**Databases in business engineering context**

**Documentation**

At the Baden-Württemberg Heidenheim Cooperative State University

In the Faculty of Economics

In Business Information Systems

submitted by

Ellen Olbrich

Gernot Igers

Jenny Graf

Justin Peterson

Paula Garkisch

Solomiia Nestor

Franziska Klement

|  |  |
| --- | --- |
| Semester: | 2023/3+4 |
| Submission date: | 13.02.2025 |
| Lecturer:  GitHub-Link: | Bohdan Pashkovskyi  [Ello211/DB\_E-Commerce](https://github.com/Ello211/DB_E-Commerce) |

# Content

[Content 1](#_Toc1317727393)

[Introduction 2](#_Toc204770219)

[Entities 3](#_Toc735107204)

[Categories 4](#_Toc1474164343)

[Customers 4](#_Toc1120182931)

[Orders 5](#_Toc713322664)

[Payments 6](#_Toc1488569496)

[Products 7](#_Toc1436523255)

[Products\_Orders 7](#_Toc1379446952)

[Shipments 8](#_Toc282026973)

[Products\_Categories 9](#_Toc934913367)

[Relationships and Their Justifications 9](#_Toc319826560)

[One-to-One (1:1) Relationships 10](#_Toc2069392195)

[One-to-Many (1:n) Relationships 10](#_Toc2070161292)

[Many-to-Many (n:m) Relationships 11](#_Toc2011074784)

[Entity-Relationship Diagram (ERD) 11](#_Toc1795808778)

[CRUD Opperations 12](#_Toc1582558056)

[MongoDB Collection Design 13](#_Toc1901359510)

[Use Case 14](#_Toc1585941173)

[General Use Cases 14](#_Toc144364869)

[E-Commerce Use Case 14](#_Toc1108691256)

[Schema Design 14](#_Toc2140188638)

[Integration with Backend 14](#_Toc743447266)

[CRUD Operations with MongoDB 15](#_Toc1313973640)

[Redis Caching Implementation 17](#_Toc1066739419)

[Definition of the Redis settings class 18](#_Toc2109934731)

[Changes to the Program.cs file 18](#_Toc1617143861)

[Changes to the appsettings.json file 18](#_Toc1367239232)

[Creating the ProductDto class 18](#_Toc340153822)

[Meaning of the ProductDto class: 19](#_Toc1356349501)

[Properties of the ProductDto class: 19](#_Toc43012806)

[Changes to the ProductsController class 19](#_Toc1691015761)

[1. addition of dependencies 19](#_Toc834656137)

[2. adding fields and constructor 19](#_Toc586356728)

[3. methods for managing the cache 20](#_Toc1934144801)

[4. changes to the controller methods 21](#_Toc187040379)

[Another Redis Desktop Manager 22](#_Toc1512135227)

[Frontend Development / MVC 22](#_Toc665682455)

[What is the MVC? 23](#_Toc337153684)

[How MVC works in a Web Application 23](#_Toc1897538681)

[Benefits of using MVC 23](#_Toc1810173982)

[The individual MVC components 23](#_Toc673378009)

[Model 23](#_Toc1598816235)

[Controllers 27](#_Toc1826366604)

[Viewmodels 32](#_Toc797540378)

[Views 34](#_Toc2020523109)

[Structure of our Website 38](#_Toc51640605)

[Future Enhancements 45](#_Toc804199208)

[Conclusion 46](#_Toc1562696748)

# Introduction

The E-Commerce Web Application is a comprehensive solution developed to manage an online business platform. This project showcases the integration of modern database solutions and web technologies to create an efficient and user-friendly e-commerce system. The application provides full CRUD (Create, Read, Update, Delete) functionality for key business entities and supports complex relationships between these entities within a relational database.

At the heart of this application is an SQL database, which manages the core business entities such as Users, Products, Orders, Payments, Categories, and Shipments. These entities are interconnected through various relationships, including one-to-one (e.g., User ↔ Profile), one-to-many (e.g., User ↔ Orders), and many-to-many (e.g., Products ↔ Categories). This relational structure enables efficient data organization, retrieval, and manipulation.

In addition to the SQL database, MongoDB has been utilized to handle unstructured or semi-structured data, such as user activity logs or product reviews. MongoDB offers flexibility in managing dynamic data with a schema-less design, which is essential for handling the varied nature of e-commerce information.

To further enhance application performance, Redis has been integrated as a caching layer. Redis optimizes the performance by caching frequently accessed data such as product details or user profiles, thus reducing database load and improving response times. The caching layer includes well-defined cache expiration policies and cache invalidation logic to ensure consistency between the cache and the database.

This documentation provides a detailed overview of the architecture, database design, and key features of the E-Commerce Web Application. It outlines the implementation of the SQL and MongoDB databases, the use of Redis for performance optimization, and the overall structure of the web application. The document serves as a guide to understanding the technical decisions and design choices made throughout the project.

# Entities

In the E-Commerce Web Application, entities represent the core components of the platform, each playing a vital role in business operations. The Users entity represents both customers and administrators, storing information such as credentials, contact details, and roles for authentication and account management. The Products entity encompasses the items available for sale, including attributes like name, price, and category, and serves as the backbone of the product catalog. Orders track customer purchases, capturing order details, total price, and the items involved. The Payments entity stores financial transaction details, such as the payment method, status, and associated order. Categories group products to improve organization and product discovery for customers. Finally, Shipments manage the logistics of delivering products, tracking the shipment status, tracking number, and delivery date. These entities are interconnected through various relationships, ensuring the smooth operation and functionality of the e-commerce system.

For the Web application all entities are defined in the Models folder of DB\_Ecommerce.Models. The primary keys and the other attributes are configured in the Configurations folder of DB\_Ecommerce.Persistence.

## Categories

The Categories entity is designed to classify products into distinct groups, enabling organized product management and improved customer navigation. By categorizing products, the platform can offer a structured browsing experience and support features like filtering and searching. This entity is essential for maintaining a logical hierarchy of products.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 1: Categories Table

**CategoryID** (Primary Key): A unique, auto-incremented integer that identifies each category. This ensures that each category has a distinct identifier for easy reference. It ensures uniqueness and simplifies referencing in relationships.

**CategoryName**: A string (varchar) that stores the name of the category (e.g., "Electronics", "Clothing"). This attribute provides a human-readable label for each category. CategoryName is necessary for displaying categories to users and for administrative purposes.

## Customers

The **Customers** entity stores information about users who interact with the e-commerce platform. It is central to personal shopping experience, managing orders, and facilitating communication. This entity ensures that customer data is accurately tracked and accessible for operational and analytical purposes.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 2: Customers Table

**CustomerID** (Primary Key): A unique, auto-incremented integer that identifies each customer. This ensures each customer has a unique identifier. It ensures unique identification of customers.

**FirstName**: A string (varchar) that stores the customer's first name. This is used for personalization and communication.

**LastName**: A string (varchar) that stores the customer's last name. This complements the first name for identification.

**Address**: A string (varchar) that stores the customer's address. This is essential for shipping and billing purposes. The Address is critical for order fulfillment.

**Birthday**: An optional date field that optionally stores the customer's date of birth. This can be used for personalized marketing or age verification. Optional fields like Birthday provide additional insights into marketing and analytics.

**AccountCreated**: A date field that records when the customer account was created. This helps track customer loyalty and activity.

**Email**: A string (varchar) that stores the customer's email address. This is used for communication and account verification.

## Orders

The Orders entity represents transactions where customers purchase products. It tracks the details of each order, including the customer, date, and total price. This entity is crucial for managing sales, inventory, and customer satisfaction.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 3: Orders Table

**OrderID** (Primary Key): A unique, auto-incremented integer that identifies each order. This ensures each order is uniquely identifiable. It ensures unique identification of orders.

**OrderDate**: A date field that records when the order was placed. This is essential for tracking order timelines. OrderDate is critical for financial and operational reporting.

**TotalPrice**: A decimal value that stores the total price of the order. This is necessary for financial tracking and reporting. TotalPrice is critical for financial and operational reporting.

**CustomerID** (Foreign Key): A reference to the Customers table. This links the order to the customer who placed it. CustomerID establishes a relationship between orders and customers, enabling customer-specific order tracking.

## Payments

The **Payments** entity stores payment details for orders. It tracks the payment method, status, amount, and any outstanding balance. This entity ensures accurate financial tracking and supports payment processing workflows.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 4: Payments Table

**PaymentID** (Primary Key): A unique integer that identifies each payment. This ensures each payment is uniquely identifiable. It ensures unique identification of payments.

**Currency**: A string (varchar) that specifies the currency used for the payment (e.g., "USD", "EUR"). This is necessary for international transactions.

**PaymentMethod**: A string (varchar) that specifies the payment method (e.g., "Credit Card", "PayPal"). PaymentMethod is essential for payment processing.

**PaymentStatus**: A string (varchar) that indicates the status of the payment (e.g., "Paid", "Pending"). This is critical for order fulfillment.

The attributes Currency, PaymentMethod, and PaymentStatus are necessary for payment processing and reporting.

**PaymentAmount**: A decimal value that stores the total amount paid. This is necessary for financial tracking.

**OrderID** (Foreign Key): A reference to the OrderID in the Orders table. This links the payment to the corresponding order. OrderID establishes a relationship between payments and orders.

**OpenPayment**: A decimal value that stores any remaining amount to be paid (if applicable). This supports partial payments. OpenPayment field supports flexible payment scenarios.

## Products

The **Products** entity represents the items available for purchase on the e-commerce platform. It stores product details such as name and price. This entity is central to inventory management and product display.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 5: Products Table

**ProductID** (Primary Key): A unique, auto-incremented integer that identifies each product. This ensures each product is uniquely identifiable. ProductID ensures unique identification of products.

**ProductName**: A string (varchar) that stores the name of the product. This is necessary for displaying products to users. ProductName and Price are critical for product display and sales transactions.

**Price**: A decimal value that stores the price of the product. This is essential for pricing and financial calculations.

## Products\_Orders

The Products\_Orders entity represents the relationship between products and orders. It tracks the quantity of each product included in an order and calculates the total price for the product in that order. This entity is essential for order fulfillment and inventory management.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 6: Products\_Orders Table

**ProductID** (Foreign Key): A reference to the ProductID in the Products table. This links the product to the order.

**OrderID** (Foreign Key): A reference to the OrderID in the Orders table. This links the order to the product. ProductID and OrderID establish the relationship between products and orders.

