**SMS Application Breakdown**

For this application, we are following Spring MVC pattern meaning which we divide our application into Model controller and view. We have dedicated packages in which we write our code.

**Repository Layer** - This layer is essential because it helps us implement the JPA repository, which provides ready-made methods to handle CRUD (Create, Read, Update, Delete) operations with the underlying database. By encapsulating the data access logic within this layer, it ensures separation of concerns, allowing other layers to focus on their respective responsibilities. The Repository Layer acts as a bridge between the application and the database, promoting modularity and maintainability.

Our repository package contains all the classes which implements JPA Repository interface. If our scenario requires us to handle any customised queries, then we are doing that here.

**Service Layer** - The Service Layer plays a crucial role as a mediator between the Repository Layer and the Controller Layer. By defining interfaces and implementing methods in respective classes, this layer encapsulates the **business logic** and orchestrates the data flow within the application. This separation allows us to maintain a clean separation of concerns, making the codebase more modular and easier to test. It also promotes code reuse by enabling different controllers to access the same business logic without repeating it.

Our Service package contains all the interfaces and the classes which implements them. All the interfaces contain the list of methods which can be exposed and be used in the controller layer. It also has classes which implements these interfaces and performs business logic wherever it is required. For ex: **UserLoginImpl** class contains required business logic for User validations.

**Controller Layer** - The Controller Layer is vital for managing the incoming requests and routing them to the appropriate service methods. It acts as the entry point to the application and handles the interaction with external clients. By defining API endpoints and their respective paths, this layer enables users to access the application's functionality through well-defined URLs. Additionally, the Controller Layer handles input validation and mapping incoming data to appropriate request objects, ensuring the proper flow of data to the Service Layer.

**This is the starting point for our application**. It fetches the Service methods where the business logic is present. (Service layer refers to repository layer within). Incoming request’s validation is being handled here.

**Entities Layer** - The Entities Layer represents the domain model of the application, containing Plain Old Java Objects (POJOs) that map to database tables. These entities encapsulate the application's data and state, helping maintain consistency and integrity of the data. By defining the structure of the domain objects, the Entities Layer acts as the foundation for the Repository Layer, facilitating data storage and retrieval. Separating entities from other layers promotes clean architecture, making the application easier to understand and maintain.

Our entity layer has all the elements and its getters and setters’ methods for us to freely use them in respective service or controller layers. Elements here are declared as private and are being fetched using getters/setters. This follows data encapsulation.

**DTO Layer** - The DTO (Data Transfer Object) Layer is essential for handling the structure of data being exchanged between the application and external clients through REST APIs. It provides a way to represent data in a format tailored to the specific needs of the clients, decoupling it from the internal domain model. By creating DTO classes, we ensure that clients receive only the required data, improving efficiency and reducing unnecessary data transfer.

In our application, we have created classes either to process the request received from the front-end or post a response back via DTO classes. This basically helps us transfer data between objects in relative class structure that is required.

Most of our application’s interfaces and methods are Public because it enables controlled access to functionality, enforcing encapsulation and protecting sensitive data. Extensibility and flexibility are achieved through public interfaces, as external modules can implement them and integrate seamlessly into the application. And we have limited the elements to private wherever it is required. But for most of it, Spring MVC requires it to be public to enable communication between the layers.