Your project structure already follows a clean and standard Java (especially Spring Boot) layout. Here's the **recommended logical sequence** you should follow **after models**:

**✅ Recommended Folder Sequence (with purpose):**

1. **models** → Define entity classes (usually annotated with @Entity)
2. **repository** → Interface for DB operations (extends JpaRepository, CrudRepository)
3. **DTO** → Data Transfer Objects for sending/receiving custom data
4. **services** → Business logic layer, contains service classes annotated with @Service
5. **controller** → API endpoints, annotated with @RestController / @Controller

**Optional But Useful (already present in your case):**

| **Folder** | **Purpose** |
| --- | --- |
| config | Configuration classes (@Configuration, security configs, etc.) |
| aspect | Aspect-Oriented Programming (AOP) logic, like logging, transactions |
| demo | Sample testing classes or dummy code (optional, not mandatory) |

**🧭 Suggested Full Sequence:**

com.example

│

├── models --> @Entity classes (tables)

├── repository --> JpaRepository interfaces

├── DTO --> Data transfer classes (for request/response)

├── services --> Business logic layer (@Service)

├── controller --> API endpoints (@RestController)

├── config --> Configuration classes (SecurityConfig, SwaggerConfig)

├── aspect --> AOP logic (logging, exception handling)

├── demo --> Temporary/demo code (optional)

**✅ Next Steps You Can Do:**

* If you haven't already:
  + Create Service interfaces and their Impl classes in services.
  + Wire them in your controllers using @Autowired or constructor injection.
  + Map DTO <--> Entity using MapStruct or manually.

Let me know if you want a full example of the flow from controller → service → repository → model.

Here is a **box diagram** (ASCII-style) of your project structure showing **how components interact in a typical Spring Boot application**, along with the **overall flow**:

**🔷 Box Diagram: Component Interaction**

[ Client / Frontend ]

│

▼

┌────────────────────┐

│ Controller │ (@RestController)

└────────────────────┘

│

▼

┌────────────────────┐

│ Service │ (@Service)

└────────────────────┘

│

┌─────────────▼─────────────┐

│ DTO / Model │

└─────────────┬─────────────┘

│

▼

┌────────────────────┐

│ Repository │ (@Repository)

└────────────────────┘

│

▼

┌────────────────────┐

│ Database (MySQL) │

└────────────────────┘

**🧩 Supporting Components:**

┌──────────────┐

│ Config │ → Security, CORS, Swagger, DB config

└──────────────┘

┌──────────────┐

│ Aspect │ → Logging, Auditing, Exception Handling

└──────────────┘

┌──────────────┐

│ Demo │ → Test code, sample logic (can be removed later)

└──────────────┘

**🔄 Overall Flow Example:**

Suppose a user makes a **POST /bookings** request to your system.

1. **Client/Frontend** sends a request.
2. **Controller** handles the request, parses input DTO.
3. **Service Layer** processes the logic (e.g., check availability, calculate price).
4. It may convert DTO to **Model** (Entity).
5. **Repository** persists it to the **Database**.
6. The saved model is converted back into a response **DTO**.
7. **Controller** returns the response.
8. **Aspect** (if applied) logs the request/response time, or handles exceptions.
9. **Config** ensures proper app settings like Swagger, security, etc.

**📌 Summary of Each Layer:**

| **Layer** | **Role** |
| --- | --- |
| **Controller** | Handles HTTP requests/responses |
| **DTO** | Transfer data between layers (to avoid exposing entities directly) |
| **Service** | Contains business logic |
| **Repository** | Interacts with DB |
| **Model** | Represents DB tables |
| **Config** | Sets up app-wide settings |
| **Aspect** | Adds cross-cutting concerns |
| **Demo** | Used for testing/sample data |

Let me know if you'd like this as a **diagram image (PNG)** or want an example using actual classes.