Analysis and Design Document

Student: Trif Marina

**Group: 30431**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <06/06/20> | <2.0> | <details> | <name> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

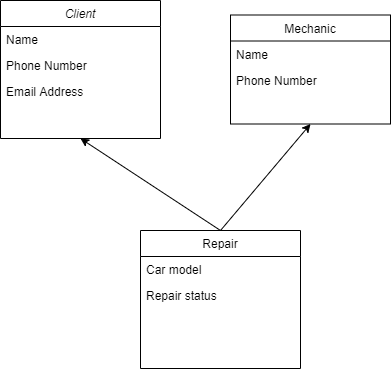
# Project Specification

The project consists a web application that deals with car service appointments. It provides a system for clients sending requests to book an appointment for a car repair to the car service and to receive approval/disapproval. It also allows users to view or modify information about the status of the repair process and provide a transparent environment for all actions in this process.

# Elaboration – Iteration 1.1

# Domain Model

The domain revolves around the car repair which contains information about the car, the status of the repair, the problem identified, the parts needed, the changes in the repair process etc. This also depends on both of the users involved in the repair: the client and the engineer. The user domain contains a separate class for each type of users, containing contact information or other essential details.



# Architectural Design

## Conceptual Architecture

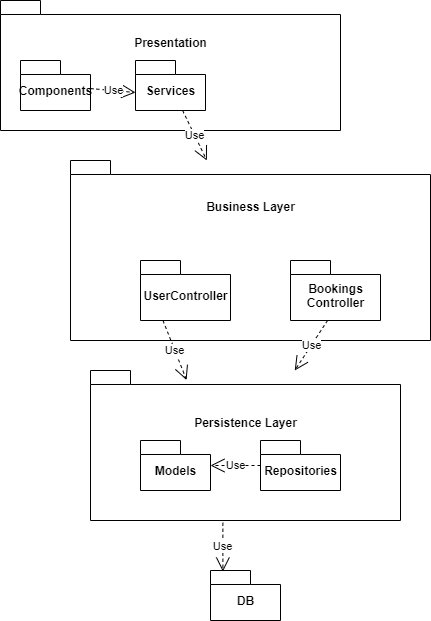
The architectural style preferred is an evolutionary one in order to easier integrate change during the development process. Since the application deals with requests, approval and status an event driven approach could be taken into consideration.

The final implementation relies on a layered architecture, specifically using the client server model. The presentation layer is made with Angular 9 and contains the components that deal with the front end data and render the views. The communication with the backend is made according to the REST API principles.

The business layer contains the controllers which receive HTTP requests and retrieve or post data with the help of repositories. The repositories are the persistent/data access layer because they are in direct contact with the application’s database context, accessing the database to retrieve or post data. The database contains tables, one for the users and one for the bookings having a one to many relationship between them.

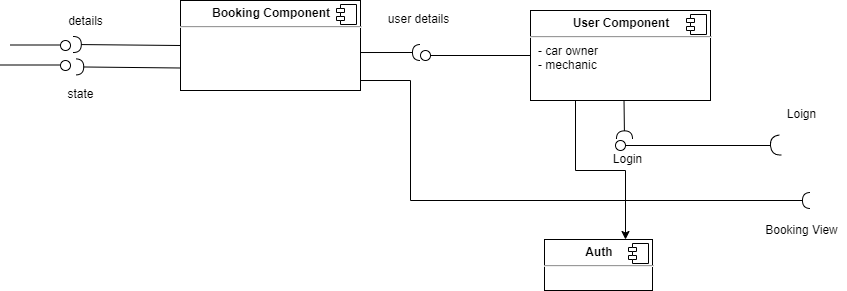
The advantages of using this architecture include ease of testability, because components belong to separate and specific layers of the application, making it more modularized. There is also a high ease of development because it is not a complex pattern to implement and its requirements are quite straightforward.

