İZMİR KÂTİP ÇELEBİ ÜNİVERSİTESİ
2010
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

Inventory and Asset Tracking Application

Software Engineering

Term Project

by

Aıda Ulutaş

Y220240040

Advisor: Prof. Dr. Femin Yalçın Küçükbayrak

January, 2024

Inventory and Asset Tracking Application

Abstract

The Inventory and Asset Tracking Application is a user-friendly web tool designed to help organizations keep track of their assets. It simplifies the process of managing and organizing different resources, making it easier for businesses to handle their belongings efficiently. With a straightforward interface and easy-to-use features, the app aims to improve how organizations handle their assets, providing a convenient solution for modern and reliable asset management.

The application employs the .NET and .NET Core frameworks, with a focus on ASP.NET, Entity Framework, and SQLite for backend development and a layered architecture with MVC for frontend development. The project is organized into distinct modules, namely "*DemirbasApp*" and "*DemirbasData*," reflecting the separation of web-related functionality and data-related operations.

"*DemirbasApp*" module encapsulates the controllers, views, and other components responsible for handling user interactions and presenting data to the users. The structure includes models, controllers, views, areas, and configuration files.

The data module encompasses the data context, models, and identity-related functionalities. The underlying database interactions are managed through Entity Framework, with a focus on SQLite as the database provider.

Keywords: inventory and asset, C#, .NET Core, Entity Framework, SQLite.

Envanter ve Demirbaş Uygulaması

Öz

"DemirbasApp", organizasyonların demirbaşları takip etmelerine yardımcı olmak için tasarlanmış kullanıcı dostu bir web aracıdır. Farklı kaynakları yönetme ve düzenleme sürecini basitleştirerek, işletmelerin demirbaşlarını verimli bir şekilde yönetmelerini kolaylaştırır. Basit bir arayüz ve kullanımı kolay özelliklerle, uygulama organizasyonların varlıklarını nasıl yönettiğini geliştirmeyi amaçlar.

"DemirbasApp" projesi, organizasyon bağlamında demirbaşların etkili bir şekilde yönetimi için tasarlanmış kapsamlı bir web uygulamasıdır. Uygulama, backend geliştirmede ASP.NET, Entity Framework ve SQLite'e odaklanan .NET ve .NET Core çerçevelerini kullanırken, frontend geliştirmede katmanlı bir mimari ve MVC'ye odaklanır.Proje, "DemirbasApp" ve "DemirbasData" olmak üzere ayrı modüllere ayrılmıştır, bu da web ile ilgili işlevselliği ve veri ile ilgili işlemleri yansıtmaktadır.

“DemirbasApp” modülü, kullanıcı etkileşimlerini yöneten ve verileri kullanıcılara sunan denetleyicileri, görünümleri ve diğer bileşenleri içerir. Yapı, modelleri, denetleyicileri, görünümleri, alanları ve yapılandırma dosyalarını içerir. Veri modülü, veri bağlamını, modelleri ve kimlikle ilgili işlevselliği içerir. Temel veritabanı etkileşimleri, Entity Framework üzerinden yönetilir ve veritabanı sağlayıcısı olarak SQLite'a odaklanır.

Anahtar Kelimeler: demirbaş, C#, .NET Core, Entity Framework, SQLite.

Acknowledgment

I extend my sincere gratitude to the university professors for their invaluable insights and guidance throughout my academic journey. A special thanks to Professor Femin Yalçın Küçükbayrak for her support and mentorship during the term project. My appreciation also goes to my fellow students for their collaboration and shared experiences.

Last but not least, I express my thanks to my family for their continuous encouragement, understanding and belief in my abilities.

Table of Contents

Abstract i

Öz ii

Acknowledgment iii

List of Abbreviations vi

List of Figures vii

1 Introduction 1

2 Technological Stack 3

2.1 Backend Development 3

2.1.1 C# Programming Language 3

2.1.2 .NET and .NET Core Frameworks 4

2.1.3 ASP.NET Core 4

2.1.4 Entity Framework 5

2.1.5 ORM (Object Relational Mapping) 5

2.1.6 Relational Database 6

2.1.7 SQLite 6

2.1.8 OOP(Object Oriented Programing) 6

2.2 Frontend Development 7

2.2.1 Razor 7

2.2.2 CSS (Cascading Style Sheets) 7

2.2.3 HTML (Hypertext Markup Language) 7

2.2.4 Bootstrap 7

2.2.5 jQuery 8

2.3 Architecture 8

2.3.1 Web Layer 8

2.3.2 Data Layer 8

2.3.3 Layer Architecture with MVC 8

3 Project Structure 10

3.1 Data Layer 10

3.1.1 Models 11

3.1.1.1 BaseEntity Model 12

3.1.1.2 ItemType Model 13

3.1.1.3 Item Model 13

3.1.1.4 Department Model 15

3.1.1.5 Employee Model 15

3.1.1.6 DeliveryHistory Model 16

3.1.2 Data Context 18

3.1.2.1 Constructor 19

3.1.2.2 DbSet Properties 19

3.1.2.3 Model Configuration 19

3.1.3 Entity Framework Identity Provider 20

3.1.4 Migration 20

3.2 Web Layer 21

3.2.1 Controllers 21

3.2.1.1 Constructor 22

3.2.1.2 Home Controller 23

3.2.1.3 ItemType Controller 24

3.2.1.4 Item Controller 31

3.2.1.5 Department Controller... 40

3.2.1.6 Employee Controller.. 43

3.2.1.7 DeliveryHistory Controller.. 47

3.2.2 Areas 48

3.2.3 Program.cs 48

References ....

List of Abbreviations

OOP	Object Oriented Programming
ORM	Object Relational Mapping
MVC	Model-View-Controller
CSS	Cascading Style Sheet
HTML	Hyper Text Markup Language
SQL	Structured Query Language
API	Application Programming Interface
CRUD	Create, Read, Update, Delete

List of Figures

Figure 3.1 Project Structure 9
Figure 3. 2 Base Entity Model 11
Figure 3.3 ItemType Model 12
Figure 3.4 Item Model.. 13
Figure 3.5 Department 14
Figure 3.6 Employee Model 15
Figure 3.7 DeliveryHistory Model 16
Figure 3.8 DataContext 17
Figure 3.9 Controllers 20
Figure 3.10 Example of Constructor and Dependency Injection 21
Figure 3.11 HomeController 22
Figure 3.12 Home/Index Page Final Output 23
Figure 3.13 ItemTypeController Index Action 23
Figure 3.14 ItemTypeController/ Index View 24
Figure 3.15 ItemTypeController /Index/Final Output 25
Figure 3.16 ItemTypeController-/Create Action 26
Figure 3.17 ItemTypeController/Create Action View 26
Figure 3.18 ItemTypeController/Create Action 27
Figure 3.19 ItemTypeController-Update Action 28
Figure 3.20 ItemTypeController/Update View 28

Figure 3.21 ItemTypeController- Update Action Final Output 29

Figure 3.22 ItemTypeController/Delete Action 29

Figure 3.23 ItemController/Index Action 30

Figure 3.24 ItemController- Index Final Output 30

Figure 3.25 ItemController/Create Action 31

Figure 3.26 ItemController/Create Action Final Output 32

Figure 3.27 ItemController/Update Action 33

Figure 3.28 ItemController/Update Action Final Output 33

Figure 3.29 ItemController/UnassignedItemList Action 34

Figure 3.30 ItemController/UnassignedItemList Final Output 35

Figure 3.31 ItemController/AssignedItemList Action 35

Figure 3.32 ItemController/AssignedItemList Action 36

Figure 3.33 ItemController/ AssignItem Action 36

Figure 3.34 ItemController/AssignItem Action Final Output 37

Figure 3.35 ItemController/Assign Action 37

Figure 3.36 ItemController/Unassign Action 38

Figure 3.37 DepartmentController/Actions 40

Figure 3.38 DepartmentController/Index Final Output 41

Figure 3.39 Create Page 41

Figure 3.40 Update Page 41

Figure 3.41 Create Page 41

Figure 3.42 EmployeeController Actions 44

Figure 3.43 Create Final Output 45

Figure 3.44 Update Final Output 45

Figure 3.45 DeliveryHistoryController/Index Action 46

Figure 3.46 DeliveryHistory/Index Final Output 47

Figure 3.47 Program.cs 49

Chapter 1

Introduction

In the environment of the modern and dynamic business world, effective asset management is a crucial element of corporate and organizational success. Monitoring, registering, and optimizing the use of company resources can be challenging and time-consuming. The "Inventory and Asset Tracking Application" is a comprehensive system for handling the assets throughout their lifecycle in a company.

Traditional methods of manual tracking are error-prone and can’t keep pace with fast-paced business environment where high productivity is crucial.

Lack of asset tracking system can result in serious issues such as maintaining inaccurate and out-of-date inventory records. It also provides better inventory control as changes occur, plus it makes easier the accounting period at the end of the fiscal year.

If your records aren’t up-to-date, you may risk financial losses, time loss and wasted resources due to incorrect and incomplete information.

Misleading information can also lead to delays and it can be time-consuming when locating specific assets. The purpose of the "Inventory and Asset Tracking Application" is to enable easy and trustworthy process of registration and tracking of company assets.

It is built in a Layered architecture using Relational Database to provide registration and tracking of company assets.

It also simplifies the onboarding process of new employees and it documents the items allocated to each employee, including dates, devices and additional details.

At the beginning of an employee’s exit process, the IT personnel manages and confirms the return of listed items.

The application stores warranty periods of electronic devices, repair information, previous users and usage duration.

In the further chapters we will look into the details of the "Inventory and Asset Tracking Application," exploring its architecture, functionality, implementation details, and the benefits it brings to organizations striving for excellence in asset management.

Chapter 2

Technological Stack

The "Inventory and Asset Tracking Application" is developed using technological stack that ensures efficiency, scalability and integration. The primary technologies used include the following:

2.1 Backend Development

2.1.1 C# Programming Language

C# is a versatile and powerful Object-Oriented Programming (OOP) language commonly used for building the backend of applications. It offers a clear program structure, facilitating code reuse and reducing development costs. With scalability and easy maintenance, C# serves as a solid foundation for a wide range of projects.

Designed to seamlessly integrate with the .NET framework, C# supports the development of diverse applications, including Windows applications and web-based systems. Its adaptability makes it suitable for both small-scale and large-scale projects.

C# continues to evolve with new features, such as LINQ, a powerful querying tool enhancing the manipulation of collections and databases. Supporting event-driven programming, C# is well-suited for developing graphical user interfaces (GUIs) and responsive applications.[1]

[1] Microsoft, "A Tour of C# Language", https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

It is versatile and can be used for various application types, including mobile applications, desktop applications, web applications, web sites, games, database applications.

2.1.2 .NET and .NET Core Frameworks

The application benefits from .NET and .NET Core Framework and comprehensive tools and libraries, which provides an environment that supports cross-platform deployment, scalability, security and integration. .NET Core enables better maintainability, updating and extending without affecting the entire system. On the other hand, .NET enables integration with other Microsoft technologies.

The .NET ecosystem provides a rich set of libraries, tools, and frameworks that facilitate rapid development. This includes support for database interactions, web development (ASP.NET), and various other functionalities that are crucial for an asset tracking application.[2]

2.1.3 ASP.NET Core

ASP.NET Core is a cross-platform, open-source framework, allowing developers to build applications that run on Windows, Linux, and macOS. ASP.NET Core embraces the MVC architectural pattern for building web applications (MVC: Models, Views, Controllers).

ASP.NET Core unifies the MVC and Web API frameworks, simplifying the development of both web pages and web APIs within the same application. ASP.NET Core integrates with Entity Framework Core, a lightweight and extensible version of Entity Framework, for data access and database interactions. Besides MVC, ASP.NET Core introduces Razor Pages, feature in ASP.NET used to create dynamic web pages with C# programming.[3]

[2] Radix, ".NET Core vs .NET Framework", https://radixweb.com/blog/net-core-vs-net-framework
[3] Microsoft, "Overview of ASP.NET Core", https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0

2.1.4 Entity Framework

Entity Framework is an open-source ORM framework for .NET applications. It simplifies database interaction, data storing and retrieving and allows automated way to store and access data. It is an ORM framework that facilitates the mapping of object-oriented domain models to a relational database, eliminating the need for manual SQL queries.

Entity Framework Core supports both Code-First (defining entities in code and generating the database schema) and Database-First (reverse engineering entities from an existing database) approaches. Entity Framework Core includes a migration system that simplifies the process of evolving the database schema over time, making it easier to manage changes in the application.

It supports a variety of relational database providers, including SQL Server, PostgreSQL, MySQL, SQLite, and more. [4]

2.1.5 ORM (Object Relational Mapping)

ORM is a technique that connects object-oriented programming (OOP) to relational databases. It creates a structured map of objects and how they are related to different tables. With ORM, programmers can interact with and manipulate objects without the need to worry about the specifics of how these objects are related to their data origins.

When the application modifies data objects, ORM automatically generates the necessary SQL code for the relational database to handle data changes. ORM manages the mapping details between objects and databases, hiding changing interfaces from developers. ORM allows for incorporating new technologies and capabilities without requiring changes to the application code.[5]

[4] C# Corner, "Entity Framework Using C#", https://www.c-sharpcorner.com/article/entity-framework-introduction-using-c-sharp-part-one/
[5] The Server Side, "Object-Relational Mapping", https://www.theserverside.com/definition/object-relational-mapping-ORM

2.1.6 Relational Database

A relational database is a type of database that uses tables to organize and store data. In a relational database, data is organized into rows and columns, where each row represents an individual record with a unique ID called the key, and each column represents a specific attribute or field of the record.

SQL is a language used to interact with relational databases. It allows performing operations like querying, updating, and managing the data. [6]

2.1.7 SQLite

SQLite is an embedded, server-less relational database management system known for its simplicity and efficiency. It operates directly on ordinary disk files, making it lightweight and easy to manage. It is open-source, cross-platform, and does not require a separate server process. SQLite supports parallel work on multiple databases, and its commands are similar to standard SQL.[7]

2.1.7 OOP (Object- Oriented Programing)

Object-Oriented Programming (OOP) is a computer programming model that revolves around the concept of "objects." In object-oriented programming, classes serve as blueprints, defining attributes and methods. Objects are instances of these classes, representing specific data, while methods encapsulate object behaviors, attributes store the object's state. There are four pillars of OOP: encapsulation, inheritance, and polymorphism. Object-oriented programming (OOP) offers several benefits, including: modularity, code reusability, maintenance, flexibility and scalability and more. [8]

[6] Oracle, "What is a Relational Database", https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables
[7] SimpliLearn, "What is SQLite", https://www.simplilearn.com/tutorials/sql-tutorial/what-is-sqlite#:~:text=SQLite%20is%20an%20embedded%2C%20server,than%20other%20database%20management%20systems
[8] Tech Target, "Object-Oriented Programing", https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

2.2 Frontend Development

Frontend development in the "Inventory and Asset Tracking Application" involves the use of several technologies and tools to create a user-friendly and responsive interface. Here's a brief overview of each component.

2.2.1 Razor

Razor is a view engine used with ASP.NET to create dynamic web pages. It allows embedding C# code within HTML markup. [9]

2.2.2 CSS (Cascading Style Sheets)

CSS is a styling language that controls the presentation of HTML elements. It is used to enhance the visual appearance of the application, providing layout and design.[10]

2.2.3 HTML (Hypertext Markup Language)

HTML is the standard markup language for creating web pages. It structures the content on the frontend, defining elements such as headings, paragraphs, and images.[11]

2.2.4 Bootstrap

Bootstrap is the most popular CSS Framework. It includes pre-designed components and styles, making it easier to create a consistent and visually appealing UI.[12]

[9] Code Academy, "What is ASP:NET Razor Pages?", https://www.codecademy.com/article/what-is-asp-net-razor-pages
[10] Tutorials Point, "What is CSS", https://www.tutorialspoint.com/css/what_is_css.htm
[11] W3Schools, "HTML Introduction", https://www.w3schools.com/html/html_intro.asp
[12] W3Schools, "What is Bootstrap", https://www.w3schools.com/whatis/whatis_bootstrap.asp

2.2.5 jQuery

Query is a lightweight, JavaScript library that simplifies tasks like HTML document traversal, event handling, and animation.[13]

2.3 Architecture

Layered architecture involves breaking down the application into layers, each with different responsibilities and functionality. Adhering to a layered architecture, this application is organized into two separate layers: web layer and data layer, each with specific responsibilities and design.

2.3.1 Web Layer

Web Layer is responsible for handling user interactions and communication with backend. Usually and also in this project web layer include Controllers and Views.

2.3.2 Data Layer

Data Layer is responsible for data storage and interaction with database and it includes Models.

2.3.3 Layered Architecture With MVC

MVC is an architectural pattern that separates application into three logical components with different responsibilities (business, presentation and data logic).

1. Model (data logic)
   Model is directly connected to database and it handles adding and retrieving data. It is a bridge between Controller and database, it listens to Controller's instructions

[13] W3Schools, "jQuery Introduction", https://www.w3schools.com/jquery/jquery_intro.asp

and it structures data in the requested form. When a request comes from the Controller the Model interacts with database and responses back to the Controller.

