**E-commerce Product Management System - Documentation**

**1. Project Overview**

This project is an E-commerce Product Management System built using ASP.NET Core Web API with Entity Framework Core and SQL Server as the database. It provides CRUD operations for managing products, categories, and orders with a structured API design. The system supports basic product categorization, order processing, and LINQ-based queries to retrieve useful sales insights.

**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\_Case1\_Palacio)

**3.2 Restore the Database**

In the extracted file look for database backup (ProductManagement.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 (ProductManagement.bak)**

* In the **Restore Database** window, choose **Device** and click the **Browse (...)** button.
* Click **Add**, locate the ProductManagement.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 **ProductManagement** database and check if tables exist.

**3.2 Get Connection string**

 **Navigate to the Project Directory**

* Open the project directory and locate **E-Commerce Product Management.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 (**ProductManagement**).
* Set **Trust Server Certificate** to **True**.
* Click **Connect**.

 **Retrieve the Connection String**

* In **SQL Server Object Explorer**, right-click the **ProductManagement** 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 **E-Commerce Product Management.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 Product Endpoints**

* **GET** /api/Products → Retrieve all products.
* **GET** /api/Products/{id} → Retrieve product by ID.
* **POST** /api/Products → Create a new product.
* **PUT** /api/Products/{id} → Update a product.
* **DELETE** /api/Products/{id} → Delete a product.

**5.2 Category Endpoints**

* **GET** /api/Categories → Retrieve all categories.
* **GET** /api/Categories/{id} → Retrieve category by ID.
* **POST** /api/Categories → Create a new category.
* **PUT** /api/Categories/{id} → Update a category.
* **DELETE** /api/Categories/{id} → Delete a category.

**5.3 Order Endpoints**

* **GET** /api/Orders → Retrieve all orders.
* **GET** /api/Orders/{id} → Retrieve order by ID.
* **POST** /api/Orders → Create a new order.
* **PUT** /api/Orders/{id} → Update an order.
* **DELETE** /api/Orders/{id} → Delete an order.

**6. LINQ Queries**

These endpoints provide advanced data retrieval:

* **GET** /api/LinqQueries/category/{id} → Retrieve all products in a specific category.
* **GET** /api/LinqQueries/last-month → Retrieve all orders placed within the last month.
* **GET** /api/LinqQueries/total-sales → Retrieve the total sales for each product.
* **GET** /api/LinqQueries/top-five → Retrieve the top 5 products with the highest sales.

**7. Data Validation Rules**

**7.1 PRODUCTS:**

* Product names must be unique.
* Price must be a valid numeric value.
* Stock quantity must be a valid positive integer.
* Product must exist in the database.

**7.2 CATEGORIES:**

* Category names must be unique.
* Category must exist in the database.

**7.3 ORDER:**

* Order date must be valid (e.g., not in the future).
* Order must exist in the database.

**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**

 **Database Type**:  
I assume that the project will use **SQL Server** for data storage. While I could have used a cloud-based database like Azure SQL or PostgreSQL for scalability, I’m opting for a local database for simplicity and better control during development.

 **Basic Authentication**:  
I assume the API will use **basic authentication** or that no authentication will be required at first. This will allow internal or basic users to access the endpoints.

**9.2 Trade-offs**

**Authentication**:

* **Tradeoff**: Adding authentication might complicate the project, especially when it’s not specified in the instructions. However, if the app is intended for production or multi-user use, authentication will be essential.
* **Why No Authentication**: The task does not specify authentication requirements, so it will be omitted in the initial version. Authentication can always be added later as an enhancement.

**DTOs**:

* **Tradeoff**: Using DTOs adds an extra layer of abstraction, which can make the code more complex. However, it allows the API to evolve independently from the database models, and it helps prevent over-exposing the internal structure of the database.
* **Why DTOs**: The task explicitly asks for DTOs, so using them ensures consistency and follows best practices for structuring API responses.

**Database Management**:

* **Tradeoff**: SQL Server is often chosen for enterprise applications due to its rich feature set, but it is more resource-heavy compared to PostgreSQL. PostgreSQL is more lightweight and can be more cost-effective, especially in cloud environments.
* **Why SQL Server**: SQL Server was assumed for simplicity and compatibility with ASP.NET Core projects.