**Quantity**: An integer that stores the quantity of the product in the order. This is necessary for inventory and pricing calculations.

**ProductOrderID** (Primary Key): A unique, auto-incremented integer that identifies each product-order relationship.

**TotalPrice**: A decimal value that stores the total price for the product in the order (calculated as Quantity \* Price). This is essential for financial tracking. The Quantity and TotalPrice are critical for order fulfillment and financial reporting.

## Shipments

The **Shipments** entity tracks the delivery of orders. It stores shipment details such as the shipment date, tracking number, delivery date, and status. This entity is essential for logistics and customer communication.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 7: Shipments Table

**ShipmentID** (Primary Key): A unique, auto-incremented integer that identifies each shipment. This ensures each shipment is uniquely identifiable. The ShipmentID ensures unique identification of shipments.

**ShipmentDate**: An integer field that records the date when the shipment was initiated. This is necessary for tracking shipment timelines.

**TrackingNumber**: A string (varchar) that stores the tracking number for the shipment. This is essential for customer communication and logistics.

**DeliveryDate**: A date field that optionally records the date when the shipment was delivered. This is useful for tracking delivery performance.

**ShipmentStatus**: A string (varchar) that indicates the status of the shipment (e.g., "Shipped", "Delivered"). This is critical for order tracking.

**OrderID** (Foreign Key): A reference to the OrderID in the Orders table. This links the shipment to the corresponding order. OrderID establishes a relationship between shipments and orders.

## Products\_Categories

The Products\_Categories entity establishes a many-to-many relationship between products and categories. It allows a product to belong to multiple categories and a category to include multiple products. This entity supports flexible product classification.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 8: Products\_Categories Table

**CategoryID** (Foreign Key): A reference to the CategoryID in the Categories table. This links the product to a category.

**ProductID** (Foreign Key): A reference to the ProductID in the Products table. This links the category to a product.

CategoryID and ProductID together establish a many-to-many relationship, enabling flexible product categorization.

# Relationships and Their Justifications

The database schema defines structured relationships among entities to ensure data integrity and optimize performance. These relationships fall into three categories: one-to-one (1:1), one-to-many (1:n), and many-to-many (n:m). Each relationship is established using foreign keys and, where necessary, intermediary tables that serve as independent entities.

For the web application all relationships are defined in the Configurations folder of DB\_Ecommerce.Persistence.

## One-to-One (1:1) Relationships

The relationship between Orders and Payments is one-to-one. Each order has exactly one corresponding payment, and each payment is linked to a single order. This structure ensures that every order is associated with a single financial transaction, preventing partial or duplicate payments for the same order. The Payments table contains the OrderID as a foreign key, enforcing this exclusivity. This design is particularly useful for tracking transactions, ensuring that every order is settled with one and only one payment. Payments is dependent on orders, i.e. an order must be created to which the payment can be assigned. Therefore, for each order one payment needs to be created. If an order gets deleted, it’s payment will be deleted automatically.

## One-to-Many (1:n) Relationships

The Customers to Orders relationship follows a one-to-many structure. A single customer can place multiple orders over time, but each order is linked to only one customer. This is achieved through the CustomerID foreign key in the Orders table, allowing the system to track all purchases made by an individual customer. Orders is dependent on customers, i. e. you need a customer before you can create an order.

Similarly, the Orders to Shipments relationship is one-to-many. A single order may require multiple shipments, particularly if products are dispatched separately. The Shipments table maintains the OrderID foreign key to establish this link. This design ensures that the system can track the shipping status of each order accurately. Shipments is dependent on orders. This means shipments can only be created if there is an order which it connects to.

The Products and Products\_Orders relationship follows a one-to-many model. A product can appear in multiple orders, but each entry in the Products\_Orders table corresponds to a specific product in a specific order. The Products\_Orders table includes ProductID as a foreign key to establish this connection. This structure is necessary for tracking which products were ordered and in what quantities, enabling efficient inventory management and order processing.

Also, the Orders and Products\_Orders relationship is one-to-many. An order can contain multiple products, but each entry in the Products\_Orders table links a product to a specific order. The OrderID is stored as a foreign key in Products\_Orders, ensuring that all products within an order are accurately recorded. This structure is essential for ensuring precise billing, inventory management, and order fulfillment. For each element in Products\_Orders you need a product and an order since the table Products\_Orders is dependent on the Products and the Orders table.

## Many-to-Many (n:m) Relationships

The Products and Categories relationship is many-to-many, as a product can belong to multiple categories, and a category can contain multiple products. This relationship is managed by the Products\_Categories table, which consists solely of foreign keys (ProductID and CategoryID). This table serves purely as a linking entity without additional attributes. For each connection in Products\_Orders you need a product and a category since the table Products\_ Categories is connecting the Products and the Categories table.

In contrast, the Orders and Products relationship, managed by the Products\_Orders table, is not just a joint table but an independent entity. An order can contain multiple products, and a product can be part of multiple orders. The Products\_Orders entity captures this relationship by storing OrderID and ProductID alongside additional attributes such as Quantity, TotalPrice, and ProductOrderID as the primary key. This structure enables precise order management, allowing the system to track the exact number of units purchased and their total cost per order.

# Entity-Relationship Diagram (ERD)

**Ein Bild, das Text, Diagramm, Screenshot, Plan enthält.

KI-generierte Inhalte können fehlerhaft sein.**

Figure 9: Entity-Relationship Diagarm of the E-Commerce Database

The Entity-Relationship Diagram (ERD) illustrates the structure of the e-commerce database and the relationships between key entities such as customers, orders, payments, products, and shipments.

Customers and orders have a one-to-many relationship, where a customer can place multiple orders, but each order belongs to one customer. The CustomerID in the Orders table connects them.

Orders and payments are linked through a one-to-one relationship. Each order has one corresponding payment, represented by the OrderID in the Payments table. This ensures accurate tracking of transactions.

Orders and Products\_Orders share a one-to-many relationship, as an order can contain many products. The OrderID in Products\_Orders connects each product to the relevant order.

Products and Products\_Orders also follow a one-to-many relationship. A product can appear in many orders, and each record in Products\_Orders represents a specific product within a specific order.

The Products\_Orders table is not just a join table but also an entity with attributes like Quantity and TotalPrice, allowing detailed tracking of orders.

Orders and shipments are connected by a one-to-one relationship, with each order linked to a shipment via the OrderID. This ensures that delivery details are associated with the correct order.

The Products\_Categories table exists in the database but is not used in the application code. While it defines a many-to-many relationship between products and categories, it is currently inactive in the system.

# CRUD Opperations

In the e-commerce application, the CRUD operations for each entity are implemented within the *Applications* directory. For every entity, there is a dedicated folder that contains separate files for each CRUD operation. Each operation (Create, Read, Update, Delete) is handled by an individual file to maintain a clear and modular structure. For example, in the Orders folder, there are files like CreateOrderCommand.cs, CreateOrderCommandHandler.cs, GetOrderQuery.cs, and so on, each responsible for a specific action.

Entities in the application may depend on others. For example, Shipments can only be created if an Order exists. This ensures data integrity by preventing actions like creating shipments or payments for non-existent orders. Such dependencies are enforced in the application logic, with error handling to prevent invalid operations.

The backend routes are defined within the MVC controllers. Each entity has its own controller responsible for processing HTTP requests related to that entity. These controllers handle the creation, retrieval, updating, and deletion of data. For instance, the OrdersController manages the lifecycle of orders, while the PaymentsController and ShipmentsController manage their respective entities. The controllers delegate the actual processing to the corresponding command handlers.

This structured approach ensures clear separation of concerns, maintainability, and scalability. By organizing the application in this way, dependencies are respected, and each entity's logic is handled consistently across the application, ensuring both clean code and data integrity.

# MongoDB Collection Design

## Use Case

### General Use Cases

MongoDB greatly differentiates itself from relational databases due to its more flexible approach to data handling. MongoDB excels in environments with dynamic data, with large amounts of data in a document-oriented structure (like e.g. JSON). It does not excel in situations where data must strictly be consistent or where data is highly relational and requires complex joins.

### E-Commerce Use Case

In this application MongoDB is used to store customer ratings for products. Customers can rate products on a numeric scale with comments detailing their experiences and expectations. The goal was to create a database with CRUD (Create, Read, Update and Delete) functionalities that dynamically react to further improvements in the review system.

A key use case is the calculation of average ratings to help lead customers to a buying decision. This should be handled directly on the product page.

Due to the variability of customer reviews, MongoDB has been chosen instead of a relational database to better accommodate different review styles. These differences may be in comment length, picture usage or tags. Furthermore, MongoDB simplifies further extension to the system due to not being dependent on a rigid data structure.

To simplify the local installation and to experiment with cloud-based MongoDB instances, the project uses MongoDB Atlas. A cloud-based approach to hosting MongoDB instances.

## Schema Design

Ein Bild, das Text, Screenshot, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.Each document containing a review consists of a document id, the product id and a user id which doubles as customer identification joined by the comment itself, a rating and a date of creation.

Figure 10 Review Model in Project

Noteworthy hereby is the assignment of the document id and creation date by the system, while the other values are assigned by the user. Those user-assigned values are then used to assign reviews to customers (user id) and products (product id).

Since MongoDB does not rely on a fixed scheme, the review system can deal with enhancements to the system, like responses to reviews or additional metadata to filter reviews more effectively in the future.

## Integration with Backend

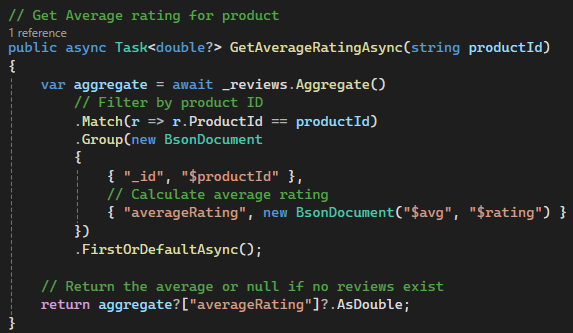
The database is connected to the rest of the project with a connection string, containing username and password for a MongoDB Atlas database and the associated connection parameters. All the necessary settings are stored in the appsettings.json file to, switch databases as easily as possible.

The connection is established in the DB\_ECommerceContext class, where the constructor sets up a new MongoDB client with the aforementioned credentials.