2. View (presentation logic)

   View component controls data presentation (HTML, CSS) to users. It is the frontend of the User Interface (UI). It presents data based on the User's action. It represents the current Model state but it never communicates with the Model, it sends data to the Controller and receives data from the Controller.
3. Controller (business logic)

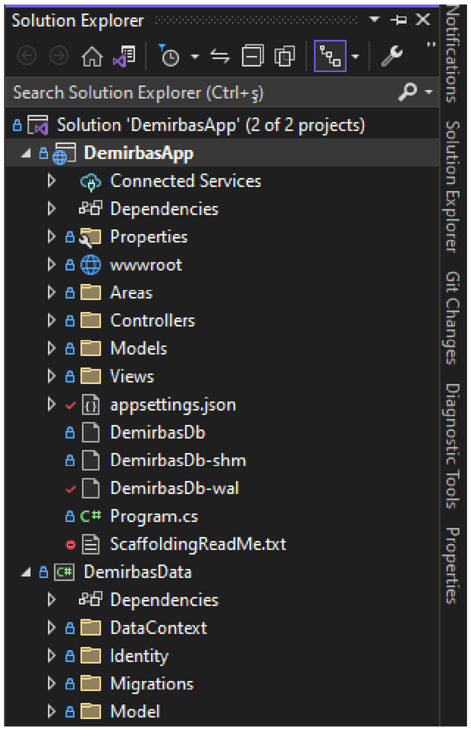
   Controller is the brain component, it handles the communication between the Model and View. It takes commands from the User through the View layer and sends them to the Model where the request is processed. Then it collects data from the Model and sends it to the View. [14]

[14] GeeksFofGeeks,"MVC Framework Intoruction", https://www.geeksforgeeks.org/mvc-framework-introduction/

Chapter 3

Project Structure

3.1 Data Layer



```
Solution 'DemirbasApp' (2 of 2 projects)
◢ DemirbasApp
    ▷ Connected Services
    ▷ Dependencies
    ▷ Properties
    ▷ wwwroot
    ▷ Areas
    ▷ Controllers
    ▷ Models
    ▷ Views
    ▷ appsettings.json
      DemirbasDb
      DemirbasDb-shm
      DemirbasDb-wal
      Program.cs
      ScaffoldingReadMe.txt
◢ DemirbasData
    ▷ Dependencies
    ▷ DataContext
    ▷ Identity
    ▷ Migrations
    ▷ Model
```

Figure 3.1: Project Structure

The data layer is responsible for the management of data, interacting with the database, and handling data-related operations. The web layer interacts with the data layer by calling methods in the controllers, which, in turn, interact with the *DataContext* to perform CRUD (Create, Read, Update, Delete) operations on the database.

This separation of responsibilities between the web and data layers follows the principles of a typical MVC (Model-View-Controller) architecture, promoting modularity, maintainability, and scalability in your application.

Components of Data Layer:

1. Models

Models define the structure of the application's data. They represent entities and relationships, serving as the blueprint for database tables.

2. DataContext

The DataContext acts as a bridge between application and the database. It includes DbSet properties for each model and configurations for data access.

3. Migrations

Migrations represent the versions of database schema and they are used to manage changes of the database schema over time. They allow evolving of the database structure with each change in the data model.

4. Identity

If an application involves user authentication and authorization, it needs to have an Identity folder containing user-related entities, configurations, and services.

3.1.1 Models

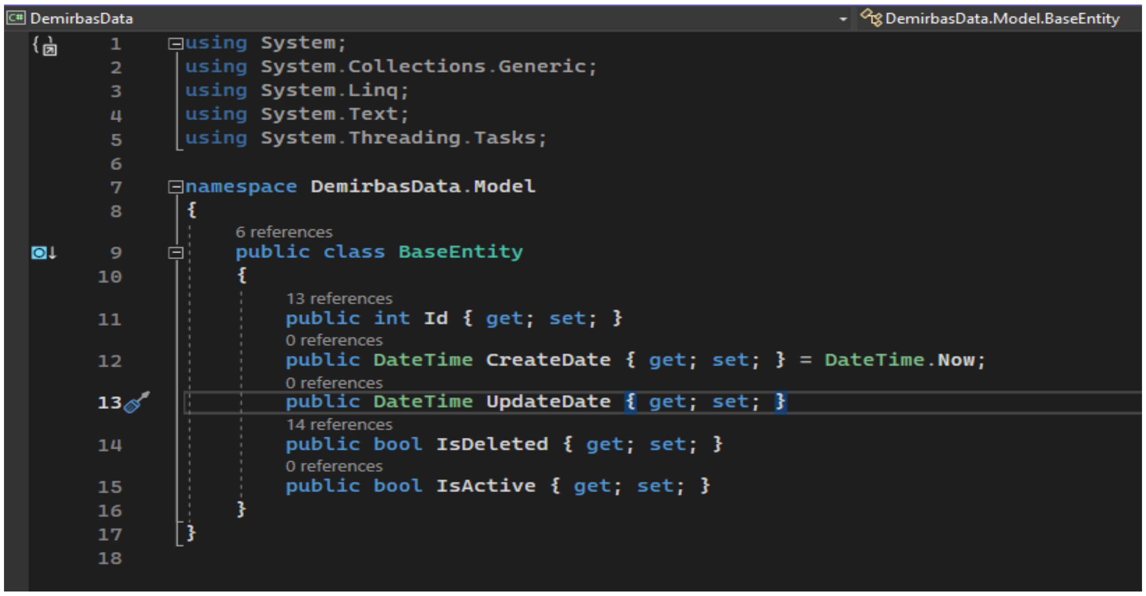
Located in *DemirbasApp/DemirbasData/Model* folder (see Figure 3.1) this section contains the data models representing entities within the application. These models define the structure and relationships of data.

3.1.1.1 BaseEntity Model

BaseEntity Model serves as a foundation model for objects in the application. It encapsulates common attributes shared by multiple entities thus reducing repetition, promoting consistency, useability and easy maintainability. Other entities within the application inherit the common properties of this model, which allows them to focus on their unique entity properties.

Properties of BaseEntity Model (See Figure 3.2)

- “Id” of *int* type serves as a unique identifier.
- “Create Date” of *DateTime* type represents date and time when the entity was initially created.
- “Update Date” keeps track of date and time of the entity’s last update.
- “IsDeleted” of *bool* type indicates if the entity has been marked as deleted. It supports soft deletion without permanent removement from the database.
- “IsActive” of *bool* type refers to entity’s current state in the system.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    6 references
    public class BaseEntity
    {
        13 references
        public int Id { get; set; }
        0 references
        public DateTime CreateDate { get; set; } = DateTime.Now;
        0 references
        public DateTime UpdateDate { get; set; }
        14 references
        public bool IsDeleted { get; set; }
        0 references
        public bool IsActive { get; set; }
    }
}
```

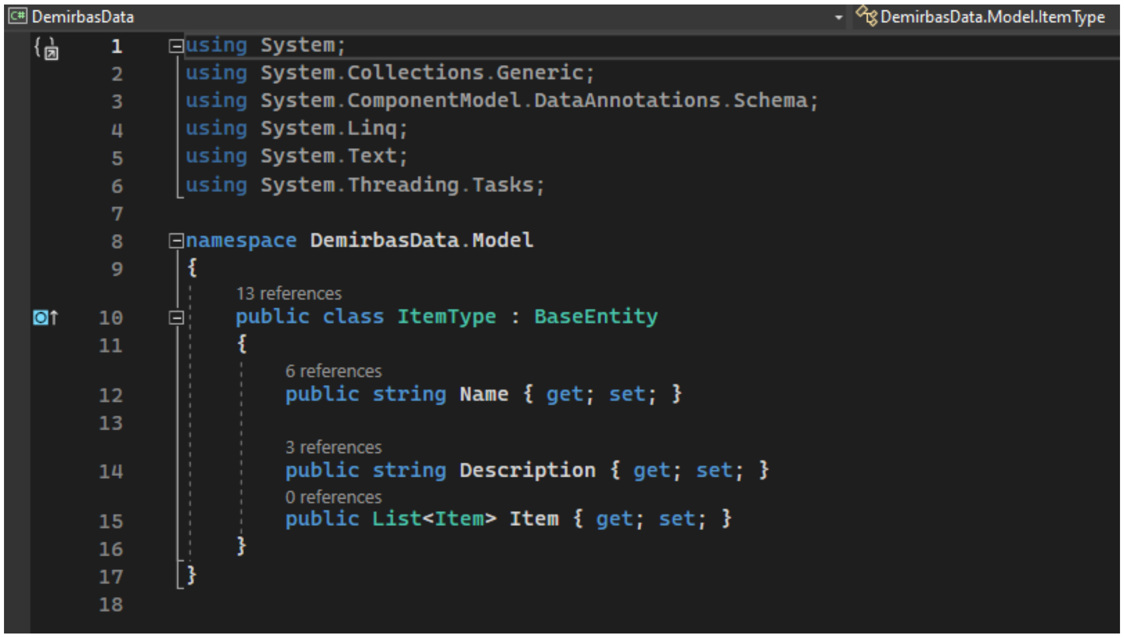
Figure 3. 2: Base Entity Model

3.1.1.2 ItemType Model

ItemType Model represents a category in the application facilitating the organization and classification of items. It extends the “BaseEntity” inheriting attributes for identification and tracking.

Properties of ItemType Model (See Figure 3.3)

- “Name” of *string* type represents the name of the item type.
- “Description” of *string* type giving additional details about the type.
- “Item” of ‘*List<Item>*’ type meaning it has a one-to-many relationship with “Item” model and it represents a collection of items associated with item type.



```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    13 references
    public class ItemType : BaseEntity
    {
        6 references
        public string Name { get; set; }

        3 references
        public string Description { get; set; }
        0 references
        public List<Item> Item { get; set; }
    }
}
```

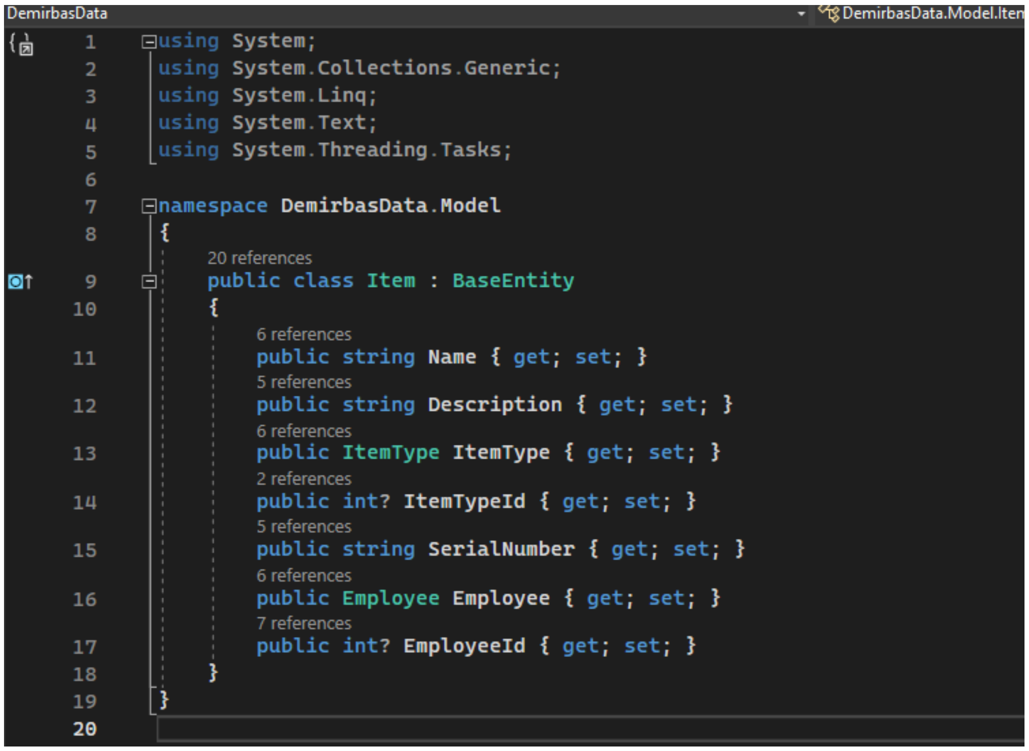
Figure 3.3: ItemType Model

3.1.1.3 Item Model

Item Model represents the individual item within the application. It captures essential details such as name, description, category (via ItemType), serial number, and the employee assigned to an item. It inherits common attributes from the *BaseEntitiy*.

Properties of Item model See Figure 3. 4) :

- “Name” represents the name or title of the item. Type: string.
- “Description” of *string* type, offers additional details about the item.
- ItemType: Type: ItemType. This property establishes a relationship with the "ItemType" model, representing the category to which the item belongs.
- “ItemTypeId” of *int* type, stores the identifier of the associated item type, establishing a link between the item and its category.
- “SerialNumber” of *string* type represents the serial number assigned to the item.
- “Employee”: Type: Employee. It establishes a relationship with the "Employee" model, representing the individual assigned to the item.
- “EmployeeId”: Type: int?. It stores the identifier of the associated employee, linking the item to its responsible individual.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    20 references
    public class Item : BaseEntity
    {
        6 references
        public string Name { get; set; }
        5 references
        public string Description { get; set; }
        6 references
        public ItemType ItemType { get; set; }
        2 references
        public int? ItemTypeId { get; set; }
        5 references
        public string SerialNumber { get; set; }
        6 references
        public Employee Employee { get; set; }
        7 references
        public int? EmployeeId { get; set; }
    }
}
```

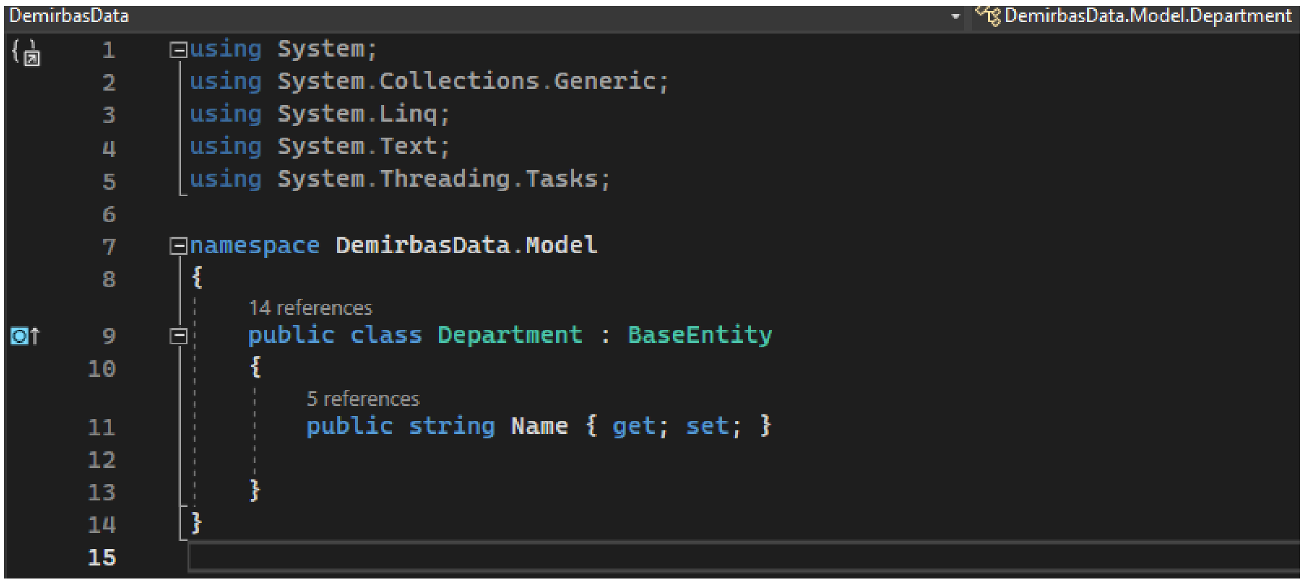
Figure 3.4: Item Model

3.1.1.4 Department Model

Department Model represents a department or organizational unit within the application. Extending the *BaseEntity* model, it inherits common attributes.

Properties of Department Model (See Figure 3. 5)

- “Name” : type: string, represents the name or title of the department.



```
DemirbasData                                          DemirbasData.Model.Department
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    14 references
    public class Department : BaseEntity
    {
        5 references
        public string Name { get; set; }

    }
}
```

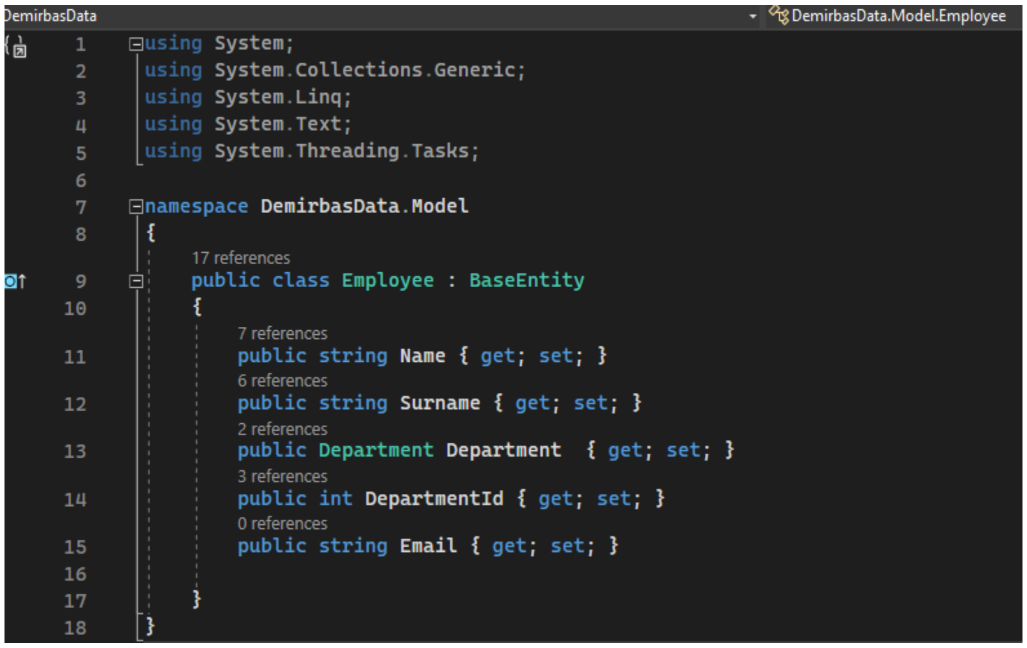
Figure 3.5: Department Model

3.1.1.5 Employee Model

Employee Model represents an individual employee within the application. Extending the *BaseEntity* model, it inherits common attributes for identification, timestamps, and status tracking. Properties of Employee Model (See Figure 3.6):

- “Name” of *string* type represents the first name of the employee.
- “Surname” of *string* type represents the last name or surname of the employee.
- “Department”: type: Department. It establishes a relationship with the "Department" model, representing the department to which the employee belongs.

