## 📝 Design Options Explanation

### 🎯 Project Objective

Develop a **digital wallet service** that enables the creation and management of financial wallets while ensuring **integrity** and **consistency** of operations, even under failure scenarios or duplicate requests. The implementation uses **idempotency** to meet resilience and consistency requirements.
The **H2 database** was chosen to facilitate development, quick testing, and local application execution.

### ⚙️ Functional Requirements

#### 1. Digital Wallet Creation

- **Description**: The system must allow creating a wallet associated with a unique user.
- **Implementation**:
    - **Endpoint**: POST /api/v1/wallets
    - **Input Parameters**:
        - userId (User ID)
        - initialBalance (Initial Balance)
    - **Response**: Wallet details, including walletId, userId, and balance.

#### 2. Idempotency Guarantee

- **Description**: The system must prevent duplicate wallet creation when multiple requests with the same **Idempotency-Key** are sent.
- **Implementation**:
    - **Header**: Idempotency-Key is used as a unique key to identify the operation.
    - **Storage**:
        - The key is persisted in the **idempotency_keys** table within the **H2 database**.
    - **Verification**: Before processing a request, the system checks if the key exists. If it does, the original operation's result is returned.
- **Result**: Ensures consistency and prevents duplication.

### 🚀 Non-Functional Requirements

#### 1. Availability

- **Requirement**: The service must always be available in a local development environment.
- **Implementation**:
    - Use **H2 Database** in **in-memory mode** to simplify development.
    - **Spring Boot** ensures seamless initialization of the embedded H2 database.

#### 2. Consistency

- **Requirement**: Wallet creation must remain consistent, even with repeated attempts.
- **Implementation**:
    - The **H2 database** supports **ACID transactions**.
    - The **idempotency_keys** table ensures consistency by storing the key and the operation result.

#### 3. Performance

- **Requirement**: Operations must be fast during development and testing.
- **Implementation**:
    - **H2**, operating in memory, allows extremely fast access during local execution.

**4. Testability**

- **Requirement**: Facilitate unit and integration tests.
- **Implementation**:
    - H2 is ideal for tests as it is **lightweight**, **fast**, and requires no external installation.
    - During tests, the database can be initialized and restarted easily.

**5. Simplicity**

- **Requirement**: Simplify the setup and local development process.
- **Implementation**:
    - **H2** is configured in application.properties to initialize automatically when running Spring Boot.

### 🛠️ Design Decisions

**1. H2 Database Usage**

- H2 was chosen for:
    - **Ease of use** during local development.
    - **Zero configuration**: Embedded database in Spring Boot.
    - **In-memory mode**: No files are generated, and the database restarts on every execution.

**2. Idempotency Table**

- A helper table **idempotency_keys** was created to store:

| Column | Type | Description |
|---|---|---|
| key | VARCHAR | Idempotency-Key (unique key). |
| response | TEXT | Result of the operation. |
| created_at | TIMESTAMP | Record creation timestamp. |

**3. Stateless API**

- The service was designed to be **stateless**, ensuring that each request is processed independently.

### 📊 How the Implementation Meets Requirements

| Requirement | Met By |
|---|---|
| Wallet Creation | RESTful endpoint POST /api/v1/wallets |
| Idempotency | Use of Idempotency-Key persisted in the H2 DB |
| Availability | H2 in-memory database + Spring Boot |
| Consistency | H2 Database with ACID transactions |
| Performance | Fast access to H2 database in memory |
| Testability | Lightweight, restartable database for tests |

### 🔗 H2 Database Configuration

In the application.properties file, H2 is configured in **in-memory mode**:

```
# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:digitalwallet
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

### Accessing the H2 Console

1. Access the console at:

   **http://localhost:8080/h2-console**

2. Configure the console:
   - **JDBC URL**: jdbc:h2:mem:digitalwallet
   - **Username**: sa
   - **Password**: *(leave blank)*

## 🛠️ Swagger Integration

### Why Swagger?

- Facilitates REST API documentation.
- Allows testing endpoints directly through a graphical interface.
- Generates interactive documentation following **OpenAPI** standards.

### Swagger Configuration

1. **Add Dependencies**
   Add the following dependency to your pom.xml file:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

2. **Access Swagger Documentation**
   After starting the application, access the Swagger UI at:

   **http://localhost:8080/swagger-ui.html**

### Example: Wallet Creation Endpoint

- **Method**: POST
- **URL**: /api/v1/wallets
- **Headers**:

| Key | Value |
|---|---|
| Idempotency-Key | 123e4567-e89b-12d3-a456-426614174000 |

- **Body**:

```
{
  "owner": "Joao"
}
```

- **Response**:

```
{
    "createdAt": "2024-12-17T20:17:28.879872",
    "updatedAt": "2024-12-17T20:17:28.880872",
    "id": 1,
    "owner": "Joao",
    "balance": 0,
        "version": 0
}
```

### 📊 Benefits of Swagger Integration

- **Automatic Documentation**: All endpoints are automatically documented.
- **Interactive Testing**: Users can test endpoints directly in the browser.
- **OpenAPI Standardization**: Ensures the API follows market standards.
- **Easy Sharing**: The documentation can be shared with developers and stakeholders.

### 🚀 Execution

1. **Start the Application**:

   mvn spring-boot:run

2. **Access Swagger Documentation**:

   http://localhost:8080/swagger-ui.html

### 🚀 Future Improvements

1. Add support for **production databases** (PostgreSQL, MySQL).
2. Add **Redis caching** for idempotency key lookups.
3. Implement **JWT authentication**.
4. Improve monitoring using tools like **Prometheus** or **Spring Actuator**.