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 11 DB\_ECommerceContext constructor

The ReviewRepository.cs file acts as Data Access Layer between the backend and database. This layer is necessary to separate business logic from database access, therefore simplifying flexibility and modularity.

In the code snippet a double or null value is returned by the GetAverageRatingAsync function, which requires a product id to collect reviews matching this id. These reviews are stored in the \_reviews collection, which gets filtered by the .match() according to the given product id. .group() then groups the reviews by product id and calculates the average of the rating fields before returning an “\_id” and “averageRating” key/value pair.

Figure 12 GetAverageRating in ReviewRepository

In the last step the function either returns null if there are no values or returns “averageRating” as double if a value is calculated.

## CRUD Operations with MongoDB

CRUD Operations in this project are first handled in the ReviewsController, where the request is processed and forwarded to a command or query function which handles the request and a handler, which deals with the response after contacting the repository.

Ein Bild, das Text, Schrift, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 13 Create Review function in controller

The code snippet above highlights the API call to create a review. The HTTP Post method is used to create a review with the content specified in CreateReviewCommand and entered in the frontend.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 14 CreateReviewCommand snippet

The CreateReviewCommandHandler then creates a new Review object according to the model filled with the input from the CreateReviewCommand.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 15 CreateReviewCommandHandler function

To save the review, it is forwarded to the repository via this code:

await \_reviewRepository.CreateReviewAsync(review);

The function returns the created review, when the command to enter it into the database is done.

In practice, the review feature is used to show an average value of ratings, defaulting to “No Ratings” if there are no ratings available in the MongoDB.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

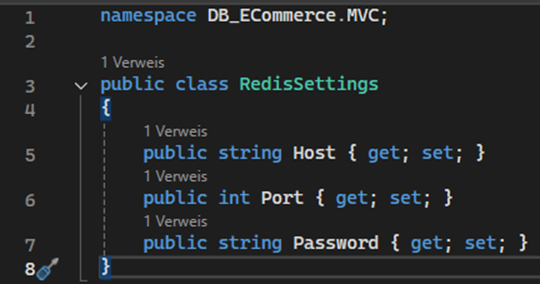
Figure 16 MongoDB usage in the project

# Redis Caching Implementation

Redis is used in this e-commerce project to cache product data to improve performance and reduce database queries. The following data is for example cached:

* Product Details: This includes ProductID, ProductName, and Price.
* Serialized JSON Objects: Data is stored in JSON format to facilitate easy retrieval and deserialization.

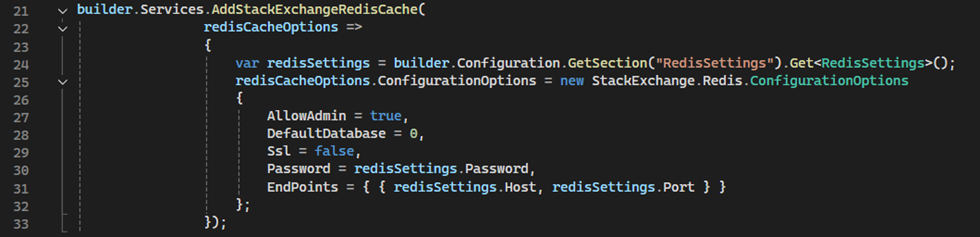
## Definition of the Redis settings class



This class defines the settings for the connection to Redis in RedisSettings.cs.

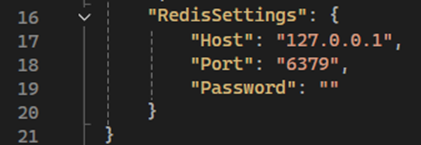
## Changes to the Program.cs file

Adding the Redis cache configuration

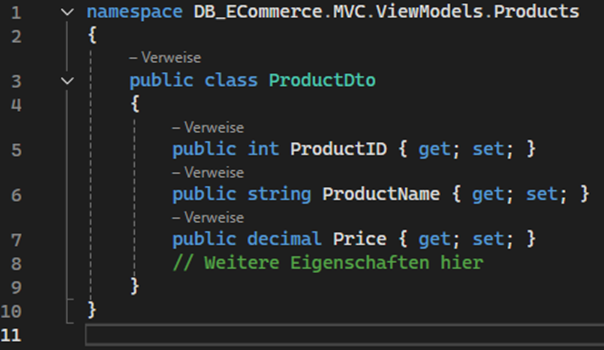
This configuration adds the Redis cache service and configures it with the settings from the appsettings.json file.

Changes to the appsettings.json file

The RedisSettings class defines the connection settings for Redis. These are stored in the appsettings.json file:



Creating the ProductDto class



The file ProductDto.cs defines a Data Transfer Object (DTO) class called ProductDto in the namespace DB\_ECommerce.MVC.ViewModels.Products.

### Meaning of the ProductDto class:

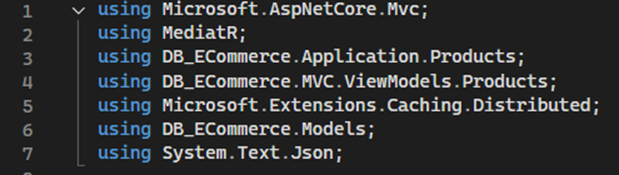
* **Data transfer:** The ProductDto class is used to transfer product data between different layers of the application, in particular between the presentation layer and the business logic or data access layer.
* **Decoupling:** By using DTOs, the decoupling of the internal data models from the external interfaces is achieved. This means that changes to the internal data models do not directly affect the external interfaces.
* **Security and control:** DTOs make it possible to transfer only the necessary data and exclude sensitive information. This increases security and control over the data that is exchanged between the layers.

### Properties of the ProductDto class:

* ProductID: A unique identifier for the product.
* ProductName: The name of the product.
* Price: The price of the product.

## Changes to the ProductsController class

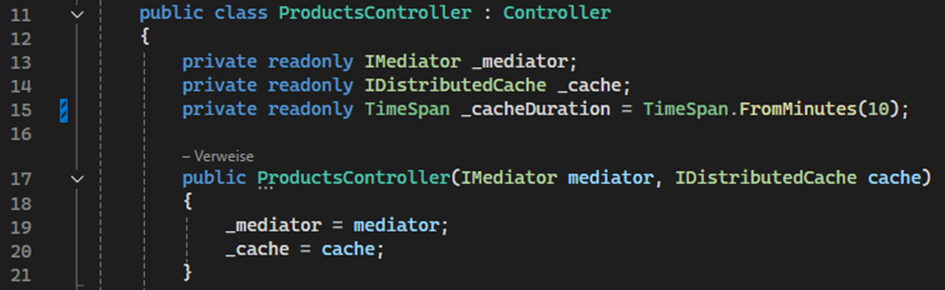
1. addition of dependencies



The following namespaces have been added to use the required classes and methods for the Redis implementation:

* **Microsoft.Extensions.Caching.Distributed:** Enables interaction with the distributed cache (Redis).
* **using System.Text.Json;** is used to provide JSON serialisation and deserialisation functions. These functions are important for converting objects to JSON format and vice versa.

### 2. adding fields and constructor



Two additional private read-only fields are defined in the ProductsController:

**\_cache:** A field of type IDistributedCache, which is used for the interaction with the distributed cache (in this case Redis).

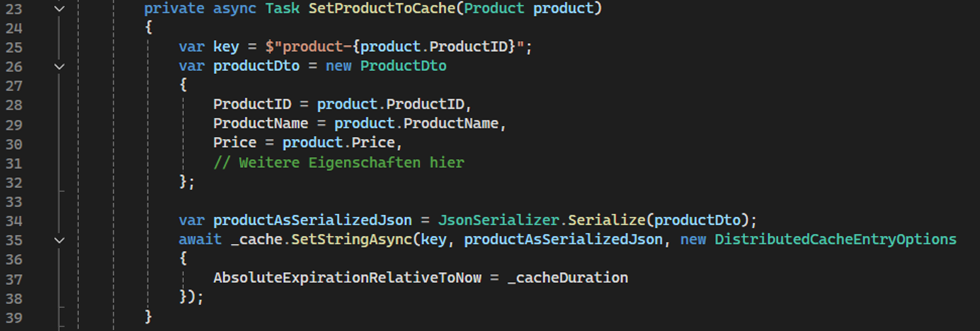
**\_cacheDuration:** A field of type TimeSpan that specifies the duration for which the data is to be stored in the cache. In this case, the cache duration is set to 10 minutes. This is a balanced choice because:

* It significantly reduces database load by serving frequently requested products from the cache.
* It ensures that product data remains relatively fresh without being stale for long periods.
* It provides a good trade-off between performance and data accuracy, ensuring updates in the database propagate to users within a reasonable timeframe.
* It minimizes the risk of users seeing outdated product details while maintaining fast response times.

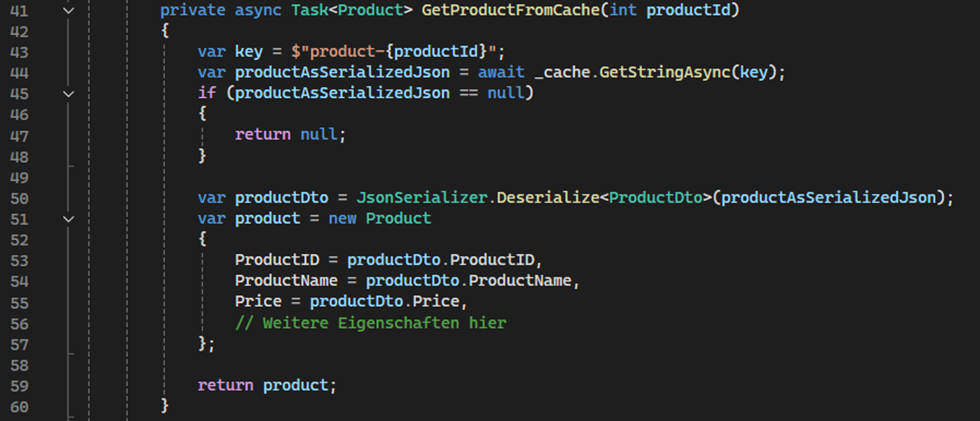
The constructor then initialises these fields.