- "DepartmentId" of *int* type stores the identifier of the associated department, linking the employee to their respective department.
- "Email" of *string* type represents the email address of the employee.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    17 references
    public class Employee : BaseEntity
    {
        7 references
        public string Name { get; set; }
        6 references
        public string Surname { get; set; }
        2 references
        public Department Department  { get; set; }
        3 references
        public int DepartmentId { get; set; }
        0 references
        public string Email { get; set; }

    }
}
```

Figure 3.6: Employee Model

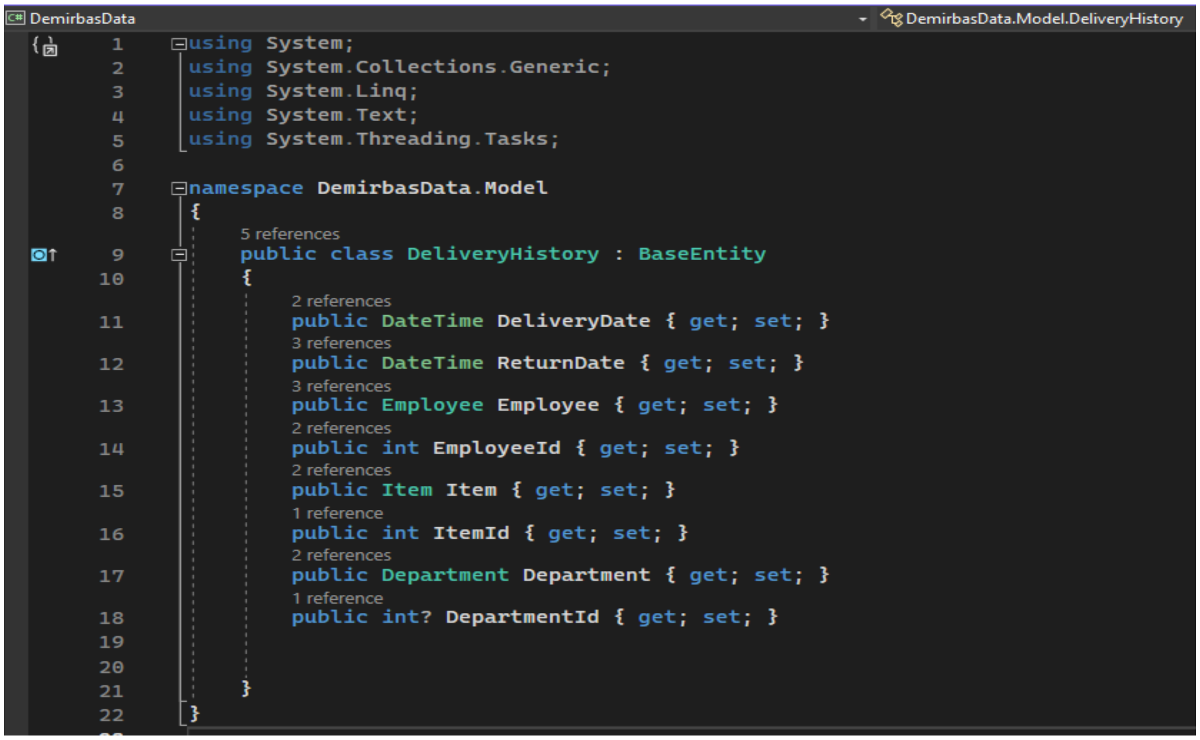
3.1.1.6 DeliveryHistory Model

The DeliveryHistory model represents the history of item deliveries and returns within the application. Extending the BaseEntity model, it inherits common attributes for identification, timestamps, and status tracking.

Properties of DeliveryHistory Model (See Figure 3.7):

- "DeliveryDate": type: DateTime, represents the date and time when the item was delivered.
- "ReturnDate": type: DateTime represents the date and time when the delivered item was returned.

- “Employee”: Type: Employee, establishes a relationship with the "Employee" model, representing the employee involved in the delivery and return.
- “EmployeeId”: type: int, stores the identifier of the associated employee, linking the delivery history to the involved employee.
- “Item”: type: Item, establishes a relationship with the "Item" model, representing the item that was delivered and returned.
- “ItemId”: type: int, stores the identifier of the associated item, linking the delivery history to the item.
- “Department”: type: Department, establishes a relationship with the "Department" model, representing the department associated with the delivery history.
- “DepartmentId”: type: int?, stores the identifier of the associated department, providing additional context to the delivery history record.

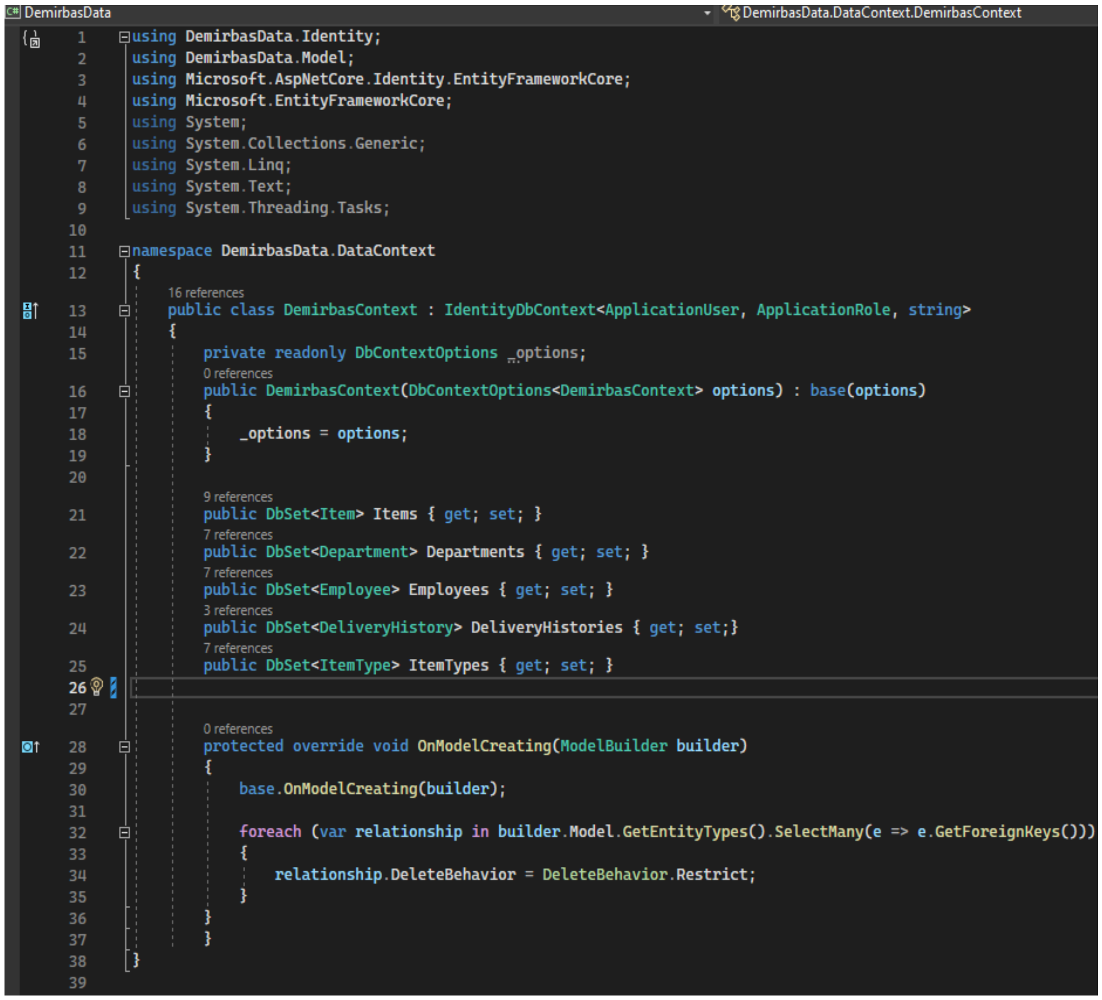


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.Model
{
    5 references
    public class DeliveryHistory : BaseEntity
    {
        2 references
        public DateTime DeliveryDate { get; set; }
        3 references
        public DateTime ReturnDate { get; set; }
        3 references
        public Employee Employee { get; set; }
        2 references
        public int EmployeeId { get; set; }
        2 references
        public Item Item { get; set; }
        1 reference
        public int ItemId { get; set; }
        2 references
        public Department Department { get; set; }
        1 reference
        public int? DepartmentId { get; set; }


    }
}
```

Figure 3.7: DeliveryHistory Model

3.1.2 Data Context

The Data module plays a crucial role in managing the persistence layer of the application. At its core is the *DataContext* (See Figure 3.8), which acts as the bridge between the application and the underlying database. The *DataContext* orchestrates data access, entity relationships, and database operations, making it a critical component of the application's data layer.It extends the *IdentityDbContext*, integrating Identity functionality for user and role management into the core data operations.It inherits from the Entity Framework's *DbContext* class, providing a powerful framework for interacting with the database.[15]



```
using DemirbasData.Identity;
using DemirbasData.Model;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemirbasData.DataContext
{
    16 references
    public class DemirbasContext : IdentityDbContext<ApplicationUser, ApplicationRole, string>
    {
        private readonly DbContextOptions _options;
        0 references
        public DemirbasContext(DbContextOptions<DemirbasContext> options) : base(options)
        {
            _options = options;
        }

        9 references
        public DbSet<Item> Items { get; set; }
        7 references
        public DbSet<Department> Departments { get; set; }
        7 references
        public DbSet<Employee> Employees { get; set; }
        3 references
        public DbSet<DeliveryHistory> DeliveryHistories { get; set;}
        7 references
        public DbSet<ItemType> ItemTypes { get; set; }


        0 references
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);

            foreach (var relationship in builder.Model.GetEntityTypes().SelectMany(e => e.GetForeignKeys()))
            {
                relationship.DeleteBehavior = DeleteBehavior.Restrict;
            }
        }
    }
}
```

Figure 3.8: DataContext

[15] Microsoft, “Get started with EF Core in an ASP:NET MVC web App”, https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-8.0

3.1.2.1. Constructor

A constructor is a special method that is used to initialize objects (See Figure 3.8, lines: 15-19). It is called when an object of a class is created.

The constructor of *DemirbasContext* initializes the context with the provided *DbContextOptions*. This is crucial for establishing the connection to the database.

3.1.2.2 DbSet Properties

A DbSet (See Figure 3.8, lines: 21-25) represents the collection of all entities in the context, or that can be queried from the database, of a given type. *DbSet* objects are created from a *DbContext* using the *DbContext.Set* method.

The *DbSet* properties represent the entities mapped to database tables. Each property corresponds to a table in the database, facilitating data access and manipulation.

3.1.2.3 Model Configuration

The *OnModelCreating* (See Figure 3.8, lines: 28-35) method is overridden to provide additional configuration for the data model. In this case, it ensures that cascade delete behavior is restricted for all relationships, preventing unintended data loss.

When creating a model, there may be additional features we want in the database. For example, if we want to create an index for the 'createdate' column in the 'Item' table and perform searches based on the date, or if we want it to come in a sorted order, we define these features here.

3.1.3 Entity Framework Identity Provider

Entity Framework Identity is a feature of Entity Framework that extends its capabilities to include user and role management. By integrating *EFIdentity*, you gain the ability to perform authentication and authorization checks easily. It supports features like user login and logout.

EF Identity includes built-in security features, such as account lockout, two-factor authentication, and token-based authentication. These features enhance the security of user accounts.[16]

3.1.4 Migration

Migrations in this project serve the purpose of managing and versioning the database schema. They enable developers to evolve the database structure over time, reflecting changes in the application's data model.

Migrations are created to capture changes in the data model, such as adding new entities, modifying existing ones, or altering relationships.

Also, migrations are applied to update the database schema.

Migrations are named descriptively to reflect the changes they introduce. Meaningful names provide clarity on the purpose of each migration.

[16] Microsoft, "Introduction to Identity on ASP:NET Core", https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio

3.2 Web Layer:

The web layer is the part of your application that handles user interactions, presentation, and communication with users. It typically consists of controllers, views, and any client-side. Components of Web Layer:

- Controllers handle incoming HTTP requests, process the input, and interact with the data layer to retrieve or modify data. They orchestrate the flow of the application.
- Views represent the user interface and display data to users. They receive input from controllers, presenting information in a format suitable for a user.
- In the Area section of the application, we can access and manage various account-related features. This includes actions such as logging in, logging out, and other account management functionalities such as password change and security.
- “appsettings.json” is a configuration file containing settings that can be accessed by the web layer. It's used to configure various aspects of the application.
- Program.cs is the entry point for your application. It configures and starts the application.

3.2.1 Controllers

Six controllers in this ASP.NET application handle user interactions, data flow, and application logic effectively (See Figure 3.9).

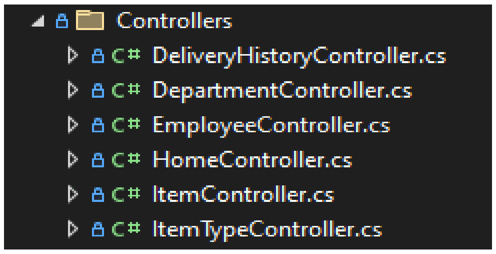


Figure 3.9: Controllers

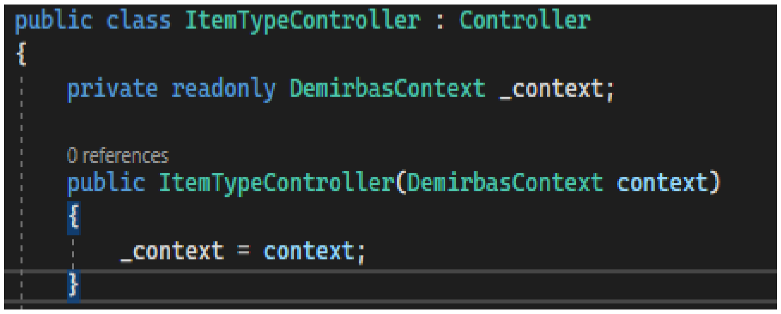
3.2.1.1 Constructor

The constructor, explained once here, plays a crucial role and will be implemented in every controller to initialize essential components.

A constructor in C# is a special method that gets executed when an instance of a class is created. The purpose of the constructor is to receive an instance of Context through dependency injection. This enables the controller to interact with the underlying database when handling requests related to a controller.

Dependency injection is a design pattern where dependencies of a class are provided externally rather than being created within the class itself. In this case, the *DemirbasContext* dependency is injected into the controller.

The *ItemTypeController* constructor initializes an instance of the controller and establishes a connection to the data layer through the *DemirbasContext*. This constructor follows the dependency injection pattern, taking a *DemirbasContext* parameter (See Figure 3.10).[17]



```
public class ItemTypeController : Controller
{
    private readonly DemirbasContext _context;

    0 references
    public ItemTypeController(DemirbasContext context)
    {
        _context = context;
    }
```

Figure 3.10: Example of Constructor and Dependency Injection

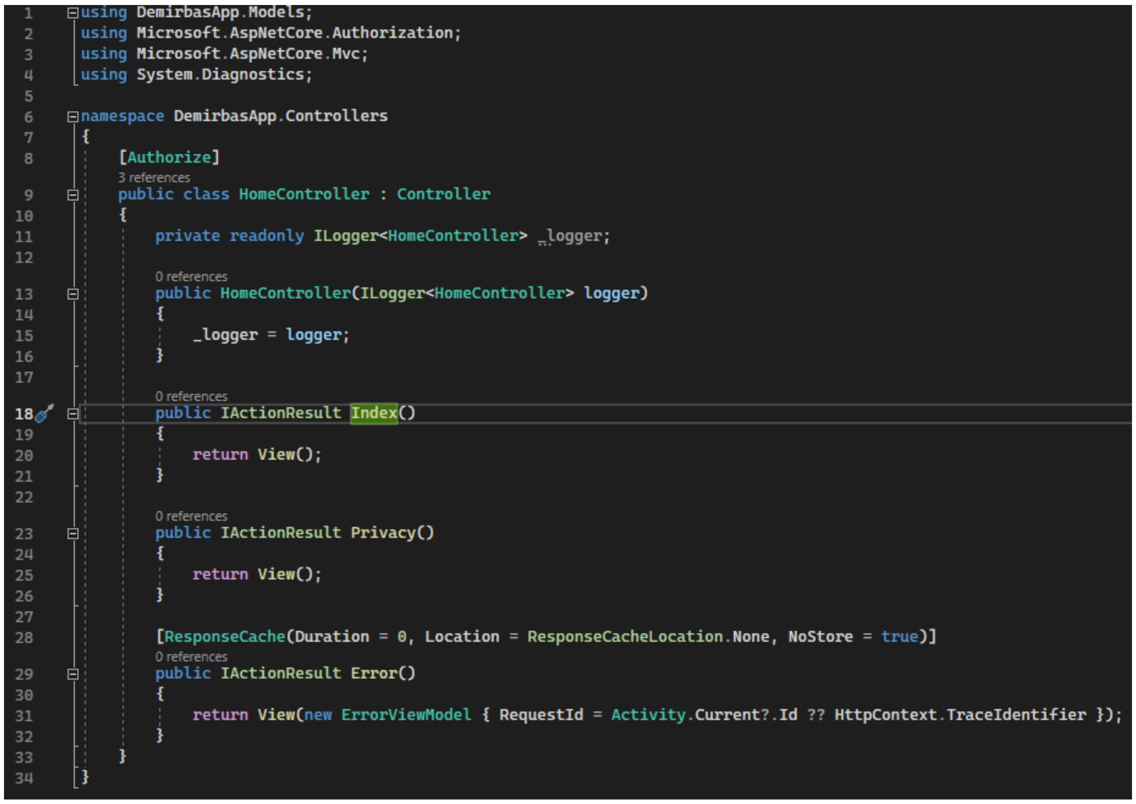
[17] Microsoft, "Constructors (C# Programing Guide), https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors

3.2.1.2 HomeController

The HomeController serves as the entry point for your application's web layer. It is responsible for handling requests related to the home page, privacy page, and error views. The controller contains the [Authorize] attribute, indicating that access to its actions requires authentication.

