**MVC Layered**: The MVC layer (Pattern) is used in software engineering to separate the application logic from the user interface. The MVC layered has 3 layer:

**MODEL:** The model defines the business logic of the application

**VIEW:** Defines the presentation layer of the application. **(UI will come in vew)**

**CONTROLLER:** Manages the flow of the application

**Guest House Booking App (Project):**

**The New data structure I am going to follow from now:**

**backend/**

**└── src/**

**├── main/**

**│ ├── java/**

**│ │ └── com/**

**│ │ └── guesthouse/**

**│ │ ├── controller/**

**│ │ │ ├── admin/**

**│ │ │ │ ├── AdminGuestHouseController.java // Class**

**│ │ │ │ ├── AdminBookingController.java // Class**

**│ │ │ │ └── AdminUserController.java // Class**

**│ │ │ └── user/**

**│ │ │ ├── UserGuestHouseController.java // Class**

**│ │ │ ├── UserBookingController.java // Class**

**│ │ │ └── AuthController.java // Class (Handles login/register)**

**│ │ │**

**│ │ ├── dto/ // DTO Classes**

**│ │ │ ├── BookingRequestDTO.java**

**│ │ │ ├── RoomDTO.java**

**│ │ │ ├── UserRegistrationDTO.java**

**│ │ │ ├── UserLoginDTO.java**

**│ │ │ └── AuthResponseDTO.java**

**│ │ │**

**│ │ ├── model/**

**│ │ │ ├── entity/**

**│ │ │ │ ├── GuestHouse.java // Entity class**

**│ │ │ │ ├── Room.java**

**│ │ │ │ ├── Bed.java**

**│ │ │ │ ├── Booking.java**

**│ │ │ │ └── User.java**

**│ │ │ └── enums/**

**│ │ │ ├── Role.java // Enum**

**│ │ │ └── BookingStatus.java // Enum**

**│ │ │**

**│ │ ├── repository/**

**│ │ │ ├── GuestHouseRepository.java // Interface**

**│ │ │ ├── RoomRepository.java // Interface**

**│ │ │ ├── BedRepository.java // Interface**

**│ │ │ ├── UserRepository.java // Interface**

**│ │ │ ├── BookingRepository.java // Interface**

**│ │ │ └── RoleRepository.java // Interface**

**│ │ │**

**│ │ ├── service/**

**│ │ │ ├── AdminService.java // Interface**

**│ │ │ ├── UserService.java // Interface**

**│ │ │ └── AuthService.java // Interface**

**│ │ │**

**│ │ ├── service/impl/**

**│ │ │ ├── AdminServiceImpl.java // Implements AdminService**

**│ │ │ ├── UserServiceImpl.java // Implements UserService**

**│ │ │ └── AuthServiceImpl.java // Implements AuthService**

**│ │ │**

**│ │ ├── config/**

**│ │ │ ├── SecurityConfig.java // Class**

**│ │ │ ├── JwtAuthenticationFilter.java // Class**

**│ │ │ ├── JwtProvider.java // Class**

**│ │ │ ├── CORSConfig.java // Class**

**│ │ │ └── SwaggerConfig.java // Class**

**│ │ │**

**│ │ ├── exception/**

**│ │ │ ├── GlobalExceptionHandler.java // Class**

**│ │ │ ├── CustomException.java // Class**

**│ │ │ ├── ResourceNotFoundException.java // Class**

**│ │ │ └── BookingConflictException.java // Class**

**│ │ │**

**│ │ ├── util/**

**│ │ │ ├── DateUtils.java // Class**

**│ │ │ └── JwtUtils.java // Class**

**│ │ │**

**│ │ └── GuesthouseBookingApplication.java // Main Class**

**│ └── resources/**

**│ ├── application.properties**

**│ ├── static/**

**│ ├── templates/**

**│ └── banner.txt**

**└── test/**

**└── java/**

**└── com/**

**└── guesthouse/**

**├── controller/ // Test classes for controllers**

**├── service/ // Test classes for services**

**├── repository/ // Test classes for repositories**

**└── integration/ // Full flow test cases**

**└── pom.xml**

[**The structure chatgpt link is here**](https://chatgpt.com/c/68183da9-0930-8002-8fa7-c720dde352bd)