### 3. methods for managing the cache

**SetProductToCache**

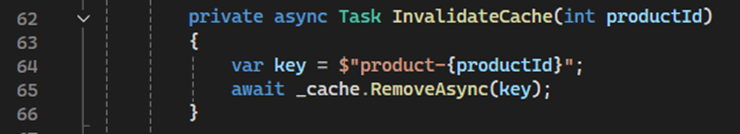
This method saves a product in the Redis cache. It takes a product, generates a unique key for caching, and assigns the product’s properties to a ProductDto object. The object is then serialized into a JSON string and stored in the cache with the unique key for a defined duration.

**GetProductFromCache**



This method reads a product from the Redis cache. Initially, a cache key is generated using the product's ID. The cache is then checked for this key; if it does not exist, null is returned. If the product is found, the JSON string is deserialized into a ProductDto and subsequently converted back into a Product object before being returned.

**InvalidateCache**

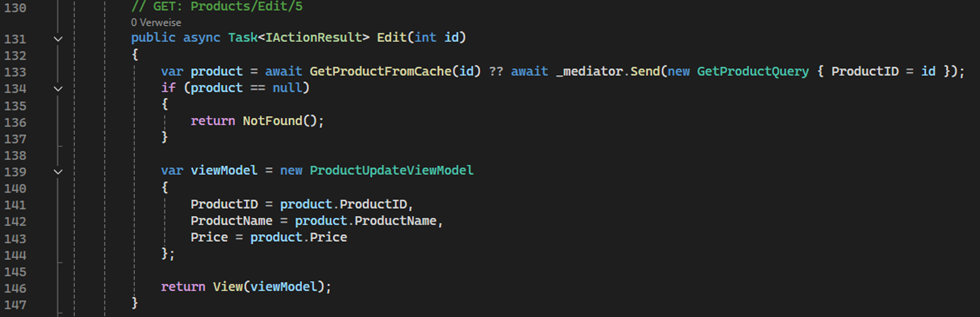


This method mirrors the previous ones: the product is identified by its ID and subsequently removed from the cache.

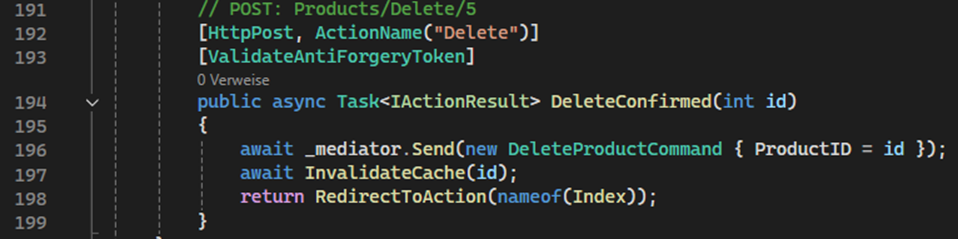
### 4. changes to the controller methods

**Create method**

In the Create method, the created product is saved in the cache after it has been successfully created.

**Edit method** In the Edit method, the updated product is saved in the cache after it has been successfully updated.

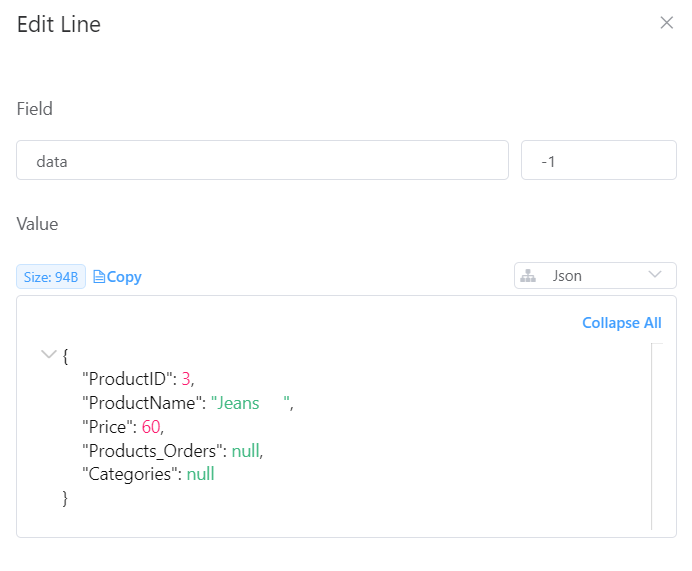
**DeleteConfirmed method**



In the DeleteConfirmed method, the product is removed from the cache after it has been successfully deleted.

Another Redis Desktop Manager

The Another Redis Desktop Manager allows viewing cached data in a structured format. It provides access to all cached entries. For example, the product 'Jeans' with ProductID 3 is listed in the cache overview. Clicking on this product displays its cached data in JSON format.



# Frontend Development / MVC

## What is the MVC?

The MVC (Model, View, Controller) is software design pattern that divides an application into three interconnected parts: Model, View, and Controller. This separation improves code organization, maintainability and scalability. It is widely used in web application development, including ASP.NET Core MVC, to create structured and efficient applications.

The Model represents the data and business logic of the application. It applies business rules, validates data, and communicates with the database. Models are usually implemented as classes in ASP.NET Core MVC applications, which use Entity Framework Core to control database interactions.

The View is responsible for displaying the user interface and presenting data to users. Because views are usually built in Razor syntax, developers can dynamically combine C# code and HTML. The controller sends data to the view, which then presents it in a user-friendly format.

The Controller functions as an interface for the View and the Model. It handles incoming HTTP requests, processes user inputs, retrieves data from the Model, and selects the appropriate View to display. Controllers define how user requests are processed and contain the application's main logic.

### How MVC works in a Web Application

1. A user makes a request to the application by entering a URL or interacting with the UI.
2. The request is sent to the appropriate Controller, which processes it.
3. The Controller communicates with the Model to retrieve or manipulate data.
4. The data is then passed to a View, which generates and returns an HTML response.
5. The user sees the rendered page in their browser.

### Benefits of using MVC

* Separation of Concerns: The code is easier to scale and maintain because the Model, View, and Controller components are all clearly defined.
* Reusability: Views and models can be reused across different parts of the application.
* Testability: Unit testing becomes more effective since business logic and user interface components are kept separate.
* Scalability: New features can be added to MVC applications while maintaining the organization of the current codebase.

## The individual MVC components

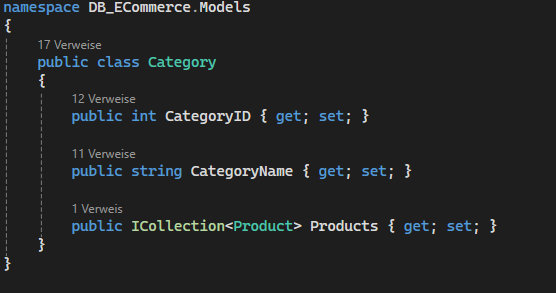
### Model

A Model in ASP.NET Core MVC is a C# class that defines the structure of an entity, such as Category, Product, or Order.

**Purpose of Models**

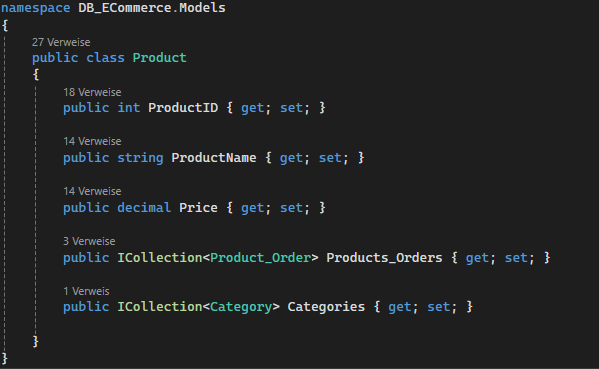
* Represent database tables when using Entity Framework Core.
* Define relationships between entities (One-to-Many, Many-to-Many).
* Contain validation rules using Data Annotations.
* Can include business logic related to the entity.

**Example of a Model (Category.cs)**



* CategoryID: The primary key for the Categories table.
* CategoryName: A required field that cannot exceed 100 characters.
* Products: A collection of products, establishing a One-to-Many relationship.

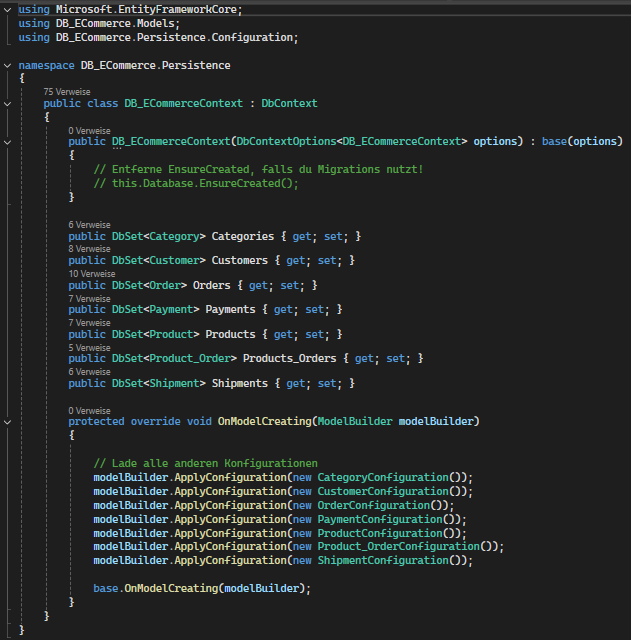
**Example of a Related Model: The Product Entity**



* ProductID: The primary key for each product.
* ProductName: Stores the name of the product.
* Price: The cost of the product.
* Products\_Orders: Many-to-Many relationship with Orders.
* Categories: Many-to-Many relationship with Categories.

**Managing Models with DbContext**

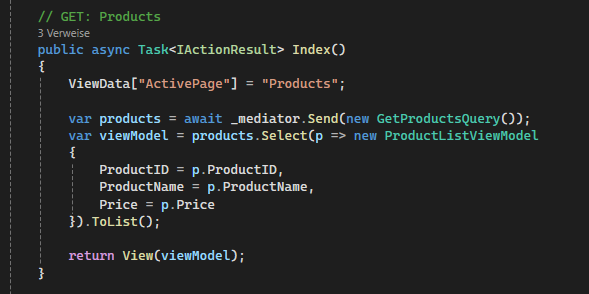
To connect these models to a database, we use DbContext, which serves as the bridge between C# objects and the database.



* DbSet<Category> and DbSet<Product>: Represent database tables.
* OnModelCreating(): Explicitly defines One-to-Many relationships.

**Using the Model in a Controller**

The Product model is used in a controller to retrieve data from the database and pass it to views.