HomeController Actions Overview (See Figure 3.11)

- Index: handles requests for the home page. It returns a view representing the main landing page of the application.
- Privacy: handles requests for the privacy page. It returns a view representing the privacy policy or related information.
- Error: is responsible for rendering error views.



```csharp
using DemirbasApp.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;

namespace DemirbasApp.Controllers
{
    [Authorize]
    3 references
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;

        0 references
        public HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
        }

        0 references
        public IActionResult Index()
        {
            return View();
        }

        0 references
        public IActionResult Privacy()
        {
            return View();
        }

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

See Figure 3.11: HomeController

Rendering of Home/Index Page (See Figure 3.12)

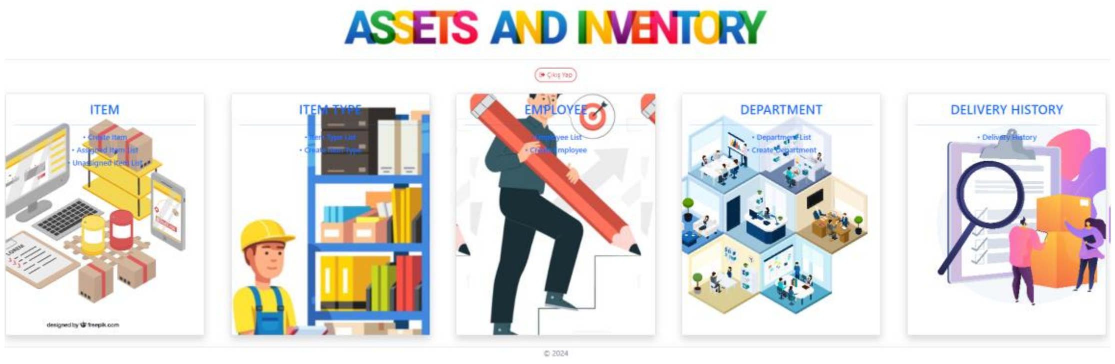


Figure 3.12: Home/Index Page Final Output

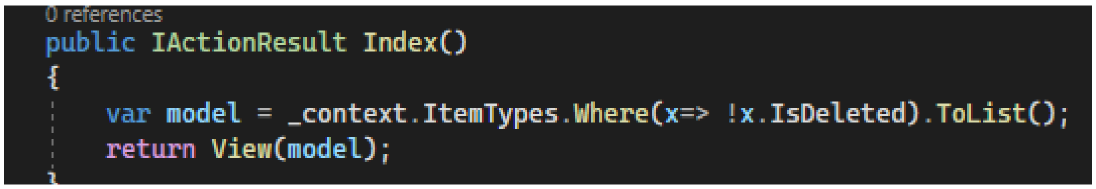
3.2.1.3 ItemTypeController

The *ItemTypeController* is responsible for handling requests related to item types in your application. It interacts with the *DemirbasContext* to perform CRUD operations on the ItemType entity. In the following sections of ItemTypeController, I will provide detailed Views for the Create and Update actions. Please note that the Views for other controllers will be represented more generally, as they share the same code structure.

ItemTypeController Actions Overview:

Index Action (See Figure 3.13)

The Index action retrieves a list of non-deleted item types from the database and passes the result to the corresponding view.



```
0 references
public IActionResult Index()
{
    var model = _context.ItemTypes.Where(x=> !x.IsDeleted).ToList();
    return View(model);
}
```

Figure 3.13: ItemTypeController Index Action

ItemTypeController - Index View (Figure 3.14)

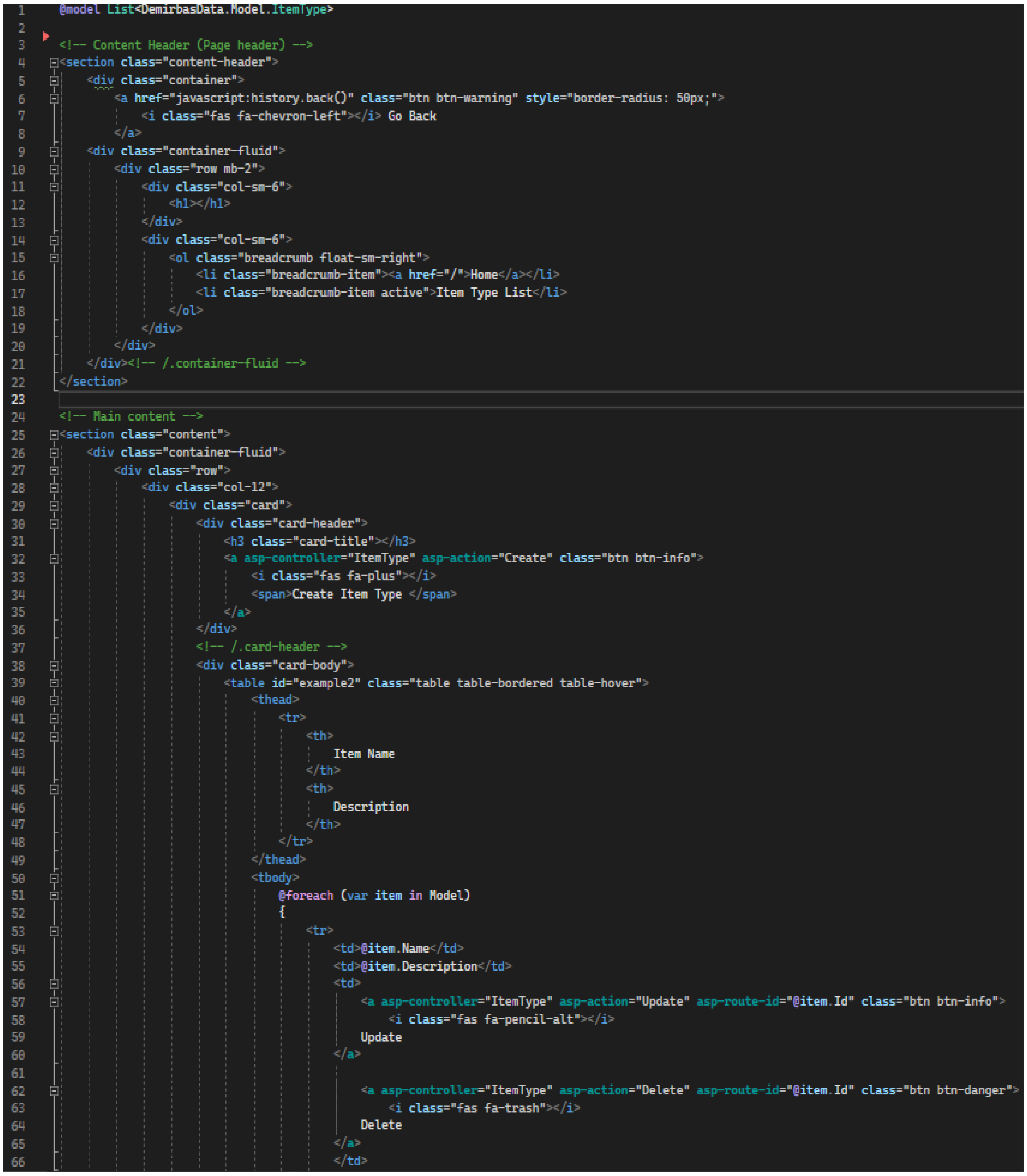
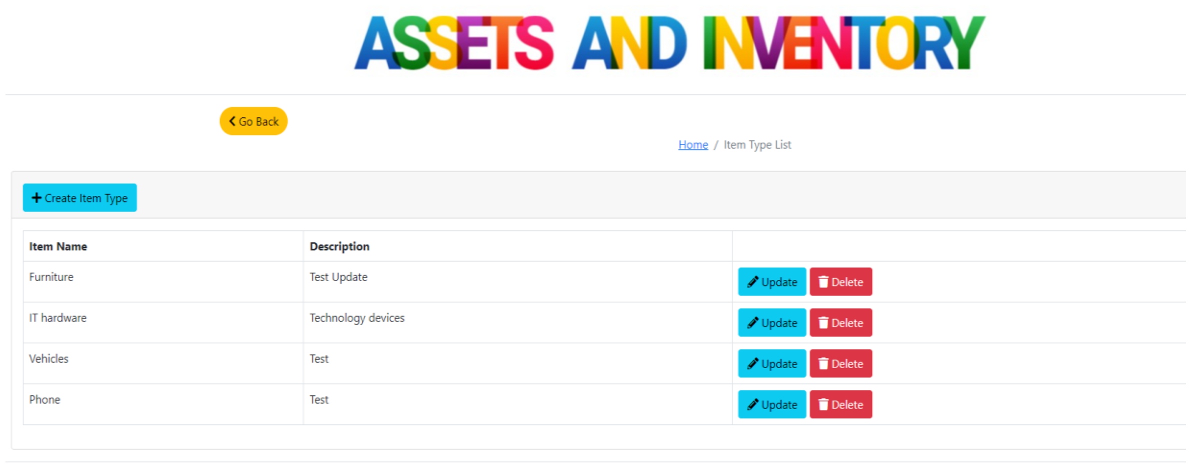

```
@model List<DemirbasData.Model.ItemType>

<!-- Content Header (Page header) -->
<section class="content-header">
    <div class="container">
        <a href="javascript:history.back()" class="btn btn-warning" style="border-radius: 50px;">
            <i class="fas fa-chevron-left"></i> Go Back
        </a>
    <div class="container-fluid">
        <div class="row mb-2">
            <div class="col-sm-6">
                <h1></h1>
            </div>
            <div class="col-sm-6">
                <ol class="breadcrumb float-sm-right">
                    <li class="breadcrumb-item"><a href="/">Home</a></li>
                    <li class="breadcrumb-item active">Item Type List</li>
                </ol>
            </div>
        </div>
    </div><!-- /.container-fluid -->
</section>

<!-- Main content -->
<section class="content">
    <div class="container-fluid">
        <div class="row">
            <div class="col-12">
                <div class="card">
                    <div class="card-header">
                        <h3 class="card-title"></h3>
                        <a asp-controller="ItemType" asp-action="Create" class="btn btn-info">
                            <i class="fas fa-plus"></i>
                            <span>Create Item Type </span>
                        </a>
                    </div>
                    <!-- /.card-header -->
                    <div class="card-body">
                        <table id="example2" class="table table-bordered table-hover">
                            <thead>
                                <tr>
                                    <th>
                                        Item Name
                                    </th>
                                    <th>
                                        Description
                                    </th>
                                </tr>
                            </thead>
                            <tbody>
                                @foreach (var item in Model)
                                {
                                    <tr>
                                        <td>@item.Name</td>
                                        <td>@item.Description</td>
                                        <td>
                                            <a asp-controller="ItemType" asp-action="Update" asp-route-id="@item.Id" class="btn btn-info">
                                                <i class="fas fa-pencil-alt"></i>
                                                Update
                                            </a>

                                            <a asp-controller="ItemType" asp-action="Delete" asp-route-id="@item.Id" class="btn btn-danger">
                                                <i class="fas fa-trash"></i>
                                                Delete
                                            </a>
                                        </td>
```

Figure 3.14: ItemTypeController/ Index View

Rendering of ItemTypeController/Index (Figure 3.15)



Item Name	Description	
Furniture	Test Update	Update Delete
IT hardware	Technology devices	Update Delete
Vehicles	Test	Update Delete
Phone	Test	Update Delete

Figure 3.15: ItemTypeController /Index/Final Output

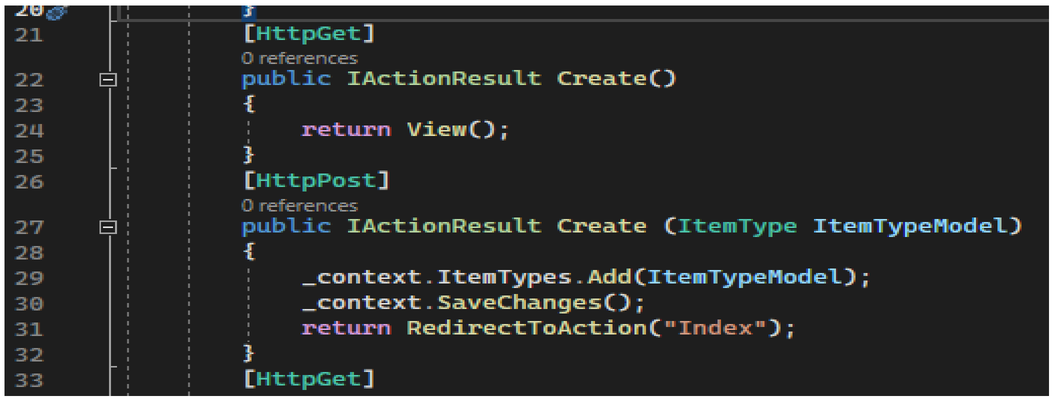
ItemTypeController- Create Action (See Figure 3.16)

HTTP GET is responsible for handling HTTP GET requests to display the form for creating a new item type. Explanation:

- When a user navigates to the "Create Item Type" page, this action is triggered.
- It returns the associated view (Create.cshtml), allowing users to input information for a new item type.

HTTP POST is responsible for handling HTTP POST requests to process the form submission and create a new item type. Explanation:

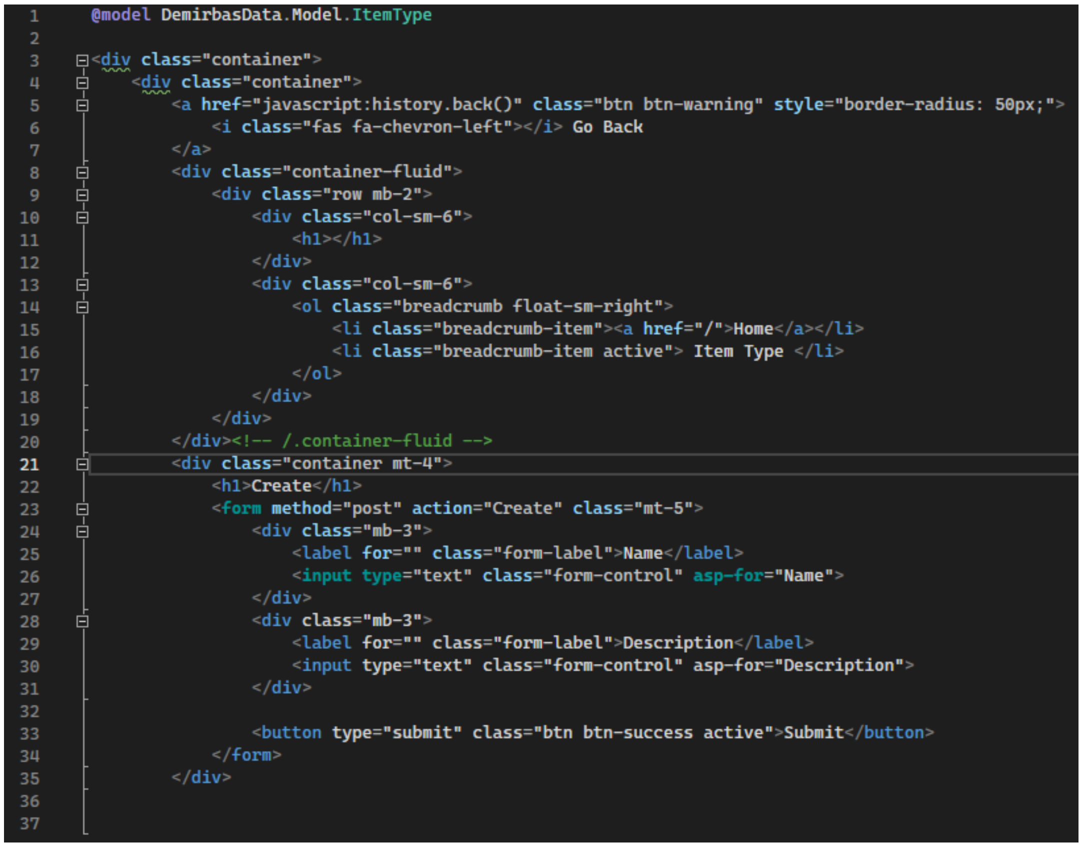
- After a user submits the "Create Item Type" form, this action is triggered.
- It receives the form data as a parameter (ItemType ItemTypeModel).
- The new ItemTypeModel is added to the ItemTypes collection in the database context (_context).
- The changes are saved to the database using *_context.SaveChanges()*.
- The action then redirects the user to the "Item Types" index page (Index action).



