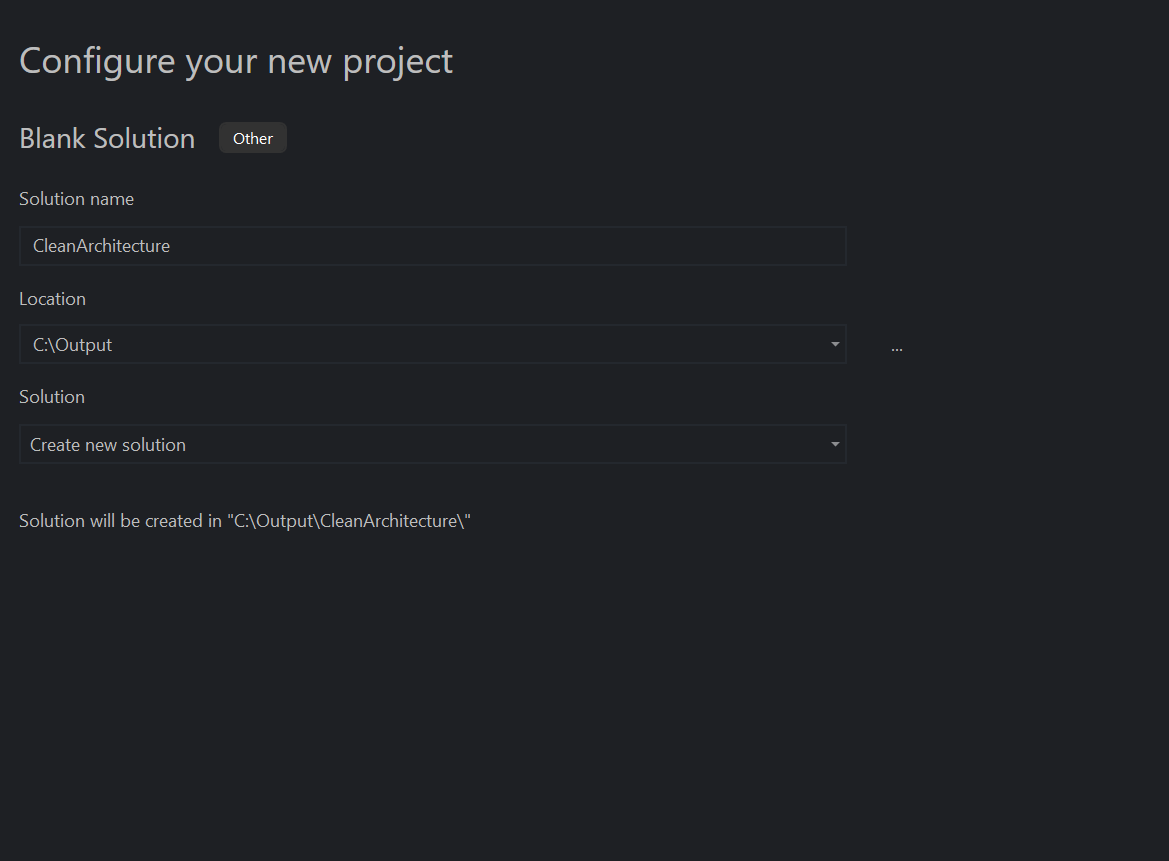
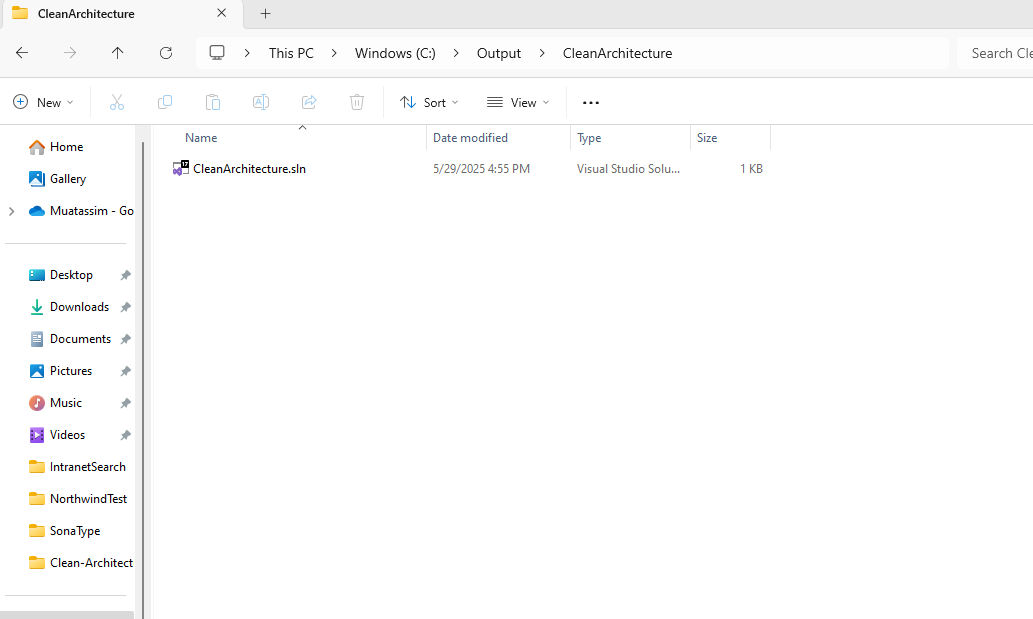
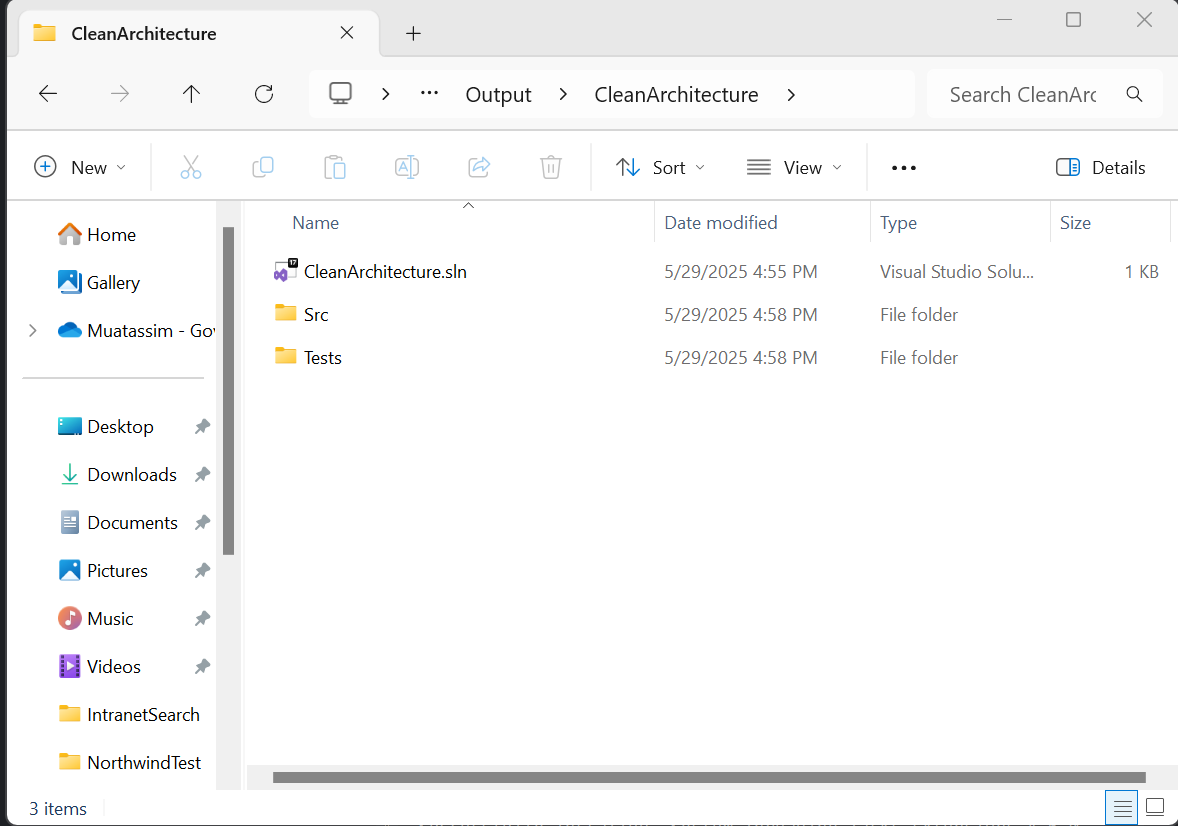
Clean Architecture Project Template -- Core