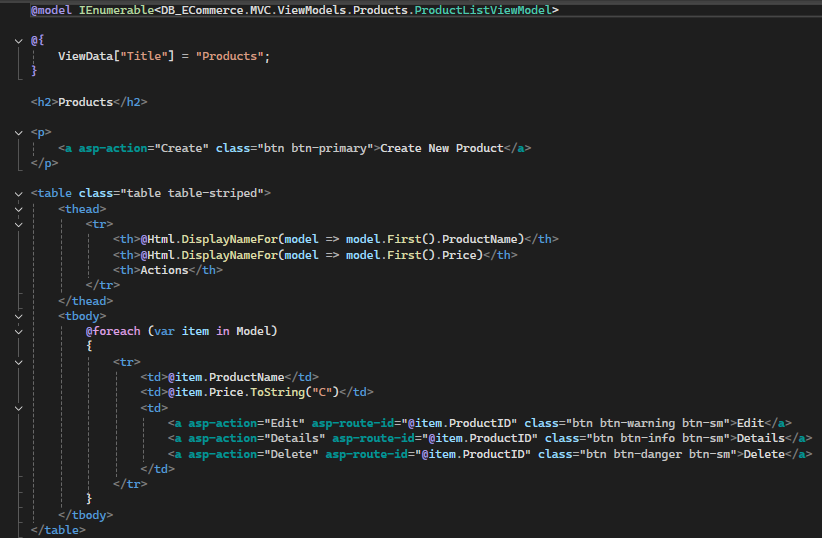


* Index() is an action method inside the ProductsController. It is executed when a user navigates to /Products/Index.
* The method uses MediatR to communicate with the application layer and fetch a list of all products.
* The GetProductsQuery() command is sent to retrieve product data from the database.
* The retrieved products are transformed into a list of ProductListViewModel, which is then passed to the view.
* Finally, the method returns the view, where the products are displayed.

**Displaying the Model in a View**

To render the products in a view (Index.cshtml), we use Razor syntax to dynamically generate the HTML content.

The following code is used in Index.cshtml to display the list of products:



* @model IEnumerable<DB\_ECommerce.MVC.ViewModels.Products.ProductListViewModel>: This declares that the view receives a list of products as a ProductListViewModel.
* Displaying Table Headers: @Html.DisplayNameFor(model => model.First().ProductName) dynamically retrieves and displays the property names of the first item in the list.
* Iterating through Products (@foreach): Each product's name and price are displayed in the table. The price is formatted as currency using .ToString("C").
* Actions Section: For each product, there are "Details," "Edit," and "Delete" buttons that allow users to manage products.

### Controllers

In the MVC design, the Controller serves as a bridge connecting the View (user interface) and the Model (data layer). It responds to user queries, uses the Model to communicate with the database, and returns relevant Views.

**Purpose of Controllers**

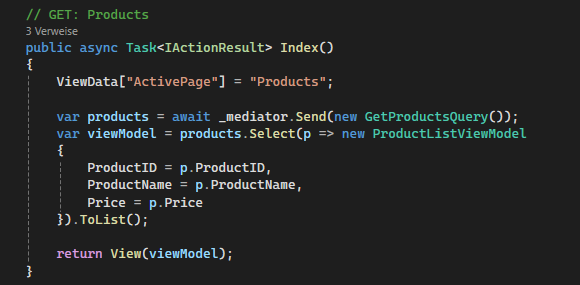
* Carries out business logic and responds to incoming queries.
* Uses the Model to retrieve data from the database.
* Uses ViewModels to pass data to Views.
* Outlines the actions and routes that control how users engage with the program.

**Example: ProductsController**



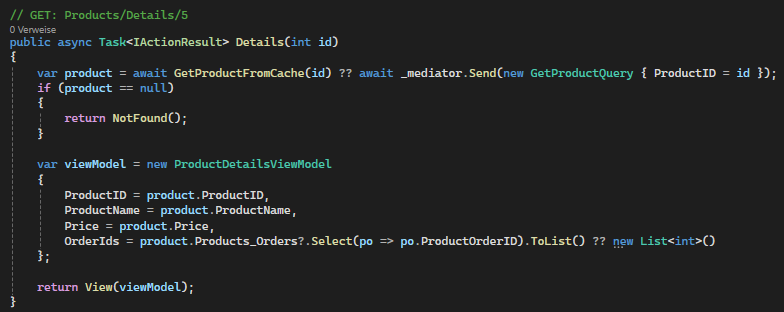
* The ProductsController inherits from Controller, making it an MVC controller.
* It uses MediatR to send and retrieve product data.
* Implements distributed caching (IDistributedCache) to improve performance.
* Defines a cache duration of 2 minutes for storing product data.

**Retrieving All Products**



* Retrieves products using MediatR (GetProductsQuery).
* Converts product data into a ProductListViewModel list.
* Passes the list to the Index View.
* Returns the View, which displays the products.

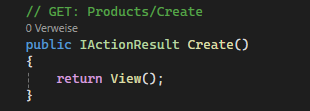
**Displaying Product Details**



* Retrieves a specific product by id from the database.
* If no product is found, returns a NotFound() response.
* Maps data to ProductDetailsViewModel.
* Returns the View with product details.

**Creating a New Product**

GET: Display Create Form:



* Returns the Create View, where users can enter product details.

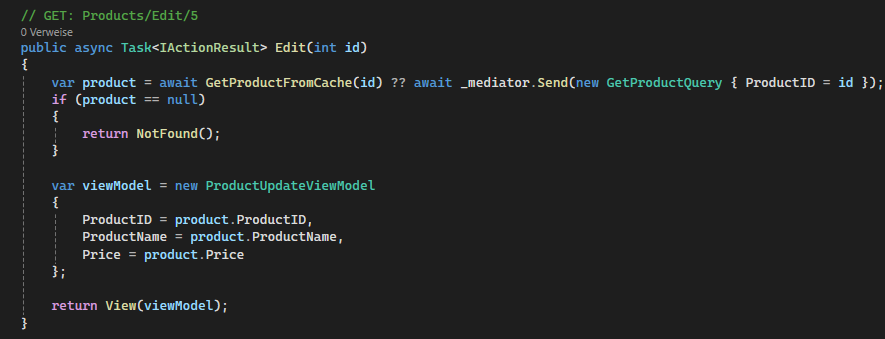
POST: Process Product Creation:



* Receives data from the Create form.
* Validates user input using ModelState.IsValid.
* Sends a CreateProductCommand via MediatR.
* Redirects back to Index if successful.

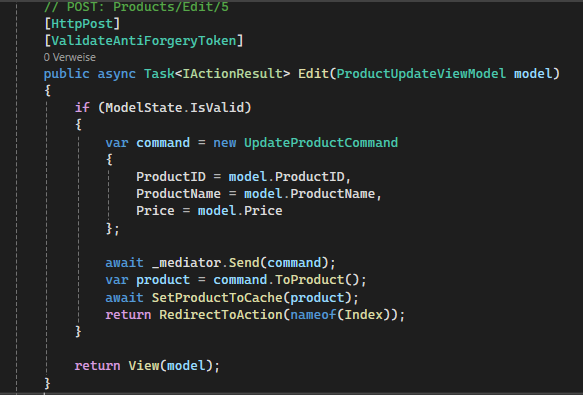
**Editing an Existing Product**

GET: Load Product for Editing



* Retrieves product data for editing.
* If the product doesn’t exist, returns NotFound().
* Passes data to the Edit View.

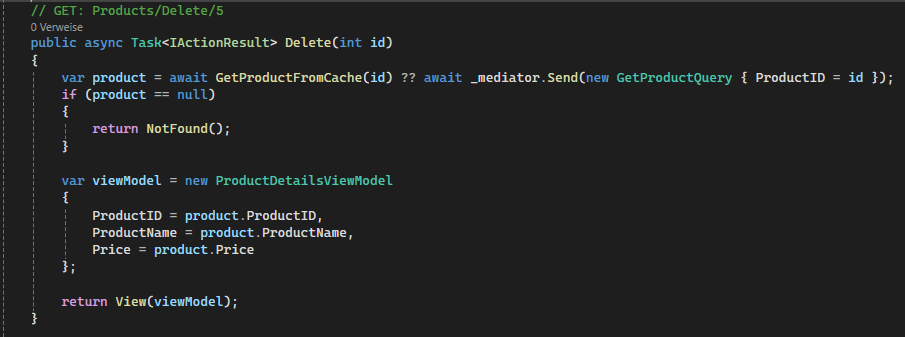
POST: Save Changes



* Checks if input is valid (ModelState.IsValid).
* Sends an UpdateProductCommand to update the database.
* Redirects to Index if successful.

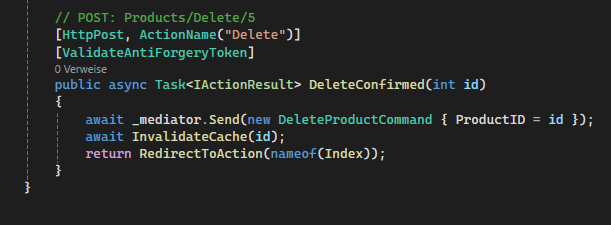
**Deleting a Product**

GET: Confirm Deletion



* Retrieves product data.
* If not found, returns NotFound().
* Passes product details to the Delete confirmation page.

POST: Confirm & Delete



* Deletes the product using DeleteProductCommand.
* Redirects to the product list (Index) after deletion.

### Viewmodels

In an MVC application, ViewModels serve as a layer in between Models and Views. Data is passed to the View after being shaped, organized, and occasionally transformed. ViewModels, in contrast to Models, cater to the particular requirements of the user interface rather than directly representing database instances.

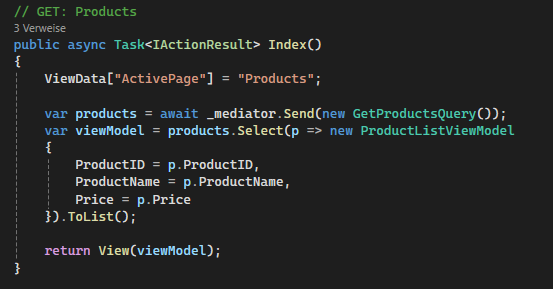
**Why Use ViewModels?**

ViewModels reduce unnecessary data retrieval and enhance speed by choosing only the properties required for the View, resulting in a tailored data representation. They help to maintain a separation of concerns, enabling the UI to show customized information while keeping Models focused on database interactions. ViewModels also improve security by limiting overposting and guaranteeing that users can only alter data that is allowed. This method maintains the application's security, maintainability, and efficiency.

