SIDIS Report

HAP

Class 3DB

Carlos Oliveira – 1220806

Vitor Barbosa - 1221412

Henrique Gonçalves - 1200968

TECHNICAL REPORT: SIDIS PROJECT (HAP - Healthcare Appointment Platform)

1. Introduction and Project Goal

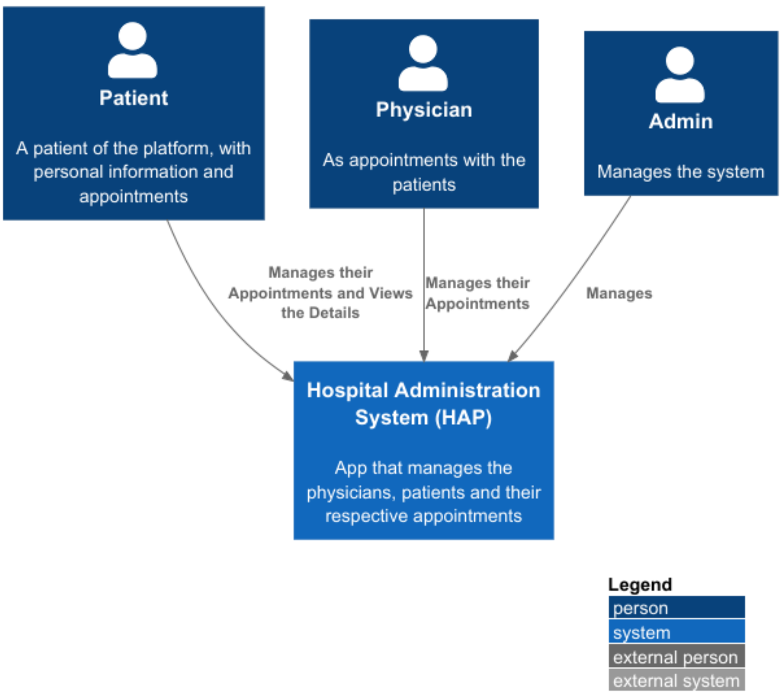
The primary objective of this project is to transform the monolithic *Healthcare Appointment Platform* (HAP) into a distributed system. This transition utilizes microservices architecture, appropriate design patterns, and distributed systems principles to ensure scalability, maintainability, and resilience.

1.1. Deployment Instructions

To deploy the HAP project, the following software and configurations are required:

- **Docker**: All microservices and their dependencies are managed within containers.
- **Docker Compose**: The compose.yaml file is used to install all images and create the necessary containers for the system to run.
- **Project Folder**: The SIDIS-25-26 folder contains all source code and technical documentation.

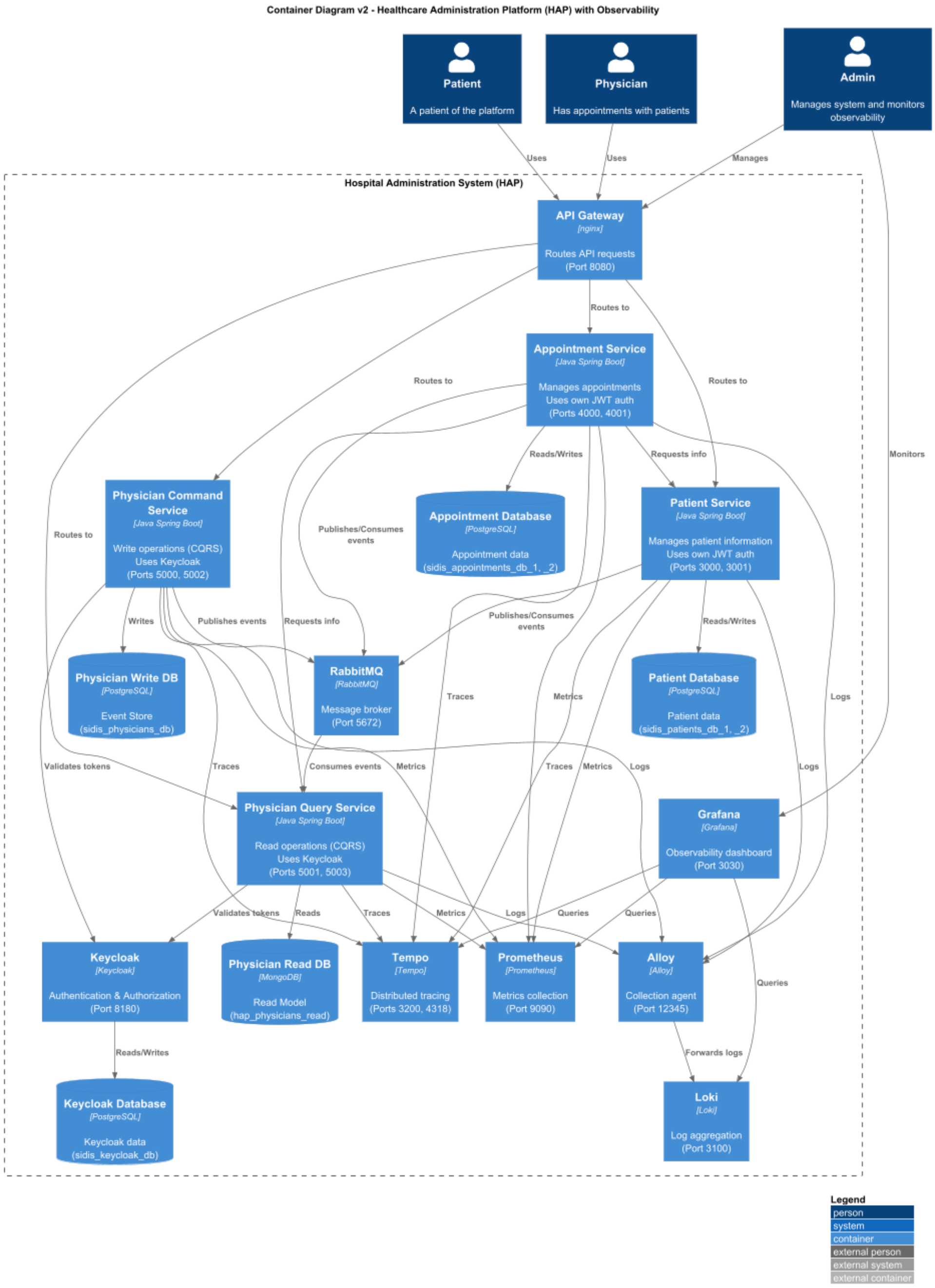
System Context diagram for the Healthcare Administration Platform (HAP).



