Bunnings Warehouse API

Introduction

- This application is a single controller API that provides CRUD for products, i.e. adding, deleting and updating products

Requirements

- The application was developed using Visual Studio **2019** Community Edition.
- To build the application .NET SDK **3.1** is required
- The database uses an **MSSQL** Server

Features

- The Solution is composed of several different projects to provide a traditional 3-tier layer architecture for separation of concerns.

SOLID principles

- **Single Responsibility Principle**
  - Classes are primarily responsible for one thing; as an example the Startup.cs has been split up into separate modules for ease of maintenance

- **Open/Closed**
  - Is facilitated using interfaces for tier to tier communication and the IoC container to define the implementations.
  - As an example, the tests use mock implementations of the interfaces, substituting fake implementations.

- **Liskov Substitution Principle**
  - Demonstrated using ApiResult<T> which is substituted by ApiResult in the AbstractController's HandleApiResult

- **Interface Segregation Principle**
  - Interfaces are used at all layers with cohesive methods (IProductService, IStartupModule)

- **Dependency Inversion Principle**
  - The Database Context and the service are injected by the IoC container into the respective constructors in ProductService and ProductController respectively

API Layer (BunningsWarehouse.Api)

- This is meant to be a simple layer that provides a REST “interface” to the services layer.
- Contains a Swagger definition to explore the API.
- The startup module has been split up into several sub modules for ease of maintenance
- The AbstractController makes use of a Func<T> to reduce repeated code when determining the response code based on the API result
- The database connection string is derived from IConfiguration

Services Layer (BunningsWarehouse.Services)

- Responsible for business logic and invoking the data layer
- Performs validation (for more complex validation scenarios, FluentValidations can be used)
- A mapping class provides a transition between entities and models
- An ApiResult is used to:
    - Provide the capacity to return the result of an operation during failure
    - Make error clear with error codes
    - Avoid throwing exceptions (should be used in exceptional circumstances)
    - Provide the capacity to return either an error or a result (ApiResult{T}) if both are possible in a method

Data layer (BunningsWarehouse.Data)

- Uses EntityFrameworkCore 3.1 with MSSQL to store data with a standard DBContext

Testing

- **Please note:** Testing can be involved even for simple cases; sample test cases have been supplied but are not meant to be exhaustive as this is a code submission. In a Production grade environment, tests would be written first to ensure that the edge cases are covered (e.g. lower case comparison for strings and trimming input) at the minimum for every public method, if not for all methods.
    - **Example:**
    - Implement AddProduct without checking for extra spaces (i.e. Trim())
    - Create test case with empty space
    - Test fails
    - Fix AddProduct to use Trim to check for unnecessary spaces
    - Test passes

- The API Layer has been written as a thin layer which is meant to be an interface
- NUnit is the testing framework
- Fluent Assertions for cleaner and simpler assertion logic
- Moq provides fake implementations

- The Controller logic is layered in such a way that few tests are required; the bulk of tests would be written for the services layer