CST-350 Activity 5 CRUD with n-Layer Design

Introduction

This lesson shows the process of creating a scalable, maintainable, and efficient web application. By the end of this lesson, you will have hands-on experience in implementing key concepts such as CRUD operations, n-layer architecture, Data Transfer Objects (DTOs), ViewModels, Forms, and Image Upload functionalities.

Learning Goals

1 **Structure an application with n-layer design**: Learn how to structure your application into distinct layers, including Data Access, Business Logic, and Presentation.

2 **Implement CRUD operations in a database**: Gain practical experience in creating, reading, updating, and deleting records in a database.

3 **Utilize DTOs and ViewModels with an n-layer design**: Understand the roles of DTOs and ViewModels in data transfer and presentation.

4 **Develop form handling and validation**: Learn to create and manage forms for user input and validation.

5 **Implement image upload functionality**: Add the ability to upload and display images within your application.

6 Utilize the async, await and Task syntax in developing methods that maintain positive UI performance.

Throughout this lesson, you will build a product management application that demonstrates these concepts. The application will include features like listing products, searching, creating, updating, and deleting products, as well as uploading and displaying product images.

Application Preview

The following screen shots will give you an idea of the features of the application.

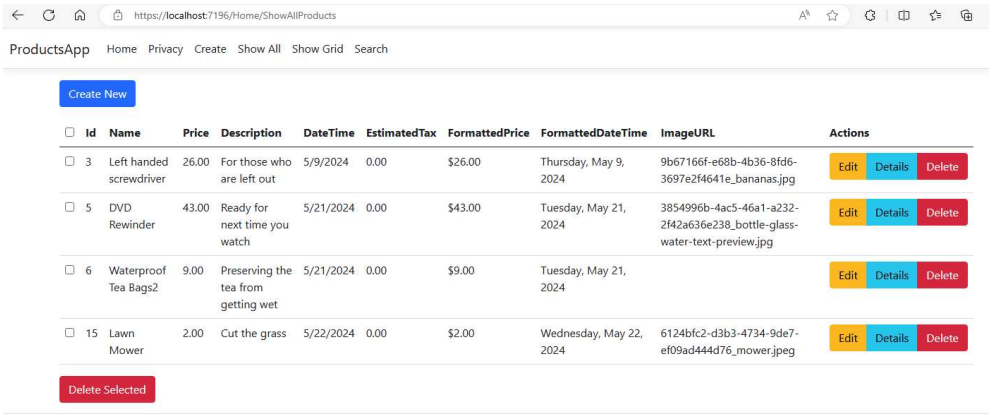


Figure 1 Product app will manage a list of products.

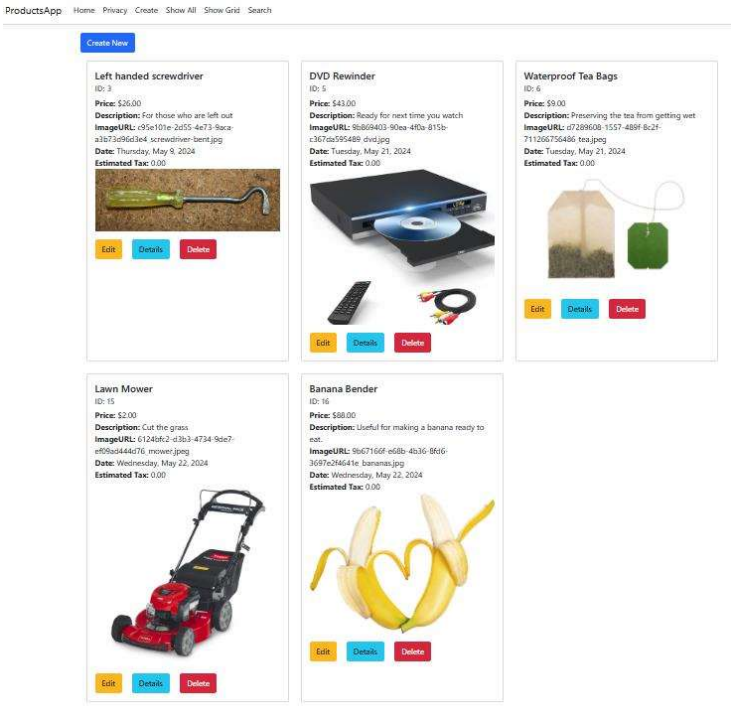


Figure 2 Products can be seen in a grid layout using Bootstrap Card elements.

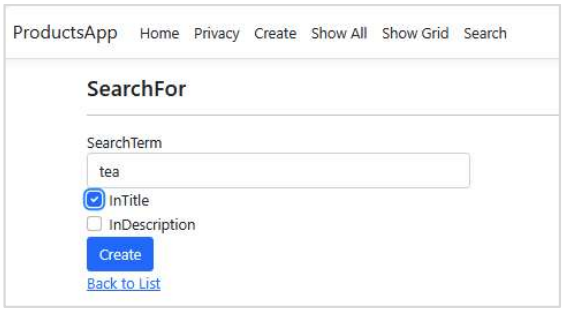


Figure 3 The search function will allow users to find a product.

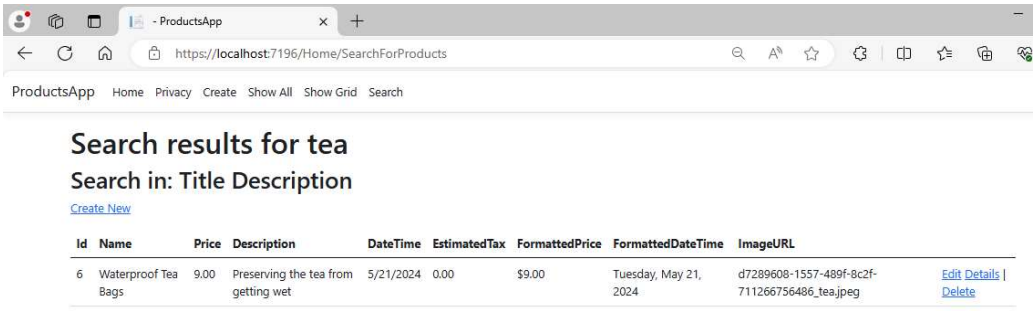


Figure 4 Search results will show matching items.

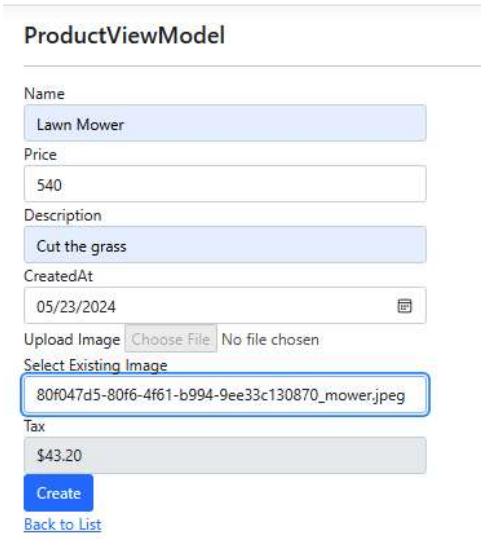
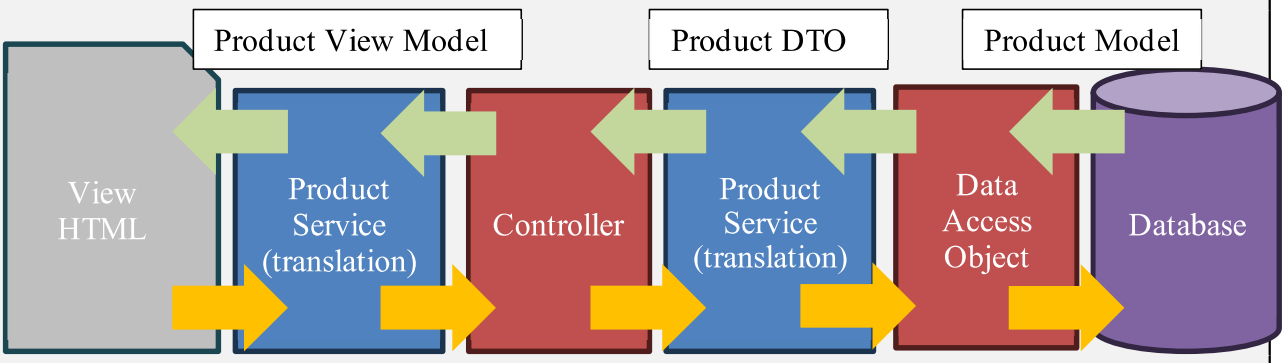


Figure 5 Creating a new product includes assigning or uploading a picture.

Application Design

In addition to the full list of CRUD features, this project will demonstrate the n-layer design structure of an application.



Design of the Application Layers

Some of the classes that we will build include the following

ProductModel: This class represents the *domain model* or *entity* and is used primarily within the data access layer (DAO) and business logic layer. It maps directly to the **database structure** and is used to interact with the data store.
ProductDTO: This class is used for data transfer. It may include additional properties (like EstimatedTax and FormattedPrice) that are not part of the domain model but are necessary for the application to function. Additionally, the data type of the Id property is a **string** instead of an int so that we could migrate to MongoDB without causing a major redesign of the application.
ProductViewModel: This class is specifically designed for the presentation layer. It includes UI-specific properties and **validation attributes**, which are necessary for rendering the product data in views and handling user input.
ProductDAO: This class is responsible for accessing the database. It uses the ProductModel to perform CRUD (Create, Read, Update, Delete) operations. The ProductDAO interacts directly with the database and returns ProductModel objects to the service layer.
ProductService: This class contains business logic and acts as an intermediary between the DAO and the controller. It uses the ProductModel to fetch data from the ProductDAO and then converts it to ProductDTO for data transfer purposes and to ProductViewModel for presentation purposes.
ProductController: This class handles HTTP requests, processes user input, and returns data to the view. It uses the ProductService to get ProductViewModel objects, which it then passes to the view.

Data Flow

Controller calls the Product Service to get product data.
Product Service fetches data from the DAO using the ProductModel.
Product Service converts the ProductModel **to** ProductDTO **for data transfer**.
Product Service converts the ProductDTO **to** ProductViewModel **for presentation**.
Controller uses the ProductViewModel **to pass data to the view**.

Design Principles

Separation of Concerns: Each layer has a specific responsibility, making the codebase more maintainable and scalable.

Data Transfer: DTOs help in transferring data efficiently between the server and client, especially when additional computed or formatted data is needed for the client.

Products List:

1. Create a new web MVC project.

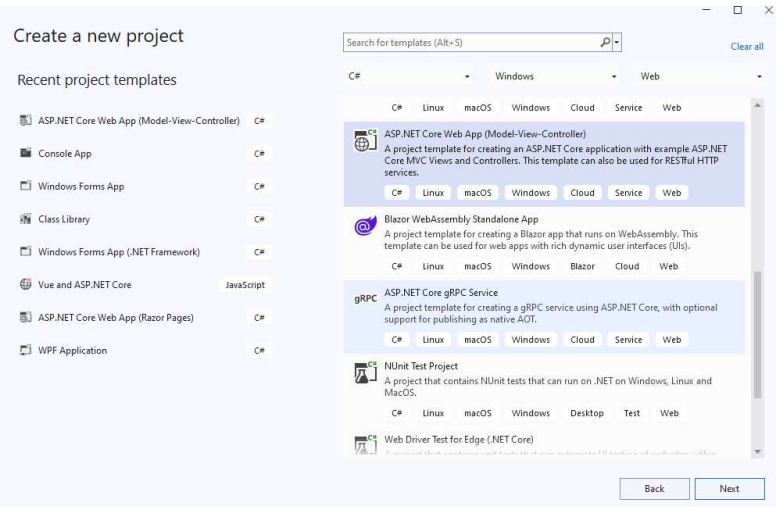


Figure 6 Creating a new web project.

2. Name the project “Poducts App”

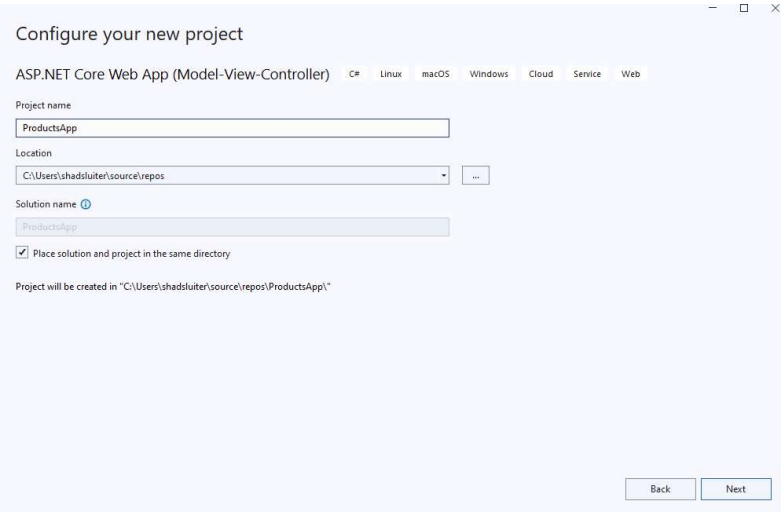


Figure 7 Naming the project ProductsApp

3. Set to current .net framework.

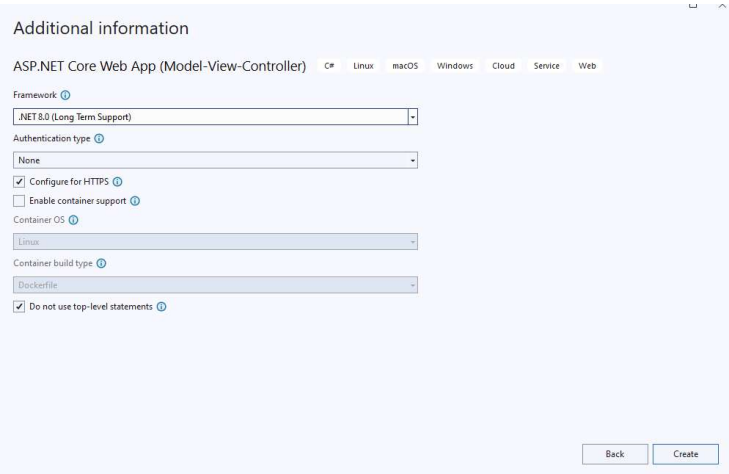


Figure 8 Using .net 8 for this project.

Product Model:

About the ProductModel

The ProductModel class is part of this application's data layer. Each instance of the ProductModel class corresponds to a single product record in the database.

Key Characteristics and Functions:

Properties: The ProductModel class defines properties that **mirror the columns** in your database table (e.g., Id, Name, Price, Description, CreatedAt, ImageURL). These properties store the actual data values for each product.
Data Structure: It provides a structured way to organize and work with product data within your C# code. This makes it easier to manipulate and pass product information between different layers of your application.
Database Interaction: The ProductModel is used by the Data Access Object (DAO) to interact with the database. The DAO translates between the ProductModel objects in your code and the corresponding records in the database table.
Data Transfer: While the ProductModel is primarily used within the data access layer, it can be converted into a Data Transfer Object (DTO) for sending data between layers of your application.

Relationship to the Application:
Database: The ProductModel directly corresponds to the structure of the "Products" table in your database.
Data Access Object (DAO): The DAO uses ProductModel to retrieve data from the database and populate ProductModel objects.
Service Layer: The service layer receives ProductModel objects from the DAO, performs any necessary business logic, and then converts them into DTOs or ViewModels for further processing.

The ProductModel is the core representation of a product within your application's codebase. It is the link between your application's logic and the data stored in the database.

1. Create a new class in the Models folder. Name it ProductModel

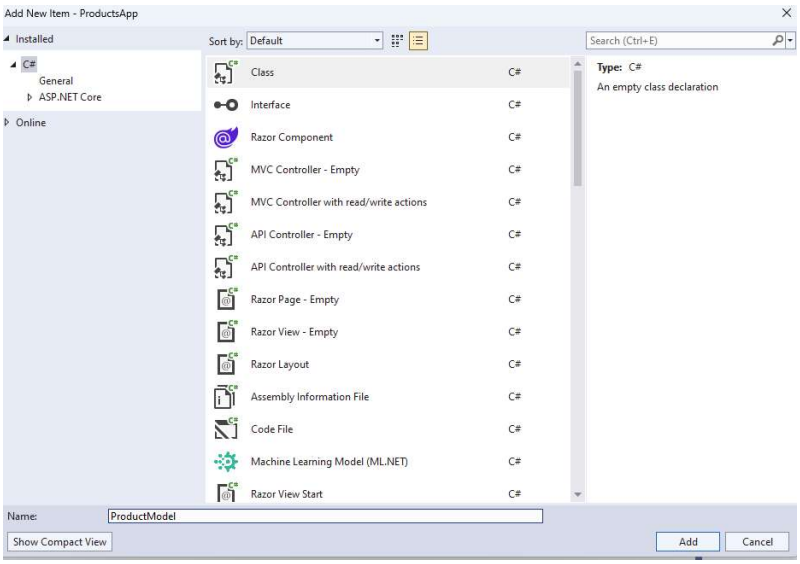


Figure 9 Adding the PorductModel to the models folder.

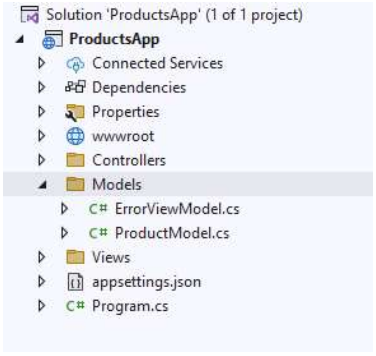


Figure 10 New file is in Models.

```
namespace ProductsApp.Models
{

    // This class is tied closely to the database schema and is used for data access.
    // in order to display products, we will need to convert this model to a ProductDTO before sending it to the view.
    public class ProductModel
    {
        public int Id { get; set; } // sql uses int for id. MongoDB uses string for id. The dto will use string for id for flexibility
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Description { get; set; }
        public DateTime CreatedAt { get; set; }
        public string ImageURL { get; set; }

        public ProductModel(int id, string name, decimal price, string description, DateTime createdAt, string imageURL)
        {
            Id = id;
            Name = name;
            Price = price;
            Description = description;
            CreatedAt = createdAt;
            ImageURL = imageURL;

        }

        public ProductModel()
        {
        }

        public override bool Equals(object obj)
        {
            return Equals(obj as ProductModel);
        }

        public bool Equals(ProductModel other)
        {
            return other != null &&
                   Id == other.Id;
        }

        public override int GetHashCode()
        {
            return Id.GetHashCode();
        }
    }
}
```

Figure 11 ProductModel properties. "Equals" override method defined.

Data Tables:

1. Open the SQL Server Object Explorer
2. Create a new Database

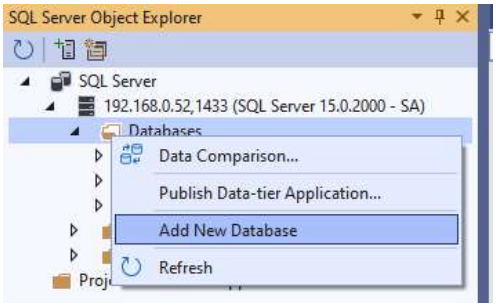


Figure 12 Adding a new database to the SQL Server.

3. Create a ProductsDatabase

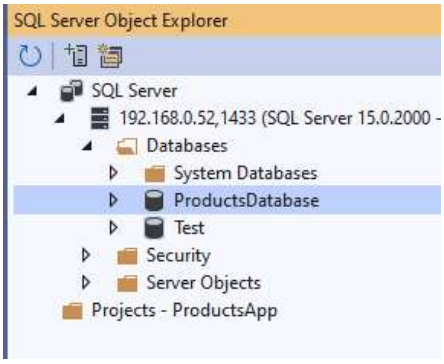


Figure 13 ProductsDatabase is running.