2. System Architecture (C4 Model)

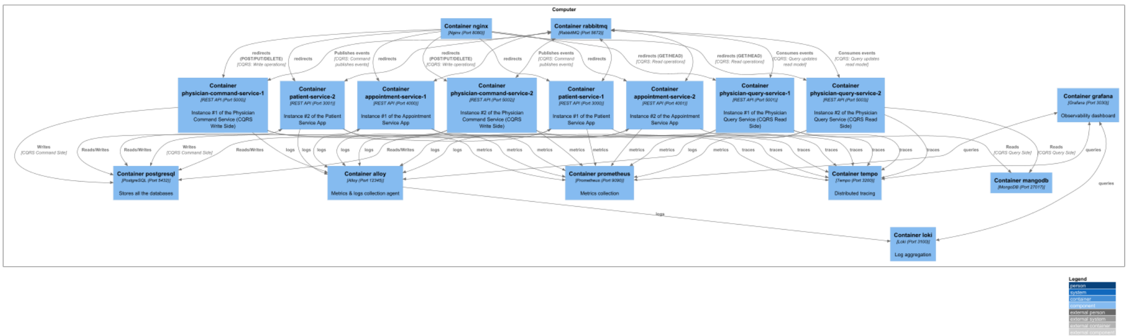
2.1. Containers and Components

The architecture is divided into independent services to isolate responsibilities and allow for independent scaling.

