**Documentation**

**Project**: Exchange Rate Service – Backend

**Scope**: Manage currencies and exchange rates, sync with external provider (e.g., OANDA), and provide REST APIs for querying stored & live rates.

**Main Branch:** main

**1. Introduction**

**Purpose**

Provide a RESTful backend that manages currencies and exchange rates, both from a local database and live from external APIs.

**Objectives**

* Standards-compliant REST APIs.
* Clear, extensible design using Clean Architecture.
* Scheduled background jobs to sync exchange rates.
* Transaction-safe persistence of rates.

**2. Clean Architecture Overview**

**2.1 Principles**

* **Dependency Rule**: Inner layers (domain) do not depend on outer layers.
* **Separation of Concerns**:
  + Controllers = HTTP I/O
  + Application Services = Orchestrate use cases
  + Domain = Core business rules
* **High Cohesion, Low Coupling**.

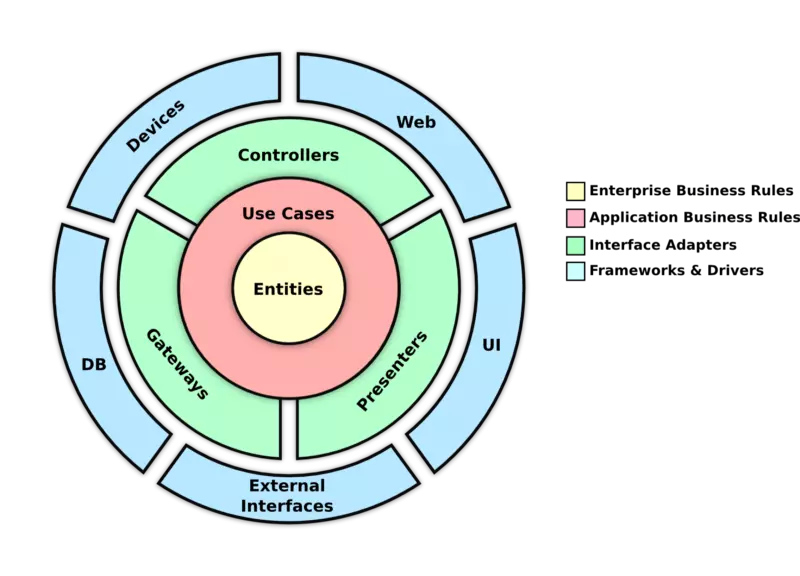
**2.2 Layers**

* **API Layer** (Controllers): accepts HTTP requests, validates input, maps DTOs -> domain models.
* **Application Layer** (Services / Use Cases): manage currency, sync exchange rates, orchestrate transactions, invoke domain logic.
* **Domain Layer** (Entities & Business Rules): core entities: Currency, ExchangeRate.
* **Infrastructure Layer** (Adapters): JPA repositories, entities, schema, external API client (OANDA), configuration..

**Project Structure:**

| project  │  ├── api # Controllers, request/response DTOs  ├── application # Services, use case orchestration  ├── domain # Entities, value objects, domain services  ├── infra # JPA repositories, mappers, DB config, adapters  └── pom.xml |
| --- |

**Diagrams**:



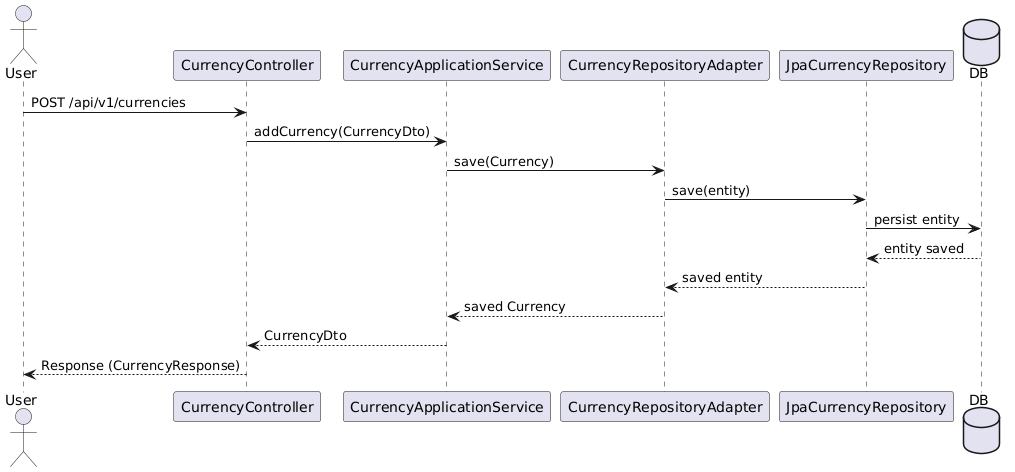
**3. System Flow Diagrams**

**3.1 Currency Management (CRUD – ex: Add Currency)**

**Scenario**: Admin registers a currency via POST /api/admin/currencies.

**Flow**:

1. Controller validates request → maps to Currency.
2. Application service checks uniqueness.
3. Calls repository to persist currency.
4. Return created resource.