| **📁 Your Folder/File** | **🧩 Purpose (One Line Explanation)** |
| --- | --- |
| controller/admin/ | Handles HTTP requests from admin users (CRUDs on bookings, users, guest houses). |
| controller/user/ | Handles HTTP requests from normal users (view, book rooms, authentication). |
| dto/ | Data Transfer Objects to pass only needed data between frontend ↔ backend or layers internally. |
| model/entity/ | Core database entities (mapped to DB tables using JPA like User, Booking, Room). |
| model/enums/ | Enum classes for constants like Role or BookingStatus. |
| repository/ | Interfaces to access database using Spring Data JPA (findById, save, etc.). |
| service/ | Business logic contracts (interfaces) that define what services must do. |
| service/impl/ | Implements business logic methods (e.g., actual login, booking logic). |
| config/ | Application configurations like security (JWT, CORS) and Swagger docs. |
| exception/ | Custom error classes and global exception handling for clean error responses. |
| util/ | Utility/helper classes reused across project (like DateUtils, JWT functions). |
| GuesthouseBookingApplication.java | Main Spring Boot class with @SpringBootApplication to start the app. |
| resources/application.properties | Configuration file for DB, port, JWT secret, etc. |
| resources/static/ | Static files like images, JS, CSS (if any). |
| resources/templates/ | HTML templates if you're using Thymeleaf (optional in full REST API). |
| resources/banner.txt | Optional branding message shown when app starts in console. |
| test/java/com/guesthouse/... | Unit and integration test classes for controller, service, etc. |
| pom.xml | Maven configuration file: declares dependencies, plugins, project settings. |

**🧭 Step-by-Step Code Flow (What to Create First)**

**🔹 1. Model Layer (Entities)**

✅ **Start here — define the structure of your database.**

Create classes in model/:

* GuestHouse.java
* Room.java
* Bed.java
* User.java
* Booking.java
* Role.java

These are JPA entities annotated with @Entity, defining your DB tables.

**🔹 2. Repository Layer**

✅ **Next — create interfaces to access DB for each model.**

Create interfaces in repository/:

* GuestHouseRepository extends JpaRepository<GuestHouse, Long>
* RoomRepository
* BedRepository
* UserRepository
* BookingRepository
* RoleRepository

These are simple interfaces. Spring Data JPA will auto-implement them.

**🔹 3. Service Layer (Interfaces)**

✅ **Now — define your app logic as service interfaces.**

In service/:

* AdminService (e.g. manage guesthouses, users, bookings)
* UserService (e.g. book rooms, view availability)

Use interfaces to keep your logic decoupled and testable.

**🔹 4. Service Implementation Layer**

✅ **Now — implement the logic in impl/ using repositories.**

In service/impl/:

* AdminServiceImpl implements AdminService
* UserServiceImpl implements UserService

This is where you do CRUD operations and business logic using repositories.

**🔹 5. Controller Layer**

✅ **Now — expose APIs via REST controllers.**

In controller/:

* admin/
  + AdminGuestHouseController.java
  + AdminUserController.java
  + AdminBookingController.java
* user/
  + UserGuestHouseController.java
  + UserBookingController.java
  + AuthController.java

These classes use @RestController, map to URLs, and call service methods.

**🔹 6. Config Layer**

✅ **Now — configure app-wide behaviors.**

In config/:

* SecurityConfig.java (Spring Security setup)
* JwtAuthenticationFilter.java (filter for JWT token)
* JwtProvider.java (generate/validate tokens)
* CORSConfig.java

Handle login, roles, JWT auth, and CORS here.

**🔹 7. Exception Layer**

✅ **Now — handle errors properly.**

In exception/:

* GlobalExceptionHandler.java → uses @ControllerAdvice
* ResourceNotFoundException.java or custom exceptions

**🔹 8. Application Entry Point**

* GuesthouseBookingApplication.java

This just runs the whole Spring Boot app.