## Package Design

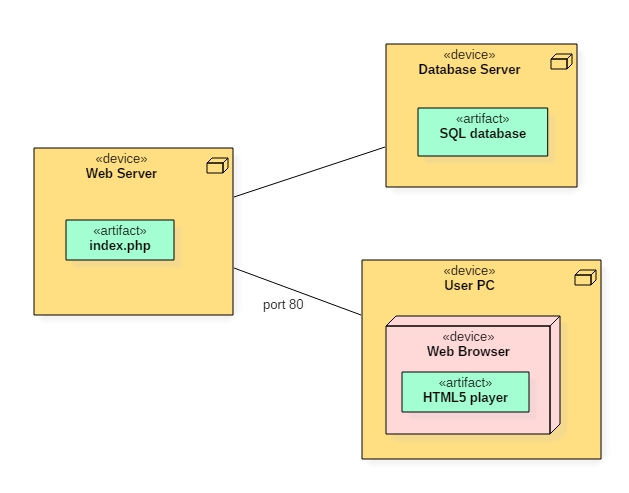


## Component and Deployment Diagrams

Component diagram



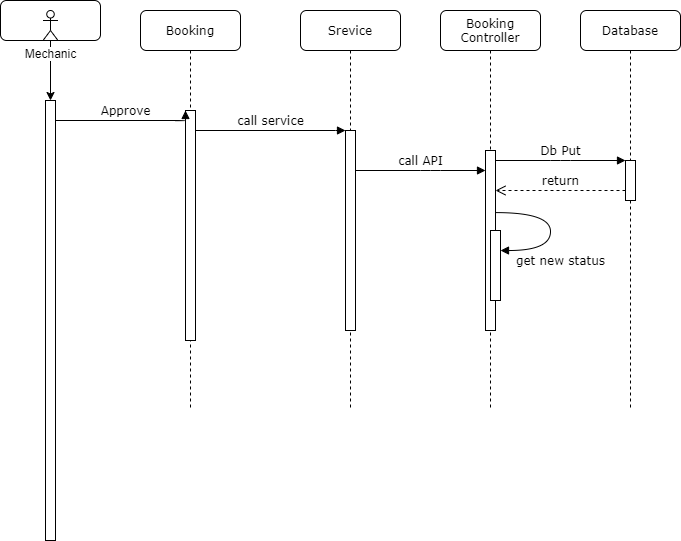
Deployment diagram

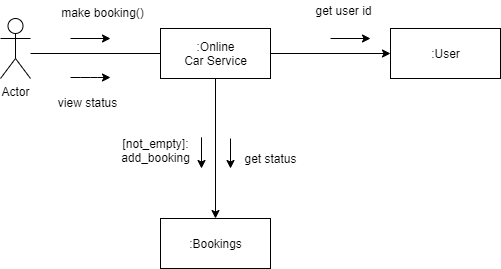


# Elaboration – Iteration 1.2

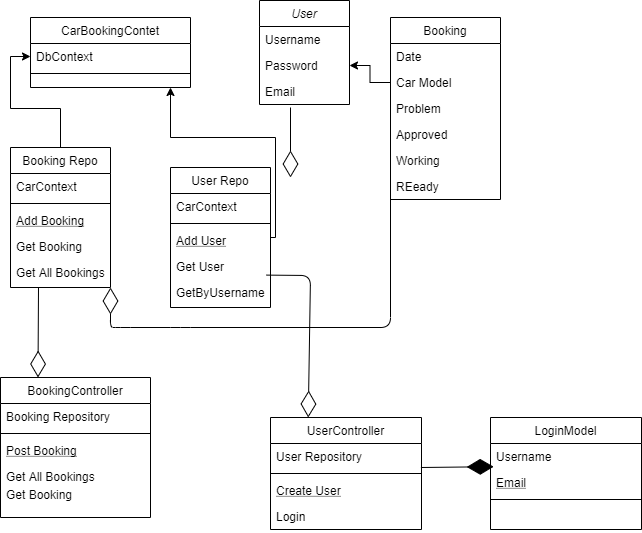
# Design Model

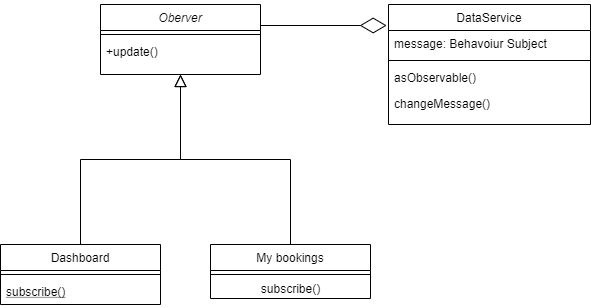
## Dynamic Behavior





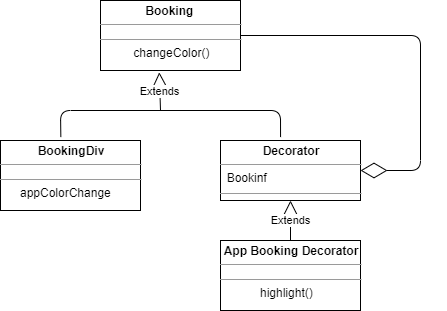
## Class Design





I used the observer design pattern for implementing the communication between two separate component with no relationship between them. This is done by creating a data service that provides a shared message as observable, while the two components that need to pass data between each other inject this service.

The motivation for using it is the need for having the user details in the ‘my-bookings’ component as well. Therefore, the dashboard subscribes the user details on the message and the my-bookings component subscribes it on the internally declared message as well, using it for selecting the bookings that were created by the user and only render those.



# Data Model

Data model for User:

public class User

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int UserId { get; set; }

public string Username { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public string Name { get; set; }

public string PhoneNumber { get; set; }

public List<Booking> Bookings { get; set; }

}

Booking model:

public class Booking

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int BookingId { get; set; }

public DateTime BookingDate { get; set; }

public string CarModel { get; set; }

public string CarProblem { get; set; }

public bool Approved { get; set; }

public bool Working { get; set; }

public bool Ready { get; set; }

public int UserId { get; set; }

public User User { get; set; }

}

Login model

public class LoginModel

{

public string Username { get; set; }

public string Password { get; set; }

}

# Unit Testing

*[Present the used testing methods and the associated test case scenarios.]*

Testing is done at the controller levels by validating data and sending exceptions when criteria is not met. For the login data flow, if the user enters the wrong credentials and attempts to log in, the server will return a Bad Request exception and notify that the user or the password is incorrect by returning this message.

Field validation is done on Angular forms on the front end by specifying whether a certain field is required or not. If it is, the form cannot be submitted unless the user provides a valid input for each field.

<input class="form-control" #***UserName***="ngModel" name="UserName" [(ngModel)]="formModel.UserName" required>

# Elaboration – Iteration 2

# Architectural Design Refinement

The development tools chosen for this application initially were the Django framework (using Python) and JavaScript for building the UI. The disadvantages of being constrained to basing an event management problem to an MVT design pattern later changed the decision to using a layered client-server architecture. The backend runs on .NET Core being implemented with Entity Framework core, storing the data using SQL Server.

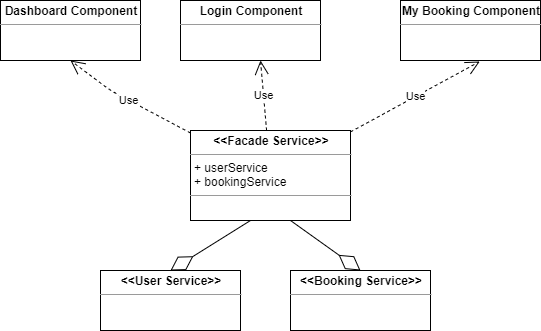
# Design Model Refinement

The main design model refinement consists in reorganizing the services used on the front end side to communicate with the backend. By applying the software design principles, the better solution was opting for a **Façade design** pattern.

This implies creating a façade service that would create all the other services, without having to inject the services in all the other components, thus creating less dependencies and the possibility to change the services without much change in the other parts of the application.

The façade service also implements the **Singleton pattern** by returning a single instance of each service. The other components will now use all the other services through the façade.

The advantages of switching to such a model



# Construction and Transition

# System Testing

Test cases will first cover all the use cases scenarios. Unit testing will be done in every module such that functionality can be ensured.

# Future improvements

Future improvements can be: receiving notifications or confirmation emails about the status of the repair, being able to pay through the application, including options for car delivery etc.

# Bibliography

[1] - <https://docs.microsoft.com/en-us/ef/core/>

[2] - <https://www.entityframeworktutorial.net/efcore/configure-many-to-many-relationship-in-ef-core.aspx>

[3] - <https://angular.io/guide/architecture>

[4] - <https://www.geeksforgeeks.org/abstract-factory-pattern/>

[5] - <https://refactoring.guru/design-patterns/abstract-factory>

[6] - <https://blog.angular-university.io/angular-jwt-authentication/>

[7] - <https://www.entityframeworktutorial.net/efcore/entity-framework-core-migration.aspx>

[8] - <https://angular.io/guide/observables>

[9] - <https://jasonwatmore.com/post/2019/07/05/angular-8-alert-toaster-notifications>

[10] - <https://fireship.io/lessons/sharing-data-between-angular-components-four-methods/>

[11] - <https://xgrommx.github.io/rx-book/content/subjects/behavior_subject/index.html>

[12] - <https://jwt.io/introduction/>

[13] - <https://angular.io/features>