**Door-to-Door Doctor Service - Software Architecture Document**

We're creating a software system that connects patients who need medical care at home with qualified doctors who can visit them. Think of it like Uber, but for medical house calls. This system needs to handle everything from booking appointments to tracking doctors' locations, managing medical records, and processing payments.

Clean Architecture

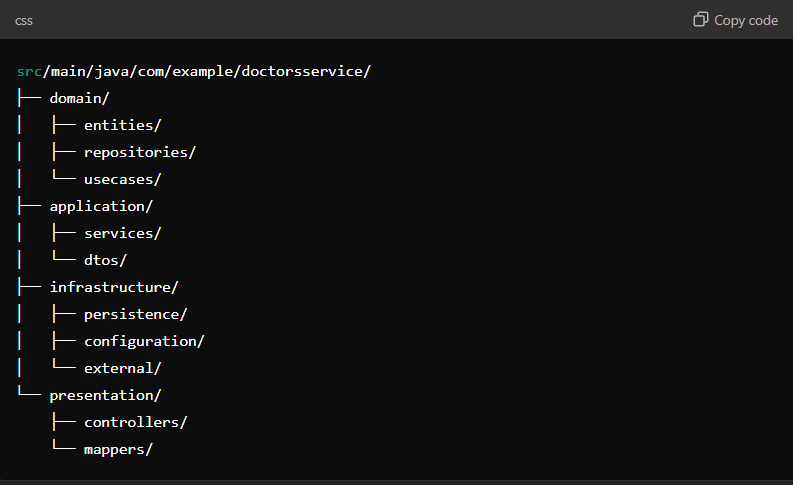
We're using Clean Architecture because it's like building a house with clear separations between the foundation, walls, and roof. Each part has its own job, but they work together perfectly. This makes our system:

- Easier to test and maintain

- Simple to update or change parts without affecting others

- More secure and reliable

- Scalable as the business grows



**Break down each layer of our architecture**:

**##1. Domain Layer (The Core)**

This is like the foundation of our house. It contains the most important business rules and doesn't depend on anything else.



**Structure and Purpose**

This domain layer is responsible for defining the core business logic and rules of the application. It is framework-agnostic and serves as the foundation for other layers in the clean architecture.

**1. entities/**

* **Purpose**: This directory contains the core business models of the application.
* **Details**:
  + **Doctor.java**: Represents a doctor and includes essential information like their name, specialization, and unique identifier.
  + **Patient.java**: Represents a patient and includes attributes like their name, age, and contact information.
  + **Appointment.java**: Represents an appointment, which links a doctor and a patient, and includes details like the date, time, and status.

**2. repositories/**

* **Purpose**: This directory defines interfaces for interacting with data sources (databases, APIs, etc.).
* **Details**:
  + **DoctorRepository.java**: Specifies methods for managing doctor data (e.g., retrieving, saving, or deleting).
  + **PatientRepository.java**: Specifies methods for managing patient data.
  + **AppointmentRepository.java**: Specifies methods for managing appointment data, such as fetching appointments for a specific doctor or patient.

**3. usecases/**

* **Purpose**: This directory implements specific business workflows or operations.
* **Details**:
  + **GetAllDoctorsUseCase.java**: Handles retrieving all doctors from the system.
  + **GetPatientAppointmentsUseCase.java**: Manages fetching all appointments for a specific patient.
  + **CreateAppointmentUseCase.java**: Handles the creation of a new appointment in the system.

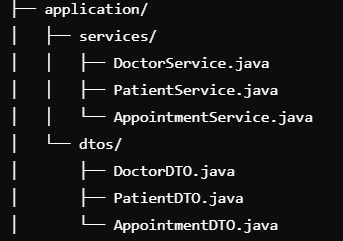
**Key Features**

1. **Separation of Concerns**: Each subdirectory serves a distinct purpose:
   * entities define the core data models.
   * repositories provide abstractions for data access.
   * usecases encapsulate business logic.
2. **Independence**:
   * The domain layer does not depend on other layers or frameworks, making it reusable and testable.
3. **Reusability**:
   * The use cases can be invoked by the application layer or presentation layer to execute specific actions.

This domain layer is a critical part of the clean architecture, ensuring that the core business logic is protected from changes in other layers like infrastructure or presentation. It allows for a clear and maintainable codebase.

**## 2. Application Layer (The Coordinator)**

This layer is like a traffic controller, making sure everything flows smoothly between different parts of the system.



**Application Layer Overview**

The **application layer** acts as a mediator between the domain layer and the outer layers (such as infrastructure or presentation). It is responsible for orchestrating use cases and preparing data for presentation.

**Structure and Purpose**

**1. services/**

* **Purpose**:
  + This subdirectory contains classes that implement the business logic exposed by the domain layer's use cases.
  + These services are responsible for coordinating tasks, calling domain use cases, and interacting with repositories to perform operations.
  + The services ensure the orchestration of workflows specific to the application's needs.
* **Details**:
  + **DoctorService.java**:
    - Provides functionality related to doctors, such as retrieving doctor details, managing their data, or delegating tasks to domain use cases and repositories.
  + **PatientService.java**:
    - Manages operations for patients, such as fetching their details or handling interactions between the domain use cases and repositories.
  + **AppointmentService.java**:
    - Handles appointment-related workflows, such as scheduling, canceling, or retrieving appointments by interacting with the domain layer.