4. Create a new table [dbo].[Products]

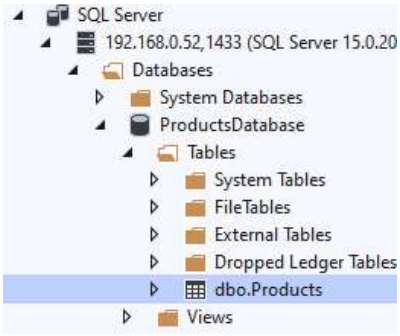


Figure 14 Products table has been added

5. Setup column names to correspond to the ProductModel class properties.

```
CREATE TABLE [dbo].[Products] (
    [Id]          INT             IDENTITY (1, 1) NOT NULL,
    [Name]        NVARCHAR (50)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Description] NVARCHAR (100)  DEFAULT (NULL) NULL,
    [CreatedAt] DATETIME NULL,
    [ImageURL] NCHAR(200) NULL,
    PRIMARY KEY CLUSTERED ([Id] ASC)
);
```

Figure 15 The Products table has six columns.

Update | Script File: dbo.Products.sql

	Name	Data Type	Allow Nulls	Default
🔑	Id	int	☐	
	Name	nvarchar(50)	☐	
	Price	decimal(18,2)	☐	
	Description	nvarchar(100)	☑	(NULL)
	CreatedAt	datetime	☑	
	ImageURL	nchar(200)	☑	
			☐	

Figure 16 Design view of the Products table.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Data Transfer Object:

About the ProductDTO

The ProductDTO (Data Transfer Object) serves as a structured container for transferring product data between different layers of the application. It acts as an intermediary between the data access layer (ProductModel) and the presentation layer (ProductViewModel). The ProductDTO typically mirrors the structure of the ProductModel, containing the essential product information like ID, name, price, description, creation date, and image URL. However, it might exclude fields that are not necessary for data transfer or include additional calculated fields like estimated tax.

In this application, the ProductDTO is primarily used to:

Transfer Data: It facilitates the transfer of product information from the data access layer (where it's retrieved from the database) to the service layer (where it might be processed or transformed) and finally to the controller.

Prepare Data for Presentation: The ProductDTO can be easily mapped to a ProductViewModel, which is specifically designed for the presentation layer. This separation allows you to add display-specific properties (like formatted prices or dates) to the ViewModel without cluttering the DTO.

1. Create another class in the Models folder.
2. Name it ProductDTO

```
4   {
5       public class ProductDTO
6       {
7           public string? Id { get; set; } // sql uses int for id. MongoDB uses string for id.
8           public string Name { get; set; }
9           public decimal Price { get; set; }
10          public string? Description { get; set; }
11          public DateTime? CreatedAt { get; set; }
12          public string? ImageURL { get; set; }
13      }
14  }
```

Figure 17 Properties of the DTO.

ProductViewModel:

About the ProductViewModel

The ProductViewModel is specifically designed for the presentation layer of the application. It acts as a bridge between the data retrieved from the database (via the ProductDTO) and the information displayed to the user in the views.

Data Presentation: It contains properties that are tailored for display in the user interface. These properties might include formatted prices, dates, or calculated values that are not directly stored in the database.

User Input: It is used to bind data to forms, allowing users to input or edit product information.

Validation: It includes validation attributes (e.g., Required, Range) that ensure the data entered by the user meets specific criteria before it's submitted to the server.

UI Customization: It can include additional properties that control the appearance or behavior of UI elements, such as whether a product is on sale or has a discounted price.

By using a ProductViewModel, you can keep your views clean and focused on presentation logic, while the underlying data and business logic are handled by the ProductDTO and ProductService.

1. Create another class in the Models folder. Name it ProductViewModel.

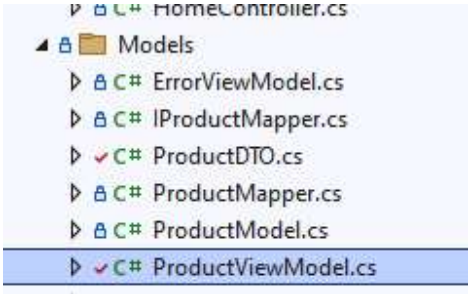
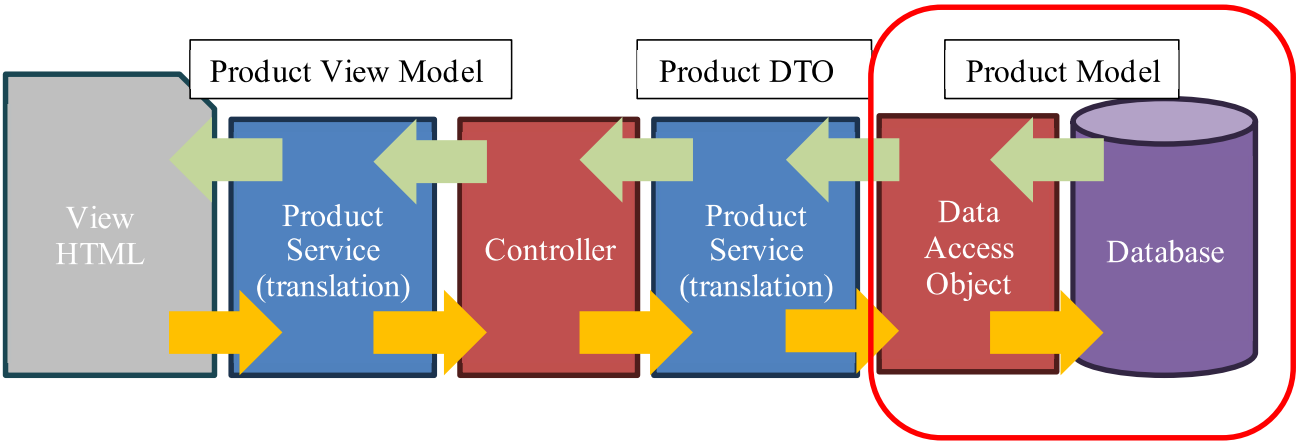


Figure 18 Location of the ProductViewModel.

```
using System;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Http;

namespace ProductsApp2.Models
{
    public class ProductViewModel
    {
        public string? Id { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        [Range(0, double.MaxValue, ErrorMessage = "Price must be a positive value")]
        public decimal Price { get; set; } // price of the product used for data entry

        [Display(Name = "Price")]
        public string? FormattedPrice { get; set; }  // formatted price for display


        [Required]
        public string? Description { get; set; }

        [DataType(DataType.Date)]
        public DateTime CreatedAt { get; set; } // used for data entry

        [Display(Name = "Created At")]
        public string? FormattedDateTime { get; set; } // formatted date for display

        public string? ImageURL { get; set; } // allowed to be null.

        [Display(Name = "Upload Image")]
        public IFormFile? ImageFile { get; set; } // allowed to be null.
                                                  // One of ImageURL (chosen) or ImageFile (upload) will be provided.


        public decimal EstimatedTax { get; set; } // estimated tax for the product

        [Display(Name = "Tax")]
        public string? FormattedEstimatedTax { get; set; } // formatted tax for display

    }
}
```

Figure 19 ProductViewModel properties

Data Access in ProductDAO:



About ProductDAO
The ProductDAO (Data Access Object) is responsible for directly interacting with the database in this application. It provides methods to perform CRUD (Create, Read, Update, Delete) operations on product data. The ProductDAO abstracts the underlying database implementation details, allowing the rest of the application to work with product data without knowing the specifics of how it's stored or retrieved.

This version of the ProductDAO is written specifically to work with an MS SQL database. Other versions of SQL or MongoDB could also be written using these exact method names.

AddProduct: Inserts a new product record into the database.
GetAllProducts: Retrieves all product records from the database.
GetProductById: Retrieves a specific product record by its ID.
UpdateProduct: Modifies an existing product record in the database.
DeleteProduct: Removes a product record from the database.
SearchForProductsByName: Searches for products by name.
SearchForProductsByDescription: Searches for products by description.

By encapsulating these database operations, the ProductDAO simplifies data access and promotes a clean separation between the data layer and the rest of the application.

1. Create a new folder in the project named Services

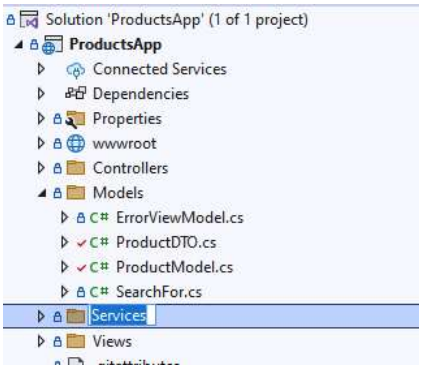


Figure 20 Services folder is now part of the project.

2. Create a new file in the services folder. Name it IProductDAO

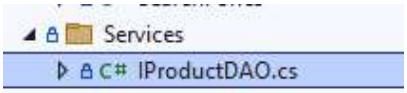


Figure 21 Interface created for IProductDAO

3. Make the file an interface with the following method names.

```
using System.Collections.Generic;
using System.Threading.Tasks;
using ProductsApp.Models;

namespace ProductsApp.Services
{
    public interface IProductDAO
    {
        Task<IEnumerable<ProductModel>> GetAllProducts();
        Task<ProductModel> GetProductById(int id);
        Task<int> AddProduct(ProductModel product);
        Task UpdateProduct(ProductModel product);
        Task DeleteProduct(ProductModel product);
        Task<IEnumerable<ProductModel>> SearchForProductsByName(string searchTerm);
        Task<IEnumerable<ProductModel>> SearchForProductsByDescription(string searchTerm);
    }
}
```

Figure 22 Method names for the DAO are listed in the interface.

About Asynchronous Programming

The methods listed in the interface have a Task return type. This is a feature of **asynchronous programming**. In standard, synchronous, programming, the flow of events is strictly sequential. However, with multiple CPUs and the long-running nature of some processes, it makes sense to start multiple processes at the same time and run instructions in parallel. In cases where one event does not need to wait for results from another it is more efficient if both processes start at the same time.

What Does async Mean?
In C#, asynchronous methods are defined using the **async** keyword. They are designed to allow non-blocking operations, meaning they can perform tasks (such as I/O operations, network calls, or **database queries**) without blocking the main thread. This is particularly useful in applications where maintaining a responsive user interface is important.

Why Use async?
Responsiveness: In GUI applications, using async methods helps keep the user interface responsive. Long-running operations like database access or network calls won't freeze the application.
Scalability: In web applications, async methods help handle more requests concurrently. They free up threads to handle other requests while waiting for I/O operations to complete.
Efficiency: Async operations make better use of system resources by allowing threads to do other work while waiting for I/O operations.

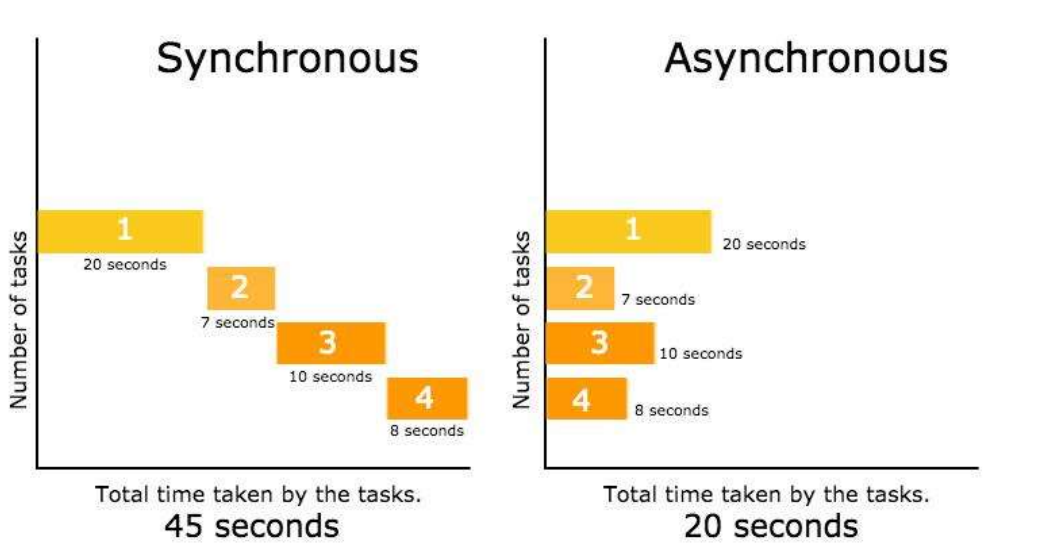


Figure 23 Asynchronous events happen in parallel.

Differences in Coding Syntax

Method Signature:

Synchronous method:

```
public IEnumerable<ProductModel> GetAllProducts()
```

Asynchronous method:

```
public async Task<IEnumerable<ProductModel>> GetAllProducts()
```

Return Type:

Async methods typically return a **Task** or **Task<T>** (e.g., **Task**, **Task<int>**, **Task<IEnumerable<ProductModel>>**).

This allows the method to be awaited.

await Keyword:

Within an async method, you use the **await** keyword to call another async method.

```
var products = await   productDAO.GetAllProducts();
```

Key Points for Developers

Marking Methods as async:

When you define a method as async, you use the **async** keyword in the method signature.

Returning Task:

Async methods should return **Task** or **Task<T>**. This indicates that the method performs an asynchronous operation.

Using await:

Inside an async method, use **await** to call other async methods. This pauses the execution of the method until the awaited task completes.

ProductDAO:

1. Create a new class in the Services folder. Name it ProductDAO.

Figure 24 ProductDAO file is in Services

2. Apply the IProductDAO interface to the class.
3. Copy the connection string from the MS SQL database. Refer to the Register and Login app from Activity 2 for review. Later, we will remove this string to the appsettings.json file.

```
1   using System.Collections.Generic;
2   using System.Data.SqlClient;
3   using System.Threading.Tasks;
4   using ProductsApp.Models;
5
6   namespace ProductsApp.Services
7   {
8       public class ProductDAO : IProductDAO
9       {
10          private readonly string _connectionString = @"Data Source=192.168.0.52,1433;Initial
              Catalog=ProductsDatabase;User ID=SA;Password=sqlserverpassword123!@#;Connect
              Timeout=30;Encrypt=True;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailove
              r=False";
11
12      }
13  }
```

Figure 25 Connection string for the Azure Edge SQL server.

4. Implement the interface.

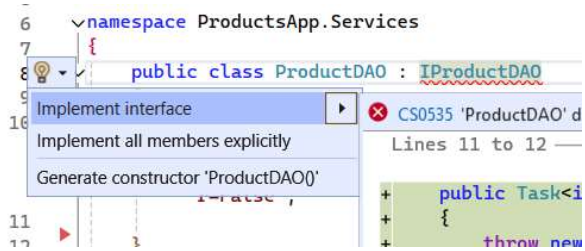


Figure 26 Implementing the methods in the Interface.

5. You should see all of the methods from the IProductDAO.

```
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Threading.Tasks;
using ProductsApp.Models;

namespace ProductsApp.Services
{
    public class ProductDAO : IProductDAO
    {
        private readonly string _connectionString = @"Data Source=192.168.0.52,1433;Initial Catalog=ProductsDatabase;User ID=SA;Password=sqlserverpassword123!
          @#;Connect Timeout=30;Encrypt=True;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

        public Task<int> AddProduct(ProductModel product)
        {
            throw new NotImplementedException();
        }

        public Task DeleteProduct(ProductModel product)
        {
            throw new NotImplementedException();
        }

        public Task<IEnumerable<ProductModel>> GetAllProducts()
        {
            throw new NotImplementedException();
        }

        public Task<ProductModel> GetProductById(int id)
        {
            throw new NotImplementedException();
        }

        public Task<IEnumerable<ProductModel>> SearchForProductsByDescription(string searchTerm)
        {
            throw new NotImplementedException();
        }

        public Task<IEnumerable<ProductModel>> SearchForProductsByName(string searchTerm)
        {
            throw new NotImplementedException();
        }

        public Task UpdateProduct(ProductModel product)
        {
            throw new NotImplementedException();
        }
    }
}
```

Figure 27 Methods are created, but do not have any useful code.

6. Complete the AddProduct method.
7. Mark the method as “async”.

```
namespace ProductsApp.Services
{
    public class ProductDAO : IProductDAO
    {
        private readonly string _connectionString = @"Data Source=192.168.0.52,1433;Initial Catalog=ProductsDatabase;User ID=SA;Password=sqlserverpassword123!
          @#;Connect Timeout=30;Encrypt=True;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

        public async Task<int> AddProduct(ProductModel product)
        {
            using (SqlConnection conn = new SqlConnection(_connectionString))
            {
                string query = "INSERT INTO Products (Name, Price, Description, CreatedAt, ImageURL) OUTPUT INSERTED.Id VALUES (@Name, @Price, @Description,
                  @CreatedAt, @ImageURL)";
                SqlCommand cmd = new SqlCommand(query, conn);
                cmd.Parameters.AddWithValue("@Name", product.Name);
                cmd.Parameters.AddWithValue("@Price", product.Price);
                cmd.Parameters.AddWithValue("@Description", product.Description);
                cmd.Parameters.AddWithValue("@CreatedAt", product.CreatedAt);
                cmd.Parameters.AddWithValue("@ImageURL", product.ImageURL);

                conn.Open();
                int newId = (int)await cmd.ExecuteScalarAsync();
                return newId; // returns the id of the newly added product
            }
        }
```

Figure 28 The method connects to the database, inserts a record and returns the Id number of the new product record.

8. You may have to install the NuGet package for SQL database access. You will do this when resolving the SQLConnection class.

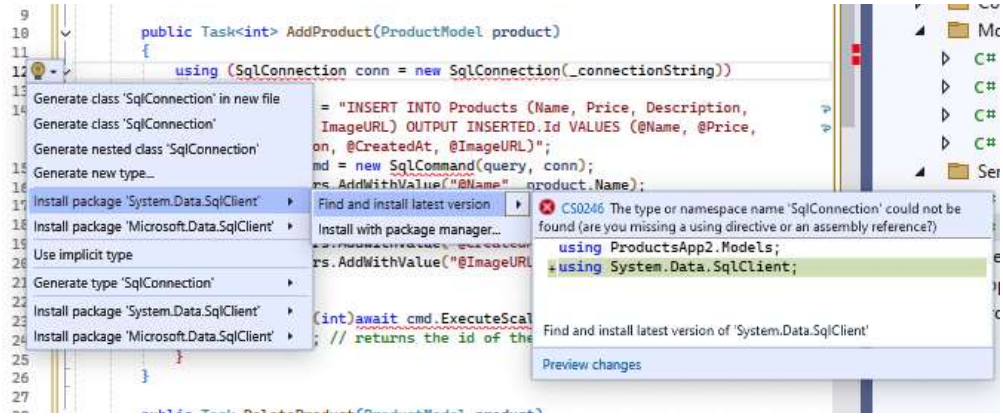


Figure 29 Adding a dependency to the project to work with SQL databases.

9. You can check in the NuGet package manager to verify that the System.Data.SqlClient is active.

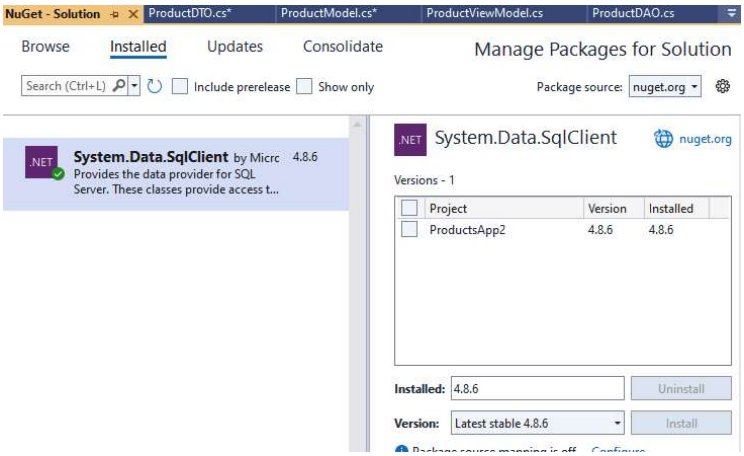


Figure 30 Installed packages include the System.Data.SqlClient class.

