

# Architectural Documentation: Medical Clinic Administration System

## 1. Introduction & Vision

### 1.1. Purpose

This document outlines the software architecture for the **Medical Clinic Administration System**. The system is designed to manage core administrative workflows, including patient information, appointment scheduling, and billing.

### 1.2. Guiding Principles

The architecture is founded on a set of modern software engineering principles to ensure the system is **scalable, maintainable, and resilient**.

- **Domain-Driven Design (DDD):** The system is modeled around the business domain, using Bounded Contexts to break down complexity and create a shared language between technical and business teams.
- **Microservices Architecture:** The system is decomposed into a suite of independently deployable services, allowing for technological diversity, fault isolation, and independent team workflows.
- **Hexagonal Architecture (Ports & Adapters):** Each microservice's core business logic is isolated from external concerns (like databases, UI, or messaging), making it highly testable and adaptable to new technologies.
- **Event-Driven Communication:** Services communicate asynchronously via events where possible. This promotes loose coupling and enhances system resilience and scalability.

## 2. System Architecture (C4 Model: Level 1 & 2)

We use the C4 model to visualize the architecture at different levels of abstraction.

### 2.1. Level 1: System Context Diagram

This diagram shows the system as a black box in its environment, identifying its users and key external dependencies.

```
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Context.puml

title System Context Diagram for Clinic Administration System

Person(patient, "Patient", "A person receiving medical care.")
Person(staff, "Clinic Staff", "A doctor, nurse, or administrator.")

System_Ext(email_gw, "Email Service", "External service for sending emails
```

```
(e.g., SendGrid).")
System_Ext(sms_gw, "SMS Gateway", "External service for sending SMS
messages (e.g., Twilio).")
System_Ext(payment_gw, "Payment Gateway", "External service for processing
credit card payments (e.g., Stripe).")
```

```
System(clinic_system, "Clinic Administration System", "Manages patient
data, scheduling, and billing.")
```

```
Rel(staff, clinic_system, "Manages appointments, patient records, and
invoices")
```

```
Rel(patient, clinic_system, "Views and manages their appointments and
bills")
```

```
Rel(clinic_system, email_gw, "Sends notifications and reminders")
```

```
Rel(clinic_system, sms_gw, "Sends notifications and reminders")
```

```
Rel(clinic_system, payment_gw, "Processes payments for invoices")
```

```
@enduml
```

### ### 2.2. Level 2: Container Diagram

This diagram zooms into the system boundary to show the high-level technical building blocks (our microservices).

```
@startuml
```

```
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-
PlantUML/master/C4_Container.puml
```

```
title Container Diagram for Clinic Administration System
```

```
Person(staff, "Clinic Staff", "Uses the system via the Web App.")
```

```
System_Ext(payment_gw, "Payment Gateway")
```

```
System_Boundary(clinic_system, "Clinic Administration System") {
    Container(spa, "Web Application", "JavaScript, React", "Provides all
user-facing functionality.")
```

```
    Container(api_gateway, "API Gateway", "Spring Cloud Gateway", "Routes
external requests to internal services. Handles authentication.")
```

```
    ContainerDb(id_db, "Identity DB", "PostgreSQL", "Stores user
credentials and roles.")
```

```
    Container(identity_svc, "Identity & Access Service", "Spring Boot",
"Manages authentication (login) and authorization (permissions).")
```

```
    ContainerDb(patient_db, "Patient DB", "PostgreSQL")
```

```
    Container(patient_svc, "Patient Management Service", "Spring Boot",
"Manages patient demographics and registration.")
```

```
    ContainerDb(scheduling_db, "Scheduling DB", "PostgreSQL")
```

```
    Container(scheduling_svc, "Scheduling Service", "Spring Boot", "Handles
appointment booking, rescheduling, and provider availability.")
```

```
    ContainerDb(billing_db, "Billing DB", "PostgreSQL")
```

```

    Container(billing_svc, "Billing Service", "Spring Boot", "Manages
invoices, payments, and insurance claims.")

    Container(notification_svc, "Notifications Service", "Spring Boot",
"Sends emails and SMS messages.")

    Container(message_bus, "Message Bus", "Apache Kafka", "Asynchronous
communication backbone for inter-service events.")
}

' User to System Relationships
Rel(staff, spa, "Uses", "HTTPS")
Rel(spa, api_gateway, "Makes API calls", "HTTPS/JSON")

' API Gateway to Services
Rel(api_gateway, identity_svc, "Authenticates/Authorizes", "REST/HTTPS")
Rel(api_gateway, patient_svc, "Manages Patients", "REST/HTTPS")
Rel(api_gateway, scheduling_svc, "Manages Appointments", "REST/HTTPS")
Rel(api_gateway, billing_svc, "Manages Invoices", "REST/HTTPS")

' Synchronous Inter-service Communication
Rel(scheduling_svc, patient_svc, "Gets patient details", "REST/HTTPS")

' Asynchronous Event-Driven Communication
Rel_Down(scheduling_svc, message_bus, "Publishes
'AppointmentCompletedEvent'")
Rel_Down(message_bus, billing_svc, "Consumes 'AppointmentCompletedEvent'")
Rel_Down(billing_svc, message_bus, "Publishes 'InvoiceIssuedEvent'")
Rel_Down(message_bus, notification_svc, "Consumes events to send
notifications")

' Database Relationships
Rel(identity_svc, id_db, "Reads/Writes")
Rel(patient_svc, patient_db, "Reads/Writes")
Rel(scheduling_svc, scheduling_db, "Reads/Writes")
Rel(billing_svc, billing_db, "Reads/Writes")

' External System Relationships
Rel(billing_svc, payment_gw, "Processes Payments", "API")
@enduml

