POLITECNICO
MILANO 1863

Travlendar+ | Software Engineering II Project

Design Document

Authors

Antonio Frighetto

Leonardo Givoli

Hichame Yessou

Professor

Elisabetta Di Nitto

Academic year 2017/2018

Contents

Chapter 1

Introduction

1.1 Purpose

The aim of this document, namely *Design Document* (DD), is to present the underlying architecture of the application Travlendar+, including the user interface and the interactions among various components and the algorithms used by specifying in details the requirements previously show in RASD. It is primarily addressed to project managers, developers, and testers so that they can be shown how the application is expected to be built.

1.2 Scope

The systems aims at offering an advanced calendar where the user can inserts as many appointment as he/she wishes, confident the system is responsible for arranging such appointments and supporting the user in his/her travels. The architecture is thought to be flexible and maintainable - in case current features are extended or new ones are added. The foundations make higher components to abstract from them, as well as principles of encapsulation, separation of concerns and strong cohesion are all applied. This has several impacts on how much existing models can be reused, how much code has to be changed and how much the behavior of a component logic is likely to be reviewed if a distinct component is subject to changes.

1.3 Definitions, Acronyms, Abbreviations

RASD: Requirements Analysis and Specification Document (previous document).

DD: Design Document (this document).

DB: Database layer.

NRDBMS: Non Relational Database Management System.

ACID: Atomicity, Consistency, Integrity and Durability.

REST: Representational State Transfer, an architectural style adopted by web services.

Back-end: Server-side application.

SoC: Separation of Concerns, a design principle.

MVVM: Model-View-ViewModel, an architectural pattern.

API: Application Programming Interface.

UI: User Interface.

UX: User eXperience.

1.4 Revision history

Version	*Date*	*Authors*	*Note*
1.1	26/11/17	Antonio Frighetto, Leonardo Givoli, Hichame Yessou	Fix & Review
1.0	22/11/17	Antonio Frighetto, Leonardo Givoli, Hichame Yessou	Initial version

1.5 Reference Documents

This document follows the RASD[1] one, and is based on the specifications document of the DD assignment[2].

1.6 Document Structure

This document is structured as follows:

Chapter 1: Introduction. This section introduces the design document by providing general information about it.

Chapter 2: Architectural Design. This section is divided into several parts and shows the main components of the systems together with their relationships, how they have been thought and designed. It shows also in detail choices made from an architectural point of view, patterns and paradigms.

Chapter 3: Algorithm Design. This section describes the most critical parts expressed through algorithms.

Chapter 4: User Interface Design. This section shows how the user interface is going to look like and behave through mockups and UX modeling diagrams.

Chapter 5: Requirements Traceability. This section aims at explaining how the decisions taken in the RASD are linked with the design elements shown here in the DD.

Chapter 6: Implementation, Integration and Test Plan. This section aims at identifying the order in which the implementation of the subcomponents is planned and the order in which the integration of subcomponents is prepared.

A **Bibliography** with various references is given at the very end, with an **Appendix** with information about the number of hours per each group member.

Chapter 2

Architectural Design

2.1 Overview

This section aims at giving an overview of the architectural components of the application, starting with a high-level approach towards a more detailed and deeper level of the technologies. We will discuss how and why we have chosen the technologies that will be used in the development of the clients and what kind of back-end solution.
Travlendar+ relies on *Amazon Web Services* (AWS), a collection of cloud computing services handled by Amazon.com which offers, among other things, servers, storage and database. It hosts our application and lays the foundation for the client-server communication. Besides benefiting of high reliability and scalability, AWS's resources enable us to have a degree of flexibility that cannot be achieved through hosting with normal services and implementation of our own Web API in the traditional way. Accessing the AWS services is really feasible thanks to the AWS SDK's available that helps to take out of coding the complexity of accessing them providing wrapped APIs in every major technology that could be used to implement the clients. Furthermore, the mobile client-side will be implemented with Xamarin.Forms, a cross-platform UI toolkit abstraction that allows us to develop a great multi-platform solution (iOS and Android) in C# with an excellent code-sharing approach. The web application will be based on Angular.js and HTML5. Angular.js is one of the most popular JavaScript frameworks for web applications, it allows to have a rapid development and its maintained by Google Engineers meaning that you will find reliable support if needed. It perfectly integrates with HTML5 and its fully compatibly with the MVVM design approach. Further details are provided in the following sections.

2.2 Component view

From a high-level point of view, the following components of the system can be identified:

Client: Implementation of the application that makes requests to the server and will be provided with a response. In Travlendar+ the clients will be the mobile application and the web application, that interacts with the server through REST APIs.

Web Server: Software running on a server able to communicate with web application clients using the HTTP(S) protocol, known also as back-end.

Application Server: This layer makes the workflow and the business logic for the client applications work. It will be implemented with AWS technologies interacting with a different component for each type of client.

Web Browser: Application for retrieving and presenting web pages and applications.

Database: This layer handles retrieval and storage of data. It will be queried by the application server. This layer must guarantee ACID properties.

We list also the following components which may not strictly belong to the component view but play a key role in our design approach:

AWS: Cloud services that offer computing power, database storage, content delivery and other scalable functionalities.

AWS Cognito: AWS component and client library that enables cross-device syncing of application-related user data.

Xamarin: Framework that delivers native Android, iOS and Windows applications using a shared C# codebase and same APIs.

These high-level components are laid down as follows.

Server-side application

Since relying on AWS services, one of the main core service used is *Amazon Cognito Sync*, a service which allows us to sync the user's data across multiple clients and platforms. Its structure is illustrated in Figure 2.1. The login is managed by *Amazon Cognito Identity* service and supports either the social platforms authentications (Facebook and Google) or the traditional email registration flow. This can be achieved by associating an identity with a dataset containing key-values pairs having a maximum size of 1 MB. Due to AWS limitations, each identity can be associated with maximum 20 datasets. Amazon Cognito Sync functions by creating a local cache for the identity data. An interaction between the application and this local cache occurs when reading or writing the keys. This allows Travlendar+ to synchronize all the changes made on a device immediately to other devices. Not only the user preferences are going to be stored but user events and means of transports too, this could be enhanced in future to other services using Amazon Cognito towards an AWS DynamoDB or and AWS Lambda if we would like to transfer some computational power towards the backend side. What's more, Amazon Cognito offers a scalable and modular way to react upon peaks in demands and traffic.

Web Server

The mobile application will interface with the application server through the wrapped RESTful APIs while the web application will run on a web server based on an Amazon Elastic Cloud Computing instance. Amazon Cognito Sync will play a key role providing a real-time synchronization of the user data and event across the clients. It provides a service of identity management within an identity pool, giving secure access to AWS services from all the platforms and synchronizing user's data. The data is persistent to the local storage first, making it available regardless the connectivity. The web application will have a server to filter the request from the clients. It's based mainly on Amazon Web Application Firewall (AWS WAF). AWS WAF will help to lower the load on the infrastructure (Layer 3 and Layer 4) from DDoS attack or other techniques and giving control over which traffic to allow or block. It includes full-featured API that can be used to create and/or deploy web security rules. The whole infrastructure is shown in Figure 2.2. The runtime environment will be operating on Node.js that simplifies the development of the web application thanks to rich and various JS modules. Besides that, we will use nginx on our Amazon EC2 instances to proxy the network traffic to the running Node.js instances. It supports sophisticated Layer 7 load balancing as well as

reverse proxy capabilities allowing a multiple web server infrastructure making the application more reliable.

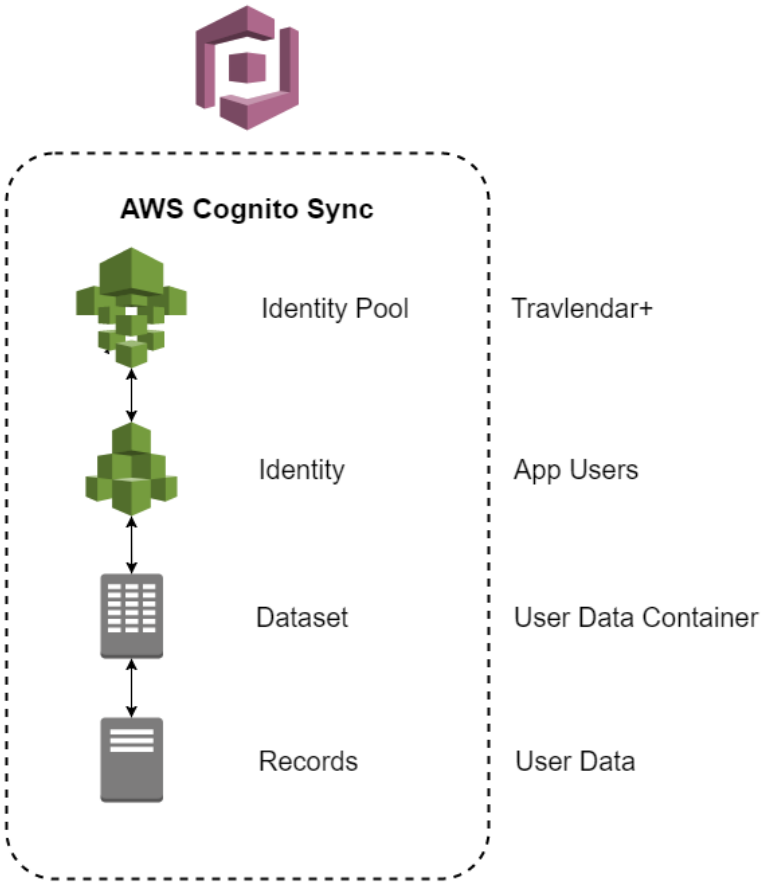

Figure 2.1: Structure of the Amazon Cognito Sync service.

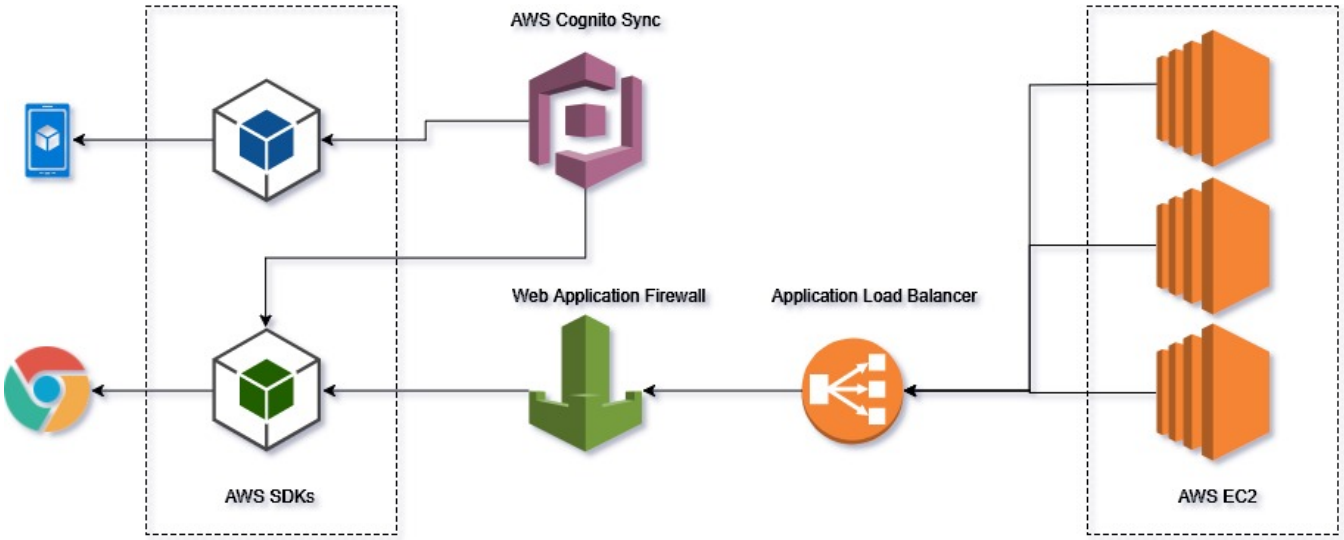

Figure 2.2: AWS SDK makes use of services exposed by the AWS infrastructure and act as mediator for the client application.