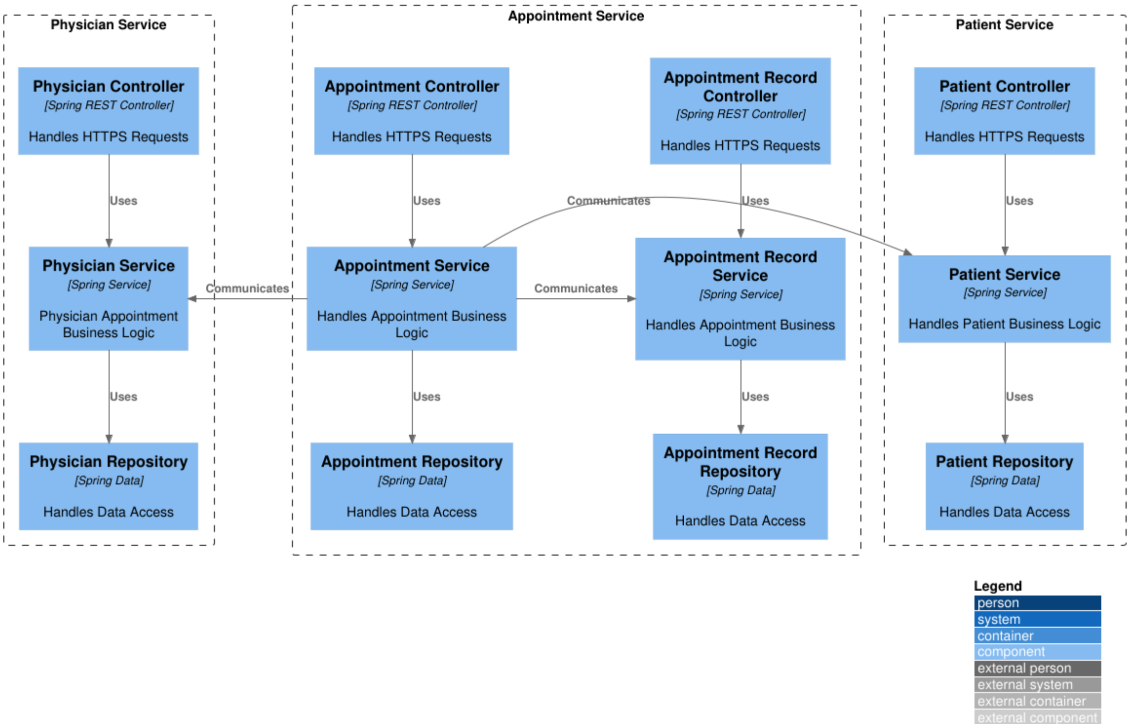


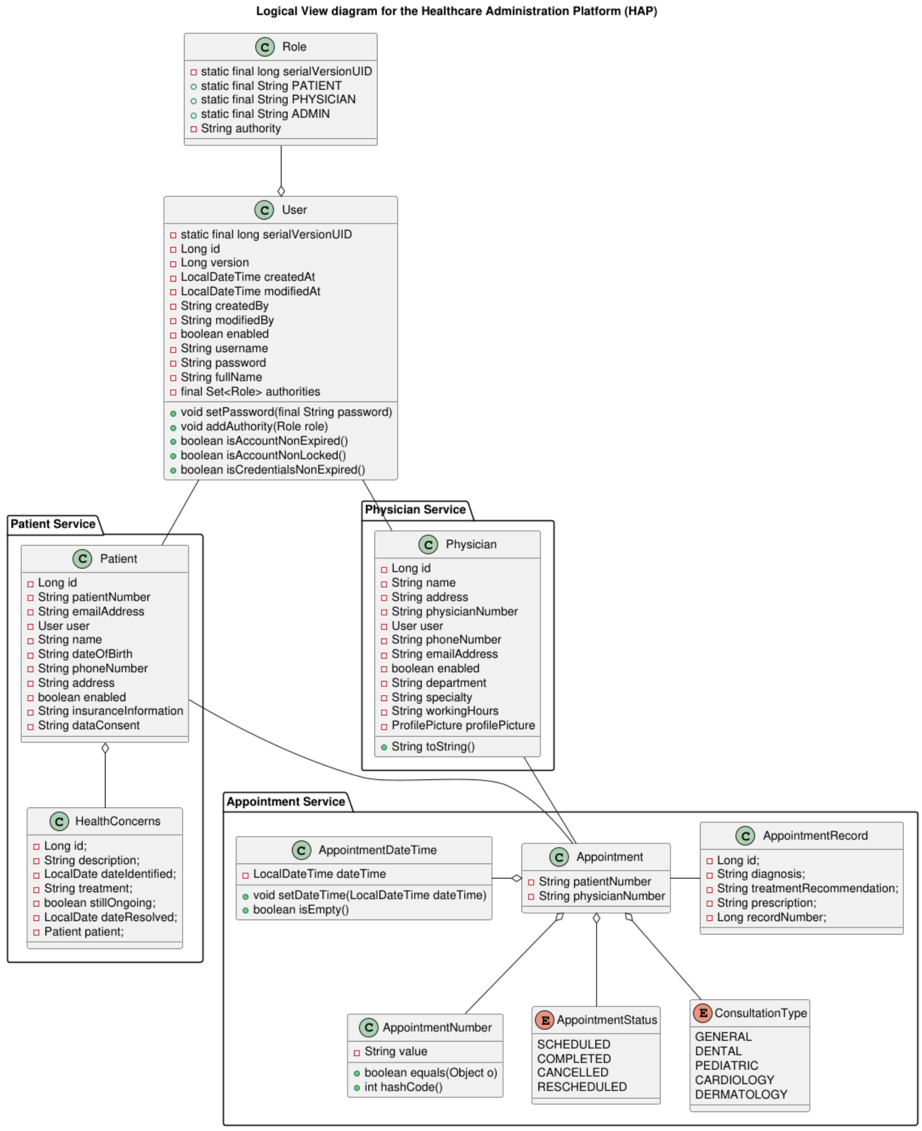
2.2. Physical and Logical Views

The physical view details the distribution of nodes across the infrastructure, while the logical view focuses on service interactions and domain organization.



Components diagram for the Healthcare Administration Platform (HAP).





3. API Gateway (Nginx)

The system uses **Nginx** as an API Gateway to provide a single entry point for all client requests, handling routing, load balancing, and SSL termination.

3.1. Gateway Features

- **Single Entry Point**: All external requests enter through port **8080** (HTTP), hiding individual service ports from the client.
- **Load Balancing**: Uses the **Least Connections** (least_conn) algorithm to distribute requests to the instance with the fewest active connections.
- **High Availability**: Each service runs **2 instances** simultaneously with automatic failover and health checks via /actuator/health.

3.2. Service Routing and CQRS Logic

- **Appointments Service**: Path /api/appointments/* routed to ports 4000/4001.

- **Patients Service**: Path /api/patients/* routed via HTTPS with specific load balancing to handle SNI.
- **Physicians Service (CQRS)**: Routing is based on the HTTP method:
    - **Read Operations (GET/HEAD)**: Routed to the **Query Service** (MongoDB backend).
    - **Write Operations (POST/PUT/PATCH/DELETE)**: Routed to the **Command Service** (PostgreSQL backend).

4. Architectural Decisions and Patterns

4.1 Bounded Contexts

The primary application of DDD was the division of the monolith into three Bounded Contexts, which became the microservices, each managing its own domain and data.

DDD Concept	Microservices	Domain Responsibility
Bounded Context 1: Patient	Patient	Manage patient data, authenticate patient users.
Bounded Context 2: Physician	Physician	Manage physician data, authenticate physician users, and manage work schedules.
Bounded Context 3: Scheduling	Appointment	Manage appointments and consultation records, coordinating with the other two services.

This segregation ensures that each business concept has clear and unique meaning and business rules within the boundaries of its own service.

4.2. Database Engine

- **PostgreSQL**: Chosen for its reliability as a shared database and write store for commands.
- **MongoDB**: Implemented as a read-optimized store for the Query side of CQRS.

4.3. CQRS Pattern

Implemented for the Physician.

- **Rationale**: The frequency of reading physician data is significantly higher than adding or dismissing physicians.

- **Synchronization**: Changes in the Command Service are synchronized to the Query Service via **RabbitMQ** event handlers.

4.4. Saga Pattern (Choreography)

For **Appointment Scheduling**, a Saga Choreography approach was designed to ensure data consistency across services without a centralized orchestrator.

5. Observability Stack

The system utilizes the **Grafana Labs** solution (Alloy, Loki, and Tempo) along with **Prometheus**.

5.1. Tool Comparison

Several tools were considered for the implementation, and a comparison can be seen in Table(adapted from H. Ahmed and H. J. Syed, “Observability in microservices: An in-depth explo ration of frameworks, challenges, and deployment paradigms,” IEEE Access, 2 2025.)

Tool	Open Source	Strengths	Limitations
Prometheus	Yes	Reliability and scalability	Does not support tracing or logs
Grafana	Yes	Sophisticated visualization	Does not collect data natively
Zipkin	Yes	Latency diagnosis and tracing	Does not include metrics or logs
Weave Scope	Yes	Container monitoring	Does not support tracing
Loki	Yes	Economical log management	Logs only
OpenTelemetry	Yes	Comprehensive framework	Requires external storage
BPFTrace	Yes	Dynamic tracing	Low-level observability
Istio	Yes	Traffic management	High complexity
Jaeger	Yes	Service analysis	Tracing only

Addressing the needs, the solutions that stood out were the ELK Stack (consisting of Elasticsearch, Logstash, and Kibana), developed by Elastic, and Alloy, Loki, and Grafana Dashboard, developed by Grafana Labs.

The Grafana Labs solution stands out due to its simplicity, efficiency, and the integration capabilities between its components (Loki for logs, Prometheus for metrics, and Tempo for traces).