Open Visual Studio: Create new Blank Solution

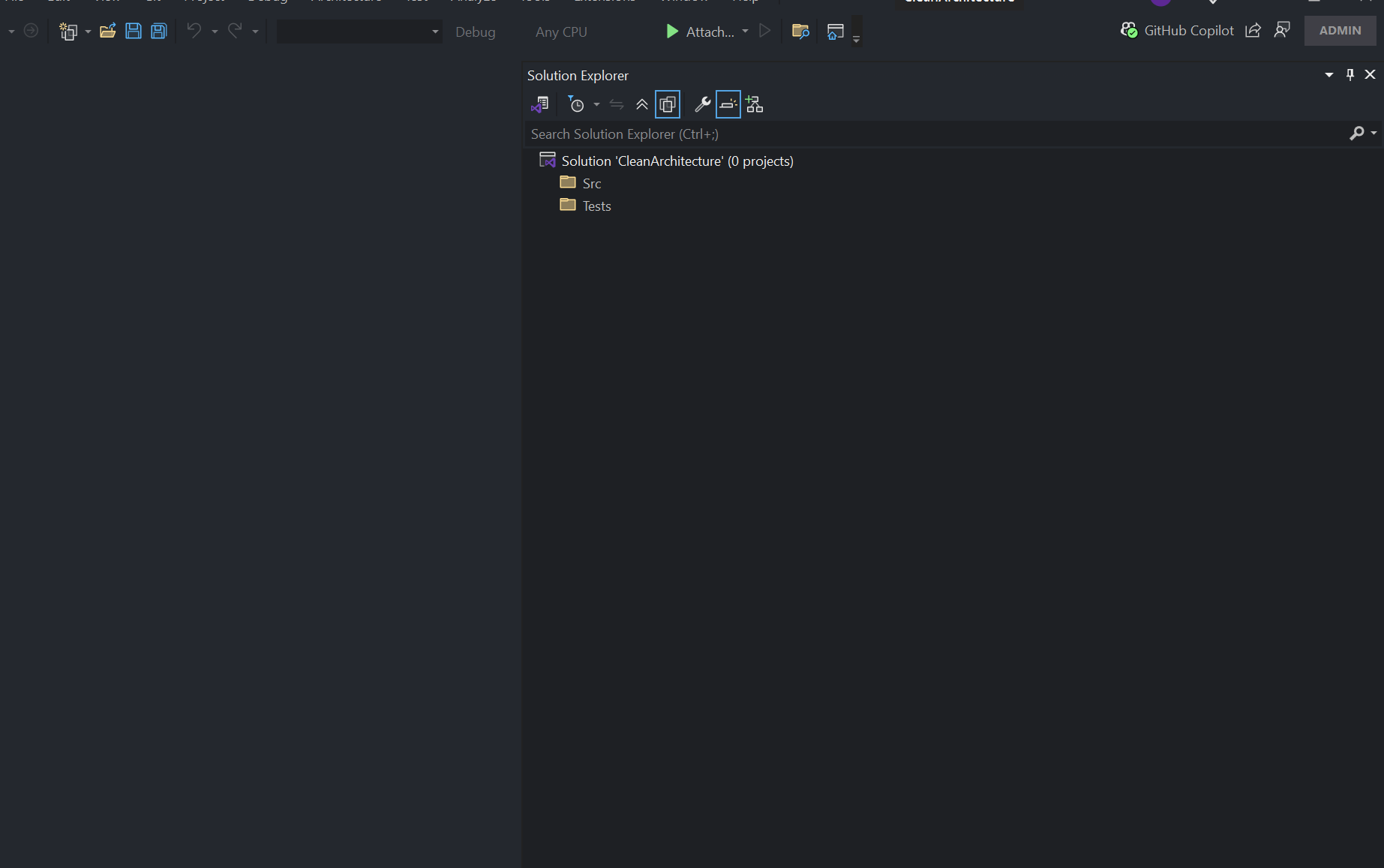


Choose directory, for example: c:\Output, your folder structure looks like



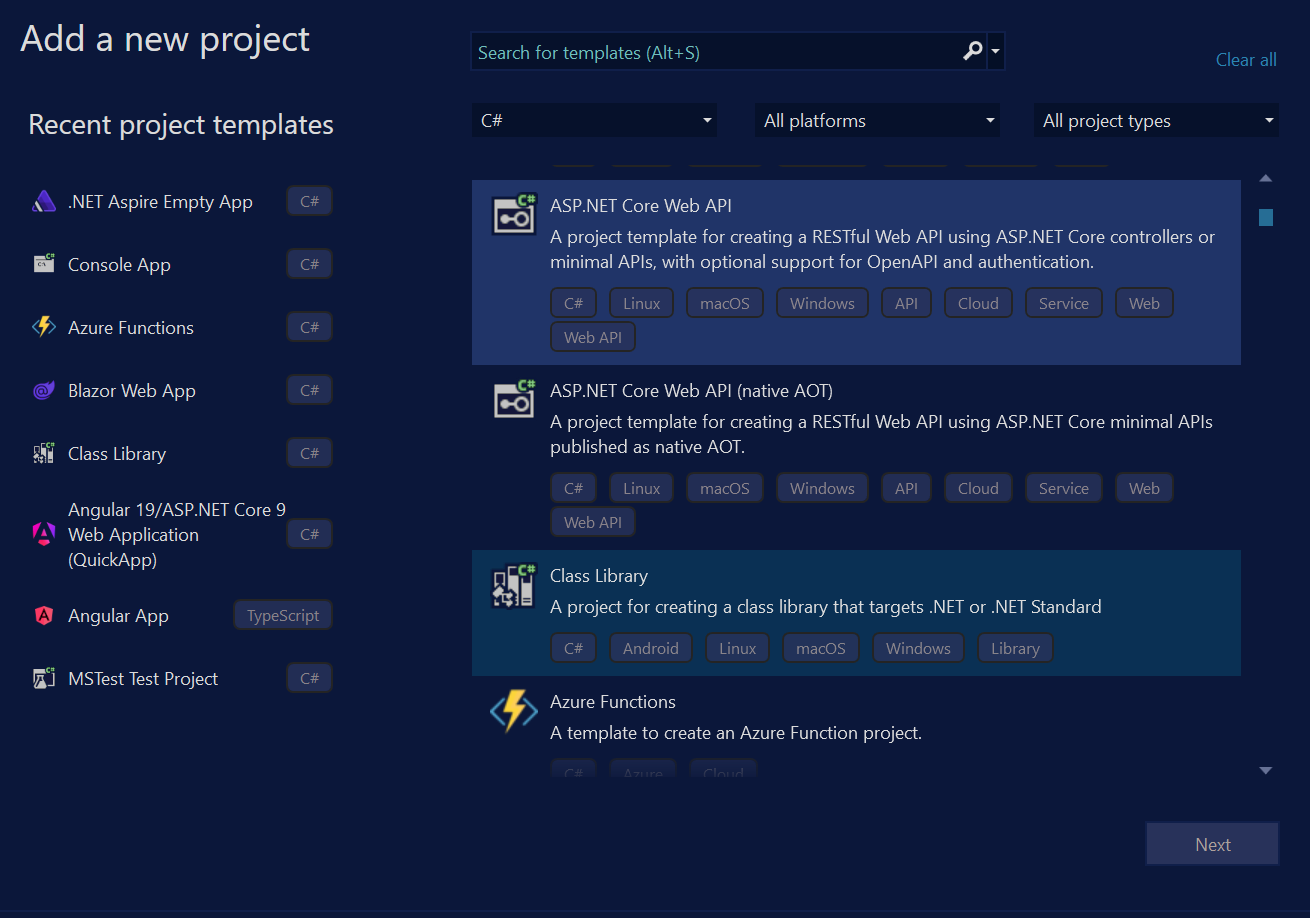


Also Add two solution folders in the solution with the same name:

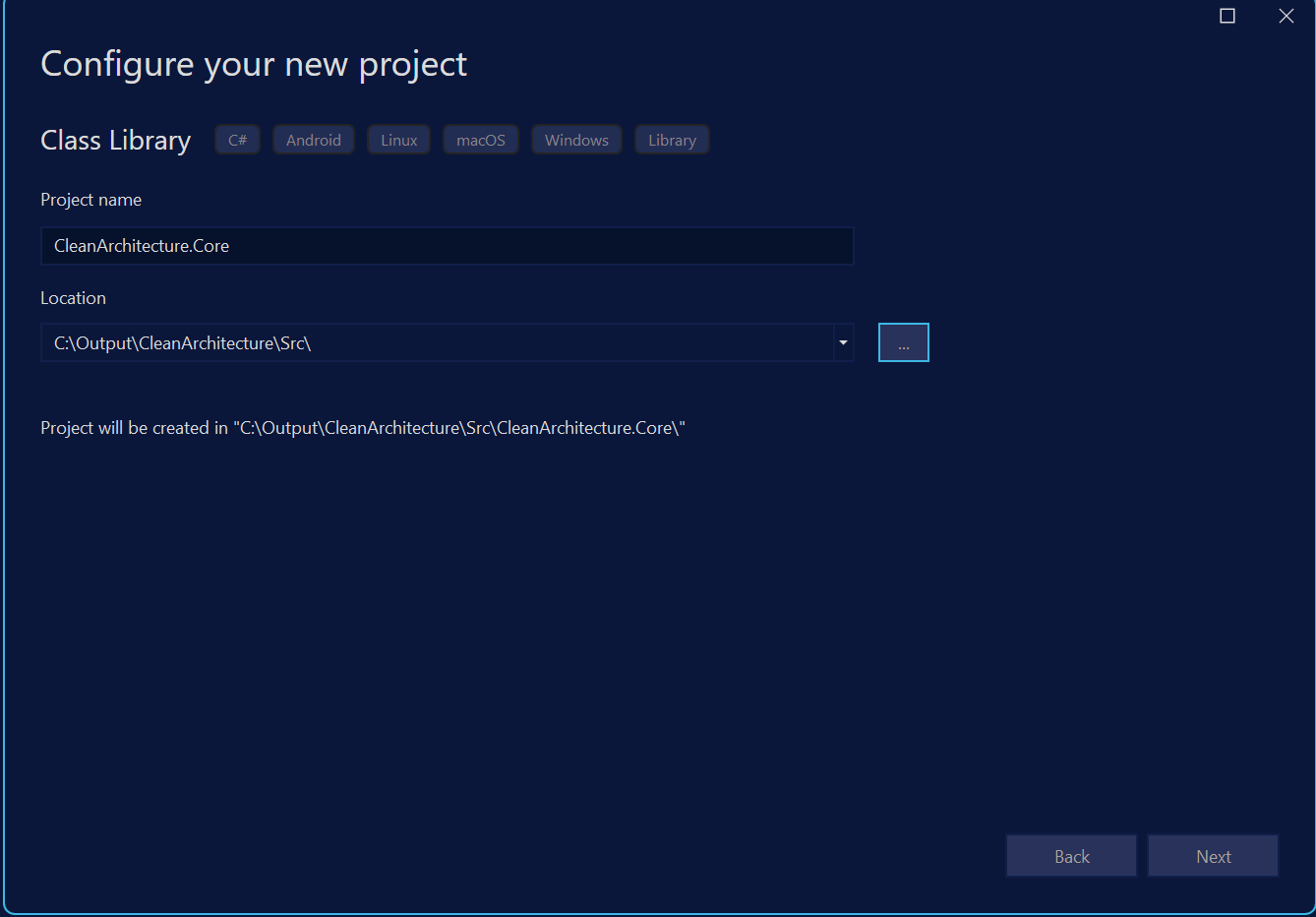


Your project should look like the screen above.

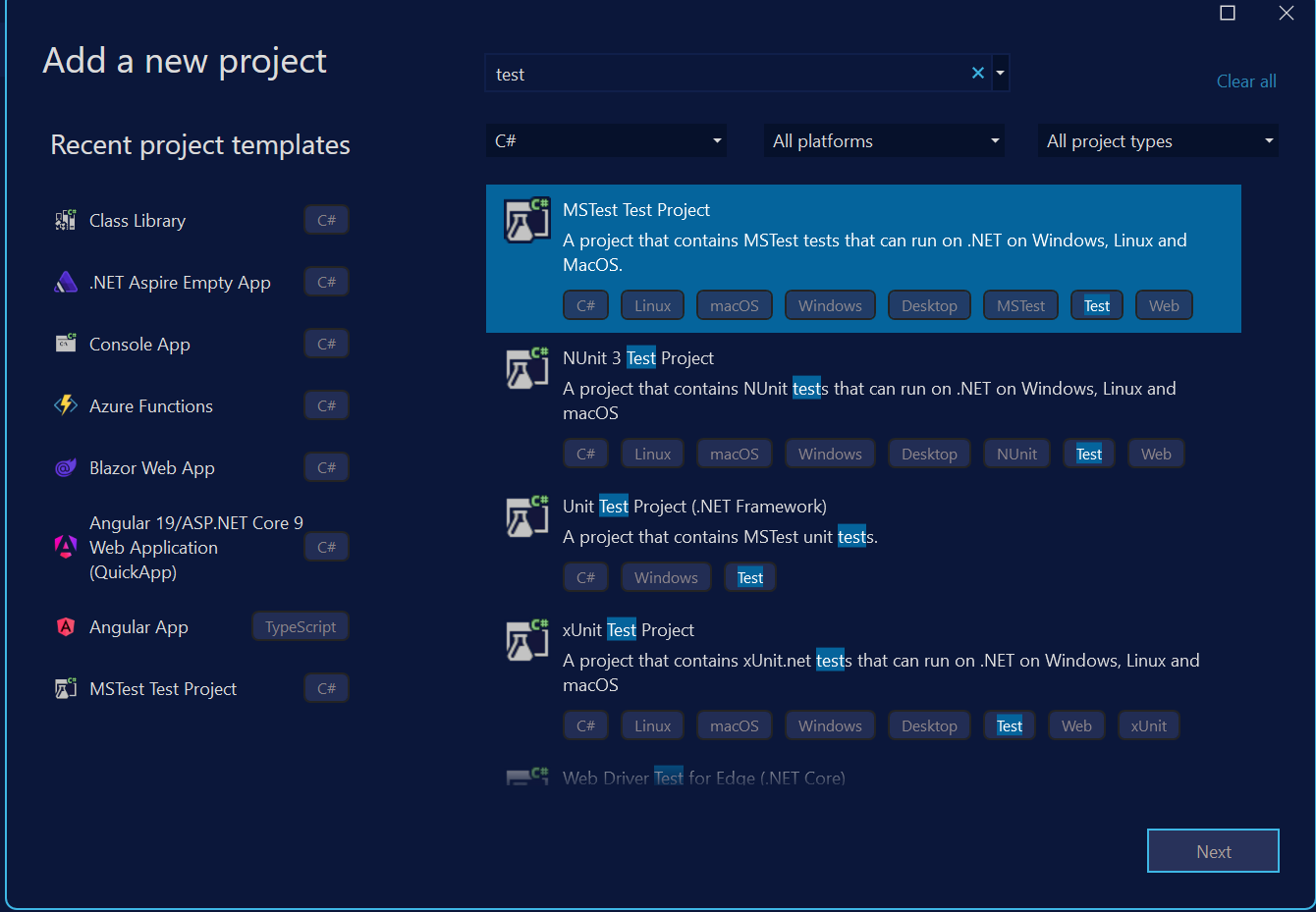
Right click on Src and add select a new project to add Class library Project:

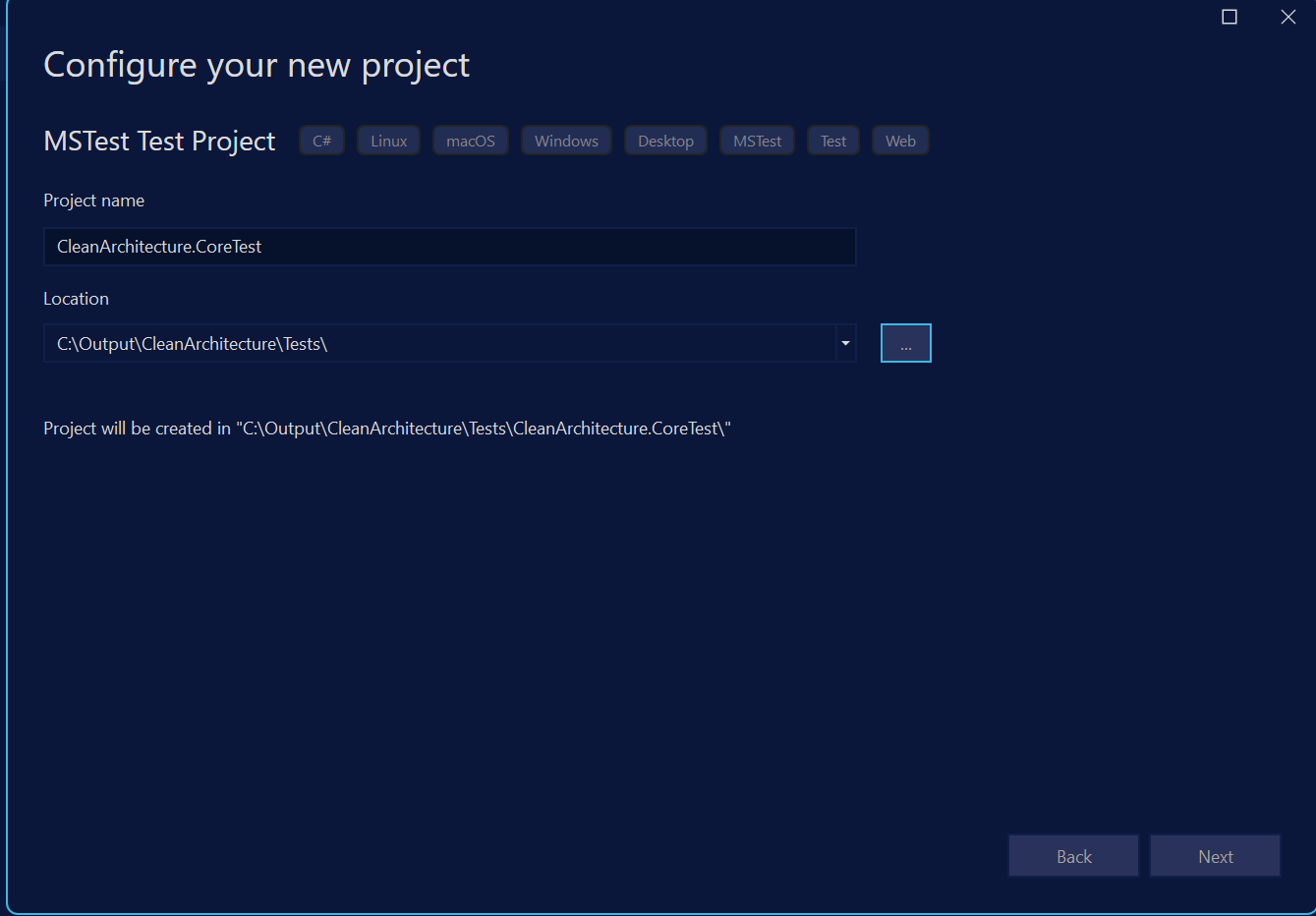


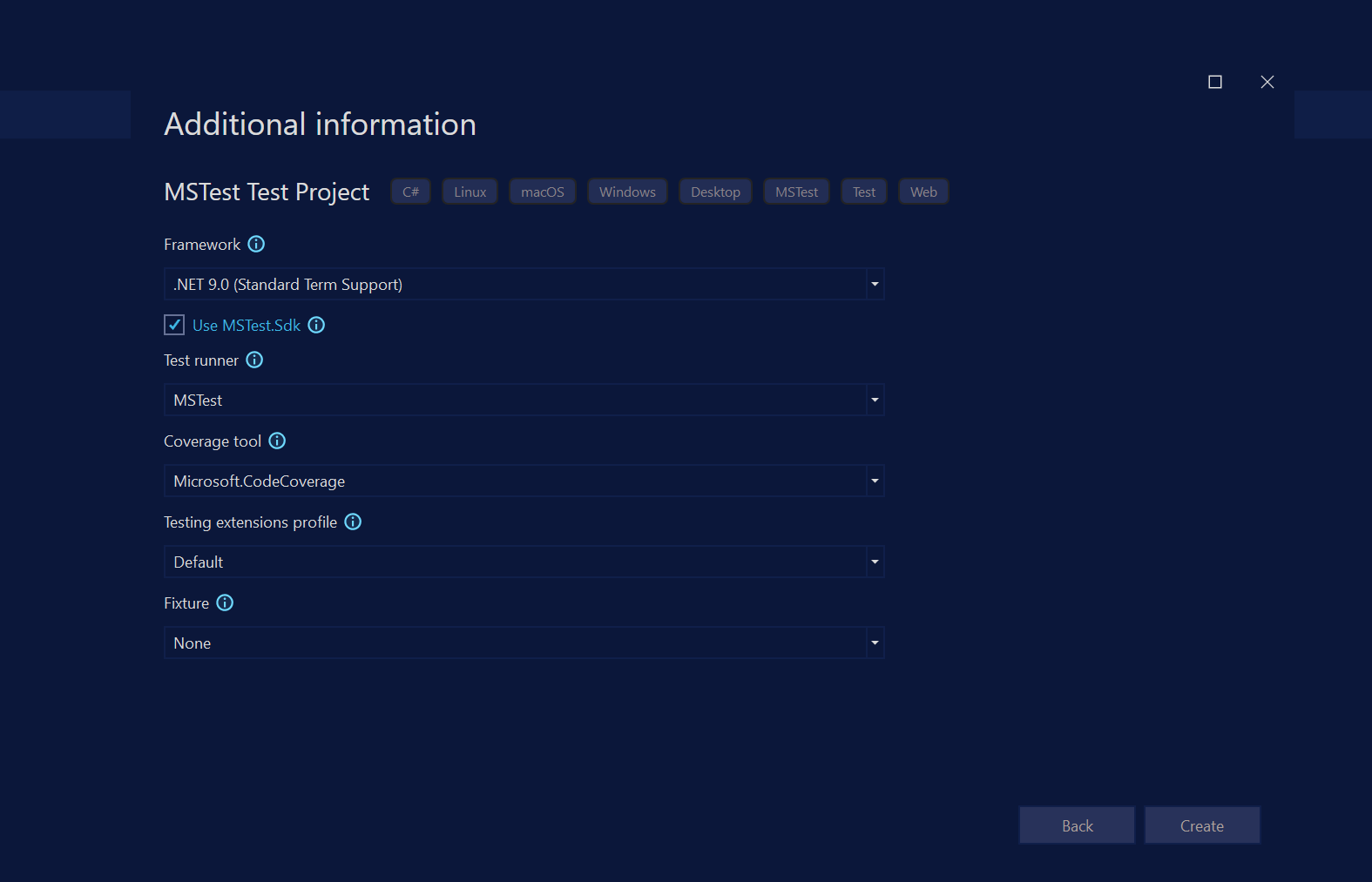
Make sure to name and location is in the newly created Src folder:



Take Defaults and create the project. Create a test project, see screens below, and be sure to select Test folder



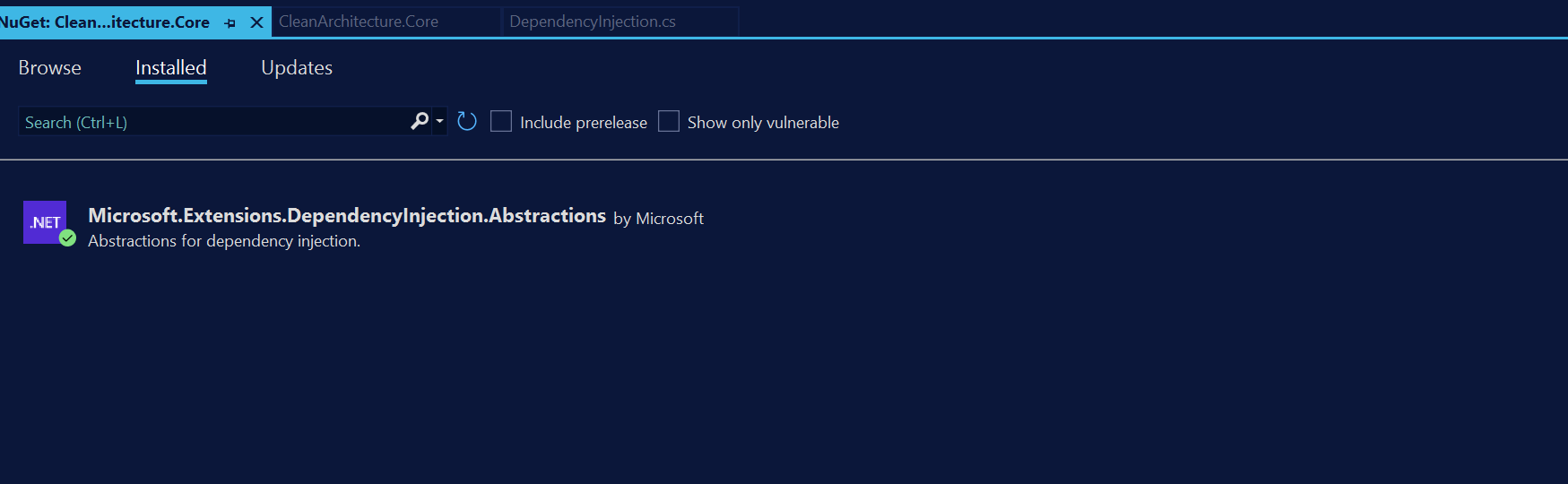




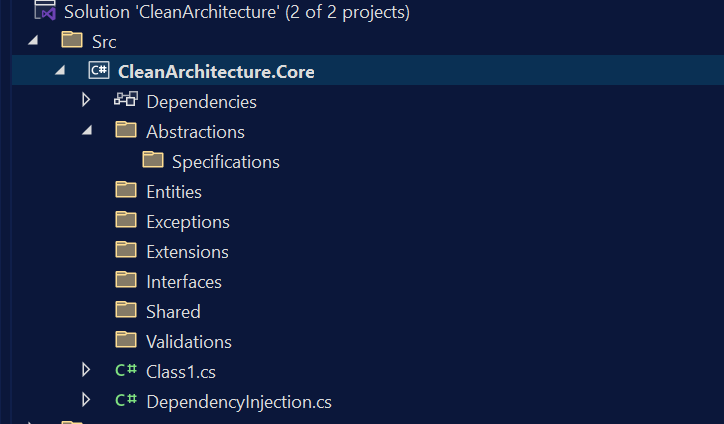
Add reference to the Core Project, ensure you can build the solution.

Create two folders Src, and Tests in the solution Folder.

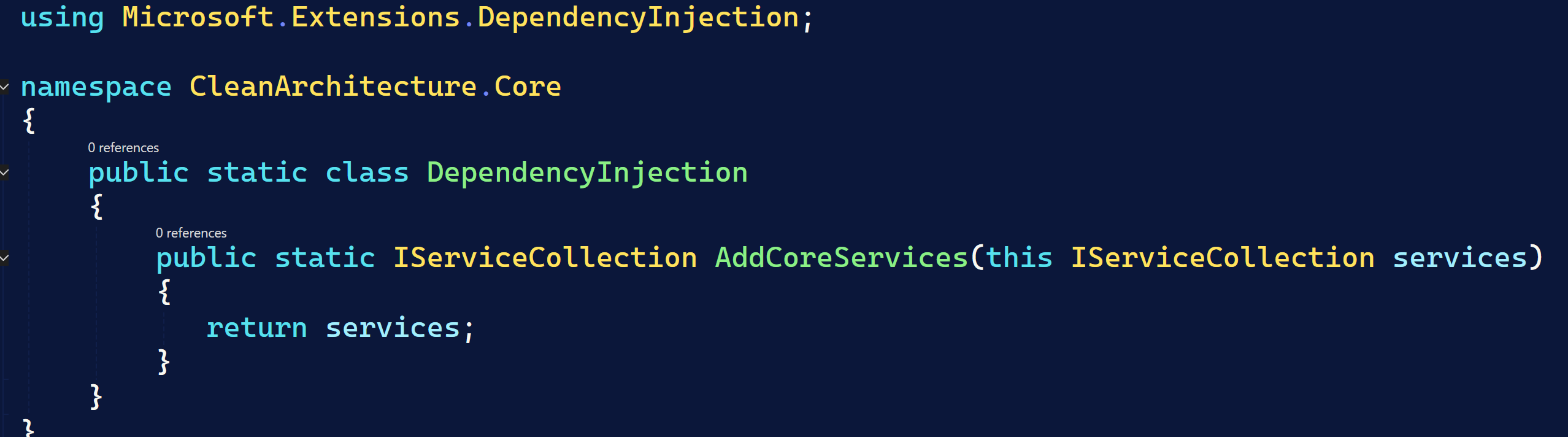
Add Nuget Package: Microsoft.Extensions.DependencyInjection.Abstractions



Add the following Folders in the Core Library:



Following is the content for DependencyInjection.cs



Delete the default Class1.cs file.

Entity Abstraction Files

Use the Entity\_1 Folder to move all files to their respective folders, ensure to change the namespace to your own:

Note: you can right click on Namespace and select change namespace to match folder structure in complete Project.

Folder/Files added are the following:

# **Entity Class (Abstractions)**

**We have two entity classes in the code base**

* **Entity<TEntityId> (generic)**
* **Entity (non-generic)**

**Purpose of Each**

1. **Entity<TEntityId>**

* **Generic: Accepts a type parameter for the entity's identifier (e.g., Guid, int, string).**
* **Use Case: When you want strong typing for entity IDs, ensuring compile-time safety and flexibility for different ID types.**
* **Features: Implements equality and hash code logic based on the ID, and provides a property for the ID.**

1. **Entity**

* **Non-generic: Does not require an ID type.**
* **Use Case: For entities that do not have a unique identifier, or when you want a base class for domain objects that don't need an ID (e.g., value objects, or for legacy reasons).**
* **Features: Provides domain event handling, but no ID or identity-based equality.**

# Error Class

**Immutable class will be used to represent error information in a clear structured way.**

# Lookup class

Will be used for the lookup data

# Result Class

The Result class is a pattern for representing the outcome of operations, encapsulating both success and failure states in a type-safe way. This approach is common in domain-driven design and functional programming to avoid exceptions for control flow and to make error handling explicit.

