Politecnico di Milano

Corso di Laurea Magistrale in Computer Science and Engineering
Dipartimento di Elettronica e Informazione

Travlendar+

Design Document

Reference professor:
Prof. Elisabetta Di Nitto

Authors:
Alessandro Aimi Matr. 899642
Roberto Bigazzi Matr. 899411
Filippo Collini Matr. 829918

Version 1.1.0
30/11/2017
Academic Year 2017-2018

Contents

Chapter 1

Introduction

This document is the Design Document (DD) of a mobile application called Travlendar+. It is mainly addressed to the software development team and its purpose is to provide an overall view of the architecture of the new system. The document defines:

- High level architecture;
- Used design patterns;
- Main components of the system;
- Runtime behavior.

1.A Description of the given problem

Travlendar+ is a mobile, calendar-based application that helps the user to manage his appointments and to a greater extent set up the trip to his destination, choosing the best means of transport depending on his needs.
Travlendar+ will choose the most suitable way to get the user to his destination between a large pool of options, considering public transportation, personal vehicles, locating cars or bikes of sharing services and walking to the destination. It will take account of weather, traffic, possible passengers if any, the user-set break times and the potential will to minimize the carbon footprint of the trip, always focusing on taking him on time to his scheduled appointments.
Eventually the user will be able to purchase the tickets he will use to reach his destination in-app. The great customizability is one of the main strengths of Travlendar+, being able to fully comply with the user needs.

1.B Definitions and Acronyms

1.B.1 Definitions

List of the definitions used in this paper:

- Reverse Proxy: the component that will send the information received from multiple servers to the correct client.
- ToTrip: often used in code to refer to a trip with the event ad destination;
- FromTrip: often used in code to refer to a trip with the event ad departure location;
- Primary Event: feasible event;
- Secondary Event: not feasible event;
- Dynamic Event: event with no to/from trips and beginning changing on various factors.

1.B.2 Acronyms

List of the acronyms used in this paper:

- RASD: Requirements analysis and specification document;
- DD: Design document;
- MOT: Means of transport;
- RMI: Remote Method Invocation;
- API: Application Programming Interface;
- UX: User experience;
- BCE: Boundary Control Entity;
- UI: User interface;
- IaaS: Infrastructure as a Service;
- REST: Representational State Transfer;
- JNDI: Java Naming and Directory Interface;

- JDBC: Java DataBase Connectivity;
- EJB: Enterprise Java Beans;
- DBMS: Data Base Management System;
- JEE: Java Enterprise Edition;
- EIS: Enterprise Information System;
- JSP: Java Server Pages;
- JSF: Java Server Faces;
- JPA: Java Persistence API.

1.C Revision History

- 26/11/2017 Version 1.0.0 - First complete drawing up of the DD;
- 30/11/2017 Version 1.1.0 - Runtime view modifications.

1.D References

Documents list:

- Mandatory Project Assignments.pdf

Online sites list:

- https://developers.google.com/maps/ (used also to design mockups)
- https://developers.google.com/google-apps/calendar/ (used also to design mockups)
- https://openweathermap.org/api
- https://www.interoute.it/what-iaas
- http://www.cloudcomputingpatterns.org
- https://www.ibm.com/support/knowledgecenter/en/SSZLC2_7.0.0/
- com.ibm.commerce.developer.doc/concepts/csdmvcdespat.htm (MVC)

1.E Document Structure

The paper is structured as follows:

- Chapter 1: Introduction to the document, including some information about its composition and the description of the problem;
- Chapter 2: This chapter shows the main components of the system and the relationships between them. It also explains the main architectural styles and patterns adopted in the design of the system;
- Chapter 3: This chapter explains how the system will work using algorithms. Java code is used to write down the most significant algorithms for the application;
- Chapter 4: This chapter shows mockups of the application and more details about the User Interface using UX and BCE diagrams;
- Chapter 5: This chapter explains how the decisions taken in the RASD are associater to design decisions;
- Chapter 6: This chapter describes the order of implementation of the subcomponents of the system and the order in which integrate such subcomponents and test the integration;
- Chapter 7: Effort spent by the authors to draw up the document.

1.F Used tools

The tools used to create this document are:

- Photoshop for mockups;
- Draw.io for diagrams;
- Atom for algorithms drawing up;
- Github as version controller and to share documents;
- LaTeX for typesetting this document;
- Texmaker as editor;

Chapter 2

Architectural Design

2.A Overview

Architectural design is of crucial importance in software engineering because it will have to take account of functional and non-functional requirements, to meet the stakeholders needs and requests, and to help not to focus only on standalone elements losing the so called big picture of the system, always adhering to general principles of good quality. An important aspect is in fact to find a good trade-off between the high-level description near to the analysis and the low-level one near to the implementation. Coming up with good quality design and architecture is mostly a matter of experience and in our field, is also known the importance of the reusability of other's people work. So, we tried to build our system with various kind of this patterns and known architectural styles.

2.B Component View

2.B.1 High level components

The picture below will give a representation of the high-level structure of the system and a general view of its main components.

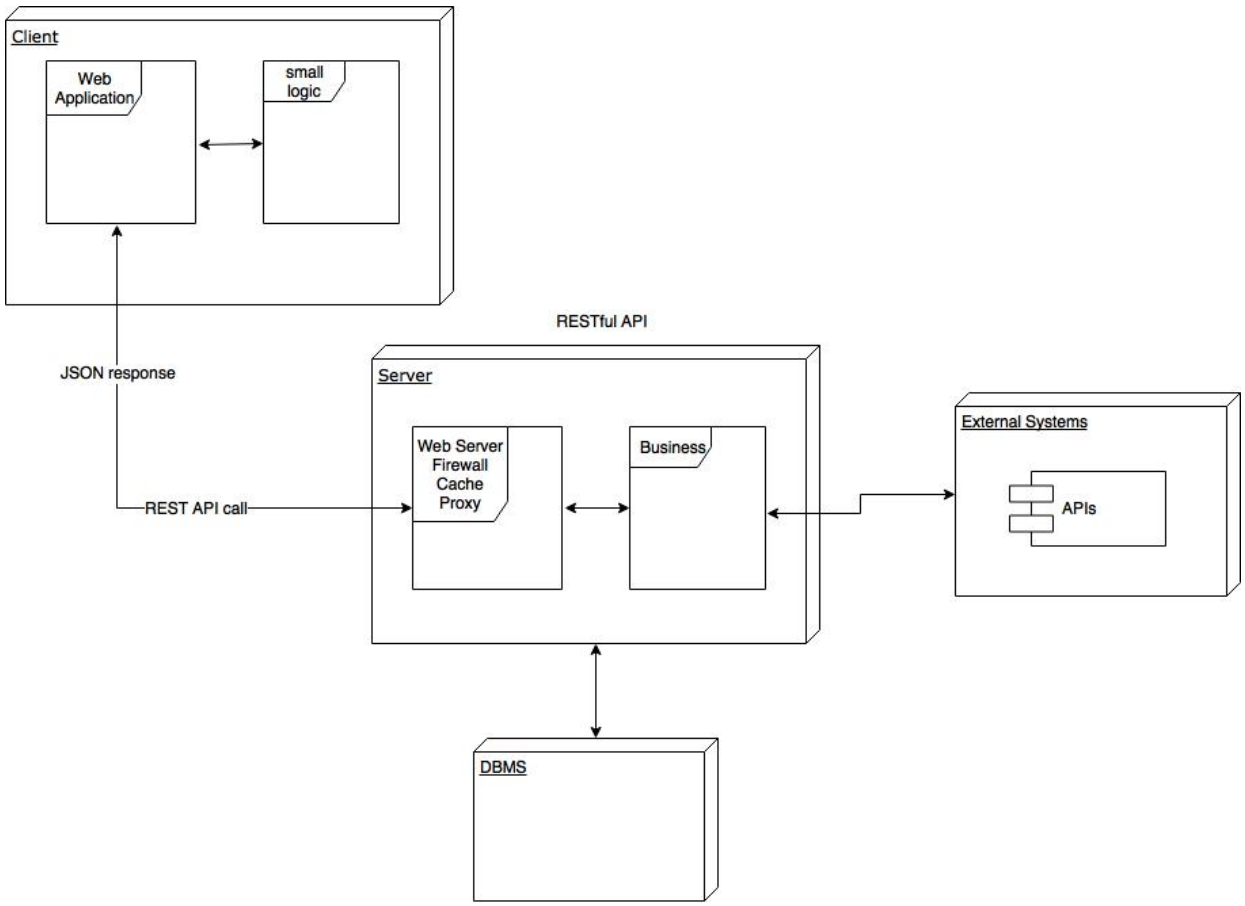

Figure 2.1: High level component view of the system

The client layer will send requests to the server through REST API call. More precisely these requests will be send to the first component of the application layer, the web server. Here the request has to pass through the firewall, then the proxy will direction it to the distributed collection of servers that represents the business layer. The proxy is also in charge to do the opposite communication, when the pieces of information will flow from the business servers, they will have to be directed to the opportune client and the reverse proxy will send to the right client JSON responses.

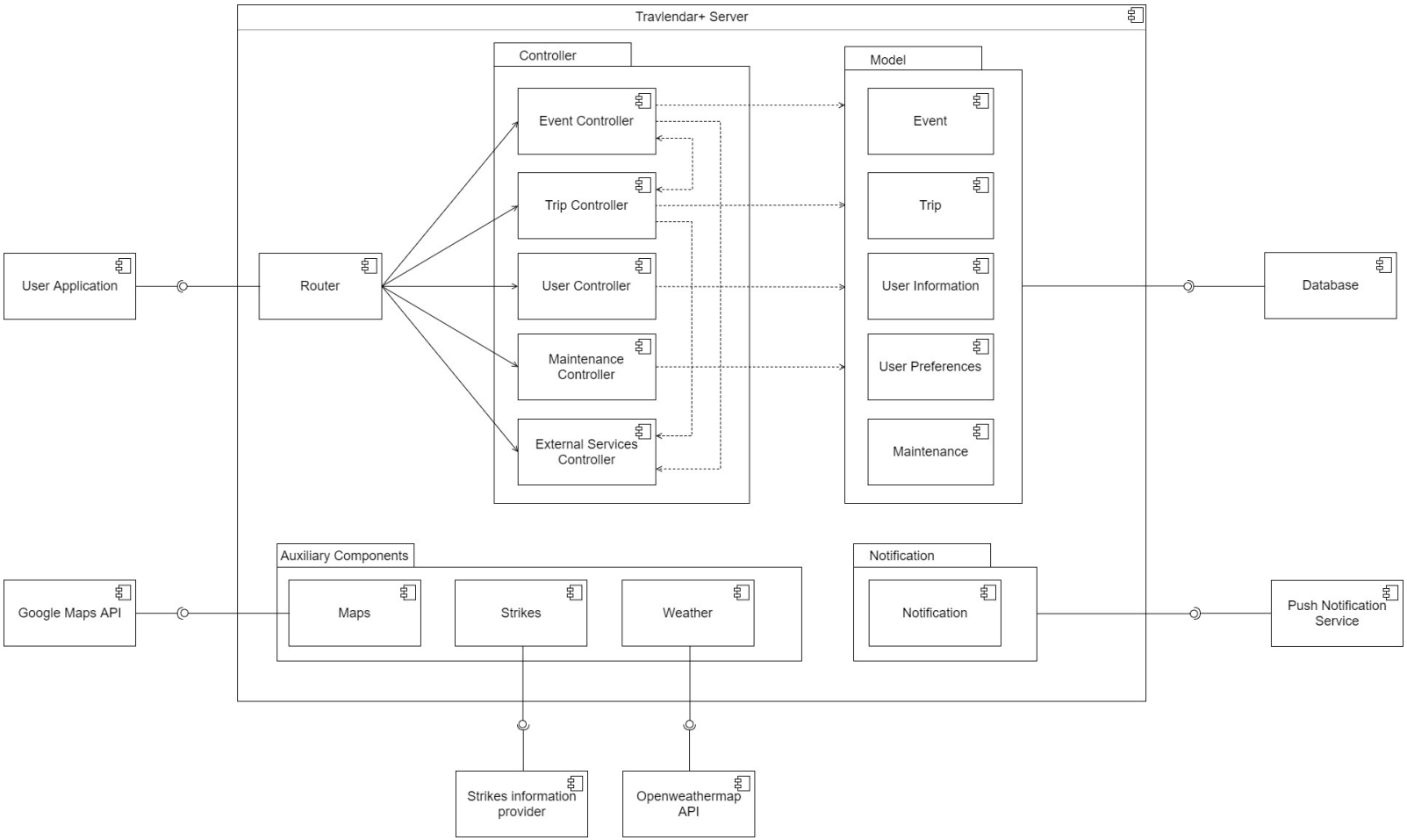

Figure 2.2: Component view of the system

The main components are divided in:

- Controller set of components
    1. Event Controller: in charge of the management of events, it will provide functions for adding, modify and delete events and will contain all the logic to do this properly;
    2. Trip Controller: in charge of the management of the trips. It will contain all the necessary logic to avoid overlapping of them and to advise the best option for the user. It also helps the purchase of the tickets;
    3. User Controller: in charge of the management of the user's info and preferences;
    4. Maintenance Controller: in charge of the management of the maintenance of the system;
    5. External Services Controller: in charge of the management of the external services connected to the system.
- Model set of Components are divided in:
    1. Event: will contain all the information concerning the events;
    2. Trip: will contain all the information concerning the trips;
    3. User Information: will contain all the information concerning the user's general info;
    4. User Preferences: will contain all the information concerning the user's preferences;
    5. Maintenance: will contain all the information concerning the maintenance of the system This layer is strictly connected to the database with which it can exchange data;
- Auxiliary Components and Notifications components are:
    1. Maps: will be connected to the Google Maps API;
    2. Strikes: will be connected to a multitude of APIs from which it will collect all the necessary info;
    3. Weather: will be connected to Open Weather Map API to collect info about weather conditions.

It is also present a Router to route request to the appropriate controller.

2.C Deployment View

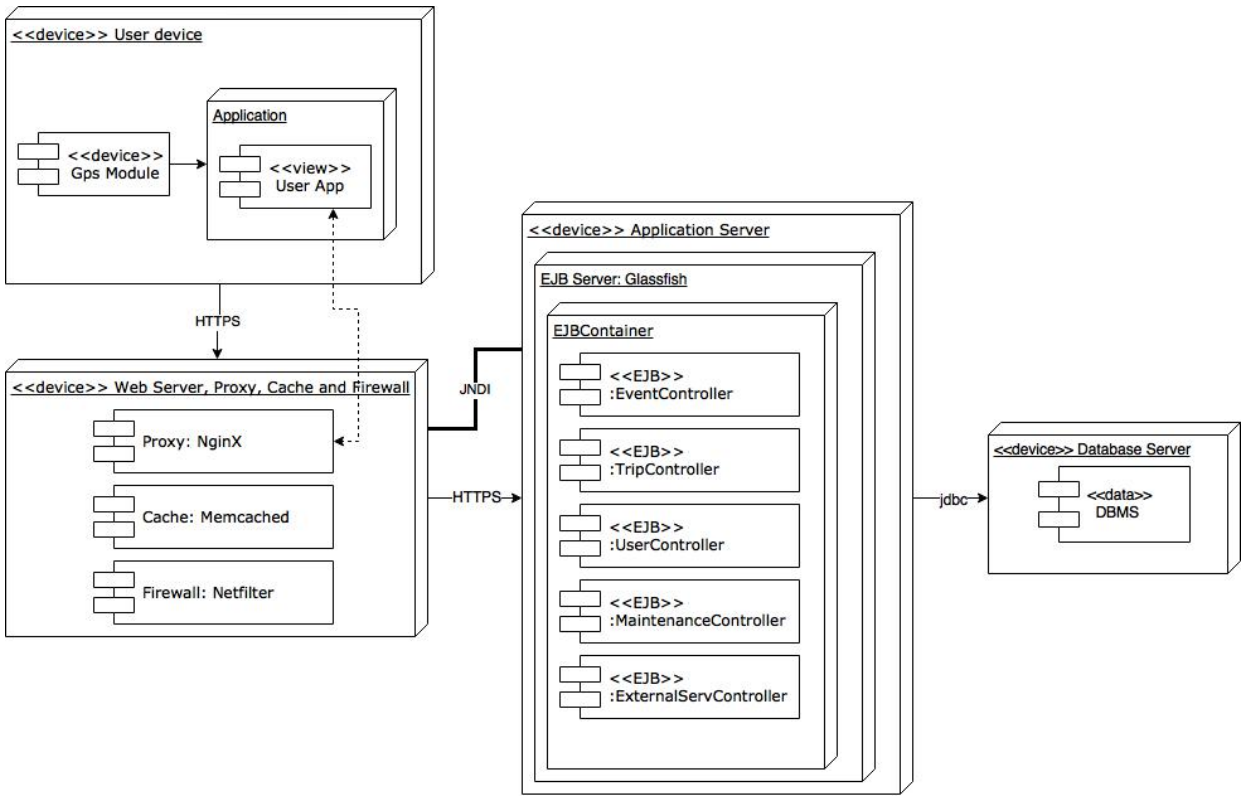

Figure 2.3: Deployment diagram of the system

Since our system is subject to several loads concerning data storage and other services we taught that the best deployment solution may be cloud computing, considering also the recent diffusion and opportunity to have access to powerful services with limited costs. To be as general as possible and avoid further implementation constraints we prefer to rely on the IaaS (Infrastructure as a Service) level. It's specifically a virtualized hardware, in other words an elaboration infrastructure. It includes offers like virtual storage space on servers, network connections, bandwidth and load balancing. Physically, the collection of hardware resources is extracted from several distributed servers and networks usually in different data centres, which maintenance is responsibility of the cloud provider. The client, on the other hand, has access to virtualized components to build his IT platforms. The main advantages are of this approach are:

- Scalability: thanks to IaaS upwards and downwards scalability is performed, delays are reduced and waste of resources is prevented;

- Costs: base hardware is configured and managed by the cloud provider, therefore no acquisition, installation and maintenance cost are necessary; the cost of the cloud service is almost proportionally to the amount of resource consumed, there are various contracts that allows to design a kind of customized service;

- Security: in IaaS configuration, physical security is ensured since it is typically a critical aspect for the provider. In-house security, on the other hand, is not usually an individual's or an organization's main business and, therefore, may not be as good as that offered by the IaaS cloud provider;

- Availability, Redundancy and Fault Tolerance: deploying different components on different machines allows the system not to totally go down if one of his components crashes. There's also no need to manage backups because lots of IaaS cloud providers offer automatic backup procedures.

On the other hand, some constraints must be respected:

- for the scaling mechanism, we have to design stateless components;

- we have also do upgrade the developed software;

- we have to take care of the maintenance of tools, database systems and the underlying infrastructure.

2.D Runtime View

In this section some sequence diagrams will be presented to describe the interactions that happen between the main components of the system when the most common functionalities are used. This is a high-level description of the actual interactions of the system-to-be, so functions and their names may be added, modified or deleted during the development process.

The functionalities considered for the runtime view are: login, adding of an event on the calendar, trip planning and trip customization.

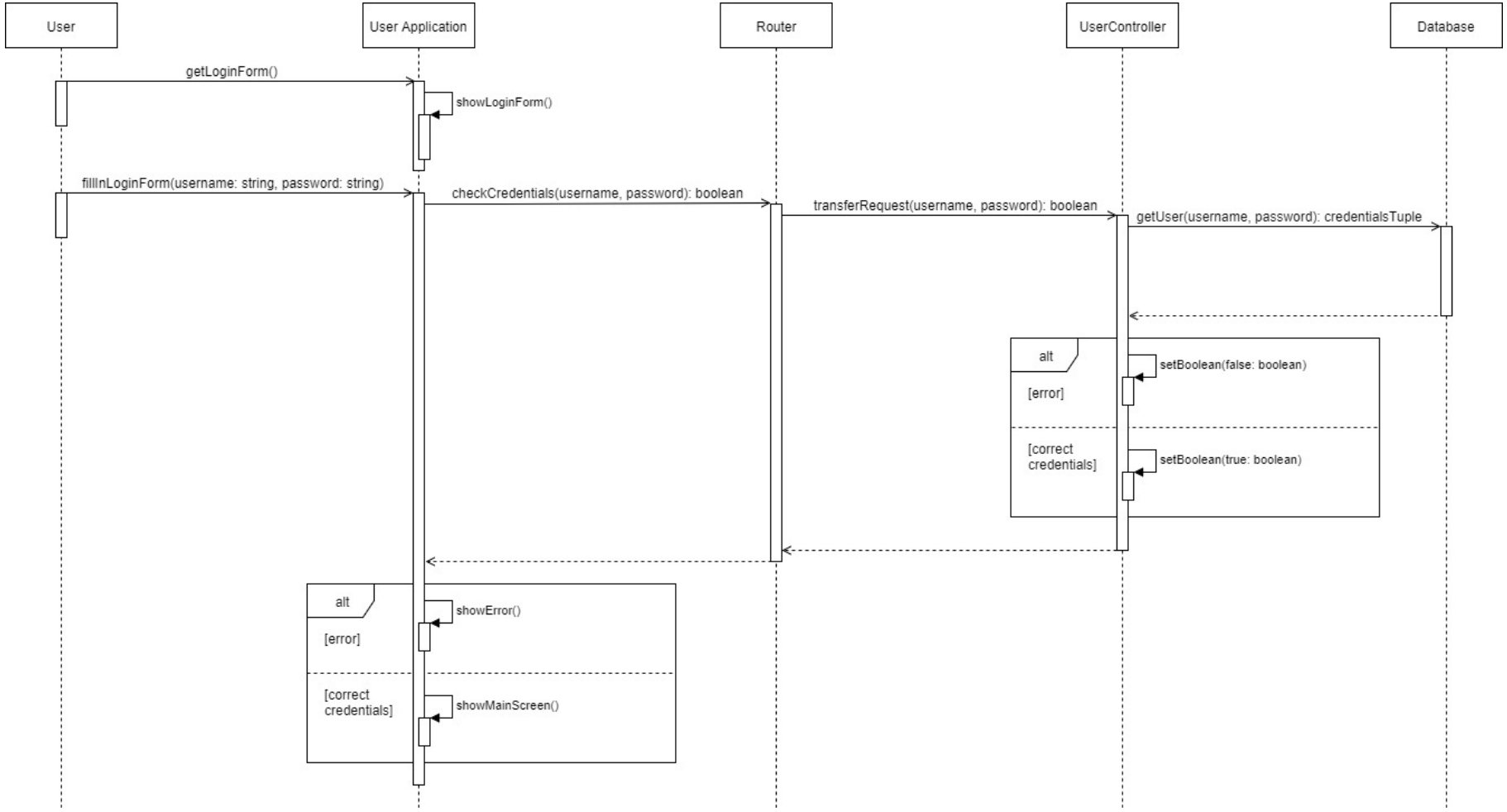

Figure 2.4: User Login runtime view

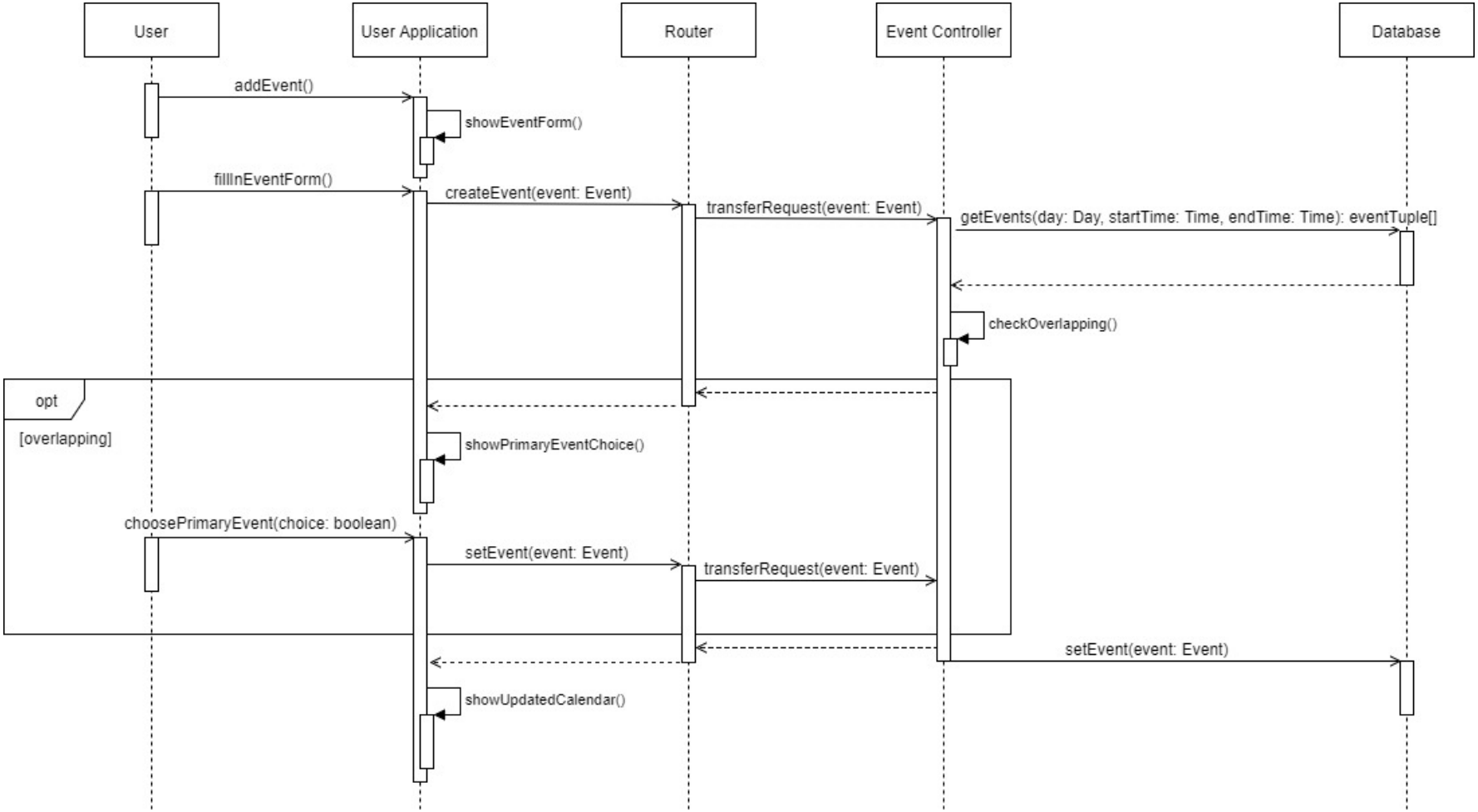

Figure 2.5: Adding of an event in the calendar runtime view

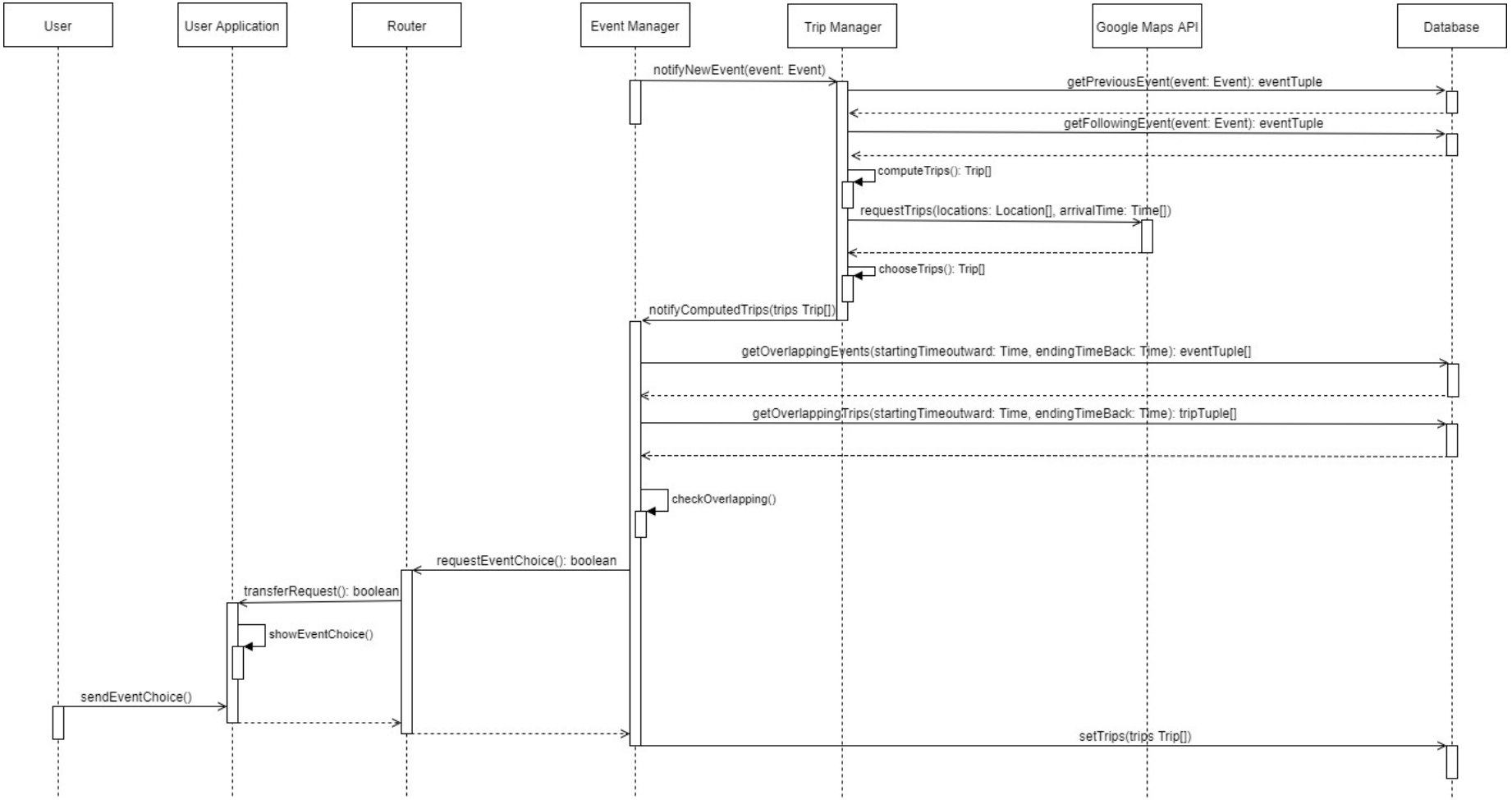

Figure 2.6: Trip planning runtime view

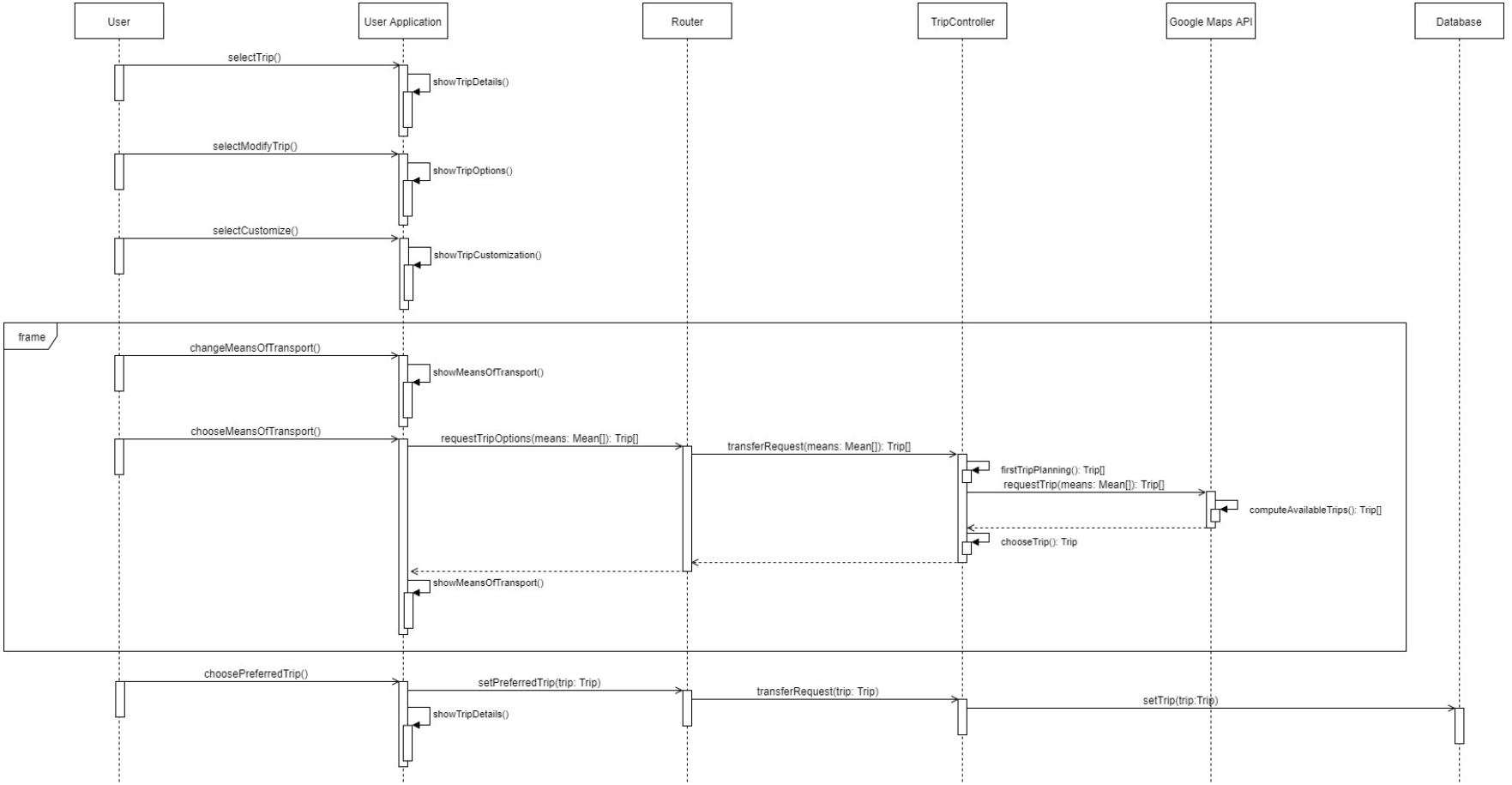

Figure 2.7: Trip customization runtime view (I assume that the trip is already saved on the device, because otherwise it will have to be synced from the database)

2.E Component Interfaces

- User Application Interface: responsible for communication between the user application and the application server. The communication will be provided with the HTTPS protocol mostly for security reasons;
- Open Weather Map: it gives information about the weather and then the system task will be to compute trips looking also to these pieces of information. For example, if it will be raining it will not suggest a bike trip;
- Strikes Information Provider: it gives to our system information about all the possible strikes concerning public means and then it will be the system that will use this information to compute the best travel option for the user;
- Google Maps API: it's used for:
  1. calculate all the trip times associated to each event, obviously for all types of means (car, bike, public means etc.);
  2. provides information about the traffic to optimize the computation of trip times;
  3. provides the static map that the user will see.
- DBMS API: through this the system can retrieve and write data from and to the database. It's strictly connected to the model through JDBC connector.

2.F Selected Architectural Styles and Patterns

2.F.1 Architectural style: Client-Server

It's the best known and most used architectural style for distributed applications. It's present a well-defined distinction between client and server, they play different roles and also, they are both accessible with a precise interface. Travlendar+ is a mobile application and will have multiple mobile users, but still the computation portion needs to be located in some point where the global view of the system can be seen. The system must guarantee scalability so, resources in the form of network segments, computers and servers must be added to the network without major interruptions of it. Said

that, we agreed that the best solution for our needs could be the Client-Server style.

- Server is invoked to provide one or a set of services, in our case, for instance, the computation of the best mobility option for the user according to all of his characteristics;

- Clients uses the provided services and initiate the communication through messages or remote invocations. They interact directly with end-users using any user interface such as graphical user interface. These are the user logged in to the application.

In the end, we will talk about web clients, which means that in our scenario, they will ask the server to provide services for them and they will not store data locally. Furthermore, the architecture is OS independent, it relies on a central server that will avoid consistency issues between different devices of the same client.

2.F.2 Three-tier architecture

The Client-Server model does not impose any constraint neither about how logical layers (presentation, application/business logic, data) have to be distributed among the deployment units nor about the physical tiers have to be designed. We taught that the best decision for us could have been the three-tier architecture where each tier is elastically scaled independently. That's composed of:

- First tier: Presentation layer located into the mobile application. Handles the interaction with the user. It's also in charge of some simple validations of the data and the interaction with external systems. Here will be handled all the visualization portion of the system based on the data received from the application layer;

- Second tier: web/business layer. Process and executes both new requests of the clients and possible variations on older and not yet completed requests. It will collect information from the users and store them into the databases, but also it will provide to them answers for their requests. More precisely, this layer's tasks are to filter, cache and forwarding the requests between clients and the application server and vice versa. It's also responsible (web layer) of the visualization to the user;

- Third tier: Data layer. Dedicated to the storage of information in persistent memory. Data is stored at different locations (replicas) to improve response time and to avoid data loss in case of failures while consistency of replicas is ensured at all times.

This architecture permits us to achieve one of the design principles, that says to decouple where possible, in fact we have a solid distinction between logic and data, and also between logic and presentation.

2.F.3 Basic architectural patterns

Stateless components

The state is handled external of application components to ease their scaling-out and to make the application more tolerant to component failures. In this system cloud resources can display low availability and also component instances can be added and removed regularly when the demand changes. So, components will be implemented in a way that the do not have any internal state. Instead it will be provided to the component with each request form external persistent storage.

Stateless components

An interactive synchronous access to the applications is provided to users, instead the interactions in the application are asynchronous if possible, to ensure loose coupling (Principle 3 of the design principles). In other words, the user interface component is like a bridge between the synchronous access of the user and the asynchronous communication with other components. Like for stateless components also these ones are obtained from external storage. The number of components are scaled based on the number of user requests through the elastic load balancer pattern.

Elastic load balancer

The number of required components instances is determined by the quantity of synchronous request coming from the users.

Processing component

Processing functionalities are divided into separate blocks and assigned to independent processing components each one implemented in a stateless way as explained above.

Elastic queue

Queues are used to distribute asynchronous requests to multiple application component instances. From the number of messages queued will be decided the number of component instances handling that requests.

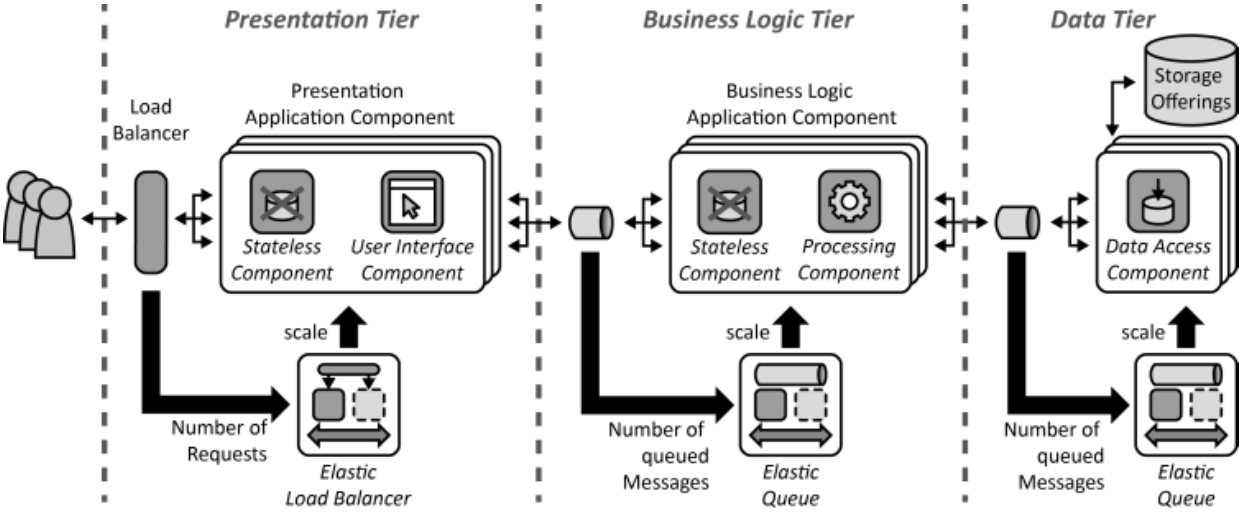

Figure 2.8: Three tier cloud application

2.G Architectural pattern

Model View Controller

The model-view-controller (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

- The model contains only the pure application data, it contains no logic describing how to present the data to a user, but only data and operations associated to that;
- The view presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it;
- The controller exists between the view and the model. It listens to events triggered by the view (so by the user) and executes the appropriate response to these events, usually calling a method on the model and passing the results both to the view and the model.

A key aspect is to separate data and its presentation. This not only makes the structure of an application simpler, it also enables code reuse and ensures loose coupling. It also guarantees the divide and conquer principle allowing parallel development by separated teams in charge of different parts of the application.
An example of how can be represented the MVC pattern is shown in the figure below.

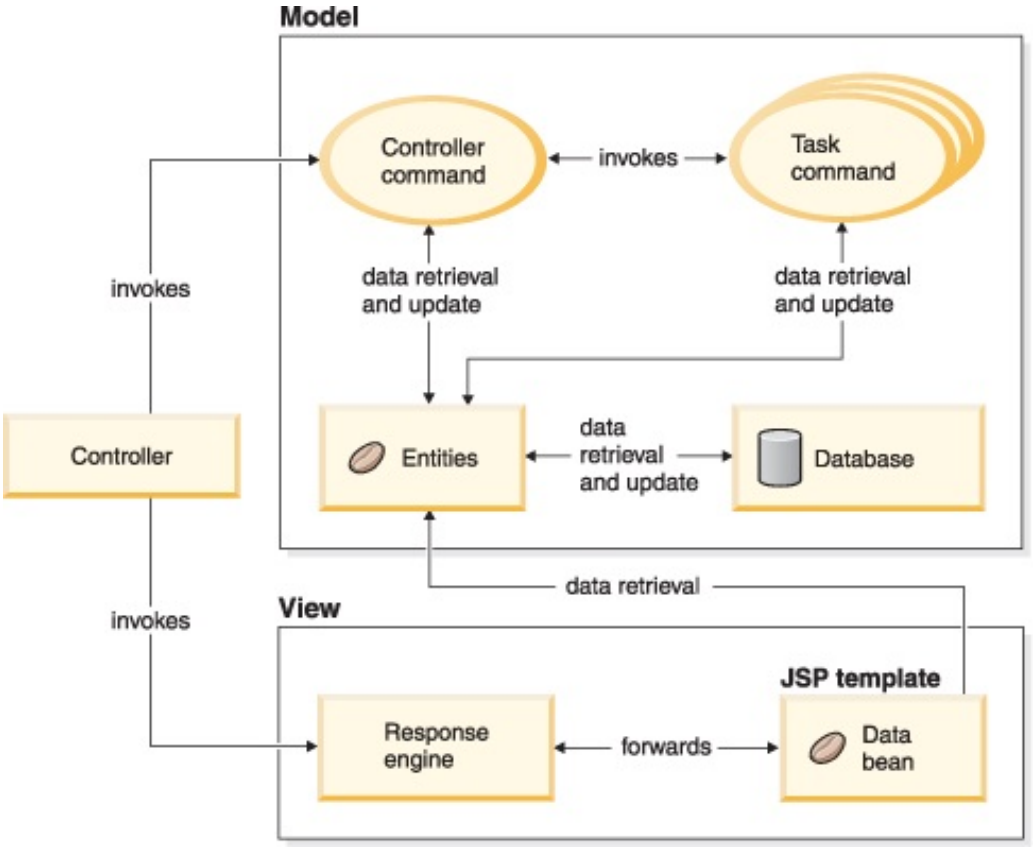

Figure 2.9: Model view controller

Commercial architectural system

To support the development and the execution on enterprise application with a lot of users and lots of requirements we suggest the usage of Java Enterprise Edition (JEE). That's a multitier architectural model composed by the client-tier running on the client machine (as we said before in this paragraph, the clients involved in Travlendar+ will be web clients), the business-tier running on the Java EE Server and the enterprise information system (EIS) consisting of databases or other external applications. This approach will help

us also to satisfy non-functional requirements like reliability, performance, security, scalability, availability, extensibility and interoperability.

The client tier will welcome web clients composed of dynamic web pages, which are generated by web components running in the web tier and a web browser, which renders the pages received from the business layer. The application tier will be divided into:

- The web layer: in charge of the presentation of the data to the user, it will collect the responses of a client request coming from different servers in the distributed business tier and will send them to the correct user written more likely in JSP or Java Server Faces. It will also be in charge of firewall, cache, proxy and reverse proxy functions;

- The business layer: managing the computing and the execution of the business logic using components called Enterprise Java Beans (EJB) and also interacting with the database through Java Persistence API (JPA).

In the end, the Enterprise Information System (EIS) is devoted to the data management and will operate like a DBMS.

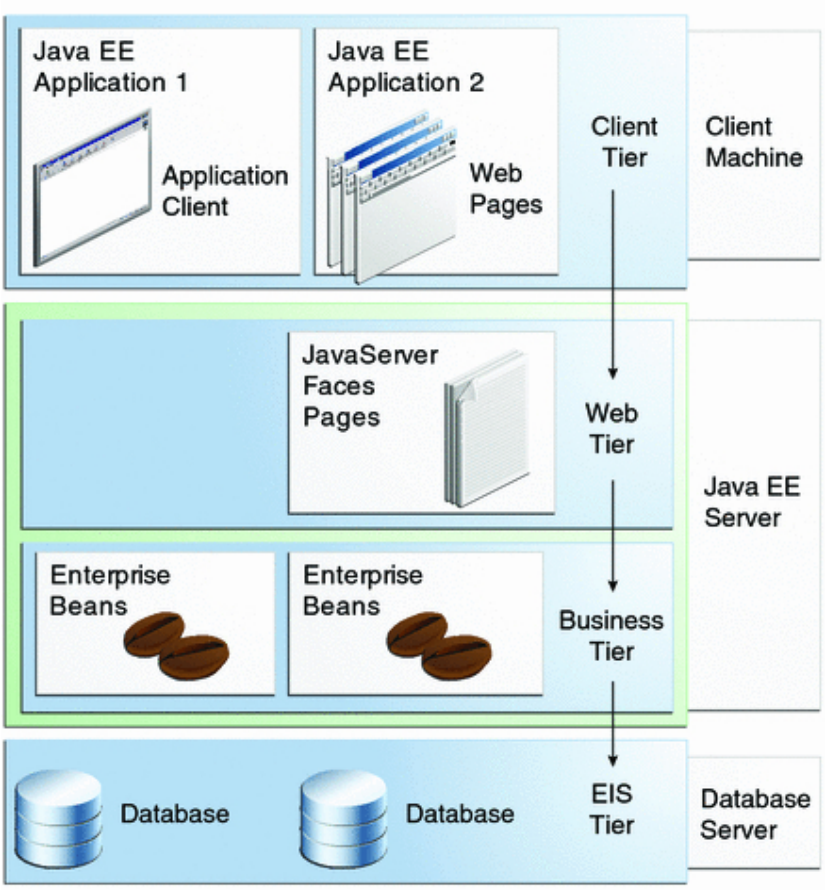

Figure 2.10: ommercial architectural system

2.H Other Design Decisions

Push Notifications

To send notifications to the Travlendar+ users we choose the Push Notification Technology, obviously supported by iOS and Android. The APIs we are going to use are:

- Apple Push Notification Service for iOS;
- Google Cloud Messaging for Android.

Chapter 3

Algorithm Design

In this section we show the idea behind the algorithms structure of the core functions of the system. We choose to elaborate on the processes concerning the creation of a new event, the update of the planned trips of all user's events, the trip planning, the calculation of the travel time for shared means of transport (due to the fact that it is not directly obtainable from Google Maps API) and the users's dynamic events scheduling, which are the main objectives of the system. After the user inputs the data of a new event, the system will first run the overlapping check algorithm, which will than run the trips update algorithm to keep them up to date. This second process is of great importance, and will be run every time there is a change in preferences, events or trips, in addition of a daily update. The main block of the update algorithm is the trip planning one, checking the feasibility of the trips and suggesting the best way to reach an event. This one will be run more frequently than usual when it is almost time to start a trip. We decided to show a flow chart with processes, serving as code description to ease the understanding, due to the more schematic form in blocks. The code is written in a pseudo java inspired on the probable class structure of the system. The objective is to show the reader the logic under the decision precesses of our sistem, made to be as practical and human-like as possible when proposing the beat travel solution.

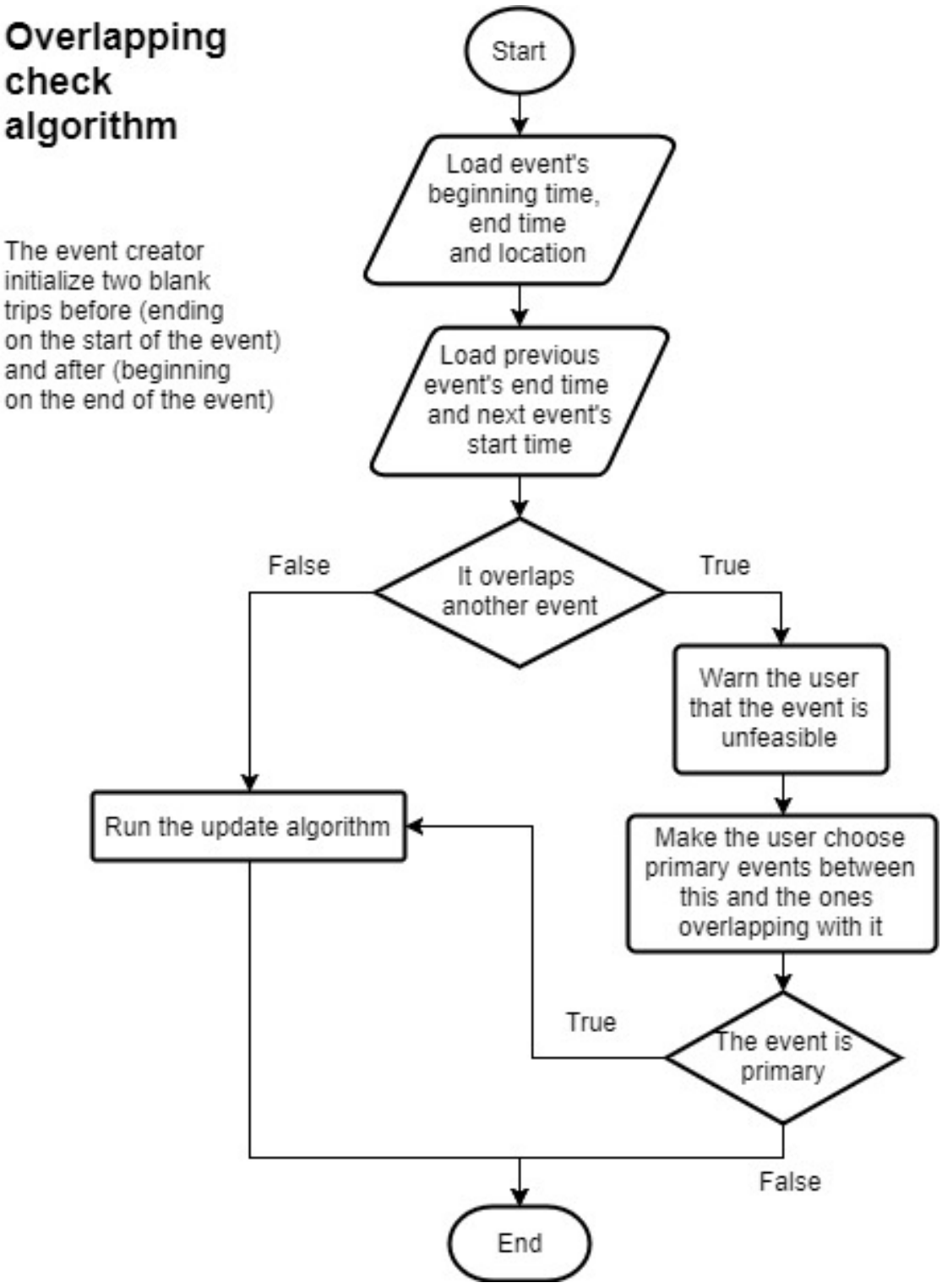

Figure 3.1: Overlap check algorithm

```
1
2  /*
3  *   Event is already initialized with to/from trip;
4  *   by default the trip to the event starts from home
5  *   and it ends on event location, instead the trip from
6  *   the event starts from event location and goes to home.
7  */
8
9  public void overlappingCheckAlgorithm(Event event){
10   ArrayList<Event> overlap = personalEventsController.
         getOverlappingEvents(event);  // in SQL !((TStart1>=TEnd)
         ||(TEnd1<=TStart))
11   if (!overlap.isEmpty()){
12     overlappingWarning(event);
13     if(getPrimaryEventChoice(event,overlap)){  // 1 if the event
           is chosen as primary instead of the ones it overlaps
           with
14       personalEventsController.removePrimaryEvent(overlap);
15       personalEventsController.addSecondaryEvent(overlap);
16       personalEventsController.addPrimaryEvent(event);
17       tripsUpdateAlgorithm();
18     } else {
19       personalEventsController.addSecondaryEvent(event);
20     }
21   } else {
22     personalEventsController.addPrimaryEvent(event);
23     tripsUpdateAlgorithm();
24   }
25 }
```

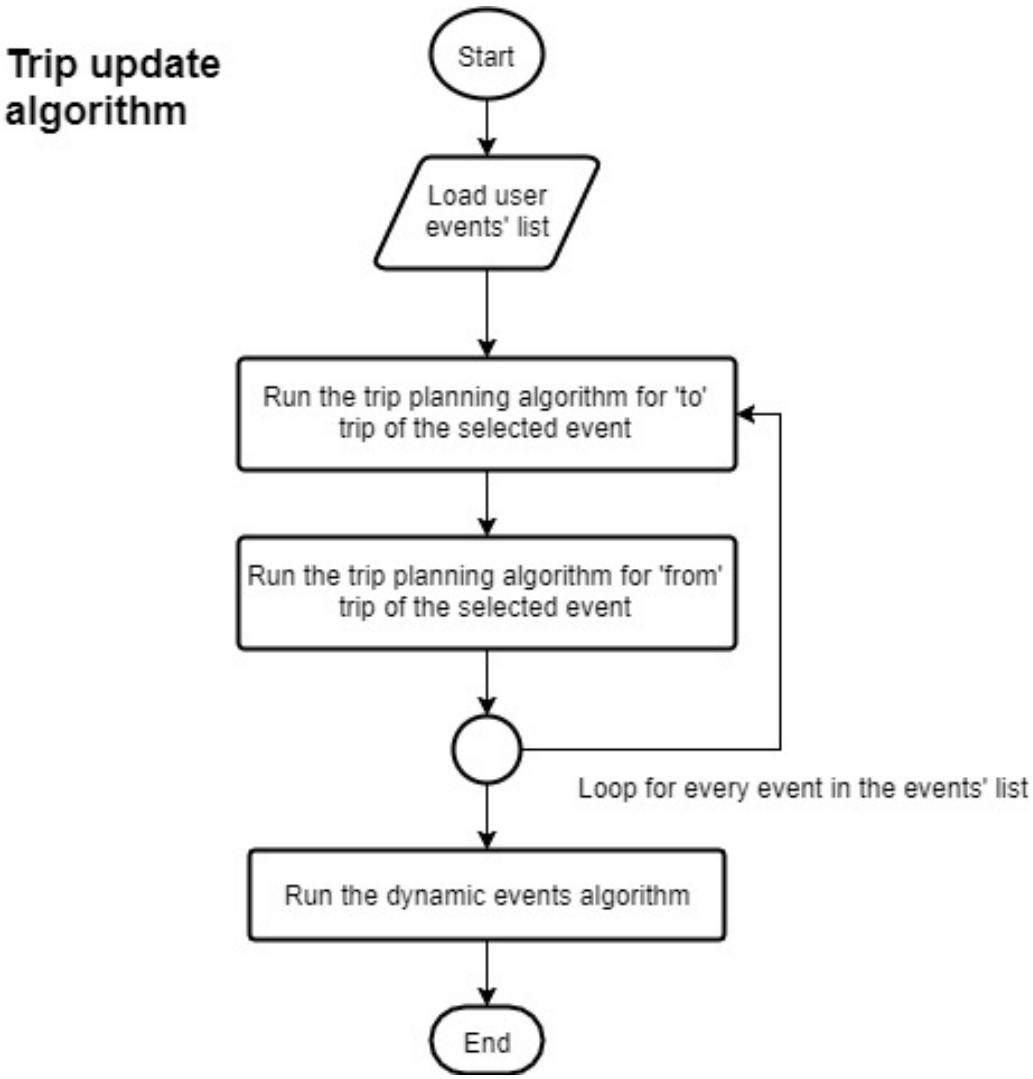

Figure 3.2: Trip update algorithm

```
public void tripsUpdateAlgorithm(){
  for(Event e : personalEventsController.getEventsList()){
    tripPlanningAlgorithm(e.getToTrip(), e);
    tripPlanningAlgorithm(e.getFromTrip(), e);
  }
  dynamicScheduleAlgorithm();
}
```

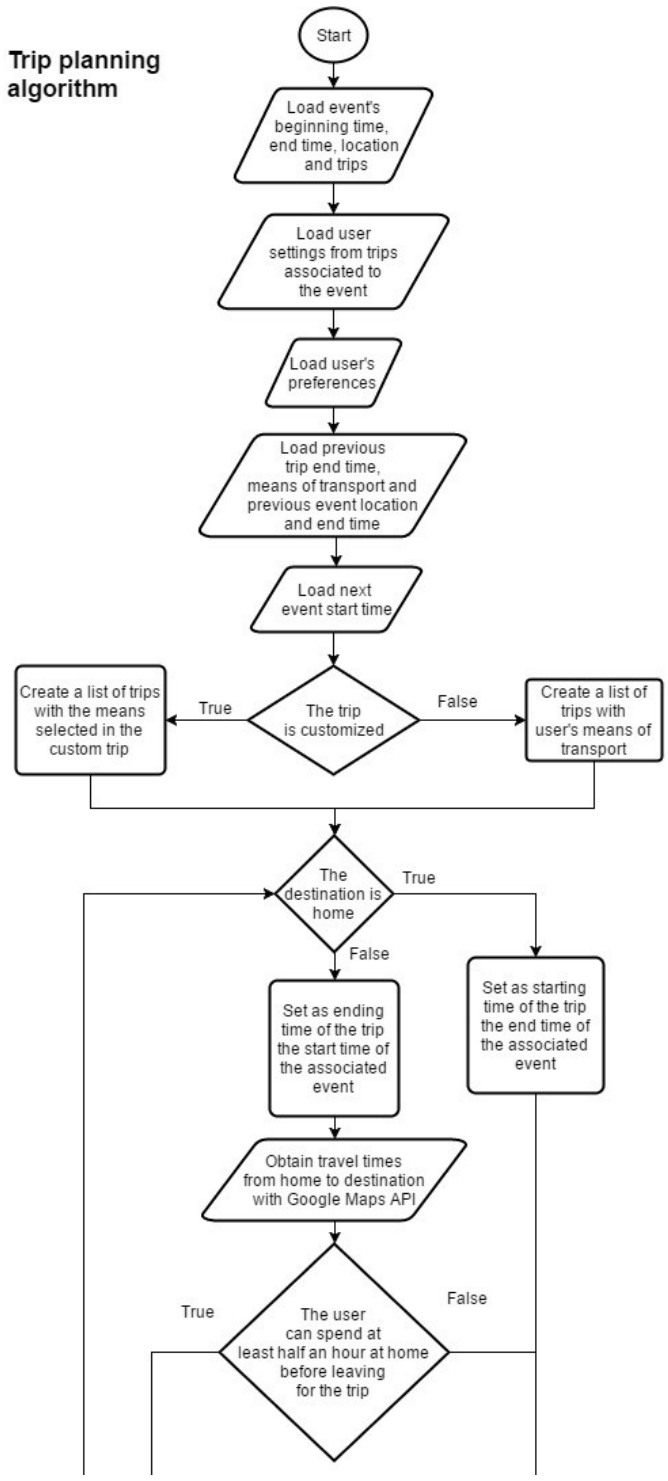

Figure 3.3: Trip planning algorithm part 1

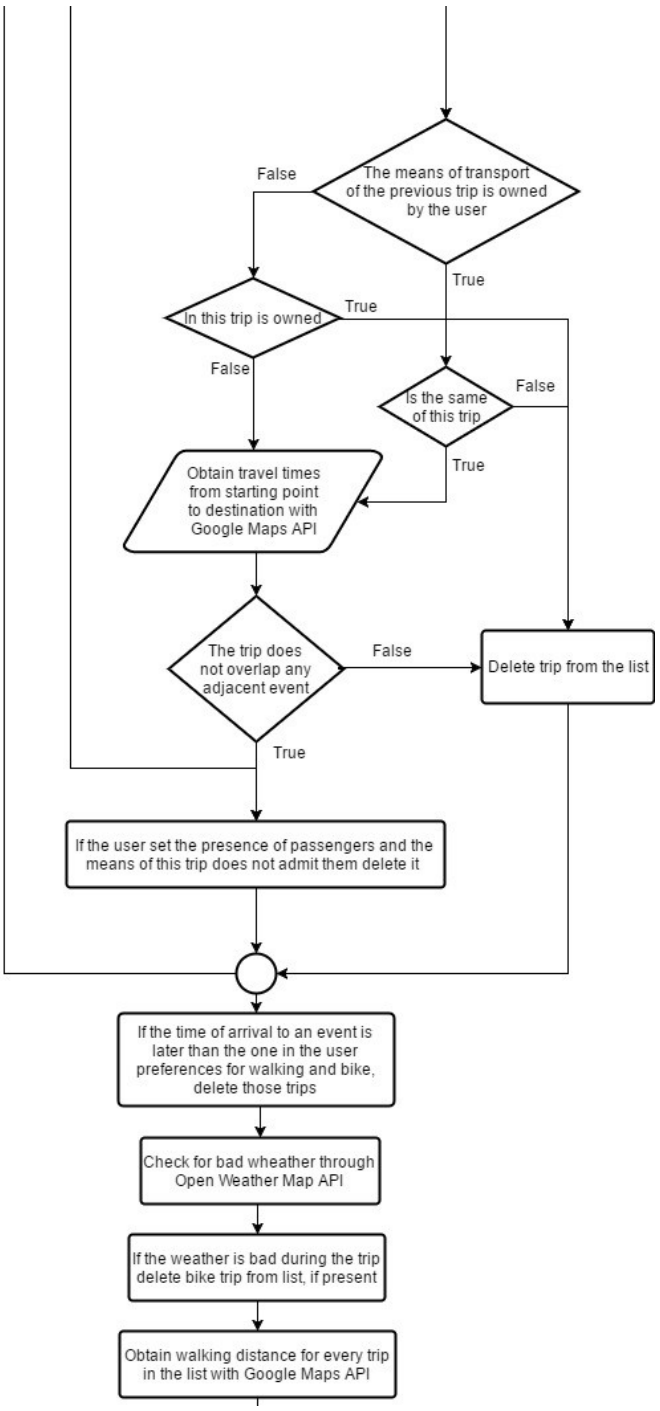

Figure 3.4: Trip planning algorithm part 2

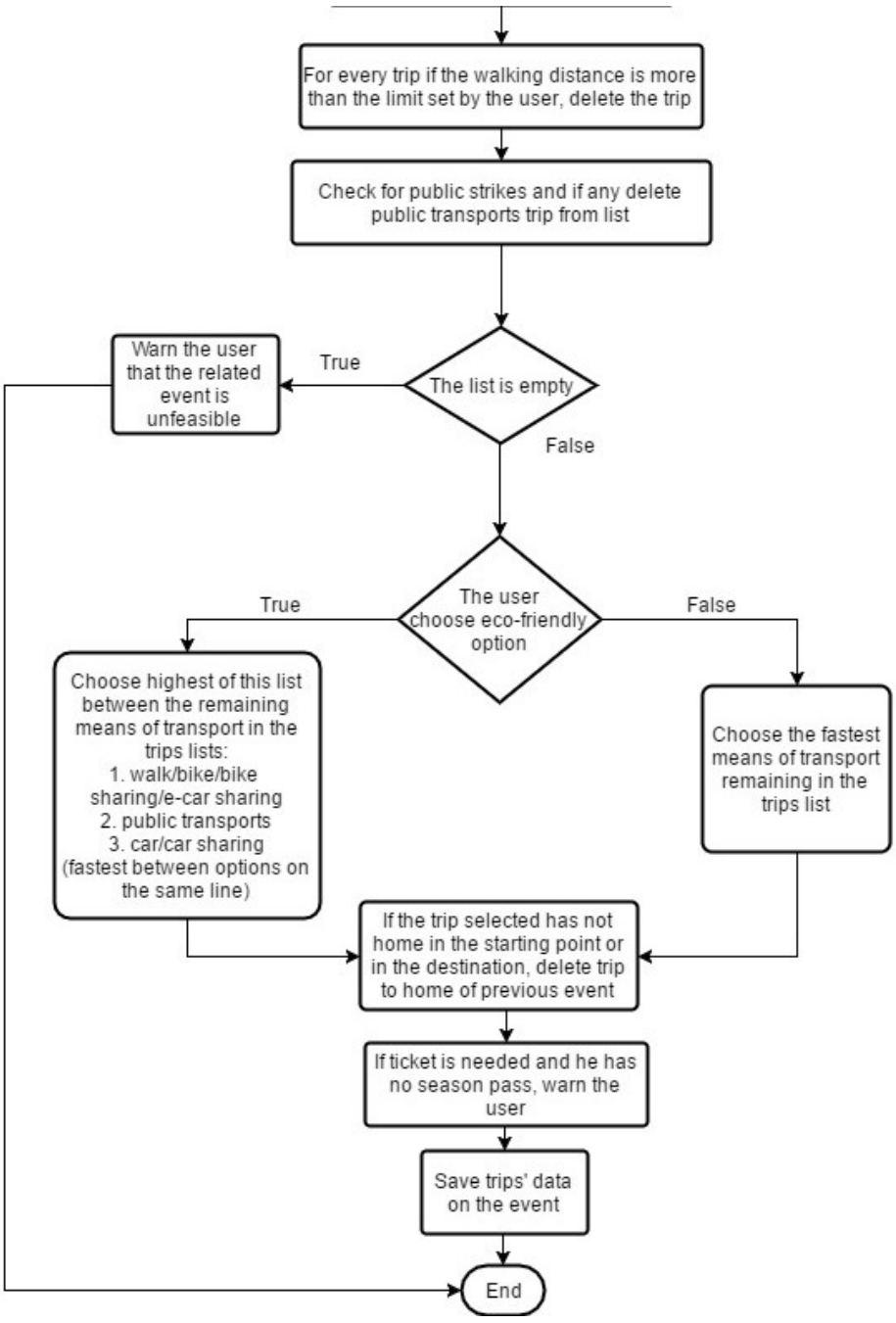

Figure 3.5: Trip planning algorithm part 3

```
1
2  public void tripPlanningAlgorithm(Trip t, Event e){
3
4    ArrayList<Trip> trips = new ArrayList<>();
5    ArrayList<MeansOfTransport> motList;
6    Trip previousTrip;
7
8    if(t.isCustom()){
9      motList = t.getCustomMOTs();
10   } else {
11     motList = user.getPreferences().getMOTList();
12   }
13
14   for(MeansOfTransport mot : motList){
15     Trip t1 = new Trip(t,mot);
16
17     if(user.getHomeLocation().isEqual(t.getDestination())){
18       previousTrip = e.getToTrip();
19       t1.setStartTime(Time.addMinutes(e.getEndTime(), 5));
20       mapsController.obtainArrivalTimeFromDepartureTime(t1);
21     } else{
22       previousTrip = personalEventsController.getPrevious(e).
             getFromTrip();
23       t1.setEndTime(Time.addMinutes(e.getStartTime(), -5));
24       mapsController.obtainDepartureTimeFromArrivalTime(t1);
25
26       if(Time.getBetweenMinutes(t1.getStartTime(), previousTrip.
             getEndTime()) >= 30){
27         trips.add(t1);
28         break;
29       }
30       t1.setStartingLocation(personalEventsController.
             getPrevious(e).getLocation());
31       t1.setStartTime(Time.addMinutes(personalEventsController.
             getPrevious(e).getEndTime(), 5));
32       mapsController.obtainArrivalTimeFromDepartureTime(t1);
33       if(t1.getEndTime().isGreater(Time.addMinutes(e.
             getStartTime(), -5))){ break; }
34     }
35
36     if(previousTrip.getMOT().isPersonal()){
37       if(!t1.getMOT().isEqual(previousTrip.getMOT())){ break; }
38     } else {
39       if(t1.getMOT().isPersonal()){ break; }
40     }
41
42     if(e.getStartTime().isLaterThan(user.getPreferences.
           getMaxTimeBikeWalk()) && (t1.getMOT.isEqual(
```

```
        MeansOfTransport.BIKE) || t1.getMOT.isEqual(
        MeansOfTransport.WALK))){ break; }
43
44      if(badWeather(t1.getStartingLocation(), t1.getStartTime())
        && t1.getMOT.isEqual(MeansOfTransport.BIKE)){ break; }
45
46      if(mapsController.getWalkDistance(t1) >= user.getPreferences
        ().getMaxWalkDistance()){ break; }
47
48      if(checkStrikes(t1.getStartingLocation(), t1.getStartTime())
         && t1.getMOT.isEqual(MeansOfTransport.PUBLICTRANSPORT)){
         break; }
49
50      if(t.hasPassengers() && !t1.getMOT().admitPassengers()){
        break; }
51
52      trips.add(t1);
53    }
54
55    if(trips.isEmpty()){
56      unfeasibilityWarning(e); // warn the user he cannot come to
          this event in time from the previous
57      personalEventsController.removePrimaryEvent(e);
58      personalEventsController.addSecondaryEvent(e);
59    }
60
61    if(t.isEcoFriendly()){ // t is initialized with the lowest
        EcoPoints (scale assigned to every means of transport) and
        with a travel time of 1 day
62      for(Trip t1 : trips){
63        if(t1.getMOT().getEcoPoint() > t.getMOT().getEcoPoint()){
64          t = t1;
65        } else if(t1.getMOT().getEcoPoint() == t.getMOT().
            getEcoPoint() && t1.getTravelTime() < t.getTravelTime()
            ){
66          t = t1;
67        }
68      }
69    } else {
70      for(Trip t1 : trips){
71        if(t1.getTravelTime() > t.getTravelTime()){ t=t1; }
72    }
73
74    if(!(user.getHomeLocation().isEqual(t.getDestination()) ||
        user.getHomeLocation().isEqual(t.getStartingLocation()))){
75      personalEventsController.getPrevious(e).setFromTrip(t);
76    }
77
78    if(t.getMOT().needsTicket() && !user.getPreferences().
```

```
        hasSeasonPass(t1.getMOT())){
79      ticketWarning(t);
80    }
81  }
```

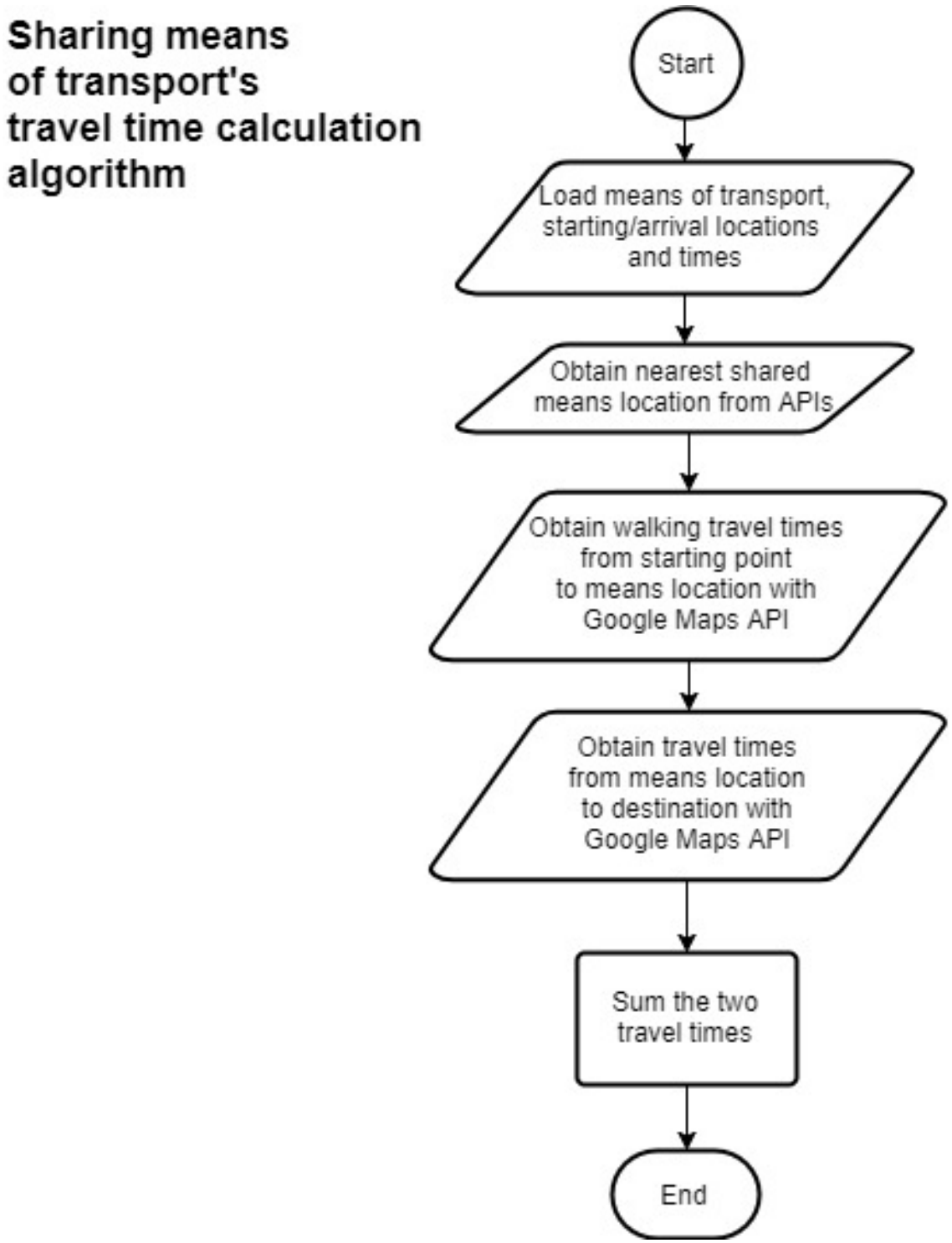

Figure 3.6: Sharing means of transport algorithm

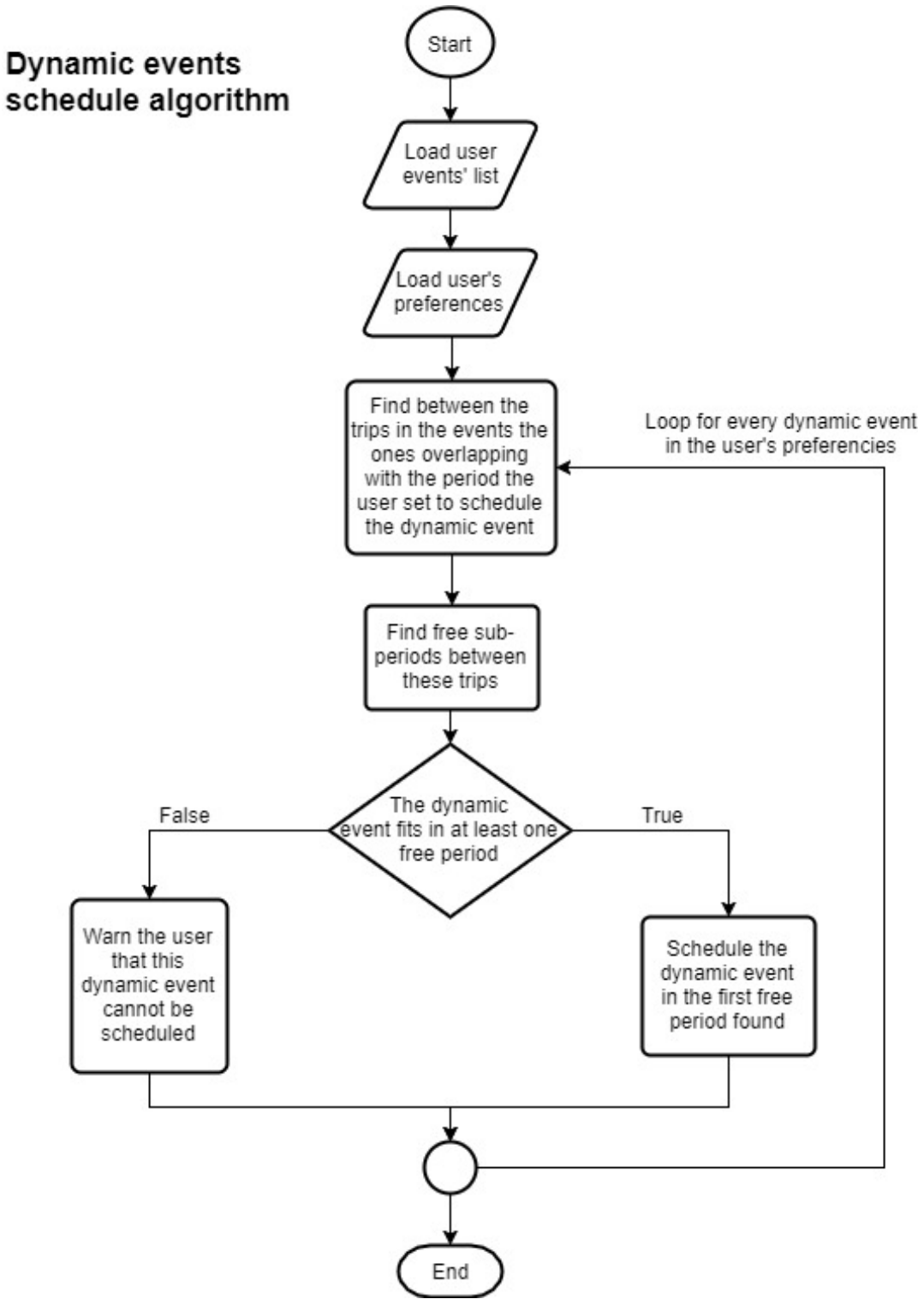

Figure 3.7: Dynamic event schedule algorithm

```
1
2  public void dynamicScheduleAlgorithm(){
3    Iterator dEvents = user.getPreferences().getDynamicEventsList
         ().iterator();
4    personalEventsController.resetDynamicEvents();
5    DynamicEvent d = dEvents.next();
6    for(Event e :  personalEventsController.getEventsList()){
7      while(e.getFromTrip().getEndTime().isAfterThan(d.
           getBeginningTimeOfPeriod())){
8        if(personalEventsController.getNext(e).getToTrip().isEqual
             (e.getFromTrip())){
9          if(Time.getBetweenMinutes(personalEventsController.
               getNext(e).getStartTime(), e.getFromTrip().getEndTime
               ()) >= d.getBreakMinutes()){
10           if(!personalEventsController.
                 isOverlappingAnotherDynamicEvent(d)){
11             personalEventsController.addDynamicEvent(d, e.
                   getFromTrip().getEndTime());
12             d = dEvents.next();
13           } else {
14             dynamicUnfeasibilityWarning(d);
15           }
16         }
17       } else {
18         if(Time.getBetweenMinutes(personalEventsController.
               getNext(e).getToTrip().getStartTime(),  e.getFromTrip
               ().getEndTime()) >= d.getBreakMinutes()){
19           if(!personalEventsController.
                 isOverlappingAnotherDynamicEvent(d)){
20             personalEventsController.addDynamicEvent(d, e.
                   getFromTrip().getEndTime());
21             d = dEvents.next();
22           } else {
23             dynamicUnfeasibilityWarning(d);
24           }
25         }
26       }
27     }
28   }
29 }
```

Chapter 4

User Interface Design

This section is a recapitulation of the section 3.B.1 (User Interfaces) of the Requirements Analysis and Specification Document and a deepening of design aspects of the user interface. The application will be developed as a mobile application for the main mobile operating systems (iOS and Android). As the system will appear the same for all the users, it will provide all the functionalities described in the RASD, in a unique user interface.

4.A Mockups

Following some mockups will provide an idea of the user interface while the user interacts with it, making use of the functionalities of the application. It includes most of the main screen that the user will face.

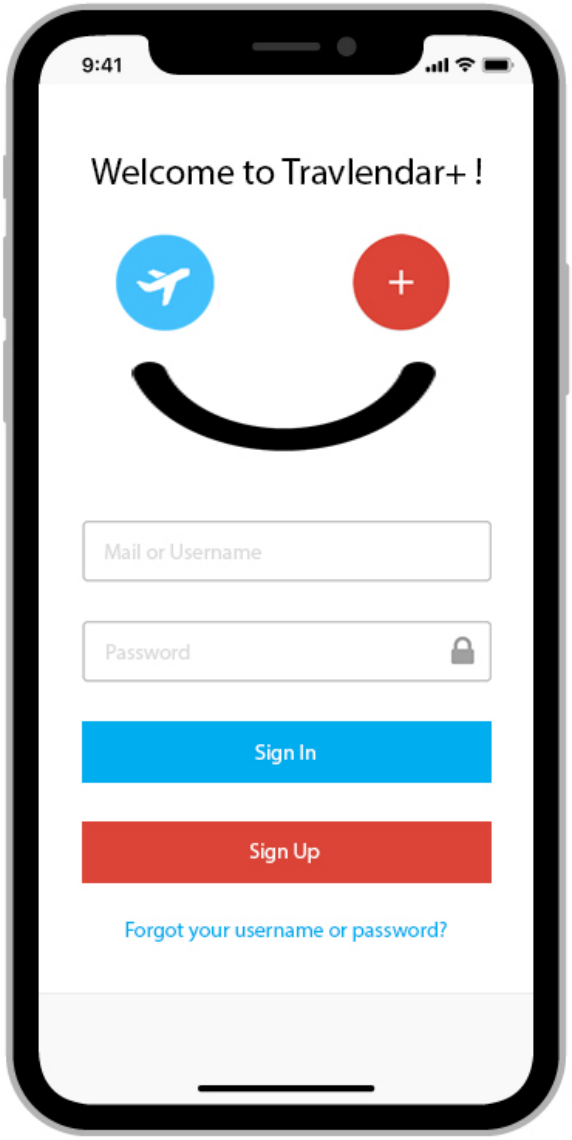

Figure 4.1: Mockup of the login screen

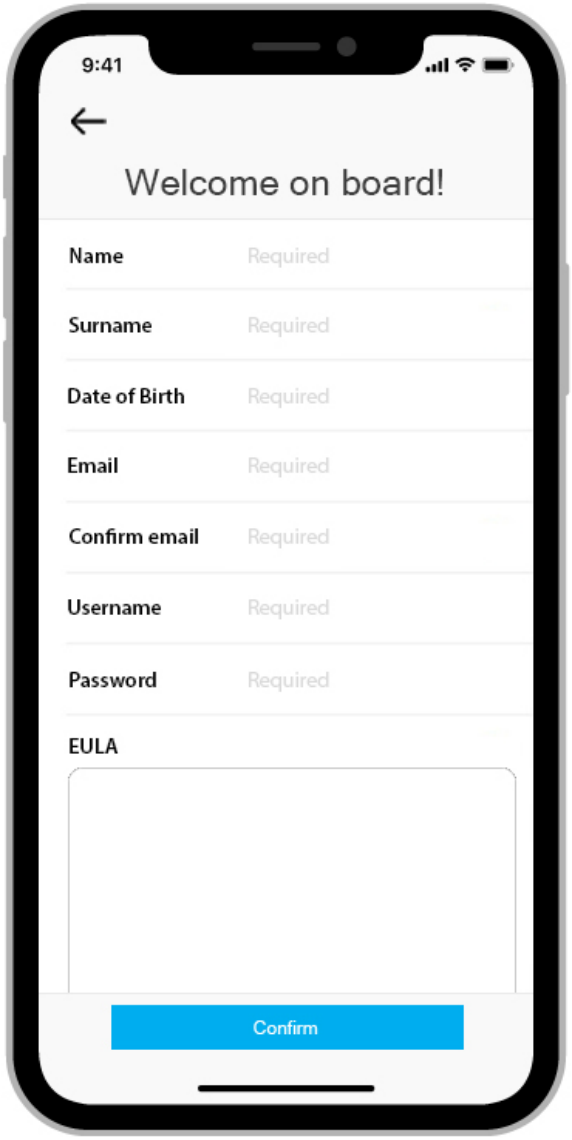

Figure 4.2: Mockup of the registration screen

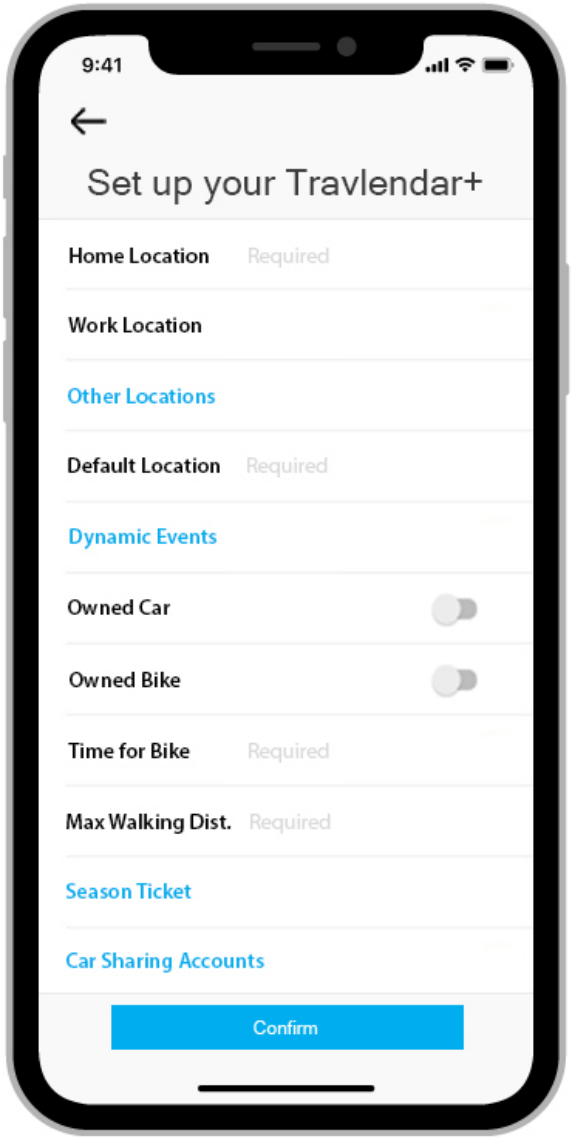

Figure 4.3: Mockup of the screen where the user can set his preferences

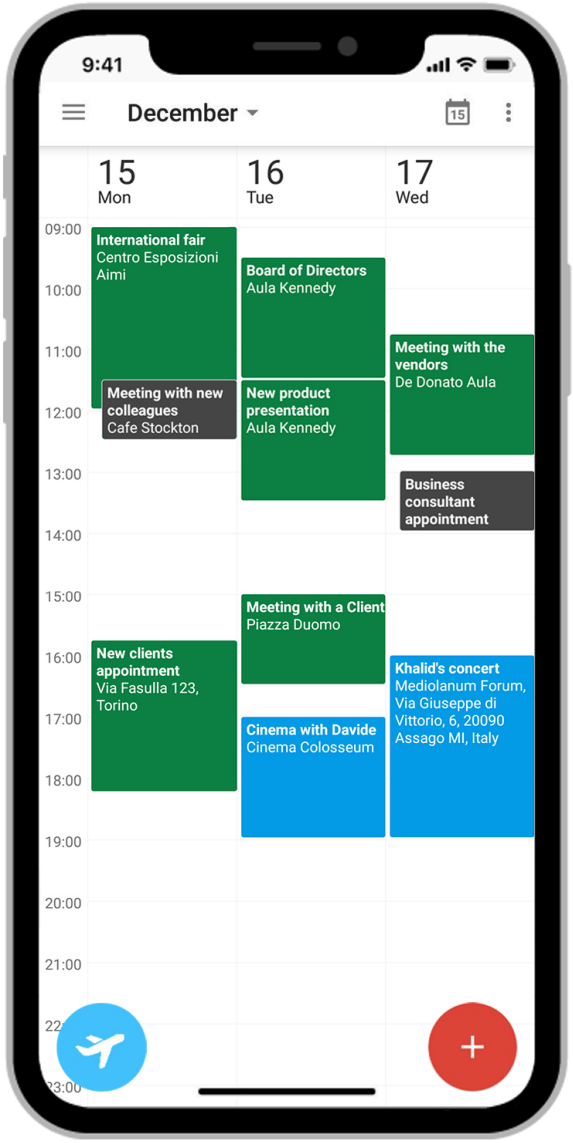

Figure 4.4: Mockup of the calendar screen with primary and secondary events

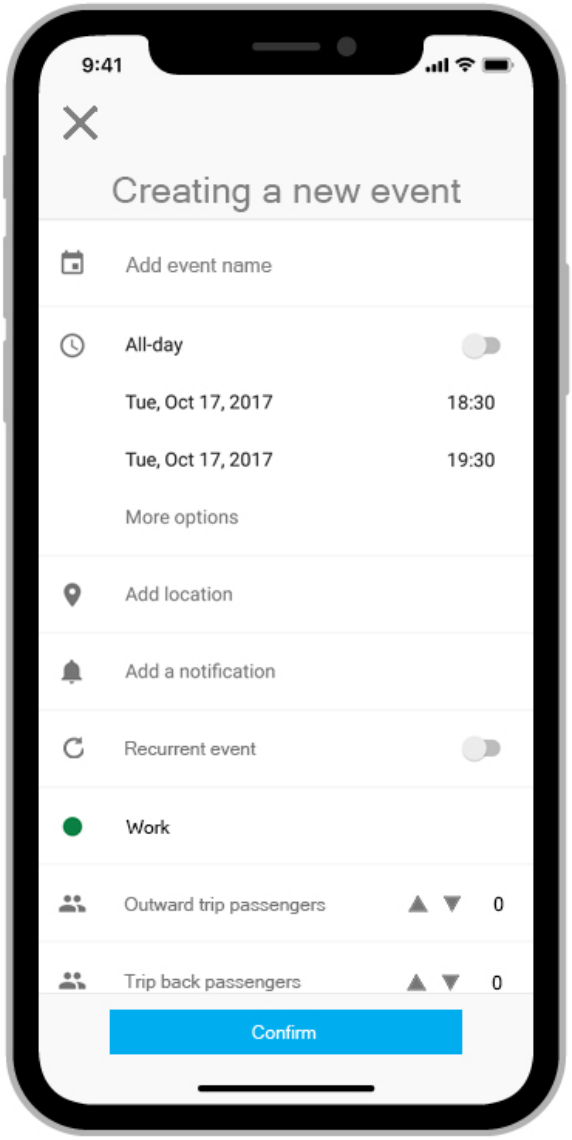

Figure 4.5: Mockup of the screen that allows to add an event

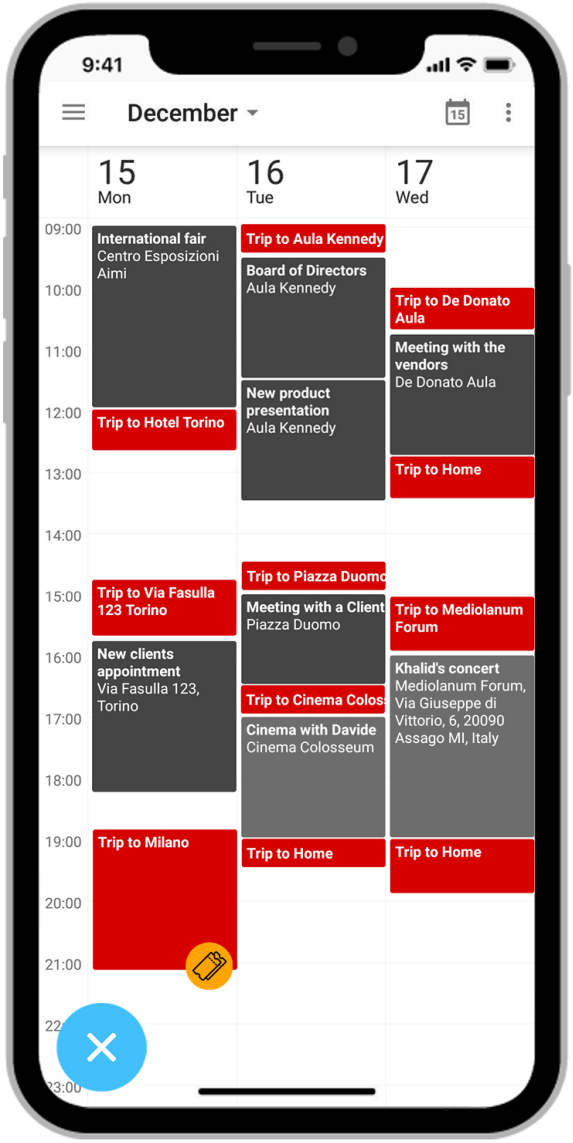

Figure 4.6: Mockup of the screen with the trips shown

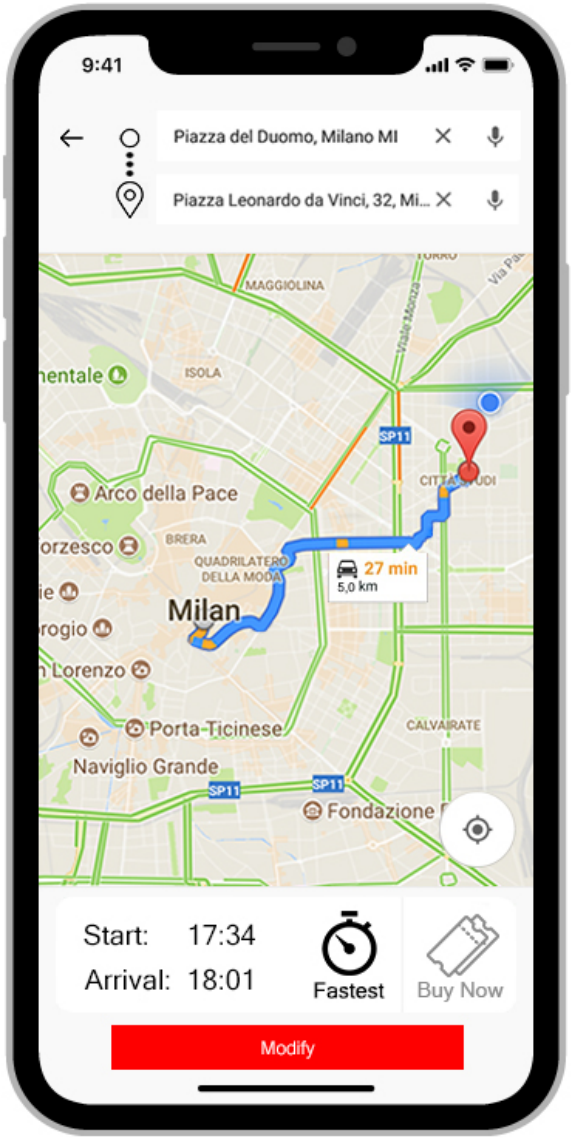

Figure 4.7: Mockup of the trip details screen

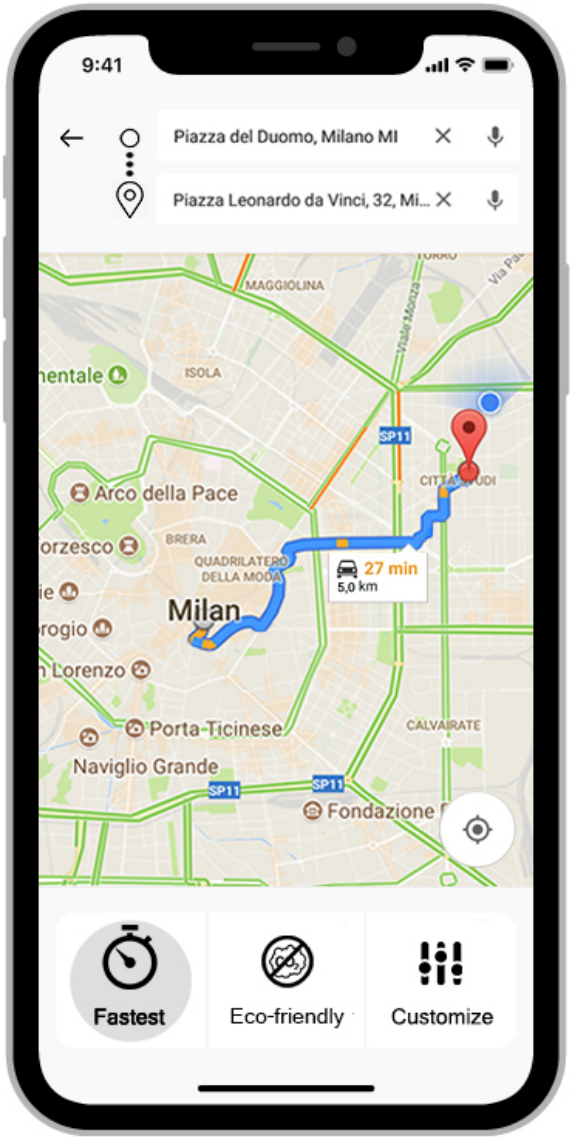

Figure 4.8: Mockup of the screen that allows the user to modify a trip

Figure 4.9: Mockup of the screen with the customization of a trip

4.B UX Diagrams

UX diagrams provide information about the user interface of the system and how the user interacts with it. For the diagram comprehension purposes, additional screens used to add specific information (other locations, dynamic

events, season tickets, etc. . .) in the preferences setup are not considered and different ways to buy tickets (opening browser or calling an external application) or reserve a sharing vehicle or bike are modeled as a single object.

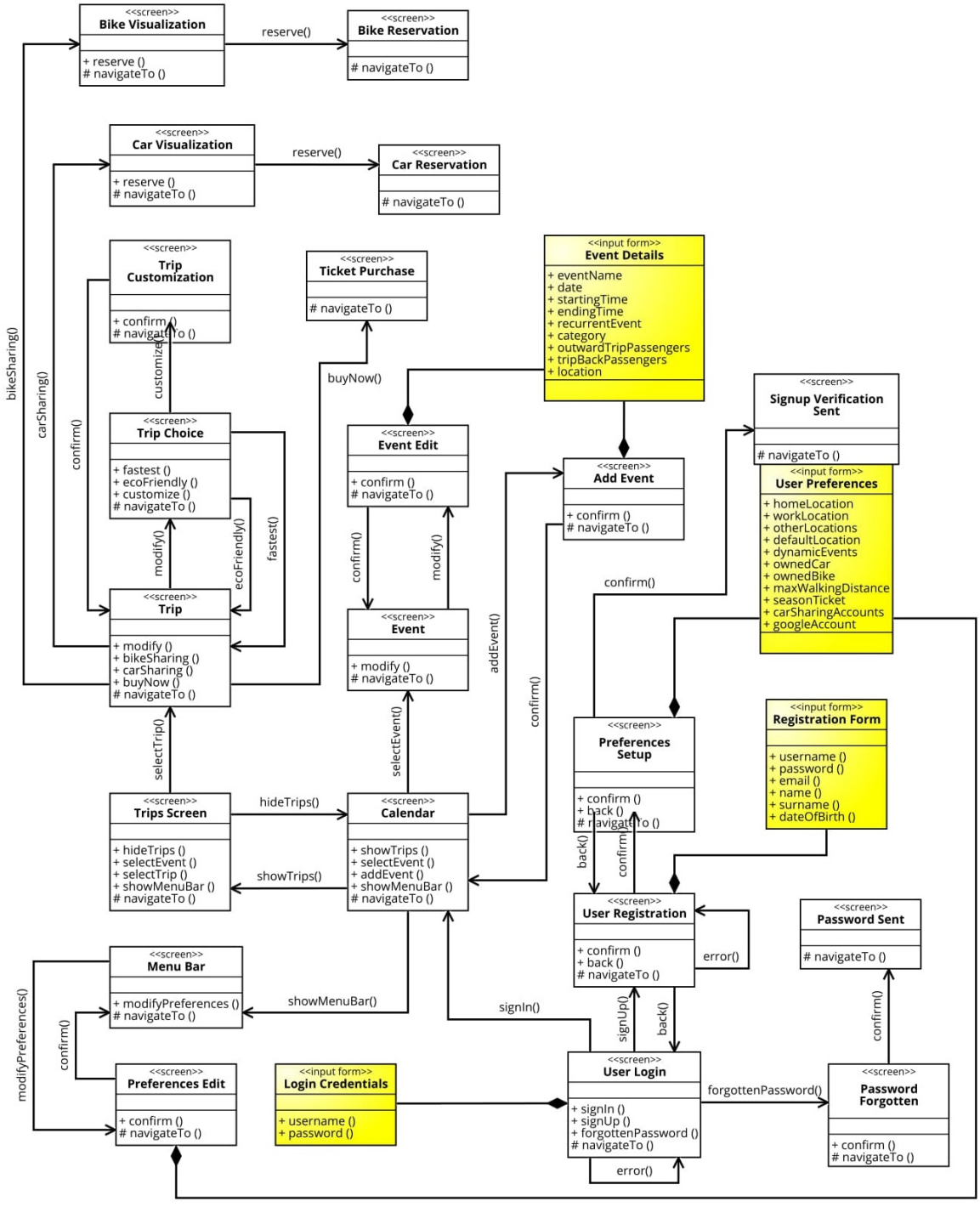

Figure 4.10: UX diagram of the application

4.C BCE Diagrams

For the implementation of the system, a Model-View-Controller design pattern is adopted and BCE diagrams are useful to show how user interactions are managed internally by the system. Boundaries are objects that interface with the users of the application; Entities object model the access to data; Controls object manage the communication between boundaries and entities.

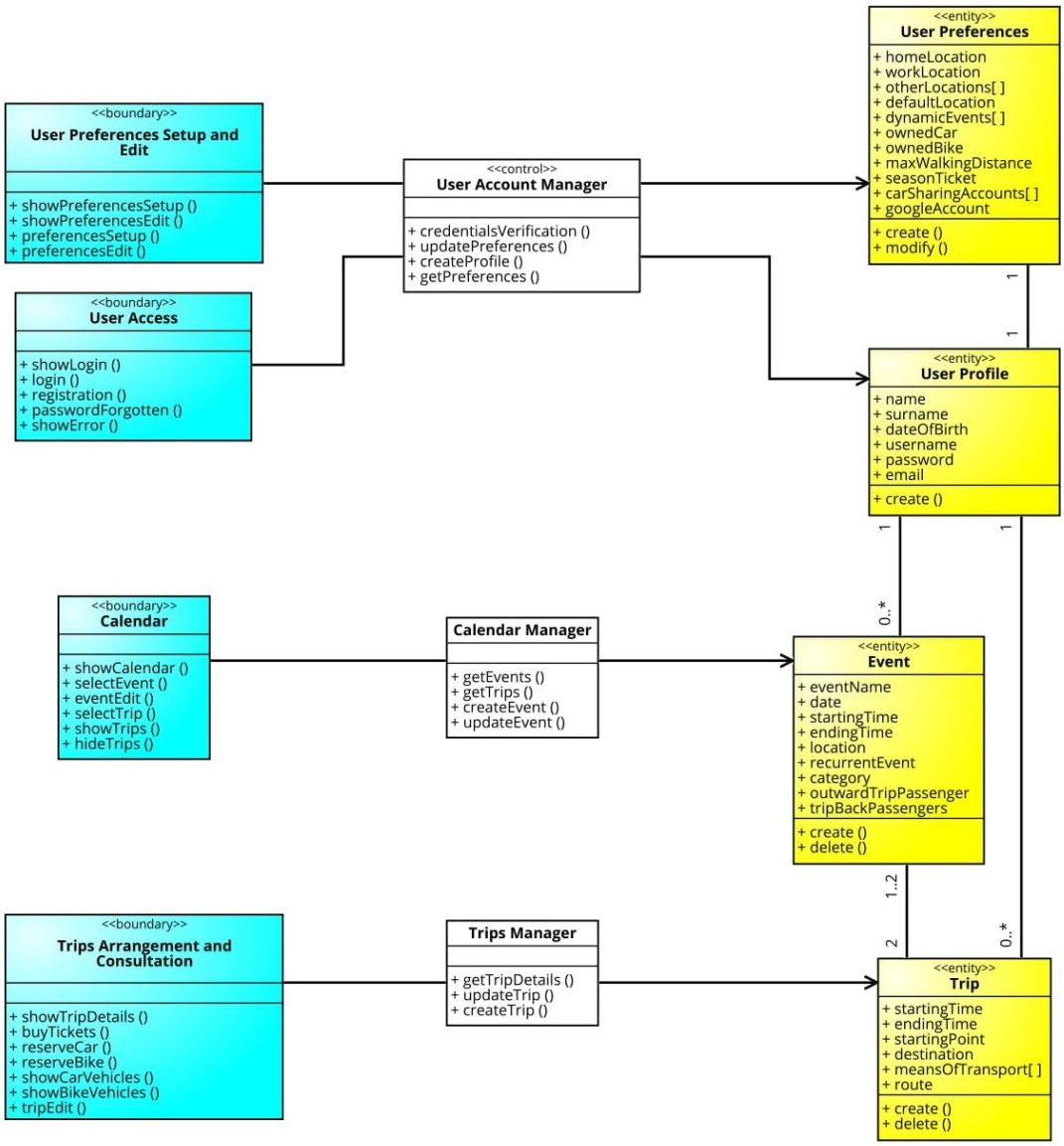

Figure 4.11: BCE diagram of the application

Chapter 5

Requirements Traceability

Providing traceability of system requirements to design components is important for determining if and how system requirements have been realised. The RTM (Requirements Traceability Matrix) also is needed to understand if all design components are necessary and to quickly understand the impact of changing a requirement. Non Functional requirements are detailed more in general in the other parts of the document. The matrix displays Functional requirements on the rows (check RASD for more deatils on section 3.A) and the components needed to satisfy these Functional Requirements on the columns.

	User controller	User information	User preferences	Event controller	Event	Trip controller	Trip	Notification	Maps	Strikes	Weather
[1,..,5]	X	X									
[6,..,13]	X		X								
[14,..,18]				X	X						
[19,..,23]						X	X				
[24]						X	X		X		
[25,..,27]						X	X	X			
[28,29]				X	X			X			
[30,32]						X	X		X		
[31,33,38,39,40]						X	X				
[34]						X	X				X
[35]						X	X			X	
[36]				X	X	X	X				
[37]	X		X			X	X				

Figure 5.1: Requirements Traceability Matrix

Chapter 6

Implementation, Integration and Test Plan

6.A Implementation Order

The implementation order of the tiers (or better layers) of the system is chosen in order to ease and speed up the integration of the whole system. The first step is to have the database tier and then we can develop the business tier because it can be tested without any client but only by making API calls and it is requested to use the mobile application. The next thing to be developed is the mobile application or the web tier, but the decision was to develop the mobile application so that a client-server system is obtained. Then in order the web tier and the web browser will be implemented.

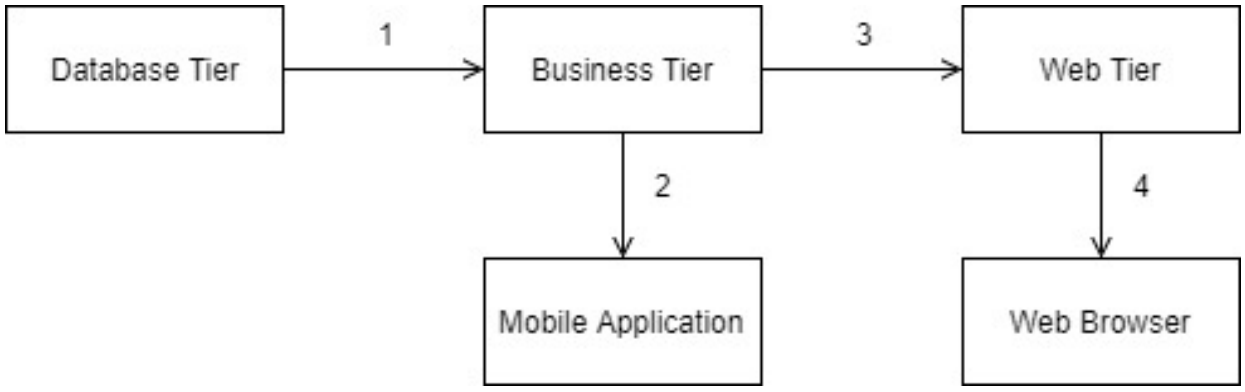

Figure 6.1: Implementation order of the system

6.B Integration and Test Plan

This section describes the preparation of the integration testing activity for all the components, to reach the completed system.

6.B.1 Entry criteria

In order to enter in the integration testing phase some condition have to be satisfied, first thing RASD and DD of the system should be delivered. Furthermore, the development percentage of the components involved in the activity of integration has to be over a certain value (80% for the components of this system) and the relative test units should be performed.

6.B.2 Elements to Be Integrated

Referring to the section 2 - Architectural design, the system can be divided in three categories of components:

- Front-end components: mobile application;
- Back-end components: the server and its components;
- External components: the components which refer to functionalities provided by external systems and the DBMS.

The front-end and the external components are independent one from each other, referring to back-end components some of them are not independent from others and need to be integrated. To fully integrate the three categories of components, some partial integration between categories need to be performed: front-end components and back-end components, and back-end components with external components.

6.B.3 Integration Strategy

The approach used for the integration testing phase is a bottom-up strategy, starting from components independent from other ones or components that depends only on one already developed, we assume we already have the unit tests for the smallest components. Then subsystem formed by the components tested, in turn, will be tested. This method will allow that a single component can be tested as soon as it's finished (or almost completed), and so to optimize the development and testing process parallelizing the two phases. Moreover, the high-level system is well separated and loosely coupled

since they correspond to different tiers, so it will not be hard to integrate later.

Chapter 7

Effort Spent

Date	A. Aimi	R. Bigazzi	F. Collini
15/11/17		1,5h	1,5h
17/11/17		3h	4h
18/11/17	3h	3h	5h
19/10/17	2h		2h
20/10/17	2h	2h	2h
21/10/17	3h	1h	3h
22/10/17	6h	4h	
23/10/17	3h	2h	3h
24/10/17	4h	4h	4h
25/10/17	7h	7h	5h
26/10/17	6h	6h	6h
Tot	36h	33,5h	35,5h

Table 7.1: Hours of work spent by the authors of the document