AwesomeGIC Bank System

1 Project Setup

1.1 Download project from github

[GitHub - JayampathiBandara/AwesomeGICBank: Awesome GIC Bank]

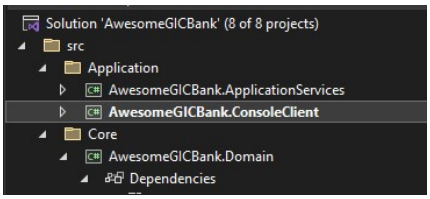
1.2 Restore Nuget Package

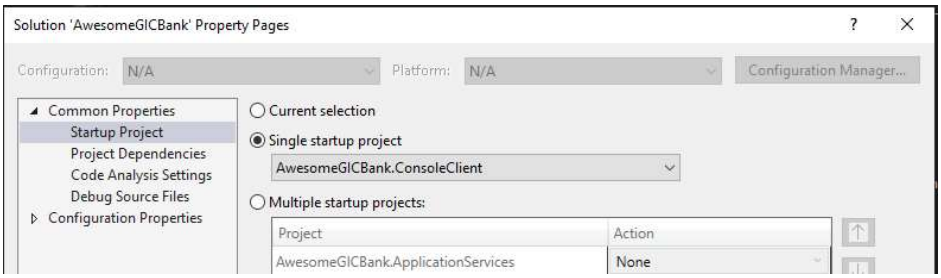
Using Package Manager Console in Visual Studio you can use any of the following

- `dotnet-restore`
- `Update-Package -reinstall`

1.3 Set Startup Project

Set AwesomeGICBank.ConsoleClient as startup Project





1.4 Setup the database in SqlServer or SQLExpress

Update connection strings in Project AwesomeGICBank.ConsoleClient

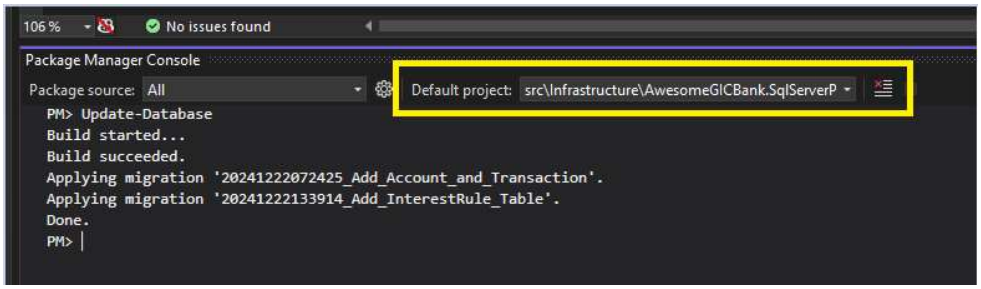
Go to `AwesomeGICBank\Application\AwesomeGICBank.ConsoleClient\appsettings.json`

```
"ConnectionStrings": {
    "GicBankConnection": "Data Source={YourMachineName};Initial
Catalog=GicBank;Integrated Security=True;
TrustServerCertificate=True;MultipleActiveResultSets=True "
  }
```

Change it according to the Setup of sqlserver in your machine
Ex: “Data Source=DESKTOP-N064N22\\SQLEXPRESS;”
Or “Data Source=DESKTOP-N064N22”

Run `Update-Database` Command in package manager console

1. Open Package manager console
2. Set AwesomeGICBank.SqlServerPersistence as Default project as mentioned in yellow box
3. Run `Update-Database`



2 [AwesomeGIC Bank System] Design Architecture

The following architectural, design patterns were used in system development.

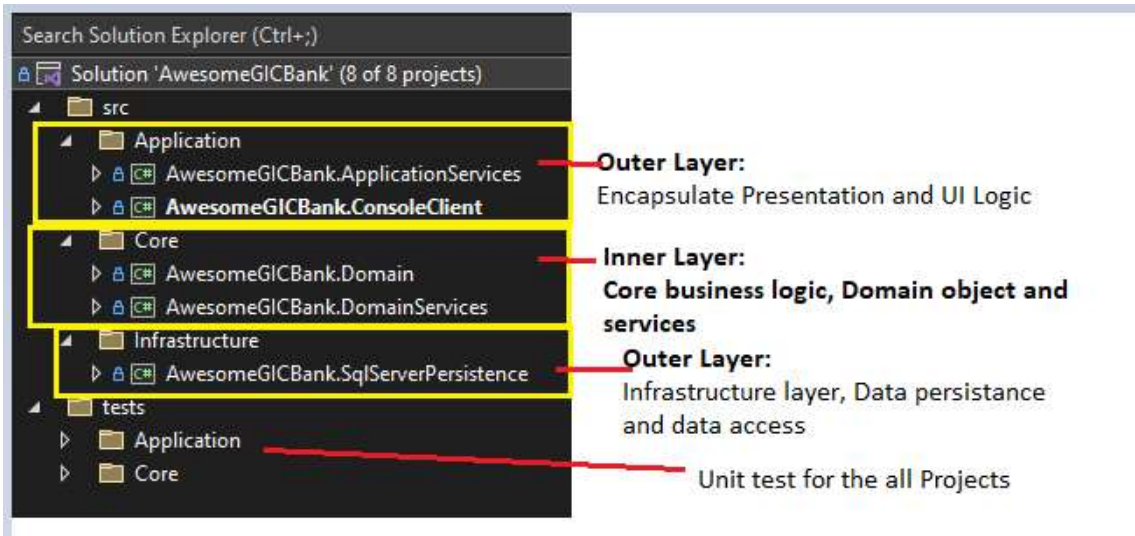
2.1 Clean Architecture

The application is designed following the principles of Clean Architecture, which emphasizes separation of concerns and the independence of layers to ensure maintainability and scalability.