Although the ELK Stack is a more established product, with a broader user community and superior search capabilities that allow it to handle complex queries and deliver results quickly, its complexity and higher resource consumption led to the conclusion that it would not be the most suitable option.

5.2. Data Collection and Flow

- **Alloy Configuration**: Reads log files from all services in /var/log/sidis/, adds service labels (e.g., service=patient-service), and sends them to Loki.
- **Prometheus Scraping**: Every 15 seconds, Prometheus pulls metrics from the /internal/prometheus endpoint of each service instance.
- **Grafana Provisioning**: Datasources (Loki, Prometheus, Tempo) are automatically configured at startup via datasources.yml.

```
┌──────────────────┐
│  PatientService  │  ──writes──> /SIDIS-25-26/logs/patient-service/app.log
└──────────────────┘
       │
┌──────────────────┐
│AppointmentService│  ──writes──> /SIDIS-25-26/logs/appointment-service/app.log
└──────────────────┘
       │
┌──────────────────────────────┐
│PhysicianCommandService       │  ──writes──> /SIDIS-25-26/logs/physician-command-service/app.log
│(Write Operations - CQRS)     │
└──────────────────────────────┘
       │
┌──────────────────────────────┐
│PhysicianQueryService         │  ──writes──> /SIDIS-25-26/logs/physician-query-service/app.log
│(Read Operations - CQRS)      │
└──────────────────────────────┘
                    │
                    │ (Alloy reads all files)
                    ▼
            ┌───────────────┐
            │     Alloy     │ (Separate service)
            │  (Container)  │
            └───────────────┘
                    │
                    │ (Adds service labels)
                    │
                    ▼
            ┌───────────────┐
            │     Loki      │ (All logs stored here with labels)
            └───────────────┘
                    │
                    │ (Grafana queries)
                    ▼
            ┌───────────────┐
            │    Grafana    │ (View all logs together, filter by service)
            └───────────────┘
```

Accessing Grafana

Once everything is running:

- **URL:** http://localhost:3030
- **Username:** admin
- **Password:** admin

You'll see all three datasources (Loki, Prometheus, Tempo) already configured.

API Contracts

The system exposes RESTful APIs documented via OpenAPI/Swagger.

General Standards:

- Protocol: HTTP/1.1 (TLS 1.2+ required).
- Format: JSON (Content-Type: application/json).
- Dates: ISO-8601

Example Contract: Create Appointment

- **Endpoint:** POST /api/v1/appointments
- **Requested body:** {"patientId": "123-abc","physicianId": "456-def","dateTime": "2024-12-25T10:00:00Z","type": "CONSULTATION"}

Deprecation Plan (Hypothetical API Change)

Scenario: We are migrating GET /api/v1/physicians (which returns a list) to GET /api/v2/physicians which supports pagination and advanced filtering.

Phase 1: Announcement

- **Action:** Mark v1 endpoint as @Deprecated in code and Swagger.
- **Header:** Add X-API-Deprecation-Date: 2025-06-01 to all v1 responses.
- **Communication:** Notify all consuming clients (frontend teams, external partners).

Phase 2: Brownout

- **Action:** Both v1 and v2 run in parallel.
- **Brownout:** Periodically inject artificial delays or warnings into v1 responses to alert developers relying on legacy endpoints.
- **Header:** Add Warning: 299 - "This API is deprecated and will be removed on 2025-06-01"

Phase 3: End of Life

- **Action:** Remove the v1 controller logic.
- **Response:** Requests to v1 return 410 Gone with a body pointing to v2 documentation.

Service Level Agreements (SLAs)

The HAP platform commits to the following SLAs for production environments:

Metric	Target	Definition
API Latency (Read)	< 200ms	95th percentile (p95) for GET requests (e.g., searching doctors).
API Latency (Write)	< 500ms	95th percentile (p95) for transactional requests (e.g., booking).
Data Consistency	Eventual (< 2s)	Time for a change in Command Service to reflect in Query Service.
Recovery Point (RPO)	5 minutes	Maximum data loss accepted in catastrophic failure (managed by DB backups).

Configuration Examples

RabbitMQ Configuration (Spring Boot)

Configuration ensuring durable queues and correct routing for the Saga events:

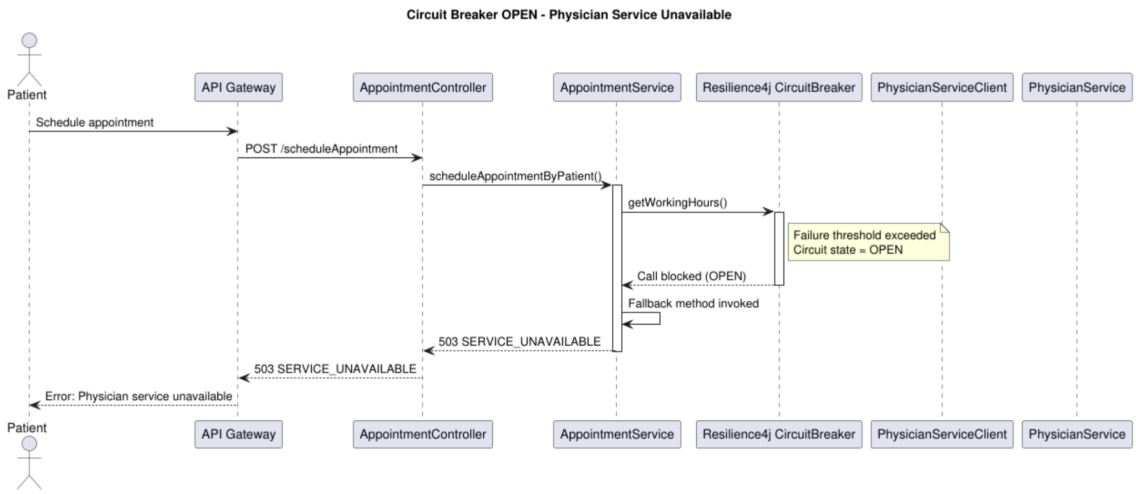
```
# application.properties

spring.rabbitmq.host=${RABBITMQ_HOST:rabbitmq}

spring.rabbitmq.port=5672

spring.rabbitmq.username=${RABBITMQ_USER}

spring.rabbitmq.password=${RABBITMQ_PASS}

# Exchange Definition for Sagas

app.rabbitmq.exchange.appointment=appointment-exchange

app.rabbitmq.queue.patient-verification=patient-verification-queue

app.rabbitmq.routing-key.appointment-created=appointment.created
```