**🔹 9. Testing Layer**

✅ **Finally — test your code.**

In test/java/com/guesthouse/:

* controller/ → Test API endpoints
* service/ → Test service logic
* repository/ → Optional: custom query tests
* integration/ → For end-to-end tests

**🧩 Optional but Good:**

* **DTOs** (for clean API input/output)
* **Mapper Classes** (MapStruct or manual)
* **Utility Classes** (e.g., DateHelper)

**✅ TL;DR: Flow Summary**

1. **Model →**
2. **Repository →**
3. **Service Interface →**
4. **Service Impl →**
5. **Controller →**
6. **Config →**
7. **Exception →**
8. **Application Main →**
9. **Test**

[For next steps click on this link:](https://chatgpt.com/share/680f365c-e540-8002-b7e7-579f3c7a7bfe)

**What type of Dependencies should I add in my Spring-boot Project:**

 Spring Web

 Spring Data JPA(you have to manually add this)

 Spring Security

 Spring Boot DevTools (optional, for live reload during dev)

 Spring Boot Starter Mail

 Validation

 MySQL Driver (or PostgreSQL if using PostgreSQL)

 Lombok

 Spring Session (optional, for session management)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Day 1(29-4-25): Started with model:**

**Annotations and there meaning:**

**@Entity:** Marks a Java class as a **database entity** (a table will be created for it).

**@Id:** Marks a field as the **primary key** (unique identifier) of the entity.

**@GeneratedValue:** Automatically **generates values** for the primary key (like auto-increment in database).

**@ManyToOne:** Defines a **many-to-one relationship** (e.g., many bookings belong to one user).

**@OneToMany:** Defines a **one-to-many relationship** (e.g., one guesthouse has many rooms).

**@JoinColumn:** Specifies the **foreign key column** name for relationships (@ManyToOne).

**@OneToOne:** Defines a **one-to-one relationship** between two entities (rare but useful sometimes).

**@ManyToMany:** Defines a **many-to-many relationship** (e.g., users and roles — a user can have many roles and vice versa).

**@Table(name = "table\_name"):** (Optional) Specifies a **custom table name** if you don't want default (class name).

**@Column(name = "column\_name"):** (Optional) Specifies a **custom column name** for a field.

**@CascadeType (used inside relationships):** Defines how **related entities** should be saved/deleted automatically.

**@FetchType (also used inside relationships):** Controls whether related data is **loaded eagerly or lazily** (performance tuning).

**Some main Annotation we are using right now:**

**@Autowired** is a Spring annotation used to automatically inject the required bean (object) into another class.

**@RestController**

* Combines @Controller + @ResponseBody.
* Tells Spring: “This class handles HTTP requests and returns data, usually in JSON format.”
* Used on top of controller classes.

🔸 **Use Case**: For any API class that sends responses like JSON/XML (not webpages)

**🔹 @RequestMapping("/base-path")**

* Sets a **base URL** for all the request methods in the class.
* Can also be used with methods (but most use @GetMapping etc. instead).

🔸 **Use Case**: If your controller handles bookings, you might use @RequestMapping("/api/bookings").

**🔹 @GetMapping("/path")**

* Maps to **GET** requests.
* Used to **retrieve or fetch** data from the server.

🔸 **Use Case**: Viewing a booking, user details, list of rooms, etc.

**🔹 @PostMapping("/path")**

* Maps to **POST** requests.
* Used to **create new** data or send form/input data.

🔸 **Use Case**: Adding a new booking, registering a new user, etc.

**🔹 @PutMapping("/path")**

* Maps to **PUT** requests.
* Used to **update an existing** full object (i.e., all fields must be provided).

🔸 **Use Case**: Editing a booking with all fields like guest name, date, room, etc.

**🔹 @PatchMapping("/path")**

* Maps to **PATCH** requests.
* Used to **partially update** data (only the fields that need to be changed).

🔸 **Use Case**: Updating only the check-out date of a booking.