**2. dtos/**

* **Purpose**:
  + This subdirectory contains **Data Transfer Objects (DTOs)**, which are used to transfer data between the application layer and the presentation layer (or between different services).
  + DTOs ensure that the data presented to the user or other layers is clean, concise, and tailored for the intended use case.
  + They also prevent direct exposure of domain entities, which helps maintain encapsulation and security.
* **Details**:
  + **DoctorDTO.java**:
    - Represents the data structure for transferring doctor-related information, such as name, specialization, and contact details, to/from the presentation layer or external services.
  + **PatientDTO.java**:
    - Represents the data structure for patient-related information, such as name, age, and contact information, in a simplified format for external interactions.
  + **AppointmentDTO.java**:
    - Encapsulates appointment-related data, such as the appointment date, doctor information, patient details, and status, for transfer across layers.

**Key Features and Benefits**

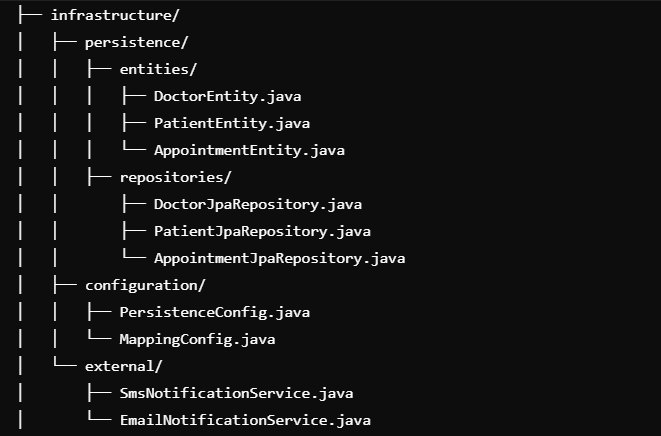
1. **Separation of Concerns**:
   * The services are responsible for implementing workflows and ensuring proper interaction with the domain layer.
   * The dtos handle data formatting and transfer, ensuring the presentation layer or external services receive structured and meaningful information.
2. **Encapsulation**:
   * Domain entities remain hidden from other layers, and only relevant information is exposed through DTOs, enhancing security and reducing coupling.
3. **Reusability**:
   * Services can be reused across multiple workflows or endpoints in the application.
   * DTOs allow consistent data representation, which simplifies data exchange and integration.
4. **Testability**:
   * The application layer can be tested independently by mocking the domain and infrastructure layers, ensuring workflows function as intended.
5. **Interfacing**:
   * By using DTOs, this layer acts as a clean interface between the domain logic and external systems or UI layers, reducing complexity and increasing maintainability.

**Purpose in Clean Architecture**

The application layer bridges the **domain layer** and the **infrastructure or presentation layers**, ensuring:

* Domain logic is executed as per the application's workflows.
* Data is properly formatted for external use or display. This keeps the codebase modular and aligned with the principles of clean architecture

**### 3. Infrastructure Layer (The Technical Foundation)**



**Infrastructure Layer Overview**

**The infrastructure layer handles implementation details such as data persistence, external integrations, and system configuration. It serves as the bridge between the domain/application layers and the external systems like databases or notification services.**

**Structure and Purpose**

**1. persistence/**

**This subdirectory contains everything related to data persistence and interaction with the database.**

* **entities/:**
  + **Contains database-specific representations of the core domain entities. These entities often map directly to database tables using ORM tools like JPA (Java Persistence API).**
  + **DoctorEntity.java: Represents the database schema for doctors, including fields like id, name, specialization, etc.**
  + **PatientEntity.java: Represents the database schema for patients with fields like id, name, contact, etc.**
  + **AppointmentEntity.java: Represents the database schema for appointments, linking doctors and patients with fields like appointmentDate, doctorId, and patientId.**
* **repositories/:**
  + **Contains the implementations of the repository interfaces defined in the domain layer. These are typically implemented using Spring Data JPA or similar tools.**
  + **DoctorJpaRepository.java: Handles CRUD operations for doctor entities in the database.**
  + **PatientJpaRepository.java: Handles CRUD operations for patient entities.**
  + **AppointmentJpaRepository.java: Handles CRUD operations for appointment entities, including custom queries if needed.**

**2. configuration/**

**This subdirectory contains classes responsible for setting up and configuring application infrastructure.**

* **PersistenceConfig.java:**
  + **Configures the database connection, entity manager, and transaction management.**
  + **Ensures the application connects to the correct database with appropriate settings.**
* **MappingConfig.java:**
  + **Configures mappings between domain entities and persistence entities, often using tools like MapStruct.**
  + **Facilitates conversion between the domain models and persistence models for seamless data flow.**

**3. external/**

**This subdirectory contains integrations with external systems or services that the application depends on.**

* **SmsNotificationService.java:**
  + **Represents a service for sending SMS notifications to users, such as appointment reminders or confirmations.**
  + **Typically integrates with an external SMS gateway API.**
* **EmailNotificationService.java:**
  + **Represents a service for sending email notifications to users.**
  + **Typically integrates with an email service provider like SendGrid, Amazon SES, or similar.**

**### 4. Presentation Layer (The Interface)**