```
        }
        [HttpGet]
        0 references
        public IActionResult Create()
        {
            return View();
        }
        [HttpPost]
        0 references
        public IActionResult Create (ItemType ItemTypeModel)
        {
            _context.ItemTypes.Add(ItemTypeModel);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
        [HttpGet]
```

Figure 3.16: ItemTypeController-/Create Action

Bellow you can see ItemTypeController- Create Action View (Figure 3.17)



```
@model DemirbasData.Model.ItemType

<div class="container">
    <div class="container">
        <a href="javascript:history.back()" class="btn btn-warning" style="border-radius: 50px;">
            <i class="fas fa-chevron-left"></i> Go Back
        </a>
        <div class="container-fluid">
            <div class="row mb-2">
                <div class="col-sm-6">
                    <h1></h1>
                </div>
                <div class="col-sm-6">
                    <ol class="breadcrumb float-sm-right">
                        <li class="breadcrumb-item"><a href="/">Home</a></li>
                        <li class="breadcrumb-item active"> Item Type </li>
                    </ol>
                </div>
            </div>
        </div><!-- /.container-fluid -->
        <div class="container mt-4">
            <h1>Create</h1>
            <form method="post" action="Create" class="mt-5">
                <div class="mb-3">
                    <label for="" class="form-label">Name</label>
                    <input type="text" class="form-control" asp-for="Name">
                </div>
                <div class="mb-3">
                    <label for="" class="form-label">Description</label>
                    <input type="text" class="form-control" asp-for="Description">
                </div>

                <button type="submit" class="btn btn-success active">Submit</button>
            </form>
        </div>


```

Figure 3.17: ItemTypeController/Create Action View

ItemTypeController- Rendering Create Action Page

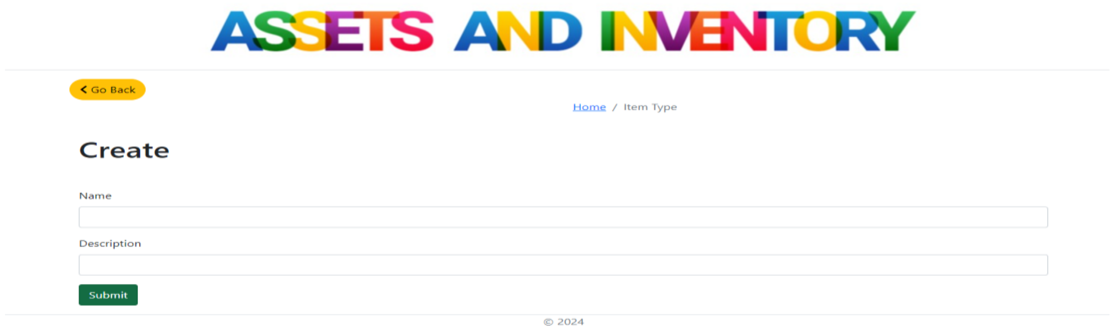


Figure 3.18: ItemTypeController/Create Action

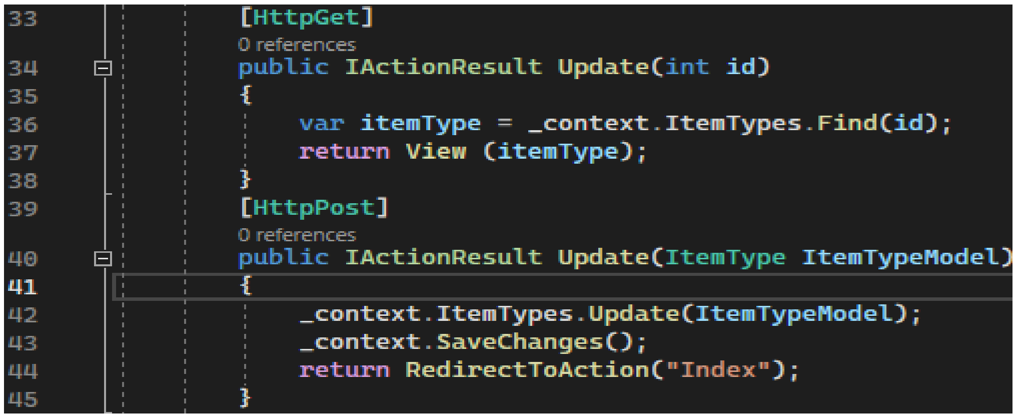
ItemTypeController-Update Action (See Figure 3.19)

HTTP GET handles HTTP GET requests to display the form for updating an existing item type. Explanation:

- When a user navigates to the "Edit Item Type" page, this action is triggered with the *id* parameter representing the ID of the item type to be updated.
- It retrieves the existing item type from the database using *Find(id)* and passes it to the associated view (Update.cshtml).
- Users see a pre-filled form with the details of the selected item type, ready for modification.

HTTP POST handles HTTP POST requests to process the form submission and update an existing item type. Explanation:

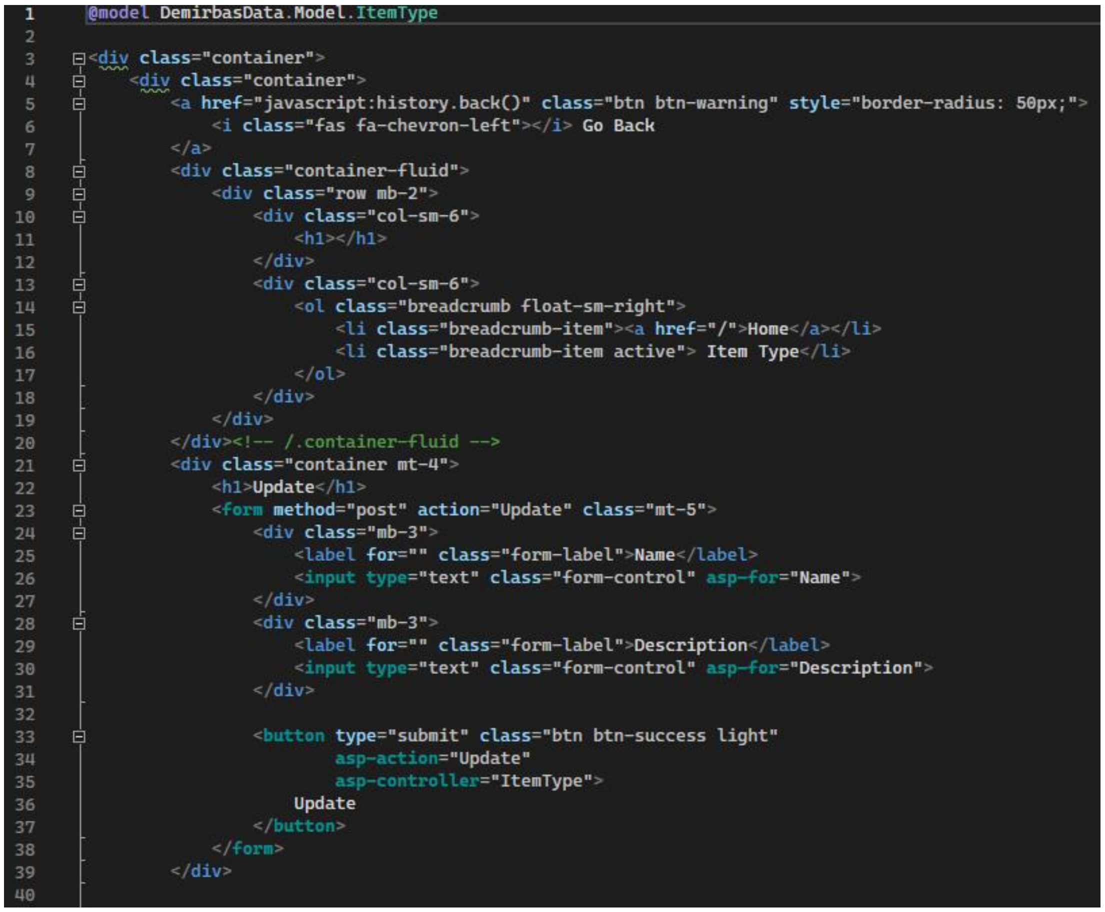
- After a user submits the "Edit Item Type" form, this action is triggered with the updated *ItemTypeModel*.
- It updates the corresponding item type in the *ItemTypes* collection of the database context (*_context*).
- The changes are saved to the database using *_context.SaveChanges()*.
- The action redirects the user to the "*ItemTypes*" index page (Index action) after a successful update.



```
        [HttpGet]
        0 references
        public IActionResult Update(int id)
        {
            var itemType = _context.ItemTypes.Find(id);
            return View (itemType);
        }
        [HttpPost]
        0 references
        public IActionResult Update(ItemType ItemTypeModel)
        {
            _context.ItemTypes.Update(ItemTypeModel);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
```

Figure 3.19: ItemTypeController-Update Action

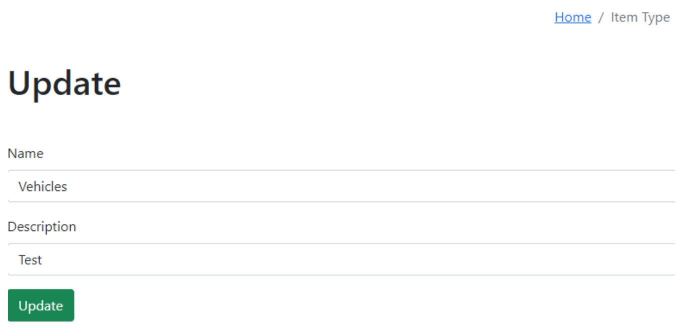
Bellow you can see ItemTypeController-Update Action View (Figure 3.20)



```
@model DemirbasData.Model.ItemType

<div class="container">
    <div class="container">
        <a href="javascript:history.back()" class="btn btn-warning" style="border-radius: 50px;">
            <i class="fas fa-chevron-left"></i> Go Back
        </a>
        <div class="container-fluid">
            <div class="row mb-2">
                <div class="col-sm-6">
                    <h1></h1>
                </div>
                <div class="col-sm-6">
                    <ol class="breadcrumb float-sm-right">
                        <li class="breadcrumb-item"><a href="/">Home</a></li>
                        <li class="breadcrumb-item active"> Item Type</li>
                    </ol>
                </div>
            </div>
        </div><!-- /.container-fluid -->
        <div class="container mt-4">
            <h1>Update</h1>
            <form method="post" action="Update" class="mt-5">
                <div class="mb-3">
                    <label for="" class="form-label">Name</label>
                    <input type="text" class="form-control" asp-for="Name">
                </div>
                <div class="mb-3">
                    <label for="" class="form-label">Description</label>
                    <input type="text" class="form-control" asp-for="Description">
                </div>

                <button type="submit" class="btn btn-success light"
                        asp-action="Update"
                        asp-controller="ItemType">
                    Update
                </button>
            </form>
        </div>

```

Figure 3.20: ItemTypeController/Update View

ItemTypeController- Rendering Update Action Page


Home / Item Type

Update

Name

Vehicles

Description

Test

Figure 3.21: ItemTypeController- Update Action Final Output

ItemTypeController- Delete Action (See Figure 3.22)

HTTP GET handles HTTP GET requests to display a confirmation page for deleting an item type. Explanation:

- When a user navigates to the "Delete Item Type" page, this action is triggered with the *id* parameter representing the ID of the item type to be deleted.
- It retrieves the existing item type from the database using *Find(id).*
- The *IsDeleted* property of the item type is set to true, marking it for deletion.
- The changes are saved to the database using *_context.SaveChanges().*
- The action redirects the user to the "*ItemTypes*" index page (Index action) after a successful deletion.

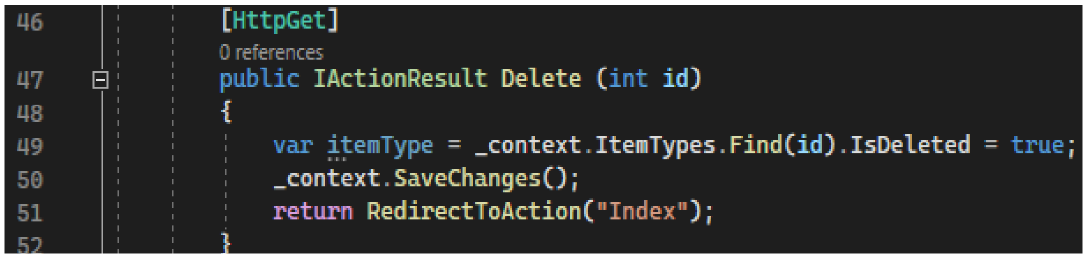

```
[HttpGet]
0 references
public IActionResult Delete (int id)
{
    var itemType = _context.ItemTypes.Find(id).IsDeleted = true;
    _context.SaveChanges();
    return RedirectToAction("Index");
}
```

Figure 3.22: ItemTypeController/Delete Action

3.2.1.4 ItemController

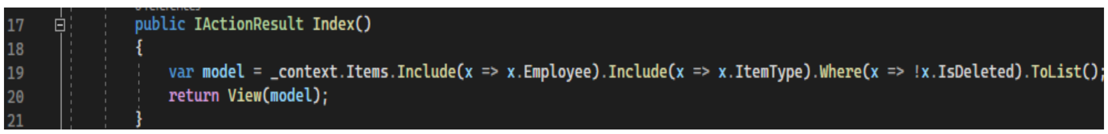
ItemController Actions Overview

Index Action (See Figure 3.23)

HTTP GET handles HTTP GET requests to display the list of items.

Explanation:

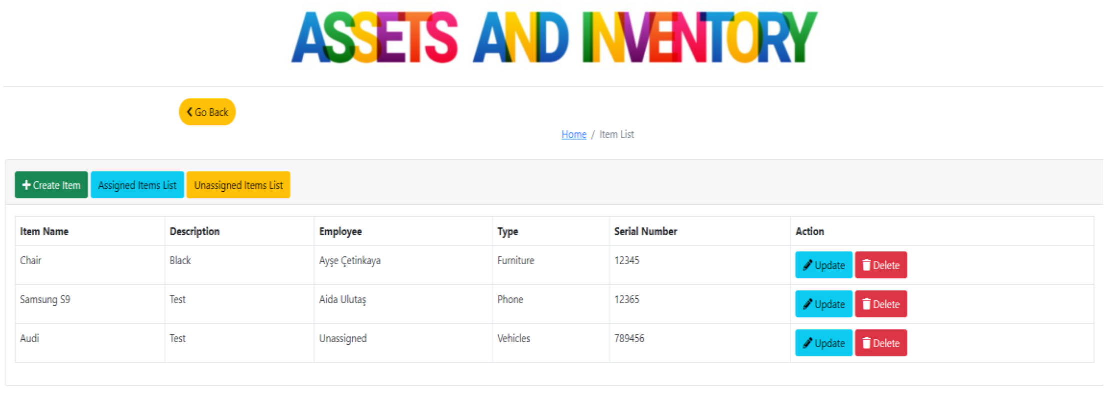
- The action retrieves a list of items from the database using *_context.Items.*
- The *Include* method is used to eagerly load related entities (Employee and ItemType) to avoid N+1 query issues.
- Items marked as deleted *(IsDeleted == true*) are excluded from the result.
- The list of items is passed as the model to the associated view (Index.cshtml).



```
public IActionResult Index()
{
    var model = _context.Items.Include(x => x.Employee).Include(x => x.ItemType).Where(x => !x.IsDeleted).ToList();
    return View(model);
}
```

Figure 3. 23: ItemController/Index Action

Bellow you can see Rendering of ItemController/Index Page



Item Name	Description	Employee	Type	Serial Number	Action
Chair	Black	Ayşe Çetinkaya	Furniture	12345	Update Delete
Samsung S9	Test	Aida Ulutaş	Phone	12365	Update Delete
Audi	Test	Unassigned	Vehicles	789456	Update Delete

Figure 3.24: ItemController- Index Final Output

ItemController- Create Action (See Figure 3.25)

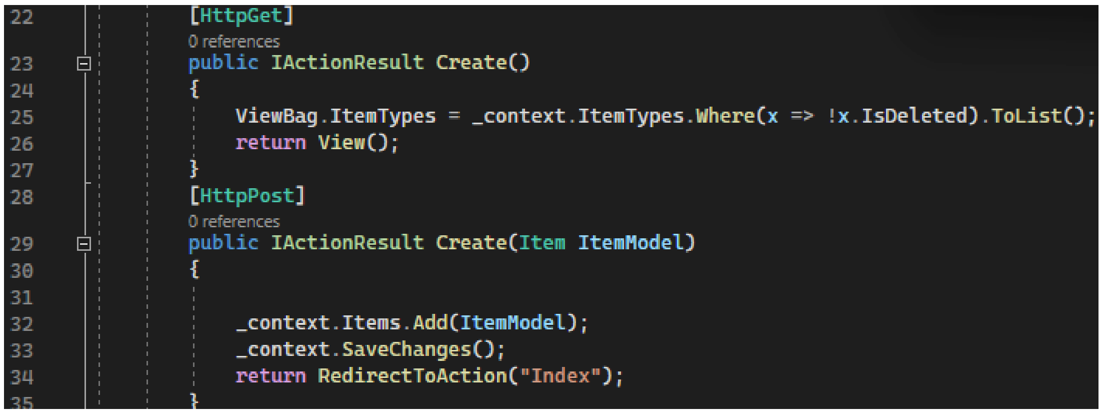
HTTP GET handles HTTP GET requests to display the form for creating a new item.

Explanation:

- The action retrieves a list of available item types from the database using *_context.ItemTypes*.
- The *Where* clause filters out item types marked as deleted (*IsDeleted)*.
- The list of item types is added to the ViewBag to make it available in the associated view (Create.cshtml).
- The action returns the view, presenting the form for creating a new item.

HTTP POST handles HTTP POST requests to process the form submission and create a new item. Explanation:

- When a user submits the form for creating a new item, this action is triggered.
- The *ItemModel* parameter represents the data submitted through the form.
- The new item is added to the Items collection in the database context *(_context)*.
- The changes are saved to the database using *_context.SaveChanges()*.
- The action redirects the user to the "Item Index" page (Index action) after a successful creation.