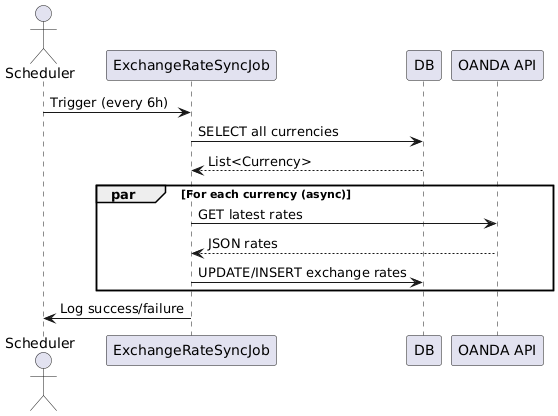


**3.2 Exchange Rate Sync – Scheduled Job**

**Scenario**: System runs job to fetch latest rates from OANDA API*.*

**Flow**:

1. Selects all currencies from DB.
2. Calls **OANDA API** asynchronously for each base currency.
3. Maps responses → domain ExchangeRate.
4. Saves/updates rates into DB
5. Logs success/failure.

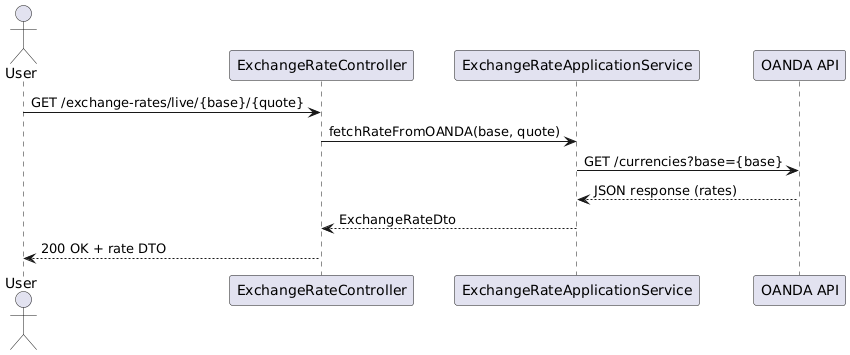


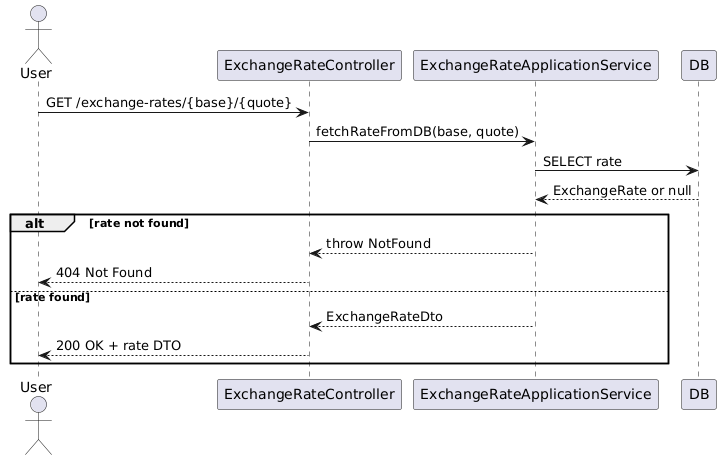
**3.3 Get Exchange Rate – Query API**

**Scenario**: *APIs for exchange rates (DB + Live from OANDA)*

**Flow**:

1. Controller validates path params.
2. Application service fetches rate from DB or from OANDA.
3. If not found → return 404.
4. Return DTO with rate info.





**4. Data Model**

**4.1 Entities (Fields)**

* **Currency:** code, name, createdAt, updatedAt
* **ExchangeRate:** id, baseCurrency, targetCurrency, averageBid, averageAsk, updatedTime

### 4.2 Relationships (ERD)

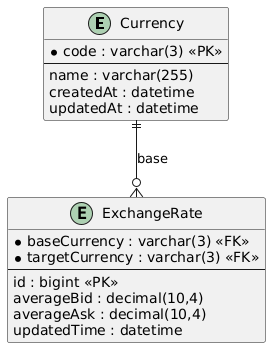
This project uses an **H2 in-memory** database

