Contents

[**1. INTRODUCTION** 2](#_Toc499395135)

[**A. Purpose** 2](#_Toc499395136)

[**B. Scope** 2](#_Toc499395137)

[**C. Definitions, Acronyms, Abbreviations** 2](#_Toc499395138)

[C.1 Definitions 2](#_Toc499395139)

[C.2 Acronyms 3](#_Toc499395140)

[**D. Revision history** 3](#_Toc499395141)

[**E. Document Structure** 3](#_Toc499395142)

[**2. ARCHITECTURAL DESIGN** 3](#_Toc499395143)

[**A. Overview** 3](#_Toc499395144)

[**B. Component view** 5](#_Toc499395145)

[**C. Deployment view** 6](#_Toc499395146)

[**D. Runtime view** 7](#_Toc499395147)

[**E. Component interfaces** 7](#_Toc499395148)

[**F. Selected architectural styles and patterns** 13](#_Toc499395149)

[**G. Other design decisions** 13](#_Toc499395150)

[G.1 Programming Language 13](#_Toc499395151)

[G.2 External APIs 13](#_Toc499395152)

[**3. ALGORITHM DESIGN** 14](#_Toc499395153)

[**4. USER INTERFACE DESIGN** 17](#_Toc499395154)

[**5. REQUIREMENTS TRACEABILITY** 18](#_Toc499395155)

[**6. IMPLEMENTATION, INTEGRATION AND TEST PLAN** 18](#_Toc499395156)

[**7.1. Implementation Plan** 18](#_Toc499395157)

[**7.2. Integration and Test Plan** 18](#_Toc499395158)

[7.2.1. Sequence of Component Integration 18](#_Toc499395159)

[7.2.2. Test Plan 20](#_Toc499395160)

[**8. EFFORT SPENT** 23](#_Toc499395161)

[**9. REFERENCES** 23](#_Toc499395162)

# **1. INTRODUCTION**

## **A. Purpose**

This document presents advanced technical details related to general design of Travelander+ application which is calendar application with recommender system for feasible mobility options. With this document, authors aim to explain and illustrate design views of the Travelander+ application by relating previously presented RASD. The main audiences for this document are the developers who aim to understand or implement:

* Components of proposed system
* Interfaces between external and internal components
* Overall architecture of system
* Employed design patterns (and also possible motivations)
* Behavior of system during runtime

## **B. Scope**

The proposed system,Travelander+, aims to serve as a personal calendar and recommender system for mobility. In particular, the application provides a platform to its users that create feasible schedules with suggestions on mobility options by considering user preferences and constraints, time and location constraints, weather, traffic and public transformation information. As advanced features, the application allows to the user know about car/bike share applications and add customized event whose duration and time interval may be different and editable. Possibly, everyone who needs to plan the his/her personal calendar and mobility options may be the user of Travelendar+.

## **C. Definitions, Acronyms, Abbreviations**

### C.1 Definitions

* Event: Any appointment or customized break by the user
* Break: An event with a flexible duration assigned between the selected start and end time
* Periodic Event: An event that is repeated with a selected frequency. (e.g. break, gym)
* Activated/Deactivated: An activated mobility option is listed in the preference list with the given constraints and priority, whereas a deactivated mobility option is not listed in the preference list at all or only on the restricted times given by the user or based on the information received from APIs (e.g. walk and bike deactivated on rainy or snowy weather).
* Restricted Time Interval: A time interval set by the user for a certain mobility option to be temporarily deactivated.
* Distance Limit: A maximum distance set by the user to deactivate the certain mobility option for any larger distance.
* Preference List: A list constructed by the user with the Travlendar+’s given mobility options by giving them priorities to be selected or to activate-deactivate them based on the user’s abilities and preferences.
* Default List: Ready to use preference lists offered by the Travlendar+.
  + Minimize Carbon Footprint
  + Minimize Expenditure
* Reachable/Unreachable: An attribute of the location of the event to be added which shows if the event is actually reachable by any means of mobility based on the already scheduled events, and if it is unreachable forbids the selection of that location.
* Public Transportation Information Provider API: General name for the public transportation APIs. Its purpose is to provide official information about the public transportation of the current city.

### C.2 Acronyms

* RASD : Requirement Analysis and Specification Document
* PTIP: Public Transportation Information Provider
* API: Application Programming Interface
* JDBC : Java Database Connectivity
* AS: Application Server
* UX: User Experience
* BCE: Boundar-Control-Entity
* MVC: Model-View-Controller

## **D. Revision history**

Version 1.0

## **E. Document Structure**

1. Introduction:

2. Architectural Design:

3. Algorithm Design:

4. User Interface Design:

5. Requirements Traceability:

6.Implementation, Integration and Test Plan:

7. Effort Spent:

8.References:

# **2. ARCHITECTURAL DESIGN**

**A. Overview :**

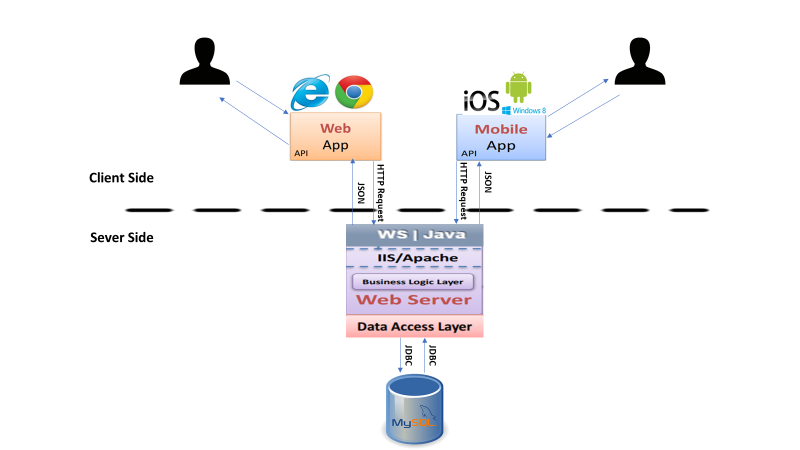
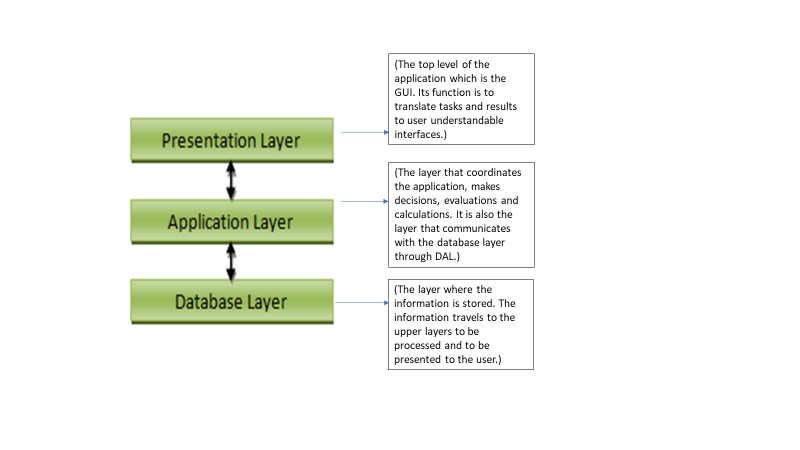


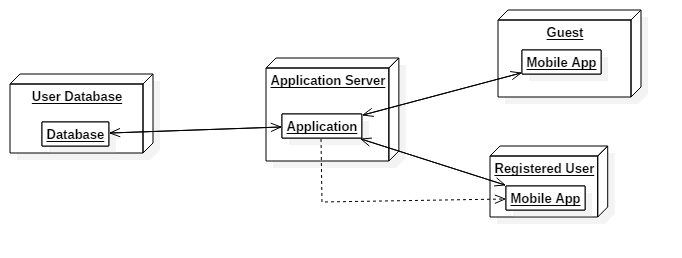
Figure 0: General Architecture

Travelander+ is built on three-tier architecture which is composed of presentation, application and data layer. These layers are charged by the following:



**High-level components and their interaction:**

Travelander+ has four higher level elements: Guest, registered user, application server and user database. Travelander+ fully operates with these components and their communication. The main component is application server (AS) which is charged for necessary computations of feasible schedules, mobility recommender system. AS communicates with both guest and registered user in different ways. The communication between registered user and application server initiates with synchronous message coming from user to AS (Application Server) which corresponds to user log-in. After this message, another synchronous message goes to user database which is consisted of all user content from AS, and queries the user credentials. In case that query is found and log in is successful, the mobile application allows to user add/edit/delete events from his/her schedule, changing on user preferences and travel constraints, visualization of personal calendar which generates synchronous messages from registered user through AS and thereafter user database. Also, AS is able to generate asynchronous messages for notifying the user about generated schedule, mobility options according to his/her input recorded in user database. Another messaging is done between AS and user database in synchronous way. During its computations, AS needs user content such as event time and locations, mobility preferences, travel constraints etc. Therefore, AS does queries to user database and after that database returns requested elements.

