**Architectural flow.**

Briefly it works as follows:

When a client makes a request to the API, the API layer is responsible for receiving the request and processing it. Then, the service layer is triggered to execute the relevant business rules for that request.

If the operation requires data access, the service requests the infrastructure layer (Infra) to interact with the database. The infrastructure layer is responsible for performing data read/write operations on the database and returning the results to the service layer.

The service layer can then process these results and construct an appropriate response to the original request. Then the API layer can return the response to the client.

The domain layer contains the business entities and is used by the other layers to represent these entities. It is important to note that the domain layer is independent of the other layers, which means that it does not directly depend on other layers and can be tested independently.

**Used packages and some details**

**Domain layer:**

Entities in this project will inherit from the abstract Base class.

Using an abstract class allows you to define common behavior for a group of related classes, without specifying the detailed implementation. This makes code maintenance and evolution easier, as changes to common behavior can be made in one place, in the abstract class, and reflected in all derived classes.

In addition to entities, the domain layer contains validation classes using the FluentValidation package. These classes are called by the service layer before performing an operation involving data validation. These validations can be performed before inserting or updating data in the database or before performing other operations that require valid data.

For example, if the service receives a request to create a new object in an API, it can call validation classes in the domain layer to ensure that all properties of the object are filled in correctly and meet certain validation criteria. If any validation fails, the service can then return an appropriate error to the API informing the user that the data provided is invalid.

**Infra layer:**

The BaseRepository class was implemented, a generic class that can be used as a base for the implementation of specific repositories for each entity of the application. Using a base class like this can reduce the amount of code needed to implement repositories and make application maintenance easier. In addition, using Entity Framework Core for data access can reduce data access implementation complexity and increase development productivity.

The IProductsRepository class is an interface that inherits from IBaseRepository<Products>. It is an interface that defines a contract for a repository that must implement the basic CRUD operations (Create, Read, Update and Delete), in addition to other operations specific to the Products entity. This interface defines two methods that allow you to search for products by name and category.

This approach of separating the interface definition from the concrete implementation of the repository follows the Dependency Inversion Principle, which says that high-level modules should not depend on low-level modules, but both should depend on abstractions. In this way, the IProductsRepository interface is an abstraction that allows classes that need to access product data to not depend directly on the concrete implementation of the repository. Instead, they depend only on the interface, which makes the code more flexible and easier to test.

In the Infra layer, we have an entity mapping class in the database using the Entity Framework Core. Using mapping classes in Entity Framework Core is a common practice to separate the data model definition from the application's business logic. This separation allows the data model to be easily changed without affecting the application's business logic, as well as making the code easier to maintain.

**API layer:**

In the TokenGenerator class we make use of the JwtBearer package, this class is responsible for generating a JSON Web Token (JWT), which is a signed token that authenticates a web request. This class uses the JWT standard, which defines a secure way to transmit messages using a compact, self-contained token in the form of a JSON object. Also added azure packages for deploy.

**Services layer:**

In the ProductsService class we use the IMapper which is an interface of the AutoMapper package to map an object of type ProductsDto to an object of type Products. This allows us to work with DTO's.

Repository Link: <https://github.com/ArielLopes888/API_ShopBridge>

Online API link: <https://shopbridgeapi.azurewebsites.net/swagger>