Virtual Bank System with Microservices, BFF, and WSO2 API Gateway

Project Goal

The primary goal of this 1-month internship project is to develop a simplified virtual banking system. This system will demonstrate key concepts of modern application architecture, including:

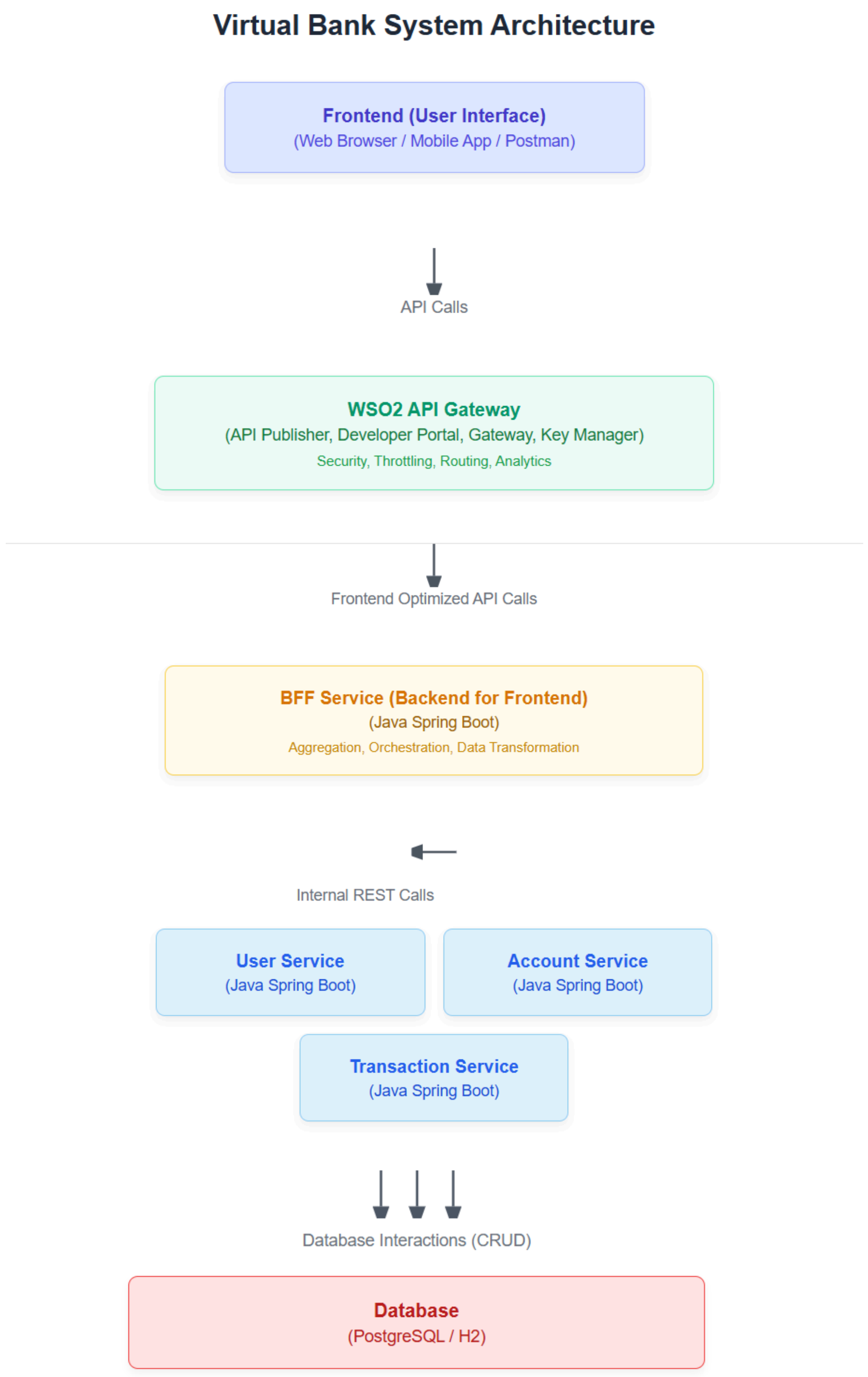
- **Java Microservices:** Independent, loosely coupled services handling specific banking functionalities.
- **Backend for Frontend (BFF) Pattern:** A dedicated aggregation layer optimized for a specific user interface.
- **WSO2 API Gateway:** Centralized management, security, and routing for all external API interactions.