10. Complete the GetAllProducts method.
11. Mark the method as an “async” operation.

```
42      public async Task<IEnumerable<ProductModel>> GetAllProducts()
43      {
44          List<ProductModel> products = new List<ProductModel>();
45
46          using (SqlConnection conn = new SqlConnection(_connectionString))
47          {
48              SqlCommand cmd = new SqlCommand("SELECT * FROM Products", conn);
49              conn.Open();
50              SqlDataReader reader = await cmd.ExecuteReaderAsync();
51
52              while (reader.Read())
53              {
54                  products.Add(new ProductModel
55                  {
56                      Id = reader.GetInt32(0),
57                      // if a column is null, return an empty string or 0
58                      Name = reader.IsDBNull(1) ? string.Empty : reader.GetString(1),
59                      Price = reader.IsDBNull(2) ? 0 : reader.GetDecimal(2),
60                      Description = reader.IsDBNull(3) ? string.Empty : reader.GetString(3),
61                      CreatedAt = reader.IsDBNull(4) ? DateTime.MinValue : reader.GetDateTime(4),
62                      ImageURL = reader.IsDBNull(5) ? string.Empty : reader.GetString(5)
63                  });
64              }
65          }
66          return products;
67      }
```

Figure 31 GetAllProducts has been completed.

We will leave the other CRUD methods in ProductDAO for later. In the meantime, let's create the rest of the layers so we can test the Create and Read functionality.

Service Class:

The service class's role in this application is to translate one type of object to the other. This translation maps the properties of one class to the corresponding properties of another.

About ProductService

The ProductService acts as the central orchestrator and business logic layer for product-related operations in this application. It's the bridge between the data access layer (ProductDAO) and the presentation layer (Controllers and Views).

Data Retrieval: Fetches product data (as ProductModel objects) from the ProductDAO.
Data Transformation: Converts ProductModel objects to ProductDTO objects for transfer and ProductViewModel objects for presentation in the UI.
Business Logic: Encapsulates business rules and calculations related to products, such as calculating estimated taxes, formatting prices, or determining if a product is on sale.
Validation: Can optionally include validation logic to ensure the integrity of product data before saving or updating it in the database.
Coordination: Coordinates interactions between the controller and the ProductDAO, abstracting away the details of data access and transformation.
Search Functionality: Implements logic to search for products based on specific criteria, such as name or description.

By handling these tasks, the ProductService keeps the controller and view layers clean and focused on their respective responsibilities. It also makes the application more maintainable and testable, as the business logic is centralized in one place.

1. Create a new file in the Services folder. Name it IproductService.

Figure 32 Creating an interface to define the ProductService class.

2. Define the methods of the Service Class. The main difference between this interface and the previous is that the service class uses the ProductViewModel class.

```
using System.Collections.Generic;
using System.Threading.Tasks;
using ProductsApp.Models;

namespace ProductsApp.Services
{
    public interface IProductService
    {
        Task<IEnumerable<ProductViewModel>> GetAllProducts();
        Task<ProductViewModel> GetProductById(int id);
        Task<int> AddProduct(ProductViewModel product);
        Task UpdateProduct(ProductViewModel product);
        Task DeleteProduct(string Id);
        Task<IEnumerable<ProductViewModel>> SearchForProducts(SearchFor searchTerm);
    }
}
```

Figure 33 ProductService methods

3. Create another file in the Services folder. Name it ProductService

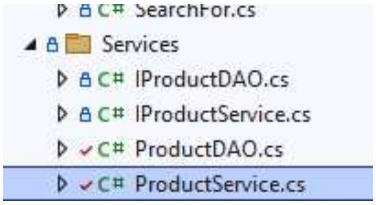


Figure 34 Location of ProductService class.

4. Apply the IProductService interface.

```
using ProductsApp.Models;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Threading.Tasks;

namespace ProductsApp.Services
{
    public class ProductService : IProductService
    {

    }
}
```

Figure 35 Implementing the interface

5. Implement the interface.

Figure 36 Right-clicking on the IProductService interface.

6. You should see the unfinished methods in the class.

```
using ProductsApp2.Models;

namespace ProductsApp2.Services
{
    public class ProductService : IProductService
    {
        public Task<int> AddProduct(ProductViewModel product)
        {
            throw new NotImplementedException();
        }

        public Task DeleteProduct(string Id)
        {
            throw new NotImplementedException();
        }

        public Task<IEnumerable<ProductViewModel>> GetAllProducts()
        {
            throw new NotImplementedException();
        }

        public Task<ProductViewModel> GetProductById(int id)
        {
            throw new NotImplementedException();
        }

        public Task<IEnumerable<ProductViewModel>> SearchForProducts(SearchFor searchTerm)
        {
            throw new NotImplementedException();
        }

        public Task UpdateProduct(ProductViewModel product)
        {
            throw new NotImplementedException();
        }
    }
}
```

Figure 37 Methods are in the ProductService class but not yet coded.

Resolve SearchFor:

1. You will notice that the SearchForProducts method has an unresolved class. We will deal with searching at a later point. For now, let's create a class and leave it empty.
2. Right click and choose "Quick actions" > Generate class 'SearchFor' in a new file.

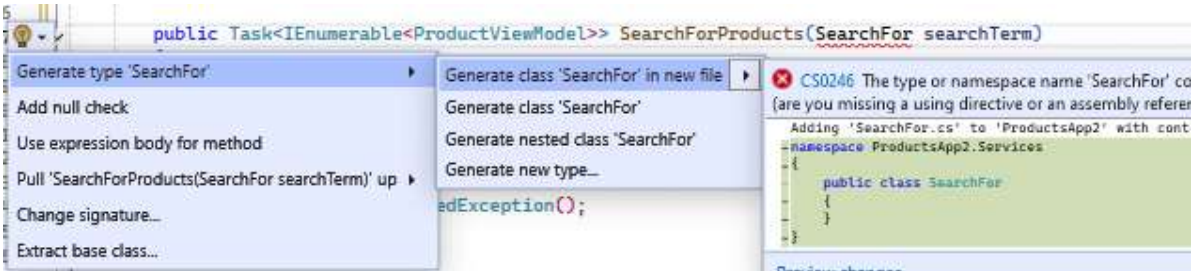


Figure 38 Creating a class to resolve the error.

3. "SearchFor" shows in the Services folder for now. Leave the class empty. No code yet.

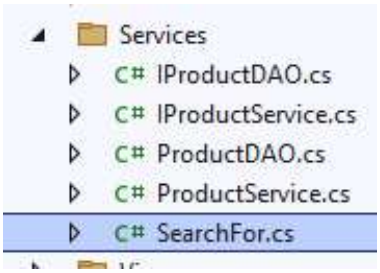


Figure 39 SearchFor exists, but does not contain any useful code.

Create and Show All Operations:

1. Add the following variables to the Product Service class: _productDAO and _productMapper.
2. Create a constructor.

```
using ProductsApp2.Models;

namespace ProductsApp2.Services
{
    public class ProductService : IProductService
    {

        private readonly IProductDAO _productDAO;
        private readonly IProductMapper productMapper;

        public ProductService(IProductDAO productDAO, IProductMapper productMapper)
        {
            _productDAO = productDAO;
            this.productMapper = productMapper;
        }
```

Figure 40 Injecting two classes into the ProductService class.

3. To resolve the error, create the IProductMapper interface in the Models folder

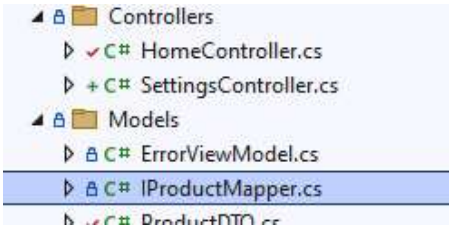


Figure 41 Creating the interface file for ProductMapper.

4. Add the following methods to the interface.

```
using ProductsApp.Models;

public interface IProductMapper
{
    ProductDTO ToDTO(ProductModel model);
    ProductModel ToModel(ProductDTO dto);
    ProductDTO ToDTO(ProductViewModel viewModel);
    ProductViewModel ToViewModel(ProductDTO dto);
}
```

Figure 42 Product mapper's methods.

5. Create the ProductMapper class in the Models folder.

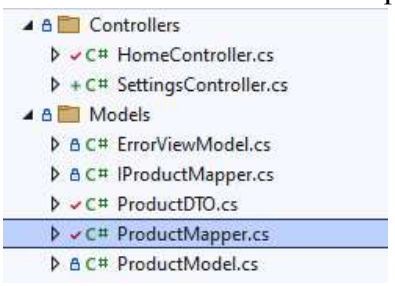


Figure 43 Location of the ProductMapper class.

6. Implement the IProductMapper interface.

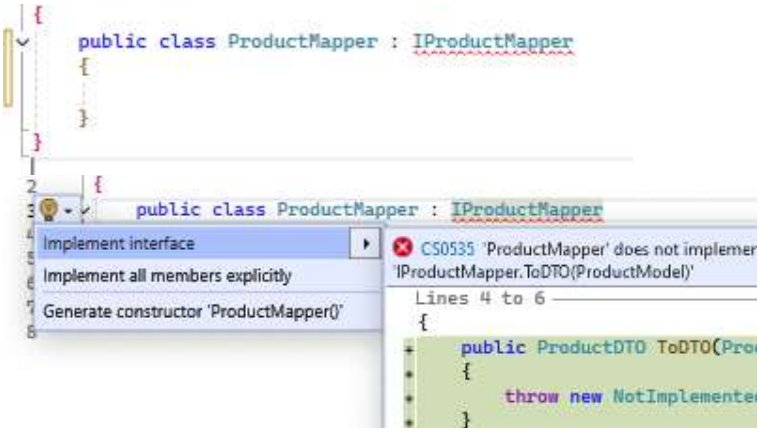


Figure 44 Implementing the interface.

```
public class ProductMapper : IProductMapper
{
    public ProductDTO ToDTO(ProductModel model)
    {
        throw new NotImplementedException();
    }

    public ProductDTO ToDTO(ProductViewModel viewModel)
    {
        throw new NotImplementedException();
    }

    public ProductModel ToModel(ProductDTO dto)
    {
        throw new NotImplementedException();
    }

    public ProductViewModel ToViewModel(ProductDTO dto)
    {
        throw new NotImplementedException();
    }
}
```

Figure 45 Methods have been created but contain no useful code yet.

7. Set default values to be used in conversions. Later, we will move these values to the appsettings.json file.

```
using ProductsApp.Models;

public class ProductMapper : IProductMapper
{
    public string CurrencyFormat { get; set; } = "C"; // currency format. Like "$1,234.56"
    public string DateFormat { get; set; } = "D"; // long date format. Like "Monday, June 15, 2029"
    public decimal TaxRate { get; set; } = 0.08m; // 8% tax rate default

    public ProductMapper(string currency, string dateFormat, decimal tax)
    {
        CurrencyFormat = currency;
        DateFormat = dateFormat;
        TaxRate = tax;
    }
```

Figure 46 Three variables contains values that will be used to format data types and calculate taxes.

8. Create conversion methods for each of the models.

The first two conversions transfer between the database class and the controller.

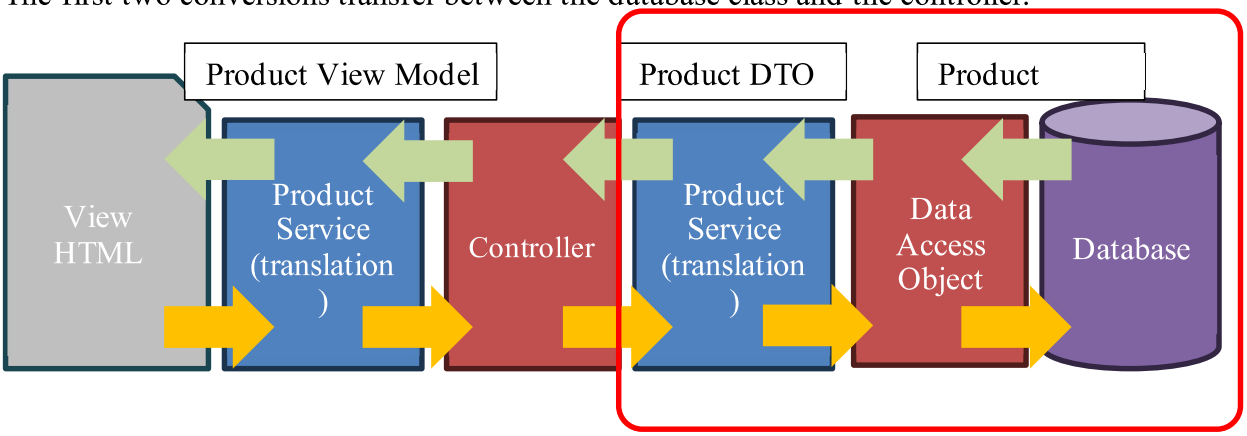


Figure 47 Focus shifts to the transition between the Database and the Service class.

```
16      // convert from ProductModel to ProductDTO
17      public ProductDTO ToDTO(ProductModel model)
18      {
19          return new ProductDTO
20          {
21              Id = model.Id.ToString(), // convert the int id to string
22              Name = model.Name,
23              Price = model.Price,
24              Description = model.Description,
25              CreatedAt = model.CreatedAt,
26              ImageURL = model.ImageURL,
27              EstimatedTax = model.Price * TaxRate,
28              FormattedPrice = model.Price.ToString(CurrencyFormat),
29          };
30      }
```

Figure 48 First conversion method maps the Product Model to the ProductDTO

```
31
32      // convert from ProductDTO to ProductModel
33      public ProductModel ToModel(ProductDTO dto)
34      {
35          if (dto.Id == null)
36          {  // comes from "Create" form.  Id is not set yet. Database will assign it.
37              return new ProductModel
38              {
39                  Name = dto.Name,
40                  Price = dto.Price,
41                  Description = dto.Description,
42                  CreatedAt = dto.CreatedAt,
43                  ImageURL = dto.ImageURL ?? string.Empty
44              };
45          }
46          // if Id is set, then it is an existing product
47          return new ProductModel
48          {
49              Id = int.Parse(dto.Id), // convert the string id to int
50              Name = dto.Name,
51              Price = dto.Price,
52              Description = dto.Description,
53              CreatedAt = dto.CreatedAt,
54              ImageURL = dto.ImageURL ?? string.Empty
55          };
56      }
```

Figure 49 This method converts a ProductDTO to a ProductModel.

The third and fourth conversions translate between the controller and the views.

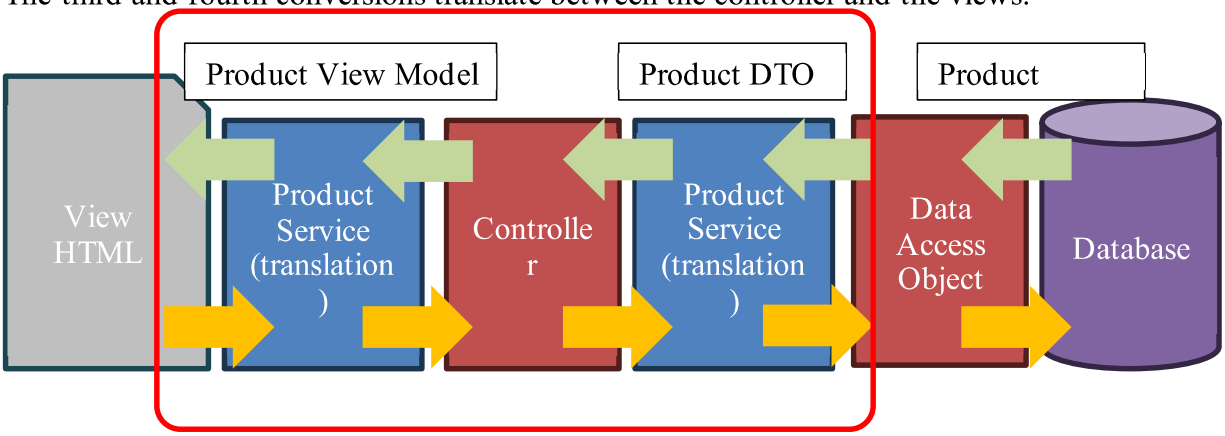


Figure 50 Focus is now on the front-end and controller.

```
57
58      public ProductDTO ToDTO(ProductViewModel viewModel)
59      {
60          // creating a new product. Id is not set yet.
61          if (viewModel.Id == null)
62          {
63              return new ProductDTO
64              {
65                  Name = viewModel.Name,
66                  Price = viewModel.Price,
67                  Description = viewModel.Description,
68                  CreatedAt = viewModel.CreatedAt,
69                  ImageURL = viewModel.ImageURL
70              };
71          }
72          return new ProductDTO
73          {
74              Id = viewModel.Id,
75              Name = viewModel.Name,
76              Price = viewModel.Price,
77              Description = viewModel.Description,
78              CreatedAt = viewModel.CreatedAt,
79              ImageURL = viewModel.ImageURL
80          };
81      }
```

Figure 51 This method converts a ViewModel to a DTO model.

```
82
83      public ProductViewModel ToViewModel(ProductDTO dto)
84      {
85          return new ProductViewModel
86          {
87              Id = dto.Id,
88              Name = dto.Name,
89              Price = dto.Price,
90              Description = dto.Description,
91              CreatedAt = dto.CreatedAt,
92              ImageURL = dto.ImageURL,
93              EstimatedTax = dto.EstimatedTax,
94              FormattedPrice = dto.FormattedPrice,
95              FormattedDateTime = dto.FormattedDateTime
96          };
97      }
98  }
```

Figure 52 This method converts a DTO model to a ViewModel.

Home Controller:

In the product management application, the HomeController plays the role of handling incoming web requests and coordinating the interaction between the user, the view, and the underlying services (like the ProductService). It's essentially the "traffic director" of the application, deciding which views to render based on the user's actions and the requested URLs.

Here's a breakdown of its main responsibilities:

Action Methods: Each public method within the HomeController class represents an "action" that the controller can perform. These actions typically correspond to specific URLs or routes in your application.

Product Actions: These are the core actions for interacting with products:
ShowAllProducts, ShowAllProductsGrid: Display lists of products.

ShowProduct: Show details of a single product.
ShowCreateProductForm, CreateProduct: Handle product creation.
ShowUpdateProductForm, UpdateProduct: Manage product updates.
ShowSearchForm, SearchForProducts: Facilitate product search.
DeleteProduct: Handle product deletion.

Views: The HomeController determines which view to render based on the action being executed. For example, the ShowAllProducts action would render a view that displays a list of products, while the ShowProduct action would render a view with the details of a single product.

Data Interaction: The HomeController often interacts with services like the ProductService to fetch or manipulate data. For instance, the ShowAllProducts action would call the GetAllProducts method of the ProductService to retrieve product data before passing it to the view.

User Input: The HomeController receives input from users through forms and other mechanisms. It then processes this input, performs validation, and passes it to the appropriate service methods for further processing or storage in the database.

Model Binding: The HomeController uses model binding to automatically map incoming request data (e.g., from forms) to ProductViewModel objects, simplifying the handling of user input.

Redirects and Navigation: It manages the flow of the application by redirecting the user to different views or actions based on the outcome of operations (e.g., redirecting to the product list after a successful product creation).

The HomeController is the central coordinator that connects user interactions with the data and business logic of your product management application. It translates user actions into method calls, fetches and prepares data for display.

1. In the HomeController add a new property to use the ProductService class by Dependency Injection. The controller will constantly be sending and receiving data with the ProductService class.

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IProductService _productService; // second item added by dependency injection

    public HomeController(ILogger<HomeController> logger, IProductService productService,)
    {
        _logger = logger;
        _productService = productService;
    }
```

Figure 53 Injecting the ProductService class into the HomeController.

2. In the ProductService class, implement the AddProduct method. Leave the other methods unfinished for the time being.

