**Financial Account Management System**

**1. Project Overview**

This application is a Financial Account Management System developed using ASP.NET Core Web API and Entity Framework Core. It allows users to manage financial accounts, perform transactions (deposits and withdrawals), and retrieve financial data via RESTful API endpoints. It supports operations such as CRUD for accounts and transactions, business logic for deposits and withdrawals, and LINQ queries for retrieving various account-related information.

**2. Prerequisites**

Before running this project, ensure you have the following installed:

* .NET 6.0 or later
* SQL Server
* Visual Studio 2022
* Swagger UI (for API testing)
* Entity Framework Core Tools

**3. Setting Up the Project**

**3.1 Clone the Repository**

Extract the provided ZIP file (BackendDev\_Case2\_Palacio)

**3.2 Restore the Database**

In the extracted file look for database backup (FinancialAccountManagementSystem.bak), follow these steps to restore it in **SQL Server Management Studio (SSMS):**

##### **Step 1: Open SQL Server Management Studio (SSMS)**

* Launch **SQL Server Management Studio (SSMS)**.
* Connect to the database server.

##### **Step 2: Open the Restore Database Window**

* In **Object Explorer**, right-click on **Databases** and select **Restore Database**.

##### **Step 3: Select the Backup File (**FinancialAccountManagementSystem**.bak)**

* In the **Restore Database** window, choose **Device** and click the **Browse (...)** button.
* Click **Add**, locate the FinancialAccountManagementSystem.bak, and select it.
* Click **OK** to confirm your selection.

##### **Step 4: Restore the Database**

* Click **OK** to start the restore process.
* Once completed, you should see a success message.

##### **Step 5: Verify the Restoration**

* Refresh the **Databases** folder in **Object Explorer**.
* Expand the restored **FinancialAccountManagementSystem** database and check if tables exist.

**3.2 Get Connection string**

 **Navigate to the Project Directory**

* Open the project directory and locate **FinancialAccountManagementSystem.sln**.
* Open the solution in **Visual Studio**.

 **Open SQL Server Object Explorer**

* Click on the **Search Bar** in Visual Studio.
* Search for **SQL Server Object Explorer** and open it.

 **Connect to the Database**

* In **SQL Server Object Explorer**, click on the **"Add SQL Server"** icon.
* Enter your **Server Name**.
* Click the **dropdown under Database Name** and select the restored database (**FinancialAccountManagementSystem)**.
* Set **Trust Server Certificate** to **True**.
* Click **Connect**.

 **Retrieve the Connection String**

* In **SQL Server Object Explorer**, right-click the **FinancialAccountManagementSystem** database.
* Select **Properties**.
* Copy the **Connection String** from the **Connection** section.
* Update the appsettings.json file with the retrieved connection string:

"ConnectionStrings": {

"DefaultConnection": "Server=YOUR\_SERVER;Database=EcommerceDB;Trusted\_Connection=True;TrustServerCertificate=True;"

}

**4. Running the Application**

 **Open the Project in Visual Studio**

* Navigate to the project directory and open **FinancialAccountManagementSystem.sln** in **Visual Studio**.

 **Ensure the Database is Configured**

* Make sure the database is restored and the connection string in appsettings.json is correctly set.

 **Run the Application**

* In **Visual Studio**, locate the **Run** button at the top (with a green play icon).

 **Verify the API is Running**

* Once the application starts, it should open in a web browser.

**5. API Endpoints**

**5.1 Accounts Endpoints**

* **GET** /api/Accounts → Retrieve all accounts.
* **GET** /api/Accounts/{id} → Retrieve account by ID.
* **POST** /api/Accounts → Create a new account.
* **PUT** /api/Accounts/{id} → Update an account.
* **DELETE** /api/Accounts/{id} → Delete an account.

**5.2 Transactions Endpoints**

* **GET** /api/Transactions → Retrieve all transactions.
* **GET** /api/Transactions/{id} → Retrieve transaction by ID.
* **POST** /api/Transactions → Create a new transaction.
* **PUT** /api/Transactions/{id} → Update a transaction.
* **DELETE** /api/Transactions /{id} → Delete a transaction.

##### **Business Logic:**

* When making a deposit, the amount is added to the account's balance.
* When making a withdrawal, the amount is subtracted from the balance. If the account has insufficient funds, the withdrawal is not processed.

**6. LINQ Queries**

These endpoints provide advanced data retrieval:

* **GET** /api/LinqQueries/transactions/{id} → Retrieve all transactions for a specific account
* **GET** /api/LinqQueries/total-balance → Retrieve the total balance of all accounts
* **GET** /api/LinqQueries/accounts-below-balance → Retrieve all accounts with a balance below a certain threshold (e.g., $100)
* **GET** /api/LinqQueries/top-five-accounts → Retrieve the top 5 accounts with the highest balances.

### ****7. Data Validation Rules****

#### **Accounts**

* **GET /api/accounts/{id}**:
  + Ensure the account exists before retrieving it.
* **POST /api/accounts**:
  + Account number must be unique before creating a new account.
* **PUT /api/accounts/{id}**:
  + Ensure the account exists before updating.
* **DELETE /api/accounts/{id}**:
  + Ensure the account exists before deleting.

#### **Transactions**

* **GET /api/transactions/{id}**:
  + Ensure the transaction exists before retrieving it.
* **POST /api/transactions**:
  + Ensure the associated account exists before creating a transaction.
  + If the transaction type is **"Withdrawal"**, check if the withdrawal amount does not exceed the account balance.
  + The transaction type must be either **"Deposit"** or **"Withdrawal"**.
* **PUT /api/transactions/{id}**:
  + Ensure the transaction exists before updating.
* **DELETE /api/transactions/{id}**:
  + Ensure the transaction exists before deleting.

**8. Testing the API**

**8.1 Using Swagger UI**

1. Run the application.
2. Test API endpoints interactively.

**9. Assumptions & Trade-offs**

**9.1 Assumptions**

Single Database with Multiple Tables:

* I assume the project will use one database containing multiple tables to manage different data entities. This simplifies setup and ensures all data is centralized for easier access and maintenance.

**9.2 Trade-offs**

**Local vs. Cloud Database:**

**• Tradeoff:** Using a local SQL Server database simplifies setup and development but limits scalability and remote access. A cloud-based database like Azure SQL offers better scalability, availability, and backup options but adds complexity and potential costs.

• **Why Local Database:** A local database is chosen for better control, faster development cycles, and no dependency on internet connectivity during the initial development phase.