This project will provide hands-on experience in designing, developing, and deploying a secure and scalable distributed system.

Core Architectural Concepts

1. **Microservices:** Each banking domain (e.g., Users, Accounts, Transactions, Logging) will be a separate Java Spring Boot microservice.
2. **Backend for Frontend (BFF):** A dedicated Spring Boot service that acts as an API aggregator and orchestrator for a specific frontend application. It simplifies complex calls from the UI by making multiple backend microservice calls and returning a consolidated response, reducing chattiness between the frontend and the external API Gateway.
3. **WSO2 API Gateway:** This will be the single entry point for all external consumers. It will handle:
   - **API Publishing and Discovery:** Defining and making APIs available to developers.
   - **Authentication and Authorization :** Securing APIs and controlling access based on user roles and permissions.
   - **Request/Response Transformations:** Modifying incoming and outgoing messages to suit backend service requirements or frontend consumption.
   - **Throttling and Rate Limiting:** Protecting backend services from overload and ensuring fair usage.
   - **Analytics and Monitoring**
   - **Routing requests:** Directing incoming API calls to the correct backend microservice or the BFF.

High-Level Architecture


Virtual Bank System Architecture
Frontend (User Interface)
(Web Browser / Mobile App / Postman)
API Calls
WSO2 API Gateway
(API Publisher, Developer Portal, Gateway, Key Manager)
Security, Throttling, Routing, Analytics
Frontend Optimized API Calls
BFF Service (Backend for Frontend)
(Java Spring Boot)
Aggregation, Orchestration, Data Transformation
Internal REST Calls
User Service
(Java Spring Boot)
Account Service
(Java Spring Boot)
Transaction Service
(Java Spring Boot)
Database Interactions (CRUD)
Database
(PostgreSQL / H2)

Microservices Breakdown

For each microservice, assume a basic Spring Boot application with a REST controller, a service layer for business logic, and a repository layer.

1. User Service

- **Purpose:** Manages user authentication, registration, and basic profile information.
- **Key Responsibilities:**
  - Securely store user credentials (hashed passwords).
  - Handle user registration and login workflows.
  - Manage user profile details.
- **Endpoints:**
  - POST /users/register: Registers a new user.
    - **Request Body Example (JSON):**
      ```
      {
        "username": "john.doe",
        "password": "securePassword123",
        "email": "john.doe@example.com",
        "firstName": "John",
        "lastName": "Doe"
      }
      ```
    - **Response Example (JSON - 201 Created):**
      ```
      {
        "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
        "username": "john.doe",
        "message": "User registered successfully."
      }
      ```
    - **Error Response (409 Conflict if username/email exists):**
      ```
      {
        "status": 409,
        "error": "Conflict",
        "message": "Username or email already exists."
      }
      ```

- POST /users/login: Authenticates a user.
  - **Request Body Example (JSON):**
    ```
    {
      "username": "john.doe",
      "password": "securePassword123"
    }
    ```
  - **Response Example (JSON - 200 OK):**
    ```
    {
      "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
      "username": "john.doe"
    }
    ```
  - **Error Response (401 Unauthorized if invalid credentials):**
    ```
    {
      "status": 401,
      "error": "Unauthorized",
      "message": "Invalid username or password."
    }
    ```
  - ★ NOTE: password will be saved in the DB hashed

- GET /users/{userId}/profile: Retrieves a user's basic profile details. Requires authentication/authorization header
  - **Response Example (JSON - 200 OK):**
    ```
    {
      "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
      "username": "john.doe",
      "email": "john.doe@example.com",
      "firstName": "John",
      "lastName": "Doe"
    }
    ```
  - **Error Response (404 Not Found):**
    ```
    {
      "status": 404,
      "error": "Not Found",
      "message": "User with ID a1b2c3d4-e5f6-7890-1234-567890abcdef not
    found."
    ```

```
}
```