```
public class ProductService : IProductService
{

    private readonly IProductDAO _productDAO;
    private readonly IProductMapper _productMapper;

    public ProductService(IProductDAO productDAO, IProductMapper productMapper)
    {
        _productDAO = productDAO;
        _productMapper = productMapper;
    }

    public async Task<int> AddProduct(ProductViewModel productViewModel)
    {
        var productDTO = _productMapper.ToDTO(productViewModel);
        var productModel = _productMapper.ToModel(productDTO);
        return await _productDAO.AddProduct(productModel);
    }

    public Task DeleteProduct(string Id)
```

Figure 54 Add Product is the first method in the ProductService class.

3. In the HomeController, add the following methods ShowCreateProductForm, CreateProduct, ShowAllProducts.

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IProductService _productService; // second item added by dependency injection

    public HomeController(ILogger<HomeController> logger, IProductService productService,)
    {
        _logger = logger;
        _productService = productService;
    }
    public IActionResult Index()
    {
        return View();
    }
    public IActionResult Privacy()
    {
        return View();
    }
    public IActionResult ShowCreateProductForm()
    {
        var productViewModel = new ProductViewModel();
        productViewModel.CreatedAt = DateTime.Now;
        return View(productViewModel);
    }
    [HttpPost]
    public async Task<IActionResult> CreateProduct(ProductViewModel productViewModel)
    {
        await _productService.AddProduct(productViewModel);
        return RedirectToAction(nameof(Index));
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
          HttpContext.TraceIdentifier });
    }
}
```

Figure 55 Two methods work in tandem, show the create form and process the new data.

4. Add a new view for the ShowCreateProductsForm.

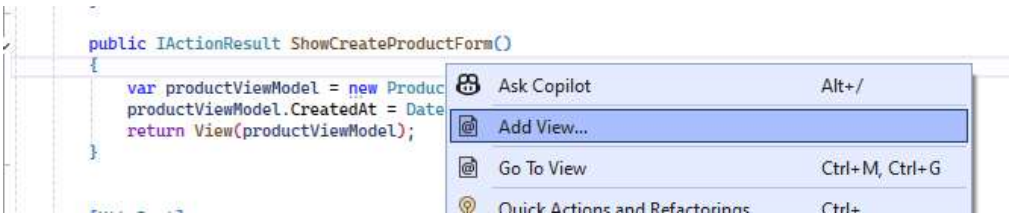


Figure 56 Creating a view.

5. Choose Razor form.

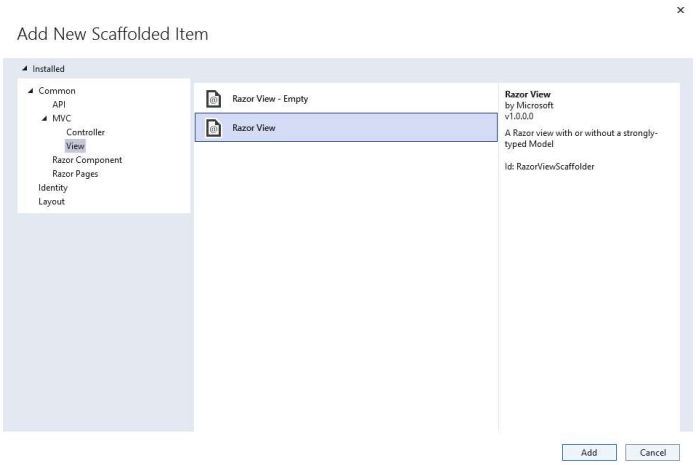


Figure 57 Choosing Razor

6. Choose model.

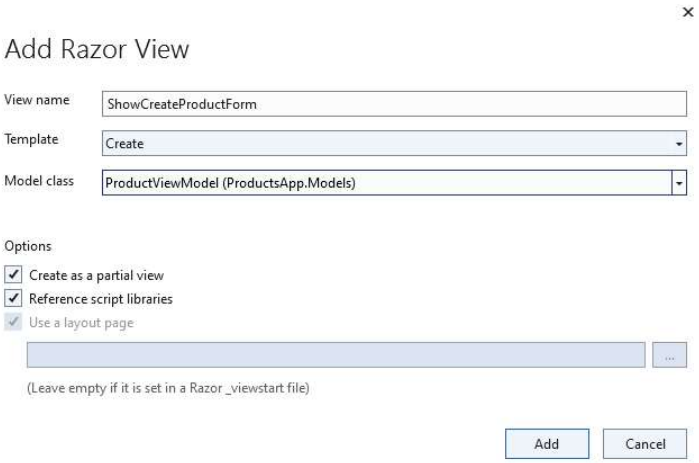


Figure 58 Choosing the Create template with the ViewModel class.

7. You should see the new form creates input fields for every property in the view model. Some of these will be deleted.

```
@model ProductsApp2.Models.ProductViewModel

<h4>ProductViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ShowCreateProductForm">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FormattedPrice" class="control-label"></label>
                <input asp-for="FormattedPrice" class="form-control" />
                <span asp-validation-for="FormattedPrice" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Description" class="control-label"></label>
                <input asp-for="Description" class="form-control" />
                <span asp-validation-for="Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="CreatedAt" class="control-label"></label>
                <input asp-for="CreatedAt" class="form-control" />
                <span asp-validation-for="CreatedAt" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FormattedDateTime" class="control-label"></label>
                <input asp-for="FormattedDateTime" class="form-control" />
                <span asp-validation-for="FormattedDateTime" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ImageURL" class="control-label"></label>
                <input asp-for="ImageURL" class="form-control" />
                <span asp-validation-for="ImageURL" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="EstimatedTax" class="control-label"></label>
                <input asp-for="EstimatedTax" class="form-control" />
                <span asp-validation-for="EstimatedTax" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FormattedEstimatedTax" class="control-label"></label>
                <input asp-for="FormattedEstimatedTax" class="form-control" />
                <span asp-validation-for="FormattedEstimatedTax" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Figure 59 Generated code for the Create form.

8. Modify the view to remove some of the formatted properties. The formatted properties are meant to be used when showing product, not when creating a new product.

```
@model ProductsApp2.Models.ProductViewModel

<h4>ProductViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="CreateProduct">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>

            <!-- removed id -->

            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>

            <!-- removed formatted price -->

            <div class="form-group">
                <label asp-for="Description" class="control-label"></label>
                <input asp-for="Description" class="form-control" />
                <span asp-validation-for="Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="CreatedAt" class="control-label"></label>
                <input asp-for="CreatedAt" class="form-control" />
                <span asp-validation-for="CreatedAt" class="text-danger"></span>
            </div>

            <!-- removed formatted datetime -->

            <div class="form-group">
                <label asp-for="ImageURL" class="control-label"></label>
                <input asp-for="ImageURL" class="form-control" />
                <span asp-validation-for="ImageURL" class="text-danger"></span>
            </div>

            <!-- removed estimated tax -->
            <div class="form-group">
                <label asp-for="FormattedEstimatedTax" class="control-label"></label>
                <input asp-for="FormattedEstimatedTax" class="form-control" readonly />
                <span asp-validation-for="FormattedEstimatedTax" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Figure 60 Modifications to the form to remove unneeded values.

9. Update Program.cs to support **dependency injection** for three classes. You will need to include services in the “using” section to resolve types.

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ProductsApp.Services;
using ProductsApp.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

// Add logging
builder.Services.AddLogging();

// Register application services
builder.Services.AddScoped<IProductDAO, ProductDAO>();
builder.Services.AddScoped<IProductService, ProductService>();

// Register the ProductMapper with constructor parameters
builder.Services.AddSingleton<IProductMapper>(serviceProvider =>
{
    // Specify the values for the parameters here
    string currencyFormat = "C";
    string dateFormat = "D";
    decimal taxRate = 0.08m;

    return new ProductMapper(currencyFormat, dateFormat, taxRate);
});

var app = builder.Build();
```

Figure 61 Program.cs contains the configuration for dependency injection.

10. Run the program.
11. Navigate to localhost home/ShowCreateProductForm

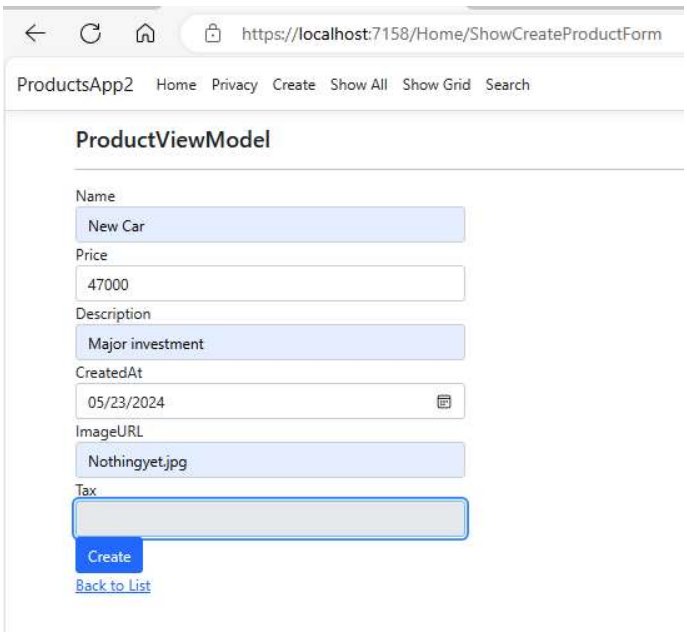


Figure 62 First view of the application - create a new product.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Even though the read only text fields are not used yet, you should be able to create a new product.

12. The CreateProduct method in the HomeController reroutes the browser to the Index method.

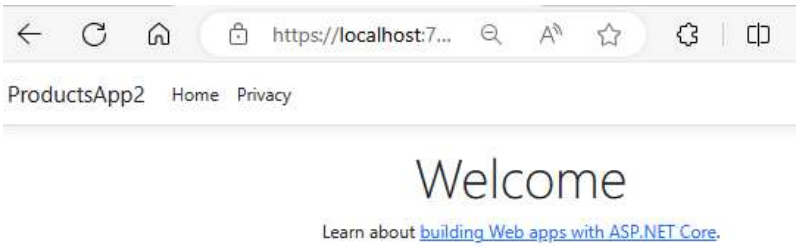


Figure 63 The Home controller returns to the default index page.

13. However, in the background the new product should be stored in the database. You can browse the database in Visual Studio.

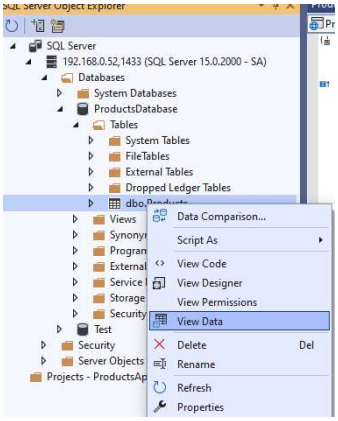


Figure 64 Viewing the contents of the SQL data.

14. You may need to refresh the table contents to see new values.

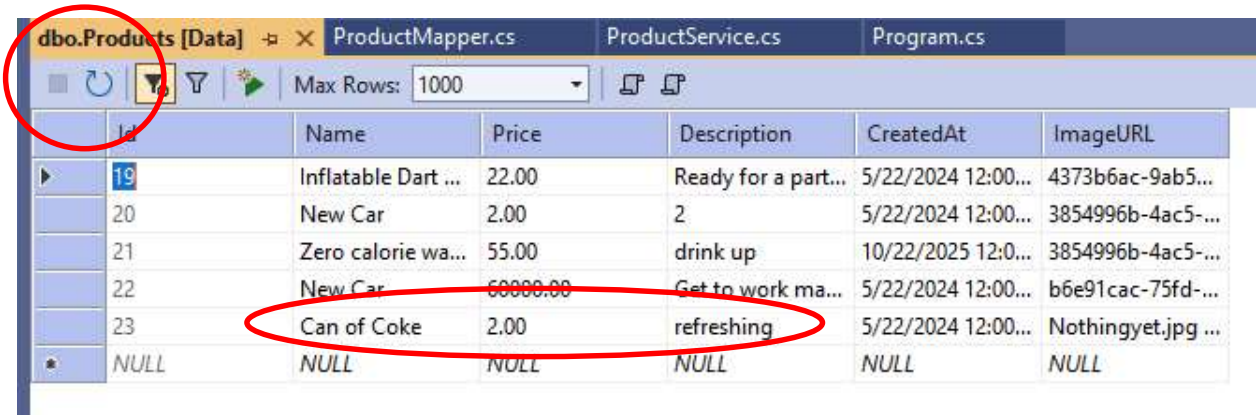


Figure 65 "Can of Coke" was just added to the table.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Show All Products:

1. In the HomeController, create the ShowAllProducts method.
2. The ShowAllProducts method shown in comments is a straightforward request to a service. It might be the first design to come to mind. It would work in our application, but it blocks the UI thread when running. We should add an async and Task addition to the return value.

```
//public IActionResult ShowAllProducts()
//{
//    List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
//    return View(products);
//}

// async method to get all products from the database
public async Task<IActionResult> ShowAllProducts()
{
    IEnumerable<ProductViewModel> products = await _productService.GetAllProducts();
    return View(products);
}
```

Figure 36 An alternate async version of ShowAllProducts

Comparison of the Two Methods

1. **Blocking vs Non-Blocking Calls**:

Synchronous Method: The call to _productService.GetAllProducts().Result blocks the current thread until the task is complete. This can lead to poor performance, especially under high load.
Asynchronous Method: Using await allows the method to be non-blocking, which means the thread is freed up to handle other requests while waiting for the async operation to complete. This improves the application's responsiveness and scalability.

2. **Code Readability and Maintainability**:

Synchronous Method: Using .Result can make the code harder to understand and maintain, as it mixes synchronous and asynchronous patterns.
Asynchronous Method: Consistent use of async/await makes the code more readable and maintainable, as it follows a clear asynchronous pattern.
You should make the HomeController methods asynchronous if the services they call are already async:

Maintaining Non-Blocking Behavior: If the ProductService and ProductDTO services are asynchronous but the controller calls them synchronously (using .Result or .Wait()), the non-blocking benefits of the asynchronous methods are lost. This can lead to thread blocking and undo the performance advantages of async/await.

Consistency: Keeping the entire stack async maintains a consistent programming model, making the code easier to understand and maintain.

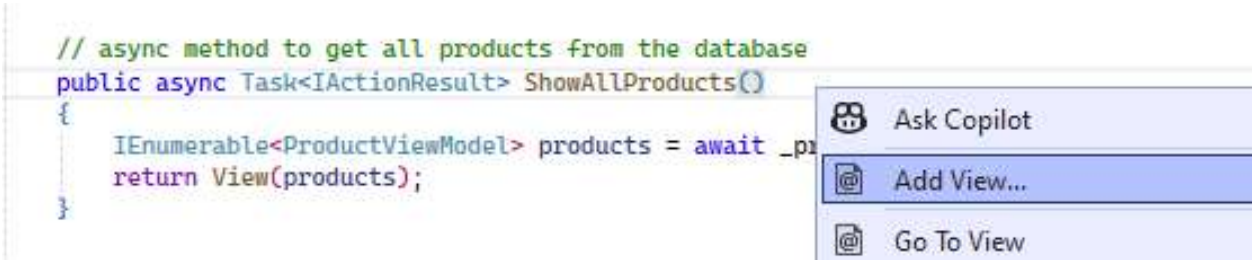


Figure 37 Generating the view to Show All Products.

3. Choose Razor

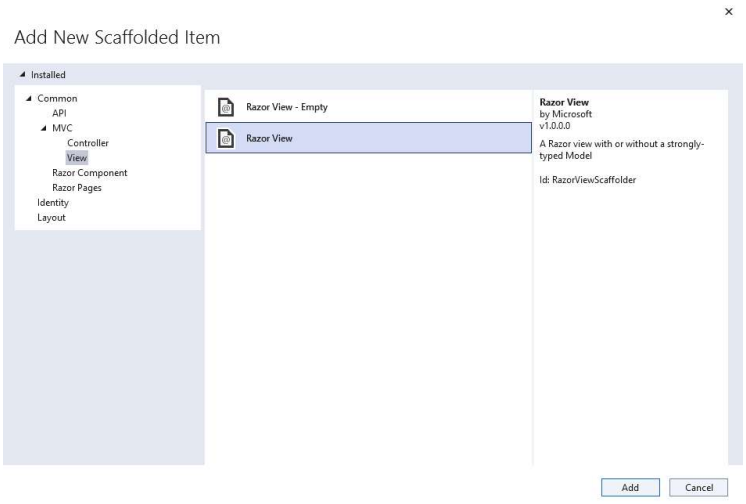


Figure 38 Choosing Razor template

4. Choose List and ProductViewModel.

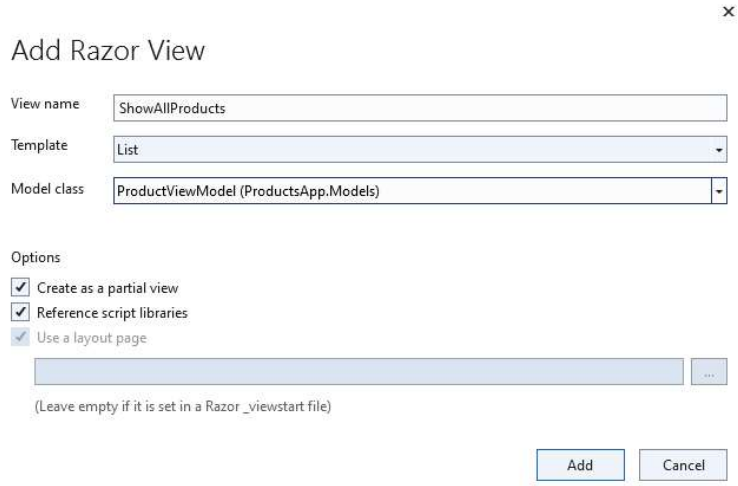


Figure 39 Selecting a list type for ProductViewModel.

9. In the ProductService class implement the GetAllProducts method.
10. Mark the method as async

```
// fetches all products from the database and maps them to the view model
public async Task<IEnumerable<ProductViewModel>> GetAllProducts()
{
    IEnumerable<ProductModel> productModels = await _productDAO.GetAllProducts(); // fetch all
      products from the database

    List<ProductViewModel> productViewModels = new List<ProductViewModel>();

    foreach (ProductModel productModel in productModels)
    {
        ProductDTO productDTO = _productMapper.ToDTO(productModel); // map the product model to
          product DTO

        // opportunity to add additional logic here if needed before mapping to view model

        ProductViewModel productViewModel = _productMapper.ToViewModel(productDTO); // map the
          product DTO to product view model
        productViewModels.Add(productViewModel);
    }
    return productViewModels;
}
```

Figure 40 The ProductService can fetch all products for the controller.

11. In _Layout.cshtml, put a link in the NavBar for Create and Show All action.

```
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">ProductsApp</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="ShowCreateProductForm">Create</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="ShowAllProducts">Show All</a>
                        </li>
```

Figure 41 Adding links to the navbar.

12. Run the program and input a new product.
13. You should be able to add a product and see the list.

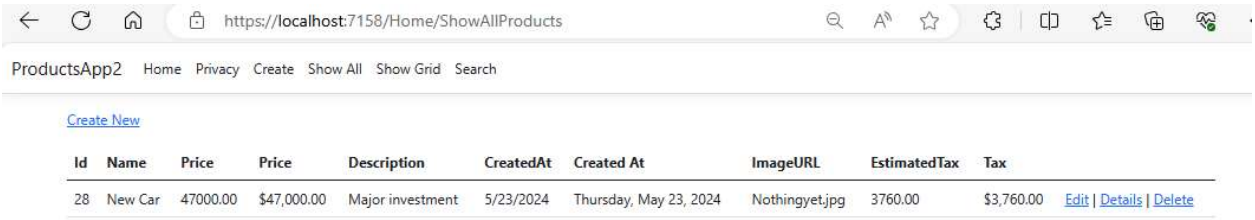


Figure 42 Show All method produces a table of products.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

14. Remove the duplicate columns.
    a. Keep the formatted versions of the Price, CreatedAt and Tax. These are intended to be used for showing product data.
    b. Remove the Price, CreatedAt and EstimatedTax columns. These are intended to be used for input forms when creating or editing data.

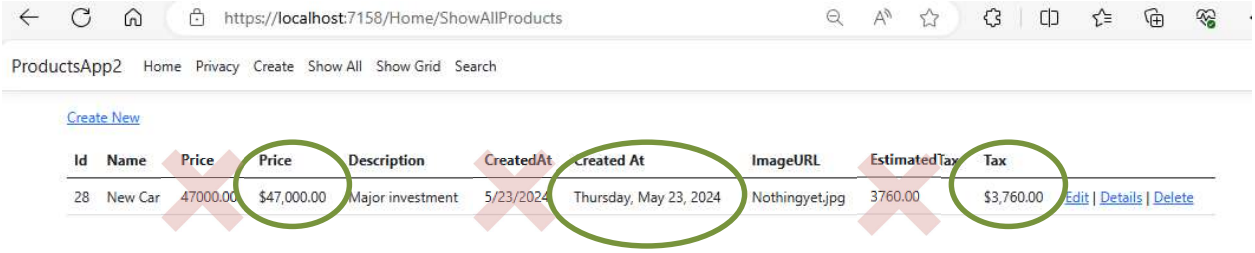


Figure 43 Removing the properties that are not meant to be shown on a report..

15. Run the program again and show all products. You should see the formatted fields remain visible, while the unformatted fields are omitted.

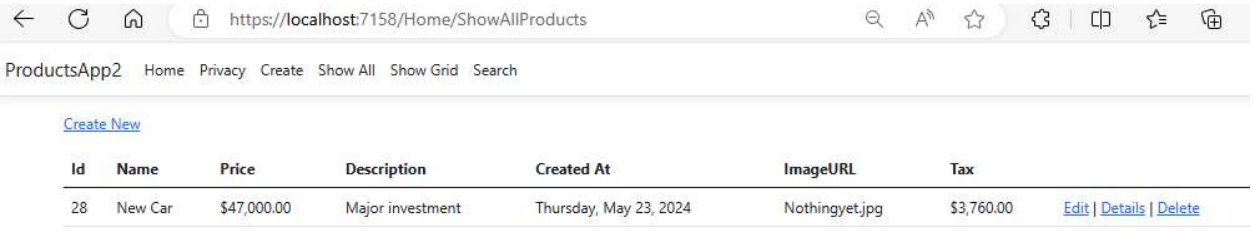


Figure 44 Updated table results.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Auto-Update Tax:

In the "Create" form, in order to dynamically change the values of a form without performing a full-page refresh, you can use JavaScript to change items.

1. In the bottom section of the ShowCreateProductForm.cshtml file there is a section for Javascript. Add the following lines.

```
@section Scripts {
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script>
        $(document).ready(function () {

            function updateComputedFields() {
                var price = parseFloat($("#Price").val());
                if (!isNaN(price)) {

                    var taxRate = parseFloat('@ViewBag.TaxRate');

                    var tax = price * taxRate;
                    var formattedTax = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' }).format(tax);
                    $("#FormattedEstimatedTax").val(formattedTax);
                }
            }
            // Update the computed fields when the user changes the price input text value
            $("#Price").on("input change", updateComputedFields);

            updateComputedFields();
        });
    </script>
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

Figure 45 jQuery code used to show the estimated tax value.

Role of jQuery in Modern Web .NET Development

While the web development landscape has evolved with the rise of modern JavaScript frameworks and libraries like React, Angular, and Vue.js, jQuery still has a role to play, especially in certain scenarios:

Legacy Applications:

Many existing .NET applications, including those using Razor views, were built with jQuery. Maintaining and extending these applications often involves continuing to use jQuery to avoid significant refactoring.

Simple Enhancements:
For small projects or when needing to add simple JavaScript functionality, jQuery can be quicker and easier to use than more complex frameworks.
Example: Adding basic form validation, animations, or simple AJAX calls.

Integration with Razor:
In Razor applications, especially those not requiring the complexity of a single-page application (SPA), jQuery can provide necessary interactivity and client-side behavior without the overhead of a full-fledged framework.
Example: Enhancing user interface elements, handling form submissions, and providing dynamic content updates.

2. In the HomeController, send the tax rate to the view as a ViewBag property.

```
public IActionResult ShowCreateProductForm()
{
    // set tax rate to static number now.  Later we will get this from appsettings.json
    ViewBag.TaxRate = 0.08m;

    var productViewModel = new ProductViewModel();
    productViewModel.CreatedAt = DateTime.Now;
    return View(productViewModel);
}
```

Figure 46 Providing the hard-coded 0.08 value to the form.

3. Run the program again and test the JavaScript code. The readonly fields should be updated based on the code in the <script> section.

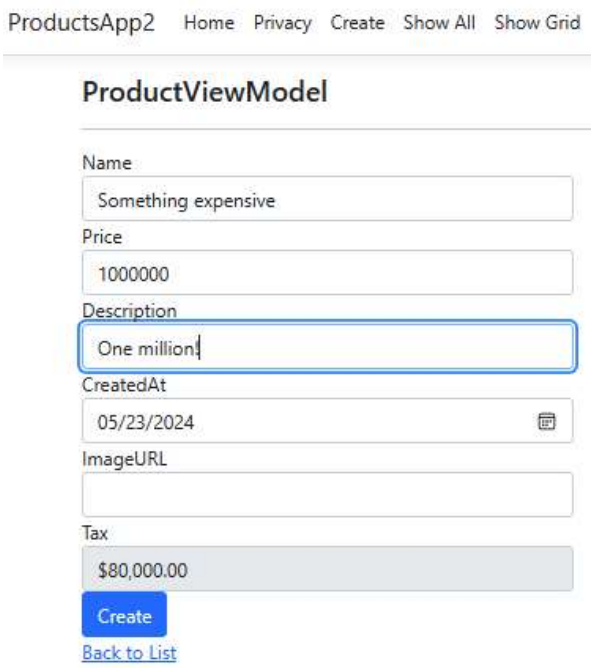


Figure 47 Taxes have been computed and shown in the read only text box.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

File Uploads:

In this section, we'll supercharge the product management application by adding the ability to upload and display product images. This will significantly enhance the visual appeal and information provided to users.

What We'll Do:
Image Storage: Create a dedicated folder within your project (e.g., "wwwroot/images") to store uploaded product images.
Controller Enhancements:
Utilize IWebHostEnvironment to access the image storage folder.
Implement methods to:
Fetch and list available image filenames.
Handle file uploads from forms.
Save uploaded files to the designated folder.
View Modifications:
Replace the ImageURL text input with a dropdown to select existing images.
Add a file input element to allow users to upload images directly.
Display thumbnails of product images in the product listing.

Checklist of Tasks:

Create Images Folder: Create a folder named "images" inside the "wwwroot" directory of your project.
Add Images: Place sample product images (JPEG or PNG) and a placeholder image (for missing images) in the "images" folder.
Inject IWebHostEnvironment: In your HomeController, inject IWebHostEnvironment to gain access to file paths and directories.
Implement Image Listing: Create a method in HomeController to fetch image filenames from the "images" folder and store them in ViewBag.
Modify Create View:
Replace the ImageURL input with a dropdown (<select>) populated with image filenames from ViewBag.
Add a file input element (<input type="file">) for image uploads.
Handle File Uploads: In the CreateProduct action, implement logic to:
Check if an image was uploaded.
Save the uploaded file with a unique name in the "images" folder.
Update the ProductViewModel.ImageURL with the saved file's path.
Modify ShowAllProducts View: Add an <img> tag to display a thumbnail of each product's image, using the ImageURL property.

Additional Considerations:
Image Validation: Implement validation to ensure uploaded files are valid images (correct file types, size limits).
Error Handling: Gracefully handle image upload errors and missing images.
Security: Sanitize file names to prevent potential security vulnerabilities.
Image Optimization: Consider optimizing uploaded images for faster loading.
Image Manipulation: You might add features to crop, resize, or watermark images.

By following these steps and incorporating these additional considerations, you'll create a more visually engaging and user-friendly product management application.

1. Create a folder named images in the wwwroot folder

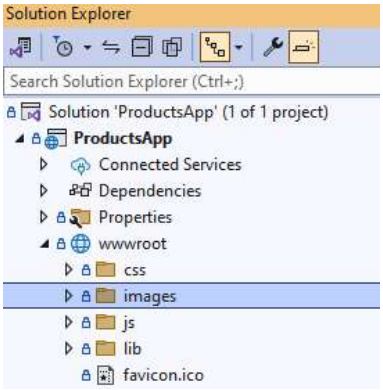


Figure 48 Created a folder for images

1. Put some jpeg or png files into the images folder. One of them is a picture of a product, the other is a placeholder for image loading errors.

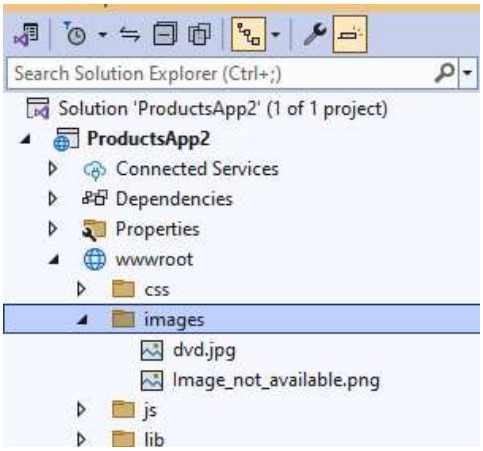


Figure 49 Two images have been added to the images folder.

The file "Image_not_available.png" is meant to be used when an image fails to load.

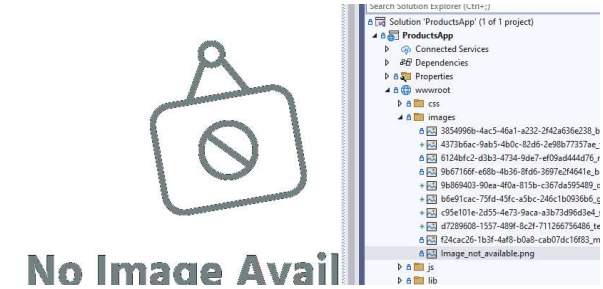


Figure 50 Image selected to stand as a placeholder for errors.

2. Make some changes to the HomeController to enable file uploads.
    a. Add an instance of IWebHostEnvironement to enable the application to read file directories.
    b. Create a method to fetch the file names in the images folder.
    c. Attach the list of file names to the Viewbag in order that the View can display a list of files.

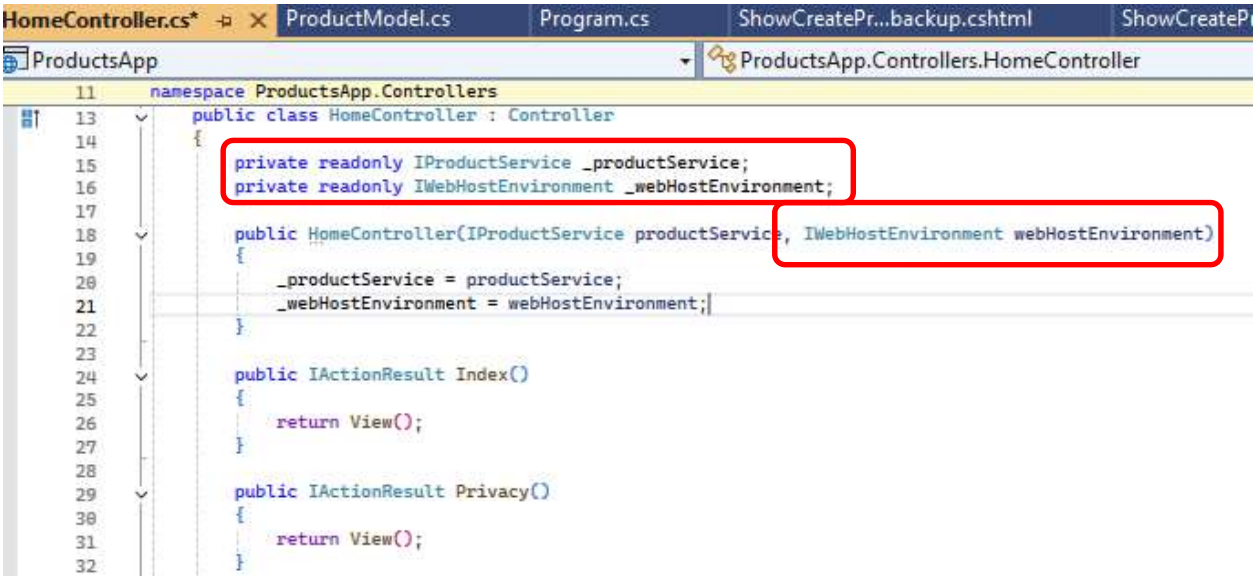


Figure 51 WebHost environment is a class that allows us to access the images folder in wwwroot.

```
public IActionResult ShowCreateProductForm()
{
    // send a list of strings to the view.  This list contains the names of all images in the images folder
    ViewBag.Images =  GetImageNames();

    // set to static number now.  Later we will get this from appsettings.json
    ViewBag.TaxRate = 0.08m;

    var productViewModel = new ProductViewModel();
    productViewModel.CreatedAt = DateTime.Now;
    return View(productViewModel);
}

// helper function to get all image names in the images folder
private List<string> GetImageNames()
{
    string imagesFolderPath = Path.Combine(_webHostEnvironment.WebRootPath, "images");

    if (!Directory.Exists(imagesFolderPath))
    {
        Directory.CreateDirectory(imagesFolderPath);
    }
    return Directory.EnumerateFiles(imagesFolderPath).Select(fileName => Path.GetFileName(fileName)).ToList();
}
```

Figure 52 Get the contents of the images folder and show it in a drop down menu for the user. Passed in by ViewBag.

3. Update the Create view. Replace the text entry for ImageURL with a Select (drop down) control.

```
23              <!-- removed formatted price -->
24
25              <div class="form-group">
26                  <label asp-for="Description" class="control-label"></label>
27                  <input asp-for="Description" class="form-control" />
28                  <span asp-validation-for="Description" class="text-danger"></span>
29              </div>
30              <div class="form-group">
31                  <label asp-for="CreatedAt" class="control-label"></label>
32                  <input asp-for="CreatedAt" class="form-control" />
33                  <span asp-validation-for="CreatedAt" class="text-danger"></span>
34              </div>
35
36              <!-- removed formatted datetime -->
37
38              <!-- replace ImageURL with a drop down select control -->
39
40              <div class="form-group">
41                  <label asp-for="ImageURL" class="control-label">Select Existing Image</label>
42                  <select asp-for="ImageURL" class="form-control" id="ImageURL">
43                      <option value="">-- Select an image --</option>
44                      @foreach (var image in ViewBag.Images as List<string>)
45                      {
46                          <option value="@image">@image</option>
47                      }
48                  </select>
49                  <span asp-validation-for="ImageURL" class="text-danger"></span>
50              </div>
51
52
53              <!-- removed estimated tax -->
54              <div class="form-group">
55                  <label asp-for="FormattedEstimatedTax" class="control-label"></label>
56                  <input asp-for="FormattedEstimatedTax" class="form-control" readonly />
57                  <span asp-validation-for="FormattedEstimatedTax" class="text-danger"></span>
58              </div>
59              <div class="form-group">
60                  <input type="submit" value="Create" class="btn btn-primary" />
61              </div>
62          </form>
63      </div>
64  </div>
```

Figure 53 The "Create" view now has a Select, or drop down menu, filled with the file names from the images folder.

4. Run the program. You should be able to select an existing image for the new product.

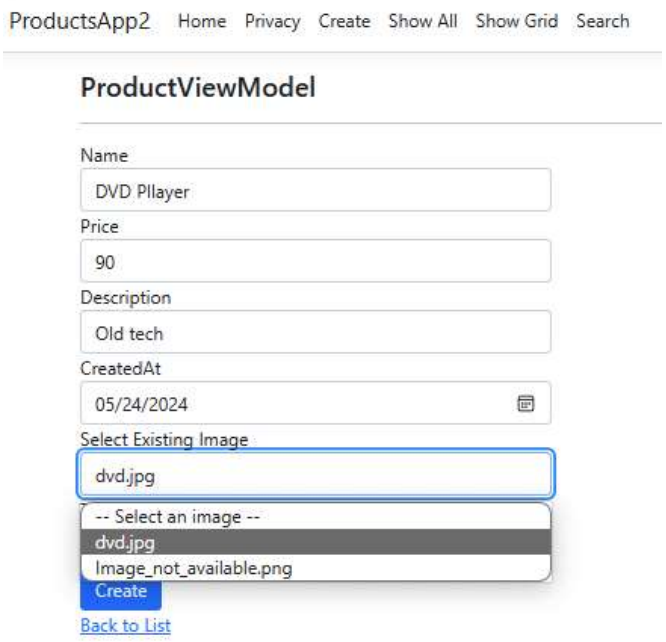


Figure 54 The create form now shows file names.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Next, we will change the form to support image uploads.

1. Update the view to accommodate the file upload and the drop down menu.

```
<h4>ProductViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="CreateProduct" enctype="multipart/form-data">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>

            <!-- removed id -->

            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>

            <!-- removed formatted price -->

            <div class="form-group">
                <label asp-for="Description" class="control-label"></label>
                <input asp-for="Description" class="form-control" />
                <span asp-validation-for="Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="CreatedAt" class="control-label"></label>
                <input asp-for="CreatedAt" class="form-control" />
                <span asp-validation-for="CreatedAt" class="text-danger"></span>
            </div>

            <!-- removed formatted datetime -->

            <!-- replace ImageURL with a drop down select control -->

            <div class="form-group">
                <label asp-for="ImageFile" class="control-label"></label>
                <input asp-for="ImageFile" type="file" class="form-control-file" id="ImageFile" />
                <span asp-validation-for="ImageFile" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="ImageURL" class="control-label">Select Existing Image</label>
                <select asp-for="ImageURL" class="form-control" id="ImageURL">
                    <option value="">-- Select an image --</option>
                    @foreach (var image in ViewBag.Images as List<string>)
                    {
                        <option value="@image">@image</option>
                    }
                </select>
                <span asp-validation-for="ImageURL" class="text-danger"></span>
            </div>
```

Figure 55 Two changes to the view allow the user to pick a file from the local computer and upload it.

2. Update the CreateProduct method in HomeController to perform the image save operation.

```
[HttpPost]
public async Task<IActionResult> CreateProduct(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.AddProduct(productViewModel);
        return RedirectToAction(nameof(Index));

    } else
    // redirect to the create product form if the model state is not valid
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = 0.08m;
        return View("ShowCreateProductForm", productViewModel);
    }

}

// helper function to upload the image file
private async Task<string> PerformFileUpload(ProductViewModel productViewModel)
{
    string uniqueFileName = "";
        // check if the user has uploaded an image. Perform the steps to save the image to the server
        if (productViewModel.ImageFile != null && productViewModel.ImageFile.Length > 0)
        {
            string uploadsFolder = Path.Combine(_webHostEnvironment.WebRootPath, "images");
             uniqueFileName = Guid.NewGuid().ToString() + "_" + productViewModel.ImageFile.FileName;
            string filePath = Path.Combine(uploadsFolder, uniqueFileName);
            using (var fileStream = new FileStream(filePath, FileMode.Create))
            {
                await productViewModel.ImageFile.CopyToAsync(fileStream);
            }
        }
    return uniqueFileName;
}
```

Figure 56 A helper function performs the upload process. The saved file name is assigned to the product's ImageURL property.

```
@section Scripts {
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script>
        $(document).ready(function () {

            function updateComputedFields() {
                var price = parseFloat($("#Price").val());
                if (!isNaN(price)) {

                    var taxRate = parseFloat('@ViewBag.TaxRate');

                    var tax = price * taxRate;
                    var formattedTax = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' }).format(tax);
                    $("#FormattedEstimatedTax").val(formattedTax);
                }
            }

            // only allow one of the image input fields to be filled in
            function toggleImageInput() {
                var imageFile = $("#ImageFile");
                var imageURL = $("#ImageURL");

                if (imageFile.val()) {
                    imageURL.prop('disabled', true);
                    imageURL.val("");
                } else {
                    imageURL.prop('disabled', false);
                }

                if (imageURL.val()) {
                    imageFile.prop('disabled', true);
                    imageFile.val("");
                } else {
                    imageFile.prop('disabled', false);
                }
            }

            // Update the computed fields when the user changes the price input text value
            $("#Price").on("input change", updateComputedFields);

            // Disable the other image input field when one is filled in
            $("#ImageFile, #ImageURL").on("input change", toggleImageInput);

            updateComputedFields();
            toggleImageInput();
        });
    </script>
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

Figure 57 jQuery updates to ensure only one type of file choice is made - upload or select an existing file.

3. Run the program. Add an image.

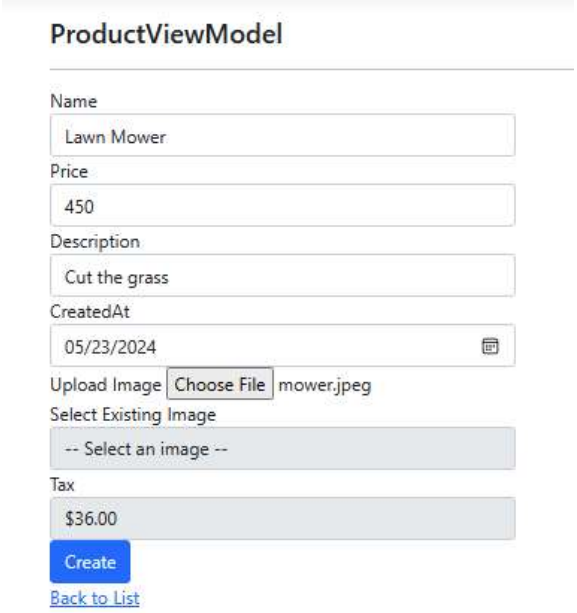


Figure 58 "Choose File" is now a button on the create form.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Let's externalize this jQuery code so we can reuse it on the Edit form which will be very similar to the Create form.

4. Create a new file in the wwwroot > js folder. Name it productForm.js

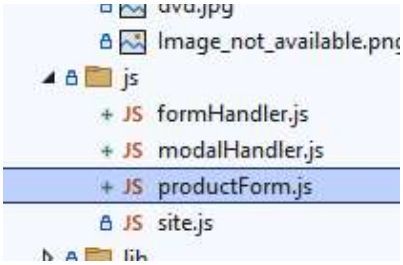


Figure 59 productForm.js is in the js folder. Two other files are here which we will create in future instructions.

5. Cut and paste the code to the new location. Make modifications shown for handling the taxRate.