Key Features

1. **Success and Failure Representation**

* IsSuccess: Indicates if the operation succeeded.
* IsFailure: Indicates if the operation failed (the inverse of IsSuccess).
* Errors: A list of Error objects describing what went wrong (empty if successful).

2. **Factory Methods**

* **Result.Success()**: Creates a successful result with no errors.
* **Result.Failure(Error error)**: Creates a failed result with a single error.
* **Result.Failure(List<Error> errors)**: Creates a failed result with multiple errors.
* **Result<TValue>.Success(TValue value)**: Creates a successful result with a value.
* **Result<TValue>.Failure(Error error)**: Creates a failed result with a value type and a single error.
* **Result<TValue>.Failure(List<Error> errors)**: Creates a failed result with a value type and multiple errors.
* **Result<TValue>.Create(TValue? value)**: Returns a success if the value is not null, otherwise a failure with a standard error.

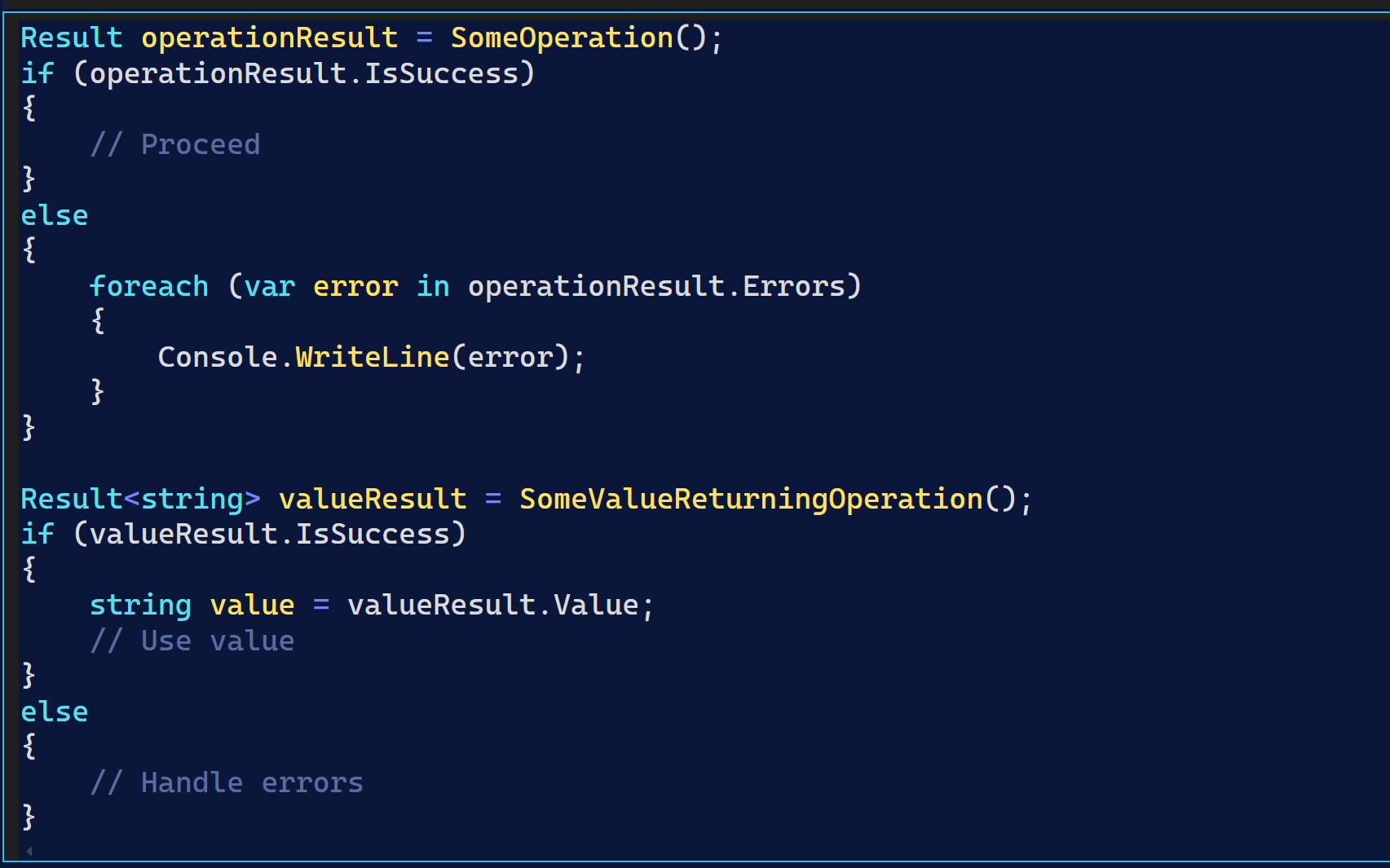
3. **Generic Support**

* **Result<TValue>**: Extends Result to carry a value of type **TValue** when the operation is successful.
* The Value property: Returns the value if successful, otherwise throws an exception if accessed on a failure.

4. **Validation**

* The constructor enforces that a successful result cannot have errors, and a failed result must have at least one error.

5. **Implicit Conversion**

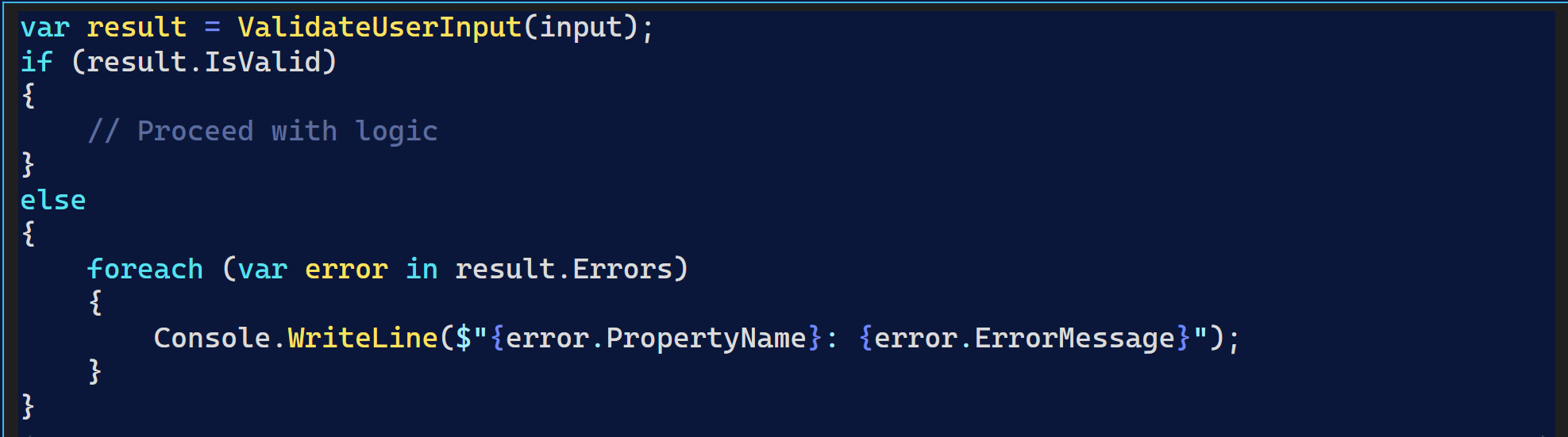
* We can implicitly convert a value of type **TValue** to **Result<TValue>**, which will create a success or failure based on whether the value is null. 

# Validation Result

The ValidationResult class is a simple structure for representing the outcome of a validation process. It is designed to capture whether validation succeeded and, if not, what errors occurred.