Mobile client application

Travlendar+ mobile application is thought to be implemented with Xamarin technologies. In addition to being an extremely powerful set of frameworks, it allows us to have a multi-platform presence and speed a little bit up the development phase while maintaining a high degree of modularity. As a result, since these technologies make use of C#, .NET SDKs is used. Particularly, Xamarin.Forms framework is used, which is, as said above, a cross-platform UI toolkit that allows us to create a native interface. While it is widely used and gained considerable fame since the very first release, Xamarin.Forms is still quite a new technology so whenever we need to build particular details whose APIs that abstract the platform may not exist, we plan to bind the native Android and iOS components by ourselves. So, we can choose to use the main control groups to create the User Interface, which are Pages, Layouts, Views and Cells that are mapped and rendered to the native equivalent and, indeed, Xamarin.iOS and Xamarin.Android are taken into account when customizing a platform-specific UI component. Xamarin.Forms applications are drawn up in the same way as any other cross-platform application. There are two viable approaches, the most common is with Portable Class Libraries (PCL) while the other is via Shared Projects. Our choice is more directed towards the PCL that may result in a DLL usable on the specific platforms for which we provide support. Figure 2.3 shows the architecture of a cross-platform application using a PCL to share code. This approach has a number of advantages like a centralized code base consumed by the platform projects, refactoring operation affects all the code within the solution and the relief of referencing itself against the platform projects. The drawback may concern platform specific libraries, which cannot be referenced inside the PCL, but this can be avoided with the Dependency Injection pattern, using an interface in the Portable Class passing platform-specific features.

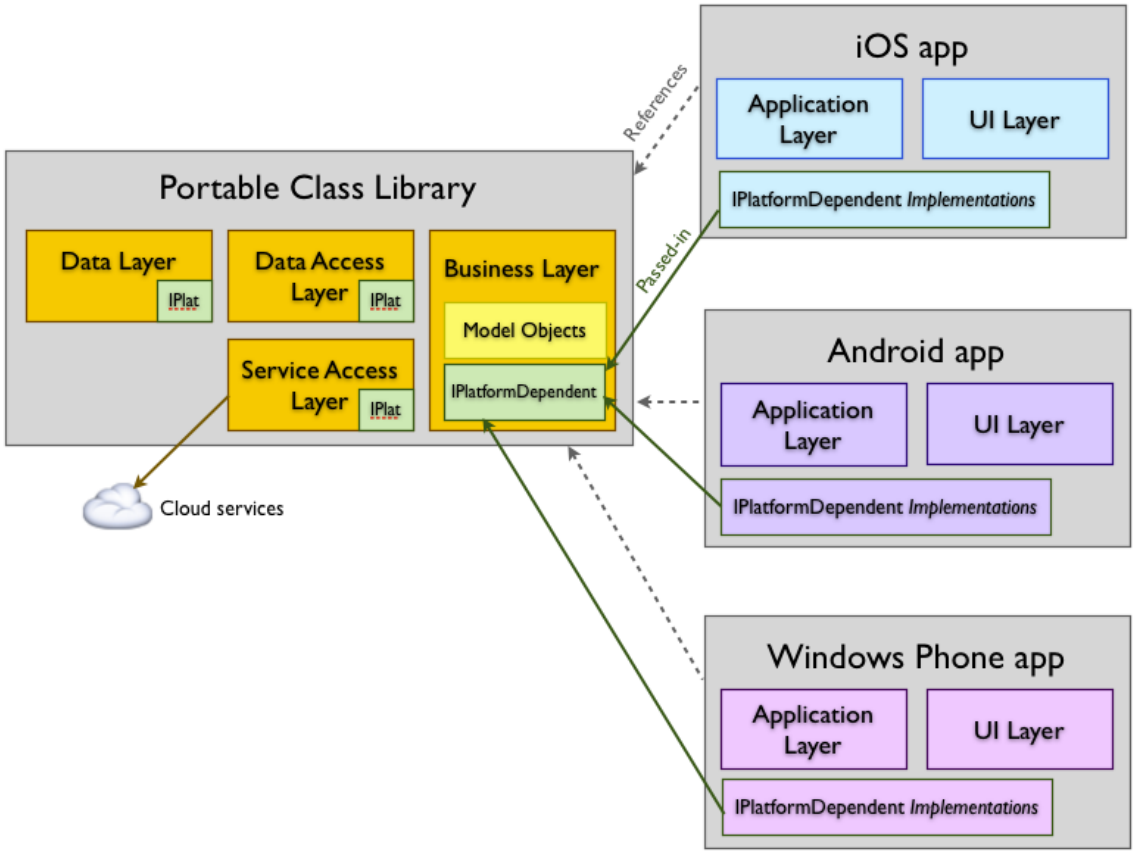

Figure 2.3: PCL Diagram[3].

Web application

As regards the web application, in order for it to be dynamic and highly responsive, the implementation consists of using the fifth standard of HTML enhanced by a structural framework, AngularJS, which indeed lets us compose HTML templates. The functionalities of the web application are almost the same of the mobile one, thus allowing real-time synchronization among the devices. The application is meant to be composed of component classes that manage the HTML templates and the business logic is implemented in services, boxing them in modules. We continue relying on the AWS SDK as for the mobile client application to implement the synchronization of the user data within the web one. AWS Cognito still plays a major role. This introduces an extra layer of abstraction from the back-end service, keeping the more sensitive pieces out of the exposed JavaScript. The web application is hosted on the AWS infrastructure having resources that will dynamically grow or shrink having a cost-effective solution. Amazon EC2 provides a re-sizable computing capacity, allowing in the most simple way the web-scale of the cloud computing.

Specific components

In detail, these are the main specific components:

TravlendarAppGUI: It provides the user interface to the mobile clients, allowing the usage of the services.

TravlendarWebGUI: It provides the user interface to the web-based clients, allowing the usage of the services.

AuthenticationService: It is responsible for the identification of the users. It provides the possibility to access the application for existing users or the registration for new users.

SyncUserInfoService: It is responsible for the retrieval and synchronization of the user's data across multiple platforms.

EventService: It provides the main functionalities of the application. It allows to create, modify and delete events.

MapService: It provides the map functionalities and computes the optimal travel route through the best combination of mobility options possible.

PurchaseService: It handles the ticket's purchase for the available travel mean. When possible it deep-links the user to the specific application, otherwise it may be redirected to the web-page.

In Figure 2.4 is shown the whole class diagram for the system. Compared with the one presented in the RASD, we better specified properties and methods.

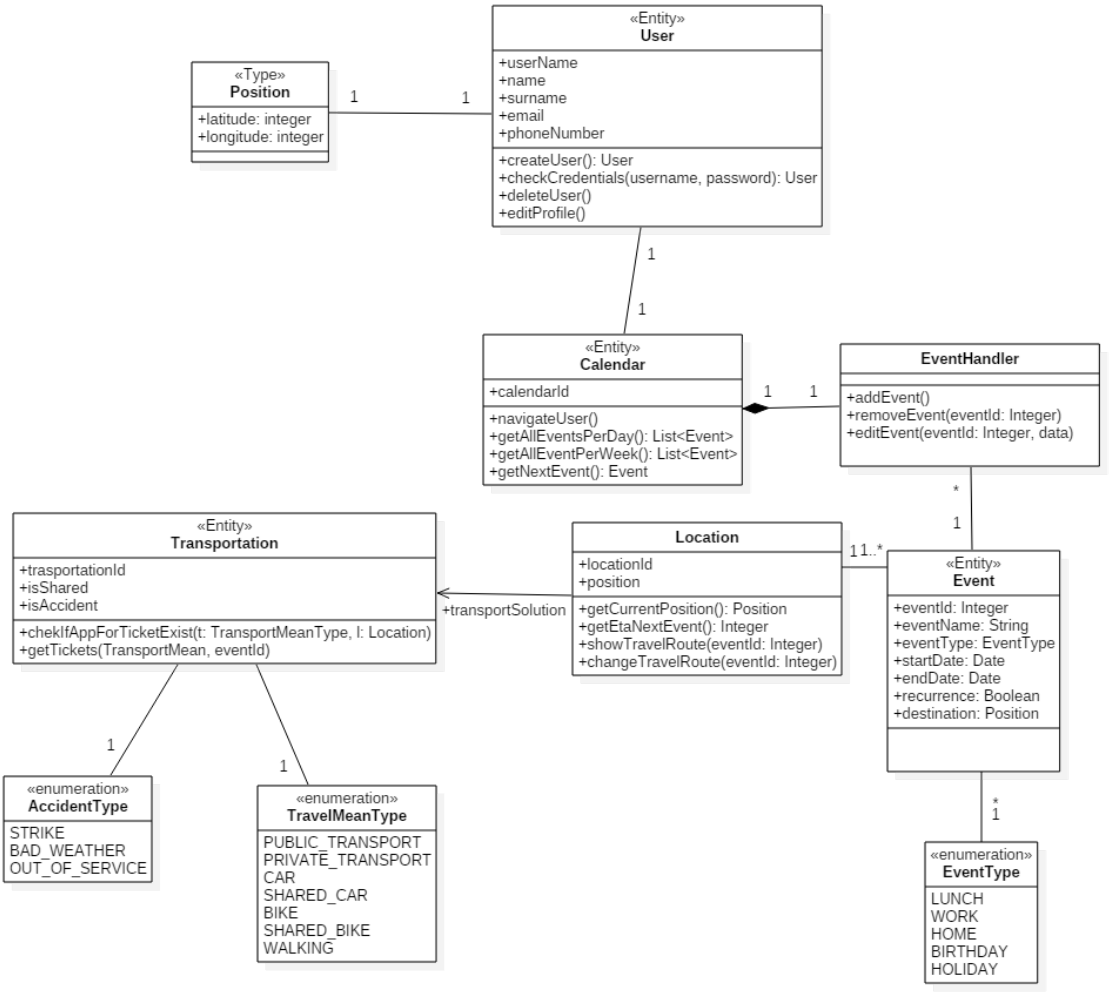

Figure 2.4: Class Diagram

2.3 Deployment view

The deployment diagram for the system is shown in Figure 2.5.