### **One** Currency **→ Many** ExchangeRate (as base)**.**

### **ExchangeRate composite key =** (baseCurrency, targetCurrency)

**4.3 Constraints & Validation**

* Currency code must be unique (e.g., "USD").Product.stock >= 0 (enforced by business logic + DB checks if applied)
* ExchangeRate (base, target) unique
* averageBid, averageAsk > 0



**5. API Design**

**5.1 Currency Endpoints**

| **Method** | **Path** | **Purpose** |
| --- | --- | --- |
| GET | /api/v1/currencies | Get all currencies |
| GET | /api/v1/currencies/{code} | Get currency by code |
| POST | /api/v1/currencies | Add a new currency |
| PUT | /api/v1/currencies/{code} | Update an existing currency |
| DELETE | /api/v1/currencies/{code} | Delete a currency by code |

**5.2 Exchange Rate Endpoints**

| **Method** | **Path** | **Purpose** |
| --- | --- | --- |
| GET | /api/v1/exchange-rates/{baseCurrency} | Get exchange rates by base (DB) |
| GET | /api/v1/exchange-rates/{baseCurrency}/{quoteCurrency} | Get exchange rate by base and quote (DB) |
| GET | /api/v1/exchange-rates/live/{baseCurrency} | Get live exchange rates by base (OANDA) |
| GET | /api/v1/exchange-rates/live/{baseCurrency}/{quoteCurrency} | Get live exchange rate by base and quote (OANDA) |