```javascript
// js/productForm.js

function updateComputedFields(taxRate) {
    var price = parseFloat($("#Price").val());
    if (!isNaN(price)) {
        var tax = price * taxRate;
        var formattedTax = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' }).format(tax);
        $("#FormattedEstimatedTax").val(formattedTax);
    }
}

// only allow one of the image input fields to be filled in
function toggleImageInput() {
    var imageFile = $("#ImageFile");
    var imageURL = $("#ImageURL");

    if (imageFile.val()) {
        imageURL.prop('disabled', true);
        imageURL.val("");
    } else {
        imageURL.prop('disabled', false);
    }

    if (imageURL.val()) {
        imageFile.prop('disabled', true);
        imageFile.val("");
    } else {
        imageFile.prop('disabled', false);
    }
}

// Bind events for dynamically loaded content
function bindDynamicEvents(taxRate) {
    $("#Price").on("input change", function () {
        updateComputedFields(taxRate);
    });
    $("#ImageFile, #ImageURL").on("input change", toggleImageInput);
}

// Initialize form fields and events
function initializeForm(taxRate) {
    updateComputedFields(taxRate);
    toggleImageInput();
    bindDynamicEvents(taxRate);
}

```

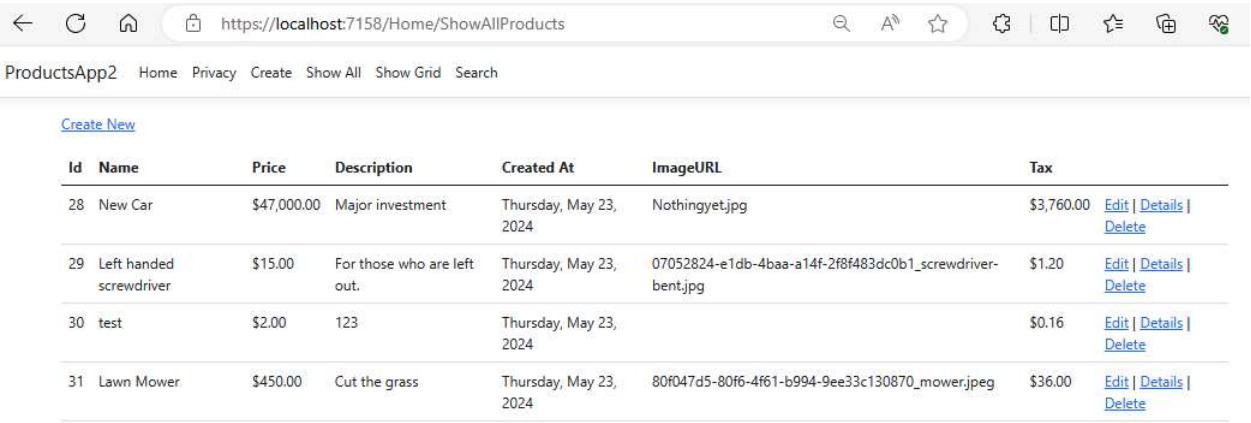
Figure 90 Method to calculate Tax Rate.

6. In the ShowCreateProductForm.cshtml file, modify the script section to reference the external file. Notice that we are passing the TaxRate value from the ViewBag.

```
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    <script src="~/js/productForm.js"></script>
    <script>
        $(document).ready(function () {
            var taxRate = '@ViewBag.TaxRate';
            initializeForm(taxRate);
        });
    </script>
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

Figure 91Utilize taxRate from the ViewBag.

7. The products should have a unique file name for every image.



Id	Name	Price	Description	Created At	ImageURL	Tax	
28	New Car	$47,000.00	Major investment	Thursday, May 23, 2024	Nothingyet.jpg	$3,760.00	Edit | Details | Delete
29	Left handed screwdriver	$15.00	For those who are left out.	Thursday, May 23, 2024	07052824-e1db-4baa-a14f-2f8f483dc0b1_screwdriver-bent.jpg	$1.20	Edit | Details | Delete
30	test	$2.00	123	Thursday, May 23, 2024		$0.16	Edit | Details | Delete
31	Lawn Mower	$450.00	Cut the grass	Thursday, May 23, 2024	80f047d5-80f6-4f61-b994-9ee33c130870_mower.jpeg	$36.00	Edit | Details | Delete

Figure 92 Showing the product details reveals that some products have ImageURL properties set.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

8. Make some visual improvements to the ShowAll view.
    a. Show a thumbnail image for each product.

```
            <th>
                @Html.DisplayNameFor(model => model.ImageURL)
            </th>
            <th>
                Thumbnails
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FormattedEstimatedTax)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Id)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>

            <td>
                @Html.DisplayFor(modelItem => item.FormattedPrice)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Description)
            </td>

            <td>
                @Html.DisplayFor(modelItem => item.FormattedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ImageURL)
            </td>
            <td>
                @if (!string.IsNullOrEmpty(item.ImageURL))
                {
                    <img src="/images/@item.ImageURL" alt="Product Image"  width = "140px" onerror="this.onerror=null;this.src='/images/Image_not_available.png';" />
                }
                else
                {
                    <span>No image available</span>
                }
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EstimatedTax)
```

Figure 93 Adding an img element to the table.

Grid Alternative to UI Tables:

Let's create an alternative way to view all the products. Instead of a table, we will use Bootstrap to show a grid of cards.

The Role of Bootstrap in Web Development

Bootstrap is a popular open-source front-end framework used for developing responsive and mobile-first websites. By default, it is included with Razor pages.

Why Bootstrap Cards Instead of Tables?

Modern Aesthetic: Cards are a sleek, modern UI element that aligns well with current design trends, making your application look more polished and engaging.
Flexibility: Cards can easily accommodate images, text, buttons, and other elements, allowing you to showcase product details in a more organized and informative way.
Responsiveness: Bootstrap's grid system and card components are inherently responsive. Your grid of cards will automatically adapt to different screen sizes, ensuring optimal viewing on desktops, tablets, and mobile devices.

Key Bootstrap Concepts:

Grid System: The .container, .row, and .col-md-4 classes create a responsive grid layout that adapts to different screen sizes.
Cards: The .card, .card-img-top, .card-body, .card-title, and .card-text classes style the product information within each card.

Customization:
Layout: Experiment with different column sizes (e.g., .col-sm-6, .col-lg-3) to control how many cards are displayed per row on various devices.
Styling: Use Bootstrap's utility classes (e.g., text-center, mt-2) to adjust spacing, alignment, and colors.
Additional Content: Add more elements to the card (e.g., buttons for "Add to Cart" or "Edit") to enhance the user experience.

1. In the HomeController, create a new method ShowAllProductsGrid

```
87          }
88
89          public IActionResult ShowAllProducts()
90          {
91              List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
92              return View(products);
93          }
94
95          public IActionResult ShowAllProductsGrid()
96          {
97              List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
98              return View(products);
99          }
100
101
102         [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
```

Figure 94 Adding a new method to show all products.

2. Take advantage of the asynchronous capabilities of C#. Make the method non-blocking for the UI.

```
//public IActionResult ShowAllProductsGrid()
//{

//    ViewBag.Images = GetImageNames();
//    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
//    List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
//    return View(products);
//}

// async version of grid
public async Task<IActionResult> ShowAllProductsGridAsync()
{
    ViewBag.Images = GetImageNames();
    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
    List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
    return View(products);
}
```

Figure 95 Alternative async version of the method.

3. Right click in the method and choose Add View.

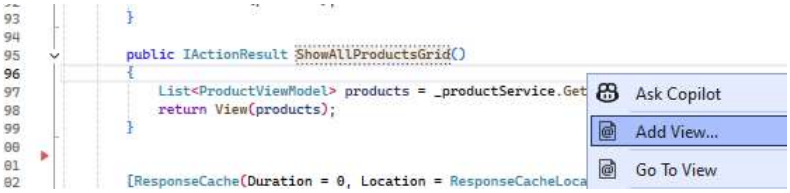


Figure 96 Generating a view to show all product.

4. Choose Razor

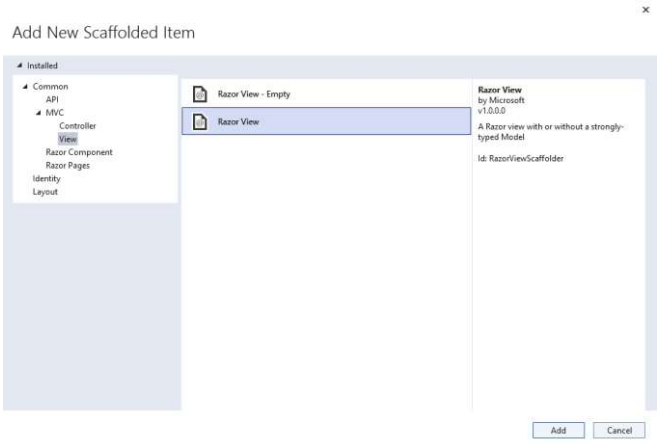


Figure 97 Razor chosen.

5. Choose List and ProductViewModel.

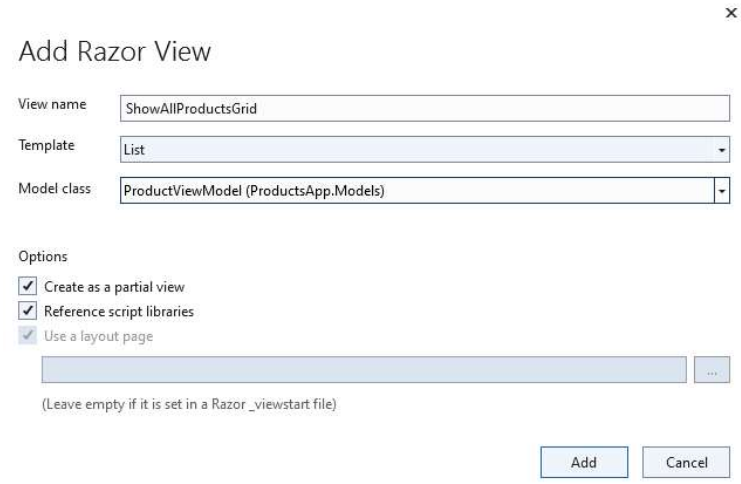
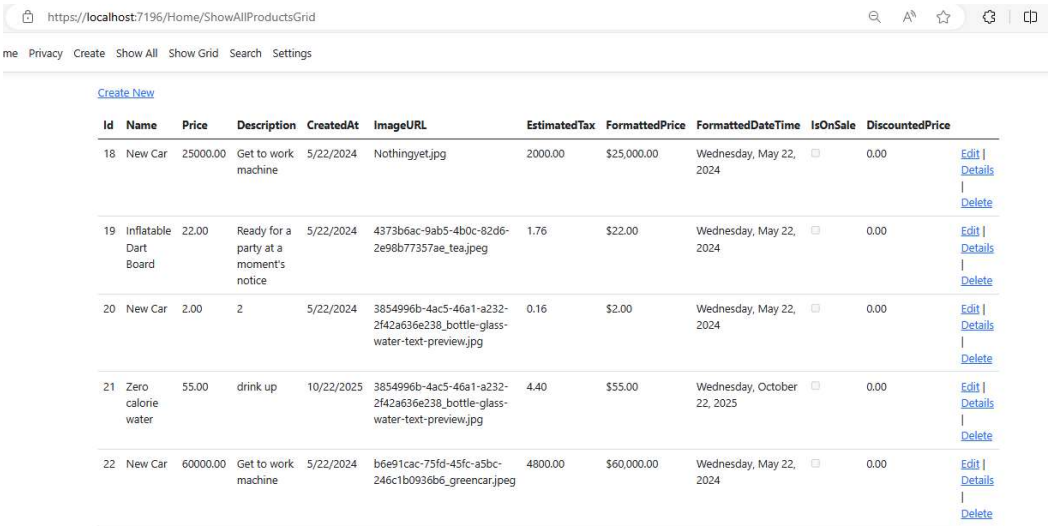


Figure 98 List of ViewModel objects chosen.

6. Run the program. It should look like another table view.



Id	Name	Price	Description	CreatedAt	ImageURL	EstimatedTax	FormattedPrice	FormattedDateTime	IsOnSale	DiscountedPrice	
18	New Car	25000.00	Get to work machine	5/22/2024	Nothingyet.jpg	2000.00	$25,000.00	Wednesday, May 22, 2024		0.00	Edit | Details | Delete
19	Inflatable Dart Board	22.00	Ready for a party at a moment's notice	5/22/2024	4373b6ac-9ab5-4b0c-82d6-2e98b77357ae_tea.jpeg	1.76	$22.00	Wednesday, May 22, 2024		0.00	Edit | Details | Delete
20	New Car	2.00	2	5/22/2024	3854996b-4ac5-46a1-a232-2f42a636e238_bottle-glass-water-text-preview.jpg	0.16	$2.00	Wednesday, May 22, 2024		0.00	Edit | Details | Delete
21	Zero calorie water	55.00	drink up	10/22/2025	3854996b-4ac5-46a1-a232-2f42a636e238_bottle-glass-water-text-preview.jpg	4.40	$55.00	Wednesday, October 22, 2025		0.00	Edit | Details | Delete
22	New Car	60000.00	Get to work machine	5/22/2024	b6e91cac-75fd-45fc-a5bc-246c1b0936b6_greencar.jpeg	4800.00	$60,000.00	Wednesday, May 22, 2024		0.00	Edit | Details | Delete

Figure 99 Standard table is the result of the generated Razor view.

7. Go to Bootstrap's website and look at the documentation for the Card element.

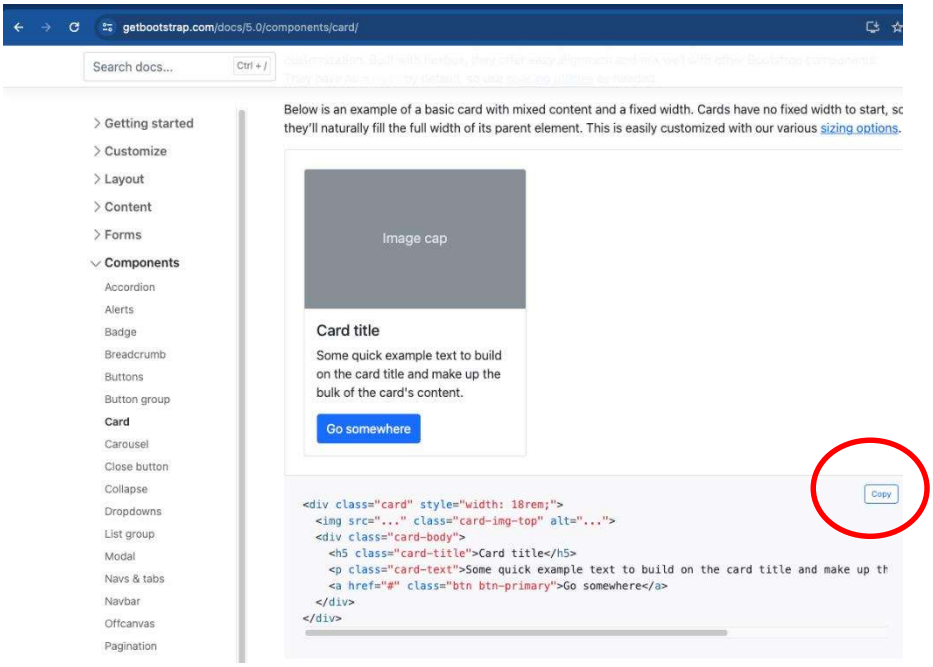


Figure 100 Bootstrap's Card documentation.

8. Copy the code for a Card.
9. Replace the <table> contents with a container, a for loop and paste the card data inside.

```
@model IEnumerable<ProductsApp2.Models.ProductViewModel>

<p>
    <a asp-action="Create">Create New</a>
</p>

<div class="container">
    <div class="row">
        @foreach (var item in Model)
        {
            <div class="col-md-4">
                <div class="card" style="width: 18rem;">
                    <img src="..." class="card-img-top" alt="...">
                    <div class="card-body">
                        <h5 class="card-title">Card title</h5>
                        <p class="card-text">Some quick example text to build on the card title and make up the bulk of the
                          card's content.</p>
                        <a href="#" class="btn btn-primary">Go somewhere</a>
                    </div>
                </div>
            </div>
        }
</div>
```

Figure 101 Most of the HTML is replaced by a container with a for loop and default Card code.

10. Run the program to see that the correct number of cards (four products are in the database) appear but are not formatted with the model data.

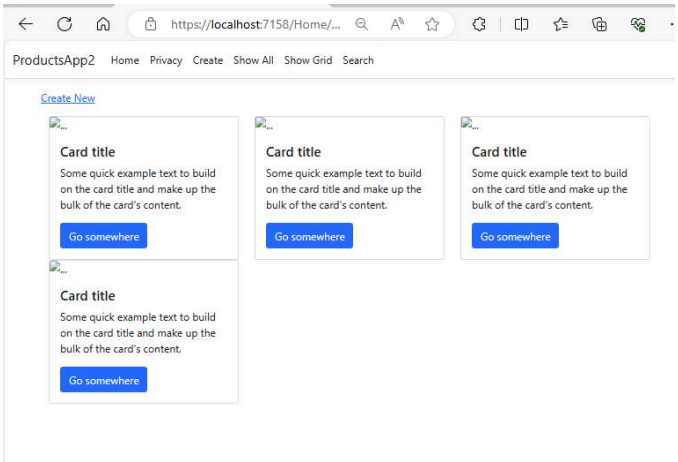


Figure 102 Generic Card elements are placed on the screen. It is not yet using the properties of the Product.

11. Make further updates to include the model data

```
@model IEnumerable<ProductsApp2.Models.ProductViewModel>

<p>
    <a asp-action="Create">Create New</a>
</p>

<div class="container">
    <div class="row">
        @foreach (var item in Model)
        {
            <div class="col-md-4">
                <div class="card" style="width: 18rem;">
                    @if (!string.IsNullOrEmpty(item.ImageURL))
                    {
                        <img src="/images/@item.ImageURL" alt="Product Image" onerror="this.onerror=null;this.src='/images/
                          Image_not_available.png';" />
                    }
                    else
                    {
                        <span>No image available</span>
                    }
                    <div class="card-body">

                        <h5 class="card-title">
                            @Html.DisplayFor(modelItem => item.Name)
                            @Html.DisplayNameFor(modelItem => item.Name)
                        </h5>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.Description)
                            @Html.DisplayFor(modelItem => item.Description)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedPrice)
                            @Html.DisplayFor(modelItem => item.FormattedPrice)
                            </p>
                        <p class="card-text">

                            @Html.DisplayNameFor(modelItem => item.FormattedEstimatedTax)
                            @Html.DisplayFor(modelItem => item.FormattedEstimatedTax)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedDateTime)
                            @Html.DisplayFor(modelItem => item.FormattedDateTime)
                            </p>
                        <a href="#" class="btn btn-primary">Details</a>
                        <a href="#" class="btn btn-warning">Edit</a>
                        <a href="#" class="btn btn-danger">Delete</a>

                    </div>
                </div>
            </div>
        }
</div>
```

Figure 103 Product properties are integrated into the card. The links at the bottom of the page are formatted to Boostrap btn class names.

9. Run the app again to see the actual product data.

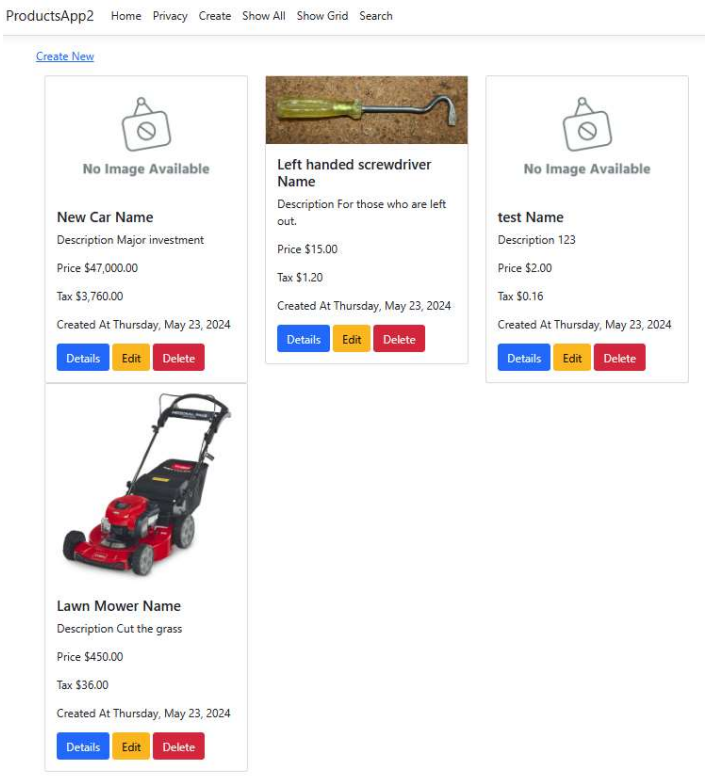


Figure 104 The look of the Grid view is now complete.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

10. Noticing the broken image for the first product, update the img tag to account for errors.

More CRUD:

So far we have been able to **create** and **read** products. Let's continue with the other parts of CRUD.

Delete-

1 Start with ProductDAO.

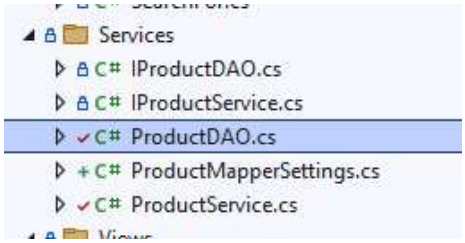


Figure 105 Updating the ProductDAO class.

2 Add a delete method. Add 'async' property.

```
public async Task DeleteProduct(ProductModel product)
{
    using (SqlConnection conn = new SqlConnection(_connectionString))
    {
        SqlCommand cmd = new SqlCommand("DELETE FROM Products WHERE Id = @Id", conn);
        cmd.Parameters.AddWithValue("@Id", product.Id);

        conn.Open();
        await cmd.ExecuteNonQueryAsync();
    }
}
```

Figure 106 New method DeleteProduct is added to ProductDAO

3 The getbyId method will also be used shortly. Add it to ProductDAO

```
public async Task<ProductModel> GetProductById(int id)
{
    ProductModel product = null;

    using (SqlConnection conn = new SqlConnection(_connectionString))
    {
        SqlCommand cmd = new SqlCommand("SELECT * FROM Products WHERE Id = @Id",
          conn);
        cmd.Parameters.AddWithValue("@Id", id);

        conn.Open();
        SqlDataReader reader = await cmd.ExecuteReaderAsync();

        if (reader.Read())
        {
            product = new ProductModel
            {
                Id = reader.GetInt32(0),
                Name = reader.IsDBNull(1) ? string.Empty : reader.GetString(1),
                Price = reader.IsDBNull(2) ? 0 : reader.GetDecimal(2),
                Description = reader.IsDBNull(3) ? string.Empty : reader.GetString
               (3),
                CreatedAt = reader.IsDBNull(4) ? DateTime.MinValue :
               reader.GetDateTime(4),
                ImageURL = reader.IsDBNull(5) ? string.Empty : reader.GetString(5)
            };
        }
    }

    return product;
}
```

Figure 107 GetProductById is also added to ProductDAO

4 In the ProductService class, also implement the delete method.

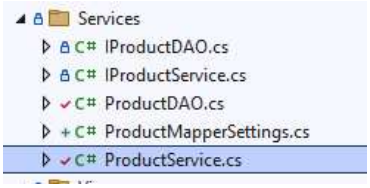


Figure 108 Updating the ProductService class.

```
public async Task DeleteProduct(string id)
{
    int productId = int.Parse(id);  // Assuming the id can be parsed to int
    var productModel = await _productDAO.GetProductById(productId);
    if (productModel != null)
    {
        await _productDAO.DeleteProduct(productModel);
    }
}
```

Figure 109 DeleteProduct is a new method in ProductService.

7. In the HomeController create a Delete method.

Figure 110 Updating HomeController

```
public async Task<IActionResult> DeleteProduct(string id)
{
    await _productService.DeleteProduct(id);
    return RedirectToAction("ShowAllProducts");
}
```

Figure 111 DeleteProduct is a method in HomeController.

8. In the ShowAllProducts view, update the links to use Bootstrap button class names.
9. Update the Delete button action to **DeleteProduct** with a parameter of **id = item.Id**
10. Use JavaScript to confirm the delete action.

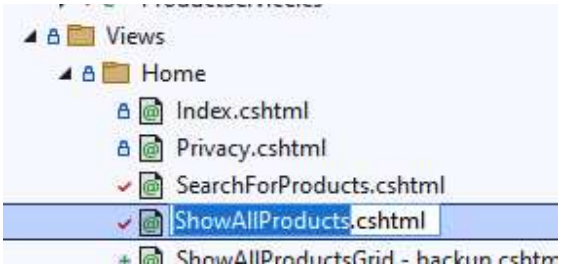


Figure 112 Upading ShowAllProducts

```
            @Html.DisplayFor(modelItem => item.FormattedPrice)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FormattedDateTime)
        </td>
        <td>
            <div class="btn-group" role="group">
                @Html.ActionLink("Edit", "ShowUpdateProductForm", new { id = item.Id }, new { @class = "btn btn-warning" })
                @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-info" })
                @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger", onclick = "return confirm('Are you
                    sure you want to delete this product?');" })
            </div>
        </td>
    </tr>
}
    </tbody>
</table>
```

Figure 113 Editing the action of the ActionLink items. The names correspond to method names in the HomeController.

11. Run the program and attempt to delete a product.

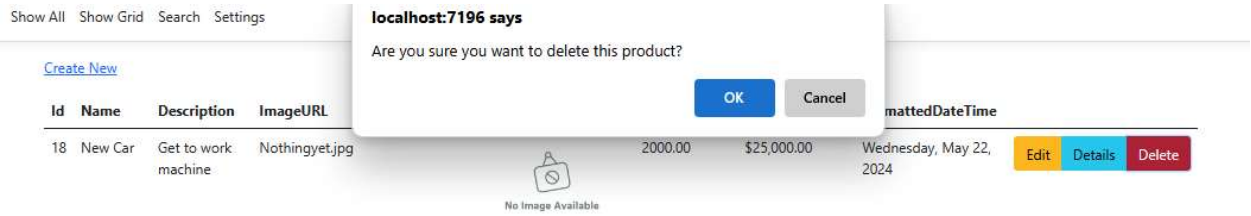


Figure 114 A small onClick JavaScript method confirms the user's desire to delete an item.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Edit Product:

Before You Begin to Edit

Editing a product is a similar process to creating a product, but with some key differences. Before you start to code, review this list of items to keep in mind as you program.

Similarities of Edit and Create

Form Handling:
Both features will involve a form for inputting product details (e.g., name, description, price, category).
Validation logic to ensure the data entered is correct will be similar.
Model Binding:
In both cases, the form data will be bound to a product model.
The server-side code will involve working with a productModel.

UI Layout:
The layout of the form (fields, labels, buttons) will be almost identical.
Both will require similar HTML/CSS structure, potentially leveraging Bootstrap for styling.
Server-Side Logic:
Both operations will involve server-side logic to handle the form submission.
Common methods will include validation, saving data to the database, and handling errors.

Key Differences between Edit and Create

User Experience:
Create Product: The user starts with a blank form and creates a new entry.
Edit Product: The user starts with an existing entry, making changes to it.

Data Population:
Create Product: The form is initialized with empty fields.
Edit Product: The form is pre-populated with existing product data, which requires retrieving the product from the database using its unique identifier (e.g., product ID).

Routing and URLs:
Create Product: Often accessed via a route like /Products/Create.
Edit Product: Accessed via a route like /Products/Edit/{id}, where {id} is the unique identifier of the product to be edited.

Database Operations:
Create Product: Involves inserting a new record into the database.
Edit Product: Involves updating an existing record, which requires fetching the current record, applying changes, and then saving the updated record.

1. In the ProductDAO class we will need to create a method UpdateProduct.

```
public async Task UpdateProduct(ProductModel product)
{
    using (SqlConnection conn = new SqlConnection(_connectionString))
    {
        SqlCommand cmd = new SqlCommand("UPDATE Products SET Name = @Name, Price = @Price, Description = @Description, CreatedAt = @CreatedAt,
          ImageURL = @ImageURL WHERE Id = @Id", conn);
        cmd.Parameters.AddWithValue("@Id", product.Id);
        cmd.Parameters.AddWithValue("@Name", product.Name);
        cmd.Parameters.AddWithValue("@Price", product.Price);
        cmd.Parameters.AddWithValue("@Description", product.Description);
        cmd.Parameters.AddWithValue("@CreatedAt", product.CreatedAt);
        cmd.Parameters.AddWithValue("@ImageURL", product.ImageURL);

        conn.Open();
        await cmd.ExecuteNonQueryAsync();
    }
}
```

Figure 115 ProductDAO method to update a row in the database.

2. In the ProductService implement UpdateProduct.

```
public async Task UpdateProduct(ProductViewModel productViewModel)
{
    var productDTO = _productMapper.ToDTO(productViewModel);
    var productModel = _productMapper.ToModel(productDTO);
    await _productDAO.UpdateProduct(productModel);
}
```

Figure 116 ProductService coordinates the translation between model types.

3. In the HomeController we will need two methods that are very similar to the Create process.

```
public async Task<IActionResult> ShowUpdateProductForm(int id)
{
    ViewBag.Images = GetImageNames();
    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
    ProductViewModel product = await _productService.GetProductById(id);
    return View(product);
}
```

```
[HttpPost]
public async Task<IActionResult> UpdateProduct(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.UpdateProduct(productViewModel);
        return RedirectToAction(nameof(Index));

    }
    else
    // redirect to the create product form if the model state is not valid
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = 0.08m;
        return View("ShowUpdateProductForm", productViewModel);
    }
}
```

Figure 117 Two methods to perform an update. First, show the form and provide the model. Second, send the form data back to the database and the images directory.

4. For the ShowUpdateProductForm, you can duplicate the form used to **create products**. Change the id, action property and some text. All other items should be the same.

Figure 118 The UpdateProduct form is based on the "create" form with some important changes.

2. Verify the jQuery code is referenced at the bottom of the page correctly.

```
            <div class="form-group">
                <input type="submit" value="Update" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    <script src="~/js/productForm.js"></script>
    <script>
        $(document).ready(function () {
            var taxRate = '@ViewBag.TaxRate';
            initializeForm(taxRate);
        });
    </script>
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

Figure 119 Section Scripts includes the productForm.js file.

3. Ensure that the "Edit" buttons on the All Products Form points to the correct method of the HomeController.

```
            @Html.DisplayFor(modelItem => item.FormattedPrice)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FormattedDateTime)
        </td>
        <td>
            <div class="btn-group" role="group">
                @Html.ActionLink("Edit", "ShowUpdateProductForm", new { id = item.Id }, new { @class = "btn btn-warning" })
                @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-info" })
                @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger", onclick = "return confirm('Are you
                  sure you want to delete this product?');" })
            </div>
        </td>
    </tr>
}
    </tbody>
</table>
```

Figure 120 The ShowAllProducts view has action links at the bottom. Ensure the actions are spelled exactly the same as method names in the HomeController.

4. Run the program and try to edit a product.
5. Copy the three links from this view and use them in the ShowAllProductsGrid view as well.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Searching:

Let's add the ability to look for products by either their name or the product description.

Search Tasks

Here is a brief summary of the tasks ahead.

ProductDAO: Create two database queries for searching products by name and description.

Services Folder: Create a new class, **SearchFor**, with properties matching the input controls on the form.

ProductService Class: Implement methods to handle the search queries together.

HomeController: Add methods to show the search form and perform the search.

ShowSearchForm.cshtml: Create a page to display the three properties of a **SearchFor** model.

SearchForProducts.cshtml: Create a view similar to **ShowAllProducts**, with additional information at the top of the page.

1. Start with ProductDAO and create two database queries for searching.

```
public async Task<IEnumerable<ProductModel>> SearchForProductsByName(string searchTerm)
{
    List<ProductModel> products = new List<ProductModel>();

    using (SqlConnection conn = new SqlConnection(_connectionString))
    {
        SqlCommand cmd = new SqlCommand("SELECT * FROM Products WHERE Name LIKE @SearchTerm", conn);
        cmd.Parameters.AddWithValue("@SearchTerm", "%" + searchTerm + "%");

        conn.Open();
        SqlDataReader reader = await cmd.ExecuteReaderAsync();

        while (reader.Read())
        {
            products.Add(new ProductModel
            {
                Id = reader.GetInt32(0),
                Name = reader.IsDBNull(1) ? string.Empty : reader.GetString(1),
                Price = reader.IsDBNull(2) ? 0 : reader.GetDecimal(2),
                Description = reader.IsDBNull(3) ? string.Empty : reader.GetString(3),
                CreatedAt = reader.IsDBNull(4) ? DateTime.MinValue : reader.GetDateTime(4),
                ImageURL = reader.IsDBNull(5) ? string.Empty : reader.GetString(5)
            });
        }
    }

    return products;
}

public async Task<IEnumerable<ProductModel>> SearchForProductsByDescription(string searchTerm)
{
    List<ProductModel> products = new List<ProductModel>();

    using (SqlConnection conn = new SqlConnection(_connectionString))
    {
        SqlCommand cmd = new SqlCommand("SELECT * FROM Products WHERE Description LIKE @SearchTerm", conn);
        cmd.Parameters.AddWithValue("@SearchTerm", "%" + searchTerm + "%");

        conn.Open();
        SqlDataReader reader = await cmd.ExecuteReaderAsync();

        while (reader.Read())
        {
            products.Add(new ProductModel
            {
                Id = reader.GetInt32(0),
                Name = reader.IsDBNull(1) ? string.Empty : reader.GetString(1),
                Price = reader.IsDBNull(2) ? 0 : reader.GetDecimal(2),
                Description = reader.IsDBNull(3) ? string.Empty : reader.GetString(3),
                CreatedAt = reader.IsDBNull(4) ? DateTime.MinValue : reader.GetDateTime(4),
                ImageURL = reader.IsDBNull(5) ? string.Empty : reader.GetString(5)
            });
        }
    }

    return products;
}
```

Figure 121 Two queries that perform searches. You could probably refactor these into a single method with an additional parameter.

2. In the Services folder create a new class, SearchFor.

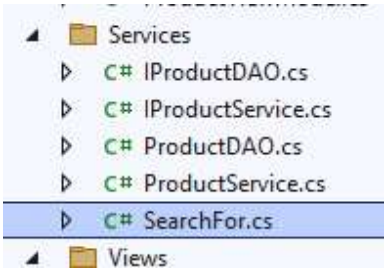


Figure 122 SearchFor class was created earlier.

3. It's properties will match the input controls on a form.

```
namespace ProductsApp.Models
{
    public class SearchFor
    {
        public string SearchTerm { get; set; }
        public bool InTitle { get; set; }
        public bool InDescription { get; set; }
    }
}
```

Figure 123 Three properties will define a search query.

4. In the ProductService class handle the search queries together.

```
public IActionResult ShowSearchForm()
{
    return View();
}

public async Task<IActionResult> SearchForProducts(SearchFor searchFor)
{
    List<ProductViewModel> searchResults = _productService.SearchForProducts(searchFor).Result.ToList();
    ViewBag.SearchTerm = searchFor.SearchTerm;
    ViewBag.InTitle = searchFor.InTitle;
    ViewBag.InDescription = searchFor.InDescription;

    return View(searchResults);
}
```

Figure 124 Data to be sent to a view that displays the search results.

5 In the HomeController add two methods to show a search form and to perform the search.

```
public IActionResult ShowSearchForm()
{
    return View();
}

public async Task<IActionResult> SearchForProducts(SearchFor searchFor)
{
    List<ProductViewModel> searchResults = _productService.SearchForProducts(searchFor).Result.ToList();
    var searchResultsAndSearchFor = new Tuple<List<ProductViewModel>, SearchFor>(searchResults, searchFor);
    return View(searchResultsAndSearchFor);
}
```

Figure 125 HomeController has two methods for searching: one to show the search form, another to display the search results.

6 The ShowSearchForm.cshtml page shows the three properties of a SearchFor model.

```
@model ProductsApp.Models.SearchFor

<h4>SearchFor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="SearchForProducts">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="SearchTerm" class="control-label"></label>
                <input asp-for="SearchTerm" class="form-control" />
                <span asp-validation-for="SearchTerm" class="text-danger"></span>
            </div>
            <div class="form-group form-check">
                <label class="form-check-label">
                    <input class="form-check-input" asp-for="InTitle" /> @Html.DisplayNameFor(model => model.InTitle)
                </label>
            </div>
            <div class="form-group form-check">
                <label class="form-check-label">
                    <input class="form-check-input" asp-for="InDescription" /> @Html.DisplayNameFor(model => model.InDescription)
                </label>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Figure 126 Search Form gets the user input.

7 The SearchForProducts.cshtml view is just like the ShowAllProducts view but with some extra information at the top of the page. You can duplicate the ShowAllProducts file and rename it.

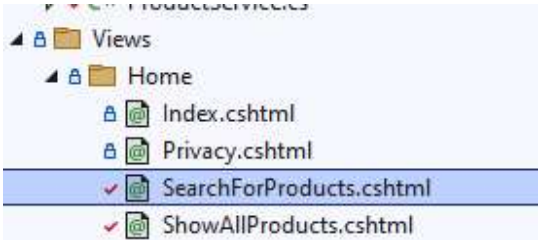
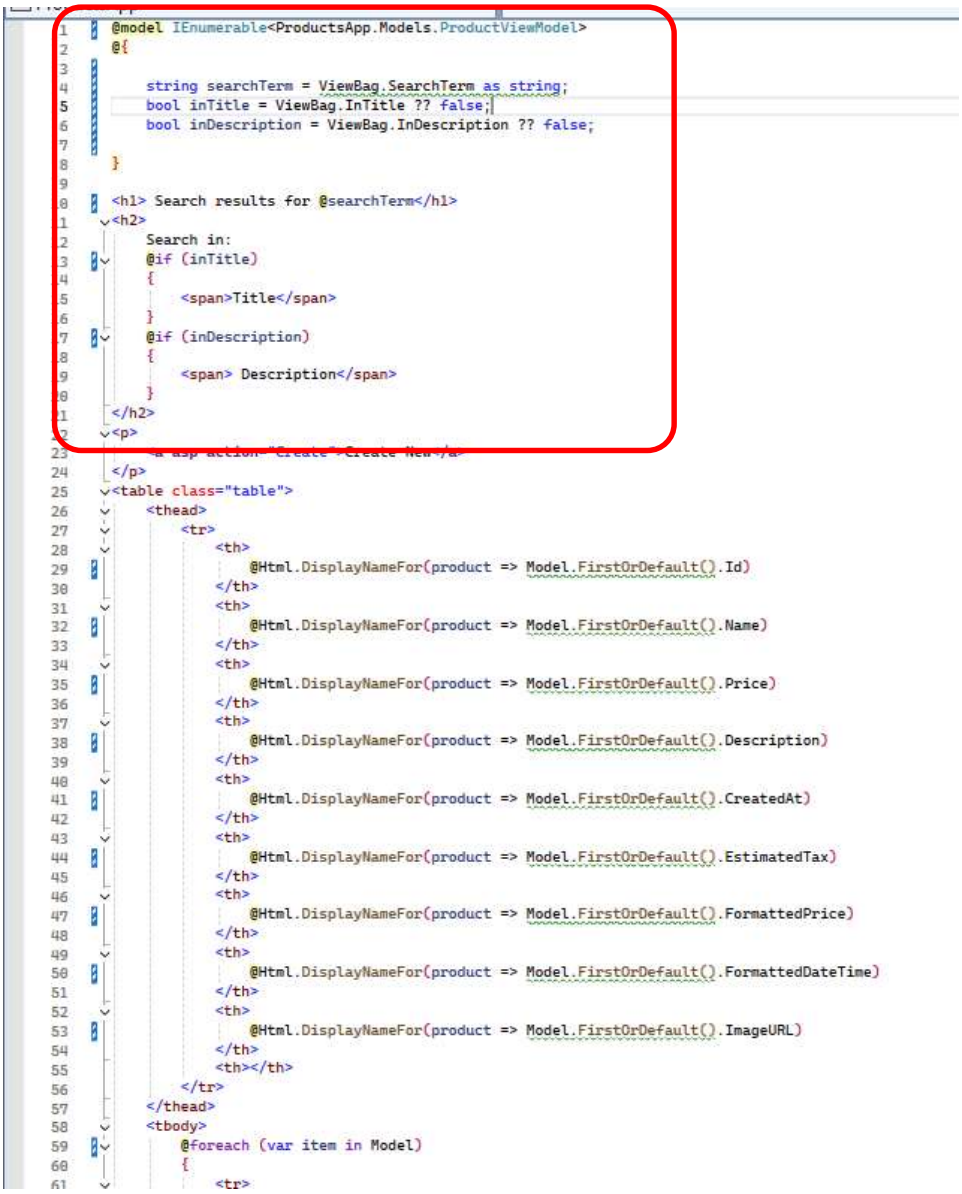


Figure 127 SearchForProducts is the view that will display a table of products that match the query.



```
@model IEnumerable<ProductsApp.Models.ProductViewModel>
@{

    string searchTerm = ViewBag.SearchTerm as string;
    bool inTitle = ViewBag.InTitle ?? false;
    bool inDescription = ViewBag.InDescription ?? false;

}

<h1> Search results for @searchTerm</h1>
<h2>
    Search in:
    @if (inTitle)
    {
        <span>Title</span>
    }
    @if (inDescription)
    {
        <span> Description</span>
    }
</h2>
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().Id)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().Name)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().Price)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().Description)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().CreatedAt)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().EstimatedTax)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().FormattedPrice)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().FormattedDateTime)
            </th>
            <th>
                @Html.DisplayNameFor(product => Model.FirstOrDefault().ImageURL)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
```

Figure 128 Top portion of the SearchForProducts page. This page was duplicated from the ShowAllProducts page. A developer could probably build the app using a single output form with some parameters.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Show One Item:

1. Update the buttons on the Show All Products page to ensure the action is correct for "Details"

```
67              @Html.DisplayFor(modelItem => item.FormattedPrice)
68          </td>
69          <td>
70              @Html.DisplayFor(modelItem => item.FormattedDateTime)
71          </td>
72          <td>
73              <div class="btn-group" role="group">
74                  @Html.ActionLink("Edit", "ShowUpdateProductForm", new { id = item.Id }, new { @class = "btn btn-warning" })
75                  @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-info" })
76                  @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger", onclick = "return confirm('Are you
                      sure you want to delete this product?');" })
77              </div>
78          </td>
79      </tr>
80  }
81  </tbody>
82  </table>
```

Figure 129 Inspecting the action links to ensure they correspond to method names in the HomeController.

2. Add a method to the HomeController.

```
public async Task<IActionResult> ShowProduct(int id)
{
    ProductViewModel product = await _productService.GetProductById(id);
    return View(product);
}
```

Figure 130 HomeController method to display a single product.

3. Create a view.

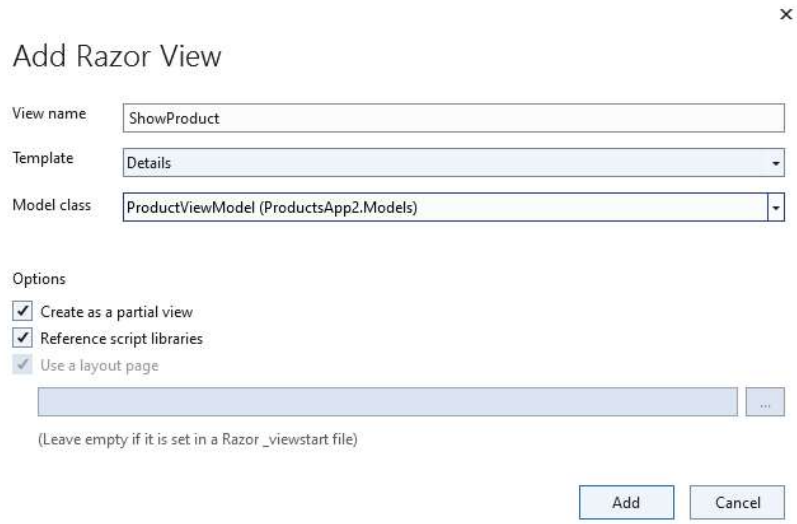


Figure 131 A "details" view is used to show the details of one item.

4. Run the program and test the function.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Why Centralized Settings?
Managing configuration settings can quickly become complex as your application grows. Hardcoding values directly into your code is inflexible and makes updating settings a nightmare. That's where appsettings.json steps in as a powerful ally.

About appsettings.json

Centralized Configuration: appsettings.json serves as a central repository for your application's settings. This includes database connection strings, API keys, logging configurations, feature flags, and more. By keeping them in one place, you avoid scattered values throughout your codebase.

Environment-Specific Settings: ASP.NET Core allows you to have different appsettings.json files for various environments (e.g., appsettings.Development.json, appsettings.Production.json). This enables you to easily tailor settings based on whether you're developing, testing, or deploying your application.

Dynamic Updates: You can change settings in appsettings.json without recompiling your application. This allows for dynamic updates to your application's behavior, even in a live environment (with caution!).

What is IConfiguration?
IConfiguration is an interface in the Microsoft.Extensions.Configuration namespace. It serves as the backbone of configuration management in ASP.NET Core. Essentially, it provides a way to read configuration data from various sources and access it in a consistent and structured manner throughout your application. IConfiguration is designed to work with a wide range of configuration sources, including:

appsettings.json
Environment variables.
Command-line arguments.
In-memory collections.
Azure Key Vault (for secure storage of sensitive values)

1. Open the appsettings.json file. It is found near the bottom of the project view.

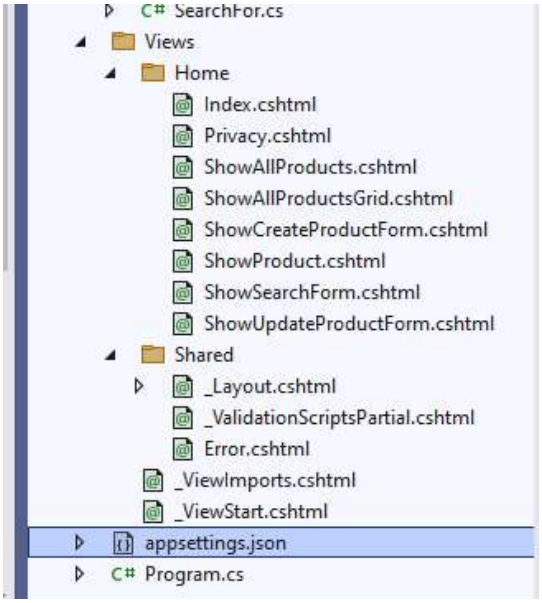


Figure 132 Location of the appsettings.json file.

2. Add two settings
    a. Copy the SQL connection string value from ProductDAO.
    b. Create the ProductMapper section with C, D and 0.8 values used in the formatting process.

Figure 133 Contents of appsettings.json are for logging, database connection and product mapping class.

3. Replace the hard-coded values in ProductDAO with references to the project configuration.

```
public class ProductDAO:IProductDAO

{
    private readonly string _connectionString;
    // = @"Data Source=192.168.0.52,1433;Initial Catalog=ProductsDatabase;User ID=SA;Password=sqlserverpassword123!@#;Connect
      Timeout=30;Encrypt=True;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

    // get the connection string from the appsettings.json file
    public ProductDAO(IConfiguration configuration)
    {
        _connectionString = configuration["SqlConnection:DefaultConnection"];
    }
```

Figure 134 ProductDAO no longer has the connection string embedded in it. It has been moved to the appsettings.json page.

4. In the Program.cs file, we used some hard-coded values that can be references in the app configuration.

```
// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddScoped<IProductDAO, ProductDAO>();
builder.Services.AddScoped<IProductService, ProductService>();

/*
builder.Services.AddScoped<IProductMapper, ProductMapper>(serviceProvider =>
{
    string currencyFormat = "C";
    string dateFormat = "D";
    decimal taxRate = 0.08m;

    return new ProductMapper(currencyFormat, dateFormat, taxRate);
});
*/
builder.Services.AddScoped<IProductMapper, ProductMapper>(

    serviceProvider => new ProductMapper(
        // get properties C D and 0.08m from appsettings.json
        builder.Configuration["ProductMapper:CurrencyFormat"],
        builder.Configuration["ProductMapper:DateFormat"],
        decimal.Parse(builder.Configuration["ProductMapper:TaxRate"])
        )
);


var app = builder.Build();
```

Figure 135 Dependency Injection options in Program.cs also have hard-coded settings that should be moved to appsettings.json

5. In the HomeController, add IConfiguration value and parameter name to the constructor.

```
namespace ProductsApp2.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;
        private readonly IProductService _productService; // second item added by dependency injection
        private readonly IWebHostEnvironment _webHostEnvironment; // third item added by dependency injection
        private readonly IConfiguration _configuration; // fourth item added by dependency injection

        public HomeController(ILogger<HomeController> logger, IProductService productService, IWebHostEnvironment webHostEnvironment,
          IConfiguration configuration )
        {
            _logger = logger;
            _productService = productService;
            _webHostEnvironment = webHostEnvironment;
            _configuration = configuration;
        }
        public IActionResult Index()
        {
```

Figure 136 The HomeController injects the IConfiguration class.

6. In the HomeController, there are four methods, that used the TaxRate value. Change all four to use the configuration settings value rather than the hardcoded 0.08 value.

```
public IActionResult ShowCreateProductForm()
{
    // send a list of strings to the view.  This list contains the names of all images in the images folder
    ViewBag.Images =  GetImageNames();

    // set to static number now.  Later we will get this from appsettings.json
    //  ViewBag.TaxRate = 0.08m;

    // get the tax rate from appsettings.json
    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);

    var productViewModel = new ProductViewModel();
    productViewModel.CreatedAt = DateTime.Now;
    return View(productViewModel);
}
```

Figure 137 TaxRate is retrieved from appsettings.json instead of being hard-coded in the controller.

```
[HttpPost]
public async Task<IActionResult> CreateProduct(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.AddProduct(productViewModel);
        return RedirectToAction(nameof(Index));

    } else
    // redirect to the create product form if the model state is not valid
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
        return View("ShowCreateProductForm", productViewModel);
    }

}
```

Figure 138 CreateProduct also references the tax value.

```
public IActionResult ShowUpdateProductForm(int id)
{
    ViewBag.Images = GetImageNames();
    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
    ProductViewModel product = _productService.GetProductById(id).Result;
    return View(product);
}

[HttpPost]
public async Task<IActionResult> UpdateProduct(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.UpdateProduct(productViewModel);
        return RedirectToAction(nameof(Index));

    }
    else
    // redirect to the create product form if the model state is not valid
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
        return View("ShowUpdateProductForm", productViewModel);
    }
}
```

Figure 139 Update a product requires the user of tax values.

5. Run the app to test the new settings properties. Resolve any errors.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Conclusions

What You Learned

In this lesson, you gained an understanding of building a full-stack CRUD application using the n-layer design approach with ASP.NET MVC. Let's summarize the key vocabulary, software design techniques, unique C# solutions, and the features of ASP.NET MVC that were covered.

Key Vocabulary

1 **CRUD Operations**: The four basic operations for managing data - Create, Read, Update, and Delete.
2 **N-layer Design**: A software architecture that separates the application into distinct layers: Data Access, Business Logic, and Presentation.
3 **DTO (Data Transfer Object)**: An object used to transfer data between different parts of the application.
4 **ViewModel**: An object that holds the data to be displayed in the view and handles user input.
5 **Forms**: HTML forms used to collect user input and submit it to the server.
6 **Image Upload**: The functionality that allows users to upload images to the server.

Software Design Techniques

1. **Separation of Concerns**: Dividing the application into layers to improve maintainability and scalability.
2. **Dependency Injection**: Injecting dependencies into classes to reduce tight coupling and improve testability.
3. **Asynchronous Programming**: Using async and await keywords to perform non-blocking operations, enhancing responsiveness and efficiency.
4. **Data Mapping**: Translating data between different layers using mappers to ensure consistency and separation.

Unique C# Solutions

1. **Async/Await**: Implementing asynchronous methods to improve application responsiveness and scalability.
2. **LINQ**: Using Language Integrated Query (LINQ) to perform queries on data collections in a readable and concise manner.
3. **Entity Framework**: Utilizing Entity Framework for data access and manipulation, providing a powerful ORM (Object-Relational Mapping) tool.
4. **Model Binding**: Automatically mapping form inputs to action method parameters, simplifying data handling in controllers.

Features of ASP.NET MVC

1. **Model-View-Controller (MVC) Pattern**: A design pattern that separates the application logic, user interface, and input control.
2. **Razor View Engine**: A markup syntax that allows embedding server-based code into web pages, providing a clean and efficient way to create dynamic content.
3. **Routing**: Defining URL patterns and mapping them to controller actions, enabling clean and user-friendly URLs.
4. **Scaffolding**: Automatically generating boilerplate code for CRUD operations, speeding up development.

Common Antipatterns and How to Avoid Them

1. **Monolithic Design**: Combining all logic into a single layer or class, leading to a tightly coupled and difficult-to-maintain codebase. Avoid this by adhering to the n-layer design.
2. **Hardcoding Values**: Embedding static values directly in the code, which can lead to inflexibility and errors. Use configuration files or constants instead.
3. **Poor Error Handling**: Ignoring exceptions or failing to implement proper error handling, resulting in an unstable application. Implement comprehensive error handling and logging.
4. **Spaghetti Code**: Writing code without a clear structure or organization, making it hard to understand and maintain. Follow design patterns and principles to keep the code organized.

Key Takeaways

1 **Modularity**: The n-layer design promotes modularity, making your code easier to manage and extend.
2 **Maintainability**: By separating concerns, each layer can be developed, tested, and maintained independently.
3 **Scalability**: Well-designed applications can scale more effectively to handle increased load and complexity.
4 **Reusability**: Components such as DTOs, ViewModels, and services can be reused across different parts of the application.
5 **Responsiveness**: Asynchronous programming enhances the responsiveness and performance of your application.

Suggestions for Further Development

Although we don't have time at the moment to extend this application, there are many more enhancements that a professional developer would consider next.

1 **Authentication and Authorization**: Implement user authentication and role-based authorization to secure your application.
2 **Data Source**: Remove MS SQL from the application and replace it with a no-SQL database such as MongoDB. The DTO should be designed to handle a change as drastic as this.
3 **Unit Testing**: Write unit tests for each layer to ensure the reliability and correctness of your code.
4 **Advanced CRUD Features**: Add pagination, filtering, and sorting to enhance the usability of your CRUD operations.
5 **RESTful APIs**: Expose your application's functionality through RESTful APIs to support integration with other systems and platforms.
6 **Client-Side Frameworks**: Integrate client-side frameworks like Angular, React, or Vue.js to create a more dynamic and responsive user interface.
7 **Performance Optimization**: Implement caching, query optimization, and other techniques to improve the performance of your application.

Check for Understanding

These questions are not graded but will help you prepare for upcoming assessments.

1. What is the primary purpose of n-layer design in software development?
   A. To increase the complexity of code
   B. To separate concerns and improve maintainability
   C. To reduce the number of files in a project
   D. To enhance code obfuscation

2. Which layer in the n-layer design is responsible for handling business logic and calculations?
   A. Data Access Layer
   B. Presentation Layer
   C. Business Logic Layer

D. Network Layer

3. What does CRUD stand for in the context of database operations?
   A. Create, Read, Update, Delete
   B. Connect, Retrieve, Upload, Download
   C. Configure, Run, Update, Debug
   D. Check, Repair, Undo, Delete

4. What is the role of a Data Transfer Object (DTO) in an application?
   A. To handle database connections
   B. To transfer data between different layers of the application
   C. To render data on the user interface
   D. To perform data validation

5. Which of the following is NOT a benefit of using Dependency Injection (DI)?
   A. Decoupling classes from their dependencies
   B. Enhancing testability
   C. Increasing tight coupling of code
   D. Improving flexibility

6. In asynchronous programming, what is the purpose of the await keyword?
   A. To block the main thread until the operation completes
   B. To run the operation synchronously
   C. To pause the execution of a method until the awaited task completes
   D. To execute the method in parallel without waiting

7. What is the main function of the Presentation Layer in an n-layer design?
   A. To manage database connections
   B. To implement business rules
   C. To interact with the user interface and handle user input
   D. To transfer data between other layers

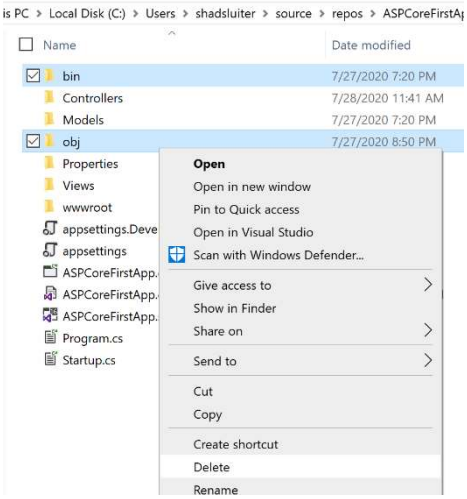
8. Which component in an n-layer architecture is typically responsible for validating user input?
   A. Data Access Layer
   B. Business Logic Layer
   C. Presentation Layer
   D. Network Layer

9. What is the advantage of using ViewModels in an MVC application?
   A. They store database connection strings
   B. They allow the direct manipulation of the database
   C. They help in presenting data specifically formatted for the view
   D. They are used to compile the application

10. Which of the following best describes the role of the Data Access Object (DAO) in an application?

A. It handles user authentication and authorization
B. It manages the flow of data to and from the database
C. It renders HTML pages for the user interface
D. It performs input validation and error handling

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Check for Understanding Answers

1. B
2. C
3. A
4. B
5. C
6. C
7. C
8. C
9. C
10. B