```

### 3. Microservice Internal Architecture (C4 Model: Level 3)

This diagram zooms into a single microservice to illustrate the Hexagonal Architecture pattern. We use the Billing Service as our example.

```

@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Component.puml

```

title Component Diagram for Billing Service (Hexagonal Architecture)

```

Container(message_bus, "Kafka", "Message Bus")
Container(api_gateway, "API Gateway", "Spring Cloud Gateway")
ContainerDb(billing_db, "Billing DB", "PostgreSQL")

```

```

package "Billing Service" {
  package "Application Core (The Hexagon)" {
    package "Domain Model" {
      Component(invoice_agg, "Invoice Aggregate", "Java",
"Encapsulates all business logic and rules for an invoice. The heart of the
service.")
    }

    package "Application Services (Use Cases)" {
      Component(invoice_svc, "InvoiceService", "Java/Spring",
"Implements use cases (e.g., CreateInvoiceUseCase). Orchestrates domain
model and repositories.")
    }

    package "Ports (Interfaces)" {
      Component(in_ports, "Inbound Ports\n(Use Case Interfaces)",
"Java Interfaces", "Defines the API of the application core (e.g.,
CreateInvoiceUseCase).")
      Component(out_ports, "Outbound Ports\n(Repository/Publisher
Interfaces)", "Java Interfaces", "Defines the requirements of the core for
external functionality (e.g., InvoiceRepository).")
    }
  }

  package "Adapters (Infrastructure)" {
    package "Inbound Adapters" {
      Component(rest_controller, "InvoiceController", "Spring MVC",
"Handles synchronous HTTP requests from the API Gateway.")
      Component(acl_listener, "AppointmentEventListener (ACL)",
"Spring Kafka", "Handles asynchronous events from other services.
Translates and delegates to the core.")
    }
    package "Outbound Adapters" {
      Component(jpa_adapter, "JpaInvoiceRepository", "Spring Data
JPA", "Implements the InvoiceRepository port for PostgreSQL.")
      Component(event_publisher, "KafkaEventPublisher", "Spring
Kafka", "Implements a port to publish domain events to Kafka.")
    }
  }
}

' Relationships
Rel(invoice_svc, invoice_agg, "Uses")
Rel(invoice_svc, out_ports, "Uses")

Rel(rest_controller, in_ports, "Calls")
Rel(acl_listener, in_ports, "Calls")

Rel(jpa_adapter, out_ports, "Implements")
Rel(event_publisher, out_ports, "Implements")

' External Connections
Rel(api_gateway, rest_controller, "Forwards requests to", "REST/HTTPS")

```

```

Rel(message_bus, acl_listener, "Receives events from", "Kafka Protocol")
Rel(jpa_adapter, billing_db, "Reads/Writes to", "JDBC")
Rel(event_publisher, message_bus, "Publishes events to", "Kafka Protocol")

@enduml

```

#### Architectural Flow Example (Creating an Invoice):

The Scheduling Service publishes an AppointmentCompletedEvent to Kafka. The AppointmentEventListener (an inbound adapter in the Billing Service) consumes this event. This adapter is our Anti-Corruption Layer (ACL). The listener uses a translator to convert the event into a CreateInvoiceCommand, which is a clean, internal representation. It calls the CreateInvoiceUseCase (an inbound port), implemented by the InvoiceService. The InvoiceService orchestrates the logic: it calls the InvoiceRepository (an outbound port) to check for duplicates. It then uses the Invoice aggregate (domain model) factory method to create a new invoice, which enforces all business rules. Finally, it calls the InvoiceRepository.save() method to persist the new invoice. The JpaInvoiceRepository (an outbound adapter) handles the translation to a JPA entity and saves it to the database.

#### 4. Cross-Cutting Concerns

**Security:** Authentication is centralized in the Identity & Access Service, which issues JWTs (JSON Web Tokens). The API Gateway is responsible for validating these tokens on every incoming request before forwarding it to a downstream service.

**Configuration Management:** Each microservice is a Spring Boot application and manages its configuration via application.yml. For production environments, a centralized configuration server (like Spring Cloud Config or HashiCorp Consul) is recommended.

**Service Discovery:** In a dynamic cloud environment, a service registry (like Eureka or Consul) should be used to allow services to find each other.

**Observability:**

**Logging:** A centralized logging solution (like the ELK Stack) will aggregate logs from all services.

**Metrics:** Each service will expose metrics using Spring Boot Actuator and Micrometer, which can be scraped by a monitoring system like Prometheus.

**Tracing:** Distributed tracing (using OpenTelemetry or Spring Cloud Sleuth) will be implemented to trace requests as they flow through multiple services.

#### 5. Technology Stack

Category	Technology	Rationale
Backend Framework	Spring Boot	Rapid development, mature ecosystem, excellent for building microservices.
Language	Java 17+	Robust, performant, and widely adopted.
Data Persistence	PostgreSQL	Powerful, open-source relational database with strong transactional support.
ORM	Spring Data JPA / Hibernate	Simplifies database interactions and integrates well with Spring.
Asynchronous Comms	Apache Kafka	High-throughput, durable message bus for event-driven architecture.

API Gateway Spring Cloud Gateway      Dynamic routing, security, and resilience patterns for APIs.  
Documentation      OpenAPI (Swagger)      Standard for defining and documenting RESTful APIs.  
Containerization      Docker      Standard for packaging applications and their dependencies.  
Orchestration      Kubernetes      Industry standard for deploying, scaling, and managing containerized apps.

<!--

[PROMPT\_SUGGESTION]Generate a CI/CD pipeline configuration using GitHub Actions for the Billing service.[/PROMPT\_SUGGESTION]

[PROMPT\_SUGGESTION]Show me the OpenAPI/Swagger documentation for the Patient Management service's REST API.[/PROMPT\_SUGGESTION]

-->