Key Features

* **Errors**:
  + A list of ValidationError objects describing what went wrong during validation.
  + The property is lazily initialized to an empty list if not set.
* **IsSuccessful**:
  + A boolean indicating if the validation was successful.
  + Set to **true** by default, but set to **false** when using the Fail factory method.
* **IsValid**:
  + A computed property that returns **true** if there are no errors (**Errors.Count == 0**), otherwise **false**.
  + This is the main indicator of validation success.
* **Error**:
  + An optional string for a general error message (not tied to a specific property).
* **Factory Members**:
  + Success: Returns a new, successful ValidationResult.
  + **Fail(List<ValidationError> errors)**: Returns a failed ValidationResult with the provided errors.



# Validation Service

The ValidationService classes in your codebase provide a standardized way to validate entities. They act as adapters that delegate validation logic to the entity itself, ensuring a consistent interface for validation across your application.

**Generic Design**

* There are two versions:
  + **ValidationService<T, TEntityId>** for entities with a generic ID (**Entity<TEntityId>**).
  + **ValidationService<T>** for entities without a generic ID (Entity).
* Both implement corresponding IValidatable interfaces, enforcing a contract for validation.

2. **Validation Logic Delegation**

* The IsValid method calls the entity’s own **IsValid()** method.
* This means the actual validation rules and logic are defined within the entity (or its base class), not in the service itself.

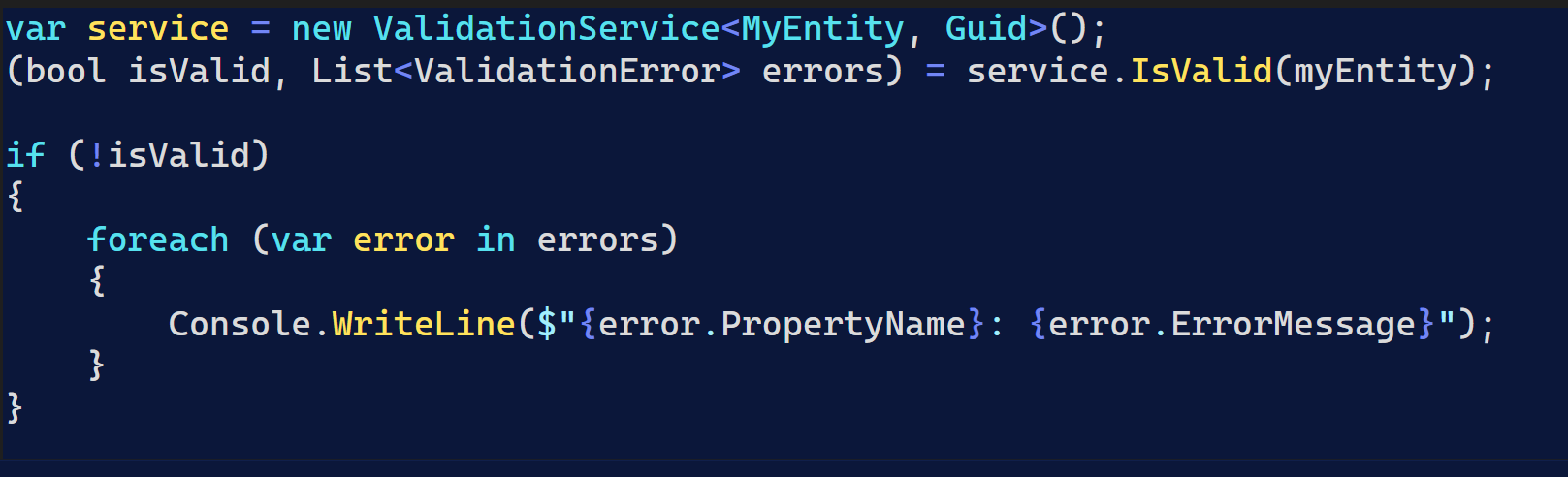
3. **Return Value**

* Returns a tuple: **(bool IsValid, List<ValidationError> Errors)**.
* IsValid: Indicates if the entity passed validation.
* Errors: A list of ValidationError objects describing any validation failures.

4. **Extensibility**

* The IsValid method is **virtual**, allowing you to override and extend validation behavior in derived services if needed.

Example:



# Specification

The Specification classes in your codebase implement the Specification Pattern, which is a way to encapsulate business rules, filtering, and query logic in reusable, composable objects. This pattern is especially useful in domain-driven design and repository implementations.

Key Features

There are two abstract base classes:

* **Specification<TEntity, TEntityId>** for entities with a generic ID.
* **Specification<TEntity>** for entities without a generic ID.

Both classes provide the same core functionality.

1. **Criteria**

* Criteria: An **Expression<Func<TEntity, bool>>** representing the main filter condition for the specification.
* You can set or add criteria using the constructor or the AddCriteria method.

2. **Includes and ThenIncludes**

* Includes: A list of expressions specifying related entities to include (eager loading), similar to Entity Framework's **.Include()**.
* ThenIncludes: A dictionary mapping an include to a list of further includes, supporting nested eager loading (like **.ThenInclude()** in EF).

3. **Ordering**

* OrderBy and OrderByDescending: Expressions specifying how to order the results.

4. **Paging**

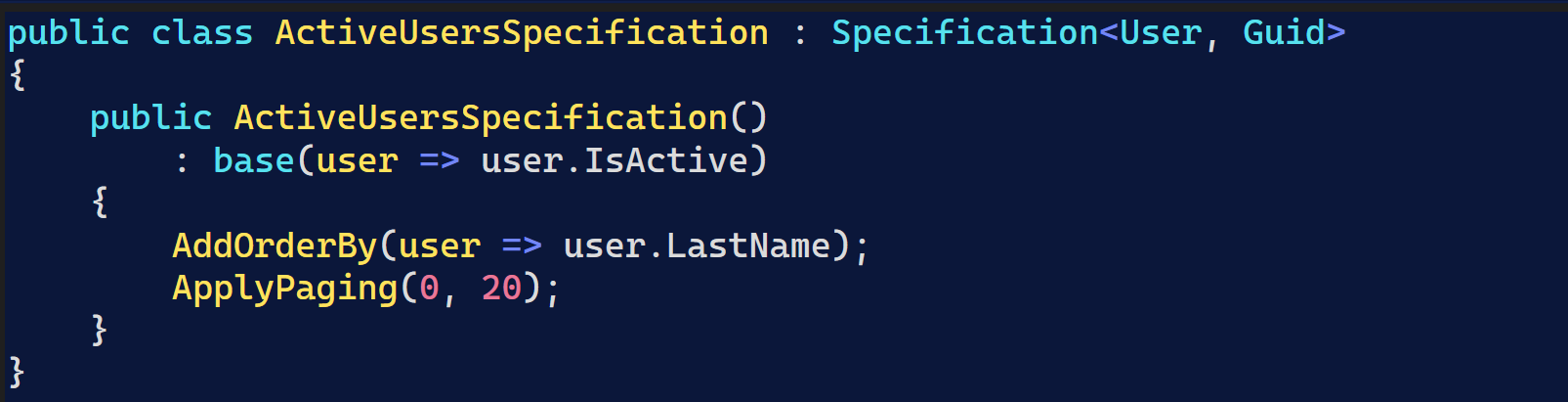
* Take and Skip: Used for pagination (how many records to take and how many to skip).
* IsPagingEnabled: Indicates if paging is applied.
* Use **ApplyPaging(skip, take)** to set these values.

5. **Protected Methods for Building Specifications**

* AddInclude, AddThenInclude, AddOrderBy, AddOrderByDescending, ApplyPaging, AddCriteria: These methods are used by derived classes to build up the specification.

Example Usage

A typical usage is to create a concrete specification by inheriting from one of these base classes and configuring the criteria, includes, ordering, and paging in the constructor:



# Generic Specification

The GenericSearchSpecification classes are concrete implementations of the Specification pattern, designed to provide a flexible, reusable way to handle common search, sorting, and paging scenarios for entities in your application.

Purpose

Encapsulate search logic: They take a SearchRequest object and translate its parameters into query criteria, sorting, and paging instructions.

Promote reusability: By using a generic approach, you can apply the same search logic to any entity type without duplicating code.

Integrate with repositories: These specifications can be passed to repository methods to generate queries dynamically based on user input or API parameters.

How It Works

Constructor Logic

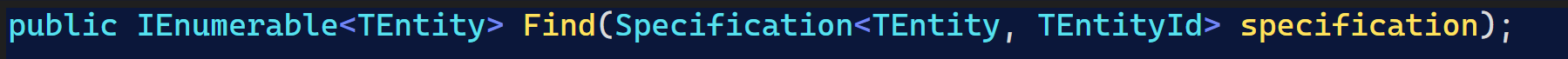
* Criteria: If SearchValue is empty, a default criteria (e => true) is added, meaning all entities are included. (You can extend this to add actual search logic based on SearchValue.)
* Sorting: If SortOrderColumn is provided, it uses reflection to build an order-by expression for that property, and applies ascending or descending order based on SortOrder.  
  If not provided, it defaults to ordering by a dummy expression (e => true).
* Paging:Sets the page size and applies paging using Skip and PageSize from the SearchRequest.

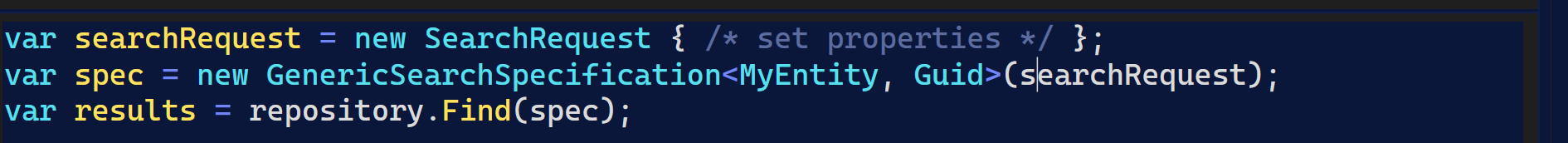
Key Members

* PageSize: Stores the requested page size.
* GetOrderByExpression: Builds a lambda expression for ordering by a property name at runtime.

Example Usage

Suppose you have a repository method like:





EntityByIdSpecification

This specification is typically used to retrieve a single entity by its unique identifier. It encapsulates the logic for filtering entities where the entity’s Id matches a given value.

*EntitiesOrderByIdSpecification*

*This specification is used to retrieve a set of entities, ordered by their* ***Id*** *property, and supports paging (i.e., limiting the number of results).*

# IAggregateRoot

Explanation in class

# ISearchSpecification.cs

The ISearchSpecification interfaces define a contract for building flexible, reusable, and composable search/filtering logic for entities in your application. They are part of the Specification pattern, which helps encapsulate query criteria, sorting, and paging in a consistent way.

There are two versions:

* **ISearchSpecification<TEntity, TEntityId>**: For entities with a generic ID.
* **ISearchSpecification<TEntity>**: For entities without a generic ID.

Key Members

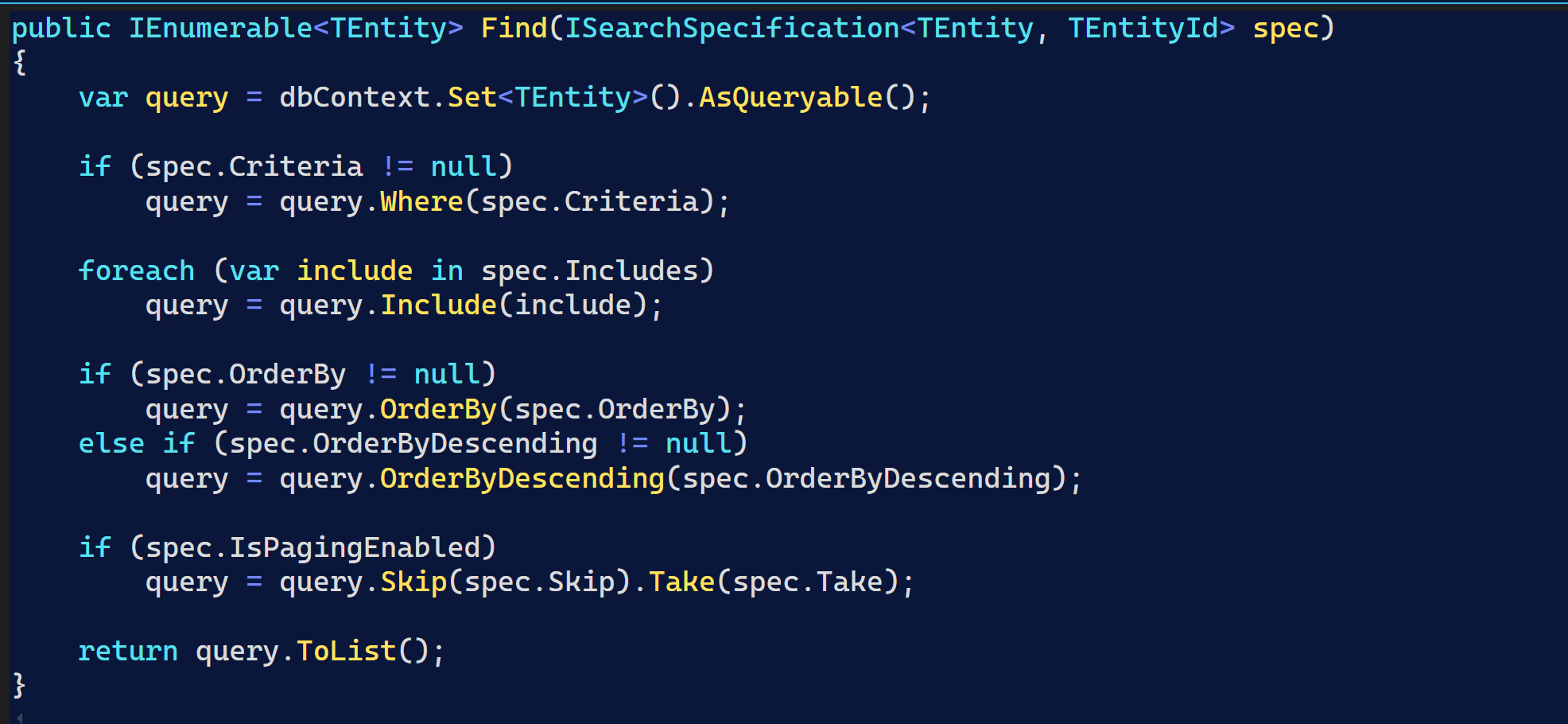
* **Criteria**  
  **Expression<Func<TEntity, bool>>? Criteria**  
  The main filter condition (e.g., "where status = active").
* **Includes**  
  **List<Expression<Func<TEntity, object>>> Includes**  
  Specifies related entities to include (eager loading).
* **OrderBy / OrderByDescending**  
  **Expression<Func<TEntity, object>>? OrderBy**  
  **Expression<Func<TEntity, object>>? OrderByDescending**  
  Expressions for sorting results.
* **Paging**  
  **int Take** — How many records to return.  
  **int Skip** — How many records to skip (for paging).  
  **bool IsPagingEnabled** — Whether paging is applied.  
  **int PageSize { get; set; }** — The size of each page.

---

Purpose

* **Abstraction**: Allows repositories or query handlers to accept specifications and build queries dynamically.
* **Reusability**: You can define specifications once and reuse them across different parts of your application.
* **Separation of Concerns**: Keeps query logic out of services/controllers and encapsulates it in specification objects.

Example Usage



Core Test Project

Add All Files/Folders from CoreTest to Test project, Ensure to change the namespace.

Add the following NugetPackage:

