Glass Booking

System Design

v1.0

01.12.2018

Abdulwahab A. Sani

Burak Sönmez

Alper Çevik

Mert Pekdüzgün

Prepared for

SE301 Software Engineering



Table of Contents

[1. Introduction 1](#_Toc433996772)

[1.1. Purpose of the System 1](#_Toc433996773)

[1.2. Design Goals 1](#_Toc433996774)

[1.3. Definitions, Acronyms, and Abbreviations 1](#_Toc433996775)

[1.4. References 1](#_Toc433996776)

[2. Current Software Architecture 1](#_Toc433996777)

[3. Proposed Software Architecture 1](#_Toc433996778)

[3.1. Overview 1](#_Toc433996779)

[3.2. System Decomposition 1](#_Toc433996780)

[3.3. Hardware Software Mapping 2](#_Toc433996781)

[3.4. Persistent Data Management 2](#_Toc433996782)

[3.5. Access Control and Security 2](#_Toc433996783)

[3.6. Global Software Control 2](#_Toc433996784)

[3.7. Boundary Conditions 2](#_Toc433996785)

[4. Subsystem Services 2](#_Toc433996786)

[5. References 2](#_Toc433996787)

SYSTEM DESIGN DOCUMENT[1]

# Introduction

We decided to implement the Model-View-Controller (MVC) architecture which is best suited for interface applications. It divides the application logic into three separate parts and it is good to use for when there is a need for different views of a system. Constraints that will affect the decision or the implementation of this architecture include access is the performance of this architecture, that is creation of the bottleneck. It will affect overall performance.

## Purpose of the System

The driving reason behind developing this system is to give restaurant goers assurance of having space/table when they go out to eat. More times than not, people go to a restaurant usually in a group and unfortunately there is no space to accommodate them all as such, our system will let them know for sure whether they will get a table or not which also saves their time and energy. It also gives the users valuable information about the restaurant and also importantly- choices. There is an expansive list that the user can choose from and also see the ratings.

## Design Goals

In design goals we try to identify which constraints can affect performance requirements and how we could try and limit these constrains. We also want a system whereby modification is easy and doesn’t affect the overall structure of our system. It is due to this consideration that we chose MVC as our software architecture. Within this structure, we can make changes to the system much easier because the Model-View-Controller are all separate entities. Also we want less dependency between subsystems, that Low coupling needs to be achieved, and also we want greater communication between components in a subsystem that is, high cohesion needs to be achieved. Another design goal we want to achieve is system failure; when a subsystem fails or cannot provide a service thus, a user command fails. We want the system to efficiently recover from this setback so user can have an error-free experience. We also want to achieve reliability so as to prevent accesses or modifications by unauthorized users. Admin functionalities should only be available to the admin, likewise manager functionalities should only be available to managers making our system safe.

## Definitions, Acronyms, and Abbreviations

Bottleneck- capacity of an application severely limited due to having a single component crowded out.

GUI- Graphical User Interface

Interface- the entity allowing for the user to communicate with the system

Resources- the entities that support the system which are limited e.g. space, data, time

MVC- Model-View-Controller

Subsystem- a self-contained system within a larger system

Status- the state of a restaurant when it is online or offline

User- either one of customer, manager, visitor or admin

## References

For access control matrix, boundary control, global software control and system architecture, we referred to the final report of “Billiard Project” developed by a set of students in Bilkent University. We compared their architectural designs with ours. Coincidentally, they also used a MVC architecture which will be good for comparison with our own.

# Current Software Architecture

For this section, the similar system we are analyzing was implemented with the client-server architecture. In this system, there is a client which is the app, then there is a server which is connected to a library known as “clarifai.ai”. The server makes API calls to this library and the clarifai.ai returns the results back to server. At the end of all this there exists a database. The database retrieves any user related data. The client sends request to the server and the server returns results. A common issue this architecture addresses is the distribution of large amounts of data.

# Proposed Software Architecture

We are going to use the MVC architecture for the new system. The Model View Controller (MVC) design pattern reuqires that an application is made up of a data model, presentation information, and control information. Each of these need to be separated into different models. MVC mostly associates to the UI / interaction layer of an application. We will still need to have a business logic layer, and some service layer and data access layer. The model contains only the pure application data, it contains no logic describing how to present the data to a user. The View displays the model’s data to the user. The view knows how to access the model’s data, but it does not know what this data means or what the user can do to change it. The Controller exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate response to the events. In most cases, the response is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically displayed in the view.

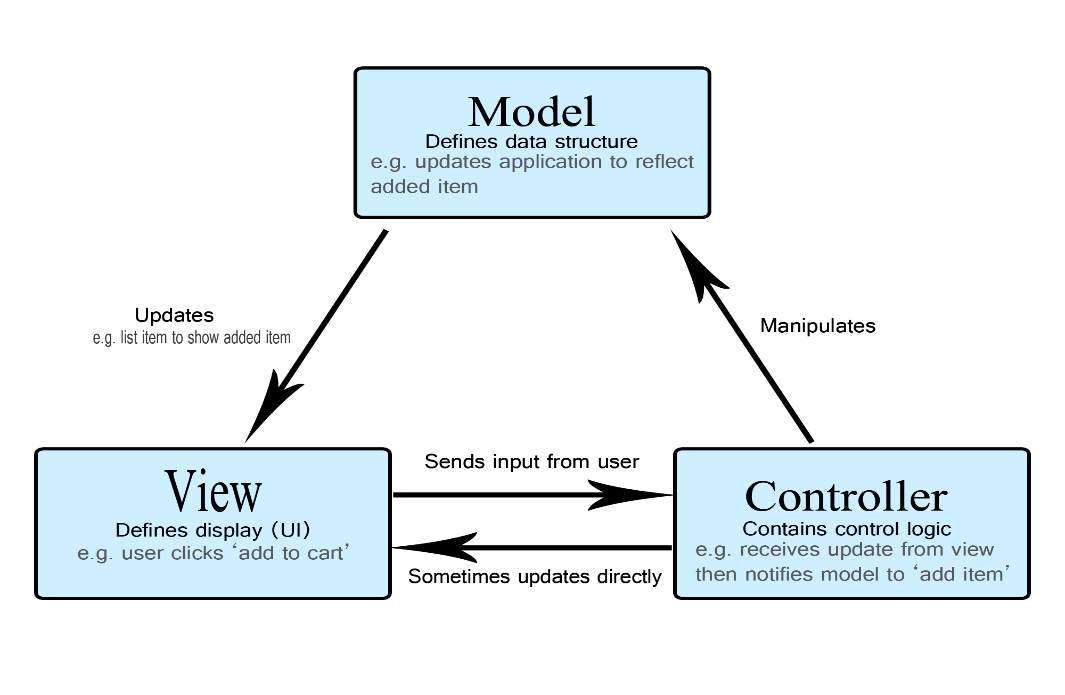
## Overview

MVC stands for Model, View and Controller. MVC separates application into three components - Model, View and Controller. It is well suited of applications with a user interface.

Model- Model maintains the data of an application, it also represents shape of the data and business logic. Model objects are responsible for retrieving and storing model state in a database.

View- The view represents the user interface. It is where commands are taken and triggers events. It views the display data to the user and also allows them to modify the data.

Controller- Controller serves as the link between the user and the system. User request is handled by the controller. Usually, user interacts with the View which will then create appropriate URL request, this request is handled by a controller.

Below is a diagram of the MVC architecture:

Advantages of using an MVC include:

* Ability to provide multiple views
* Modification doesn’t affect the entire system
* Faster development process
* Support for asynchronous technique

Disadvantages of using an MVC include:

* Increased complexity
* Performance bottleneck
* Inefficiency data access in view

## System Decomposition

We have six subsystems in our system. Each one has a unique set of features and functionalities it provides to other subsystems. We achieved this by decomposing our domain model and using the object model. Then we filtered which were most crucial in achieving the tasks our system requires. Our subsystems are as follows:

**User Management subsystem**

This subsystem manages user interactions with the system. It is responsible for allowing the user to reserve, cancel reservation, searching and rating restaurants. It is also responsible for monitoring user login and logout. Services for this subsystem include:

* reserve()
* cancelReservation()
* searchRestaurant()
* rateRestaurant()
* authentication()

**Managing Restaurant subsystem**

This subsystem relates to the modification, addition and removal of restaurant by the manager of a restaurant. It is also responsible for setting the status of the restaurant. Services provided by this subsystem include:

* addRestaurant()
* updateRestautant()
* editRestaurant()
* disableRestaurant()
* restaurantStatus()

**Restaurant Management subsystem**

This subsystem responsible for the information about restaurants and where changes to registered restaurants shall be made. It allows for retrieving information about the status of the restaurants. When a restaurant is going to be registered by a manager, this subsystem allows for setting the name, location and capacity (table and seats). Services provided by this subsystem include:

* setRestaurantLocation()
* setRestaurantName()
* capacity()
* onlineOnlineStatus()
* setRating()
* getRestaurantName()

**Reservation Handling subsystem**

This subsystem is the one responsible for controlling reservations. When a customer is reserving a restaurant and is choosing date and time, this subsystem provides the service. It also assigns a reservation ID to the customer which is a unique identifier to know which customer owns which reservation. Services provided by this subsystem include:

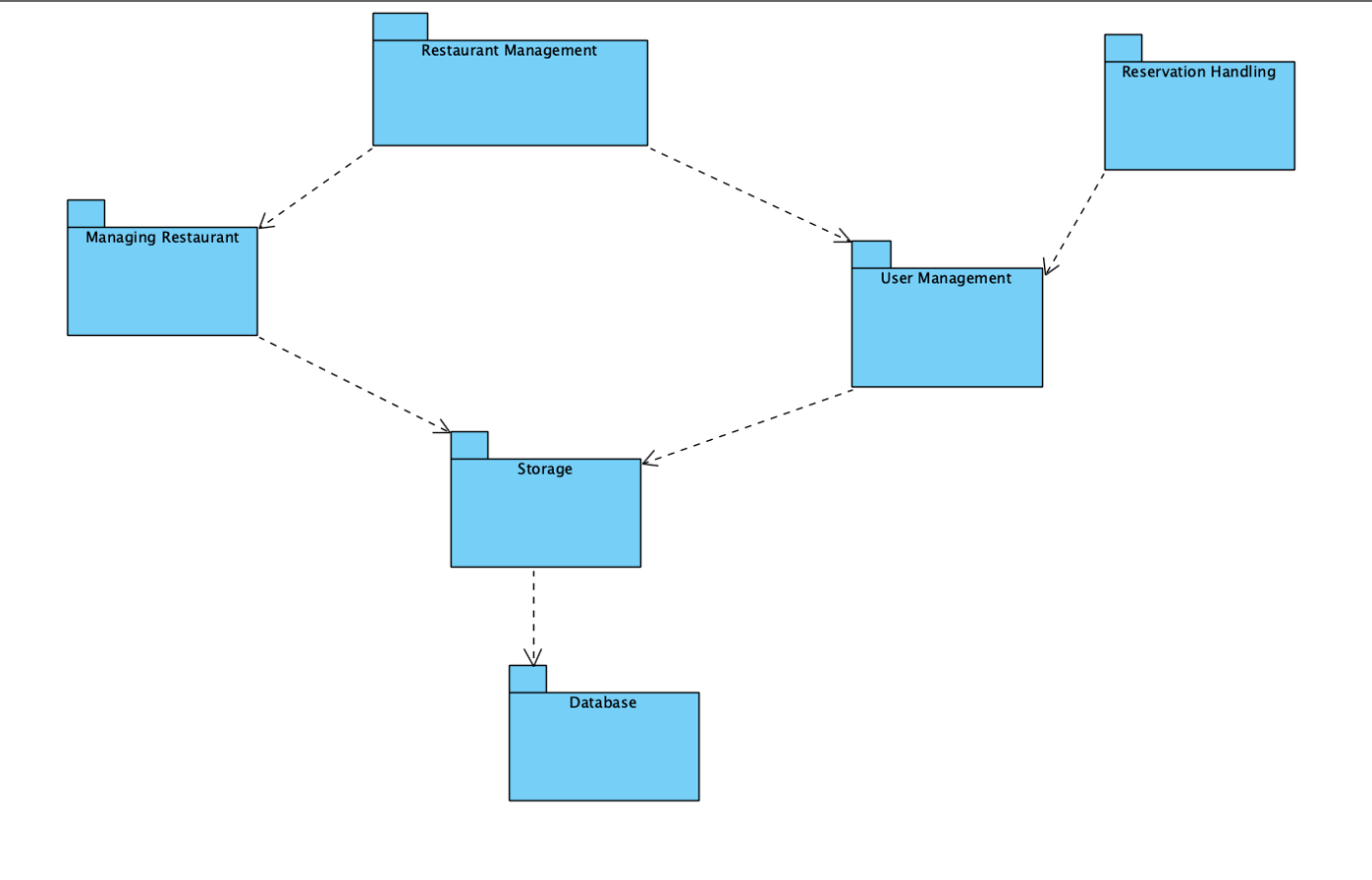
* date()
* setReservationID
* setEnable

**Database subsystem**

This subsystem is responsible for storing all our data. It is setup in such a way that a storage accesses the database and all other subsystems contact the storage for any modifications or additions to data assets. This is done in an effort to reduce dependency on the database and for security breach issues.

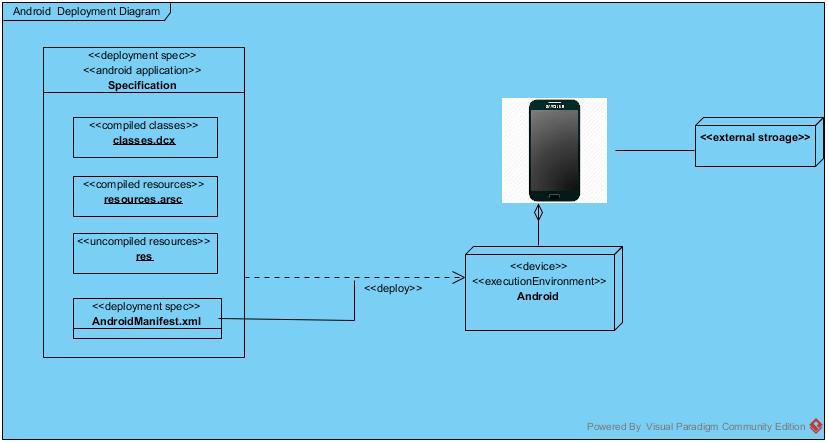
**Storage subsystem**

This subsystem is the bridge between the other subsystems and the database. If any subsystem wants to access database for data retrieval or update, the storage provides the service and then it accesses the database itself.



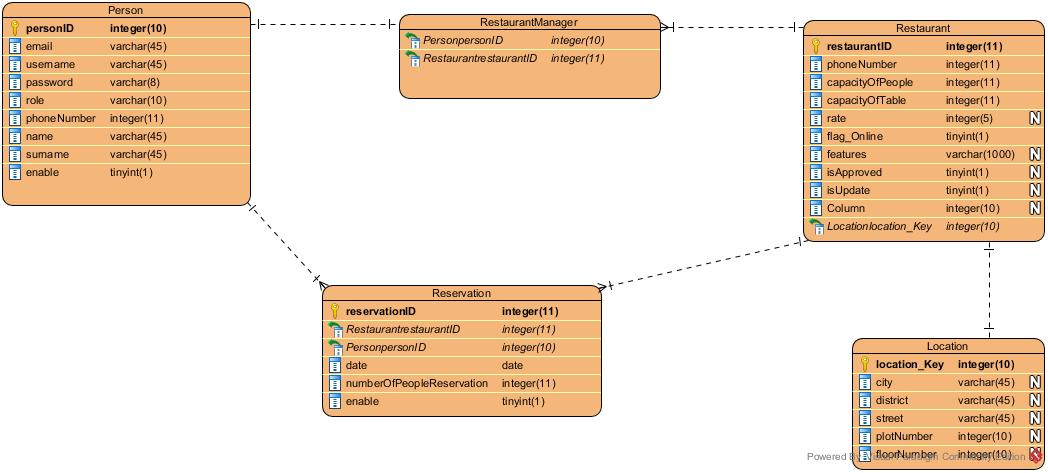
## Hardware Software Mapping

Describe how subsystems are assigned to hardware and off-the-shelf components. It also lists the issues introduced by multiple nodes and software reuse.



## Persistent Data Management

Describe the persistent data stored by the system and the data management infrastructure required for it. This section typically includes the description of **data schemes, the selection of a database, and the description of the encapsulation of the database**.



## Access Control and Security

Describe the user model of the system in terms of an access matrix. This section also describes security issues, such as the selection of an authentication mechanism, the use of encryption, and the management of keys.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Restaurant | authentication | Reservation |
| Visitor | searchRestaurant(),viewRestaurant() | SignUp() | - |
| Customer | searchRestaurant(),viewRestaurant(), chooseRestaurant(), giveRateForRestaurant() | Login(),Logout() | makeReservation(), cancelReservation(), chooseNumberOfGuest() , changeReservation(),  viewMyReservation() |
| Manager | addRestaurantRequest(), updateRestaurantInfo(), setOnlineOrOfflineRestaurant(), showReservationList(), viewMyRestaurants(), fieldsInformation(), disableRestaurantRequest() | Login(),Logout() | - |
| Admin | approveAddRestaurant(), rejectAddRestaurant(),  aproveUpdateRestaurantInfo(),  rejectUpdateRestaurantInfo() | Login(),Logout() | - |

## Global Software Control

For our software control we have decided to go with event-driven control due to the requirements and nature of our system. The sequencing of actions in our system are directed by an external factor or event generated by an actor -in our case the manager, admin, visitor and the customer- to achieve a goal. Our user interface is where the commands come from and from where events are mainly generated. This leads us to selecting explicit control and under explicit we derive centralized control and from centralized control we derive event-based control. In centralized control, one object controls everything, it can be thought of as a “spider” web. This makes change in our control structure easy to change and well suited for our system since we use a GUI. Let’s say in our system a manager wants to add a restaurant, first he has to interact with the interface, he goes to add restaurant on the menu which is invoking the addRestaurant() method and this service is provided by the managing restaurant subsystem. This subsystem communicates with the restaurant management subsystem which provides services for adding restaurant name, location, capacity and status. The restaurant management subsystem communicates with the storage subsystem which uses the services of the database to add the new restaurant with its details to the system. So the action of adding restaurant by the manager has triggered this sequence of events. Because many users access change and access data in the system and resources are shared, this creates concurrency issues some of which are: when two managers are simultaneously sending requests for adding a restaurant, they are making use of the same resources and subsystems at the same time and the requests are being delivered to the admin inbox at the same time. Or when a manager is editing restaurant profile and a user is trying to make a reservation for that restaurant. Some data information about the restaurant can change and user is accessing that data while change is occurring.

## Boundary Conditions

**Initialization**

· In the first step admin must to initialize the system because the admin manages whole system.

· User starts to use application with pressing application icon.

· A login screen shows up, there are two options one is to login as a previously created profile and the other is to create a profile.

· At this point the data that will be accessed at start-up time is user login data from storage and from database

· User chooses to login and system retrieves the username and passwords to check with what user entered. If it is matches, user directed to main page of customer or manager or admin.

· If create a profile is chosen, system must access usernames to check for the duplication in usernames. User must choose his/her role there is two options one of manager and the other one customer. If what user entered as username is not the same with the ones in database and the whole blank are filled, user directed to main page of the application for manager or user.

· If user chooses to go as visitor, s/he is directed to explore page of the application.

**Termination**

· User has a chance to exit the program any time s/he wants. If s/he wants to terminate during some action, s/he is lost all changes which her/his do.

· User has a chance to quit action while s/he is doing something like making reservation or adding new restaurant (return to main menu).

· The information about the system that are kept in memory is cleaned up so that memory should be emptied and so the system could start from the beginning next time.

· Updates to the database are communicated through the storage. All subsystems communicate to the storage when there is a request for data access, update or retrieval.

· In the case that a subsystem terminates, other subsytems get a notification or alert about this termination. The execution fails and restarts or is cancelled completely.

· Individual subsystems are not allowed to terminate because of dependency of other subsystems on them.

**Failure**

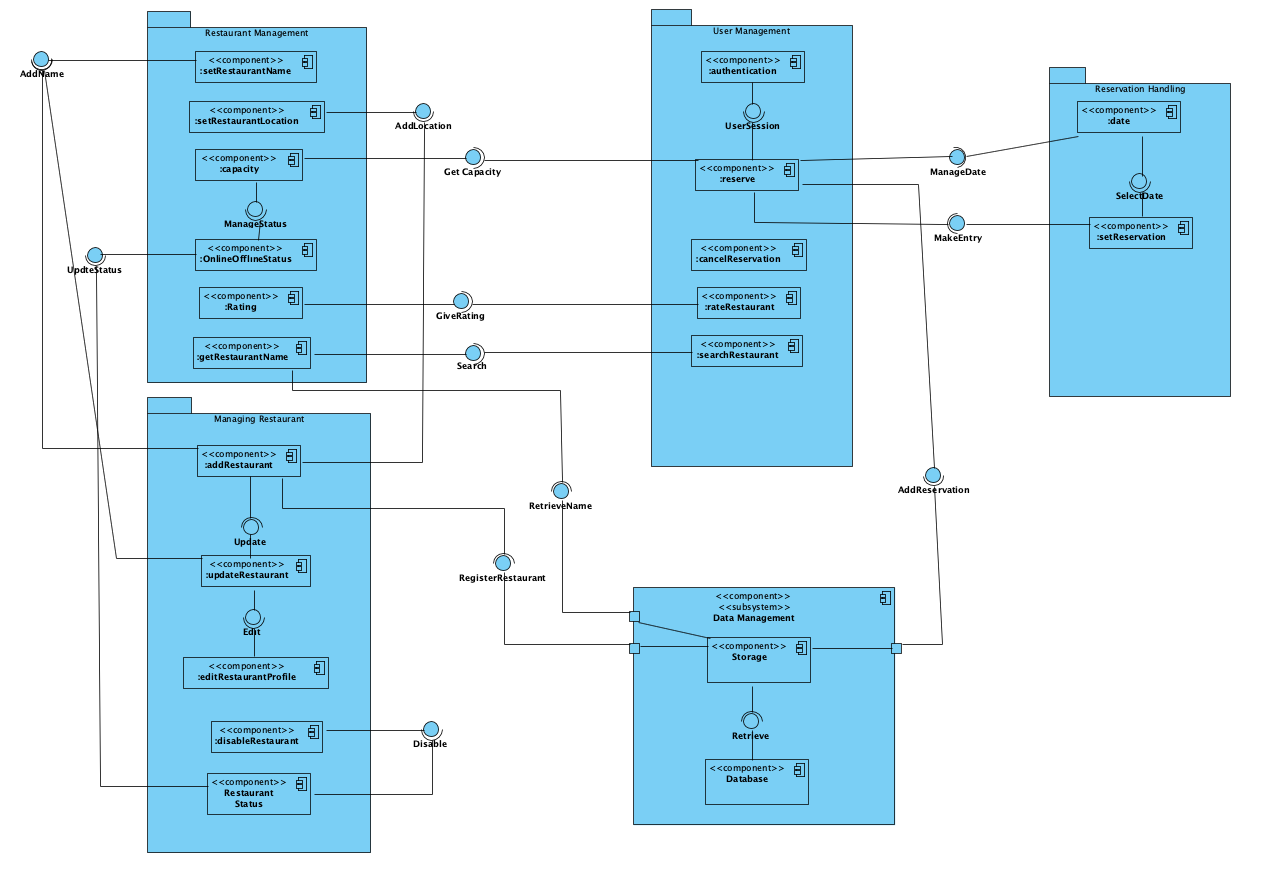
· If database connection is lost, there is a reconnect mechanism which attempts to reestablish the connection to the database.

· Checkpoint system can be applied to prevent database connection problem. System is connected to database and required data is written regularly. If a problem occurs, system will be returned to its previous errorless state.

· System failure can occur due to a subsystem terminating. To prevent this individual subsystems are not allowed to terminate. And in the case that it terminates forcefully, a notification is sent to other subsystems and this issue can be resolved easier.

# Subsystem Services

Describe the **services provided by each subsystem**. Although this section is usually empty or incomplete in the first versions of the SDD, this section serves as a reference for teams for the boundaries between their subsystems. The interface of each subsystem is derived from this section and detailed in the Object Design Document.



# References

1. Bruegge B. & Dutoit A.H.. (2010). *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 3rd ed.
2. <http://www.tutorialsteacher.com/mvc/mvc-architecture>
3. <https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/>
4. https://www.geeksforgeeks.org/mvc-design-pattern/
5. Enes Taylan, Huseyin Guler, Alperen Eraslan, Omer Durmus (2009). *Billiard Project, Object Oriented Software Engineering.*
6. Baekchun Kim, Jiwon Shin, Hailey Lee, Sung Woo Park, Sienna Schmid (2017). *Not Trash.*