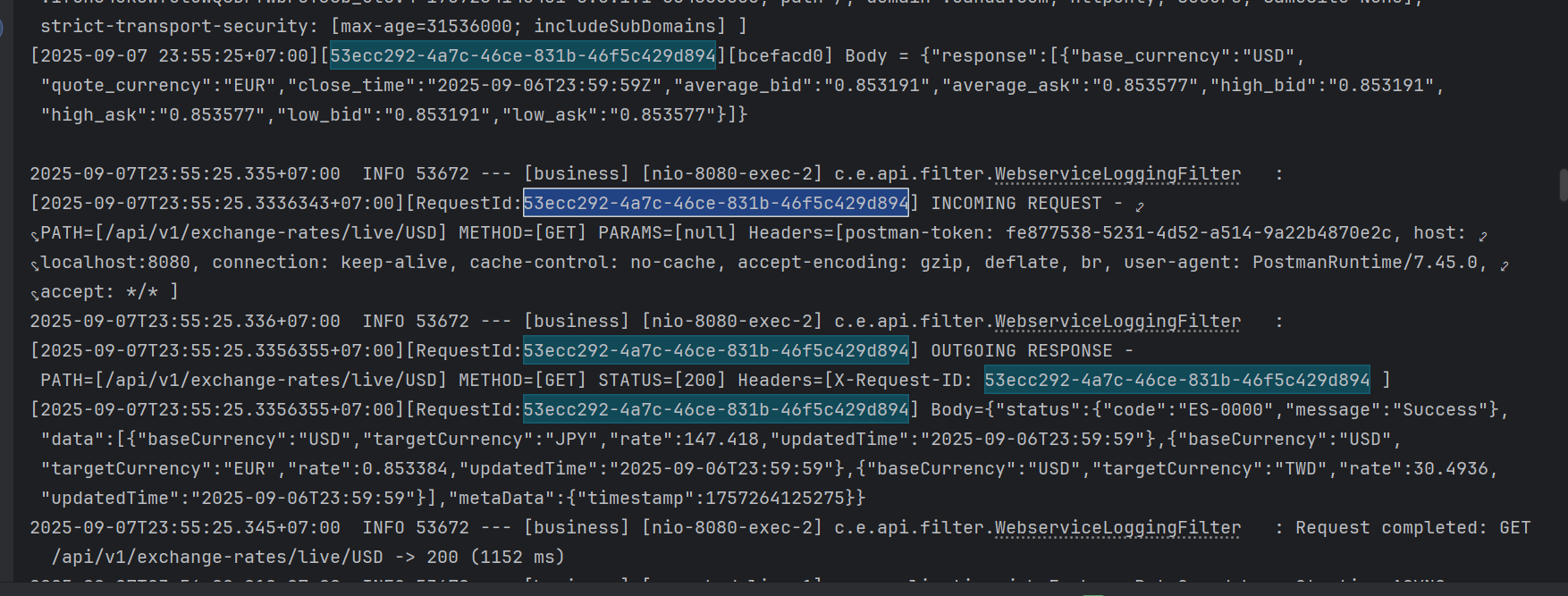
**5.3 Error Model (standardized)**

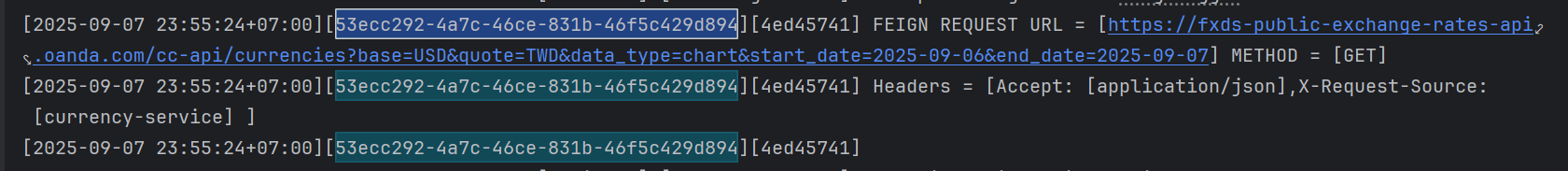
* **404 Not Found →** Currency or rate missing.
* **400 Bad Request** → Invalid code/format
* **500 Internal Server Error →** OANDA API failure.

**6. Extra (Plus) Requirements**

**6.1. Request & Response Logging**

* **X-Request-ID**: generated for each incoming API request to trace the full flow across the system.
* **Trace-ID**: generated for outbound calls (e.g., OANDA) to trace external API interactions.
* **REST APIs**: log request and response bodies using OncePerRequestFilter.
* **External APIs**: log request and response bodies using WebClient filters.
* **Purpose**: enables end-to-end tracing, debugging, auditing, and monitoring.