**How ViewModels are used in Controllers**

In a Controller, ViewModels help structure data before sending it to the View.

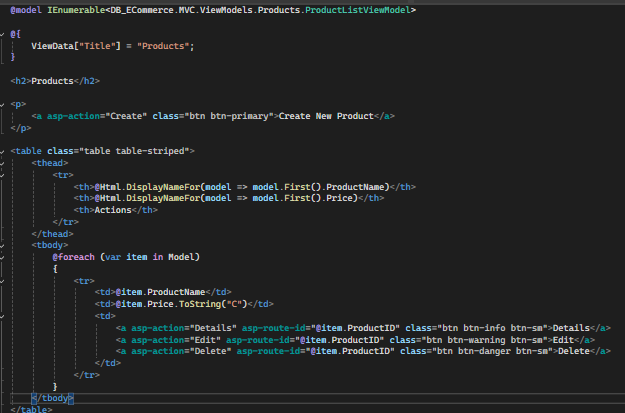
Example: Using ProductListViewModel in a Controller



* The Index action retrieves a list of products.
* It converts each Product into a ProductListViewModel using FromProduct().
* The View only receives necessary data, improving performance and maintainability.

**Displaying Data in a View**

The View receives the ViewModel and renders it dynamically.



* @model IEnumerable<ProductListViewModel>: Defines that the View expects a list of Product ViewModels.
* @item.ProductName and @item.Price: Displays only the necessary properties.
* The View never interacts with the database directly, maintaining the MVC separation of concerns.

### Views

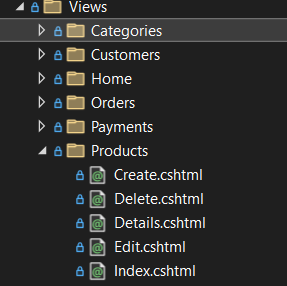
In the MVC design, the View component is in charge of showing the user the data. It offers the user interface (UI) for interacting with the application and specifies how information is displayed. Views are typically written using Razor syntax in ASP.NET Core MVC and use HTML, CSS, and C# code to dynamically render data retrieved from the Model.

**Purpose of Views:**

* Provide the user with data in an understandable and organized manner.
* Utilize UI components and forms to get user input.
* To change or retrieve data, speak with the Controller.
* Use Razor syntax (@model, @foreach, etc.) to support the display of dynamic content.
* Use CSS, JavaScript, and Bootstrap styles to increase UI responsiveness.

**Creating a view**

A View is created inside the /Views/ folder in an MVC project. In /Views/, each controller has a matching folder that contains several views associated with that controller.

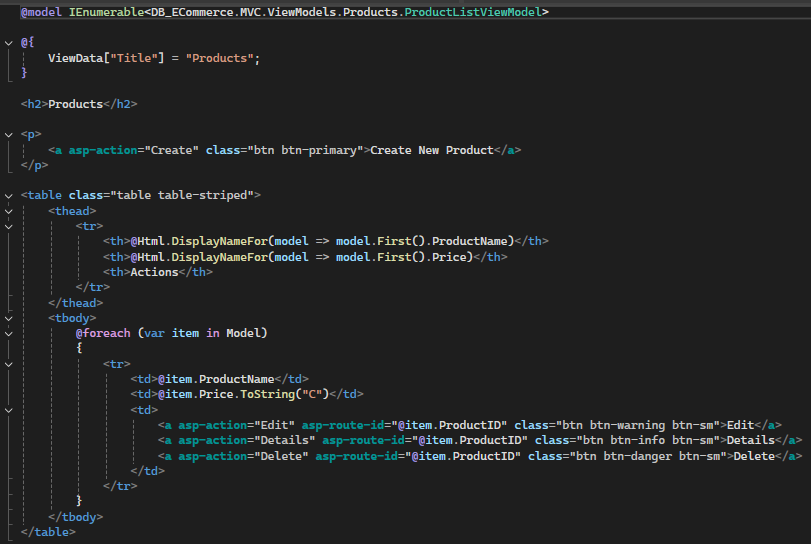


For example, in a ProductsController, the following views exist:

* Index.cshtml (List of products)
* Details.cshtml (Product details)
* Create.cshtml (Form to create a new product)
* Edit.cshtml (Form to edit an existing product)
* Delete.cshtml (Confirm deletion)

**Displaying Data in a View**

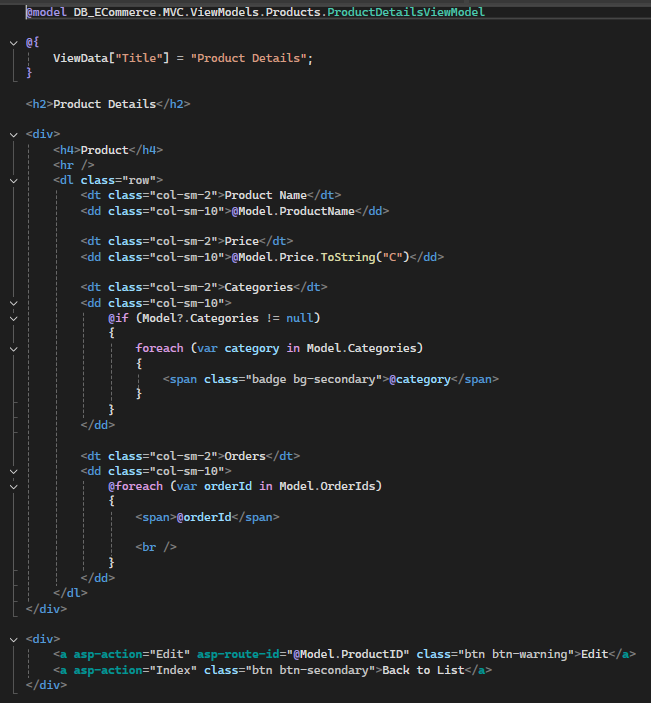
A View can receive data from the Controller through a ViewModel or directly from a Model. The @model directive defines the type of data the View expects.



* @model IEnumerable<DB\_ECommerce.MVC.ViewModels.Products.ProductListViewModel>: Defines that the View expects a list of products.
* @Html.DisplayNameFor(model => model.First().ProductName): Dynamically retrieves the column headers.
* @foreach (var item in Model): Iterates through the list of products.
* @item.ProductName: Displays the product name.
* @item.Price.ToString("C"): Formats the price as currency.
* The action buttons (Details, Edit, Delete) allow interaction with individual products.

**Rendering Dynamic UI with Views**

Views are not static HTML pages. They dynamically render content based on the data passed from the Controller. For exampke, in the Products/Details.cshtml View:



* @Model.ProductName: Displays the product name dynamically.
* @Model.Price.ToString("C"): Formats the price as currency.

The Categories section:

* Uses a @foreach loop to iterate through all assigned categories.
* Each category is wrapped in a Bootstrap badge (<span class="badge bg-secondary">@category</span>) for styling.

The Orders section:

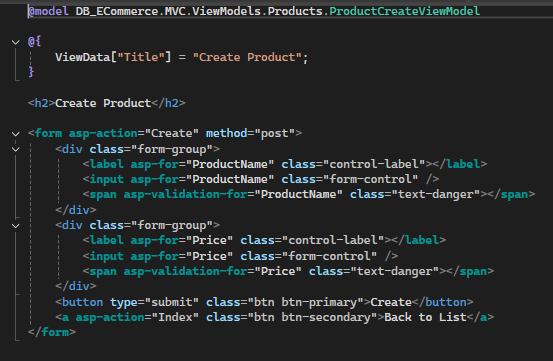
* Iterates through all associated Order IDs (@foreach (var orderId in Model.OrderIds)).
* Displays each Order ID in a new line (<br />).

Navigation buttons:

* The Edit button allows modifying the product.
* The Back to List button navigates to the product list.

**Using Forms to Accept User Input**

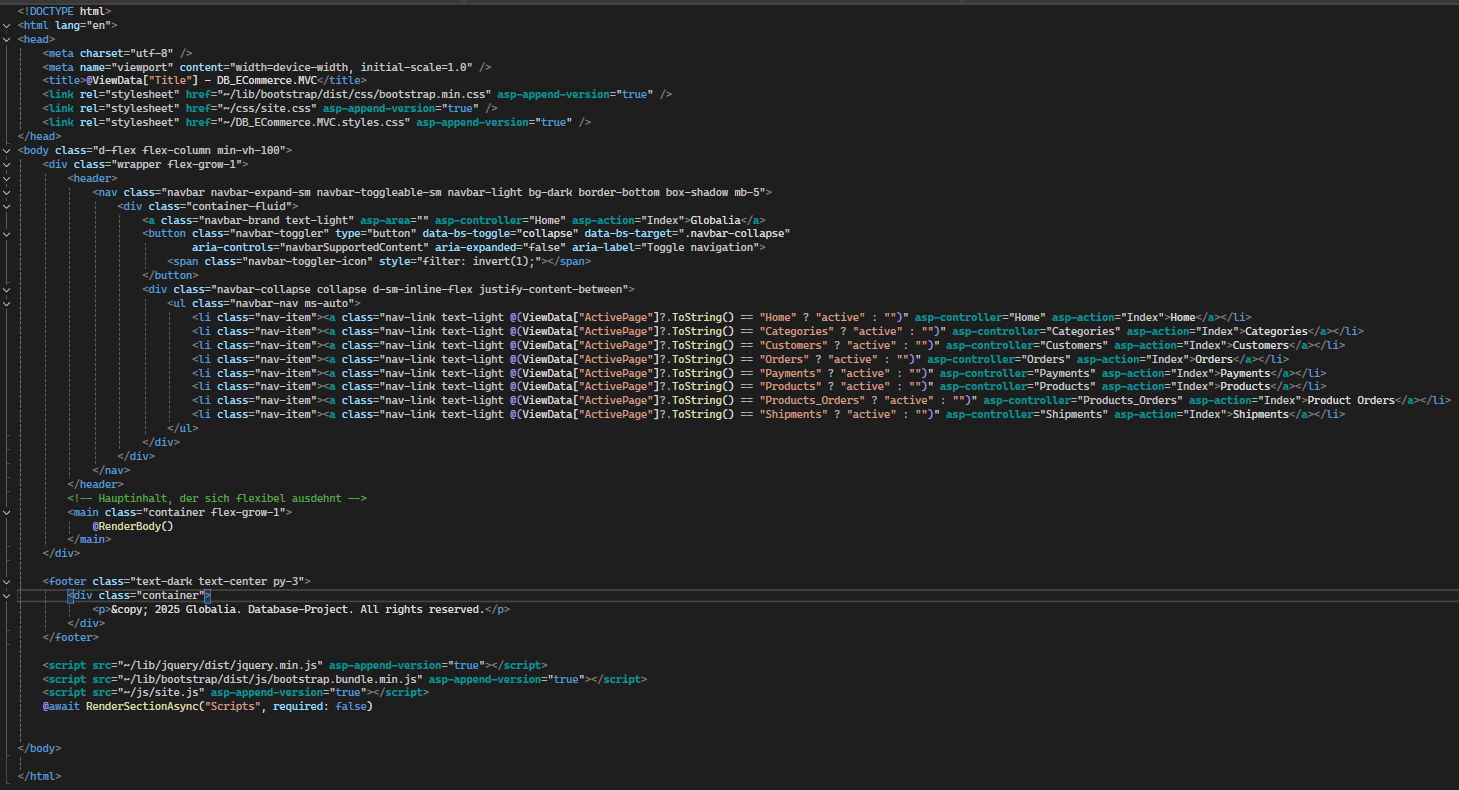
Views also contain forms that allow users to submit data (e.g., creating or editing a product).



