ManageCalendars* interface
- [G8] Notify the user that it's time to leave for the appointment
  - [R29][R30] Scheduling Services: Event Manager Module through *ActivitiesScheduler* interface

6 Implementation, Integration and Test Plan

In this section it is described which it would be the most useful way to proceed with the implementation. It is assumed that the database has already been created according to the previously specified ER diagram in the Figure 7, and it's up and running on a PostgreSQL instance. The Client and the Server part will be elaborated in the same time, starting from the Scheduling Services.

6.1 Scheduling Services

The first subcomponent that should be implemented is the EventManager Module, since this subcomponent is the most complex of our system, and to prevent any bugs from appearing at the end of the project, causing problems and propagating them in other parts of the application. It is composed of a simple CRUD API and of a scheduler. The scheduler main algorithm is described in the Sub Section 3.1.

The other subcomponent is the Calendar Module, that could be developed apart from the EventManager, since it is simple and there is no need of a complex logic.

6.2 User Services

The Account Manager Module is one of the first used in the flow of a typical connection made by the client application, so it should be implemented at the beginning. Since it is not related to the EventManager Module, it can be developed in parallel.

The Settings Module is a CRUD API, and it can be developed right after the Account Manager Module.

The Tickets Module is one of the last subcomponents that should be implemented, since it's an enhancement of the application, and not strictly needed to make the system work as intended.

6.3 Testing

The testing will be performed with the paradigm of the Test Driven Development, in which a test is written at the beginning of the development, and after that it is implemented the code that succeed in satisfying that test. The TDD is described in Figure 11

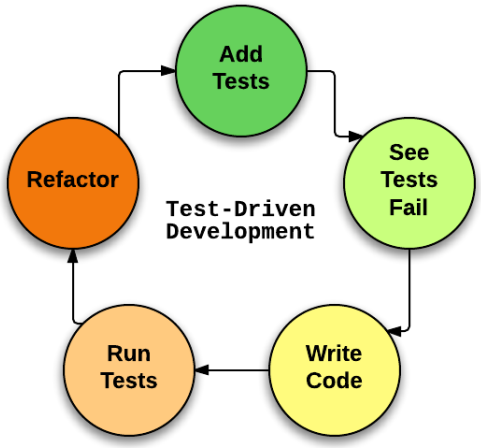


Figure 11: Test Driven Development Flow Chart

On the Server Side the test will be performed with the framework **mocha**, while on the Client they will be written using the **Swift Unit Test**.

7 Effort Spent

- Filippo Calzavara: 24.30h
- Giovanni Filaferro: 24.30h
- Benedetto Maria Nespoli: 24.30h

8 References

- RASD document previously delivered.

8.1 Tools used

The tools used to developed this document are:

- GitHub: to versioning this document
- Source Tree: for a better use of Git Hub
- draw.io: to make all the schemas shown in this document
- LyX: to make this LaTeX document