```
        [HttpGet]
        0 references
        public IActionResult Create()
        {
            ViewBag.ItemTypes = _context.ItemTypes.Where(x => !x.IsDeleted).ToList();
            return View();
        }
        [HttpPost]
        0 references
        public IActionResult Create(Item ItemModel)
        {

            _context.Items.Add(ItemModel);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
```

Figure 3.25: ItemController/Create Action

ItemController- Create Action Rendering Page

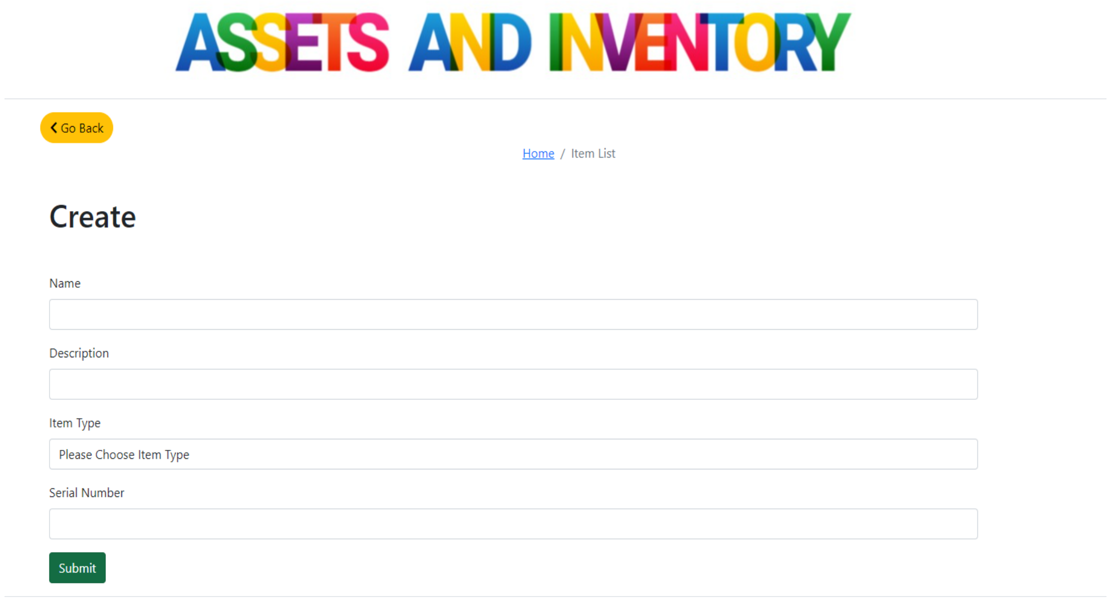


Figure 3.26: ItemController/Create Action Final Output

ItemController- Update Action (See Figure 3.27)

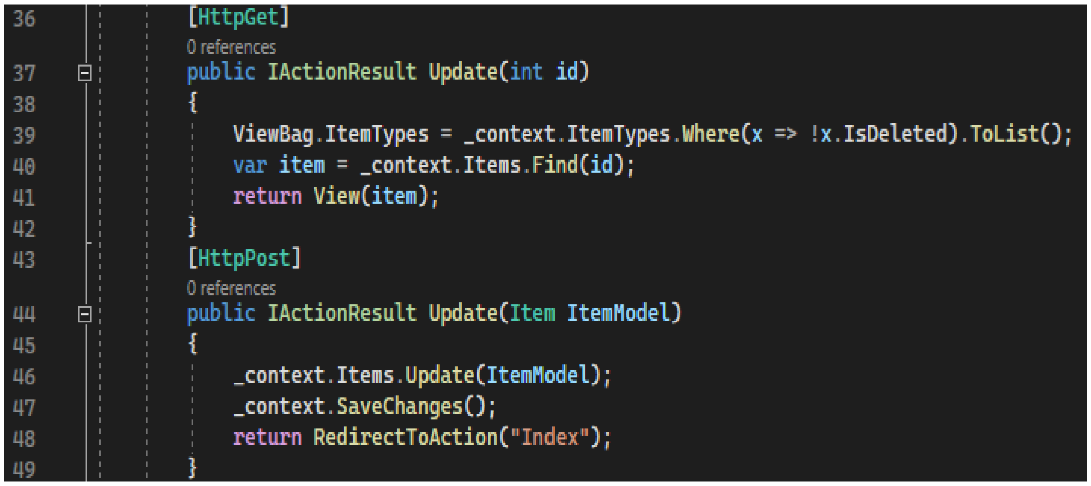
HTTP GET handles HTTP GET requests to display the form for updating an existing item. Explanation:

- The action retrieves a list of available item types from the database using *_context.ItemTypes*.
- The *Where* clause filters out item types marked as deleted (*IsDeleted*).
- The list of item types is added to the *ViewBag* to make it available in the associated view (Update.cshtml).
- The action retrieves the existing item with the specified id from the database using *Find(id)*.
- The item is passed to the view, allowing users to edit its details.

HTTP POST handles HTTP POST requests to process the form submission and update an existing item.

Explanation:

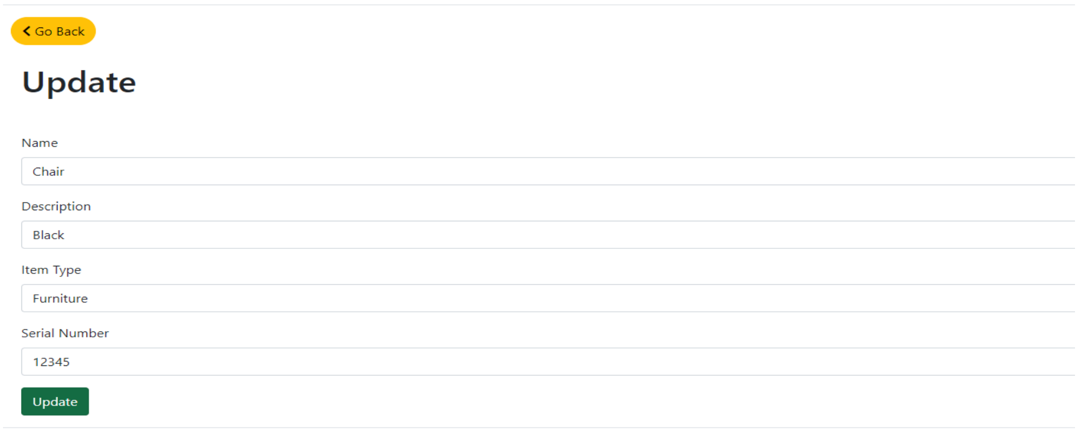
- After a user submits the form for updating an item, this action is triggered.
- The *ItemModel* parameter represents the updated data submitted through the form.
- The existing item in the database is updated with the new values from *ItemModel*.
- The changes are saved to the database using *_context.SaveChanges()*.
- The action redirects the user to the "Item Index" page (Index action) after a successful update.



```
[HttpGet]
0 references
public IActionResult Update(int id)
{
    ViewBag.ItemTypes = _context.ItemTypes.Where(x => !x.IsDeleted).ToList();
    var item = _context.Items.Find(id);
    return View(item);
}
[HttpPost]
0 references
public IActionResult Update(Item ItemModel)
{
    _context.Items.Update(ItemModel);
    _context.SaveChanges();
    return RedirectToAction("Index");
}
```

Figure 3.27: ItemController/Update Action

ItemController- Rendering Update Action Page (Figure 3.28)



Update

Name	Chair
Description	Black
Item Type	Furniture
Serial Number	12345

Figure 3.28: ItemController/Update Action Final Output

ItemController- Delete Action

HTTP GET handles HTTP GET requests to display a confirmation page for deleting an item. Explanation:

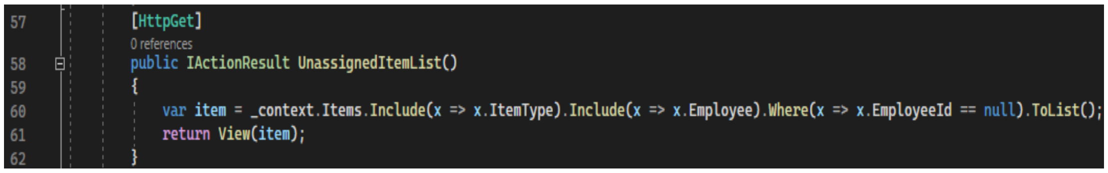
- When a user navigates to the "Delete Item" page, this action is triggered with the *id* parameter representing the ID of the item to be deleted.
- It retrieves the existing item from the database using *Find(id).*
- The *IsDeleted* property of the item is set to true, marking it for deletion.
- The changes are saved to the database using *_context.SaveChanges().*
- The action redirects the user to the "Item Index" page (Index action) after a successful deletion.

ItemController- UnassignedItemList Action (Figure 3.29)

HTTP GET handles HTTP GET requests to display a list of unassigned items.

Explanation:

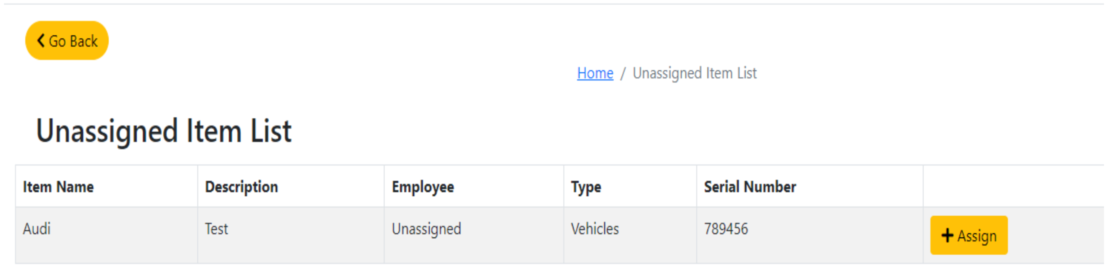
- The action retrieves a list of items from the database using *_context.Items.*
- The *Include* method is used to eagerly load related entities (ItemType and Employee) to avoid N+1 query issues.
- Only items with a null value for the *EmployeeId* property (unassigned items) are included in the result.
- The list of unassigned items is passed as the model to the associated view (UnassignedItemList.cshtml).



```
[HttpGet]
0 references
public IActionResult UnassignedItemList()
{
    var item = _context.Items.Include(x => x.ItemType).Include(x => x.Employee).Where(x => x.EmployeeId == null).ToList();
    return View(item);
}
```

Figure 3.29: ItemController/UnassignedItemList Action

ItemController- Rendering UnassignedItemList Action Page (Figure 3.30)



Unassigned Item List

Item Name	Description	Employee	Type	Serial Number	
Audi	Test	Unassigned	Vehicles	789456	+ Assign

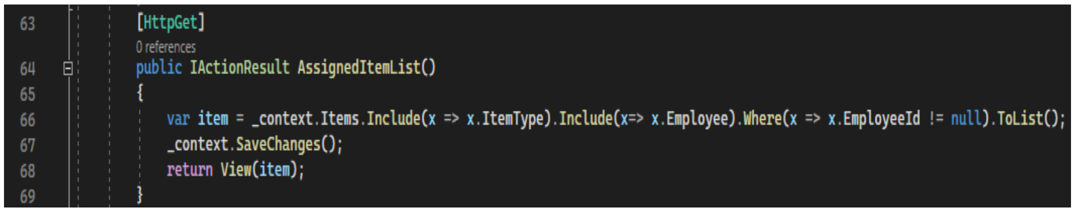
Figure 3.30: ItemController/UnassignedItemList Final Output

ItemController- AssignedItemList Action (See Figure 3.31)

HTTP GET handles HTTP GET requests to display a list of assigned items.

Explanation:

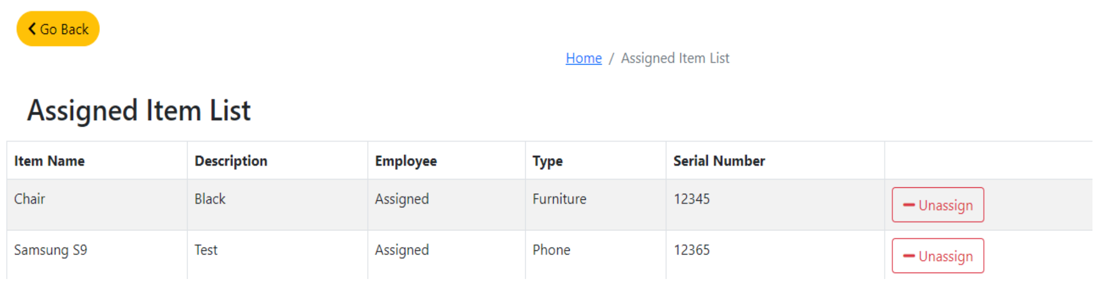
- The action retrieves a list of items from the database using *_context.Items.*
- The *Include* method is used to eagerly load related entities (ItemType and Employee) to avoid N+1 query issues.
- Only items with a non-null value for the *EmployeeId* property (assigned items) are included in the result.
- The list of assigned items is passed as the model to the associated view (AssignedItemList.cshtml).



```
[HttpGet]
0 references
public IActionResult AssignedItemList()
{
    var item = _context.Items.Include(x => x.ItemType).Include(x=> x.Employee).Where(x => x.EmployeeId != null).ToList();
    _context.SaveChanges();
    return View(item);
}
```

Figure 3.31: ItemController/AssignedItemList Action

ItemController- Rendering AssignedItemList Action Page (Figure 3.32)



Assigned Item List

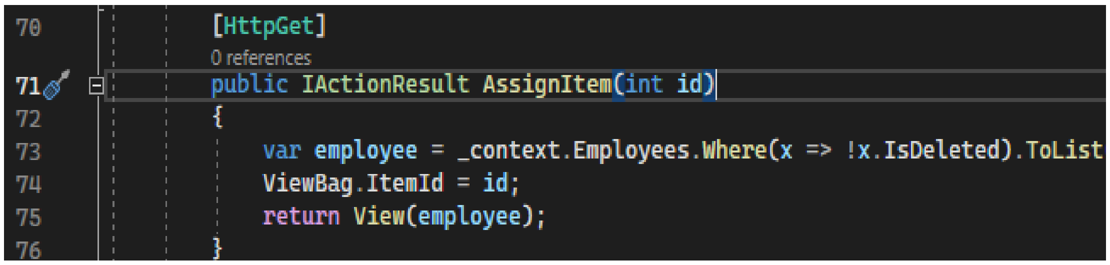
Item Name	Description	Employee	Type	Serial Number	
Chair	Black	Assigned	Furniture	12345	Unassign
Samsung S9	Test	Assigned	Phone	12365	Unassign

Figure 3.32: ItemController/AssignedItemList Action

ItemController- AssignItem Action (Figure 3.33)

HTTP GET handles HTTP GET requests to display the form for assigning an item to an employee. Explanation:

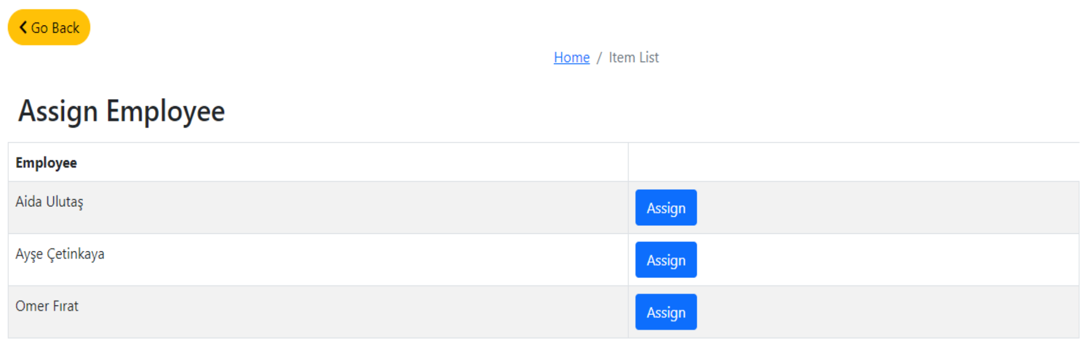
- The action retrieves a list of available employees from the database using *_context.Employees.*
- The *Where* clause filters out employees marked as deleted (*IsDeleted).*
- The list of employees is added to the *ViewBag* to make it available in the associated view (AssignItem.cshtml).
- The *ItemId* is added to the *ViewBag* to capture the ID of the item being assigned.