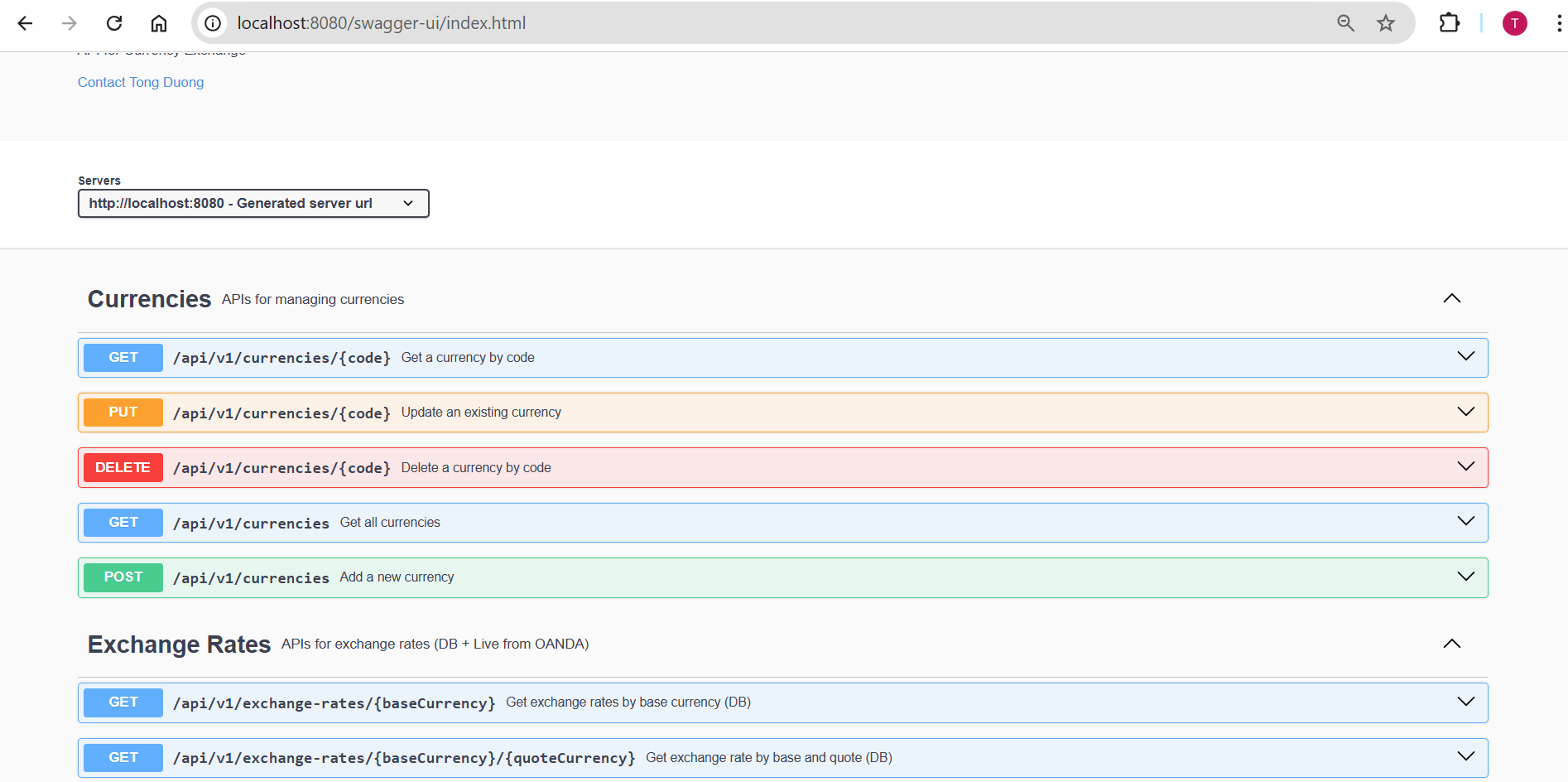




**6.2. Swagger UI**

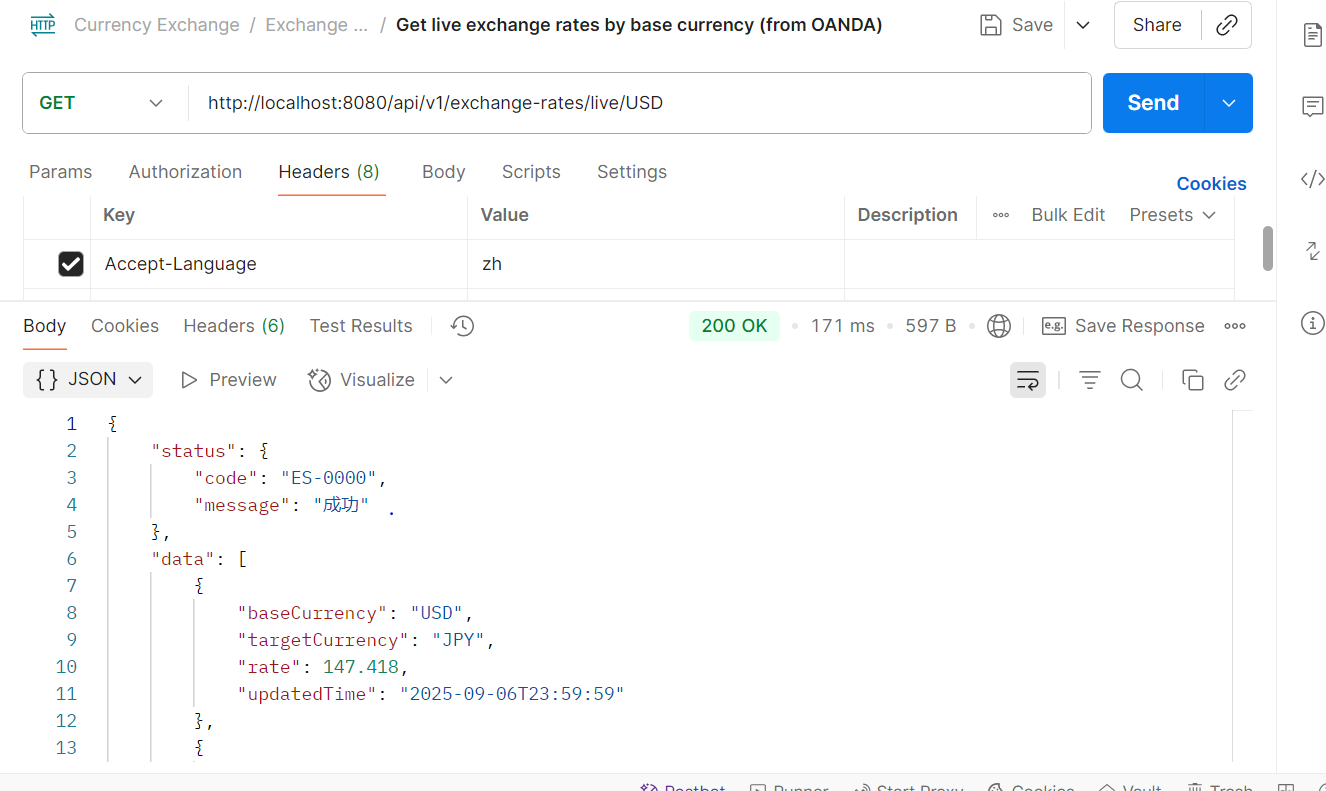
**Path:** [*http://localhost:8080/swagger-ui/index.html*](http://localhost:8080/swagger-ui/index.html)

