Build a Currency Rate Service

Objective:

Develop a service with an endpoint that retrieves currency rates from a given API (find a mock with details here: https://github.com/illenko/currencies-mocks), combines it, and returns in a specified format.

API Spec:

- **GET /currency-rates**
  Response:

Success (200): JSON array of currency rates

JavaScript

```
{
    "fiat": [
        {
            "currency": "USD",
            "rate": 123.33
        },
        {
            "currency": "EUR",
            "rate": 1232.22
        }
    ],
    "crypto": [
        {
            "currency": "BTC",
            "rate": 1232.22
        },
        {
            "currency": "ETH",
            "rate": 234.56
        }
    ]
}
```

Data not retrieved from the crypto endpoint and does not exist in DB (200):

```javascript
{
     "fiat": [
        {
            "currency": "USD",
            "rate": 123.33
        },
        {
            "currency": "EUR",
            "rate": 1232.22
        }
    ],
    "crypto": []
}
```

Data not retrieved from any endpoint and no records in DB (200):

```javascript
{
    "fiat": [],
    "crypto": []
}
```

Request Flow for `/currency-rates` Endpoint

1. **Request Received:**
    - The service receives a GET request to the `/currency-rates` endpoint.
2. **Fetch Data from Mock APIs:**
    - The service makes HTTP requests to the two mock APIs:
        - **Fiat Currency API:** Fetches fiat currency rates.
        - **Cryptocurrency API:** Fetches cryptocurrency rates.
    - The requests are typically made using non-blocking asynchronous techniques (e.g., WebClient in Spring WebFlux).
3. **Handle API Responses:**
    - **Successful Response:**
        - If both API requests are successful, the fetched data is parsed and stored in the database.
        - The stored data is then returned as the API response.
    - **Error or Timeout:**

        - If either API request fails or times out, the service attempts to retrieve the latest data from the database (for data expected to be retrieved from this endpoint).
        - If the data is found in the database, it's used to build a response.
        - If the data is not found in the database, an empty array can be returned.
4. **Return Response:**
    - The service constructs the final JSON response, combining the fiat and cryptocurrency data.
    - The response is sent back to the client with a 200 OK status code.

Scenario: Error from Fiat API, Success from Crypto API

1. Request Received:

- The service receives a GET request to the `/currency-rates` endpoint.

2. Fetch Data from Mock APIs:

- **Fiat Currency API:**
    - The service sends a request to the Fiat Currency API.
    - The API returns an error response (e.g., 500 Internal Server Error).
- **Cryptocurrency API:**
    - The service sends a request to the Cryptocurrency API.
    - The API returns a successful response with cryptocurrency rates.

3. Handle API Responses:

- **Fiat Currency API:**
    - The service handles the error and logs it for debugging.
    - It then attempts to retrieve the latest fiat currency rates from the database.
    - If the data is found, it's used to construct the response for the fiat currency section.
    - If the data is not found, an empty array is used for the fiat currency section.
- **Cryptocurrency API:**
    - The fetched cryptocurrency rates are parsed and stored in the database.
    - The data is used to construct the response for the cryptocurrency section.

4. Return Response:

- The service combines the fiat and cryptocurrency data, even if the fiat data is empty or retrieved from the database.
- The final JSON response is:

JavaScript

```
{
    "fiat": [],
    "crypto": [
        {
            "currency": "BTC",
            "rate": 12345.67
        },
        {
            "currency": "ETH",
            "rate": 234.56
        }
    ]
}
```

- The response is sent back to the client with a 200 OK status code.

Optional Enhancements:

- **Unit Tests:** Write unit tests to verify the functionality of the service.
- **Integration Tests:** Set up integration tests to validate the interaction with the API and database.
- **Logging:** Implement logging to capture important events and debug issues.

Expected Output:

Link to the project on Github with a short Readme present.

Hints:

- You can use the `WebClient` class in Spring Webflux to make HTTP requests to the API.
- To handle errors and ensure a 200 response, consider using `onErrorResume` and `defaultIfEmpty` operators.

Note:

While the preferred stack is Spring Boot Webflux, Spring Data R2DBC, and PostgreSQL, candidates with experience in other frameworks or technologies are welcome. The evaluation will focus on the candidate's ability to solve the problem and demonstrate their understanding of core concepts.