EarlyCare Gateway

System Documentation

Contents

1 Introduction

This documentation describes the EarlyCare Gateway system, in terms of API endpoints, database architecture and deployment techniques.

It follows a microservices architecture, with a Gateway, an Authentication Service, a Data Processing Service, an Explainable AI Service, and an Audit Service, according to the Figure 1 below.

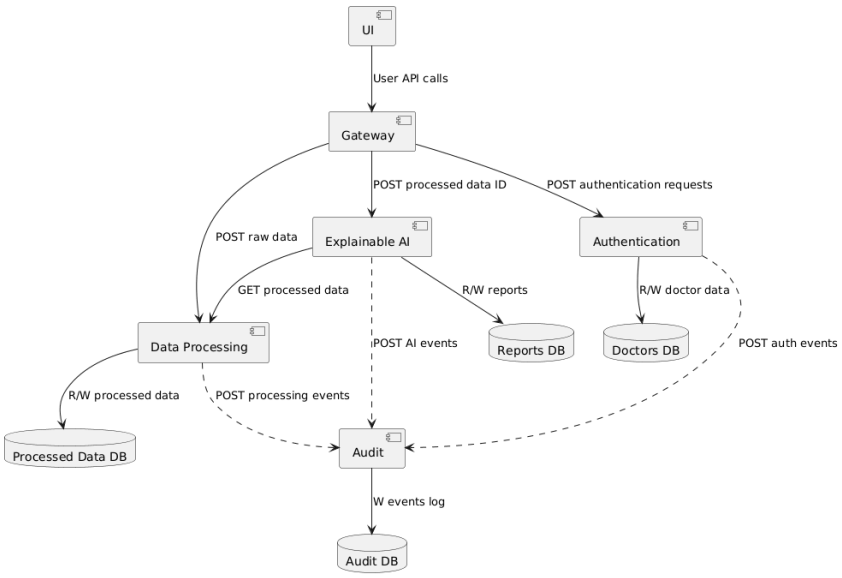

Figure 1: System Architecture Overview

Communication between services uses JSON format for requests and responses. Authentication is handled with JSON Web Tokens (JWT), and error handling uses standard HTTP status codes with JSON payloads.

1.1 Swagger OpenAPI Documentation

FastAPI automatically generates interactive API documentation accessible at `/docs`. This interface allows to inspect all endpoints, view request and response schemas, and perform tests directly from the browser.

1.2 Status and Error Codes (RFC 7807)

All the API endpoints illustrated in this documentation follow the RFC 7807 standard for error format and status codes, as shown in Table 1.

Status	**Description**
200 OK	Success
201 Created	Resource successfully created
400 Bad Request	Invalid input, missing parameters
401 Unauthorized	Invalid/expired JWT or credentials
404 Not Found	Resource not found

Table 1: Status and Error Codes Overview

1.3 Project Structure

The following is the high-level structure of the EarlyCare Gateway project (in-depth example of one microservice).

```
earlycaregateway/
+-- backend/
|   +-- authentication/
|   |   +-- app/
|   |   |   +-- models/
|   |   |   +-- services/
|   |   |   +-- schemas/
|   |   |   +-- routes/
|   |   |   +-- utils/
|   |   |   +-- main.py
|   |   +-- Dockerfile
|   |   +-- requirements.txt
|   +-- gateway/
|   +-- data_processing/
|   +-- explainable_ai/
|   +-- audit/
+-- frontend/
|   +-- public/
|   |-- src/
|   +-- Dockerfile
+-- docker-compose.yaml
+-- .env
+-- run.bat (run.sh)
```

- **backend/**: Contains all microservices (Gateway, Authentication, Data Processing, Explainable AI, Audit)
- **frontend/**: React-based UI application
- **docker-compose.yaml**: Orchestrates services for local deployment
- **Dockerfile**: Base image and build instructions
- **requirements.txt**: Python dependencies
- **.env**: Environment variables for configuration
- **run.bat/run.sh**: All-in-one system startup script

2 User Endpoints

2.1 POST /register

- **Purpose:** Register a new doctor
- **Authentication:** None
- **Request Body:**

```
{
  "name": "Mario",
  "surname": "Rossi",
  "email": "mario.rossi@email.com",
  "password": "RossiMarioPassword25"
}
```

- **Response Body (success):**

```
{
  "message": "Doctor registered successfully"
  "doctor_id": 1
}
```

- **Response Body (error):**

```
{
  "error": "Email already in use"
}
```

- **Status Codes:**
  - 201 Created – registration successful
  - 400 Bad Request – invalid input or email already exists
- **Flow Notes:**
  - Gateway forwards to Authentication Service
  - Authentication Service hashes password and creates unique `doctor_id`
  - Asynchronous notification sent to Audit Service

2.2 POST /login

- **Purpose:** Authenticate doctor and issue JWT
- **Authentication:** None
- **Request Body:**

```
{
  "email": "mario.rossi@email.com",
  "password": "RossiMarioPassword25"
}
```

- **Response Body (success):**

```
{
  "message": "Doctor logged in successfully",
  "token": "jwt_token"
}
```

- **Response Body (error):**

```
{
  "error": "Invalid credentials"
}
```

- **Status Codes:**
  - 200 OK – login successful
  - 401 Unauthorized – invalid credentials
- **Flow Notes:**
  - JWT contains `doctor_id, name, surname` in payload, signed with SHA256 secret
  - Gateway forwards token to UI for subsequent requests
  - Audit Service notified asynchronously

2.3 POST /analyse

- **Purpose:** Execute diagnosis
- **Authentication:** JWT required (`Authorization: Bearer <JWT>`)
- **Request Body:**

```
{
  "strategy": "text"/"img_rx"/"img_skin"/"numeric"/"signal",
  "patient_hashed_cd": "hash_string",
  "raw_data": "serialized_data_or_b64"
}
```

- **Response Body (success):**

```
{
  "message": "Report saved into the database successfully",
  "report":
  {
    "id": 1,
    "doctor_id": 3,
    "patient_hashed_cf": "hash",
    "processed_data_id": 5,
    "created_at": "2025-12-11  15:10:00",
    "strategy": "text",
    "diagnosis": "Cardiovascular / Pulmonary: Flu",
    "confidence": 0.82,
    "explanation": "Cough and cold are characteristic symptoms of
       the flu."
  }
}
```

- **Response Body (error):**

```
{
  "error": "Invalid token"
}
```

- **Status Codes:**
  - 200 OK – analysis completed
  - 400 Bad Request – invalid/missing data
  - 401 Unauthorized – invalid or expired JWT
- **Flow Notes:**
  - Gateway validates JWT via Authentication Service
  - Gateway sends data to Data Processing Service
  - Data Processing Service saves processed data in its database and returns `processed_data_id`
  - Explainable AI retrieves data and executes model
  - Diagnosis saved in reports database
  - Audit Service notified asynchronously

2.4 GET /reports

- **Purpose:** Retrieve all reports for the doctor, optionally filtered by patient
- **Authentication:** JWT required (`Authorization: Bearer <JWT>`)
- **Query Parameters (optional):** `patient_cf`
- **Response Body:**

```
{
  "message": "Report(s) retrieved successfully",
  "reports": [
      {
          "id": 1,
          "doctor_id": 3,
          "patient_hashed_cf": "hash",
          "processed_data_id": 5,
          "created_at": "2025-12-11 20:00:00",
          "strategy": "text",
          "diagnosis": "Cardiovascular / Pulmonary: Flu",
          "confidence": 0.82,
          "explanation": "Cough and cold are characteristic
             symptoms of the flu."
      },
      {
          "id": 2,
          "doctor_id": 1,
          "patient_hashed_cf": "hash2",
          "processed_data_id": 4,
          "created_at": "2025-12-12  10:30:00",
          "strategy": "signal",
          "diagnosis": "Regular ECG",
          "confidence": 0.90,
          "explanation": "No signs of disease based on the ECG
             provided."
      },
```

```
        ...
    ]
}
```

- **Status Codes:**
  - 200 OK – reports retrieved
  - 401 Unauthorized – invalid JWT
  - 404 Not Found – no reports found
- **Flow Notes:**
  - Gateway validates JWT and extracts `doctor_id`
  - Calls Explainable AI to query reports database with optional `hashed_patient_cf`

3 Internal Endpoints

Authentication Service exposes POST /validate to verify JWT and /register, /login endpoints for login and registration. Data Processing Service exposes POST /process to process raw data and GET /data/processed-data-id to retrieve processed data. Explainable AI Service exposes POST /analyse for performing diagnosis and GET /reports[?patient_id=id] to retrieve reports. Audit Service exposes POST /log to record events from other services.

3.1 Authentication Service

POST authentication/register

This internal endpoint is invoked by the Gateway when a registration request is received. It creates a new doctor account, hashes the password, stores the record in the `doctors` table, and logs the operation.

Example request:

```
{
  "name": "Mario",
  "surname": "Rossi",
  "email": "mario.rossi@email.com",
  "password": "RossiMarioPassword25"
}
```

Example response:

```
{
  "message": "Doctor registered successfully"
  "doctor_id": 1,
}
```

If the email already exists, the service returns a 400 status with an error message.

POST authentication/login

This endpoint validates credentials during login. If the email and password match an existing record, the service generates and returns a JWT.

Example request:

```
{
  "email": "mario.rossi@mail.com",
  "password": "RossiMarioPassword25"
}
```

Example successful response:

```
{
  "message": "Doctor logged in successfully"
  "token": "jwt_token"
}
```

If authentication fails, it returns a 401 error with a corresponding message, and the event is logged.

POST authentication/validate

This endpoint verifies the validity of a JWT sent by the Gateway. The request must include the token in JSON form. If the token is valid, the service returns the associated doctor identifier. Otherwise, it returns an error.

Example request:

```
{
  "token": "jwt_token"
}
```

Example successful response:

```
{
  "message": "Token validated successfully"
  "doctor_id": id
}
```

Example error response:

```
{
  "error": "Invalid or expired token"
}
```

Status codes include 200 for valid tokens, 400 for missing or malformed tokens, and 401 for invalid or expired tokens. After every validation attempt, the Authentication Service notifies the Audit Service about the outcome to ensure traceability.

3.2 Audit Service

The Audit Service is responsible for recording and retrieving logs from all microservices. Its endpoints follow a structured JSON format and operate asynchronously. It implements an Observer Pattern that decouples monitoring concerns from core functionalities.

POST /audit/log

This endpoint receives log entries from other services. Each log includes the service name, the type of event, a description, and optional identifiers related to doctors, processed data, or reports.

Example request:

```
{
  "service": "authentication",
  "event": "register_success",
  "description": "Doctor registered successfully",
  "doctor_id": 1,
  "patient_hashed_cf": "null",
  "report_id": "null",
  "data_id": "null"
}
```

Example response:

```
{
  "message": "Log created successfully",
  "log_id": 100,
  "created_at": "2025-11-25 12:00:00"
}
```

Typical status codes include 201 when logs are successfully written and 400 for malformed payloads.

GET /audit/logs

This endpoint allows internal retrieval of all stored logs. It supports optional query parameters such as `service`, `event`, `doctor_id` or `patient_hashed_cf` to filter results.

Example response:

```
{
    "message": "Log(s) retrieved successfully",
    "logs": [
        {
            "id": 1,
            "service": "authentication",
            "event": "register_success",
            "description": "Doctor registered",
            "doctor_id": 1
            "patient_hashed_cf": null,
            "report_id": null,
            "data_id": null,
            "created_at": "2025-12-09 11:30:00"
        },
        ...
    ]
}
```

If no logs match the query, the service returns an empty array with status 200. In case of internal errors, a 500 error is produced with a diagnostic message.

Input Validation Tables

POST /register

Field	Type	Constraints
name	string	Required, min 2 chars, whitespace stripped
surname	string	Required, min 2 chars, whitespace stripped
email	string	Required, valid email format
password	string	Required, 8-100 chars, whitespace stripped

POST /login

Field	Type	Constraints
email	string	Required, valid email format
password	string	Required, min 1 char

POST /analyse

Field	Type	Constraints
strategy	string	Required, "text"/"img_rx"/"img_skin"/"numeric"/"signal"
patient_hashed_cf	string	Required, non-empty
raw_data	string	Required, serialized JSON or Base64

GET /reports

Parameter	Type	Constraints
patient_hashed_cf	string	Optional, non-empty if provided

POST /audit/log

Field	Type	Constraints
service	string	Required, 1-50 chars
event	string	Required, 1-50 chars
description	string	Required, 1-100 chars
doctor_id	int	Optional,
patient_hashed_cf	string	Optional, 1-255 chars
report_id	int	Optional
data_id	int	Optional

4 Database

The persistence layer of the system is built upon a relational database to ensure structured data storage and referential integrity.

4.1 PostgreSQL Architecture

The chosen database engine is PostgreSQL. To balance the architectural principles of microservice isolation with the practical need for deployment simplicity in a testing environment, a shared database pattern was adopted. While there is a single physical database instance, the architecture maintains a logical separation of concerns. Each

microservice is responsible for a distinct set of entities, simulating the boundaries of a distributed system.

4.2 Tables Schema

The schema is composed of four primary tables, with some logical relationships among them.

Listing 1: PostgreSQL Schema Definition

```
CREATE TABLE IF NOT EXISTS doctors (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    hashed_password VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS processed_data (
    id SERIAL PRIMARY KEY,
    type VARCHAR(20) NOT NULL,
    data TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP NOT
        NULL
);

CREATE TABLE IF NOT EXISTS reports (
    id SERIAL PRIMARY KEY,
    doctor_id INTEGER NOT NULL,
    patient_hashed_cf VARCHAR(255) NOT NULL,
    processed_data_id INTEGER NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP NOT
        NULL,
    strategy VARCHAR(255) NOT NULL,
    diagnosis VARCHAR(255) NOT NULL,
    confidence FLOAT NOT NULL,
    explanation TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS logs (
    id SERIAL PRIMARY KEY,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP NOT
        NULL,
    service VARCHAR(50) NOT NULL,
    event VARCHAR(50) NOT NULL,
    description VARCHAR(100) NOT NULL,
    doctor_id INTEGER,
    patient_hashed_cf VARCHAR(255),
    data_id INTEGER,
    report_id INTEGER
);
```

5 Deployment

Each backend microservice (Authentication, Audit, XAI, Data Processing, Gateway) shares a common build strategy defined in the `Dockerfile`. The base image utilized is

`python:3.11-slim`.

5.1 Dockerfile

To accelerate build times and leverage Docker's layer caching mechanism, the build process is split into two distinct stages:

1. **Dependency installation:** The `requirements.txt` file is copied, and `pip install` is executed.
2. **Source code copy:** The application source code is copied only after the dependencies are installed.

This strategy ensures that if the source code changes but the dependencies remain the same, Docker reuses the cached dependency layer, reducing the build time during iterative development.

5.2 Docker Compose

Service orchestration is defined in the `docker-compose.yaml` file, which describes a complete environment composed of eight interconnected services:

- **External services**:
  - `postgres`: The persistent relational database (PostgreSQL), initialized via scripts mapped to the `/docker-entrypoint-initdb.d` volume.
  - `pgadmin`: A web-based administration interface for database management, accessible on port 8080.
- **Backend services**: All backend services are configured with bind-mounts (e.g., `./backend/gateway:/app`) and run with the `--reload` flag to enable hot-reloading.
  - `auth_service`: Identity management (Host Port: 8000).
  - `audit_service`: Asynchronous logging (Host Port: 8001).
  - `gateway`: System entry point (Host Port: 8002).
  - `xai_service`: Explainable AI engine (Host Port: 8003).
  - `data_service`: Data processing unit (Host Port: 8004).
- **Frontend**:
  - `ui`: A React application served on port 3000, dynamically configured via the `REACT_APP_API_URL` environment variable.

5.3 Environment Configuration

Adhering to security best practices, sensitive configuration is strictly decoupled from the code. The `.env` file centralizes environment variables, including database credentials (`POSTGRES_USER`, `POSTGRES_PASSWORD`), security keys (`SECRET_KEY`, `GOOGLE_API_KEY`), and internal service discovery URLs.

5.4 Automated Pipeline

To streamline testing and system startup, automation scripts have been provided for both Windows (`run.bat`) and Unix-like systems (`run.sh`).