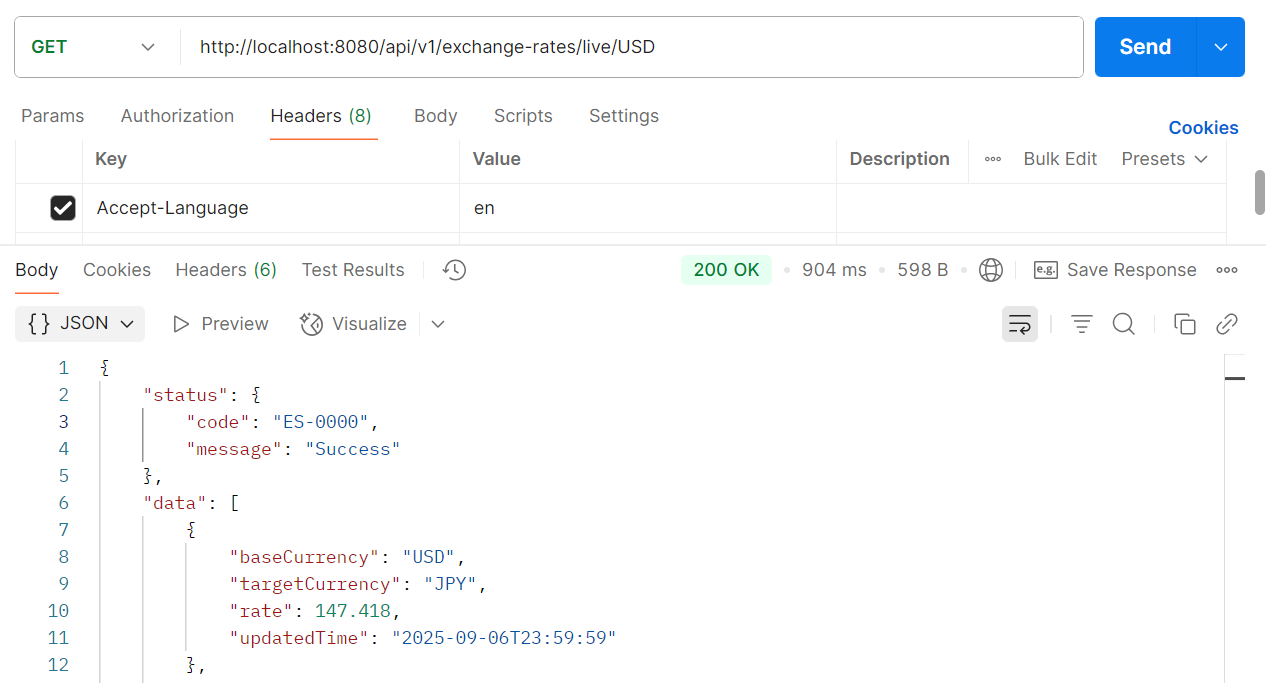
* Integrate springdoc-openapi-ui.
* Auto-generate OpenAPI docs from annotations.
* Accessible at /swagger-ui.html for easy exploration and testing.

