POLITECNICO
MILANO 1863

TRAVLENDAR+

DD

Design Document

Kostandin Caushi 898749

Marcello Bertolini 827436

Raffaele Bongo 900090

Date 26/11/2017

Version 1

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

This document has the aim of entering into the detail of Travlendar+ system. We will show the software architecture that we have designed for our system with different levels of abstraction, analyzing deeply the main components. Additionally we will exhibit a runtime view of the system, showing as the various components will interact between themself specifying architectural and design pattern used. Finally we will present the critical algorithm implemented, a requirement - software componets correspondency and an high level plan about implementig and integrating the various components.

1.2 Scope

Travlendar+ is a calendar-based application that has the aim of managing the many meetings, events and appointments that a user has to deal with every day.

The system will let the user create events in his personal calendar, checking if he is able to reach them on time and supporting his choices about the way of reaching the location.

Travlendar+ will also give the user the possibility to buy tickets of a town's public and private means of transport and it will also allow him to manage his travels to reach other cities, creating specific travel events in the calendar section. The system will offer other additional features :

- The possibility to register the season ticket for the public transport. Travlendar+ will notify the user when the expiry date is near.
- The possibility to set the starting time, ending time and the preferred duration of every day lunch. The system will guarantee to reserve at least 30 minutes for this purpose.
- In case of outdoor trips, the user will be able to insert the period he will spend out of town and the system will suggest him the most convenient transport tickets available, keeping in mind the information given.
- The possibility of setting the anticipation time for reaching the various events. The system will warn the user when he needs to leave in order to arrive on time.

1.3 Definitions

1.4 Acronyms

1.5 Abbreviations

1.6 Reference Documents

1.7 Document Structure

2 Architectural Design

2.1 Overview

In this chapter we will analyze the proposed architecture and components of the Travlendar+ system.

The proposed architecture has three tiers :

- *Presentation Tier :* represented by Browser and Mobile App. It's how the system shows himself to the user.
- *Web and Business Tier :* represented by Web Server, which contains javascript and html code in order to create dynamic pages, and Application Server, which contains all the system's logic (the so called Business Logic).
- *Database Tier :* represented by DB Server, that contains and manages persistent data in an efficient way.

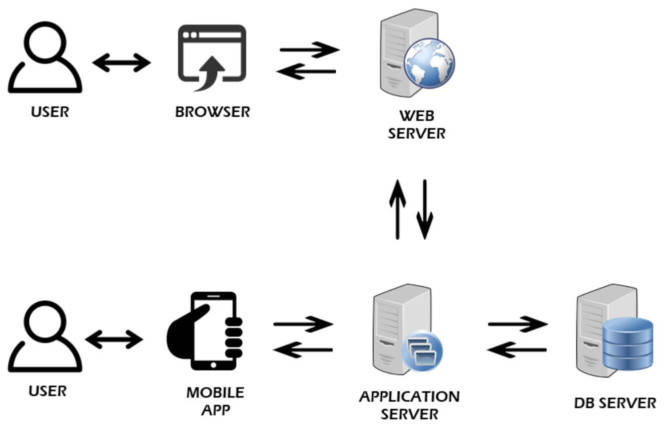

Figure 1: Proposed Architecture

2.2 High Level Components and Their Interactions

Here we have a proposed High Level Component Diagram

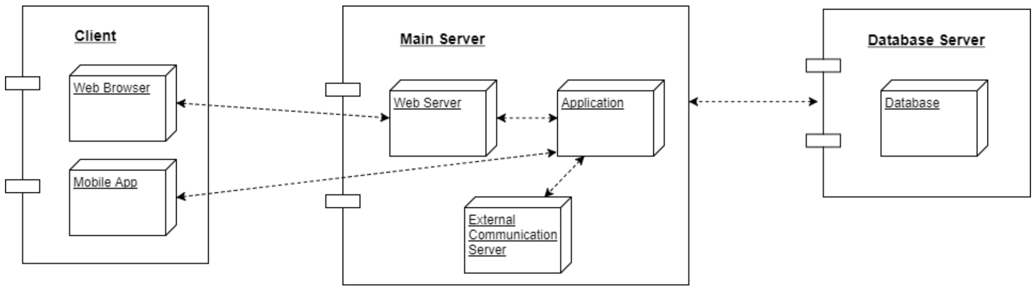

Figure 2: Components High Level

Analyzing this Diagram we can see :

- *On the Client Side :* The Mobile App of Travlendar+ for all users that uses a smartphone and has already downloaded it or the Web Browser for all the others.
- *On the Server Side :* The Web Server that, as told before, creates dynamic html and js pages, for the Web Browser, using data elaborated by the server's logic and the Application that is actually the server's logic. There is also a third component, the External Communication Server, that manages the communication of our server with the external ones, such as Google servers or Transport Service servers, in order to send and recieve informations and data from them.
- *On the Database Side :* The Database used to contain and manage all data and informations that our system needs to handle.

2.3 Component View

TEXT HERE

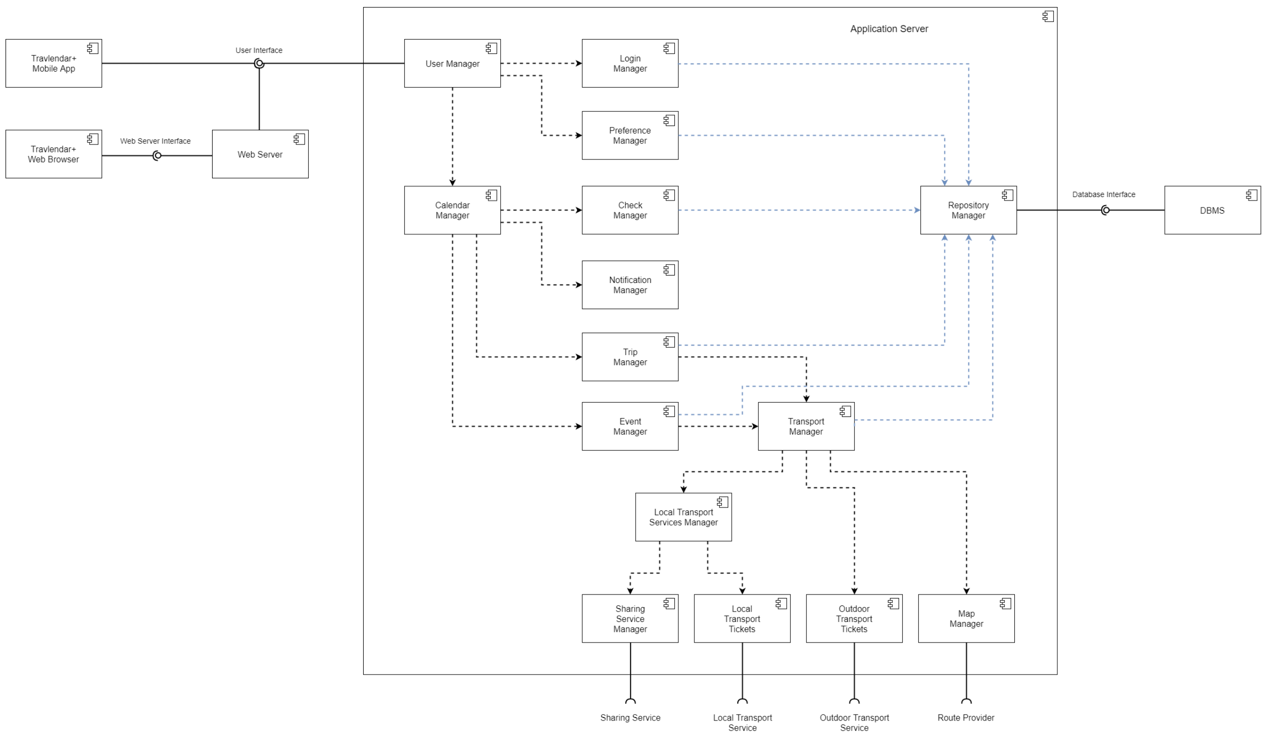

Figure 3: Component View

2.4 Deployment View

TEXT HERE

2.5 Runtime View

TEXT HERE

2.6 Selected Architectural Styles and Patterns

Our system architecture proves to be a mix between three well known architectural styles, in particular:

- *Client/Server Architectural style :*
- *Main program with subroutines architectural style*
- *Service oriented Architectural style*
- *Event Based Architectural style*

The system is divided in three main layers: Presentation Layer, Business Layer and Data Layer.

The presentation layer is both a web-based application and a mobile application. For the web application we will have a very thin client with the only aim of performing requests, through a web server, to the Business layer and receiving the HTML pages with the demanded information. On the other hand, the mobile application will not require an interface with the web server because it will have all the logic for communicating directly with the main server.

The main server will contain all the logic of the system. The main component of the logic will be the User manager. This component will perform as the Main program in the Main program with subroutines style: it will receive the requests from the clients and call the right sub-component for accomplishing the goal related to the request.

Our system, being a calendar based application, will have the main issue of managing a richly structured body of information. In our case these information will be persistently stored in multiple Data Bases, each one containing a precise schema with a precise kind of data, and they always have to reflect the true state of affairs. For this reason every client's requests that will implicate a changing data operation is preceded by a set of controls performed by defined control components designated to the database's consistency maintenance.

if the requests will pass the check steps a special component called Repository Manager is activated. This component will offer function that performs all the operation that change the DBs' state with an high level of abstraction. In fact these methods will be constituted by a composition of DBMS operation that accomplish the aim of the requests: in this way the complexity of articulated DB operation is hidden to the caller and is very simple to add new methods making this interaction highly scalable.

An essential our system business, as said in the RASD, will be to provide directions related to the precise travel means, sharing means' position and give tickets information about local and outdoor public transport. All these information will not be stored in our DBs, but they will be retrieved with API REST requests to fitting external services. For this goal we will use a SOA style. In our system we will have a component called Transport manager that will be able to distinguish the kind of the request and demand the aim of submitting it to the right external service to another precise component devoted to a certain type of external services connections. Once received the information, their manipulation will be performed internally to the business logic of the system. In these way we will have the possibility of adding new components for incoming kind of transportation, also personalized, in a very simple and scalable way.

The last style exploited is the Event based Architectural style, and it is used for the notification system always attending particular events that, if happens, triggers the component that produce a notification that is immediately sent to the right client.

2.7 Design patterns

- *MVC*: The Database server contains all the software's data and constitutes the model part. The presentation layer with the web-based application and the mobile application is the view that is released to the user. Finally the main server, that is the business logic layer, it's the control part.

- *OBSERVER:* Our software has to be able to, for instance, advising the user that he has to take a certain mean in a specific time for arriving on time, or that the registered season ticket is expiring or also the possibility of a strike or a bad meteorological condition. All these events have to be supported by a notification system.This needs will be implemented with this pattern. There will be a specific group of event listener, namely objects that extend a common abstract class. When a process that could generate a notification starts, one or more concrete references of event listeners are got by the objects involved into the process devoted to perform operations and eventually change the variables, or a set of variables of interest. The system will have as much kind of event listener as is necessary and they could be added freely for future expansion. When during such a process a variable of interest for a listener is changed some checks are performed and, under some particular conditions, a notification is created and sent to the user.

- *STRATEGY:* The filters applied on the data for showing particular results based on the user's choices, for example for advising the best means of transport for a destination, will be implemented with a strategy pattern. The class the will put in order the results will have an instance of an abstract class that will be implemented by some concrete class representing different ordering strategy. In this way the system will be able to adapt his strategy runtime and it will be very simple to add new strategy creating new classes that will implement the abstract strategy class previous mentioned.

- *COMPOSITE:* We will use this pattern in our system for performing in a cleaned and elegant way the various check that will have to be apply on the user input. There will be a abstract class Checker that will be extended by Composite checkers and Checkers. Each composite checker contains one or more checkers and both implements the method check. The class that will have to manage a specific user input will have one or more composite checkers. When a method that have to process the user input data for passing them to an external service or to another component that will insert such a data in a DB, the check method of all the compiste checker in such a class will be called and each composite checkers will recursively call his checker's check method. In this way all the check will be applied and if something is wrong, the subsequent operation will not be executed and all the possible problems linked with that will be avoided. Thanks to this pattern is possible to create new checkers/composite checkers very easily and deal with complex and composite check that will be needed in future software expansions.

- *FAÇADE:* As guideline of our implementation, for all the complex operations that will require multiple method's calls from different classes we will use a façade pattern. In this way we will able to hide a complex logic operation within a single method's call. In this way we will simplify the software maintenance for future changing needs.

3 Algorithm Design

A very delicate phase of our system is that one in which the user input is checked. This is so important because the most user's operations in Travlendar+ terminates writing data on the Data Bases and if the system letted wrong input to be written in the persistent memory, will rise big problems of inconsistency.

The operations that the user is allowed to do, work all with similar data but the checks that have to be performed are different, sometimes very complex and also some of them need a check's combination concerning single operations.

Moreover, giving a quick look on the rapid world's evolution, it sounds quite possible that new kinds of event and operations could be implemented in the system and so we needs that create a modular and scalable way of facing this problem is a issue of primary importance.

We will take on this problem using the composite pattern, that has been described item by item in the design pattern paragraph. We give downward an example of how will be implemented a checking process from the received request to the writing of the data into the DB.

CONTINUARE CON SCREEN DELL'ALGORITMO SPIEGATO