* asp-for="ProductName": Binds the input field to the ProductName property.
* asp-for="Price": Ensures the price is numeric and has decimal support.
* asp-validation-for="ProductName": Displays validation errors if the field is empty.
* The Submit button sends the form data to the Controller.

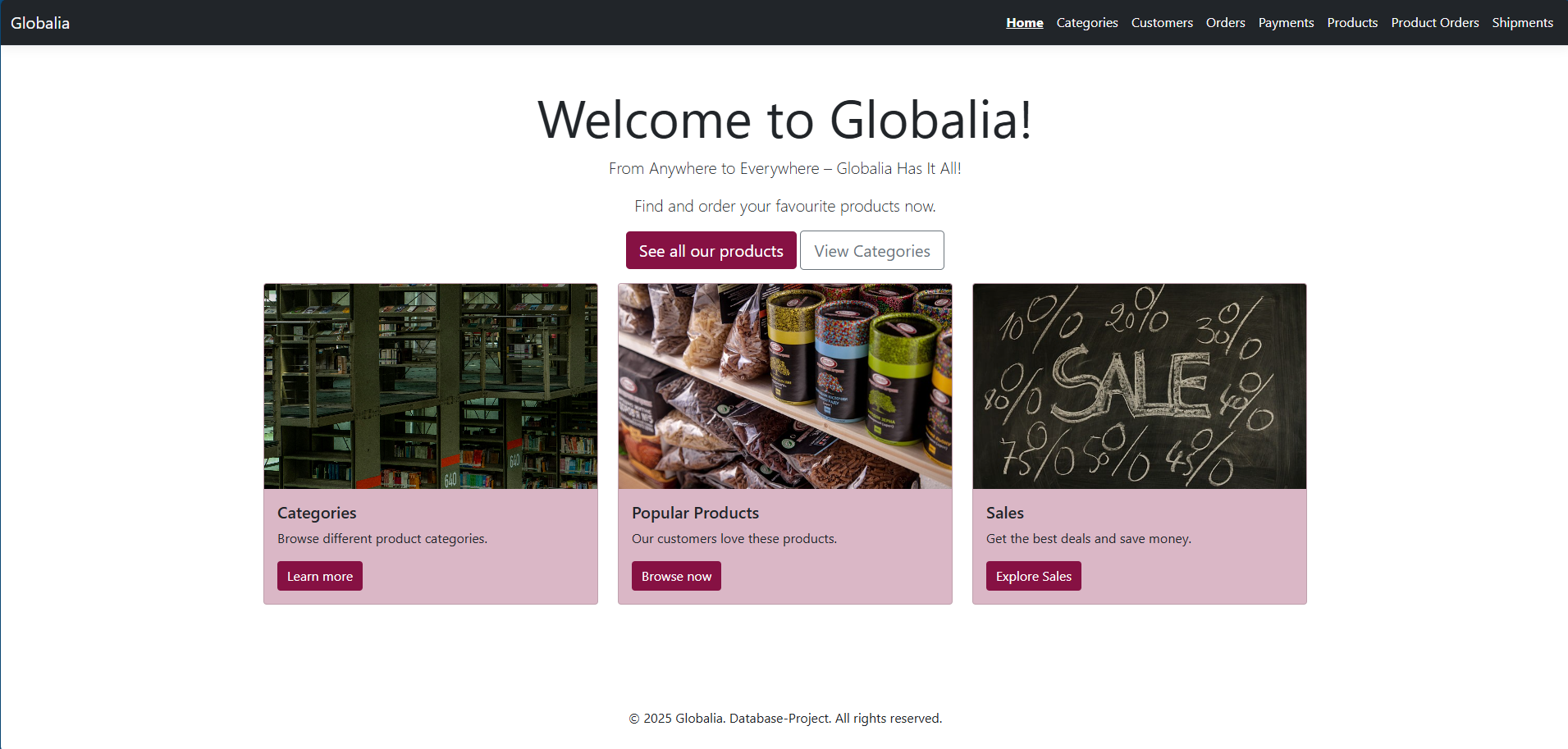
**View Layouts and Reusability**

Instead of writing the same header, navigation, and footer for every View, MVC uses a Layout View (\_Layout.cshtml) to maintain a consistent structure across all pages.



* @RenderBody() → Inserts the content of each View into the layout.
* The Navigation Bar provides easy access to different pages.
* The Footer remains consistent across all Views.

## Structure of our Website



Our homepage is designed to be engaging. It contains key features like Sales, Products and Categories.

The top navigation bar contains our logo "Globalia" on the left, linking to the homepage and navigation links on the right for Categories, Customers, Orders, Payments, Products, Product Orders and Shipments. The "Home" link is highlighted, indicating that this is the active page.

**Code: Navigation Bar in \_Layout.cshtml**



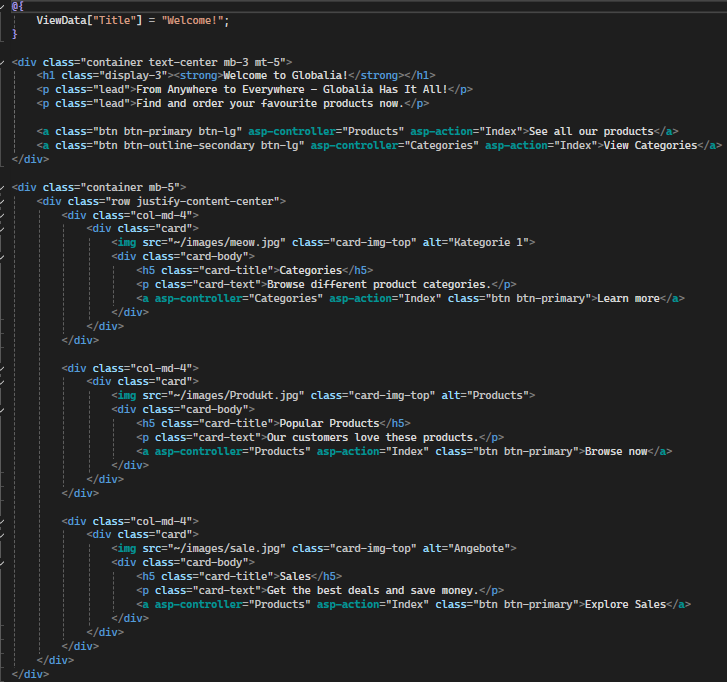
* Uses Bootstrap classes for responsiveness.
* The logo links back to the homepage.
* Navigation links direct users to important sections.

Below the navigation bar, the main welcome message is displayed in a large, centered format.

The homepage highlights three main sections in Bootstrap cards:

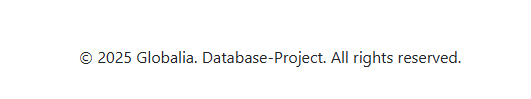
* **Categories:** Browse different product categories.
* **Popular Products:** Showcases the most loved products.
* **Sales:** Displays ongoing discounts and special deals.

**Code: Index.cshtml**

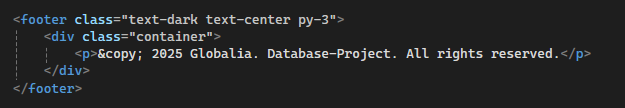


* Uses Bootstrap's grid system for responsive alignment.
* Images represent different sections (can be found in wwwroot/images).
* Each card has a title, short description, and button for navigation.

At the bottom of the page, a simple footer provides branding.

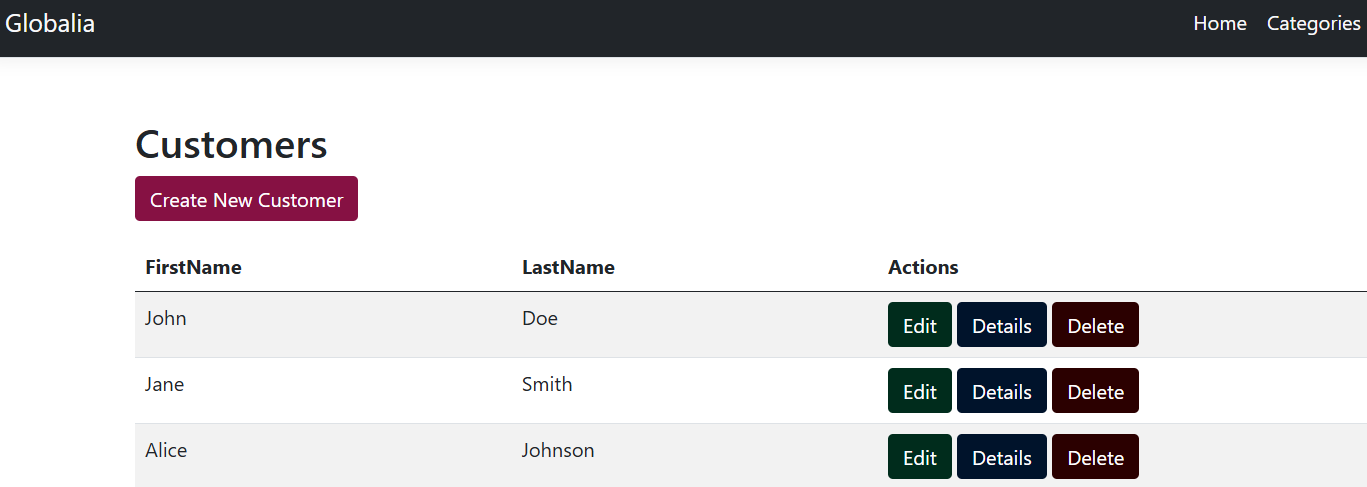


**Code: Footer in \_Layout.cshtml**



* The footer remains fixed at the bottom.
* Contains a copyright notice.