Resilience Configuration (Resilience4j)

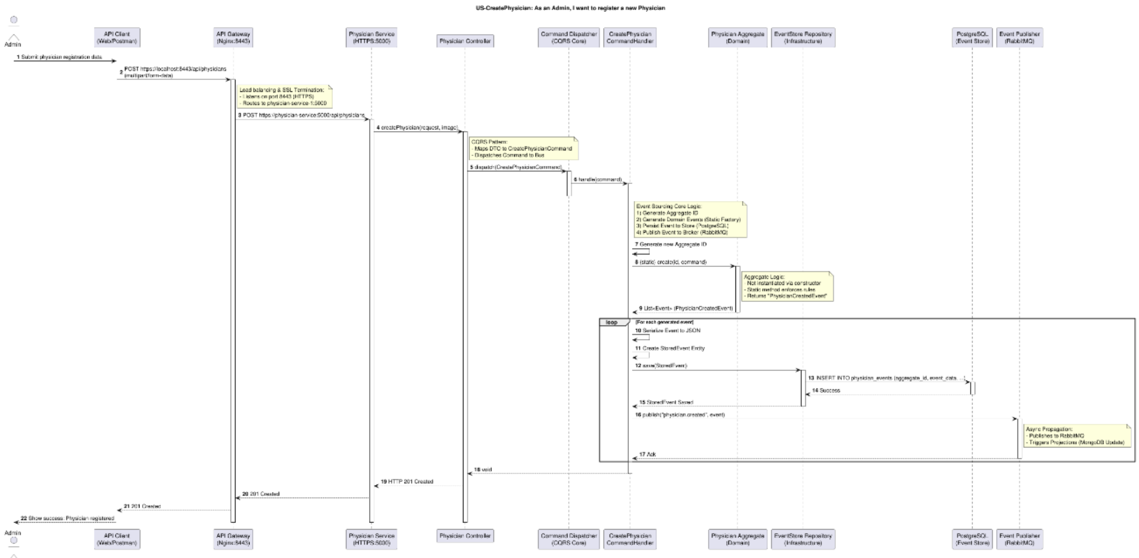
Applied to synchronous calls (e.g., getting basic data) to prevent cascading failures:

```
#application.properties

resilience4j:

 circuitbreaker:

  instances:

   patientService:

    registerHealthIndicator: true

    slidingWindowSize: 10

    failureRateThreshold: 50

    waitDurationInOpenState: 5s
```

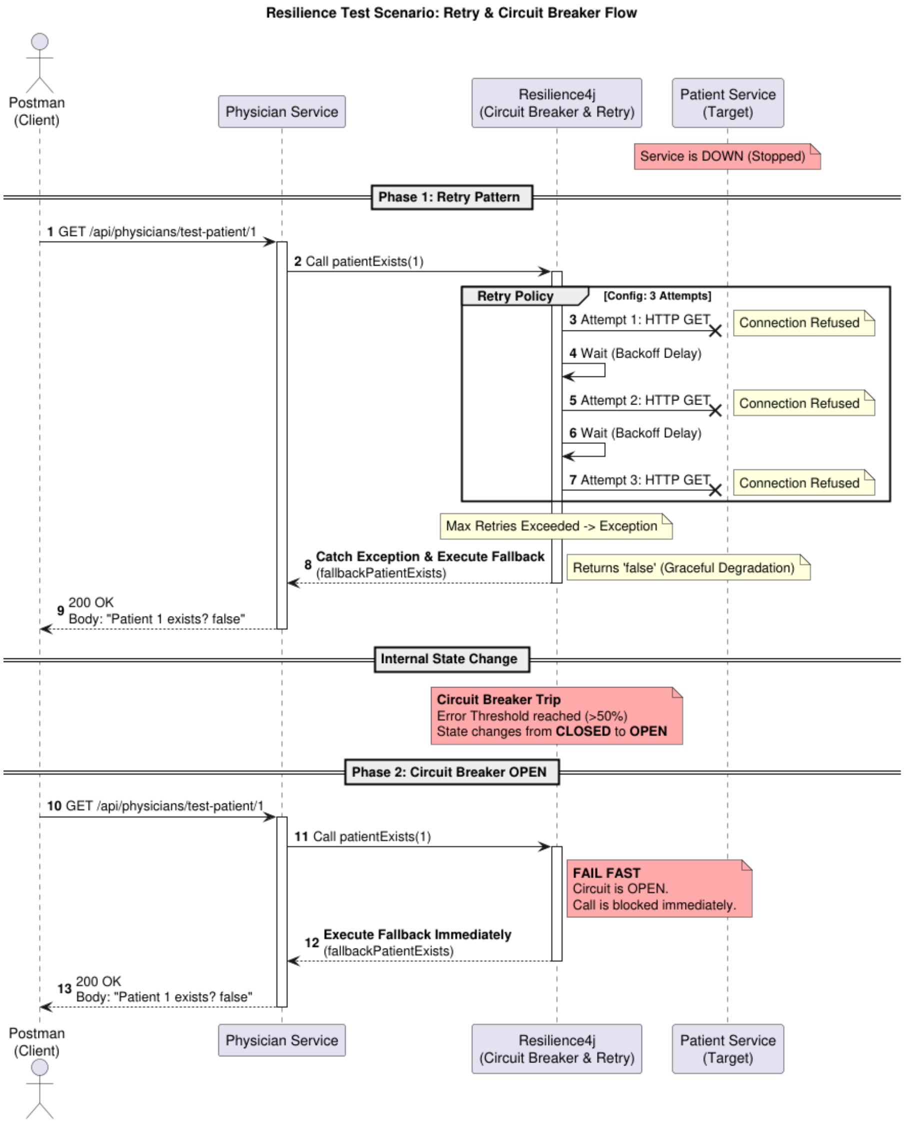
System Diagrams

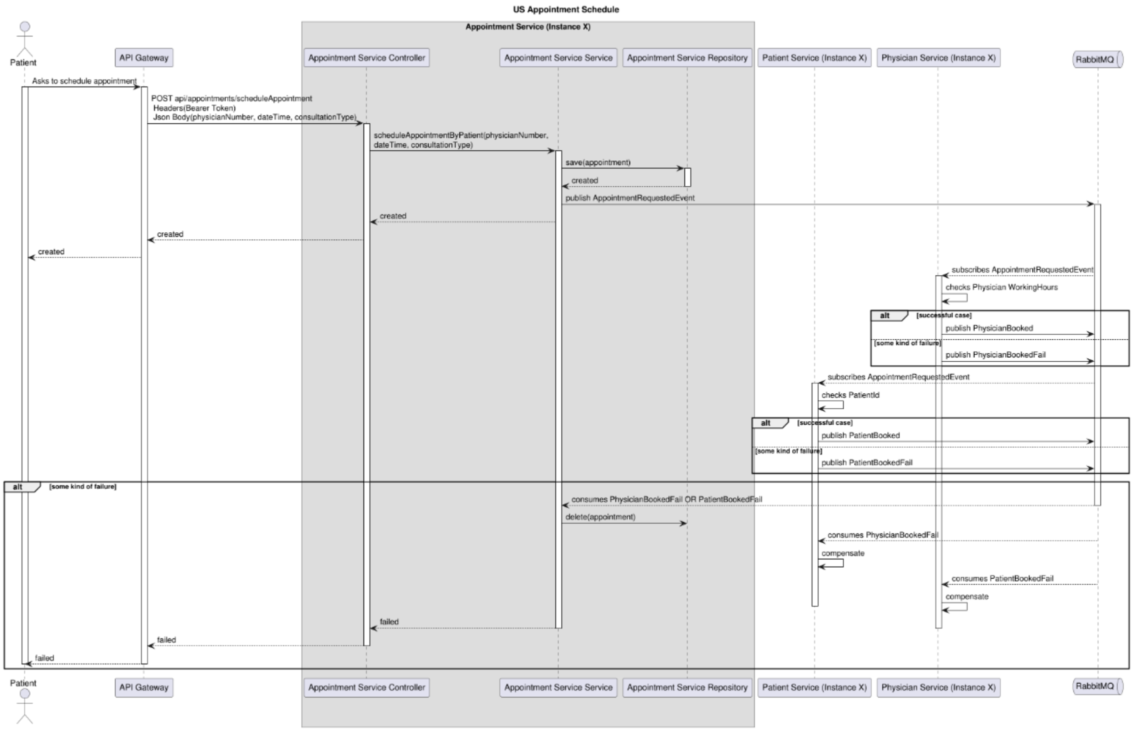


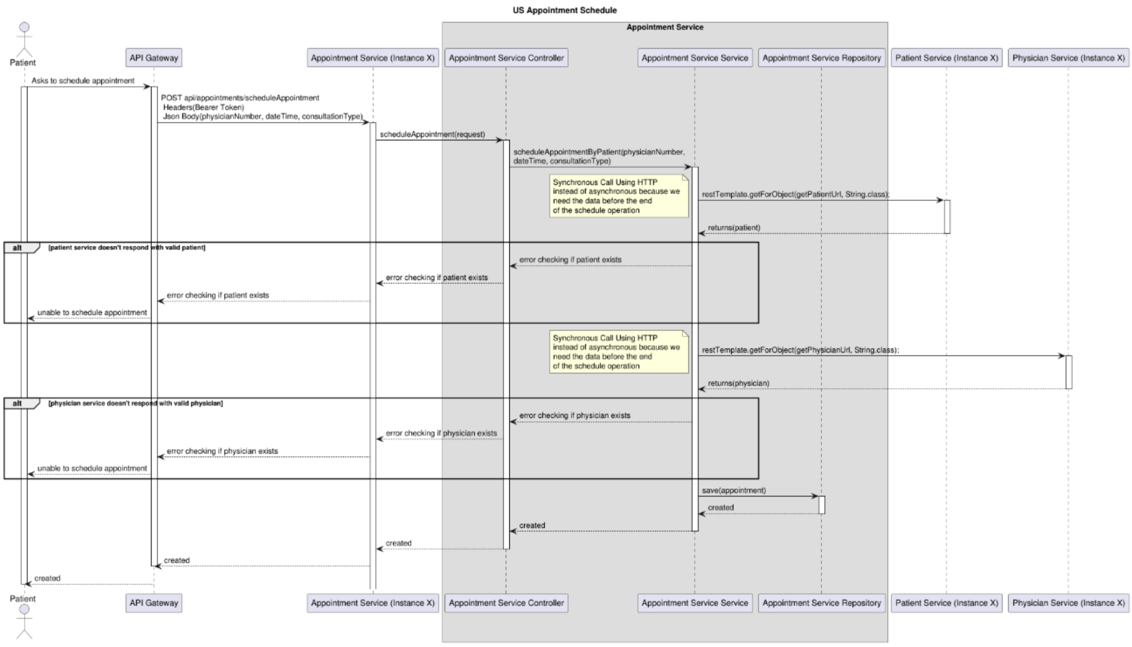
Circuit Breaker OPEN - Physician Service Unavailable
Patient
API Gateway
AppointmentController
AppointmentService
Resilience4j CircuitBreaker
PhysicianServiceClient
PhysicianService
Schedule appointment
POST /scheduleAppointment
scheduleAppointmentByPatient()
getWorkingHours()
Failure threshold exceeded
Circuit state = OPEN
Call blocked (OPEN)
Fallback method invoked
503 SERVICE_UNAVAILABLE
503 SERVICE_UNAVAILABLE
Error: Physician service unavailable



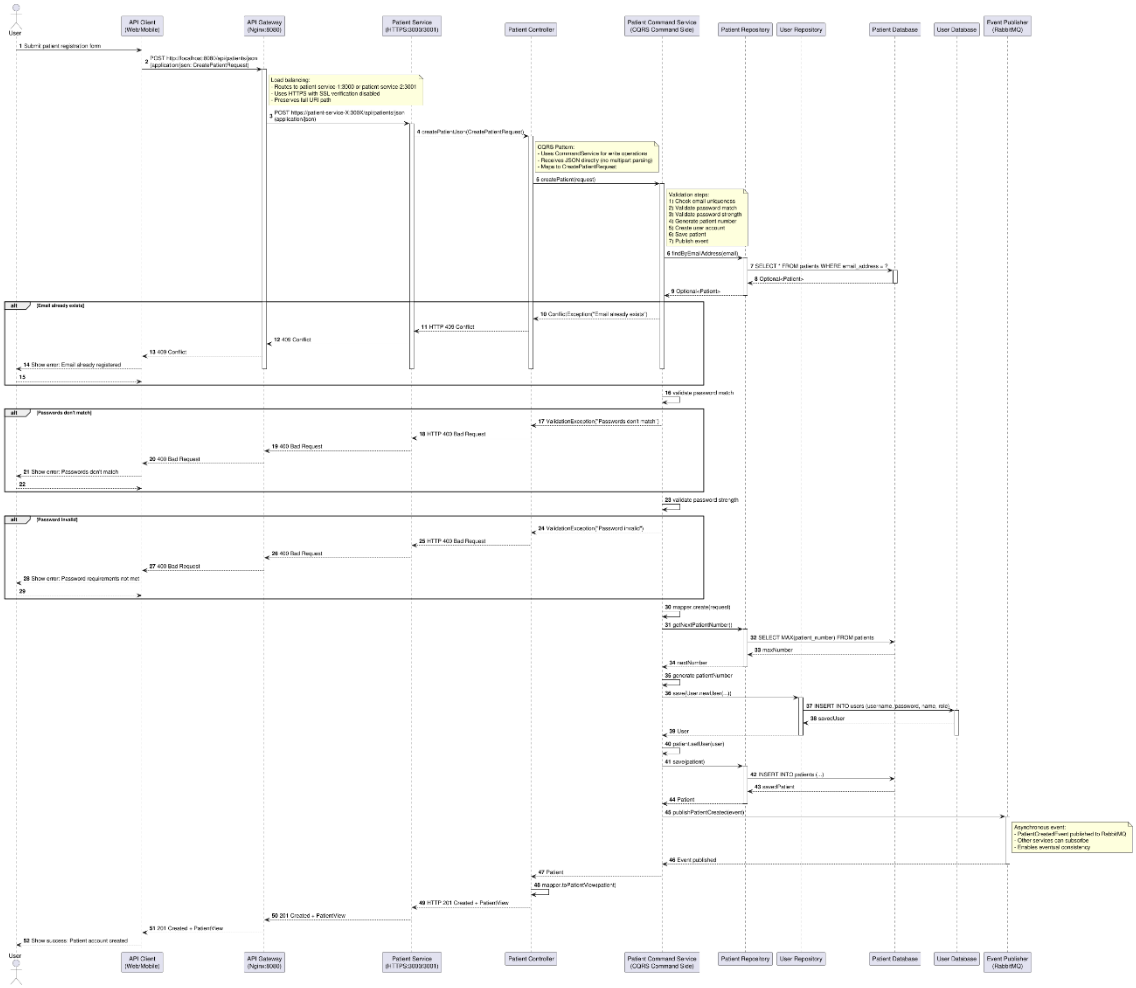
US-CreatePhysician: As an Admin, I want to register a new Physician
Admin
API Client (Web/Postman)
API Gateway (Nginx:8443)
Physician Service (HTTPS:5000)
Physician Controller
Command Dispatcher (CQRS Core)
CreatePhysician CommandHandler
Physician Aggregate (Domain)
EventStore Repository (Infrastructure)
PostgreSQL (Event Store)
Event Publisher (RabbitMQ)
1 Submit physician registration data
2 POST https://localhost:8443/api/physicians (multipart/form-data)
Load balancing & SSL Termination:
- Listens on port 8443 (HTTPS)
- Routes to physician-service-1:5000
3 POST https://physician-service:5000/api/physicians
4 createPhysician(request, image)
CQRS Pattern:
- Maps DTO to CreatePhysicianCommand
- Dispatches Command to Bus
5 dispatch(CreatePhysicianCommand)
6 handle(command)
Event Sourcing Core Logic:
1) Generate Aggregate ID
2) Generate Domain Events (Static Factory)
3) Persist Event to Store (PostgreSQL)
4) Publish Event to Broker (RabbitMQ)
7 Generate new Aggregate ID
8 (static) create(id, command)
Aggregate Logic:
- Not instantiated via constructor
- Static method enforces rules
- Returns "PhysicianCreatedEvent"
9 List<Event> (PhysicianCreatedEvent)
loop
[For each generated event]
10 Serialize Event to JSON
11 Create StoredEvent Entity
12 save(StoredEvent)
13 INSERT INTO physician_events (aggregate_id, event_data, ...)
14 Success
15 StoredEvent Saved
16 publish("physician.created", event)
Async Propagation:
- Publishes to RabbitMQ
- Triggers Projections (MongoDB Update)
17 Ack
18 void
19 HTTP 201 Created
20 201 Created
21 201 Created
22 Show success: Physician registered


Resilience Test Scenario: Retry & Circuit Breaker Flow
Postman (Client)
Physician Service
Resilience4j (Circuit Breaker & Retry)
Patient Service (Target)
Service is DOWN (Stopped)
Phase 1: Retry Pattern
1 GET /api/physicians/test-patient/1
2 Call patientExists(1)
Retry Policy
[Config: 3 Attempts]
3 Attempt 1: HTTP GET
Connection Refused
4 Wait (Backoff Delay)
5 Attempt 2: HTTP GET
Connection Refused
6 Wait (Backoff Delay)
7 Attempt 3: HTTP GET
Connection Refused
Max Retries Exceeded -> Exception
8 Catch Exception & Execute Fallback (fallbackPatientExists)
Returns 'false' (Graceful Degradation)
9 200 OK Body: "Patient 1 exists? false"
Internal State Change
Circuit Breaker Trip
Error Threshold reached (>50%)
State changes from CLOSED to OPEN
Phase 2: Circuit Breaker OPEN
10 GET /api/physicians/test-patient/1
11 Call patientExists(1)
FAIL FAST
Circuit is OPEN.
Call is blocked immediately.
12 Execute Fallback Immediately (fallbackPatientExists)
13 200 OK Body: "Patient 1 exists? false"
Postman (Client)
Physician Service
Resilience4j (Circuit Breaker & Retry)
Patient Service (Target)

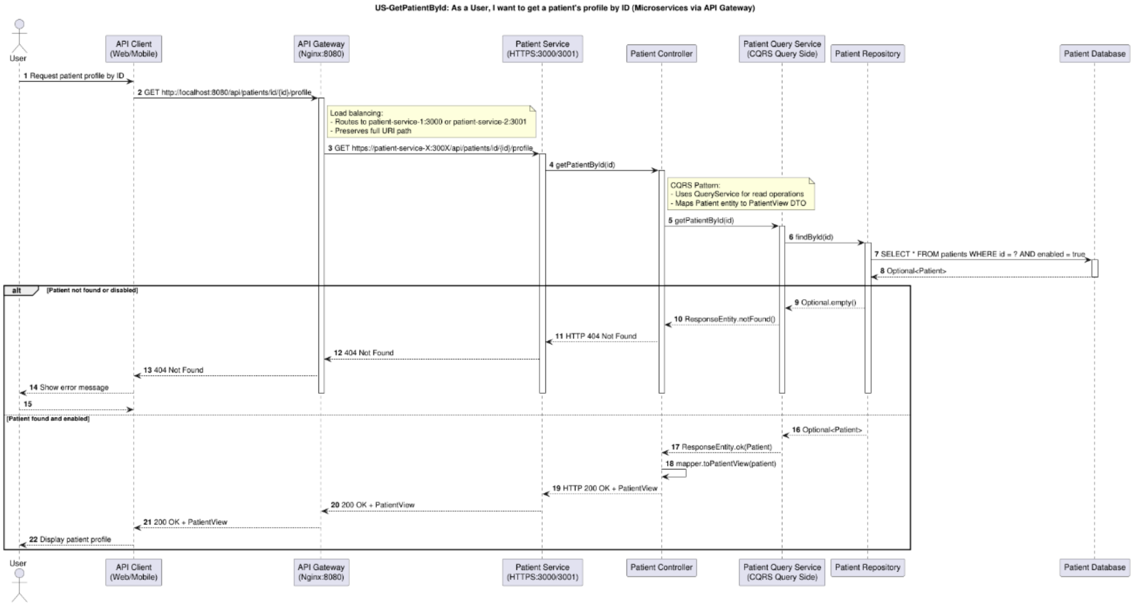

US Appointment Schedule
Appointment Service (Instance X)
Patient
API Gateway
Appointment Service Controller
Appointment Service Service
Appointment Service Repository
Patient Service (Instance X)
Physician Service (Instance X)
RabbitMQ
Asks to schedule appointment
POST api/appointments/scheduleAppointment
Headers(Bearer Token)
Json Body(physicianNumber, dateTime, consultationType)
scheduleAppointmentByPatient(physicianNumber, dateTime, consultationType)
save(appointment)
created
publish AppointmentRequestedEvent
created
created
created
subscribes AppointmentRequestedEvent
checks Physician WorkingHours
alt
[successful case]
publish PhysicianBooked
[some kind of failure]
publish PhysicianBookedFail
subscribes AppointmentRequestedEvent
checks PatientId
alt
[successful case]
publish PatientBooked
[some kind of failure]
publish PatientBookedFail
alt
[some kind of failure]
consumes PhysicianBookedFail OR PatientBookedFail
delete(appointment)
consumes PhysicianBookedFail
compensate
consumes PatientBookedFail
compensate
failed
failed
failed


US Appointment Schedule
Appointment Service
Patient
API Gateway
Appointment Service (Instance X)
Appointment Service Controller
Appointment Service Service
Appointment Service Repository
Patient Service (Instance X)
Physician Service (Instance X)
Asks to schedule appointment
POST api/appointments/scheduleAppointment
Headers(Bearer Token)
Json Body(physicianNumber, dateTime, consultationType)
scheduleAppointment(request)
scheduleAppointmentByPatient(physicianNumber, dateTime, consultationType)
Synchronous Call Using HTTP instead of asynchronous because we need the data before the end of the schedule operation
restTemplate.getForObject(getPatientUrl, String.class);
returns(patient)
alt
[patient service doesn't respond with valid patient]
error checking if patient exists
error checking if patient exists
error checking if patient exists
unable to schedule appointment
Synchronous Call Using HTTP instead of asynchronous because we need the data before the end of the schedule operation
restTemplate.getForObject(getPhysicianUrl, String.class);
returns(physician)
alt
[physician service doesn't respond with valid physician]
error checking if physician exists
error checking if physician exists
error checking if physician exists
unable to schedule appointment
save(appointment)
created
created
created
created
created

US-CreatePatient: As a User, I want to create a new patient account



US-GetPatientById: As a User, I want to get a patient's profile by ID (Microservices via API Gateway)



US-08: As a Patient, I want to view my appointments (Microservices)