Design Pattern

The design pattern our system follows and suits our implementation the most is the Model View Controller (MVC). The MVC pattern divides our software part into three main components:

* **Model**: Includes the classes that represents stored data. The Model can only see and interact with the Controller.
* **View**: Includes the interface that the user can view and interact with
* **Controller**: Includes the system logic and interacts with both the View and Model components by retrieving inputted data from user and passing it to the model after processing any required logic with it.

The MVC structure explains the main steps our system follows. The user interacts with the user interface of our system (application or website) and whenever the user requests to input or output a certain data from the view, the view sends that request to the controller along with any inputted data. Afterwards, the controller processes the request and retrieves the needed data from the model, and then sends the appropriate output (results) back to the view to be shown to the user.

View

Controller

Model

Advantage of using this pattern:

* Separation of Concerns:

The MVC pattern separated our system into components that are independent from each other, this is called “separation of concerns”. A failure can easily be found because of the components independency, and an edit can be done on any of the components without disturbing other components.

MVC allowed us to separate our logic from our presentation layer. This “separation of concerns” allowed us to quickly find and edit portions of our code whenever needed for example if a miss or a fail happened in one of the algorithms, we can easily re-write it without having to interrupt the user interface.

* Re-usability and Flexibility:

The MVC pattern can increase the scalability of our system, this is because a function can easily be added or changed in a component without changing other components. Hence, our system is easily flexible to scalability. Additionally, different component of our system can be reused in other systems to give the same service.

Due to the components independency, the UI (View component) can be changed without editing other components of the system. Additionally,

The UI can be completely changed without touching the model in any way. Additionally, using MVC might help in increasing the scalability of the application(How?).

* High Cohesion and Low Coupling:

The MVC model automatically makes the system have high cohesion and low coupling. This is because each component in the system must be related in functionality and serve the system with a specific job, which makes the functions in one component highly cohesive. Also, the different components in the system are lowly coupled, meaning changing one component, whether a major or minor change, does not interrupt the work of other components.

Applying such principle making our system high cohesion in the way that all functions in a single class are functionally related and preform methods that complete a specific job. Also, it makes our system low coupling which mean changing something major in one class should not affect the others.

We have used Model View Controller (MVC) pattern in our system. We chose to implement MVC pattern because it is more suitable for what we are trying to implement. The MVC pattern splits code into one of three MVC components.

* **Model**: which are the classes that represent the data to be stored. It interacts with the controller and has no knowledge of the interface. For example, Reservation, User, Spot, etc.
* **View**: includes the user interface and what the user can interact with contains what is visible to the user (or allows the user to interact with elements on the screen). For example, all the xml files in the application.
* **Controller**: interacts with the view and the model by responding to user input and retrieving data the user requests. It also manages the application logic. For example, LoginActivity, SignupActivity, ReservationActivity, ShowReservationActivity, etc.

A simple way to think about our application structure is the following:

The user interacts with the view (user interface) then the Controller handles the user input and transfers the information to the model. The controller does all the computation and processing between the model and taking the inputs and showing the output from and to the view.