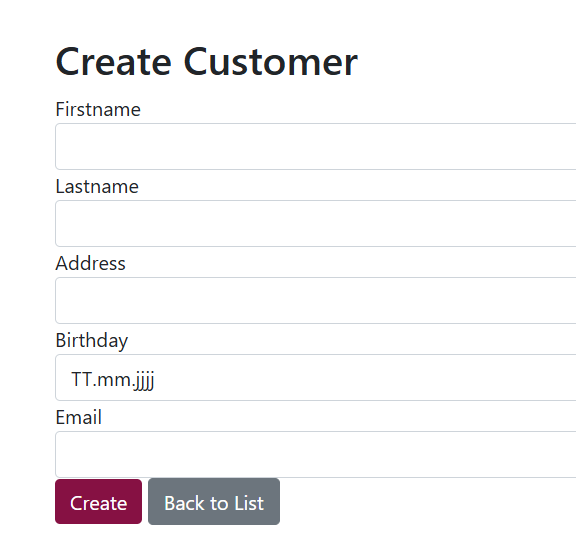
Clicking on one of the options in the navigation bar, for example, "Customers" redirects the user to this page:



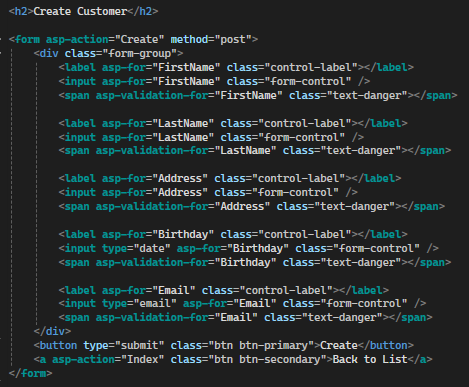
**Functionalities on this page:**

1. Create a New Customer

* Clicking the "Create New Customer" button redirects to a form where users can enter new customer details.
* This form typically consists of fields like First Name, Last Name, Email, and Address.

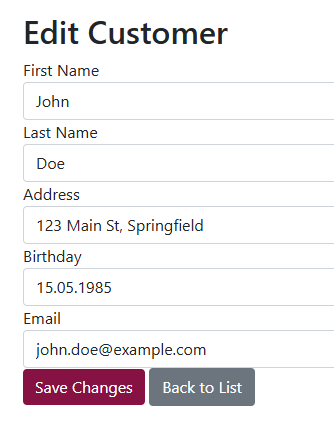


**Code: Create.cshtml**



1. Edit an Existing Customer

* The "Edit" button (green) allows users to modify customer details.
* Clicking it navigates to an Edit form pre-filled with the customer’s current data.

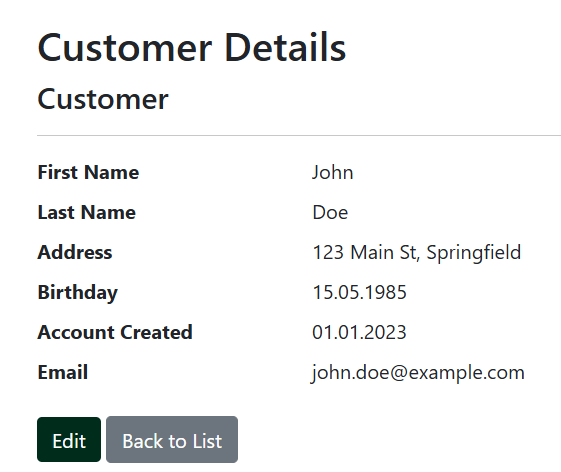


**Code: Edit.cshtml**

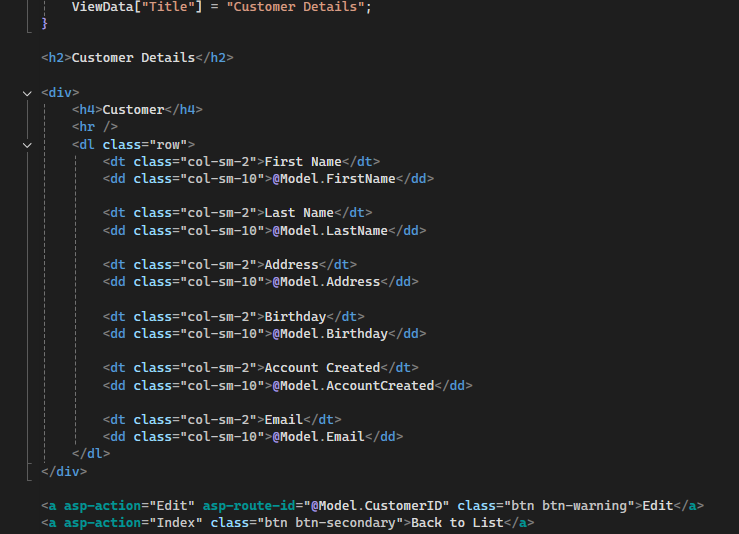


1. View Customer Details

* The "Details" button (blue) opens a page displaying full customer information.
* This may include order history, registered date, and additional details.

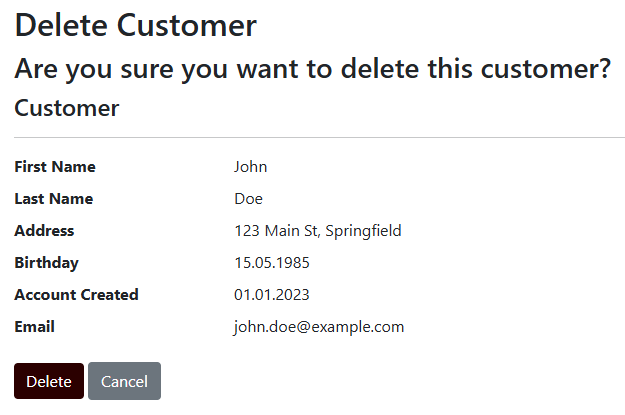


**Code: Details.cshtml**

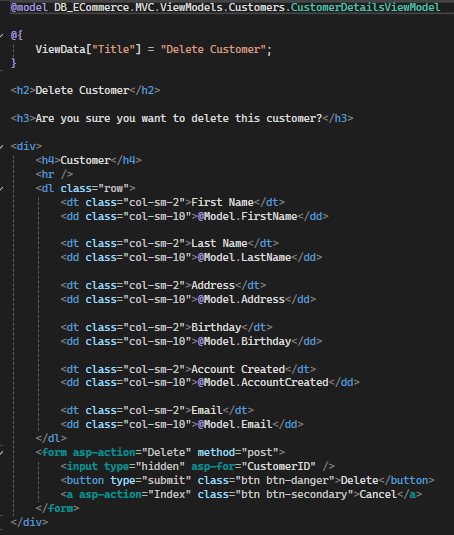


1. Delete a Customer

* The "Delete" button (red) prompts the user to confirm before permanently removing the customer from the system.



**Code: Delete.cshtml**



The Create, Edit, Details, and Delete functionalities provide users with essential tools to interact with the customer data. The web application's responsive design (using Bootstrap Grid System, Adaptive Buttons and Tables, Mobile-Friendly Navigation) makes it accessible on a variety of screen sizes, further improving the user experience.

By combining well-structured functionalities with a responsive design, the application ensures both usability and maintainability.

# Future Enhancements

The current E-Commerce database is designed to support core functionalities such as product management, order processing, customer data tracking, and payment handling. However, as the platform grows and user demands evolve, several enhancements can be implemented to improve scalability, performance, and user experience. Below are some proposed future enhancements.

**1. Database Optimization:** Implement indexing strategies to improve query performance. Optimize table structures by normalizing data where necessary. Enable partitioning for large tables to improve scalability.

**2. Security Enhancement:** Enforce encryption for sensitive data such as customer information and payment details. Implement role-based access control (RBAC) for database users. Regularly audit and log database activities for compliance.

**3. Data Integrity and Constraints:** Improve foreign key constraints to ensure data consistency. Implement cascading updates and deletes where necessary. Enforce data validation at the database level to reduce errors.

**4. Performance Tuning:** Enable query caching for frequently executed queries. Analyze execution plans and optimize slow queries. Regularly update statistics for efficient query processing.

**5. Scalability Improvements:** Transition to a cloud-based database for better availability. Implement read replicas to distribute query load. Introduce sharding for handling high transaction volumes.

**6. Feature Enhancements:** Add a new table for tracking customer reviews and ratings. Implement a product recommendation system based on customer purchase history. Enhance reporting capabilities with additional analytical views and stored procedures.

**7. Backup and Recovery:** Automate daily database backups with offsite storage. Implement point-in-time recovery to minimize data loss. Test backup restoration procedures periodically.

**8. Compliance and Regulations:** Ensure GDPR and PCI DSS compliance for handling customer data. Implement data retention policies for legal requirements. Provide audit trails for financial transactions.

# Conclusion

The e-commerce database has been designed with scalability, flexibility, and integrity in mind. Through the implementation of well-defined relationships between entities, such as one-to-many and many-to-many, the structure supports a wide range of business operations including order processing, product management, and customer interactions. The separation of entities into their respective tables, along with the clear definitions of primary and foreign keys, ensures that the data remains consistent and organized.

The CRUD functionality, handled through individual command files within the *Applications* directory, ensures that the system can efficiently manage the creation, retrieval, updating, and deletion of data. The MVC controller structure ties the backend routes to the application logic, ensuring smooth integration between the user interface and the database layer.

Overall, this approach ensures that the database can efficiently handle the operations of the e-commerce platform while maintaining data integrity and enabling future scalability. The design adheres to best practices and provides a robust foundation for both current and future application features.