2. Account Service

- **Purpose:** Manages bank accounts for users, including account creation, retrieval, and balance inquiries.
- **Key Responsibilities:**
  - Create and manage unique bank accounts.
  - Maintain account balance and status.
  - Provide account details to authorized callers.
- **Endpoints:**
  - POST /accounts: Creates a new bank account for a specified user.
    - **Request Body Example (JSON):**
      ```
      {
        "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
        "accountType": "SAVINGS",
        "initialBalance": 100.00
      }
      ```
    - **Response Example (JSON - 201 Created):**
      ```
      {
        "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
        "accountNumber": "1234567890",
        "message": "Account created successfully."
      }
      ```
    - **Error Response (400 Bad Request if invalid type/balance):**
      ```
      {
        "status": 400,
        "error": "Bad Request",
        "message": "Invalid account type or initial balance."
      }
      ```
  - GET /accounts/{accountId}: Retrieves details of a specific bank account.
    - **Response Example (JSON - 200 OK):**
      ```
      {
        "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
        "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
        "accountNumber": "1234567890",
      ```

  "accountType": "SAVINGS",
  "balance": 100.00,
  "status": "ACTIVE",
  "createdAt": "2025-06-30T10:00:00Z"
}

- **Error Response (404 Not Found):**
```
{
  "status": 404,
  "error": "Not Found",
  "message": "Account with ID f1e2d3c4-b5a6-9876-5432-10fedcba9876
not found."
}
```

- GET /users/{userId}/accounts: Lists all accounts associated with a given user.
  - **Response Example (JSON - 200 OK):**
```
[
  {
    "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
    "accountNumber": "1234567890",
    "accountType": "SAVINGS",
    "balance": 100.00,
    "status": "ACTIVE"
  },
  {
    "accountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
    "accountNumber": "0987654321",
    "accountType": "CHECKING",
    "balance": 500.50,
    "status": "ACTIVE"
  }
]
```

  - **Error Response (404 Not Found):**
```
{
  "status": 404,
  "error": "Not Found",
  "message": "No accounts found for user ID
a1b2c3d4-e5f6-7890-1234-567890abcdef."
```

}

Scheduled Job: Inactivate Stale Accounts

- **Module:** `Account Service`
- **Purpose:** Automatically mark accounts as inactive if they have been idle for over 24 hours.

- **Description**

  A scheduled job runs within the **Account Service** to maintain account activity status. The job identifies and **inactivates any accounts** that meet the following criteria:

  - The account is currently marked as **active**.
  - The account's **latest transaction occurred more than one day ago**

  Once identified, the account status will be updated from `ACTIVE` to `INACTIVE`.

- **Schedule**
  - The job is executed **every 5 minutes**

3. Transaction Service

- **Purpose:** Handles financial transactions (deposits, withdrawals, transfers) and maintains transaction history.
- **Key Responsibilities:**
  - Record all financial movements.
  - Update account balances (via Account Service).
  - Provide detailed transaction history.
- **Endpoints:**
  - **POST /transactions/transfer/initiation**: Initiates a fund transfer between two accounts. By insert in db transaction initiated.
    - **Request Body Example (JSON):**
      ```
      {
        "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
        "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
        "amount": 30.00,
      ```

```
  "description": "Transfer to checking account"
}
```

- **Response Example (JSON - 200 OK):**

```
{
  "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
  "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
  "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
  "amount": 30.00,
  "description": "Transfer Initiated.",
  "timestamp": "2025-06-30T10:10:00Z"
}
```

- **Error Response (400 Bad Request for invalid accounts, 400 Bad Request for insufficient funds):**

```
{
  "status": 400,
  "error": "Bad Request",
  "message": "Invalid 'from' or 'to' account ID."
}
```

- **POST /transactions/transfer/execution:** execute fund transfer between two accounts. This will involve two updates to the Account Service (debit from one, credit to another) and update in db from initiated to (Success or Failed).
  - **Request Body Example (JSON):**

```
{
  "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
  "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
  "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
  "amount": 30.00,
  "description": "Transfer to checking account"
}
```