```
[HttpGet]
0 references
public IActionResult AssignItem(int id)
{
    var employee = _context.Employees.Where(x => !x.IsDeleted).ToList
    ViewBag.ItemId = id;
    return View(employee);
}
```

Figure 3.33: ItemController/ AssignItem Action

ItemController – Rendering AssignItem Action Page



Assign Employee

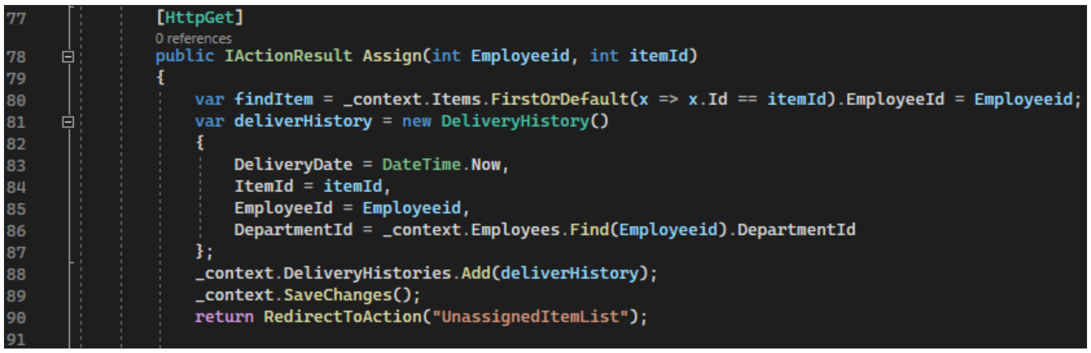
Employee	
Aida Ulutaş	Assign
Ayşe Çetinkaya	Assign
Omer Fırat	Assign

Figure 3.34: ItemController/AssignItem Action Final Output

Item Controller – Assign Action (See Figure 3.35)

HTTP GET handles HTTP GET requests to process the assignment of an item to an employee. Explanation:

- The action retrieves the item with the specified *itemId* from the database using *FirstOrDefault*.
- The *EmployeeId* property of the item is updated with the new value (Employeeid).
- A new DeliveryHistory record is created to log the assignment details.
- The changes are saved to the database using *_context.SaveChanges()*.
- The action redirects the user to the "Unassigned Item List" page (UnassignedItemList action) after a successful assignment.



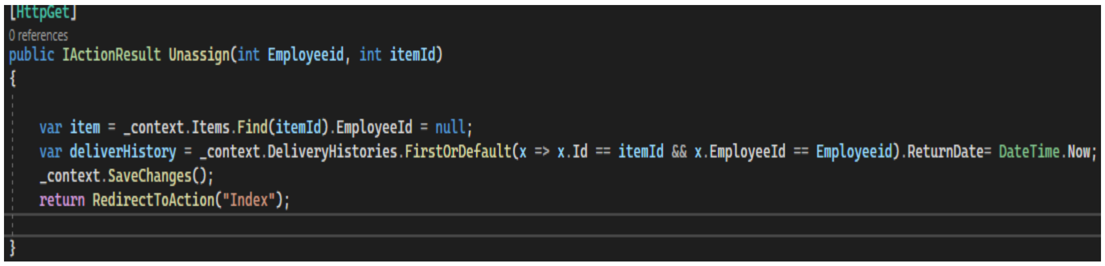
```
[HttpGet]
0 references
public IActionResult Assign(int Employeeid, int itemId)
{
    var findItem = _context.Items.FirstOrDefault(x => x.Id == itemId).EmployeeId = Employeeid;
    var deliverHistory = new DeliveryHistory()
    {
        DeliveryDate = DateTime.Now,
        ItemId = itemId,
        EmployeeId = Employeeid,
        DepartmentId = _context.Employees.Find(Employeeid).DepartmentId
    };
    _context.DeliveryHistories.Add(deliverHistory);
    _context.SaveChanges();
    return RedirectToAction("UnassignedItemList");
```

Figure 3.35: ItemController/Assign Action

Item Controller – Unassign Action (See Figure 3.36)

HTTP GET handles HTTP GET requests to process the unassignment of an item from an employee. Explanation:

- The action retrieves the item with the specified *itemId* from the database using *Find*.
- The *EmployeeId* property of the item is set to null to indicate that it is unassigned.
- The associated *DeliveryHistory* record is retrieved using *FirstOrDefault* based on the item and employee IDs.
- The *ReturnDate* property of the delivery history is updated to record the unassignment date and time.
- The changes are saved to the database using *_context.SaveChanges()*.
- The action redirects the user to the "Item Index" page (Index action) after a successful unassignment.



```
[HttpGet]
0 references
public IActionResult Unassign(int Employeeid, int itemId)
{

    var item = _context.Items.Find(itemId).EmployeeId = null;
    var deliverHistory = _context.DeliveryHistories.FirstOrDefault(x => x.Id == itemId && x.EmployeeId == Employeeid).ReturnDate= DateTime.Now;
    _context.SaveChanges();
    return RedirectToAction("Index");

}
```

Figure 3.36: ItemController/Unassign Action

3.2.1.5 DepartmentController

The *DepartmentController* manages CRUD operations for departments. Its actions are similar to previous controllers, handling index listing, creation, updating, and deletion of department records. DepartmentController Actions (See Figure 3.37)

Index:

- Displays a list of departments that are not marked as deleted.
- Retrieves the list from the database using *_context.Departments* and filters out deleted departments.
- Renders the list in the associated view (Index.cshtml).

Create (GET):

- Displays the form for creating a new department.
- Renders the form in the associated view (Create.cshtml).

Create (POST):

- Handles the form submission for creating a new department.
- Adds the new department to the database using *_context.Departments.Add*.
- Saves changes to the database using *_context.SaveChanges*.
- Redirects the user to the "Index" page after a successful creation.

Update (GET):

- Displays the form for updating an existing department.
- Retrieves the department with the specified ID from the database using *Find*.
- Renders the form pre-filled with department details in the associated view (Update.cshtml).

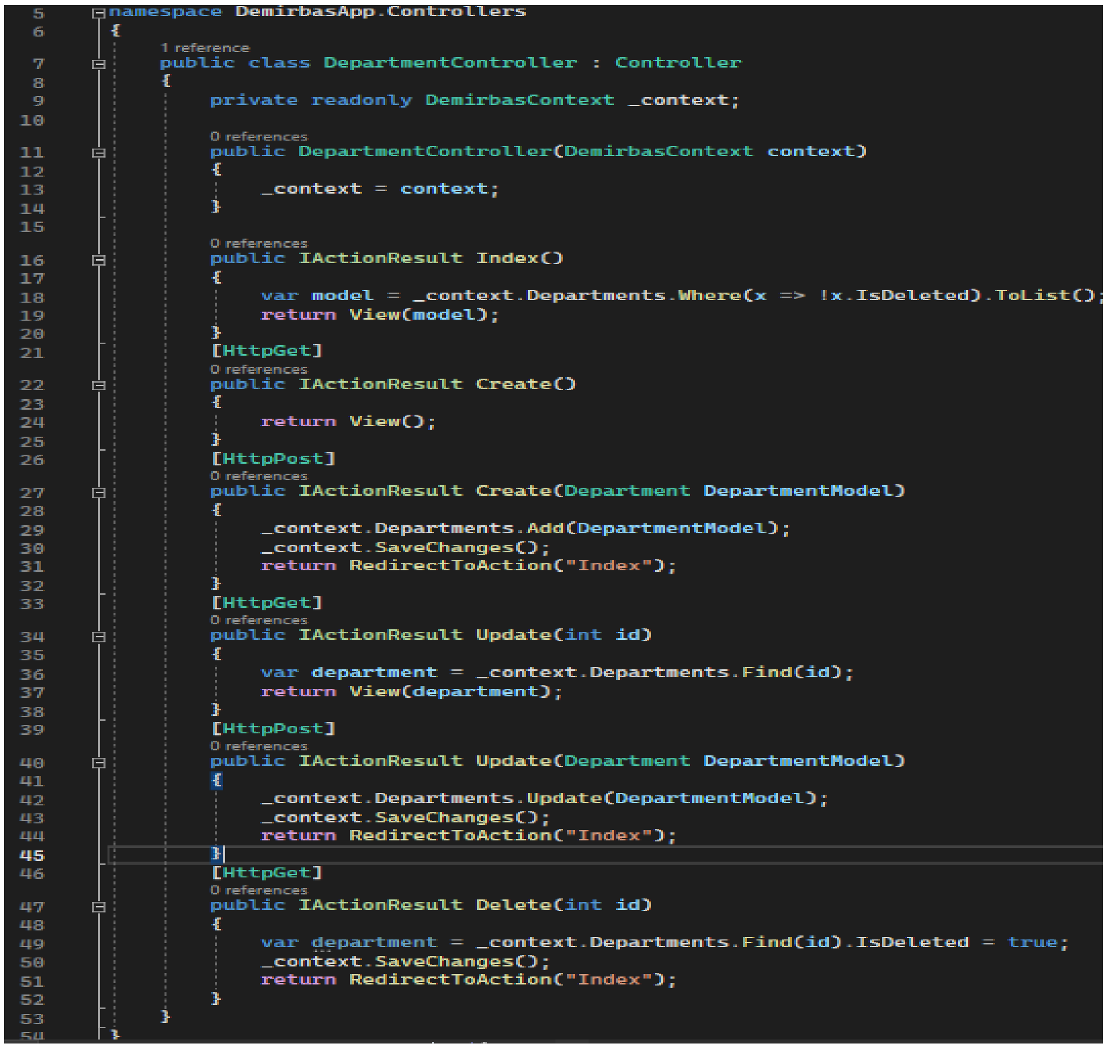
Create (POST):

- Handles the form submission for updating an existing department.

- Updates the existing department in the database using *_context.Departments.Update*.
- Saves changes to the database using *_context.SaveChanges*.
- Redirects the user to the "Index" page after a successful update.

Delete:

- Marks the department with the specified ID as deleted.
- Retrieves the department from the database using *Find* and sets *IsDeleted* to true.
- Saves changes to database using *_context.SaveChanges* and redirects to "Index".

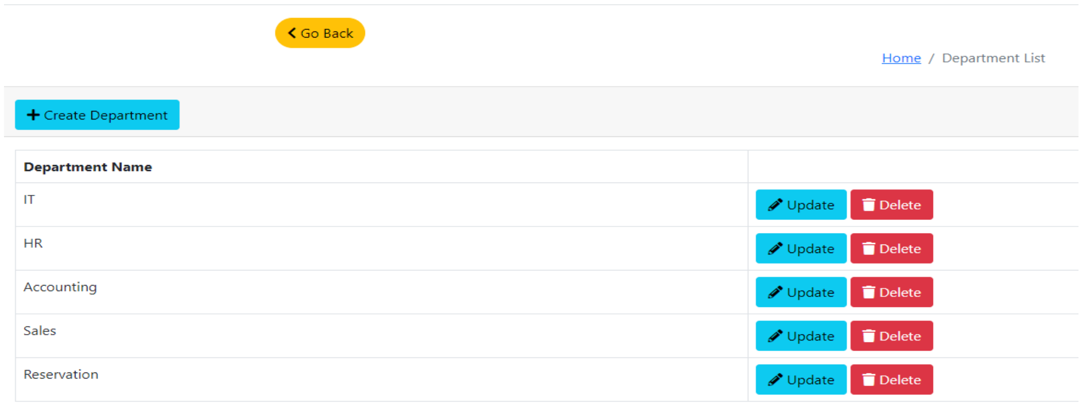


```
namespace DemirbasApp.Controllers
{
    1 reference
    public class DepartmentController : Controller
    {
        private readonly DemirbasContext _context;

        0 references
        public DepartmentController(DemirbasContext context)
        {
            _context = context;
        }

        0 references
        public IActionResult Index()
        {
            var model = _context.Departments.Where(x => !x.IsDeleted).ToList();
            return View(model);
        }
        [HttpGet]
        0 references
        public IActionResult Create()
        {
            return View();
        }
        [HttpPost]
        0 references
        public IActionResult Create(Department DepartmentModel)
        {
            _context.Departments.Add(DepartmentModel);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
        [HttpGet]
        0 references
        public IActionResult Update(int id)
        {
            var department = _context.Departments.Find(id);
            return View(department);
        }
        [HttpPost]
        0 references
        public IActionResult Update(Department DepartmentModel)
        {
            _context.Departments.Update(DepartmentModel);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
        [HttpGet]
        0 references
        public IActionResult Delete(int id)
        {
            var department = _context.Departments.Find(id).IsDeleted = true;
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```

Figure 3.37: DepartmentController/Actions

Similar to other controllers, these actions follow a common pattern for handling CRUD operations. The associated pages for these actions (Index.cshtml, Create.cshtml, Update.cshtml) are below:

Rendering Index Page (Figure 3.38)

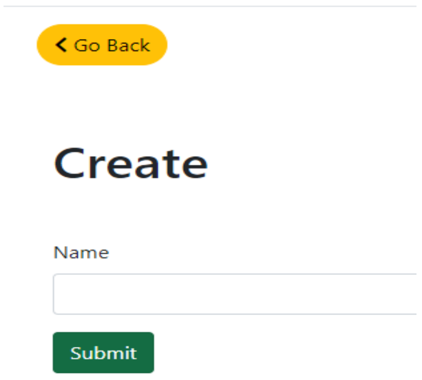


Department Name	
IT	Update Delete
HR	Update Delete
Accounting	Update Delete
Sales	Update Delete
Reservation	Update Delete

Figure 3.38: DepartmentController/Index Final Output

Rendering Create Page (Figure 3.38)

Rendering Update Page (Figure 3.39)



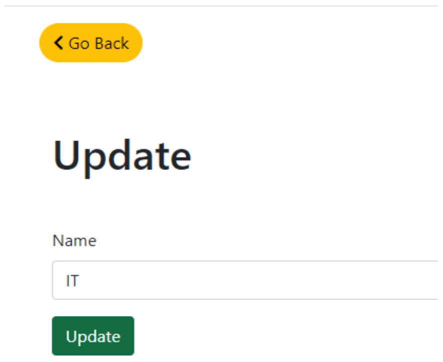


Figure 3.39: Create Page

Figure 3.40: Update Page

3.2.1.6 EmployeeController

The *EmployeeController* manages CRUD operations for employees. Similar to previous controllers, it handles actions for listing employees, creating new employees, updating employee details, and marking employees as deleted.

EmployeeController Actions (See Figure 3.41)

Index:

- Displays a list of employees with associated department information.
- Retrieves the list from the database using *_context.Employees*.
- Eagerly loads the associated department information using *Include*.
- Filters out deleted employees with *Where*.
- Renders the list in the associated view (Index.cshtml).

Create (GET):

- Displays the form for creating a new employee.
- Retrieves a list of available departments from the database using *_context.Departments*.
- Renders the form in the associated view (Create.cshtml) and includes the list of departments in the *ViewBag*.

Create (POST):

- Handles the form submission for creating a new employee.
- Adds the new employee to the database using *_context.Employees.Add*.
- Saves changes to the database using *_context.SaveChanges.*
- Redirects the user to the "Index" page after a successful creation.

Update (GET):

- Displays the form for updating an existing employee.
- Retrieves the employee with the specified ID from the database using *Find*.

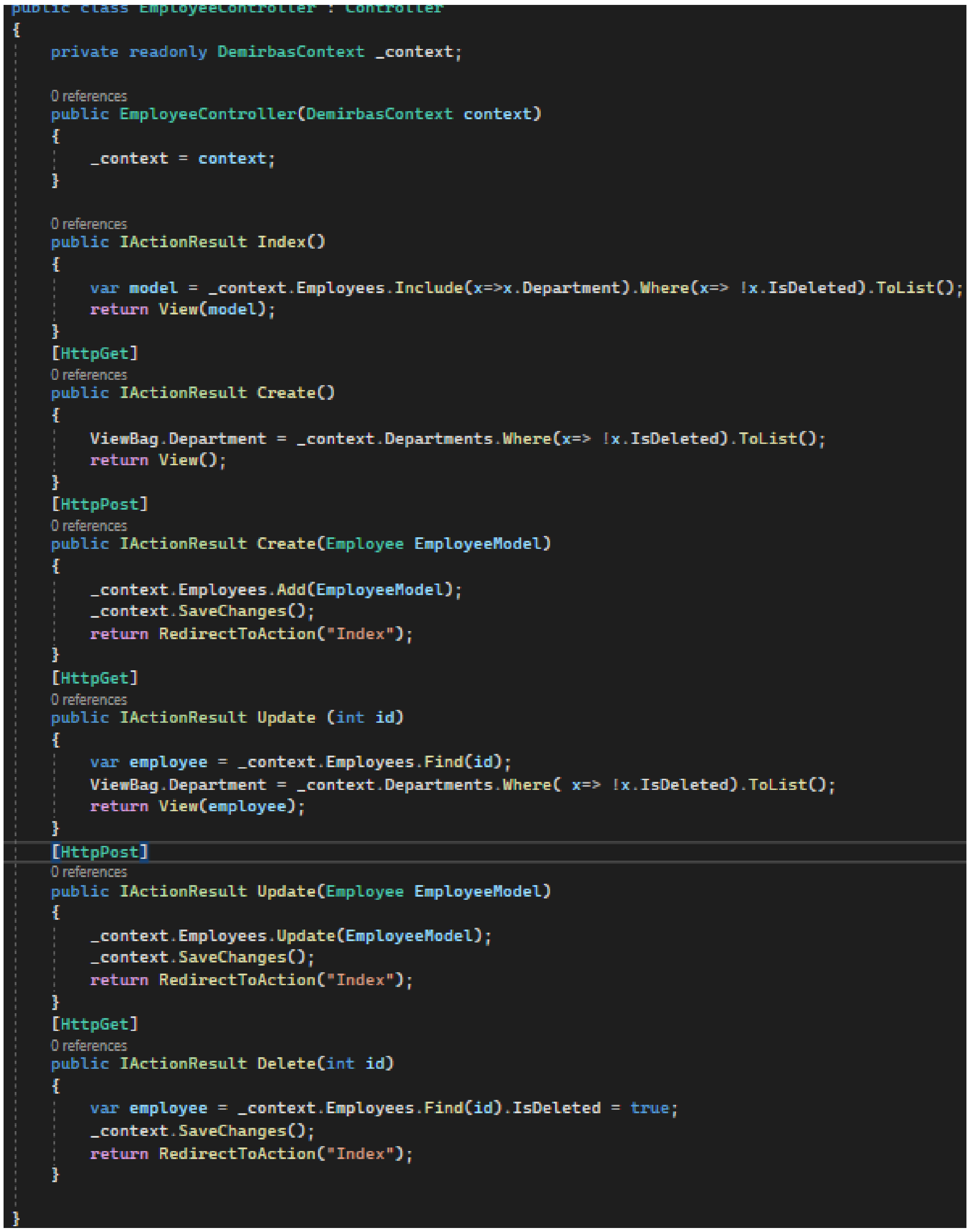
- Retrieves a list of available departments from the database using *_context.Departments.*
- Renders the form pre-filled with employee details and includes the list of departments in the *ViewBag*.
- Renders the form in the associated view (Update.cshtml).