2.4 Runtime view

Here we describe the dynamic behavior of the system. In particular, how the software and the aforementioned various logical components interact with each other. Such interaction is shown through the following sequence diagrams (Figures 2.6, 2.7, 2.8, 2.9).

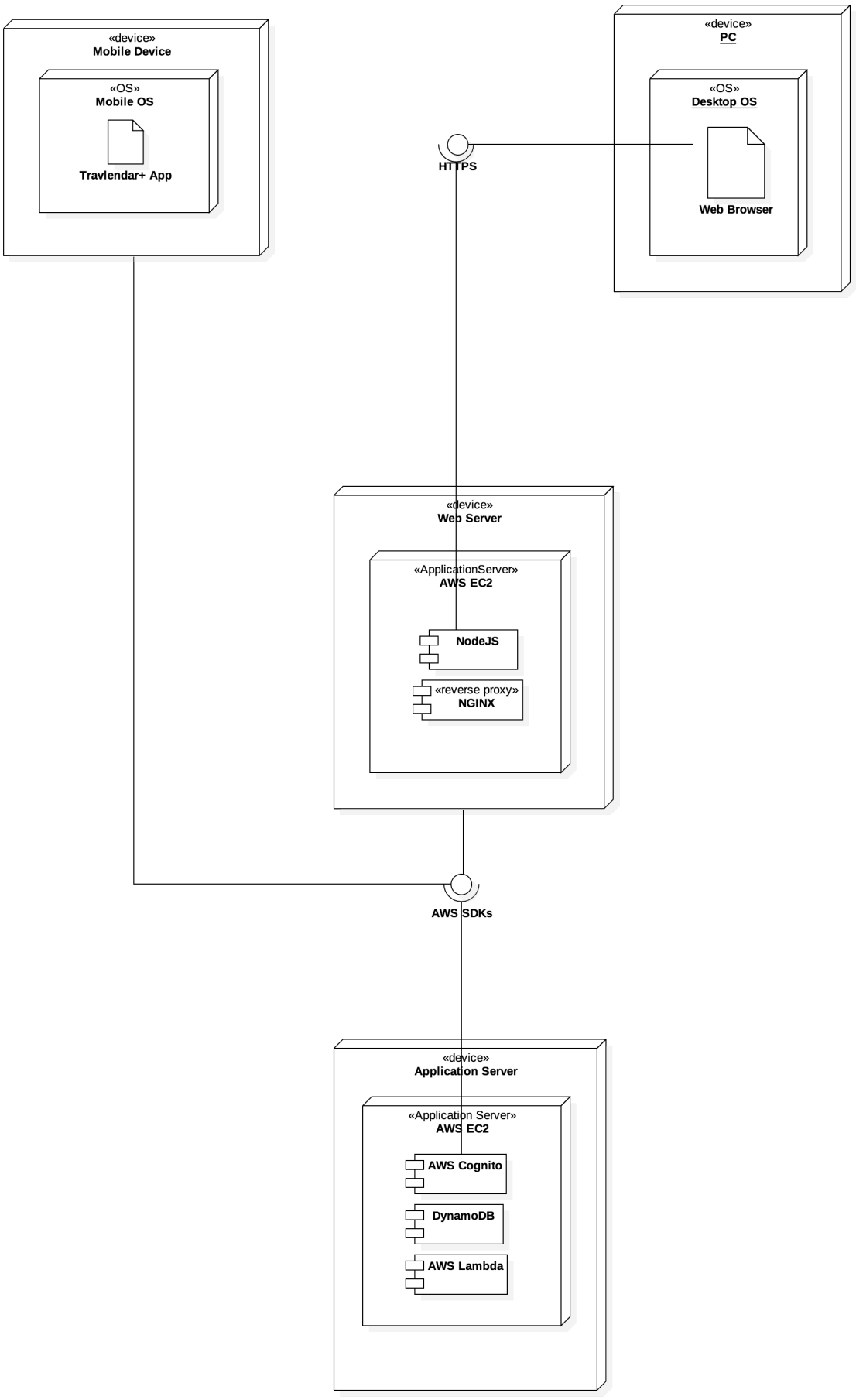

Figure 2.5: Deployment Diagram

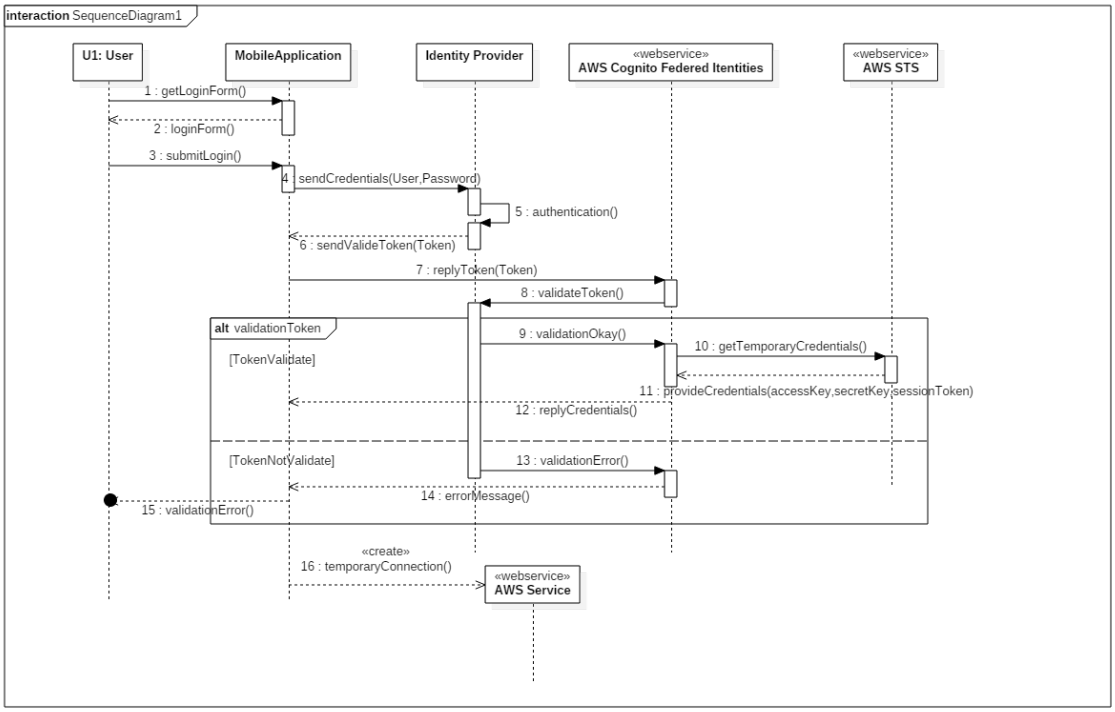

Figure 2.6: Sequence diagram of the login with the AWS Web Service.

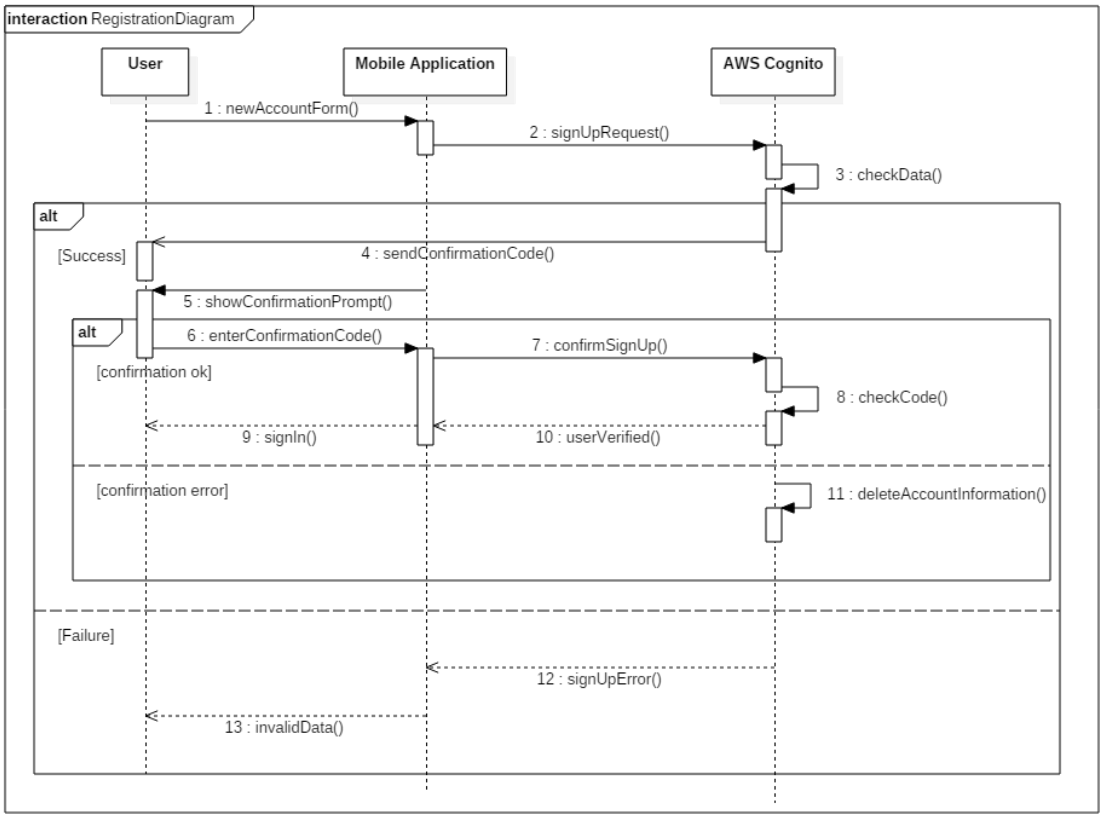

Figure 2.7: Sequence diagram of the mobile registration with the AWS Service.

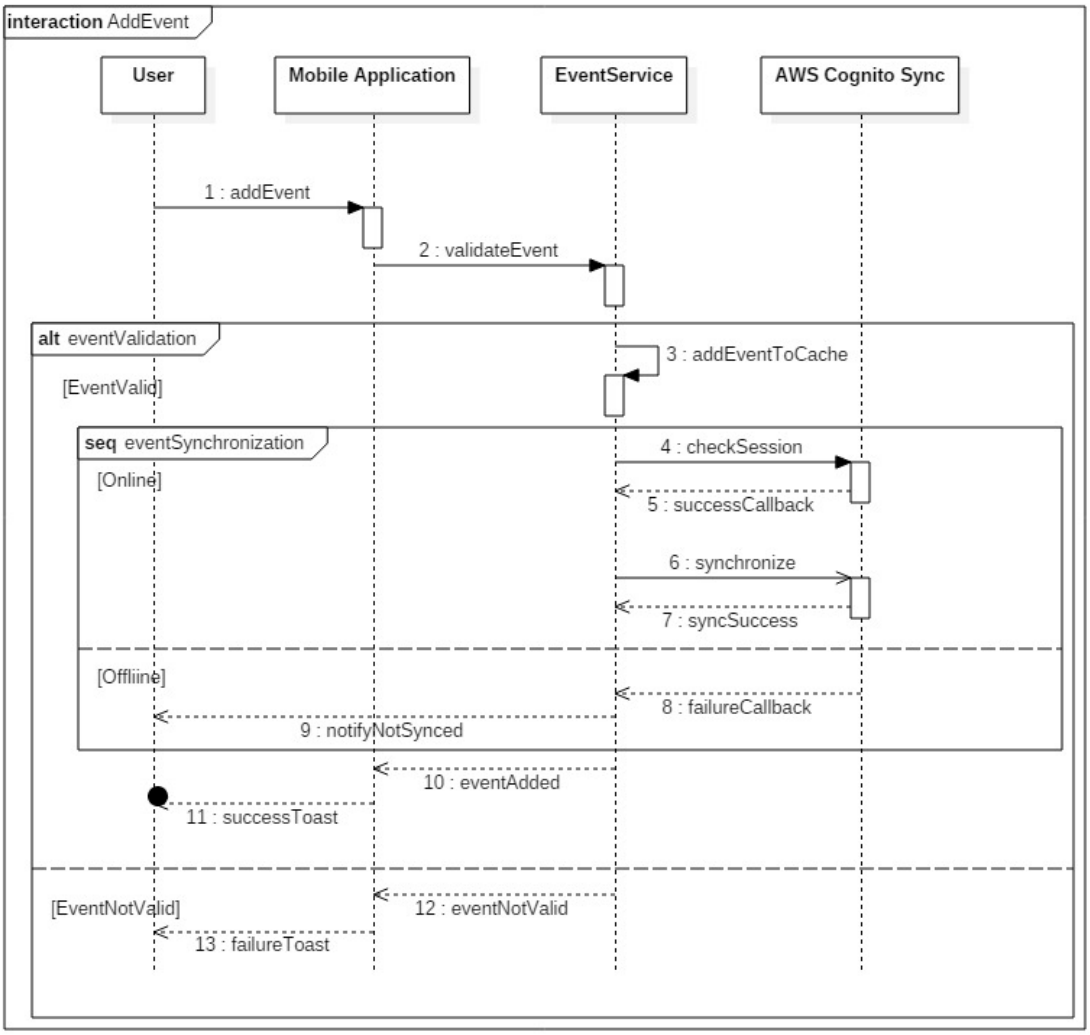

Figure 2.8: Sequence diagram for adding an event.

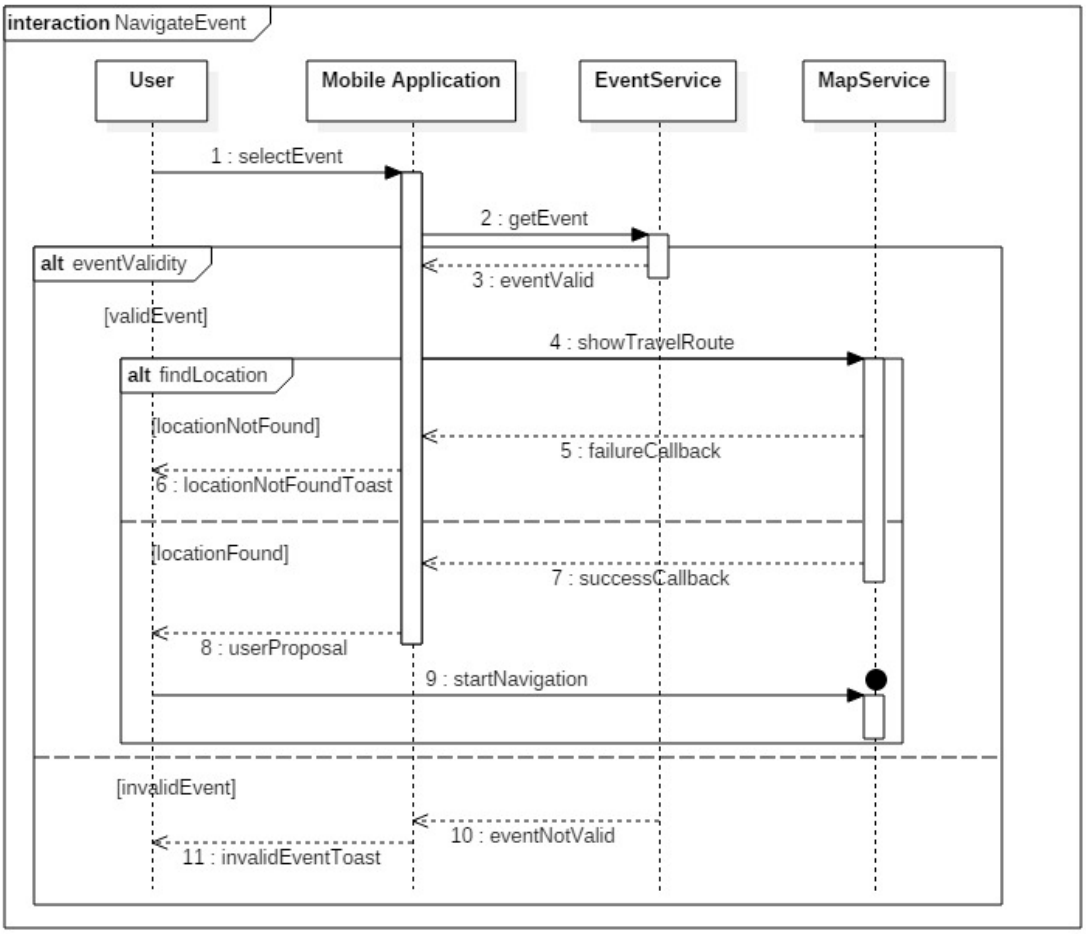

Figure 2.9: Sequence diagram for navigating to an event.

2.5 Component interfaces

The following section describes the interfaces by means of which the components communicate with each other. Note that the front-end of the system (both of the mobile application and the web one) must communicate with the application server through RESTful APIs over HTTPS. The RESTful interface is wrapped by AWS SDK in the application server.

AuthenticationService

Methods implemented by the AuthenticationService interface:

`registerUser(info)`: This method creates a new user with AWS Cognito.

`loginUser(data)`: This method authenticates an existing user using the AWS Cognito platform.

`checkSession(token)`: This method validates the current session with AWS Cognito.

UserInformationService

Methods implemented by the UserInformationService interface:

`synchronizeData(token)`: This method synchronizes the local cache of the AWS Cognito SDK with the AWS Cognito service.

EventService

Methods implemented by the EventService interface:

`addEvent(data)`: This method adds an event to the calendar and synchronizes it with AWS Cognito.

`editEvent(eventId, data)`: This method edits a specific event in the calendar and synchronizes it with AWS Cognito.

`removeEvent(eventId)`: This method removes a specific event in the calendar and synchronizes it with AWS Cognito.

`getEvent(eventId)`: This method returns the given event

`getNextEvent()`: This method returns the user's next event.

`getAllEventsPerDay()`: This method returns all the events scheduled by the user per day.

`getAllEventsPerWeek()`: This method returns all the events scheduled by the user per week.

MapService

Methods implemented by the MapService interface:

`getCurrentPosition()`: This method return current user position.

`getETANextEvent()`: This method returns the travel time for the user's next event.

`showTravelRoute(eventId)`: This method shows on the map the route for a specific event.

`changeTravelRoute(eventId)`: This method attempts to select a different route if the previous one did not satisfy the user.

PurchaseService

Methods implemented by the PurchaseService interface:

`checkIfAppForTicketExist(idTransportMean, location)`: This method checks if there's a third-party application which handles the purchase ticket for a specific transport mean, and if so redirects the user to the application of the dedicated transport mean to buy tickets.

`getTickets(idTransportMean, eventId)`: This method retrieves through callbacks the ticket bought by the user on the transport mean application.

2.6 Selected architectural styles and patterns

Here the main architectural styles and design patterns are identified with the related reason.

Network architecture

Client-server model

From a network point of view, a well-known structure model is doubtless the client-server one, which is for sure extensively used across networks. Depending on the service provided, such model starts taking place at the following levels:

- Mobile application (client) talks to APIs located on the server (when querying for specific data or handling records on the database).
- Web application front-end (client) communicates with the server application.
- Web server serves pages when user's browser makes a request.

Client-server model is shown in Figure 2.10.

Client architecture

MVVM Architectural Pattern

The Client architecture is based on the pattern Model-View-ViewModel that facilitates the separation of the GUI from the business logic. Being a cross-platform application this will enable a high percentage of shared code across the implementations, speeding the development of the application itself and future features. The primary purpose of MVVM is that the code is broken up into classes with a small number of well-defined responsibilities, serving an easier to understand, maintain and updatable code. Secondly, the MVVM pattern can bring flexibility to the development of the project allowing developers and designer to work simultaneously. It increases as well the application testability, fragmenting and separating the UI logic into different classes from the business logic. Its a natural pattern for XAML implementations with key enablers like the rich data binding stack and dependency properties. This sequence contributes to the connection between the UI/View to the ViewModel.

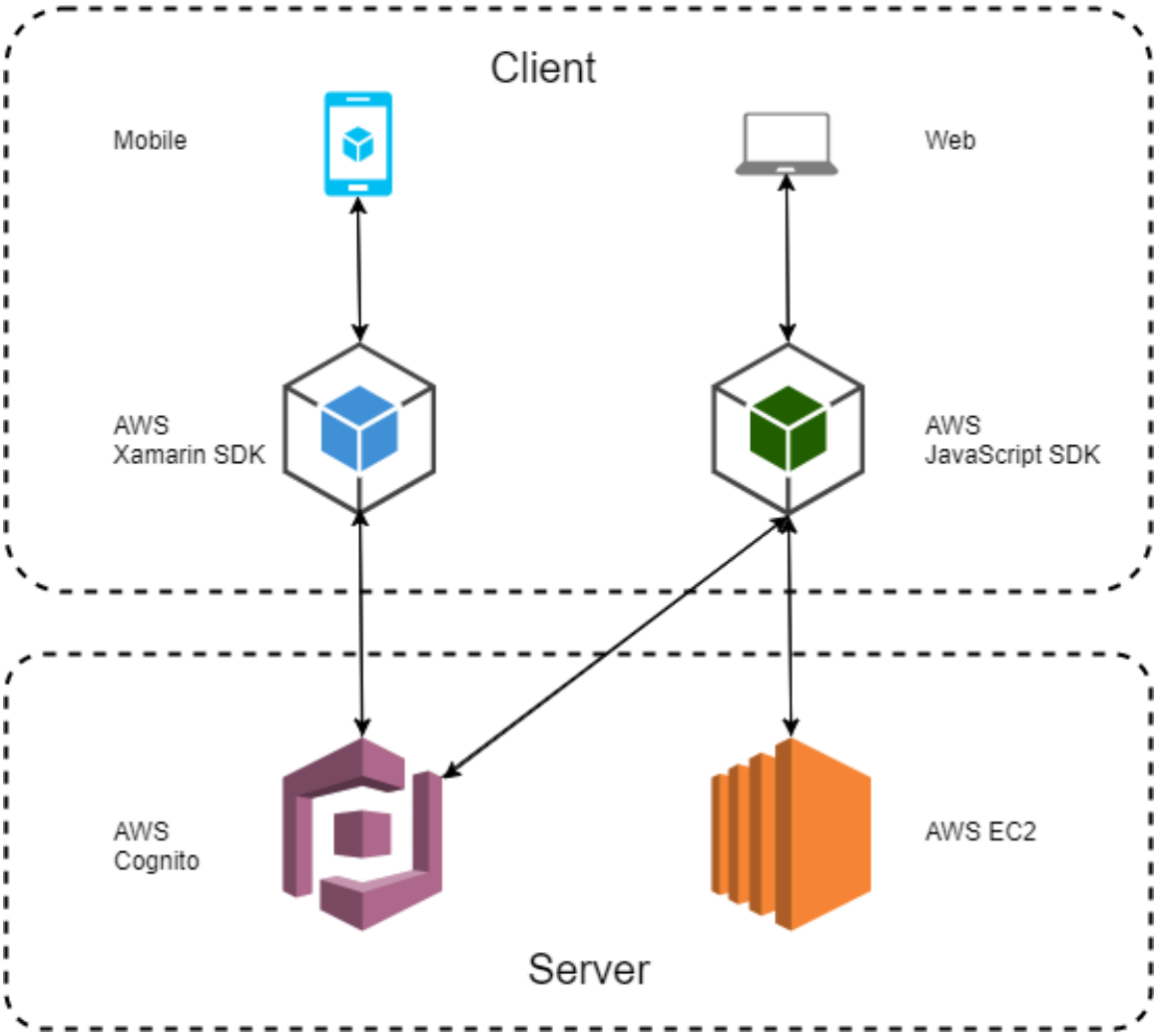

Figure 2.10: Client-server model in our system.

Design patterns

Inversion of Control and Dependency Injection

The second major aspect that will characterize the client implementation is the Inversion of Control design paradigm. It's a broader concept that aims to give more control to the specific components of the application rather than creating "manually" every object. Inversion of Control will imply also the Dependency Injection pattern meaning that the approach used to create instances of objects that other objects rely upon without knowing at compile time which classes will provide the functionality itself. Dependency Injection makes the testing easier, allowing the switch of real service implementation with stubs almost painless. These aspects are compliant with the GRASP guidelines, specifically in the High Cohesion evaluative pattern. It attempts to keep objects properly directed on their goal developing the whole system into a low coupling system with several classes and subsystems.

Separation of Concerns

Strictly related to the already mentioned patterns, we include also a principle of fundamental importance in order to achieve separation of our application into several distinct features that overlap in functionality with minimum "interference" possible. MVVM pattern perfectly allows us to achieve such principle, known as separation of concerns. By means of information hiding (which is realized with encapsulation), every module will have its own interface to communicate with each other and hide the internal business logic. More specifically, in the context of MVVM, Views do not know (almost) nothing about each other, especially what is going on in the ViewModel, which contains, as said, the workflow of the application and connects the UI with the business logic itself through data binding. Just to mention some benefits we can obtain are surely maintainability and reusability.

2.7 Other design decisions

The user's credentials are not stored in the device, rather the AWS SDK provides a way to save the tokens used to access the AWS Services. Those tokens will be hashed and stored in an AES encrypted SQLite database. The key to access it is generated upon the installation of the application, providing a unique random key stored in the respective Keystore/Keychain platform implementations. Travlendar+ will rely on Google Maps for everything that concerns the route calculation and map implementations on the clients. Being a leader in the field, it provides a wide support to the APIs integrations as well as for the SDK to be implemented on the clients. It helps to delegate much of the effort to a well-established component in which the users can feel comfortable to use.

Chapter 3

Algorithm Design

Here we focus on the definition of the most relevant algorithmic part of the system. The first one, and perhaps the most important one must be able to provide to the user the best travel mean (or, at most, a combination of them) as quickly as possible. It is supposed to consider the user preferences but it needs to compute its own calculation as well, and in some cases even some predictions. To explain the logic of the algorithm we thought we'd better show it through pseudocode. While it may resembles a specific object-oriented language a lot, code shown is still pseudocode, it just includes some common feature of high-level languages (e.g. object initialization) and instead of returning an artificial type we directly thought what suitable type could have been.

Event validation algorithm

```
/* This function will check that there are no events overlapping with
 ↪  each other and ensures that special types of event are always taken
 ↪  into account. It returns true if the event can be added correctly,
 ↪  false otherwise. */
bool validateEvent(Event e) {
  /* Check if event is a type of event that does not last the entire day.
   ↪  */
  if (!e.allDay) {
    /* Check if there is another event that has already been inserted at
     ↪  that specific time slot. eventOverlaps(Event) just takes the
     ↪  current event as argument and checks if there's another event
     ↪  whose time interval (start time and end time of same day)
     ↪  coincides with the current one. If it returns true, we return
     ↪  false and event cannot be validated properly. */
    if (EventOverlaps(e)) {
      return false;
    }
```

```
    /* If an event is not of a special type (like lunch or a custom event
     ↪ added by the user) and there are no more free time slots to
     ↪ insert a normal event, the event is refused. Particularly,
     ↪ NoLeftTimeForEvents(Event) receives the event as argument and
     ↪ checks if the event starts and finish within a range which is
     ↪ reserved for special events. For example, it checks if the event
     ↪ starts at 11:30 (or later) and finishes at 14:30 (or earlier) and
     ↪ ensures that (at least, depends on user preferences) half an hour
     ↪ can still be used for lunch. If there's no more free space except
     ↪ for lunch, it returns true. */
    if (e.typeEvent != SPECIAL_TYPE && NoLeftTimeForEvents(e)) {
      return false;
    }
    /* If event is adjacent to another one (adjacentEvents(Event)), which
     ↪ means, just as the current event finishes a new event occurs in
     ↪ the very next minutes, and destination of the latter one is far
     ↪ away from the current position (farAwayDestination(Event)), then
     ↪ we make the user aware of it by displaying an alert. We still
     ↪ return true though. */
    if (adjacentEvents(e) && farAwayDestination(e)) {
      displayAlert("Event's destination will be hardly reached on
       ↪ time.");
    }
  }

  /* In the other cases we can safely return true, in fact we allow that
   ↪ there are multiple all-day events occuring in the same day (maybe a
   ↪ recurring event, a birthday or a holiday) and no events are going
   ↪ to overlap with each other. */
  return true;
}
```

Travel mean selection algorithm

```
/* This function will return the best travel mean for the specific route
 ↪ of type Route passed as a parameter. Specifically, it returns a list
 ↪ of objects of type TravelMean, whose class, besides containing the
 ↪ travel mean, includes also the duration, start and end points
 ↪ (latitude and longitude) of each travel mean. In fact, if there are
 ↪ multiple travel means suggested, each of them must have a start and
 ↪ end location. Note that from a point of view of implementation
 ↪ choices, adopting a list may not be the best solution since very few
 ↪ means can be returned at most, it is just more to give the idea. */
List<TravelMean> selectBestTravelMeans(Route r) {
  /* Object travelMeans is initialized */
  List<TravelMean> travelMeans = new List<TravelMean>();
```

```
/* Initialitizing three flags, one to check public transport
 ↪  availability, the other one for bad weather conditions, and the
 ↪  latter to see if event type requires user to be in a hurry or, for
 ↪  instance, there are other passengers. */
bool availablePublicTransport = badWeather = otherPassengers = false;

/* We check that the destination is supposedly reachable, since there
 ↪  may be no feasible travel means to be returned. This happens
 ↪  especially if user's destination is considerably far away from his
 ↪  current position. Therefore IsLocationUnreachable(), after checking
 ↪  that longitude and latitude corresponds to a location that can be
 ↪  actually reached, returns true if the location is unreachable, and
 ↪  we return the execution to the caller. */
if (IsLocationUnreachable(r.destination)) {
  return null;
}

/* If settings have been recently changed or this is the first time
 ↪  that a travel route is computed, then we update userPreferences
 ↪  object (with global scope) by retrieving updated info from
 ↪  getUserPreferences() function. Both functions return a boolean. */
if (SettingsChanged() || IsFirstTimeTravelRoute()) {
  userPreferences = getUserPreferences();
}

/* We check at the beginning if there's availability for public
 ↪  transportation means. The following function checks in real-time if
 ↪  it's public holiday (and there may be time changes), if there's a
 ↪  strike or means are out of service (for instance, during the night
 ↪  there may be no available public means). It requires the current
 ↪  position to be passed so that checks are performed according to the
 ↪  city. */
if (IsPublicTransportationAvailable(r.origin)) {
  /* If the function returns true we set availablePublicTransport to
   ↪  true. */
  availablePublicTransport = true;
}

/* We check if it might rain right now or within few minutes always
 ↪  based on user's position. */
if (IsBadWeather(r.origin)) {
  badWeather = true;
}

/* We check if the user has specified when adding the event that there
 ↪  should be other passengers. */
if (OtherPassengers()) {
  otherPassengers = true;
}
```

```
/* NearbyDestination() instance method of Route class is called to
 ↪  check whether the distance between the current position of the user
 ↪  and the destination is greater than a certain threshold. Depending
 ↪  on how far the user is and if he/she is with other passengers or
 ↪  not, specific means are suggested. */
if (r.NearbyDestination() && !otherPassengers) {
  /* We check that the flag for the bad weather is false. */
  if (!badWeather) {
    /* If the user's option to ride a bike is enabled and has its own
     ↪  bike, we suggest using it... */
    if (userPreferences.IsBikeEnabled() &&
     ↪  userPreferences.HasItsOwnBike()) {
      /* We add own bike as mean, default origin and destination in
       ↪  this case (and in the others above) are used. */
      travelMeans.Add(OWN_BIKE);
    }
    /* ... otherwise we try locating the nearest bike sharing system
     ↪  via LocateBikeSharingSystem(). It takes the current position as
     ↪  parameter and returns true if a near shared-bike could be
     ↪  located otherwise false. If true, it adjusts the destination
     ↪  argument passed as reference by adding the path to be done to
     ↪  reach the bike. */
    else if (userPreferences.IsBikeEnabled() &&
     ↪  LocateTravelMean(BIKE_SHARING, r.origin, ref r.destination)) {
      /* We add the bike as possible mean to the list, we pass the
       ↪  arranged destination to the Add() method as well. */
      travelMeans.Add(r.origin, r.destination, BIKE_SHARING);
    }
    /* If no bike sharing could be located, then suggest walking. */
    else {
        travelMeans.Add(WALKING);
    }
  }
  /* Options in case it's rainy */
  else {
    /* If the flag availablePublicTransport is on and there is an
     ↪  underground train station nearby then we suggest it as possible
     ↪  mean. */
    if (availablePublicTransport && LocateTravelMean(UNDERGROUND_TRAIN,
     ↪  r.origin, ref r.destination));
        travelMeans.Add(r.origin, r.destination, UNDERGROUND_TRAIN);
    }
    else {
      /* We still suggest walking as possible mean, even if it's rainy
       ↪  since distance should be pretty short. */
      travelMeans.Add(WALKING);
    }
  }
```

```
  }
  /* If the user is travelling with its own car across the city, then it
   ↪  doesn't make sense to suggest public transports. */
  else if (userPreferences.HasItsOwnCar()) {
    travelMeans.Add(OWN_CAR);
  }
  /* All other choices. */
  else {
    /* If car sharing option is enabled and it can be located nearby, we
     ↪  suggest using it. */
    if (userPreferences.option == SHARED_CAR &&
     ↪  LocateTravelMean(CAR_SHARING, r.origin, ref r.destination)) {
      travelMeans.Add(CAR_SHARING);
    }
    /* If the user expressed the preference to always use a taxi, we
     ↪  suggest using it. */
    else if (userPreferences.option == TAXI) {
      travelMeans.Add(TAXI);
    }
    else {
      if (availablePublicTransport) {
        /* If there's availability for public transportation means, we
         ↪  pass execution to CalculateRouteWithPublicTransports(bool)
         ↪  function which has the task of calculating the itinerary
         ↪  using underground trains and buses. It may return a single or
         ↪  multiple travel means and takes a boolean as argument. If
         ↪  carbon footprint option is enabled it tries to retrieve the
         ↪  path with - if possible and feasible - a part to do on foot.
         ↪  */
        travelMeans.AddMultipleMeans(CalculateRoute
        WithPublicTransports(userPreferences.option == CARBON_FOOTPRINT ?
         ↪  true : false));
      }
      /* If no public transports are available our last solution should
       ↪  be suggesting a taxi. */
      else {
        travelMeans.Add(TAXI);
      }
    }
  }

  /* We return the transports found. */
  return travelMeans;
}
```

Pathfinding algorithm

Concerning the algorithm for the optimal path, as said, we will rely on a Google Maps component, so it will be up to Google Maps to find the shortest and the best path. Remind though

that it does nothing but applying a graph traversal algorithm. Looks like Google has picked up a slightly different version of the Dijkstra search algorithm[4]. After adding weights to the graph, it takes the shortest distance between two locations (treated as nodes) among the possible paths (treated as edges).

Chapter 4

User Interface Design

In this section, we show an overall view of the different user interfaces of Travlendar+ for each client. The UX diagram shows both the user flow and the different screens of the application (illustrated in Figure 4.1). In Figure 4.2 the User Interface of the mobile implementation integrating the user flow across Travlendar+ is shown. In Figures 4.3, 4.4 and 4.5 we show some mockups related to the web application. As can be seen, user interface of the web and mobile application are meant to be as similar as possible so that a continuous experience across the clients can be provided.
The general UX Diagram is composed of 2 parts:

- The **white** one describes the common functionalities of the web and mobile implementation.
- The **yellow** one describes the functionalities implemented only by the mobile client.

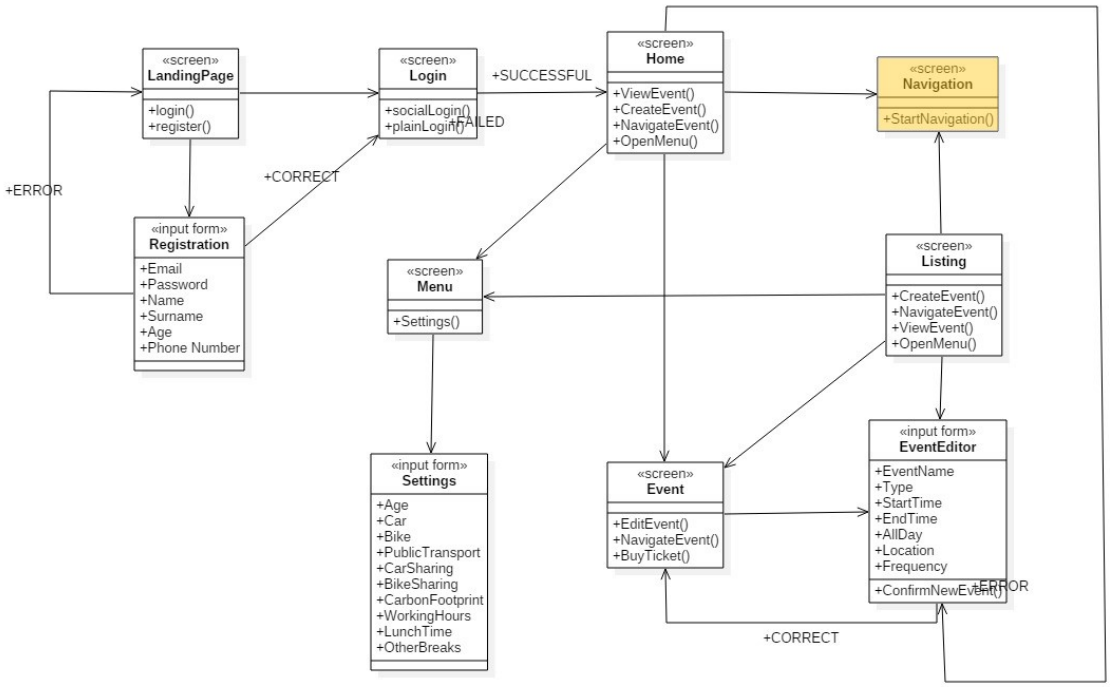

Figure 4.1: UX Diagram of the client implementation.

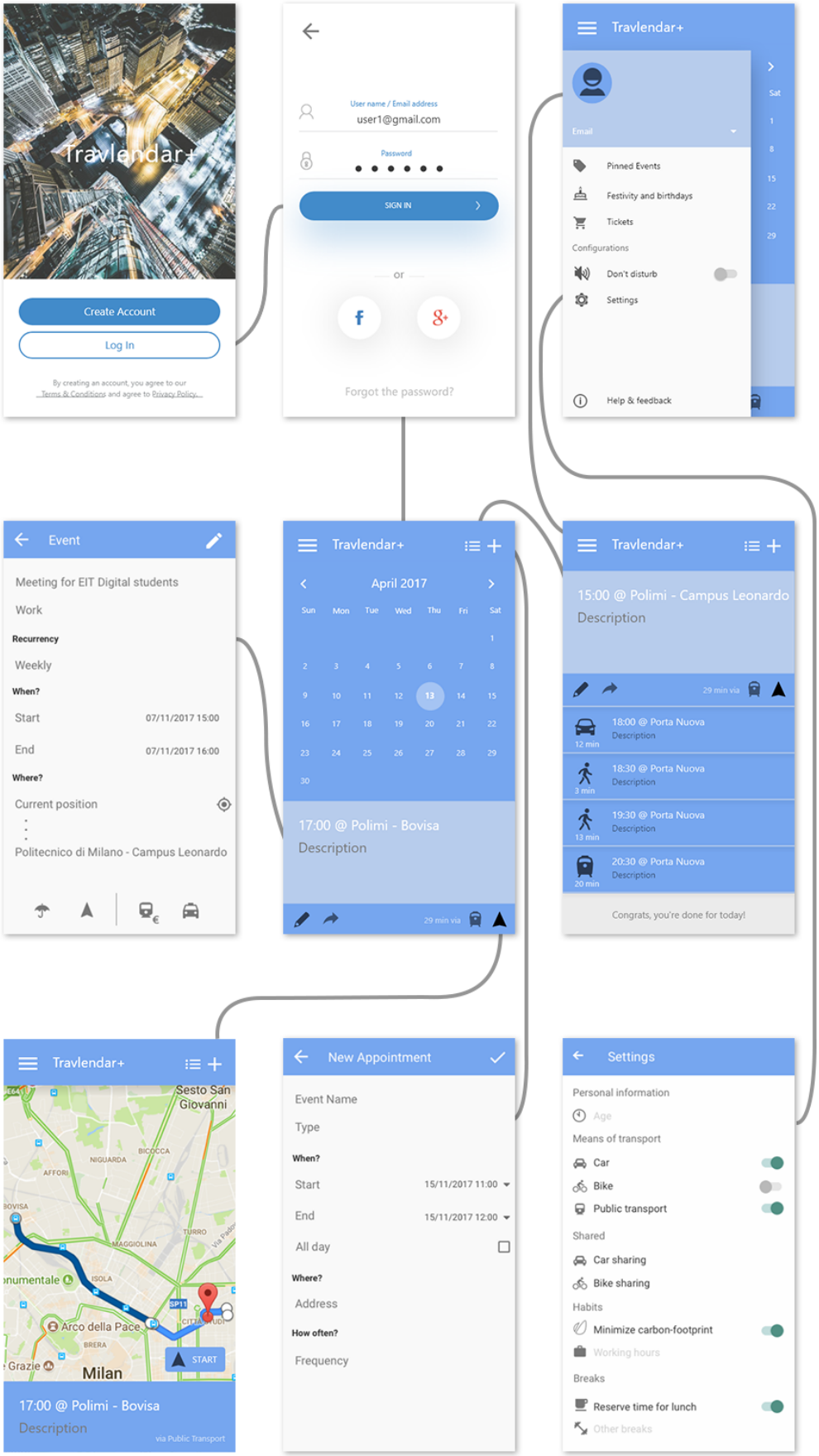

Figure 4.2: UI Diagram for the mobile implementation.

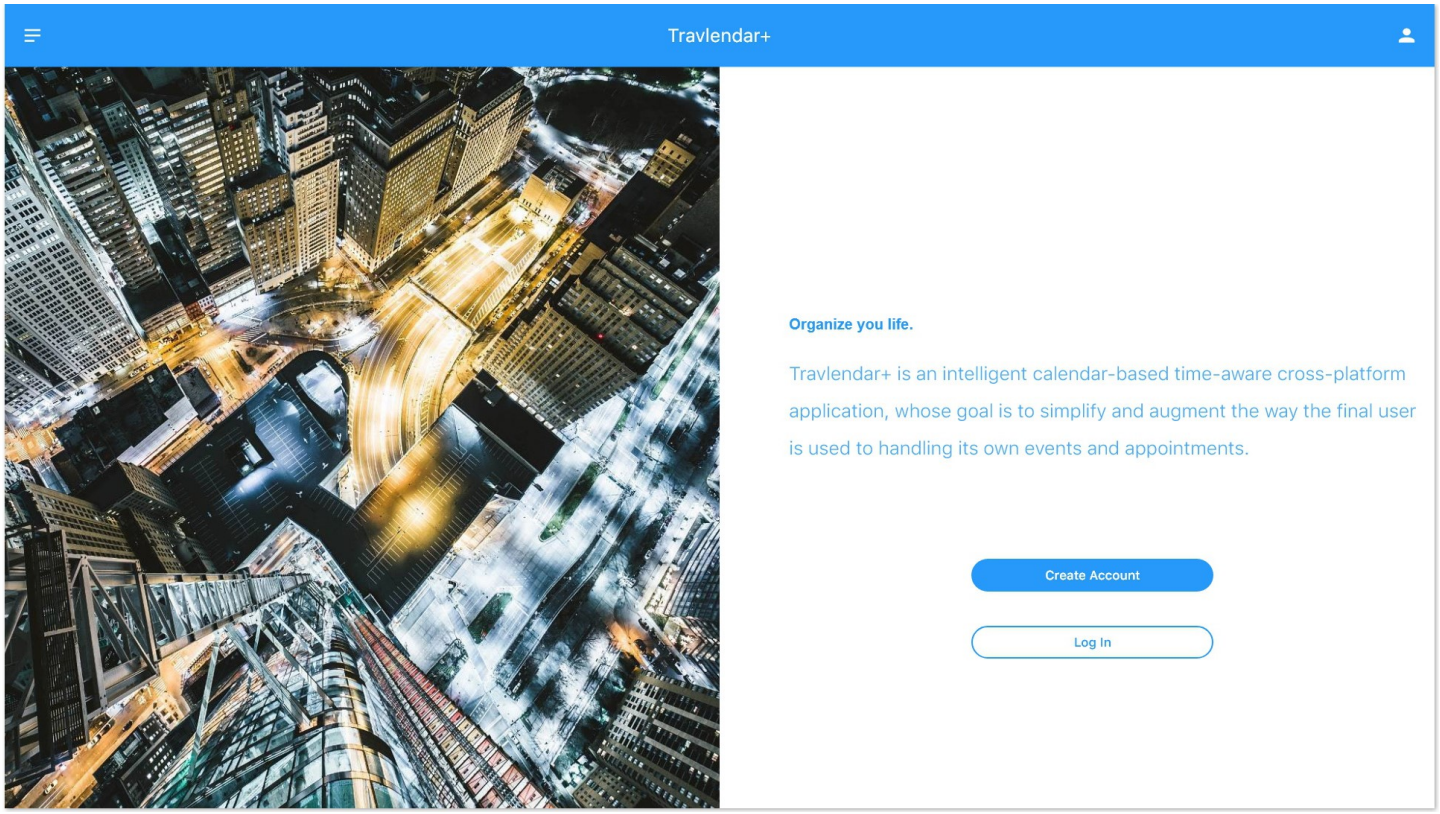

Figure 4.3: Mockup of the landing page for the web application.

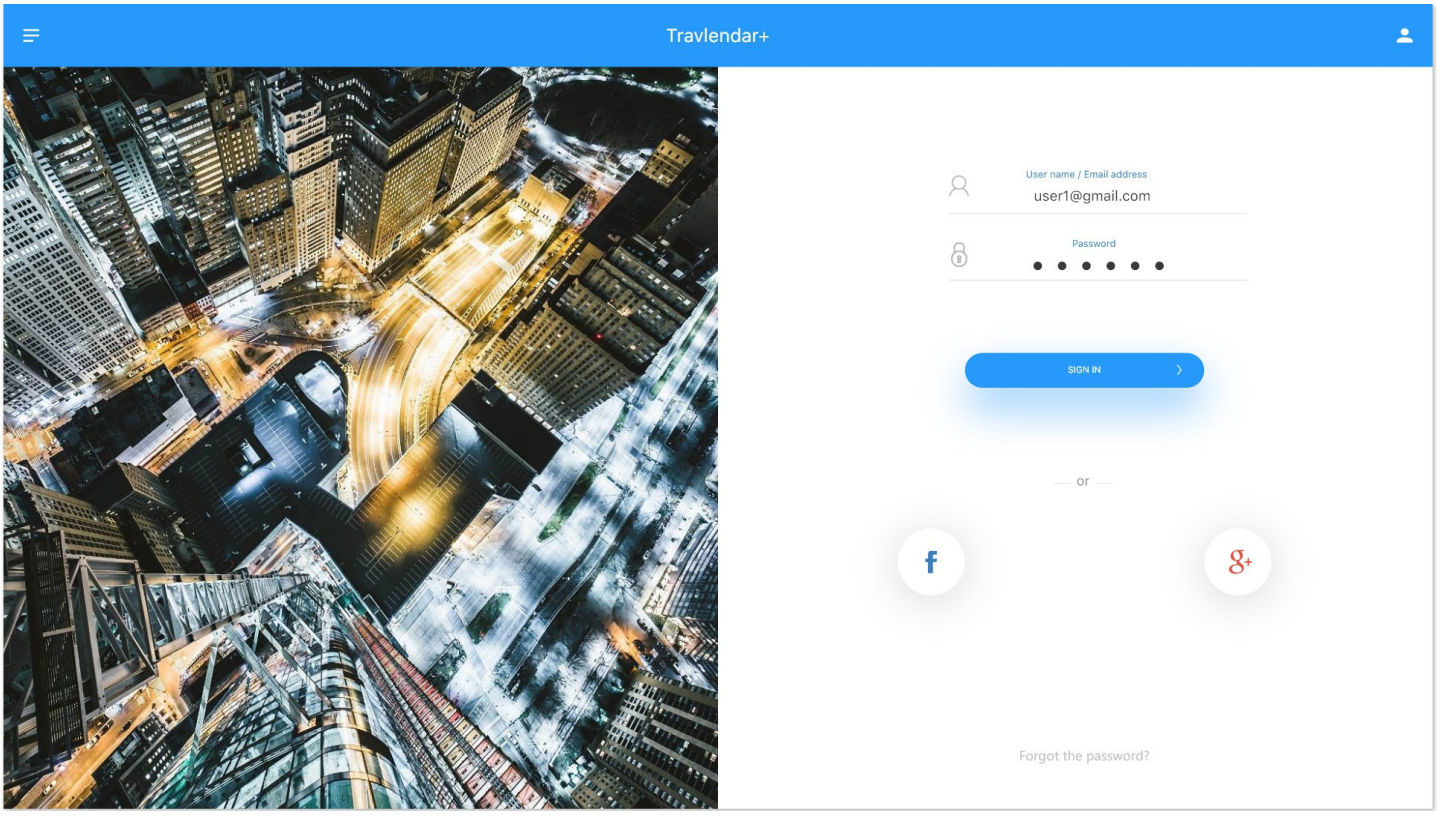

Figure 4.4: Mockup of the login page for the web application.

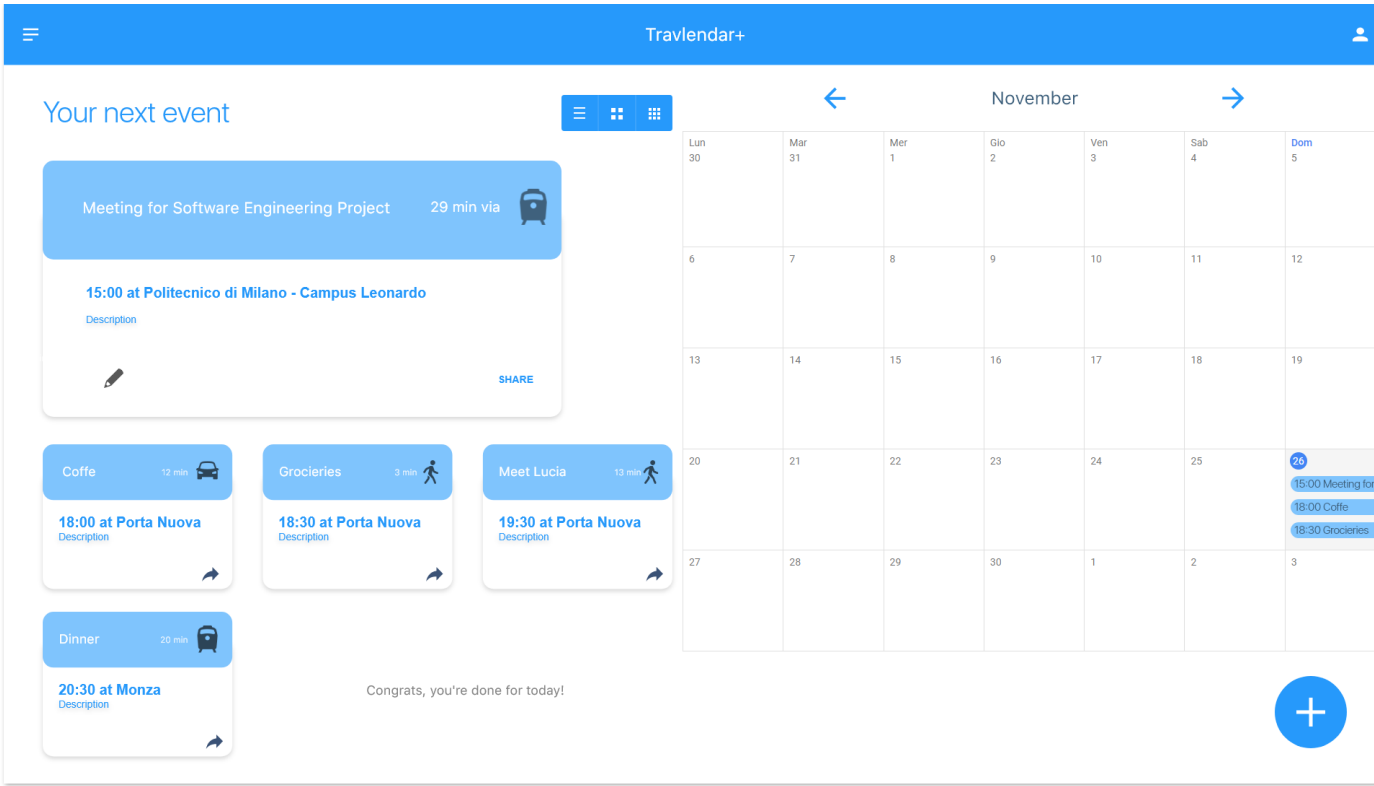

Figure 4.5: Mockup of the main dash page for the web application.

Chapter 5

Requirements Traceability

Here we explain how the system's requirements, previously defined in the RASD document, can map the aforementioned design components.

Component (DD)	**Requirements (RASD)**
AuthenticationService	• User registration with own form. • User login through third-party services. • User profile management.
SyncUserInfoService	• Permits user events to be synchronized across different platforms.
EventService	• Appointment schedules organization. • Provide overview of appointments through a grid and all schedules through a list view. • Create a new appointment. • Modify an existent appointment. • Delete an appointment.
MapService	• Travel route arrangement (location of sharing systems, find optimal paths, etc.)
PurchaseService	• Permits user to buy tickets within the application (if possible).

Table 5.1: Table that maps components to their respective functional requirements.

Chapter 6

Implementation, Integration and Test Plan

Implementation

Here we explain how a possible good order of implementation of subcomponents might help us to fulfill and build the application. Its achievement implies the following steps:

- We first plan to configure all server components (since they are those which will be queried by the client application afterwards), which, basically, ranges from creating a user pool (AWS Cognito) for data synchronization and authentication to setting up the cloud storage (AWS S3 or DynamoDB) for backing up and archiving data in order to provide the best user experience.
- We focus on the creation of the client application, whose order of modules (handlers) is illustrated in Figure 6.1.

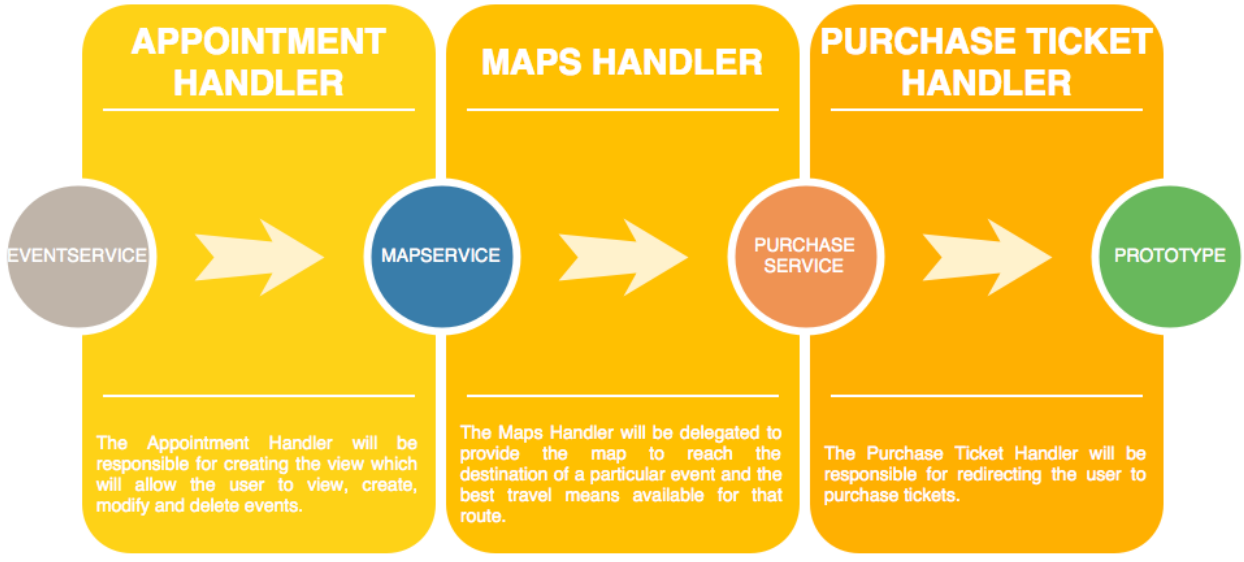

Figure 6.1: Order of main components of the client application to be deployed.

Integration

Here we describe how integration strategies are supposed to be performed. Specifically, we explain how different components are integrated in our system and which strategy we adopt in

order to integrate all components efficiently.

Elements to be integrated

The following list represent every component grouped into several subsystems:

Core Data

- AWS Cognito Sync
- AWS Data Manager
- AWS S3 / DynamoDB DBMS

Account Management

- AWS Cognito for Authentication
- AWS Cognito for Account Management

Event Management

- Appointment Manager
- Notification Manager

Maps Management

- Google Maps API
- Route Calculation Manager

Ticket Management

- Local Public Transportation API
- Tickets Handler

Interfaces

- Web Application GUI
- Mobile Application GUI (Android and iOS)

Integration Testing Strategy

The integration strategy of choice is the bottom-up approach. This choice is suitable in our case, because it allows the tester to focus on each main component as little as possible since we assume we have already the unit test on smallest component, so we can proceed from the bottom. Furthermore, the high-level subsystems described in the previous section are well separated and loosely coupled; they communicate through well-defined interfaces (AWS SDK, HTTPS), so they will be integrated quite easily at a later time. In this way components do not use stubs, so it will be possible to limit the stubs in order to accomplish the integration.

Component / System Testing

Due to the complex nature of testing the entire system, we plan the integration testing following two point of view,, both of them cataloged through a dependency-driven order: the detailed component testing and the top-view subsystem testing.

Component Testing

This section illustrates how every component will be integrated along in order to constitute a subsystem.

Core Data

The first three elements to be integrated are the AWS Data Manager, the AWS S3 and the DynamoDB Database Management System components. These are the main part of our application because every other components relies on AWS Data Manager to perform queries on the underlying data structure.

Account Management

The Account Management subsystem relies on the AWS Cognito component. Amazon Cognito lets us add user sign-up/sign-in and access control to our web or mobile application quickly, easily and safely. Furthermore Amazon Cognito provide a User Pool, a fully managed service in which we can control all the information of the users registered on Travlendar+ in a safe way. It need the Core Data subsystem to operate correctly.

Event Management

The Event Management subsystem relies on the Appointment Manager and the Notification Manager. The Appointment Manager creates the view which allows the user to view, create, modify and delete events. The Notification Manager handles every notification sent by the system to both server and client side. They both need the Core Data, the Tickets Management and the Maps Management subsystems to operate correctly.

Tickets Management

The Tickets Management subsystem relies on the Tickets Handler and the Local Public Transportation API. The Local Public Transportation API is responsible to operate with the external application in order to allow user to purchase tickets. The Tickets Handler handles the tickets purchased and the information about. They both need the Core Data subsystems to operate correctly.

Maps Management

The Maps Management subsystem relies on the Google Maps API and the Route Calculation. The Google Maps API allows us to provide a the map to reach the destination of a particular event. The Router Calculation computes the best travel means available for the a specific route. They both need the Core Data subsystems to operate correctly.

Interfaces

The Interface subsystem relies on the App GUI and the Web GUI. The both need the Account Management and Event Management subsystems to operate correctly.

Subsystem Testing

In Figure 6.2, we show the order in which each subsystem will be integrated.

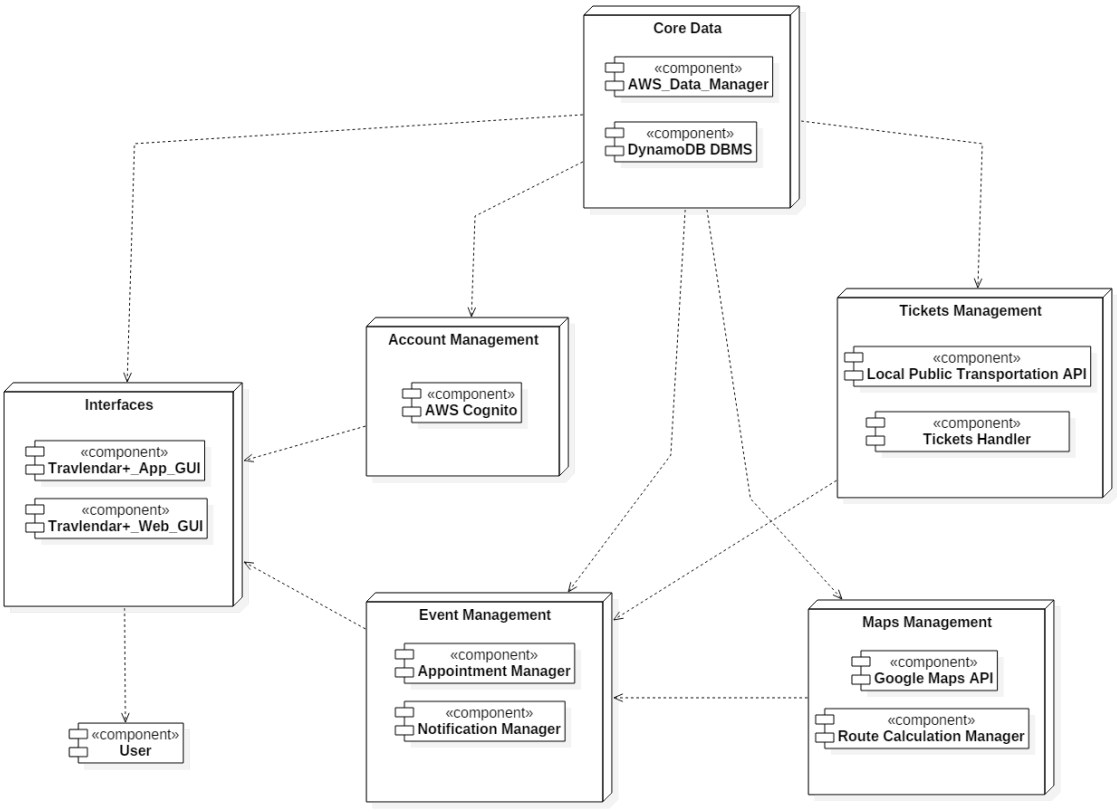

Figure 6.2: Subsystem Integration Sequence

Testing

We plan to use the following software tools to automate the various tests:

xUnit.net: is a free, open source, community-focused unit testing tool for the .NET Framework. We plan to use it for unit test of single components. xUnit features are:

- Facts are tests that are always true, which test invariant conditions.
- Theories are tests that are only true for a particular set of data.

Calaba.sh: Calabash consists of libraries that enable test code to interact programmatically with native and hybrid apps. We use this tools to test the UI of Travlendar+. The interaction consists of a number of end-user actions:

- Gesture: Touches or gestures, for instance tap, swipe and rotate.
- Assertion.
- Screenshots: Screendump the current view on the current device model.

JMater: is a powerful tool which may be used to test the performance of subsystems, in particular of our web application.

Xamarin.Profiler; a great tool that allows us to collect information about our mobile application. We use it to find potential memory leaks, resolve performance bottlenecks, and add polish to the application.

Browser's debugging: Since all the main web engines (Google Chrome, Mozilla Firefox, Safari and Microsoft Edge) will be fully supported for the web application, browser's debugging can help too.

If necessary, we might plan to write our own unit tests as well through **Xamarin.UITest**, which is a testing framework that allows us to automate our UI acceptance testing.

Bibliography

[1] Antonio Frighetto, Leonardo Givoli, and Hichame Yessou. *Requirements Analysis and Specification Document*. 2017.

[2] Elisabetta Di Nitto and Matteo Rossi. *AA 2017-2018 Software Engineering 2 — Mandatory Project goal, schedule, and rules*. 2017.

[3] Introduction to Portable Class Libraries. https://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/. Last time retrieved: 20th of November, 2017.

[4] Dijkstra's algorithm and Google Maps. https://dl.acm.org/citation.cfm?id=2638494. Last time retrieved: 20th of November, 2017.

Appendix

Hours of work

All statistics about commits and code contribution are available on our repository on GitHub. Please, remind that all commits on our repository have been reviewed together. Also the whole work has been equally divided and everyone helped each other.

- Antonio Frighetto: 21 hours.
- Leonardo Givoli: 21 hours.
- Hichame Yessou: 21 hours.