  - **Response Example (JSON - 200 OK):**

```
{
  "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
  "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
  "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
  "amount": 30.00,
  "description": "Transfer success.",
  "timestamp": "2025-06-30T10:10:00Z"
}
```

Error Response (400 Bad Request for invalid accounts, 400 Bad Request for insufficient funds):

```
{
  "status": 400,
  "error": "Bad Request",
  "message": "Invalid 'from' or 'to' account ID."
}
```

- GET /accounts/{accountId}/transactions: Retrieves the transaction history for a specific account.
  - **Response Example (JSON - 200 OK):**

    ```
    [
     {
       "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
       "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
       "amount": 50.00,
       "description": "Cash deposit",
       "timestamp": "2025-06-30T10:05:00Z",
       "currentBalance": 150.00
     },
     {
       "transactionId": "x1y2z3a4-b5c6-7890-1234-567890fedcba",
       "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
       "amount": -30.00,
       "description": "Transfer to checking account",
       "timestamp": "2025-06-30T10:10:00Z",
       "currentBalance": 120.00
     }
    ]
    ```

  - **Error Response (404 Not Found):**

    ```
    {
    ```

```
  "status": 404,
  "error": "Not Found",
  "message": "No transactions found for account ID
f1e2d3c4-b5a6-9876-5432-10fedcba9876."
}
```

4. BFF Service (Backend for Frontend)

- **Purpose:** Provides a simplified and optimized API layer for the frontend, aggregating data from multiple microservices and orchestrating complex workflows. This service will call the internal microservices directly.
- **Key Responsibilities:**
  - Aggregate data from multiple backend services into a single response.
  - Orchestrate complex business flows involving multiple microservices.
  - Transform data to suit the frontend's needs.
  - Handle internal service communication (e.g., using WebClient for asynchronous calls).
- **Endpoints:**
  - GET /bff/dashboard/{userId}: Fetches user profile, all associated accounts, and recent transactions for each account. This is a prime example of aggregation.
    - **Process:**
      1. Call GET /users/{userId}/profile from User Service.
      2. Call GET /users/{userId}/accounts from Account Service.
      3. For each account, asynchronously call GET /accounts/{accountId}/transactions from Transaction Service.
      4. Combine all responses into the final aggregated JSON.
    - **Response Example (JSON - 200 OK):**

```
{
  "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
  "username": "john.doe",
  "email": "john.doe@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "accounts": [
    {
      "accountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
      "accountNumber": "1234567890",
      "accountType": "SAVINGS",
      "balance": 120.00,
      "transactions": [
        { "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
"amount": 50.00, "type": "DEPOSIT", "description": "Cash deposit",
"timestamp": "2025-06-30T10:05:00Z" },
        { "transactionId": "x1y2z3a4-b5c6-7890-1234-567890fedcba",
"amount": -30.00, "type": "TRANSFER", "description": "Transfer to
checking account", "timestamp": "2025-06-30T10:10:00Z" }
      ]
    },
    {
      "accountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
      "accountNumber": "0987654321",
      "accountType": "CHECKING",
      "balance": 530.50,
      "transactions": [
        { "transactionId": "y9z8x7c6-v5b4-3210-9876-543210abcdef",
"amount": 30.00, "type": "TRANSFER", "description": "Received from
savings", "timestamp": "2025-06-30T10:10:00Z" }
      ]
    }
  ]
}
```

- **Error Response (404 Not Found for user, or 500 Internal Server Error for downstream service issues):**

```
{
  "status": 500,
  "error": "Internal Server Error",
  "message": "Failed to retrieve dashboard data due to an issue with
downstream services."
}
```

- POST /bff/transactions/transfer/initiation: to initiate transfer.
  - **Request Body Example (JSON):**

```
{
  "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
  "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
  "amount": 30.00,
  "description": "Transfer to checking account"
}
```

- **Response Example (JSON - 200 OK):**
  ```
  {
    "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
    "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
    "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
    "amount": 30.00,
    "description": "Transfer Initiated.",
    "timestamp": "2025-06-30T10:10:00Z"
  }
  ```

- **Error Response (400 Bad Request for invalid accounts, 400 Bad Request for insufficient funds):**
  ```
  {
    "status": 400,
    "error": "Bad Request",
    "message": "Invalid 'from' or 'to' account ID."
  }
  ```