**🔹 @DeleteMapping("/path")**

* Maps to **DELETE** requests.
* Used to **remove** a resource (like delete booking/user).

🔸 **Use Case**: Deleting a guest’s booking by ID.

**I am going to code in this way. Start in this order:**

1. Role
2. User
3. GuestHouse
4. Room
5. Bed
6. Booking

**Day2. After completing everything in model now we have to create repository**

|  |  |
| --- | --- |
| GuestHouseRepository | Access GuestHouse data (usually just one static record in your case). |

|  |  |
| --- | --- |
| RoomRepository | Manage rooms in the guest house. |

|  |  |
| --- | --- |
| BedRepository | Manage beds inside rooms. |

|  |  |
| --- | --- |
| UserRepository | Handle user data and login via email (findByEmail). |

|  |  |
| --- | --- |
| BookingRepository | Work with booking records (create/view bookings). |

|  |  |
| --- | --- |
| RoleRepository | Handle roles like "ADMIN" and "USER" (findByName). |

**✅ Phase 1: Set Up Core Backend Functionality**

**🔹 Step 1: Start with the model/ package**

Start by defining the main database entities:

* User.java (with roles and login details)
* GuestHouse.java (name, address, description)
* Room.java (linked to GuestHouse)
* Bed.java (linked to Room)
* Booking.java (linked to Room and User)
* Role.java (USER / ADMIN enum or entity)

✅ Make sure every class uses @Entity, has @Id, and proper relationships like @OneToMany, @ManyToOne, etc.

**Step 2: Set up Repositories**

In the repository/ package, create interfaces for each model:

public interface GuestHouseRepository extends JpaRepository<GuestHouse, Long> {}

Same for RoomRepository, UserRepository, etc.

**🔹 Step 3: Implement Authentication**

Set up:

* User.java with username/email + password
* Role.java as enum or entity (USER, ADMIN)
* AuthController.java with login/register endpoints
* SecurityConfig.java with JWT + password encoding

✅ Test Register & Login API using Postman.

**✅ Phase 2: Add Core Business Logic**

**🔹 Step 4: Create Services**

In service/impl/, implement logic:

* UserServiceImpl.java → user registration, booking history
* AdminServiceImpl.java → CRUD for guest house, rooms

Write interfaces in service/ like UserService.java, etc.

**🔹 Step 5: Create Controllers**

Split between:

* admin/ → Guest house/room management (AdminGuestHouseController.java)
* user/ → Viewing guest houses, booking rooms (UserGuestHouseController.java, UserBookingController.java)

Use @RestController and map routes like /api/admin/guesthouses, /api/user/bookings, etc.

**🔹 Step 6: Add Booking Logic**

In Booking.java, handle:

* Room ID
* Dates
* Total cost
* Linked to a user

Then build:

* UserBookingController.java for making/viewing bookings
* AdminBookingController.java for managing them

**✅ Phase 3: Connect Frontend (React.js)**

Once your backend is ready and tested with Postman:

**🔹 Step 7: Start building pages in React**

Pages to build:

* Login/Register
* Browse Guest Houses (User)
* Guest House Detail with Room Info
* Booking Form
* Admin Dashboard (Add/edit guest house & rooms)

Use axios to call Spring Boot APIs.

**Bonus: File Upload (Locally)**

If you're letting admins upload guest house images:

* Use MultipartFile in Spring Boot
* Save files to /uploads/ and store path in DB
* Serve static files using Spring WebMvcConfigurer

**Day 2 continue : I have created a service and service impl**

**In Service we will create an interface and in service impl we will create a class which implements that interface.**

**But why we are doing this?**

We are doing this for providing the privacy of the our database because we don’t want to share our data directly on the frontend

**Controller ‘s Annotations:(Controller package ma su aavse)**

**@RestController,@RequestMapping,@GetMapping,@PostMapping,@PutMapping,@DeleteMapping,**

**@PatchMapping-**It will used for partially updating data

FOR Front-End click on this link: [LINK](https://chatgpt.com/c/680b6d12-e898-8002-9125-d0ed8bfe4d26)