 Figure X. Higher Level Components

## **B. Component view**

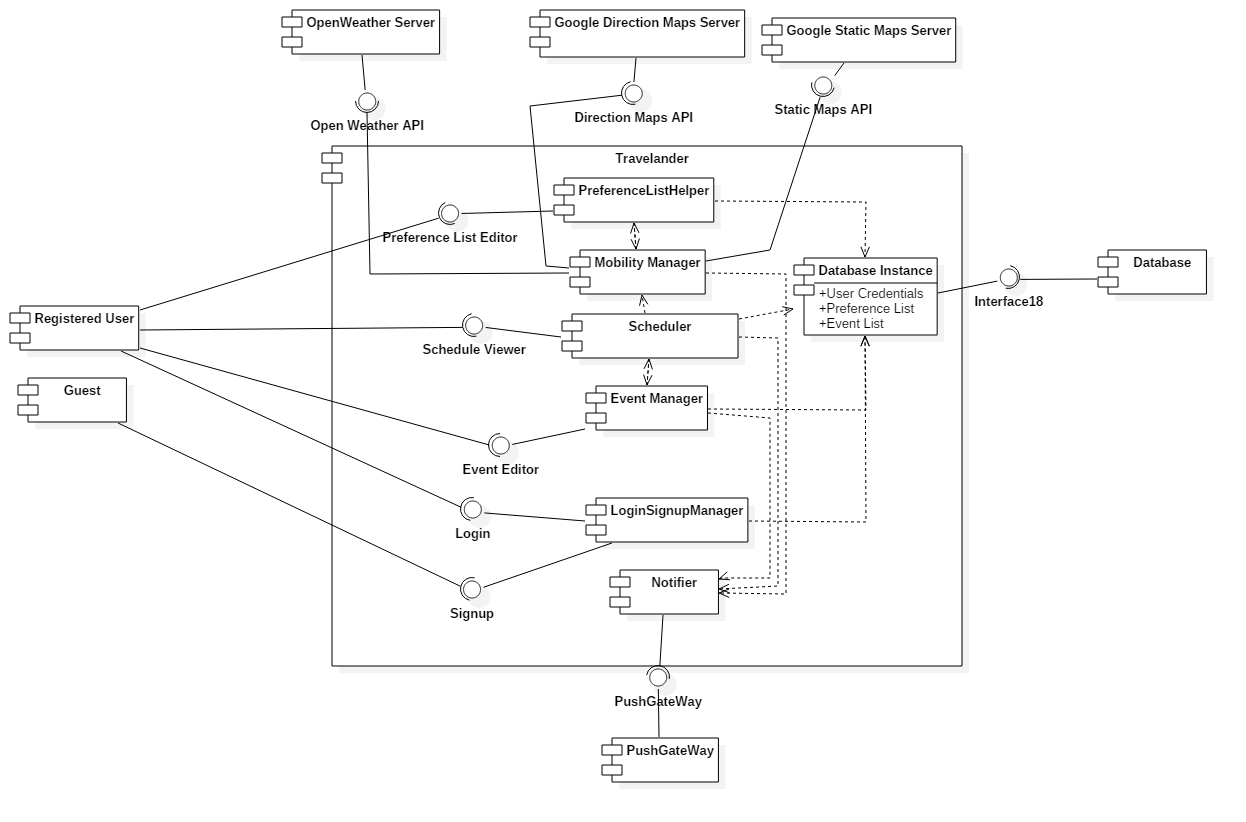


Figure Y. Component Diagram

**Registered User:** User who is previously registered and authenticates with correct credentials. All data related to registered user are recorded in the database.

**Guest:** User who is not currently registered or logged in.

**Event Manager:** This component allows to user to add/delete/edit event operations through his/her schedule. It has dependency to scheduler component for informing user whether the requested operation is feasible or not.

**Scheduler:** It computes the feasible schedule for the user for currently added events and under mobility constraints (such as user preferences, weather, traffic, etc.)

**Mobility Manager:** Mobility manager choses feasible mobility option by running elimination algorithms.

**Register Manager:** This component proceeds the user registration and login operations by the validation of the database.

**Database Instance:** This is a model for representing the compact elements in the database.

**Preference List Helper:** This component provides an interface to user to input his/her mobility preference list.

**Notifier:** This component enables to application to send notifications to user by deploying push notifications.

**Database Entry Instance:** This is a model in the application for a database entry and it establishes an interface with the database.

## **C. Deployment view**

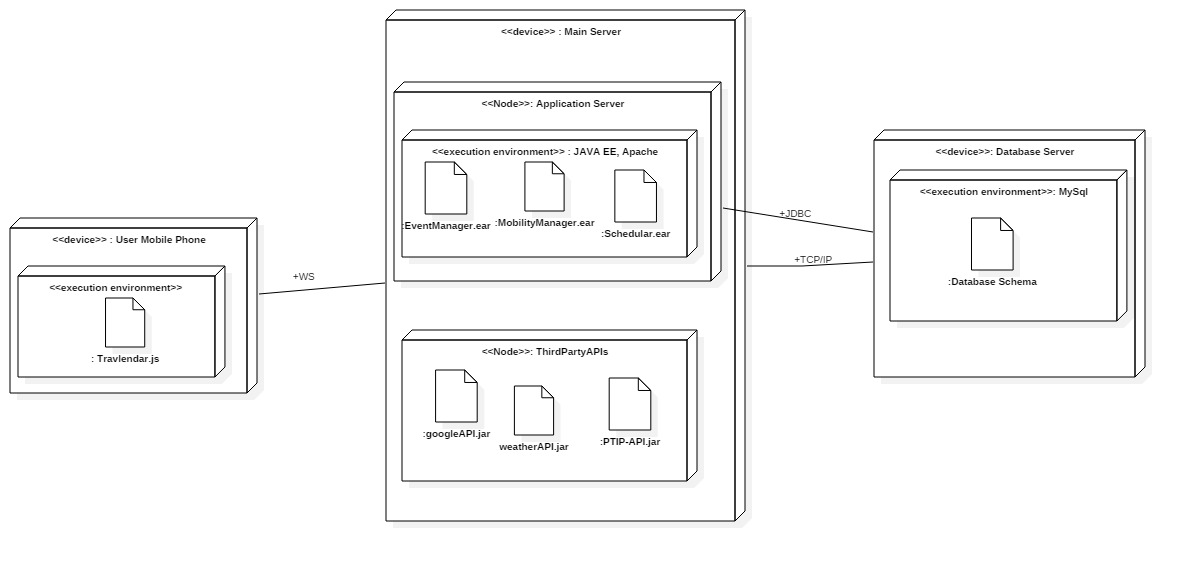


Figure XX: Deployment Diagram

**D. Runtime view**

### D.1. New User Registration

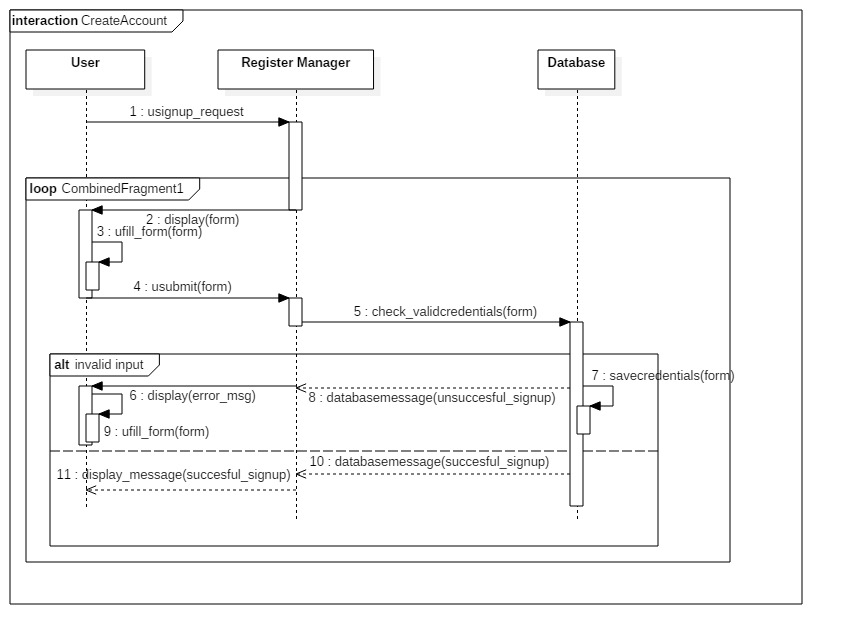


Figure T. Create Account Sequence Diagram

The activity starts when the user submits the new account details on the Travlendar+ application through the signup form, the Register Manager then checks the validity of the entered credentials and if the inputs are in the correct form stores the new account tuple on the database, if the inputs are invalid Travlendar+ app displays an error message to the user.

### D.2. User Login

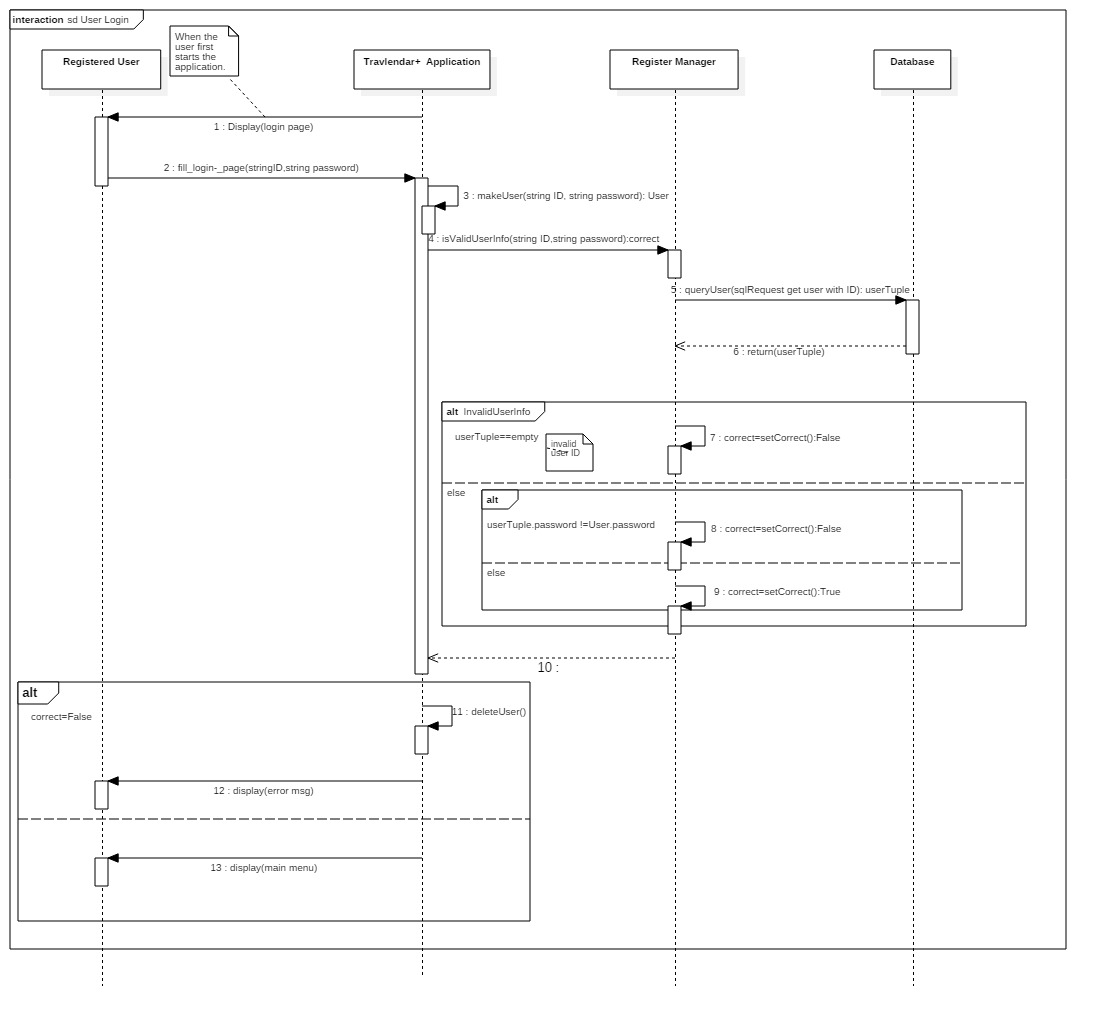


Figure G. User Login Sequence

The activity starts when the user submits his/her existing account’s credentials on the Travlendar+ application through the login form, the Register Manager then checks the validity of the credentials by querying the database on the entered userID, and confirming whether or not the account exists and if it does the password is correct. If the credentials are correct Travlendar+ application displays the main menu to the user, otherwise it displays an error message.

### D.3. Add Event

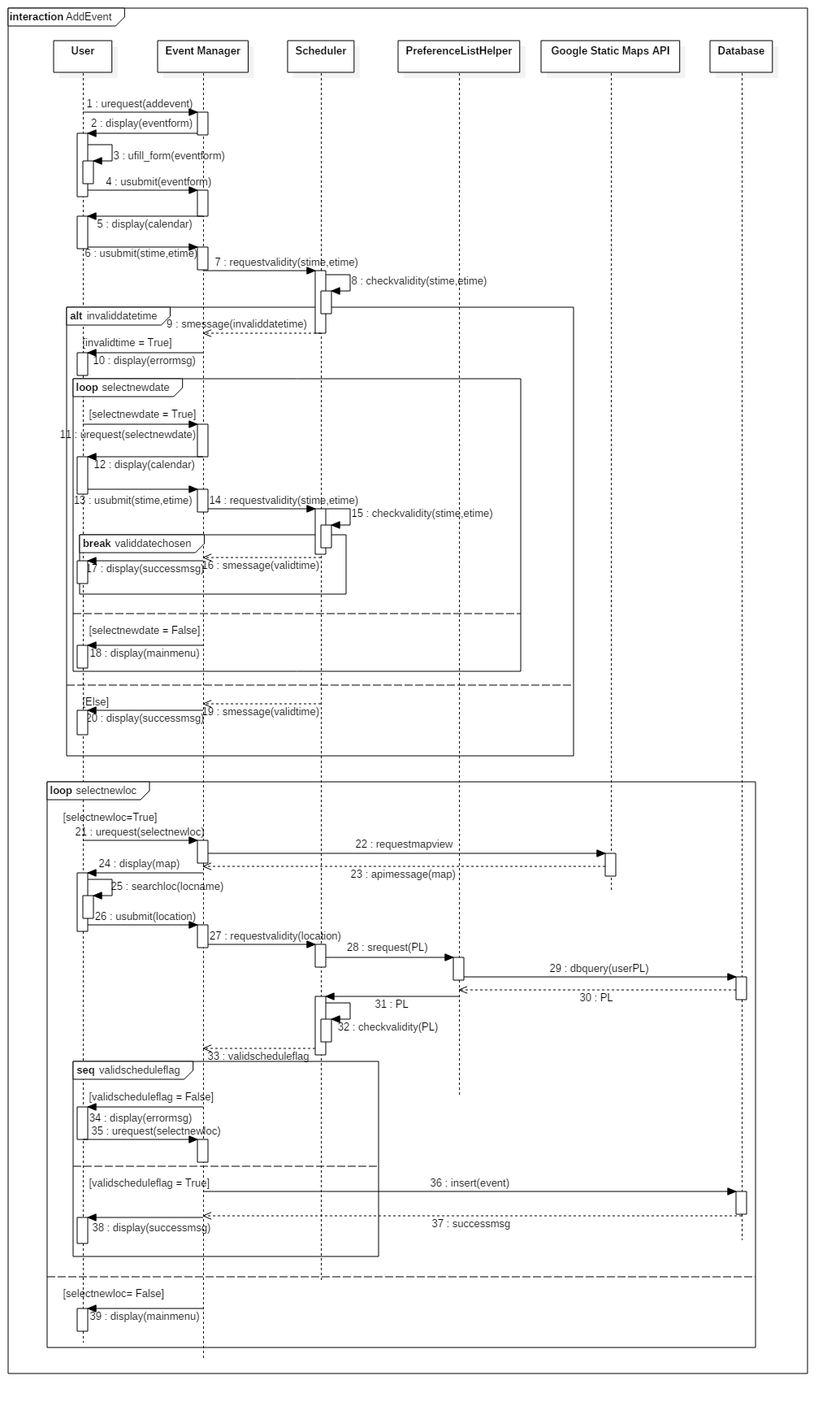


Figure H. Add Event Sequence Diagram

The activity starts when the user selects Add Event on the main menu of the Travlendar+ application. The user enters the event name and then moves onto the event start and end time selection, Travlendar+ app displays the calendar view for the time interval selection. Event Manager requests a validity check from the Scheduler, the Scheduler retrieves the current Event List from the database, the makes the necessary time and overlap constraints checks and returns if the time interval is allocated or not. If it is, Travlendar+ displays an error message to the user and suggests new time interval selection. If the user decides to select new start and end times for the event, the same validity checking process repeats, if the user does not want to select a new time interval Travlendar+ returns the main menu and the event is not stored on the database. If the time interval is valid, Event Manager gets the related Map from the external API and Travlendar+ displays the map for location selection. The user submits the location, then Event Manager requests reachability check from the Scheduler, the Scheduler requests the current user mobility preference list from the Preference List Helper. Then checks if any mobility option can be assigned to the new event considering the free time slot between the new and existing events on the list and the mobility option travel durations. If the location is not reachable Travlendar+ displays an error message and asks for new location selection. If the user selects a new location, the same reachability check repeats, if not Travlander+ returns the main menu. If the location is reachable, the Event Manager stores the new event on the database by updating the event list.

### D.4. Manage Mobility Preferences

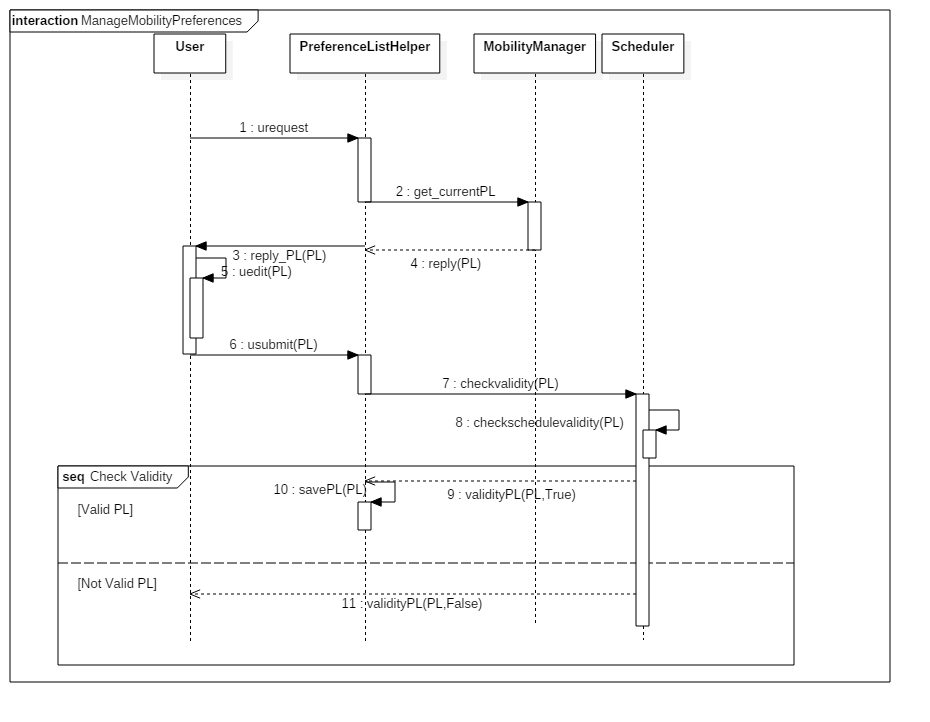


Figure P. Manage Preferences Sequence Diagram

The activity starts when the user constructs or edits his/her preference list through the Preference List Helper, Mobility Manager, aside from the user input, updates the Preference list by periodically retrieving updates from the external APIs (openWeather,googleMaps etc.) and changing the mobilities activation status base on weather condition, transportation schedule etc.

### D.5. Add Break

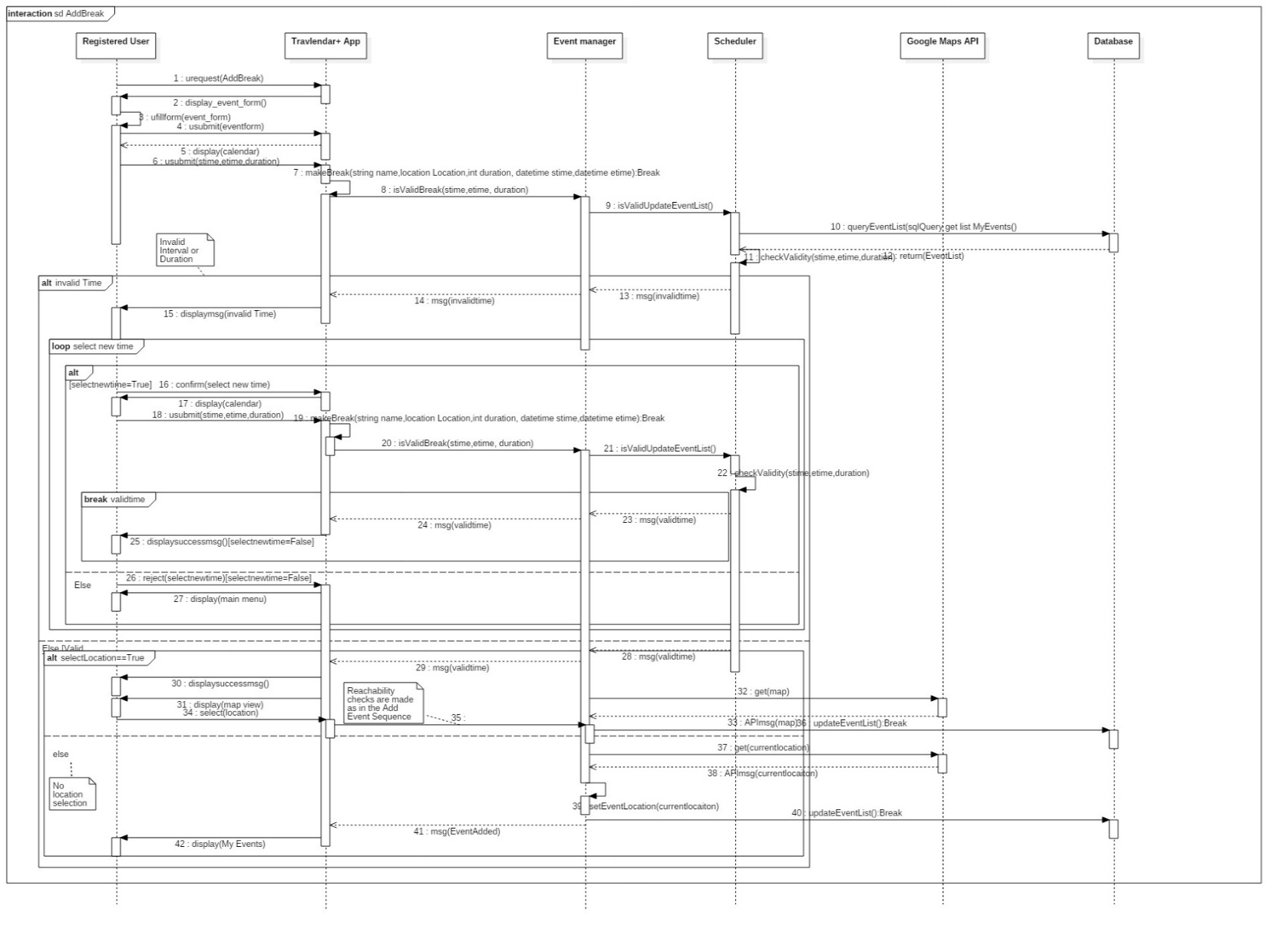
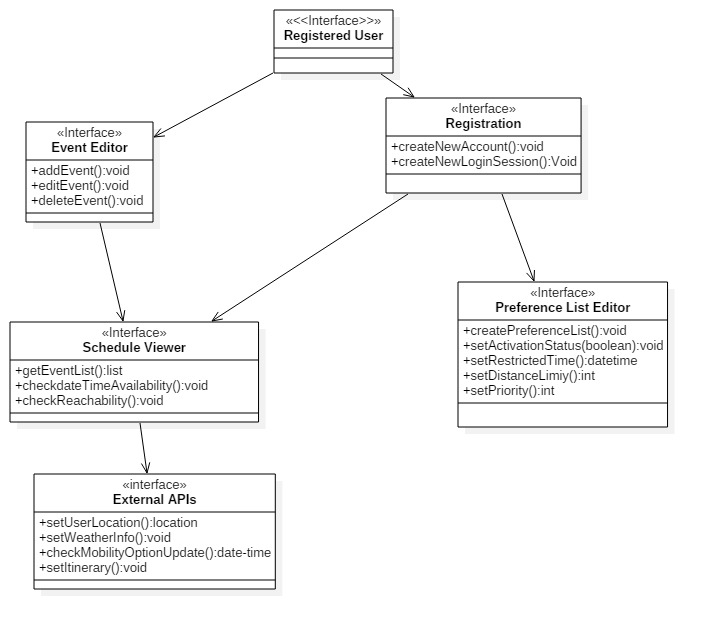


Figure R. Add Break Sequence Diagram

The activity follows the same sequence with the Add Event except the Scheduler checks different time interval constraints and overlapping rules and it also checks the validity of selected break duration within the interval.

## **E. Component interfaces**



**F. Selected Architectural Styles and Patterns**

### F.1. Overall Architecture

The application, as mentioned in the overview section, will have a 3-tier architecture which is a client-server architecture in which the business logic, data access, data storage and user interfaces are developed and maintained separately. It consists of a presentation layer (thin client), a business logic layer, and a database access layer, and offers a number of benefits, such as:

1. Ability to update or change any tier without affecting the others.
2. Increased reliability and independence from the underlying services.
3. Since the tiers are independent, it is possible to parallelize the development and use different developers.
4. Increased ability to scale the application up.

### F.2. Design Patterns

* **MVC-** Model-View-Controller pattern will be used in our application in the presentation layer, and will be implemented using AngularJs framework. The main functions and the flow of the model are described below:

1. Controller receives the requests and decides if any business logic will be used for it.
2. If business logic is not needed, then it will directly go to the appropriate View. Such as, request of a SignUp form.
3. If business logic is required, then it will process the request by calling the necessary BLL class, and then redirects to View.
4. In the view if there is any data to be displayed, the data will be taken from the appropriate model.

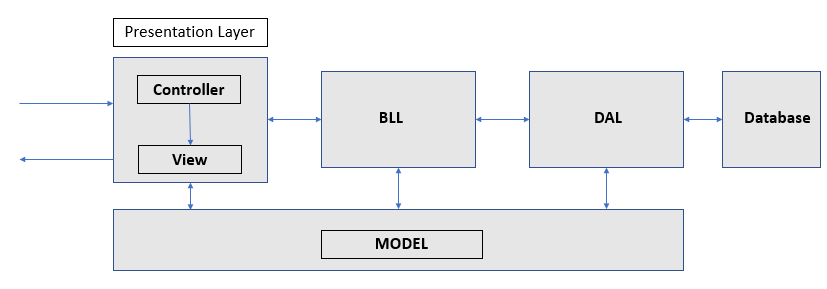


Figure YY. MVC Flow Diagram

* **BCE-** Boundary-Control-Entity pattern is used in the business logic layer of the Travlendar+ application. The BCE diagram is given below:

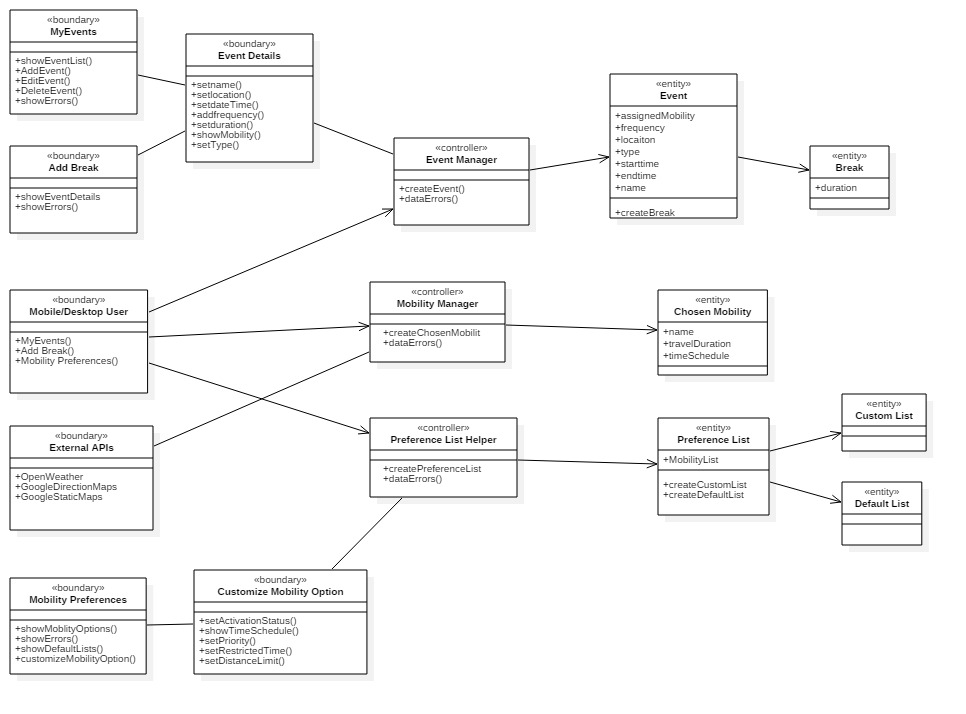


Figure ZZ: BCE Diagram

## **G. Other design decisions**

### G.1 Programming Language

The overall implementation will be done in Java programming language. Java is widespread and well-known programming language for all type of developers. Also, it enables easy implementation and usage of each external (e.g. External APIs, DBMS) and internal components (e.g. Scheduler Algorithms). During its long history, Java language proofs its reliability and applicability to multi-tier systems which is the case for Travelander+.

### G.2 External APIs

One of the additional design decision is related to external APIs that are deployed for getting information on locations and maps, travel durations and weather forecasting. During researches on several APIs, Google Map APIs related to static map and directions and OpenWeather APIs are founded as viable choices by considering the overall system needs and availability of the APIs.

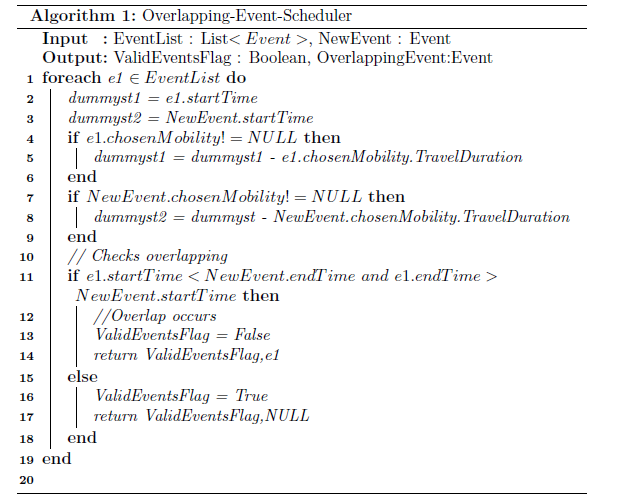
OpenWeather API is an open source application that provides weather forecasting data in a simple and clear JSON, XML or HTML format which can be easily processed by employed parts of the system. Higher accuracy and reliability of the application is also another reason why it is chosen. The company, itself, collects data from 40.000+ stations and runs advanced data science algorithms for the most accurate forecasting. In addition, clear and ordinate documentation is also presented for the developers [X].

Google Maps Direction is another API choice of our system which basically provides all the necessary information related to suggested travel options. Particularly, It enables to reach required information of estimated travel durations and availability of mobility options. There exists a few API that can fully or partially give this kind of services and Google Maps Direction is the most advanced and accurate API among others. Also, the most enhanced documentation is served for the developers who wishes to integrate it to any kind of application. Similarly, the presented data is in JSON format which can be easily parsed and processed by the commands of mainly used developer frameworks. [Y]

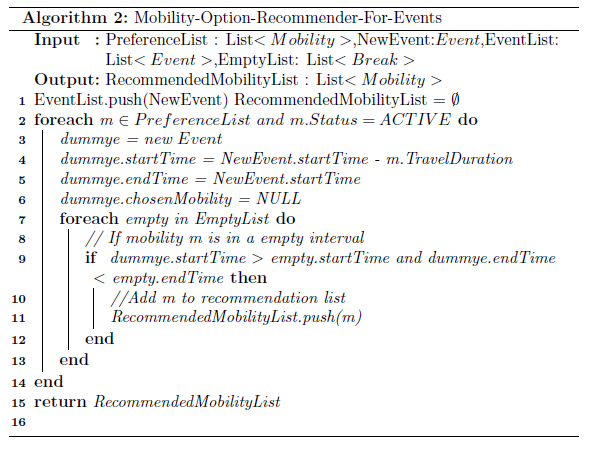
Google Static Maps API mainly provides the map visualization and advanced location search. This API is deployed due to its reliability, enhanced location database, user friendly map visualization. Besides, Static Maps has more compatibility with other external APIs. Similar to Direction API, its data is also represented in JSON format and enhanced documentation is presented for all kind of developers. [Z]

**3. ALGORITHM DESIGN**

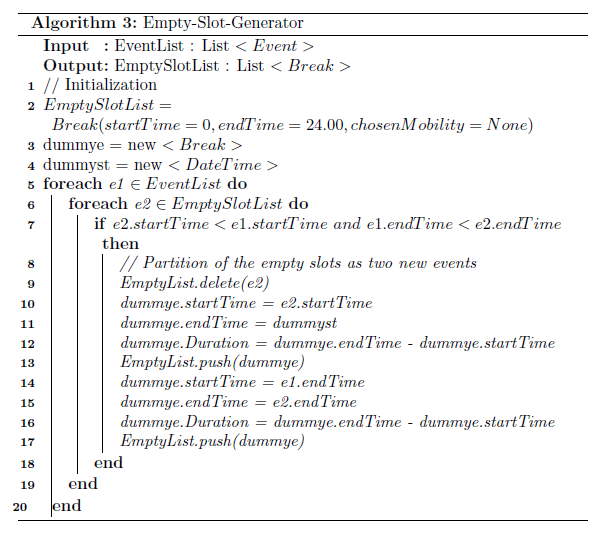
The most important algorithms for this project is related to scheduling and mobility option recommendation algorithms that can generate a feasible calendar by regarding event/break overlaps, travel durations, user preferences and constraints. In the first five algorithm, necessary functions for addEvent and addBreak abilities of application are defined. In the last two, a basic flow is described for addEvent and addBreak by using previously defined functions.



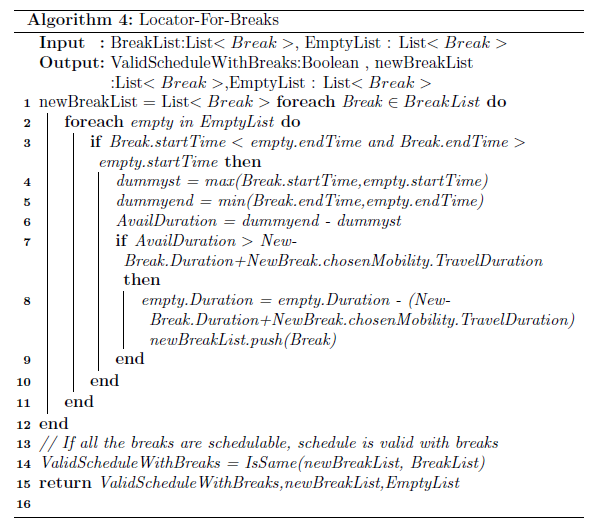
Algorithm 1 checks the whole schedule whether any overlapping event with new event exist or not. For this function, breaks are not considered only events are checked.



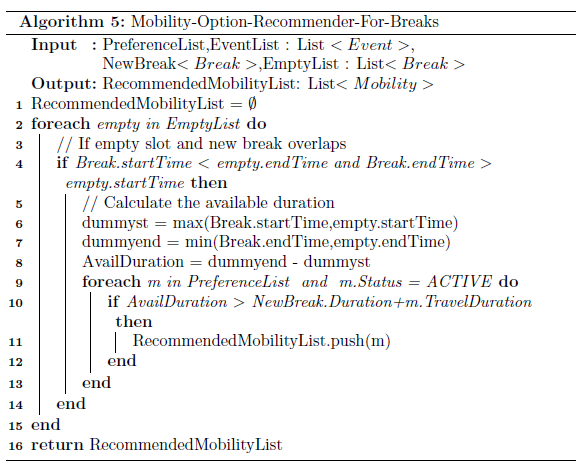
Algorithm 2 is the recommender algorithm that enables to create a mobility recommendation list for a new event. Basically, it lists all the preferred mobility options that fit with the overall schedule. Mobility options are assumed to be listed in an order in the preference list by the user and the most preferred one by the user is picked first through the iterations.



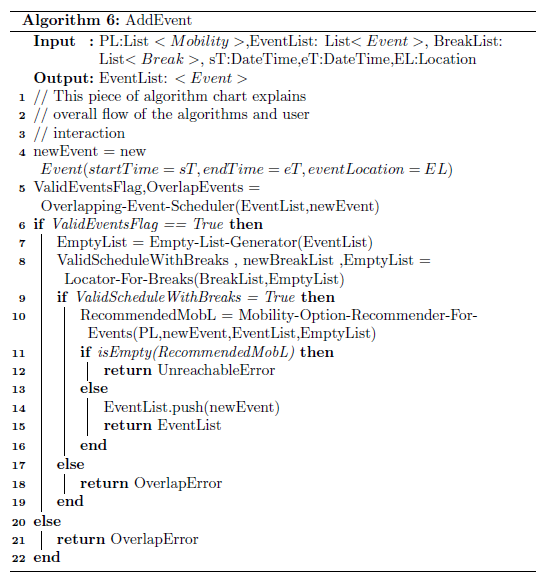
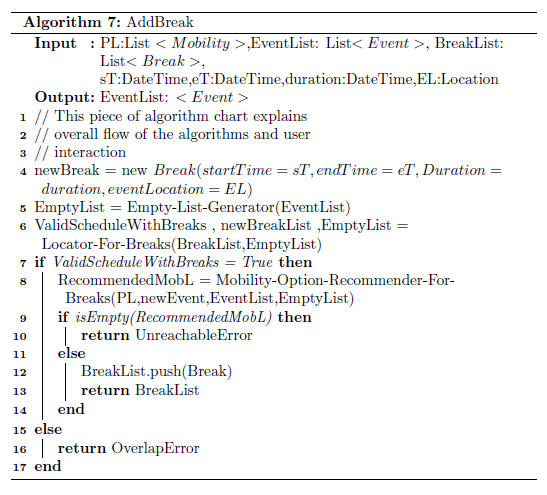
Algorithm 3 generates the empty slots that the breaks might fit in.



Algorithm 4 locates the existing breaks to given event and empty slot list.



Algorithm 5 generates mobility option recommendation list for the given break.



Algorithm 6 and 7 is the main flows for the addEvent and addBreak operations of application, respectively. These operations require calling the previous functions in a correct order with correct parameters for reliability and maintainability of the schedule. addEvent operation requires preference list, new event details and existing events and breaks as user and database input (see UX diagram/AddEvent for user input details.). During the algorithm, it checks whether the new event has any overlap with other events, all breaks are still schedulable with new event addition and finally tries to generate mobility recommendation which does not create a conflict with the rest of the schedule. Since breaks durations can be shifted according to other events and breaks, addition of a new event requires new adjustment on break times. addBreak operation is easier than the former one. It requires preference list, event list, break list, and new break details. (see UX diagram/AddBreak for user input details ) Similar to addEvents, the algorithm obtain empty slots from adding existing events and breaks. After that, an empty slot which can cover new break and related mobility options is searched. Both of these flows guarantee that all the events and breaks can be doable and reachable with suggested mobility options which are proposed under user and environment constraints.

**4. USER INTERFACE DESIGN**

## **4.1. Mockups**

The mockups can be found in RASD under the Specific Requirements in section A1.1.

## **4.2. UX Diagram**

The UX Diagram shows how users perform the main functions of the Travlendar+ application.

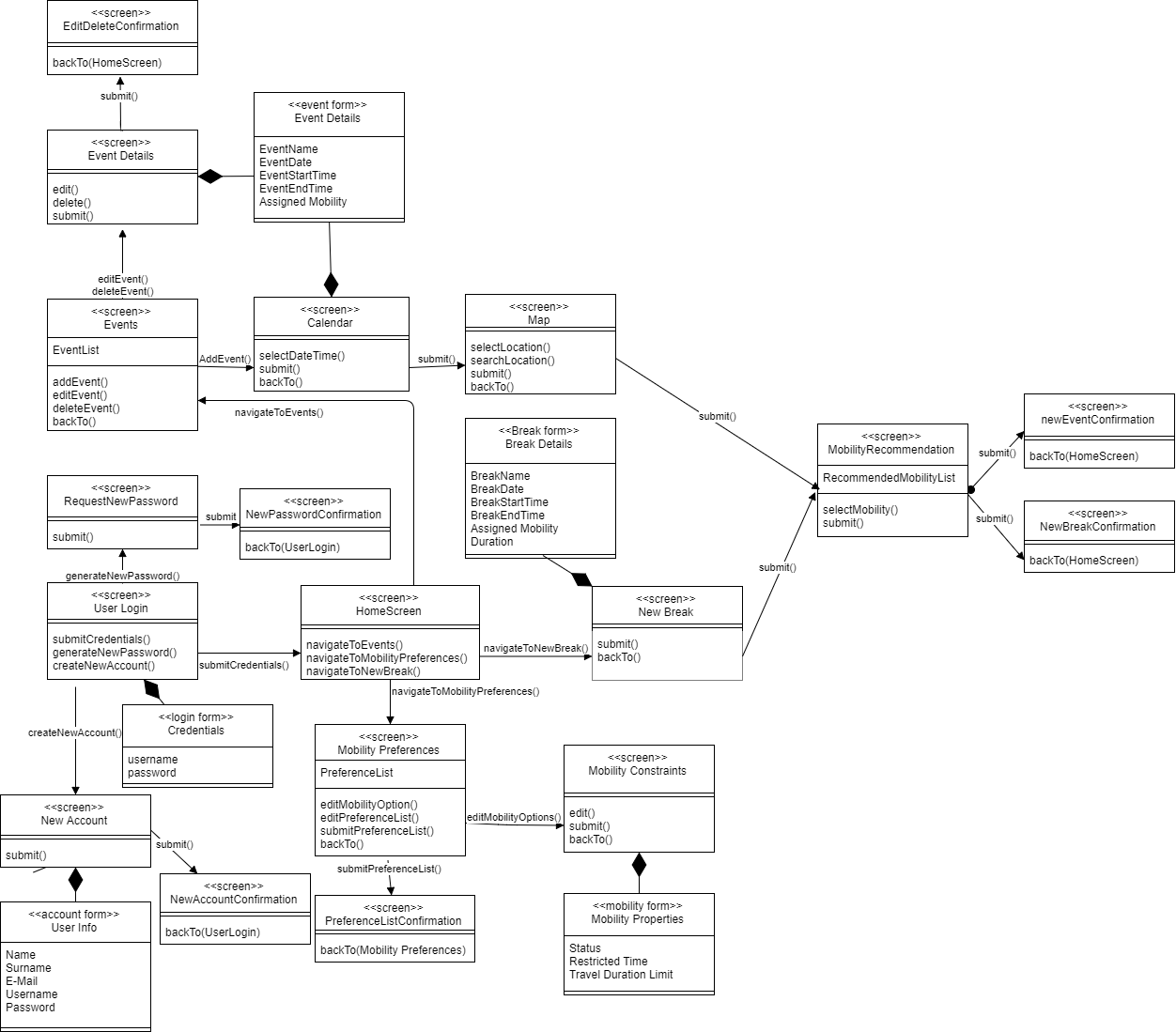


Figure TT. UX Diagram

**5. REQUIREMENTS TRACEABILITY** :Explain how the requirements you have defined in the RASD

map to the design elements that you have defined in this document.

**6. IMPLEMENTATION, INTEGRATION AND TEST PLAN**

## **6.1. Implementation Plan**

After the Requirement Analysis and Specifications Document is completed and the prescriptive architecture and the design patterns of the system to be developed is determined, the implementation of the components of the system will be done using frameworks suitable to the decided patterns. The implementation of the systems will follow a bottom-up approach aligned with the integration strategy, starting from the low-level modules and working up to top i.e. starting from the database layer related components up to the presentation layer which enables more time for developments and testing.

In order to test the integration of components, some low-level modules and external APIs that are deployed by most of the upper-level components should be implemented, and the main features of the rest of the components should also have been developed and their unit tests should have been performed. Such as,

* The DBMS should be configured and operating, and the database instance component should be fully implemented in order to test all the components that need access to the database.
* The testing of the Mobility Manager requires that the scheduler component, the openWeather API, the Google static Maps API and the PTIP APIs ( the Google direction Maps API ) to be fully implemented and available.
* For testing the Event Manager component, the scheduler must be fully implemented.

## **6.2. Integration and Test Plan**

Following the bottom-up strategy of the implementation, a similar approach is taken for integrating the components as in starting with the testing of the components that are least dependent to the other components in order to properly function. After testing these single components, the testing will be continued by the subsystems that are constructed by these previous components. This integration strategy will enable the parallelization of the implementation and development by allowing the subsystems to be tested as soon as their required main features are completed.

### 6.2.1. Sequence of Component Integration

Three of the main components of the User Mobile/Web Services need the DBMS to fully and properly function. Also, the Mobility Manager component requires all the external APIs to be fully available and the scheduler component to be implemented. Finally, the Event Manager component also requires the Scheduler component to be completed and the Notifier needs access to the Push Gateway.



Figure A. Preference List Helper Component

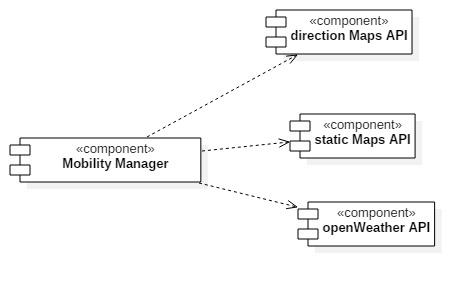


Figure B. Mobility Manager Component

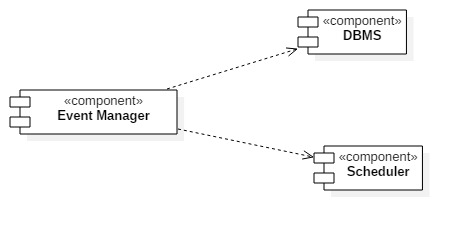


Figure C. Event Manager Component

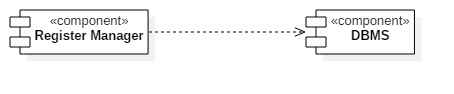


Figure D. Register Manager Component



Figure E. Notifier Component

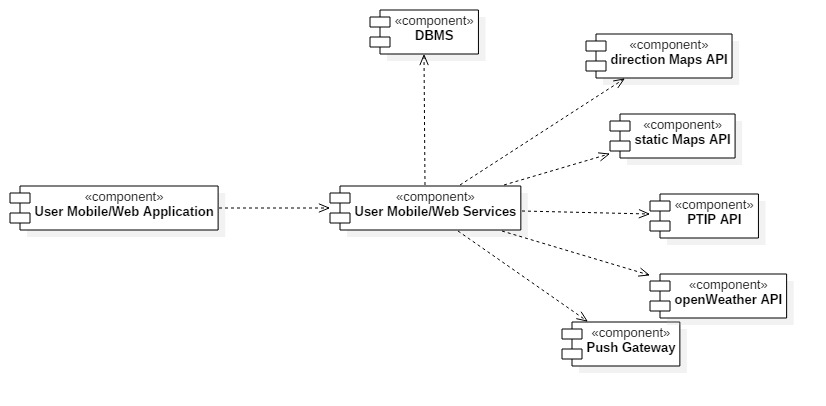


Figure F. System Integration

The integration sequences of the system components are given on the figures above, the arrows indicate dependency on the other component.

### 6.2.2. Test Plan

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.1** |
| **Test ID** | I1.1 T1 |
| **Components** | Register Manager, DBMS |
| **Input specification** | Create New Account |
| **Output specification** | Check if the new account is available from the database   * if the method is invoked on a non-valid object, the account is not created |
| **Description** | Register Manager sends an update query to the DBMS in order to append the new account to the list of existing accounts |
| **Environmental Needs** | - |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.1** |
| **Test ID** | I1.1 T2 |
| **Components** | Register Manager, DBMS |
| **Input specification** | Login with existing account |
| **Output specification** | Check if the login is successful   * Exception: if the method is invoked on a non-valid object, login is denied |
| **Description** | Register Manager queries the DBMS in order to verify the credentials |
| **Environmental Needs** | - |

|  |  |
| --- | --- |
| **Event Manager** | **Integration test case I1.2** |
| **Test ID** | I1.2 T1 |
| **Components** | Preference List Helper,DBMS |
| **Input specification** | Save Preference List |
| **Output specification** | Check if the default or custom preference list is successfully stored in the DBMS. |
| **Description** | Preference List queries the DBMS to verify the user preferences are successfully stored |
| **Environmental Needs** |  |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.3** |
| **Test ID** | I1.3 T1 |
| **Components** | Mobility Manager, External APIs |
| **Input specification** | Retrieve weather, mobility option schedule, location and itinerary updates |
| **Output specification** | Check if the updates are retrieved with the given periodicity |
| **Description** | Mobility Manager retrieves necessary information from the external APIs to be used for updating the preference list and the schedule |
| **Environmental Needs** |  |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.4** |
| **Test ID** | I1.4 T1 |
| **Components** | Scheduler,DBMS |
| **Input specification** | Get EventsList |
| **Output specification** | Check if the current EventList is successfully retrieved from the DBMS |
| **Description** | Scheduler retrieves the current EventList from the DBMS to be used on time and reachability checks. |
| **Environmental Needs** | - |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.5** |
| **Test ID** | I1.5 T1 |
| **Components** | Preference List Helper, Mobility Manager, DBMS |
| **Input specification** | Update Preference List |
| **Output specification** | Check if the preference list is correctly updated and stored in the DBMS |
| **Description** | Preference List Helper receives periodic updates from the Mobility Manager based on the information coming from the external APIs, and updates the list item attributes accordingly and saves. |
| **Environmental Needs** | I1.2, I1.3 |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.6** |
| **Test ID** | I1.6 T1 |
| **Components** | Event Manager, Scheduler, DBMS |
| **Input specification** | Add Event/ check date-time |
| **Output specification** | Check if the new event to be added has a valid time interval |
| **Description** | Event Manager requests validity check from the Scheduler, the Scheduler retrieves the EventList from the DBMS and checks the validity. |
| **Environmental Needs** | I1.4 |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.6** |
| **Test ID** | I1.6 T2 |
| **Components** | Event Manager, Scheduler, DBMS, Preference List Helper, Mobility Manager |
| **Input specification** | Add Event/ check location |
| **Output specification** | Check if the new event to be added has a valid time interval |
| **Description** | Event Manager requests validity check from the Scheduler, the Scheduler gets the current EventList from the DBMS, checks if a mobility can be assigned to the free time slot between two events based on user preference list and travel durations and returns if the location is reachable by any possible option or not. |
| **Environmental Needs** | I1.6T1, I1.5 |

|  |  |
| --- | --- |
| **Register Manager** | **Integration test case I1.6** |
| **Test ID** | I1.6 T3 |
| **Components** | Event Manager, Scheduler, DBMS |
| **Input specification** | Add Break/ check date-time |
| **Output specification** | Check if the date-time interval and the duration of the new break to be added can be placed on the current EventList |
| **Description** | The Event Manager requests validity check from the Scheduler, the Scheduler retrieves the current EventList from the DBMS and checks if the break duration that is not the empty slot can be placed on the list |
| **Environmental Needs** | - |

**7. EFFORT SPENT** :In this section you will include information about the number of hours each

group member has worked for this document.

# **8. REFERENCES**

[X]. OpenWeather API : <https://openweathermap.org/api>

[Y]. Google Static Maps API: https://developers.google.com/maps/documentation/static-maps/

[Z]. Google Maps Direction API: <https://developers.google.com/maps/documentation/directions/>

[T] RASD Document 1.0