The architecture is structured into concentric layers:

- **Inner layer** represent the core business logic and domain models, ensuring the application logic is independent of external systems.
- Two **Outer layers** handle
  - infrastructure concerns, such as data access.
  - User interaction, and presentation.

Domain-Driven Design (DDD) principles are integrated into the core layer to model the domain effectively, focusing on the ubiquitous language and bounded contexts to align the software with the business's needs.



2.2 Repository Pattern

- The Repository Pattern is adopted to decouple data persistence logic from domain services.
- It provides an abstraction layer over the data access layer, enabling interchangeable data sources (e.g., SQL, NoSQL, or in-memory) and easier unit testing by mocking repositories.

2.3 Unit Of work Pattern

- The Unit of Work Pattern works in conjunction with the Repository Pattern to ensure the atomicity of transactions.
- It keeps track of changes to entities during a business transaction and commits them as a single operation to maintain consistency.

2.4 Template method Pattern

- The Template Method Pattern is used to define the skeleton of an algorithm in a base class, allowing subclasses to override specific steps.
- In this project, it is utilized to create reusable UI components that provide a consistent user experience while enabling custom behaviour in specific scenarios.
  Ex:.`TRansactionUserInterface, BankStatementUserIterface, InterestRuleInterface`

2.5 Factory Pattern

- The Factory Pattern is implemented to encapsulate the logic of creating objects based on user input.
- It simplifies object creation by delegating it to sub classes, which are particularly useful for generating user interfaces dynamically based on runtime conditions.
- Ex:.`TRansactionUserInterface, BankStatementUserIterface, InterestRuleInterface`

2.6 Event sourcing Pattern

- The Event Sourcing Pattern is used to maintain a transaction history by persisting a series of events rather than directly storing the current state.
- The final state is derived by replaying these events, ensuring an auditable and traceable history of all operations.
  Ex: Store Bank transaction and calculate balance and interest

2.7 Mediator Pattern

- The Mediator Pattern is implemented using the MediatR NuGet package to decouple interactions between components (e.g., layers, services).
- This ensures that components communicate indirectly through a mediator, simplifying maintainability and reducing dependencies.

2.8 CQRS Principle

- The CQRS Principle is applied to separate read and write operations.
- Commands handle data mutations (e.g., creating transactions), while queries handle read-only operations (e.g., generating account statements).
- This segregation enhances scalability and allows independent optimization of read and write operations.

2.9 Solid Principle

- SOLID principles are adhered to throughout the project, ensuring that the code is modular, maintainable, and extendable.
- Each of the above-mentioned design patterns aligns with SOLID principles to promote clean and robust design.

3 Libraries

3.1 EF Core 8

- Entity Framework Core **(EF Core) 8** is employed as the **Object-Relational Mapper (ORM)** to facilitate interaction with the underlying **SQL Server database**.
- A **Code-First** approach is used, where the database schema is generated from the code. This is paired with **Fluent API** to define and configure entity mappings, ensuring flexibility and maintainability of the schema.
- The **EF Core migration tool** is utilized to manage database schema changes and synchronize the database with the code model.
- The Fluent API is used extensively for configuring ORM mappings, ensuring that domain objects within the core layer remain persistence-ignorant, thus adhering to Clean Architecture principles.

3.2 IServiceCollection

- Dependency injection is handled using the `Microsoft.Extensions.DependencyInjection` library, which provides the **IServiceCollection** interface.

- Both Domain Services and SQL Persistence projects have their own registries with dependency injection extensions, enabling the configuration of services.
  These services are finally registered within the `ConfigureServices` method in `Program.cs`, promoting loose coupling and easier unit testing by decoupling service instantiation from the business logic.

3.3 AutoMapper NuGet Package:

- **AutoMapper** simplifies the mapping between **Data Transfer Objects (DTOs)** and domain objects across various layers of the application.
- This library reduces the amount of repetitive mapping code by automating the conversion process, improving code readability and maintainability.

3.4 Fluent Validation NuGet Package

- **Fluent Validation** is used to enforce **business rules** on **DTO objects** to ensure **data integrity** before any interaction with the core domain.
- It provides a fluent interface to define validation rules for complex objects, ensuring that only valid data is passed through the system, reducing errors and improving system reliability.

3.5 MediateR NuGet Package

- **MediatR** is utilized to decouple interaction between different layers and libraries of the application.
- By using MediatR, the application becomes more maintainable, as it promotes command and query segregation (CQRS) and reduces direct dependencies between the application layers.
- It allows for easier communication between components by using commands, queries, and events without requiring direct references.

3.6 Xunit and Moq NuGet Packages

- Xunit and Moq are used for unit testing logic, rules, and methods within the application.
- Xunit provides a framework for writing and running unit tests, while Moq is used for creating mock objects and stubbing dependencies to simulate different scenarios and ensure that the system behaves correctly under various conditions.