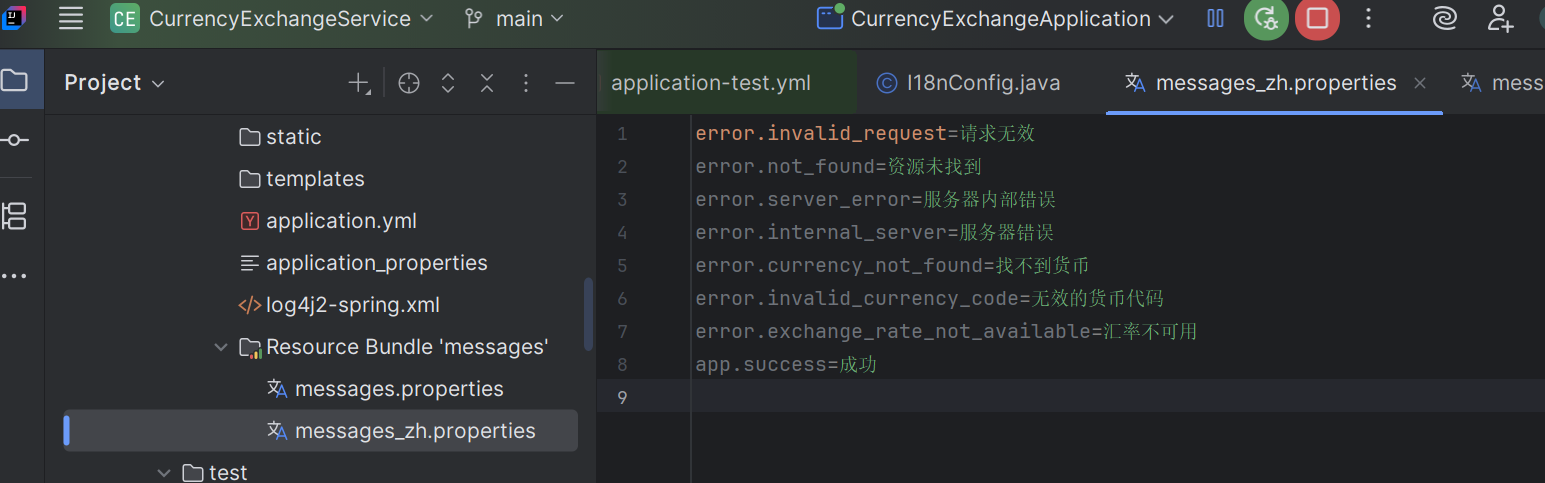


**6.3. i18n (Internationalization)**

* Provide localized messages via messages.properties (default), messages\_zh.properties (Chinese).
* Error and success messages returned according to Accept-Language header.







**6.4.** **Global Error Handling**

* Centralized exception handling with **@ControllerAdvice + @ExceptionHandler**.  
  Unified API error format:

| {  "status": {  "code": "DM-5000",  "message": "Server error"  },  "data": null,  "metaData": {  "timestamp": 1757264941801  }  } |
| --- |

* Consistent handling of validation errors, business rule violations, and system errors.



**6.5. Format Response**

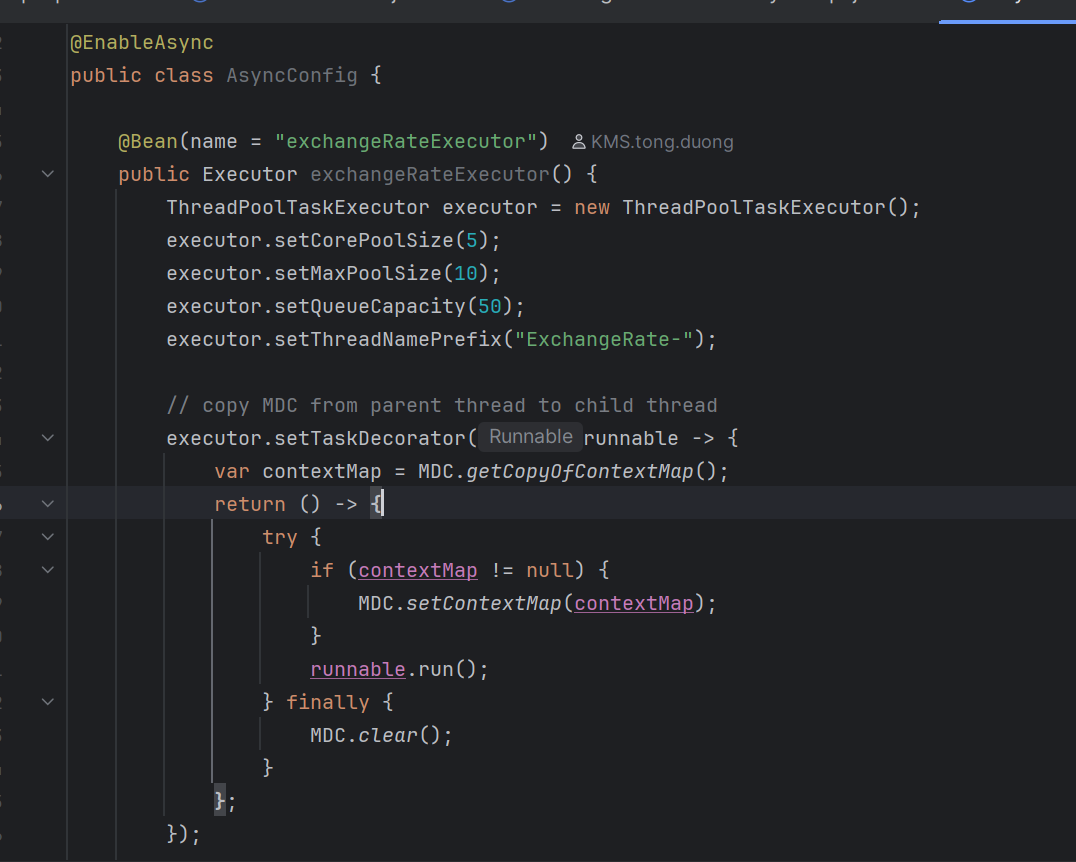
The API response should follow a standardized structure that includes:

* **code** – application-specific status code
* **status** – success or failure indicator
* **message** – human-readable description
* **meta** – optional metadata (e.g., pagination info, request id)

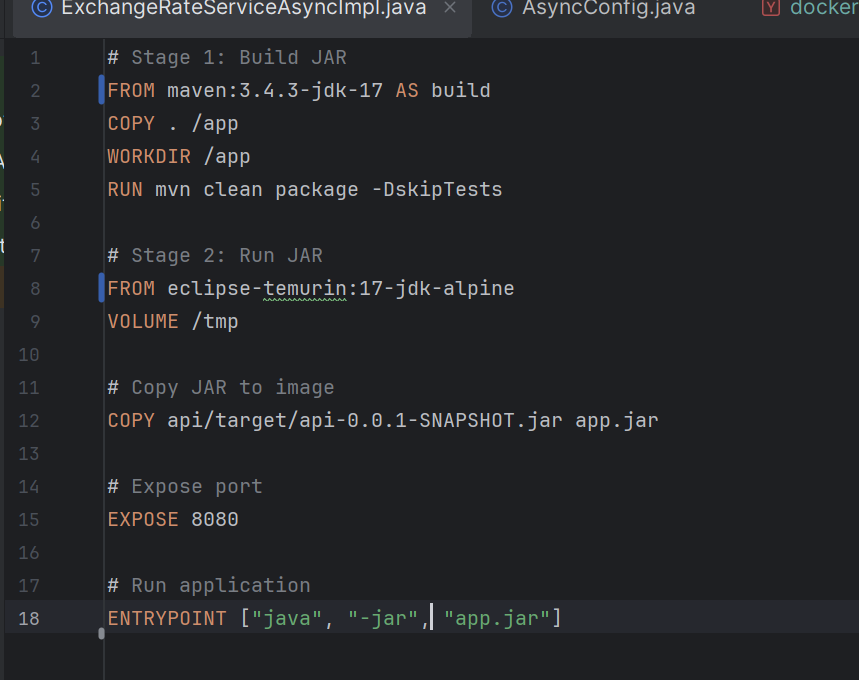
| {  "status": {  "code": "DM-5000",  "message": "Server error"  },  "data": **null**,  "metaData": {  "timestamp": 1757264941801  } } |
| --- |

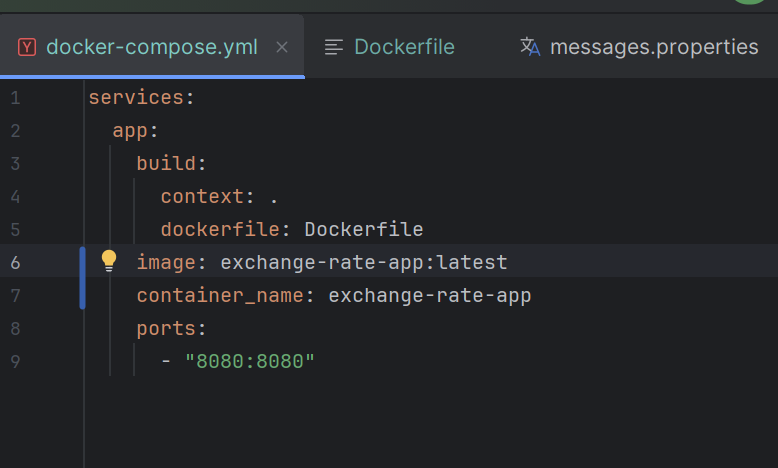
| {  "status": {  "code": "ES-0000",  "message": "Success"  },  "data": [  {  "baseCurrency": "USD",  "targetCurrency": "JPY",  "rate": 147.418,  "updatedTime": "2025-09-06T23:59:59"  },  {  "baseCurrency": "USD",  "targetCurrency": "EUR",  "rate": 0.853384,  "updatedTime": "2025-09-06T23:59:59"  },  {  "baseCurrency": "USD",  "targetCurrency": "TWD",  "rate": 30.4936,  "updatedTime": "2025-09-06T23:59:59"  }  ],  "metaData": {  "timestamp": 1757265077113  } } |
| --- |

**6.6 Async Job Executor**

Configure **ThreadPoolTaskExecutor + @Async** to run jobs in parallel without blocking.  


**6.7 Docker Compose**





**7. Run & Usage**

**Requirements**

* JDK 17
* Maven 3.8+
* Docker

There are 3 ways to run:

1. **Run (Maven, module api)**

| # Windows .\mvnw.cmd -f .\api\pom.**xml** **spring-boot**:run  # macOS/Linux ./mvnw -f ./api/pom.**xml** **spring-boot**:run |
| --- |

1. **Package & Docker (optional)**

| ./mvnw -DskipTests package  docker build -t exchange-rate-app .  docker run --rm -p 8080:8080 exchange-rate-app |
| --- |

1. **Docker Compose**

| docker compose up -d --build |
| --- |

**Base URL**: [*http://localhost:8080*](http://localhost:8080)

**Test (run all tests in api)**

| ./mvnw clean test |
| --- |