- POST /bff/transactions/transfer/execution: to execute the transaction.
  - **Request Body Example :**
    ```
    {
      "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
      "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
      "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
      "amount": 30.00,
      "description": "Transfer to checking account"
    }
    ```

  - **Response Example (JSON - 200 OK):**
    ```
    {
      "transactionId": "t1r2a3n4-s5a6-7890-1234-567890abcdef",
      "fromAccountId": "f1e2d3c4-b5a6-9876-5432-10fedcba9876",
      "toAccountId": "g7h8i9j0-k1l2-3456-7890-abcdef123456",
      "amount": 30.00,
      "description": "Transfer success.",
    ```

```
  "timestamp": "2025-06-30T10:10:00Z"
}
```

- **Error Response (400 Bad Request for invalid accounts, 400 Bad Request for insufficient funds):**

```
{
  "status": 400,
  "error": "Bad Request",
  "message": "Invalid 'from' or 'to' account ID."
}
```

5. Logging Microservice (Kafka Consumer)

- A dedicated **Logging Microservice** acts as a Kafka **consumer**, listening to the logging topic.
- Upon receiving a message, it:
  - Parses the log data.
  - Extracts the key values (message, messageType, dateTime).
  - Inserts the information into a **dump table** in its local database for further use.

Logging Flow Description (Using Kafka)

2. **Request and Response Logging in Microservices**
   - After each **incoming request** is processed and just before sending the **response**, every microservice will prepare a log message.
   - This message contains either the **JSON request** or **JSON response**, appropriately escaped for transmission.
   - A Kafka **producer** in the microservice is responsible for sending this message to a central Kafka topic.

Message Format Sent by Producer

Each log message sent to Kafka has the following structure:

```
{
     "message": "<escaped JSON request or response>",
     "messageType": "Request" | "Response",
      "dateTime": "<ISO 8601 timestamp>"
  }
```

3.
    - **message**: The actual request or response body in JSON format (escaped as needed).
    - **messageType**: Indicates whether the content is a "Request" or "Response".
    - **dateTime**: The timestamp of when the log was generated.

4. **Logging Microservice (Kafka Consumer)**
    - A dedicated **Logging Microservice** acts as a Kafka **consumer**, listening to the logging topic.
    - Upon receiving a message, it:
        - Parses the log data.
        - Extracts the key values (message, messageType, dateTime).
        - Inserts the information into a **dump table** in its local database for further use.

5. **Dump Table**
    - The dump table acts as a centralized storage point for all logs.
    - It can be queried later for debugging, auditing, or monitoring purposes.

API Gateway Break down

The gateway will be the entry point for any incoming requests, enforcing authentication and throttling to the resources.

the gateway will have 4 APIs and 1 API product as follows

API	Description	Resource	BE endpoint
Register	this API will be used to register a new user	/register	/users/register
Login	this API will reroute the client request to the login API provided by BFF	/login	/users/login
Dashboard	this API will reroute the client request to the dashboardAPI provided by BFF	/dashboard/{userId}	/bff/dashboard/{userId}
Transactions	this API will reroute the client request to the transactions APIs provided by BFF	/initiation /execution	/bff/transactions/transfer/initiation /bff/transactions/transfer/execution
vbank (API product)	this product will be used to package above APIs into 1 package	/vbank	/login /dashboard/{userId} /transactions/transfer/initiation /transactions/transfer/execution

these APIs must be secured using OAUTH2 and API key

also we need to create 2 applications (one for portal and another for mobile app)

App name	Description
vbank portal	this app represents all requests coming from a web portal
vbank mobile	this app represents all requests coming from a mobile application

*for each request we need to identify the calling app and add a header for app name in the request to the BE

header name: APP-NAME, **allowed values**: ['PORTAL','MOBILE']

Technology Stack

- **Backend Language:** Java 11
- **Microservices Framework:** Spring Boot
- **Build Tool:** Maven/gradle
- **API Gateway:** WSO2 API Manager
- **Database: MySQL-Postgree-H2 (choose an of them)**
- **API Testing:** Postman