Update (POST)

- Handles the form submission for updating an existing employee.
- Updates the existing employee in the database using *_context.Employees.Update*.
- Saves changes to the database using *_context.SaveChanges.*
- Redirects the user to the "Index" page after a successful update.

Delete:

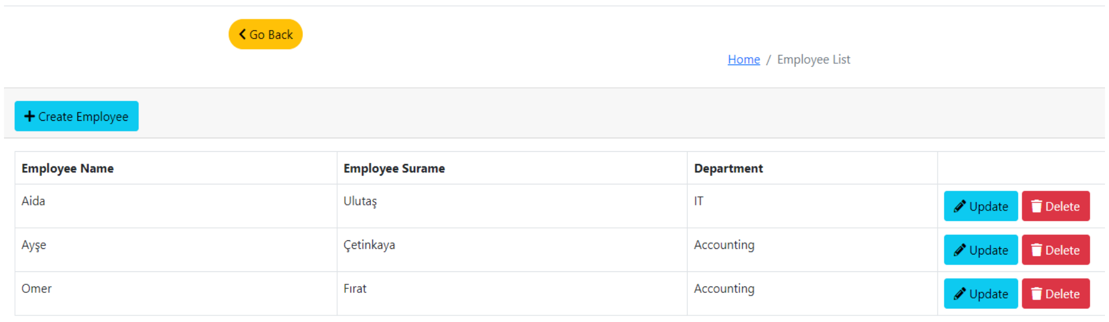
- Marks the employee with the specified ID as deleted.
- Retrieves the employee from the database using *Find* and sets *IsDeleted* to true.
- Saves changes to the database using *_context.SaveChanges.*
- Redirects the user to the "Index" page after a successful deletion.



```
public class EmployeeController : Controller
{
    private readonly DemirbasContext _context;

    0 references
    public EmployeeController(DemirbasContext context)
    {
        _context = context;
    }

    0 references
    public IActionResult Index()
    {
        var model = _context.Employees.Include(x=>x.Department).Where(x=> !x.IsDeleted).ToList();
        return View(model);
    }
    [HttpGet]
    0 references
    public IActionResult Create()
    {
        ViewBag.Department = _context.Departments.Where(x=> !x.IsDeleted).ToList();
        return View();
    }
    [HttpPost]
    0 references
    public IActionResult Create(Employee EmployeeModel)
    {
        _context.Employees.Add(EmployeeModel);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    [HttpGet]
    0 references
    public IActionResult Update (int id)
    {
        var employee = _context.Employees.Find(id);
        ViewBag.Department = _context.Departments.Where( x=> !x.IsDeleted).ToList();
        return View(employee);
    }
    [HttpPost]
    0 references
    public IActionResult Update(Employee EmployeeModel)
    {
        _context.Employees.Update(EmployeeModel);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    [HttpGet]
    0 references
    public IActionResult Delete(int id)
    {
        var employee = _context.Employees.Find(id).IsDeleted = true;
        _context.SaveChanges();
        return RedirectToAction("Index");
    }

}
```

Figure 3.41: EmployeeController Actions

EmployeeController Rendering Index Page (Figure 3.42)

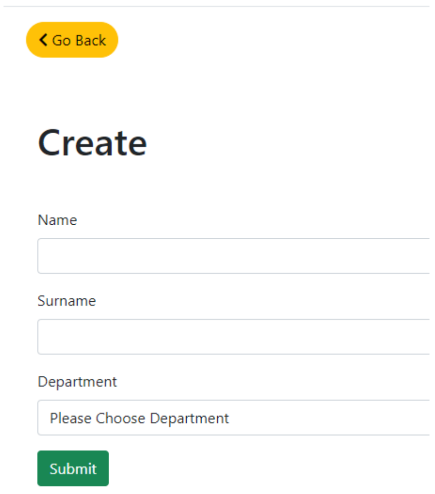


Employee Name	Employee Surame	Department	
Aida	Ulutaş	IT	Update Delete
Ayşe	Çetinkaya	Accounting	Update Delete
Omer	Fırat	Accounting	Update Delete

Figure 3.42: EmployeeController/Index Final Output

Create Rendering Page (Figure 3.42)

Update Rendering Page (Figure 3.43)



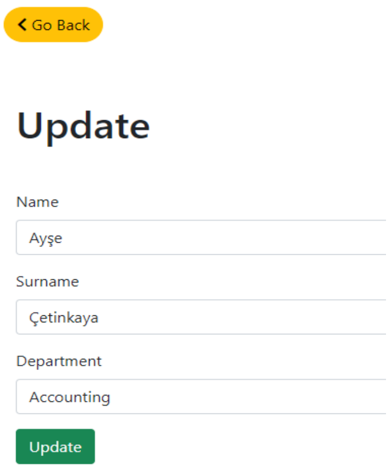


Figure 3.43: Create Final Output

Figure 3.44: Update Final Output

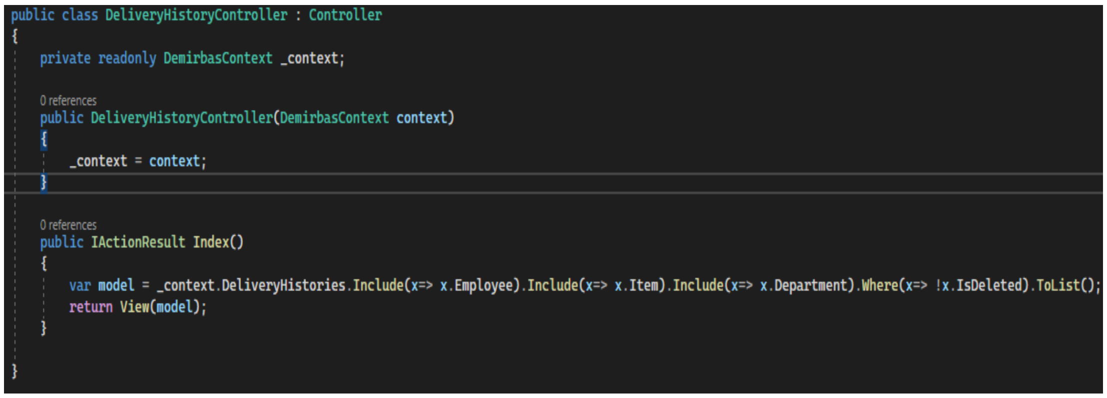
3.2.1.7 DeliveryHistoryController

The *DeliveryHistoryController* manages the viewing of delivery history records. It includes an action to list delivery history records, providing details about the employees, items, and departments involved in each delivery. The *DeliveryHistoryController* focuses on providing a read-only view of the delivery history records. Additional actions for creating, updating, or deleting delivery history records can be added if needed.

Actions:

Index (See Figure 3.45)

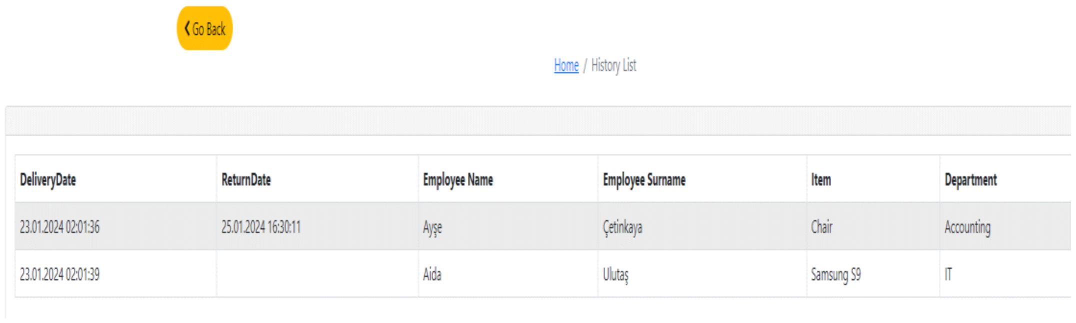
- Displays a list of delivery history records.
- Retrieves the list from the database using *_context.DeliveryHistories.*
- Eagerly loads associated employee, item, and department information using *Include*.
- Filters out deleted delivery history records with *Where*.
- Renders the list in the associated view (Index.cshtml).



```
public class DeliveryHistoryController : Controller
{
    private readonly DemirbasContext _context;

    0 references
    public DeliveryHistoryController(DemirbasContext context)
    {
        _context = context;
    }

    0 references
    public IActionResult Index()
    {
        var model = _context.DeliveryHistories.Include(x=> x.Employee).Include(x=> x.Item).Include(x=> x.Department).Where(x=> !x.IsDeleted).ToList();
        return View(model);
    }

}
```

Figure 3.45: DeliveryHistoryController/Index Action

DeliveryHistory Rendering Index Page (Figure 3.46)



DeliveryDate	ReturnDate	Employee Name	Employee Surname	Item	Department
23.01.2024 02:01:36	25.01.2024 16:30:11	Ayşe	Çetinkaya	Chair	Accounting
23.01.2024 02:01:39		Aida	Ulutaş	Samsung S9	IT

Figure 3.46: DeliveryHistory/Index Final Output

3.2.2 Areas

The "Identity" folder within the "Areas" directory serves as the dedicated location for the Entity Framework Core Identity Provider to handle user authentication and authorization processes. This includes the implementation of essential functionalities such as login, reset password, logout, and registration.

3.2.3 Program.cs

The Program.cs (See Figure 3.47) sets up an ASP.NET Core application with MVC, Identity, SQLite database, Razor Pages, and various middleware for routing, authentication, and authorization.

Service Configuration:

- ‘AddControllersWithViews’: Adds MVC services to the application.
- ‘AddDbContext<DemirbasContext>’: Configures the application to use the ‘DemirbasContext’ for database operations, with SQLite as the database provider.

Identity Configuration:

- AddIdentity<ApplicationUser, ApplicationRole>: Configures Identity services using custom ApplicationUser and ApplicationRole classes.
- AddEntityFrameworkStores<DemirbasContext>: Specifies that Identity data should be stored in the DemirbasContext database.
- AddDefaultUI() and AddDefaultTokenProviders(): Configures Identity to include default UI and token providers.

Razor Pages Configuration:

- AddRazorPages(): Adds Razor Pages services to the application.

Application Configuration:

- app.Environment.IsDevelopment(): Checks if the application is running in the development environment.

Middleware Configuration:

- UseExceptionHandler: Configures exception handling to redirect to the error page in non-development environments.
- UseHsts(): Configures HTTP Strict Transport Security (HSTS) for enhanced security.

Routing and Authorization:

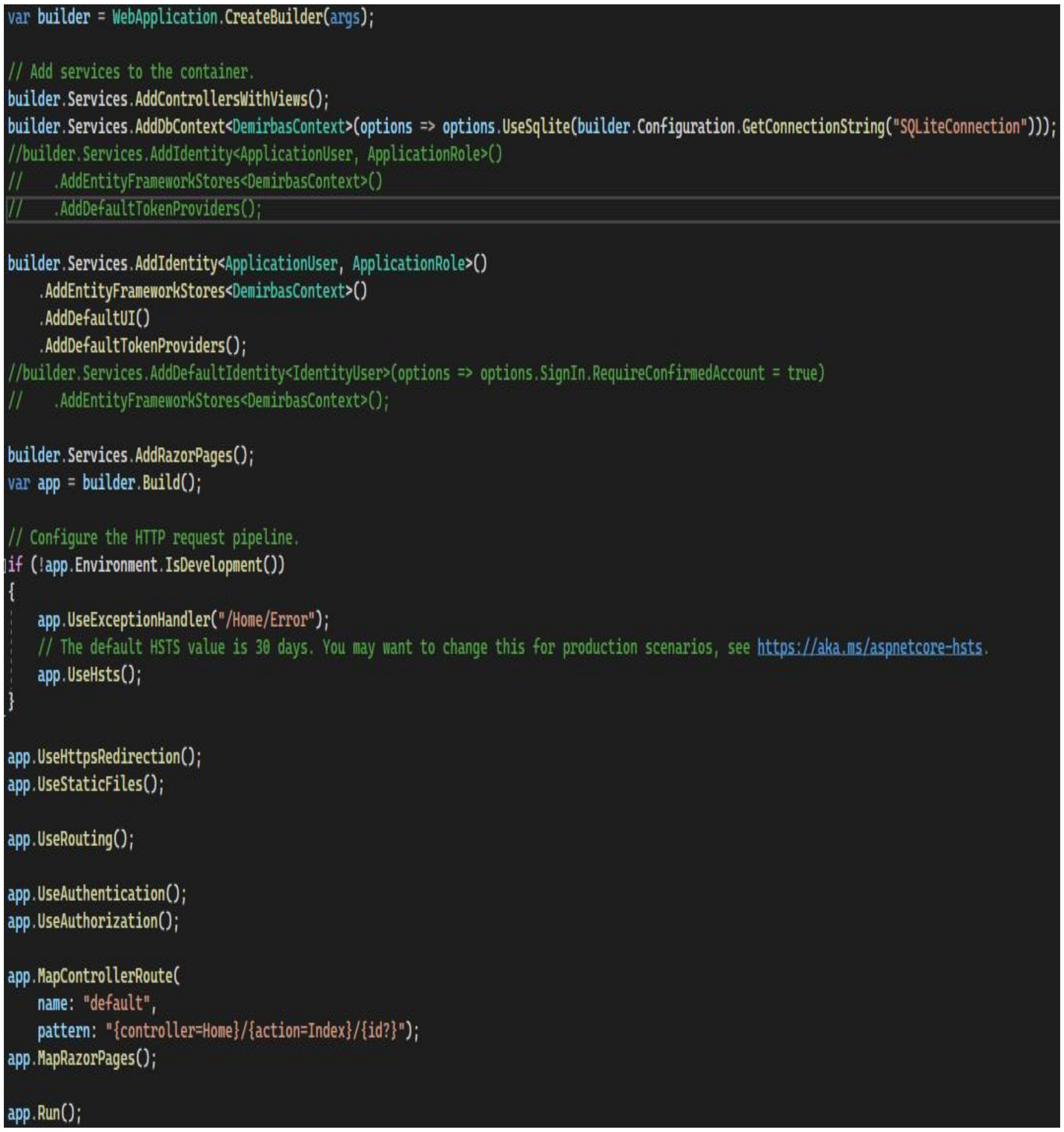
- UseHttpsRedirection(): Redirects HTTP requests to HTTPS.
- UseStaticFiles(): Configures the application to serve static files.
- UseRouting(): Configures routing for the application.
- UseAuthentication() and UseAuthorization(): Enable authentication and authorization.

Endpoint Mapping:

- MapControllerRoute: Defines a default route for MVC controllers.

App Run:

- app.Run(): Runs the application.
- This Program.cs sets up an ASP.NET Core application with MVC, Identity, SQLite database, Razor Pages, and various middleware for routing, authentication, and authorization. It's a typical configuration for a web application.



```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<DemirbasContext>(options => options.UseSqlite(builder.Configuration.GetConnectionString("SQLiteConnection")));
//builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
//    .AddEntityFrameworkStores<DemirbasContext>()
//    .AddDefaultTokenProviders();

builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
    .AddEntityFrameworkStores<DemirbasContext>()
    .AddDefaultUI()
    .AddDefaultTokenProviders();
//builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
//    .AddEntityFrameworkStores<DemirbasContext>();

builder.Services.AddRazorPages();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.MapRazorPages();

app.Run();
```

Figure 3.47: Program.cs

References

Microsoft. “A Tour of the C# Language.” Accessed 30 January 2024. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Radixweb. “.NET Core vs .NET Framework: Which Beats the Other in App Development?”. Accessed 30 January 2024. https://radixweb.com/blog/net-core-vs-net-framework

Microsoft.” Overview of ASP.NET Core”. Accessed 30 January 2024. https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0

C#Corner. “Entity Framework Using C#”. Accessed 30 January 2024. https://www.c-sharpcorner.com/article/entity-framework-introduction-using-c-sharp-part-one/
TechTarget/The Server Side. “Object-Relational Mapping”. Accessed 30 January 2024. https://www.theserverside.com/definition/object-relational-mapping-ORM

Oracle. “What is a Relational Database”. Accessed 30 January 2024. https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables

SimpliLearn. “What is SQLite”. Accessed 30 January 2024. https://www.simplilearn.com/tutorials/sql-tutorial/what-is-sqlite#:~:text=SQLite%20is%20an%20embedded%2C%20server,than%20other%20database%20management%20systems

Tech Target. “Object-Oriented Programing”. Accessed 30 January 2024. https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

Code Academy. “What is ASP:NET Razor Pages?”. Accessed 30 January 2024. https://www.codecademy.com/article/what-is-asp-net-razor-pages

Tutorials Point. “What is CSS”. Accessed 30 January 2024. https://www.tutorialspoint.com/css/what_is_css.htm

W3Schools. “HTML Introduction”. Accessed 30 January 2024. https://www.w3schools.com/html/html_intro.asp

W3Schools. “What is Bootstrap”. Accessed 30 January 2024. https://www.w3schools.com/whatis/whatis_bootstrap.asp

W3Schools. “jQuery Introduction”. Accessed 30 January 2024. https://www.w3schools.com/jquery/jquery_intro.asp

GeeksForGeeks. ”MVC Framework Intoruction”. Accessed 30 January 2024. https://www.geeksforgeeks.org/mvc-framework-introduction/

Microsoft. “Get started with EF Core in an ASP:NET MVC web App”. Accessed 30 January 2024.

https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-8.0

Microsoft. “Introduction to Identity on ASP:NET Core”. Accessed 30 January 2024. https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio

Microsoft. “Constructors (C# Programing Guide). Accessed 30 January 2024. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors