

Guida scritta da Marco Guerriero

Riferimenti: www.pluralsight.com, by Gill Cleeren

Indice

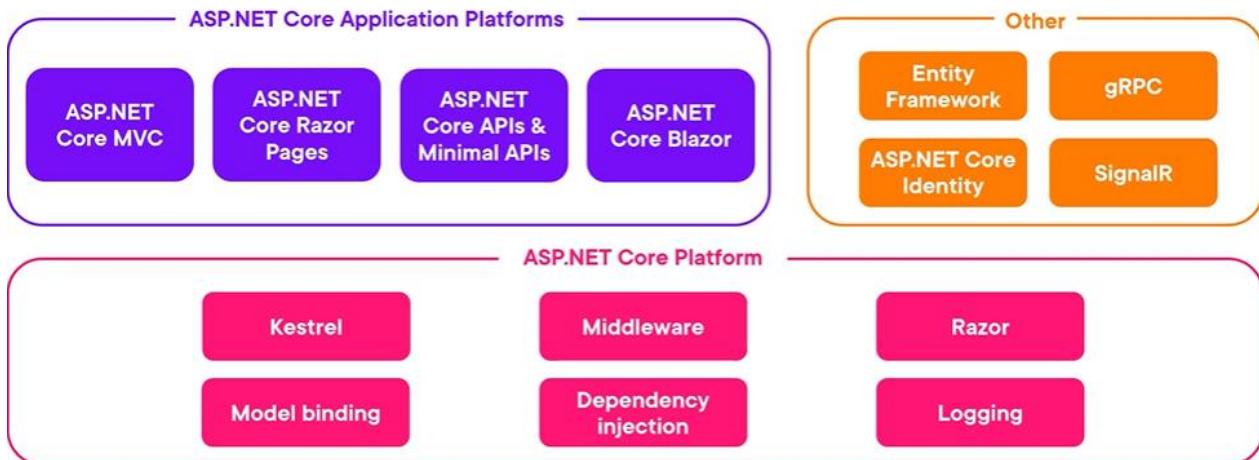
- [Understanding ASP.NET Core and ASP.NET Core MVC](#)
 - What Is ASP.NET Core?
 - ASP.NET Core Components
 - Why ASP.NET Core
- [Setting up an ASP.NET Core MVC Application](#)
 - Creating a New Project
 - Demo: Creating and Running a New Project
 - Demo: Creating a New Project Using the CLI
 - Demo: Using VS Code and the C# Dev Kit
 - Exploring a New Project
 - Demo: Exploring a New Project in Visual Studio
 - Configuring the Site
 - Demo: Configuring the Application
- [Creating the First Page](#)
 - Introducing the MVC Pattern
 - Creating the Model and the Repository
 - Demo: Creating the Model and the Repository
 - Creating the Controller
 - Demo: Adding the Controller
 - Adding the View
 - Demo: Creating the First View
 - Demo: Using a View Model
 - Adding Extra View Files
 - Demo: Adding Extra View Files
 - Styling the View
 - Demo: Styling the View
- [Working with Real Data Using Entity Framework Core](#)
 - Introducing Entity Framework Core
 - Adding EF Core to the Application
 - Demo: Adding EF Core to the Application
 - Creating the Repository
 - Demo: Creating the Repository
 - Using Migrations
 - Demo: Creating the Database Using Migrations
 - Demo Adding Seed Data

- [Adding Routes and Navigation](#)
 - Adding Routes and Navigation
 - Demo: Adding Routes to the Application
 - Configuring Routes
 - Navigating with Tag Helpers
 - Demo: Adding Navigation to the Site
- [Improving the Views in the Application](#)
 - Using Partial Views
 - Demo: Adding a Partial View
 - Creating the Shopping Cart
 - Demo: Creating the Shopping Cart
 - Working with View Components
 - Demo: Creating View Components
 - Creating a Custom Tag Helper
 - Demo: Creating the Email Tag Helper
- [Working with Forms and Model Binding](#)
 - Creating a Form Using Tag Helpers
 - Demo: Creating the Order Form
 - Understanding Model Binding
 - Demo: Accessing Posted Data Using Model Binding
 - Adding Validation
 - Demo: Adding Validation
 - Demo: Adding Clientside Validation
 - Understanding Razor Pages
 - Demo: Recreating the Form Using Razor Pages
- [Testing the Application Components](#)
 - Understanding Unit Tests
 - Writing Unit Tests
 - Demo: Testing the Controllers
 - Demo: Testing a Tag Helper

- [Making the Site Interactive](#)
 - Searching Using JavaScript and an ASP.NET Core API
 - Creating an ASP.NET Core RESTful API
 - Demo: Setting up the API
 - Creating the API Responses
 - Demo: Completing the API
 - Adding jQuery and Ajax
 - Demo: Creating the Search Page with Ajax and the API
 - Introducing ASP.NET Core Blazor
 - Demo: Exploring a New Blazor Project
 - Demo: Creating the Search Page Using Blazor
- [Bringing in Authentication and Authorization](#)
 - Understanding ASP.NET Core Identity
 - Demo: Adding ASP.NET Core Identity to the Application
 - Adding Authentication to the Site
 - Demo: Adding Authentication
 - Using Authorization
 - Demo: Using Authorization
- [Deploying the Site to an Azure App Service](#)
 - Understanding the Azure App Service
 - Demo: Looking at the Azure Portal
 - Deploying the Application to an Azure App Service
 - Demo: Deploying the Site

Understanding ASP.NET Core and ASP.NET Core MVC

The Architecture of ASP.NET Core



ASP.NET è un Open Source **framework** di sviluppo web creato da Microsoft, utilizzato per costruire applicazioni web, servizi web e API. È parte integrante della piattaforma .NET e consente agli sviluppatori di creare siti web dinamici, scalabili e sicuri utilizzando tecnologie come **HTML, CSS, JavaScript** e **C#** (o **VB.NET**).

Componenti Principali di ASP.NET:

1. ASP.NET Core:

- ASP.NET Core è una versione moderna, cross-platform (funziona su Windows, macOS e Linux) e open-source di ASP.NET.
- Consente la creazione di applicazioni web e API in modo più modulare e performante rispetto alla versione classica.
- È progettato per essere cloud-ready e altamente scalabile.

2. ASP.NET MVC (Model-View-Controller):

- Un framework basato sul pattern MVC, che separa l'applicazione in tre componenti principali:
 - Model:** Rappresenta i dati e la logica di business.
 - View:** Gestisce la presentazione dei dati (interfaccia utente).
 - Controller:** Gestisce le richieste dell'utente e collega il Model alla View.
- ASP.NET MVC permette una migliore organizzazione del codice e una separazione delle responsabilità.

3. ASP.NET Web API:

[Home](#)

- Utilizzato per costruire API RESTful, ossia interfacce che permettono la comunicazione tra applicazioni o tra applicazioni e client (come browser o applicazioni mobile).
- Le API Web API sono comunemente utilizzate per sviluppare servizi che restituiscono dati in formato JSON o XML, utilizzabili da applicazioni client.

4. ASP.NET Web Forms (versione classica, ormai meno utilizzata):

- Era una tecnologia più tradizionale che permetteva lo sviluppo rapido di applicazioni web con un approccio "drag-and-drop", ma che ha meno flessibilità e controllo rispetto a MVC.

5. Blazor:

- Parte di ASP.NET Core, Blazor consente di costruire interfacce utente interattive utilizzando C# anziché JavaScript. Può funzionare sia lato server che client (tramite WebAssembly).

Come Funziona ASP.NET:

1. Gestione delle richieste:

- Quando un utente invia una richiesta a un'applicazione ASP.NET (come visitare una pagina web o chiamare un'API), la richiesta viene gestita dal server web (come IIS o Kestrel).
- ASP.NET processa la richiesta e la invia al Controller (nel caso di MVC) o alla logica definita (nel caso di Web API o Web Forms).

2. Routing:

- ASP.NET utilizza un sistema di routing per mappare le URL alle azioni del Controller o alle pagine specifiche.
- Questo permette di definire URL amichevoli e organizzare meglio le risorse dell'applicazione.

3. Sicurezza:

- Supporta vari metodi di autenticazione (OAuth, JWT, cookie-based authentication, ecc.) e autorizzazione, permettendo agli sviluppatori di proteggere le risorse.
- Integrato con sistemi di sicurezza come Identity per gestire utenti, ruoli e permessi.

4. Middleware:

- ASP.NET Core utilizza un pipeline middleware per processare richieste e risposte. Ogni componente nel middleware ha una funzione specifica, come logging, gestione della sessione o autenticazione.

Perché usare ASP.NET:

1. **Scalabilità:** Adatto a progetti di piccole e grandi dimensioni, con supporto nativo per lo sviluppo su cloud.
2. **Prestazioni:** ASP.NET Core è noto per la sua velocità e ottimizzazione, soprattutto in ambienti cloud.
3. **Cross-platform:** ASP.NET Core può essere eseguito su Windows, Linux e macOS, garantendo flessibilità.
4. **Supporto di Microsoft:** Essendo sviluppato da Microsoft, ha un forte supporto e una vasta documentazione.
5. **Sicurezza integrata:** Supporta tecniche avanzate per proteggere le applicazioni web, come l'autenticazione a più fattori e la protezione contro attacchi XSS e CSRF.

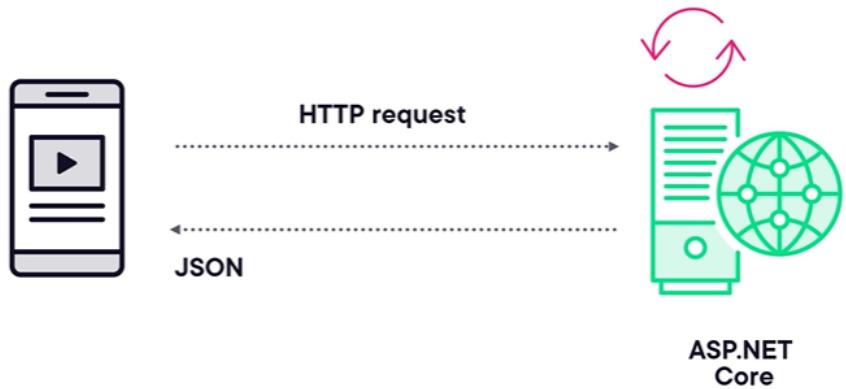
What Can We Build with ASP.NET Core?



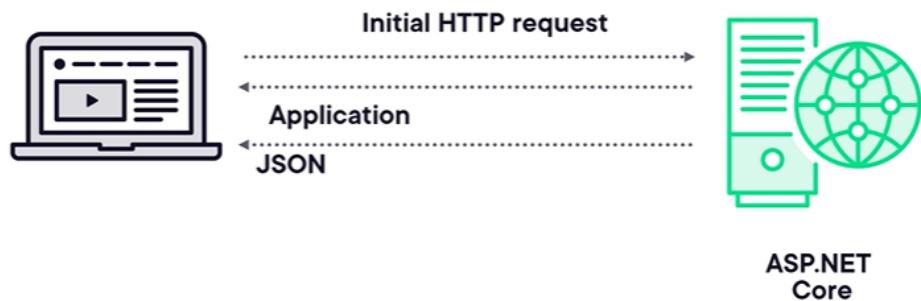
Server-side Rendered Applications



ASP.NET Core Services



Client-side Rendered Applications



Setting up ASP.NET Core MVC Application

- Apprendimento
- Obiettivo
- Repository
- Prerequisiti: HTML, CSS, C#, WebDevelopment
- Preparazione e analisi della soluzione
 - Getting your machine ready: Creazione del progetto per Windows, MacOS e Linux. (Interfaccia || CLI)
 - Exploring the generated files
 - Configure the Site (Program.cs, Kestrel): Teoria, già presente durante la creazione
 - Configuration of the Application (Service Registration, Middleware): Teoria
 - **Configuration of the Application: DEMO**

Apprendimento:

(Questo corso è presente su PluralSight by Gill Cleeren)



- Learn how to build a full web application using ASP.NET Core
- Razor, navigation, model binding, tag helpers, view components, forms
- Apply application frameworks: ASP.NET Core MVC, Razor Pages, and Blazor
- Build an API using ASP.NET Core and use it from JavaScript
- Secure the application using ASP.NET Core Identity

Obiettivo: Creation of a modern web applications using ASP.NET Core.

You'll Want to Create Dynamic Content



List of products



Shopping cart



Authenticate

Repository:

<https://github.com/marcoWarrior/BethanysPieShop>

Preparazione e analisi della soluzione:

E' necessario un **framework** web che generi HTML per i nostri utenti, esso verrà eseguito sul server (quindi sul **back-end**), perciò utilizzerò **ASP.NET Core Framework**.

Getting your machine ready:

Visual Studio 2022

.NET 8

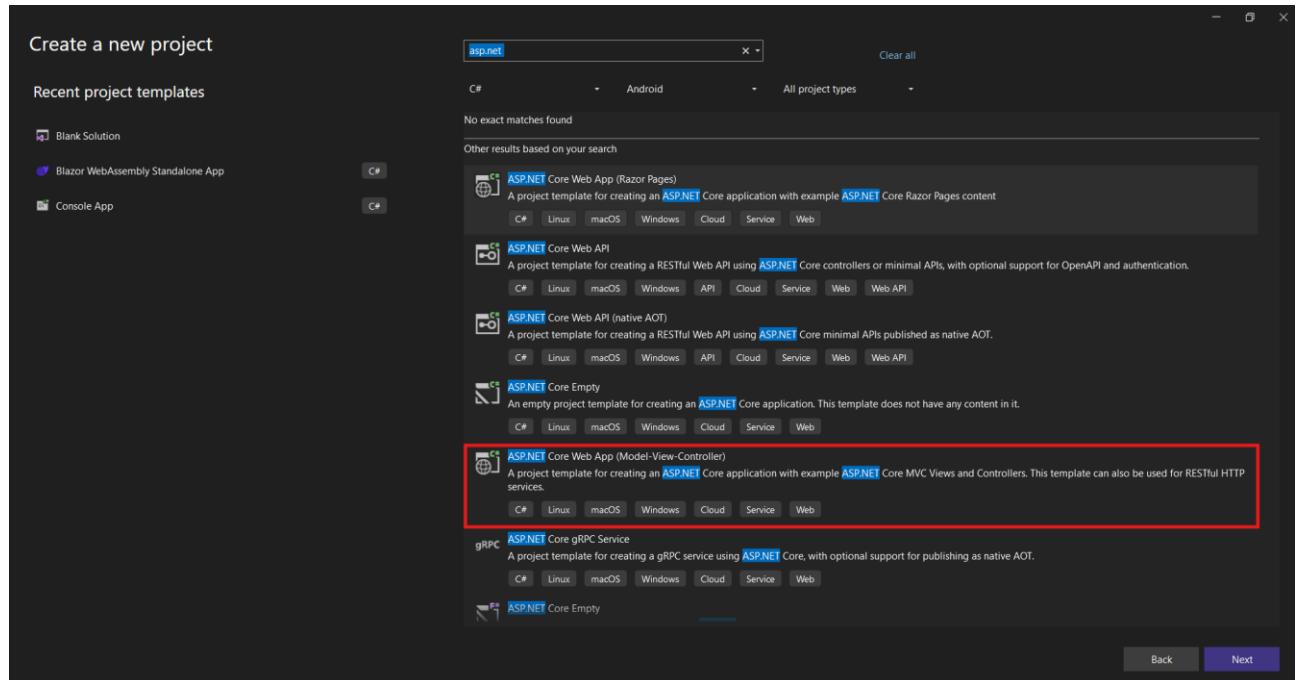
Setting up an ASP.NET Core MVC Application

- Creating a new project
- Exploring the generated files
- Configuring the site
- How ASP.NET Core handles a request

Creating a new project

TRAMITE INTERFACCIA VISUAL STUDIO (Windows)

Home



Configure your new project

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Project name

Location ...

Solution name ⓘ

Place solution and project in the same directory

Project will be created in "C:\Users\m.guerriero\source\repos\PluralSightRepo\ASP.NET-Core-Fundamentals\SampleApplication\SampleApplication\"

Additional information

ASP.NET Core Web App (Model-View-Controller)

C# Linux macOS Windows Cloud Service Web

Framework [i](#)
.NET 8.0 (Long Term Support)

Authentication type [i](#)
None

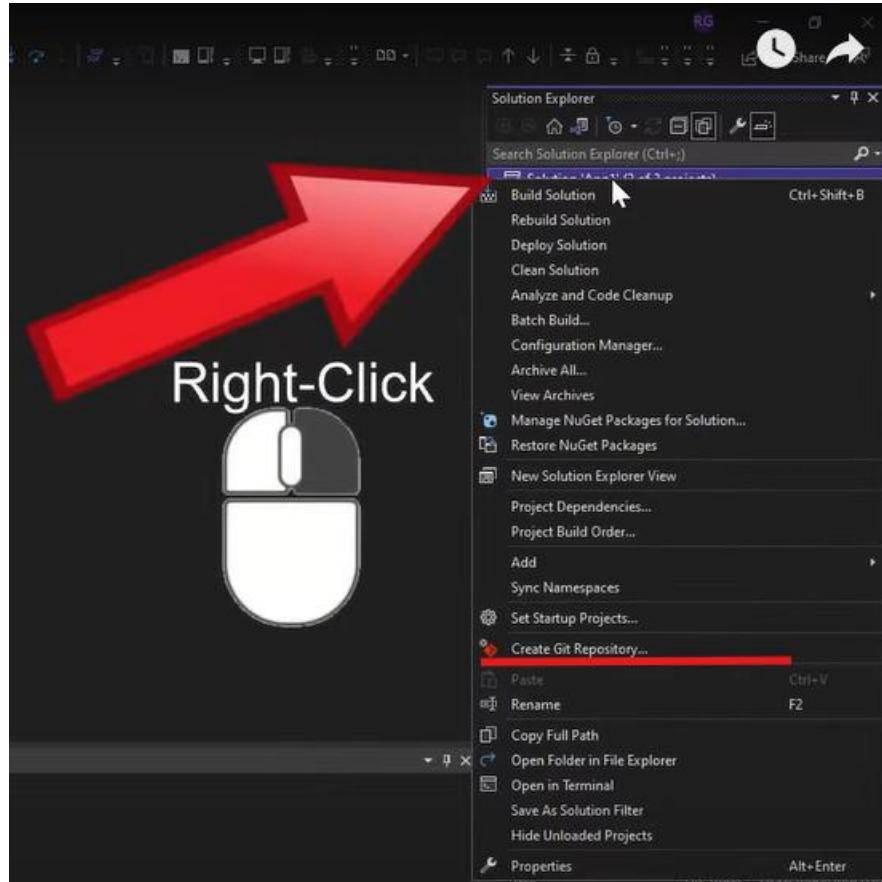
Configure for HTTPS [i](#)
 Enable container support [i](#)

Container OS [i](#)
Linux

Container build type [i](#)
Dockerfile

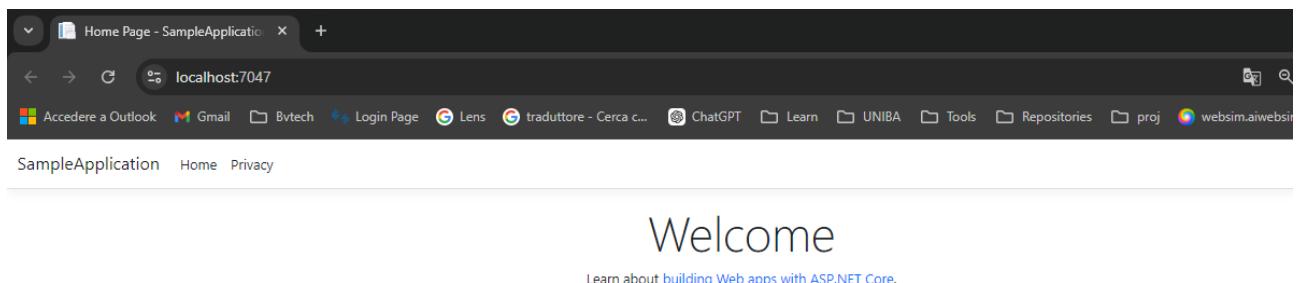
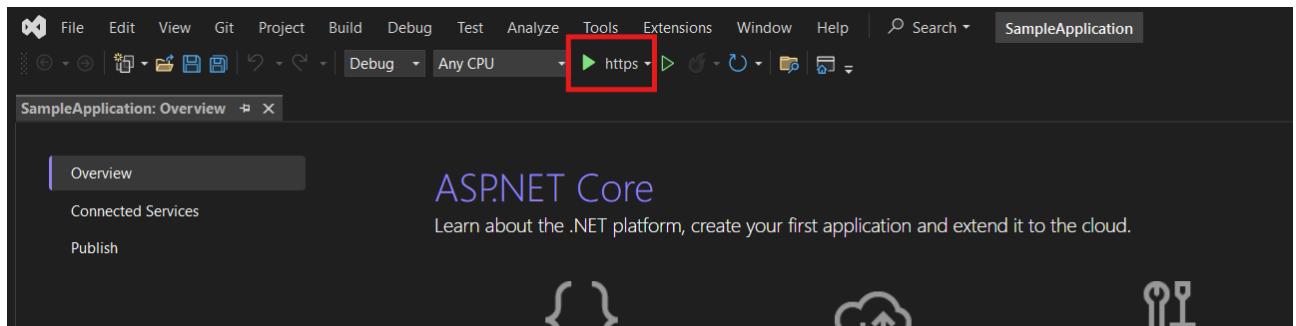
Do not use top-level statements [i](#)
 Enlist in .NET Aspire orchestration [i](#)

Consiglio: Creare una repository GitHub da Visual Studio nel seguente modo:



Home

La nostra applicazione è già pronta per essere eseguita:



Inoltre è già responsive.

SE AVESSI VOLUTO CREARE TALE PROGETTO DA CLI:

[Home](#)

```
PS D:\code\asp> mkdir DemoFromCli

    Directory: D:\code\asp

Mode                LastWriteTime         Length Name
----                -----          ----- 
d-----        2/4/2024 5:47 PM            DemoFromCli

PS D:\code\asp> dotnet new
The 'dotnet new' command creates a .NET project based on a template.

Common templates are:
Template Name      Short Name  Language  Tags
-----            -----       -----      -----
Blazor Web App     blazor      [C#]       Web/Blazor/WebAssembly
Class Library       classlib    [C#],F#,VB Common/Library
Console App        console     [C#],F#,VB Common/Console
Windows Forms App  winforms   [C#],VB    Common/WinForms
WPF Application    wpf        [C#],VB    Common/WPF

An example would be:
dotnet new console

Display template options with:
dotnet new console -h
Display all installed templates with:
dotnet new list
Display templates available on NuGet.org with:
dotnet new search web
```

Home

The screenshot shows a web browser displaying the Microsoft Learn documentation for .NET default templates. The search bar at the top contains the query "dotnet new". The left sidebar has a tree view of command references, and the right side lists various project templates. The "ASP.NET Core Web App (Model-View-Controller)" template is highlighted with a red box and a cursor is hovering over it. Below the list, a banner reads "NET Core MVC come fatto fatto nella demo precedente utilizzando dotnet new mvc.".

Name	Type	Language	Description
ASP.NET Core Test Project	nunit	[C#], F#, VB	Test/NUnit
ASP.NET Core Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
Blazor Server App Empty	blazorserver-empty	[C#]	Web/Blazor
Blazor Web App	blazor	[C#]	Web/Blazor
Blazor WebAssembly App	blazorwasm	[C#]	Web/Blazor/WebAssembly
Blazor WebAssembly App Empty	blazorwasm-empty	[C#]	Web/Blazor/WebAssembly
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp, razor	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
Razor Class Library	razorclasslib	[C#]	Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	webapi	[C#], F#	Web/WebAPI/Service/WebAPI

Dopo aver ricercato il tipo di architettura, nel nostro caso **mvc**:

The screenshot shows a Windows PowerShell window with the following command history:

```
PS D:\code\asp> cd ..\DemoFromCli\
PS D:\code\asp\DemoFromCli> dotnet new mvc
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/aspnetcore/8.0-third-party-notices
for details.

Processing post-creation actions...
Restoring D:\code\asp\DemoFromCli\DemoFromCli.csproj:
  Determining projects to restore...
  Restored D:\code\asp\DemoFromCli\DemoFromCli.csproj (in 41 ms).
Restore succeeded.

PS D:\code\asp\DemoFromCli> dir
```

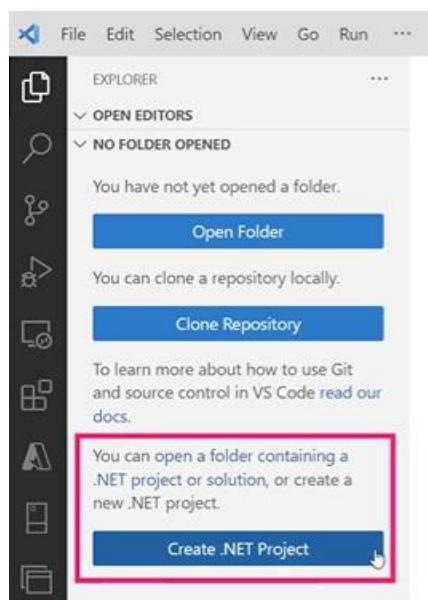
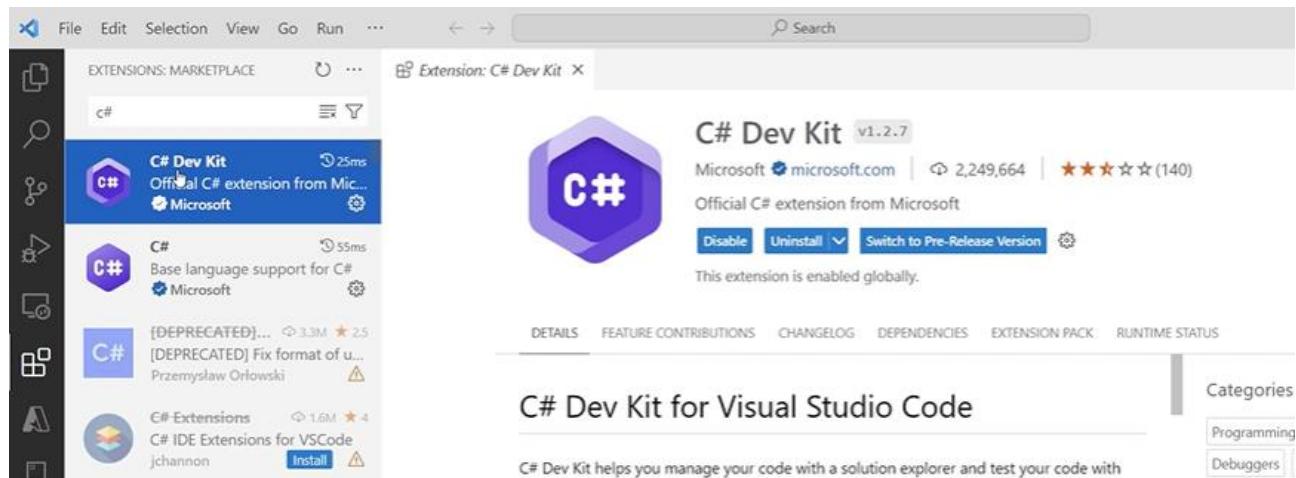
Directory: D:\code\asp\DemoFromCli

Mode	LastWriteTime	Length	Name
d----	2/4/2024 5:49 PM	127	appsettings.Development.json
d----	2/4/2024 5:49 PM	151	appsettings.json
d----	2/4/2024 5:49 PM	219	DemoFromCli.csproj
-a---	2/4/2024 5:49 PM	670	Program.cs
è stata effettivamente creata una nuova applicazione.			

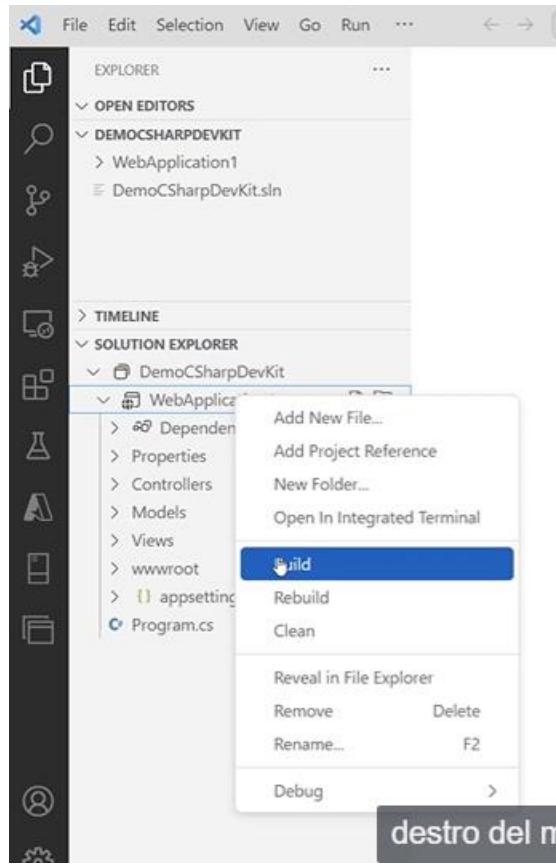
Successivamente con *dotnet run* eseguiamo l'applicazione.

[Home](#)

SE AVESSIMO USATO VSC (quindi MacOs || Linux)



[Home](#)

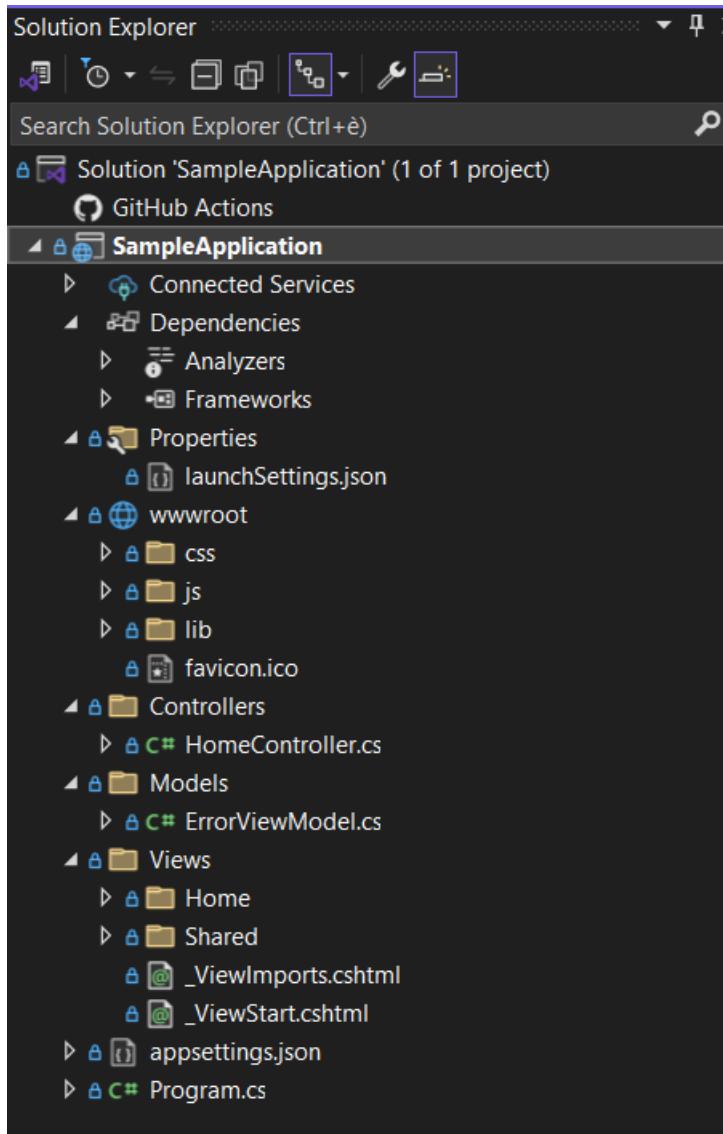


Per avviarlo basta cliccare Debug (voce presente nell'immagine precedente)

Exploring the generated files

Una soluzione raggruppa più progetti

[Home](#)



Viene creata una soluzione e un progetto al suo interno.

Servizi connessi ci consentono di fare riferimento ad altri servizi

Le dipendenze elencheranno tutti i riferimenti che abbiamo nel nostro progetto (es: NuGet packages)

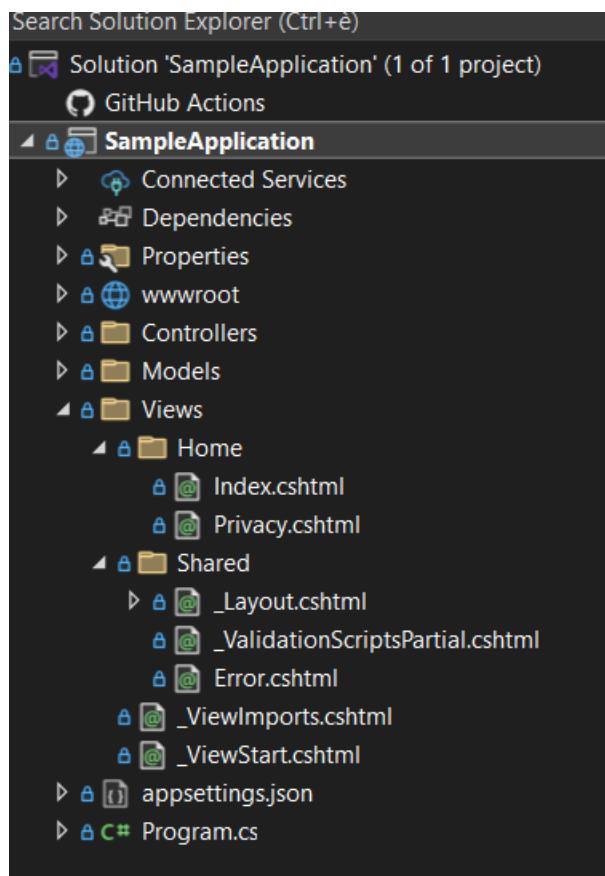
Properties contiene un file JSON chiamato launchsetting.json (lo esamineremo nella DEMO)

Program.cs contiene la classe program, ovvero il fulcro

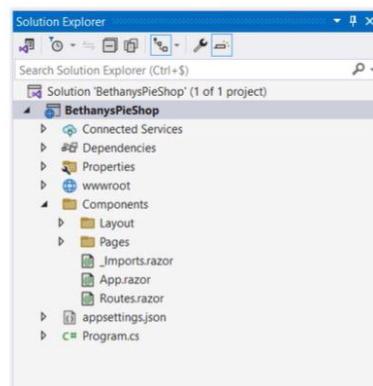
appsetting.json, impostazione per archiviare e configurare e connettere al DB

Razor Pages:

Home



Project Structure (Blazor Server-side Rendering)



Nel wwwroot Folder sono contenuti i file statici, non saranno accessibili:

The wwwroot Folder



wwwroot/image1.jpg

<http://bethanyspieshop.com/image1.jpg>

The csproj File

```
<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>

</Project>
```

Adding Dependencies

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
</ItemGroup>

</Project>
```

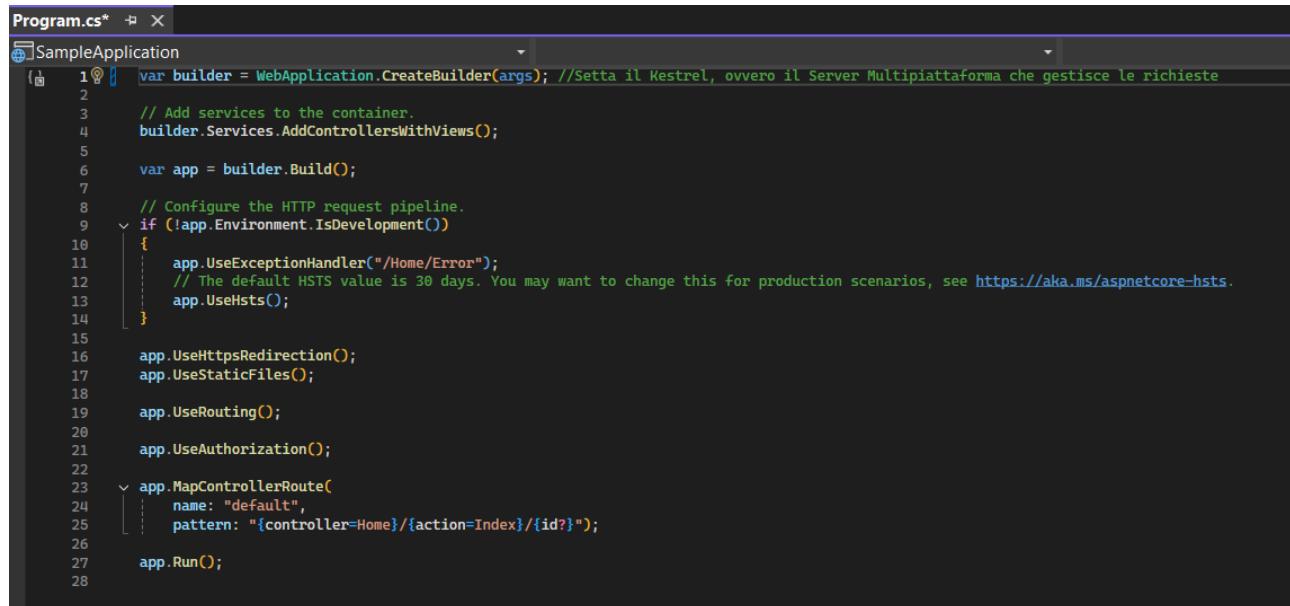
Osservazione: contiene i riferimenti disponibili nel Framework, per sempio NuGet.

Configure the Site

Program.cs

E' la classe che si occuperà di avviare il Server, per ascoltare le richieste in arrivo e configurare l'applicazione.

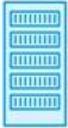
Il tutto avverrà proprio all'interno di ASP.NET Core, più precisamente nel Kestrel



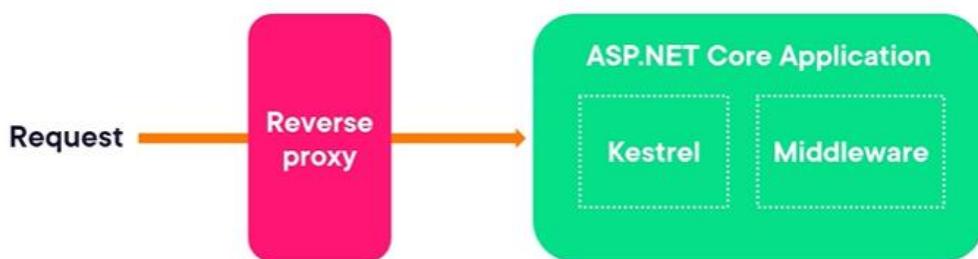
```
Program.cs*  ⌂ X
SampleApplication
1 var builder = WebApplication.CreateBuilder(args); //Setta il Kestrel, ovvero il Server Multipiattaforma che gestisce le richieste
2
3 // Add services to the container.
4 builder.Services.AddControllersWithViews();
5
6 var app = builder.Build();
7
8 // Configure the HTTP request pipeline.
9 if (!app.Environment.IsDevelopment())
10 {
11     app.UseExceptionHandler("/Home/Error");
12     // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
13     app.UseHsts();
14 }
15
16 app.UseHttpsRedirection();
17 app.UseStaticFiles();
18
19 app.UseRouting();
20
21 app.UseAuthorization();
22
23 app.MapControllerRoute(
24     name: "default",
25     pattern: "{controller=Home}/{action=Index}/{id?}");
26
27 app.Run();
28
```

L'istruzione principale, ovvero `WebApplication.CreateBuilder(args)` serve per:

The CreateBuilder Method

-  Set up Kestrel server
-  Configure IIS integration
-  Specify content root
-  Read application settings

Sidestep: Running a Server with Kestrel



In fase di sviluppo, puoi usare Kestrel direttamente, ma per ambienti di produzione è altamente raccomandato l'uso di un reverse proxy.

[Home](#)

Note: se utilizzi Kestrel da solo, non c'è automaticamente un reverse proxy a filtrare le richieste. Se desideri che le richieste vengano filtrate o gestite da un reverse proxy, dovrai avere una file web.config

il file **web.config** non è presente perché non è ancora necessario. Verrà creato automaticamente quando pubblicherai l'applicazione per **IIS**.

IIS (Internet Information Services) è un **server web** sviluppato da Microsoft per ospitare applicazioni web su **Windows**. Gestisce richieste HTTP e HTTPS, fornendo servizi come:

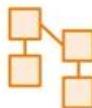
- **Hosting di siti web** e applicazioni (ASP.NET, PHP, etc.).
- **Reverse proxy** per applicazioni backend come ASP.NET Core (usando Kestrel).
- **Gestione SSL**, bilanciamento del carico e sicurezza.

È ampiamente utilizzato in ambienti **Windows Server** per applicazioni aziendali.

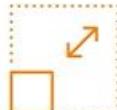
Configuration of the Application



Service Registration



All classes used by the application



More modular approach



More dependencies will need to be injected

Service registration:

Tutte le classi che possiamo utilizzare nella nostra applicazione vengono chiamate **SERVIZI**

Codice modulare che utilizza piccole classi, queste classi dovranno essere istanziate e qui si utilizza la registrazione del servizio.

Ipotizziamo di avere la seguente situazione:

AController: Classe che dovrà eseguire alcune registrazioni, ci riferiamo alla funzionalità di registrazione di una classe con un servizio

chiamato LoggerService

LoggerService: Questo servizio sa come eseguire la registrazione a tutte le classi che ora desiderano registrare qualcosa possono creare un'istanza del servizio di regisra e lasciarle eseguire la registrazione.

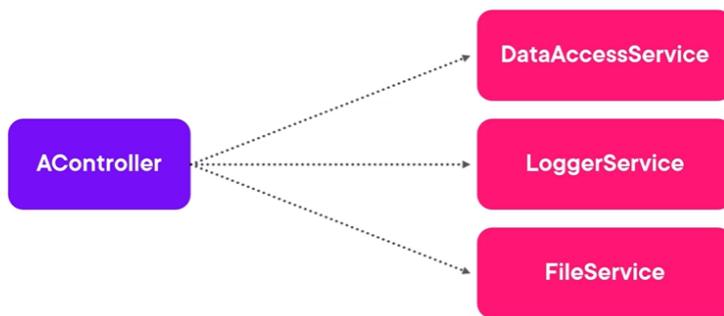
Using Services



Il controller ha la responsabilità di creare una istanza di LoggerService,

Vorrà utilizzare anche altri servizi come:

Initializing Dependencies

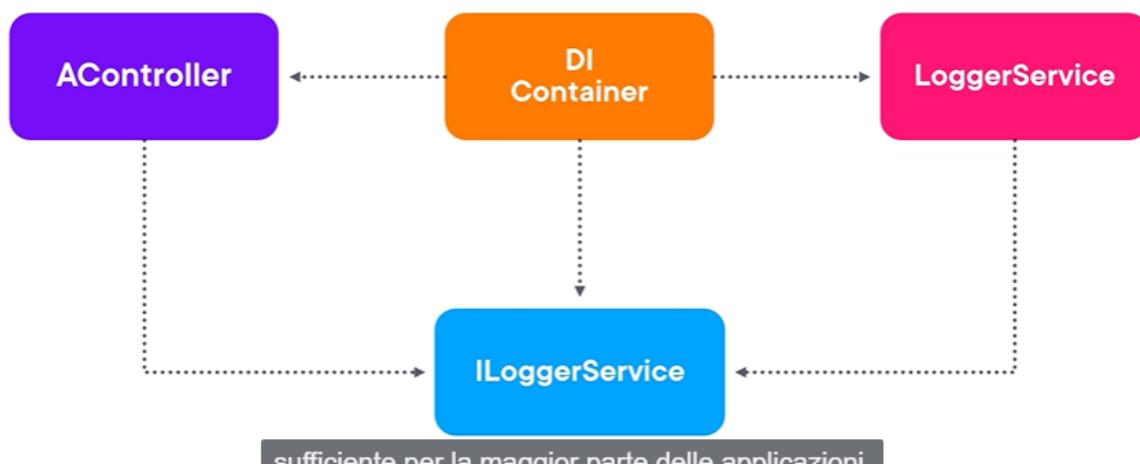


Quindi quando volgiore creare una nuova istanza di Acontroller sarò responsabile de tutte questa altre istanze (che sono pesanti)

Un buon approccio è quello delle **Dependency Injection (DI)**:

Una 3° parte ILoggerService crea le istanze:

Introducing Dependency Injection (DI)



Dove nel nostro Program.cs vogliamo registrare servizi, sappiamo che builder è un tipo di web application builder, **che espone la collection dei servizi, a questa raccolta possiamo aggiungere i nostri servizi:**

Registering Services

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddScoped<ILoggerService, LoggerService>();

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

Framework services

Custom services

Una volta registrato, dove ora abbiamo bisogno di un'istanza, la faremo **INIETTARE OrderController** usa **ILoggerService**, il servizio che abbiamo registrato nel nostro contenitore:

```
public class OrderController : Controller
{
    private readonly ILoggerService _loggerService;

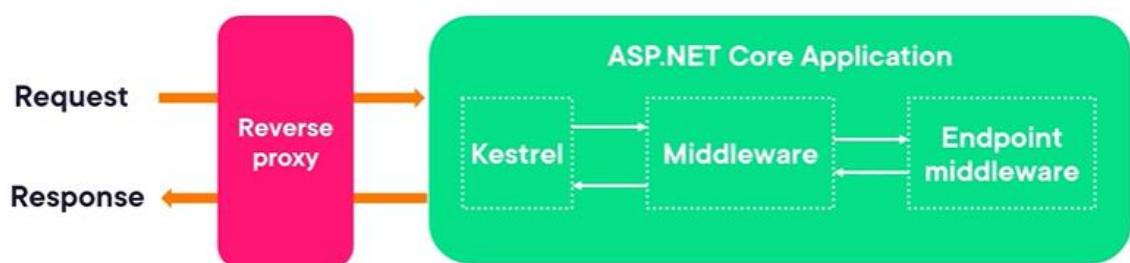
    public OrderController(ILoggerService loggerService)
    {
        _loggerService = loggerService;
    }
}
```

Using Services

Injected via constructor

Quindi, la registrazione dei servizi era uno dei compiti del Program.cs, un altro compito o il secondo è quello della creazione del **middleware**:

Handling Requests with Middleware



The Middleware Request Pipeline



Pipeline consists out of set of components



Components work on request or response

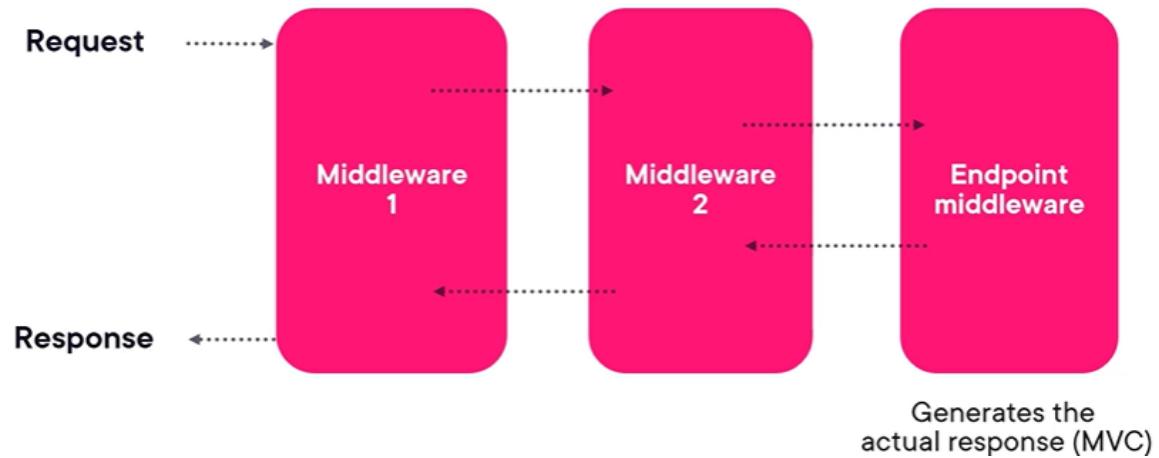


Many built-in components



Endpoint middleware sits at the end

The Middleware Request Pipeline



Middleware nel Program.cs:

Middleware Request Pipeline

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllersWithViews();

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

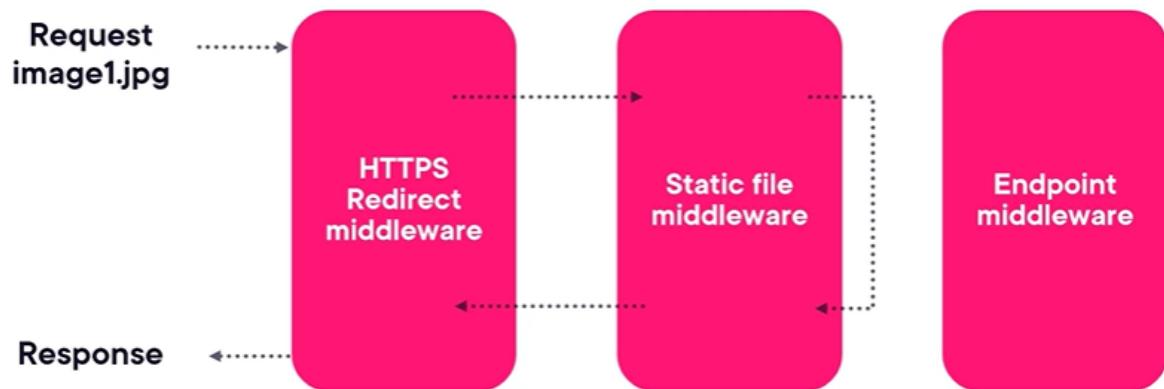
tutti questi sono metodi che collegano il middleware.

Nel nostro caso abbiamo scelto di lavorare con Static file middleware, che sono quelli posizionati nella wwwroot.

Exploring Static File Middleware



Exploring Static File Middleware



La richiesta non proseguirà il percorso verso la pipeline.

Osservazione: La richiesta non prosegue, in quanto ha trovato già l'immagine nella sezione dei file statici, presente nella seconda componente del Middleware.

Ne consegue l'importanza dell'ordine dei componenti del Middleware e lo studio della documentazione, nell'immagine precedente il modello MVC presente nell'Endpoint Middleware non è nemmeno a conoscenza di ciò che è avvenuto, ciò ha migliorato le prestazioni del programma.

[Home](#)

In ASP.NET 6 e quindi in modelli più vecchi, Program.cs era diviso in due parte, quindi era presente anche Startup.cs:

What About the Old Model?

```
Program.cs
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
    }

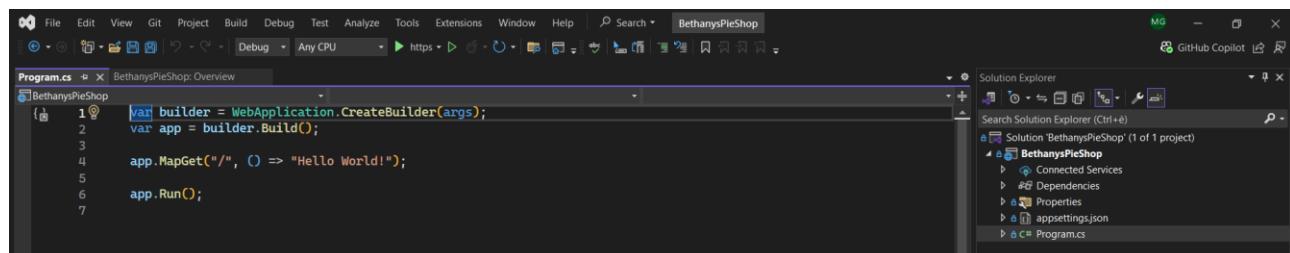
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        ...
    }
}
```

middleware è stata effettivamente eseguita in una startup.

Configuration of the Application: DEMO

La demo mostra come replicare la base ASP.NET core MVC partendo da un progetto vuoto (ASP.NET core Empty)

Quindi creiamo un **nuovo progetto vuoto**:



Partiamo dal file più importante, Program.cs

Attualmente non esiste una classe, non esiste un dominio statico.

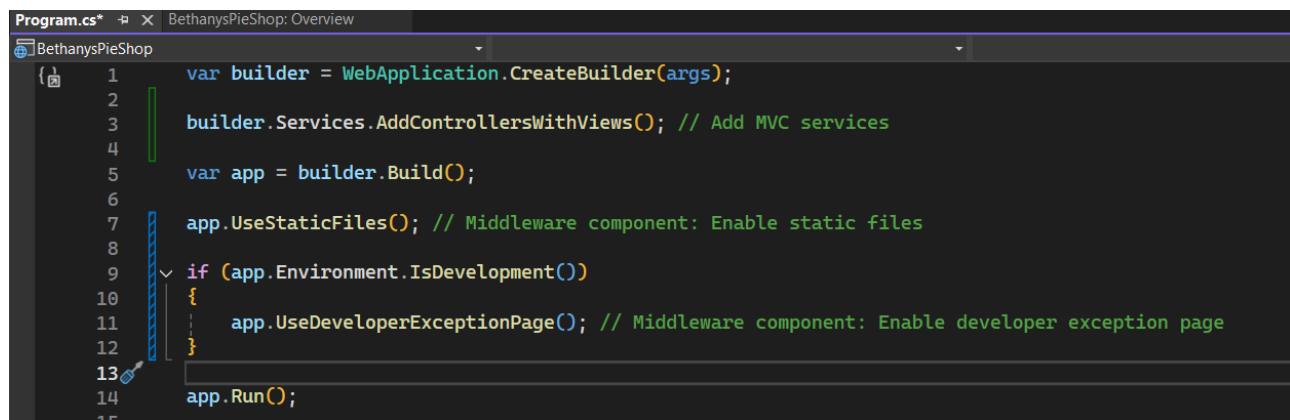
Verrà quindi eseguito come un app console

Possiamo notare che è già presente il builder, il quale svolge operazioni di configurazioni già citate come:

- caricare tutte le impostazioni presenti nel file appsettings.json;
- si assicura che il nostro Kestrel sia incluso e configurerà IIS;
- imposta la cartella wwwroot che conterrà il contenuto statico;
- ci darà anche accesso alla raccolta dei servizi dove effettuiamo la DI.

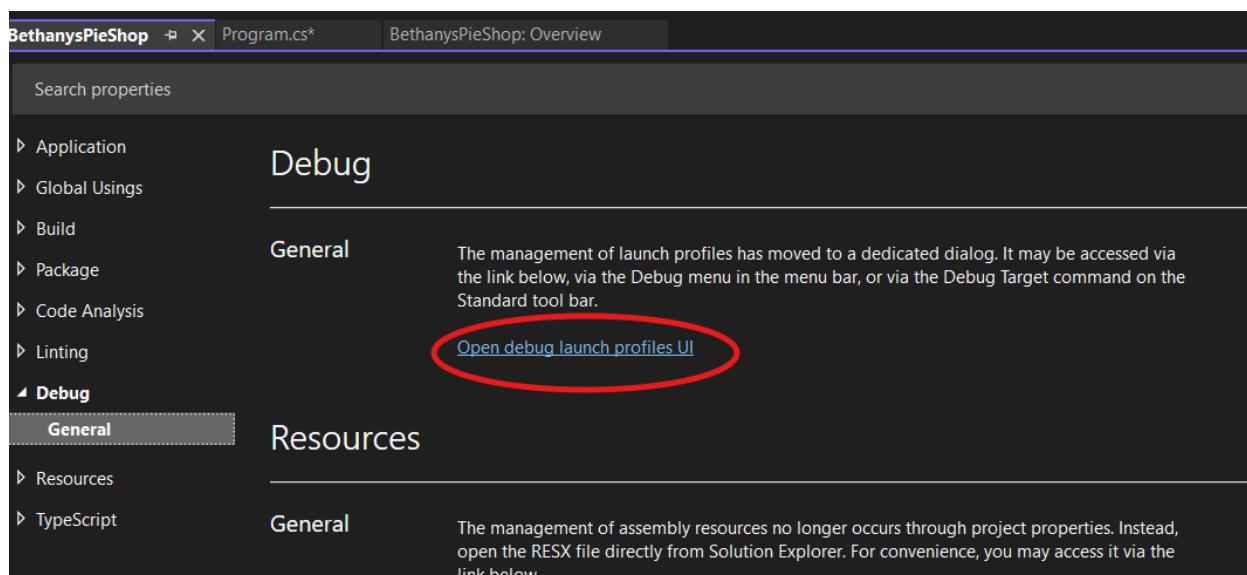
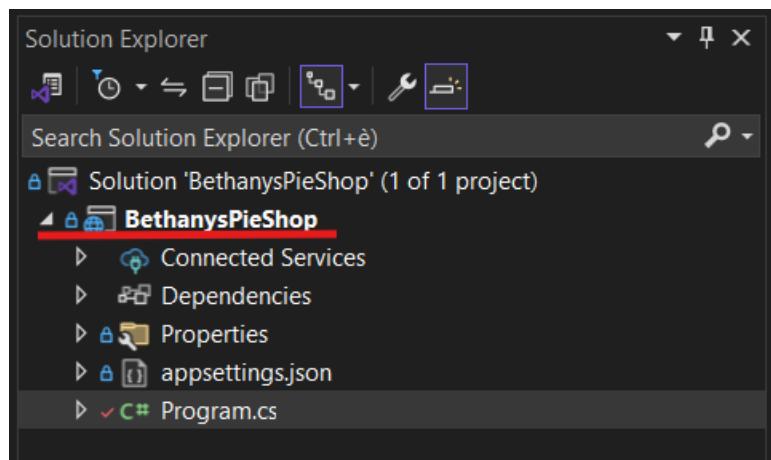
Modifichiamo il Program.cs proposto e aggiungiamo il primo servizio e due componenti middleware:

Home



```
Program.cs*  ✎ X | BethanysPieShop: Overview
BethanysPieShop
1 var builder = WebApplication.CreateBuilder(args);
2
3 builder.Services.AddControllersWithViews(); // Add MVC services
4
5 var app = builder.Build();
6
7 app.UseStaticFiles(); // Middleware component: Enable static files
8
9 if (app.Environment.IsDevelopment())
10 {
11     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
12 }
13
14 app.Run();
15
```

Per verificare l'ambiente di esecuzione click destro sulle proprietà del progetto, *Properties > Debug > Open debug launch profiles UI*.



[Home](#)

Aggiungiamo un altro componente del middleware:

```
Program.cs
1 var builder = WebApplication.CreateBuilder(args);
2
3 builder.Services.AddControllersWithViews(); // Add MVC services
4
5 var app = builder.Build();
6
7 app.UseStaticFiles(); // Middleware component: Enable static files
8
9 if (app.Environment.IsDevelopment())
10 {
11     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
12 }
13
14 app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
15
16 app.Run();
17
```

The code editor shows the `Program.cs` file from a .NET Core application named `BethanysPieShop`. The line `app.MapDefaultControllerRoute();` is highlighted with a red underline, indicating a syntax error or a warning.

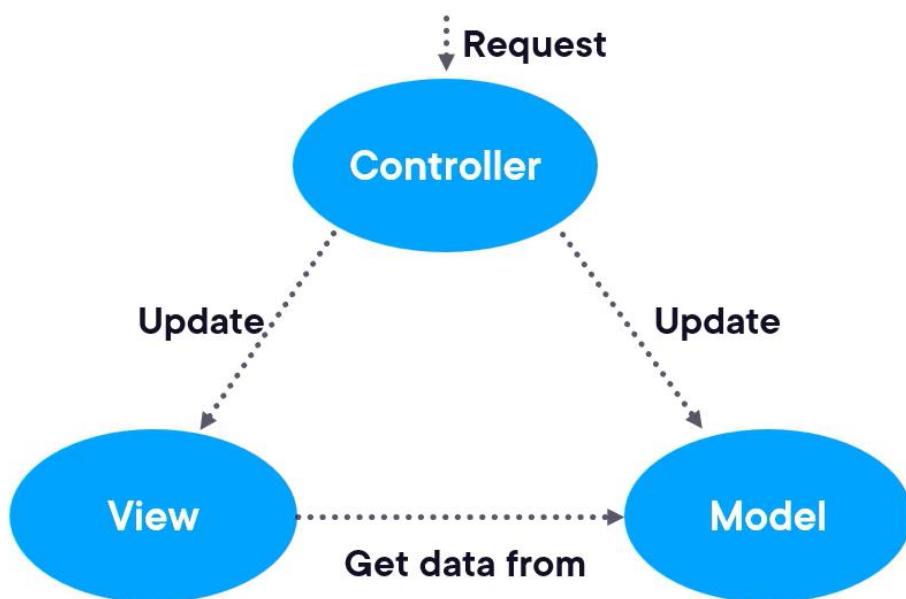
Se avviamo ora avremo un errore nel nostro client, in quanto non sono presenti ancora Views.

La documentazione continuerà in [ASP.NET Core Fundamentals 2](#)

Creating the First Page

- Introducing the MVC Pattern
- Creating the Model and the Repository
- Demo: Creating the Model and the Repository
- Creating the Controller
- Demo: Adding the Controller
- Adding the View
- Demo: Creating the First View
- Demo: Using a View Model
- Adding Extra View Files
- Demo: Adding Extra View Files
- Styling the View
- Demo: Styling the View

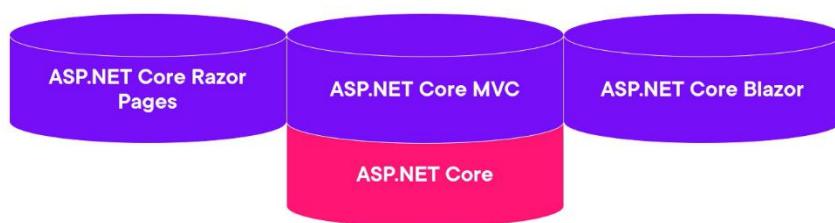
MVC Pattern



Model: Contengono i dati con cui lavora l'utente, ovvero classi del modello di dominio e le operazioni per lavorare con questi dati. Sono presenti anche classi che consentono l'interazione con il db, quindi anche i repository faranno parte del modello.

Controller: Sono il collante tra il modello e la vista, forniranno la logica che funziona sul modello, e forniranno i dati da visualizzare nella vista.

View: Conterrà la logica che mostra i dati all'utente in modo che possano, ancora una volta, essere elaborati da un'azione del controller



Creating the Model and the Repository

Sample Model Class

Nullable is used here

```
public class Pie
{
    public int PieId { get; set; }
    public string Name { get; set; }
    public string? ShortDescription { get; set; }
    public string? LongDescription { get; set; }
    public string? AllergyInformation { get; set; }
    public decimal Price { get; set; }
    public string? ImageUrl { get; set; }
    public string? ImageThumbnailUrl { get; set; }
    public bool IsPieOfTheWeek { get; set; }
    public bool InStock { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

Il repository permette al nostro codice di utilizzare oggetti senza sapere come sono stati persistiti.

Creeremo un'interfaccia e la sua implementazione.

La classe repository si occuperà della persistenza

```
public interface IPieRepository
{
    IEnumerable<Pie> AllPies { get; }
    Pie GetPieById(int pieId);
}
```

The IPieRepository Interface

Registering the Repository



Registering Services

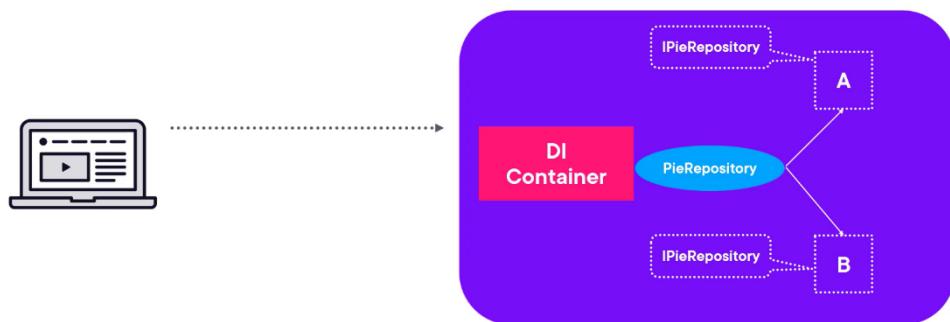
AddTransient

AddSingleton

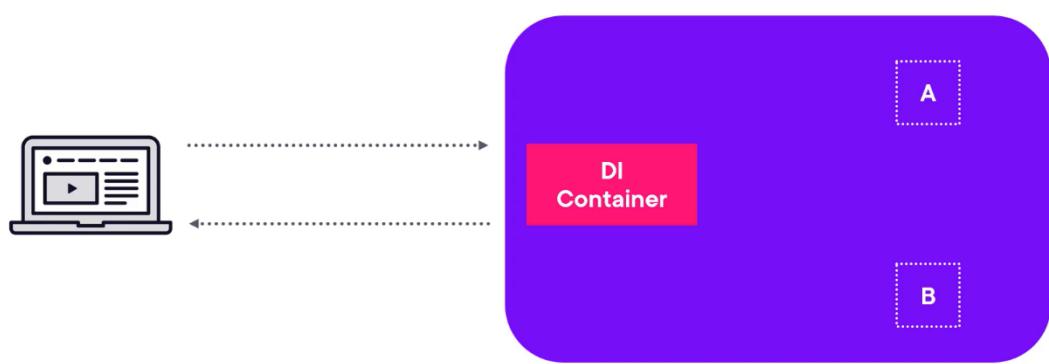
AddScoped

- **AddTransient:** Una nuova istanza per **ogni utilizzo**. (Quando il servizio **non mantiene stato** e deve essere leggero. Ideale per servizi stateless, come utility o helper.)
- **AddSingleton:** **Una sola istanza** per l'intera applicazione. (Quando il servizio deve **mantenere uno stato condiviso** e la creazione di più istanze sarebbe inefficiente, ad esempio per cache o impostazioni globali.)
- **AddScoped:** Una nuova istanza per **ogni richiesta HTTP**. (Quando il servizio ha bisogno di uno stato per la durata di una singola richiesta, ma non oltre. Ideale per servizi che interagiscono con il database o gestiscono transazioni all'interno di una richiesta HTTP.)

Understanding AddScoped



B utilizzerà la stessa istanza PieRepository, quando la richiesta viene gestita e la viene inviata al client, l'oggetto verrà scartato:



Demo: Creating the Model and the Repository

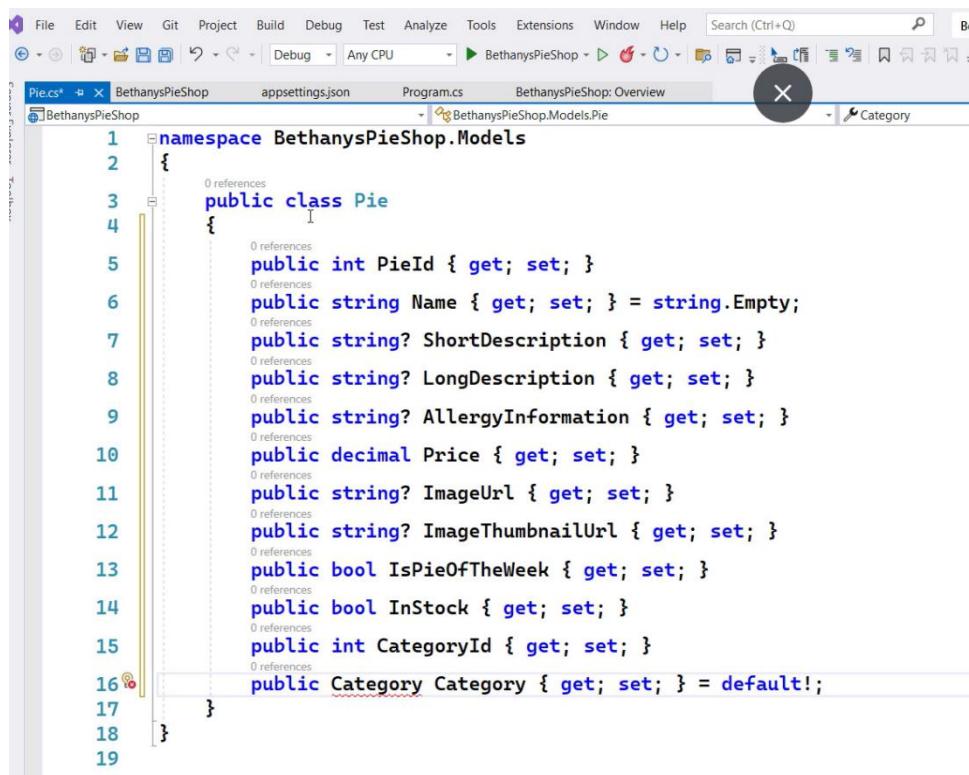
Creiamo la cartella Models dentro il progetto BethanysPieShop, e successivamente la tua classe che chiameremo Pie:

A screenshot of Visual Studio showing the code editor with Program.cs open. The code is as follows:

```
1 var builder = WebApplication.CreateBuilder(args);
2 
3 builder.Services.AddControllersWithViews();
4 
5 var app = builder.Build();
6 
7 app.UseStaticFiles();
8 
9 if (app.Environment.IsDevelopment())
10 {
11     app.UseDeveloperExceptionPage();
12 }
13 
14 app.MapDefaultControllerRoute();
15 
16 app.Run();
17 
```

The Solution Explorer on the right shows the project structure with a context menu open over the 'Models' folder. The 'Add' submenu is expanded, and the 'Class...' option is highlighted with a blue selection bar.

Home

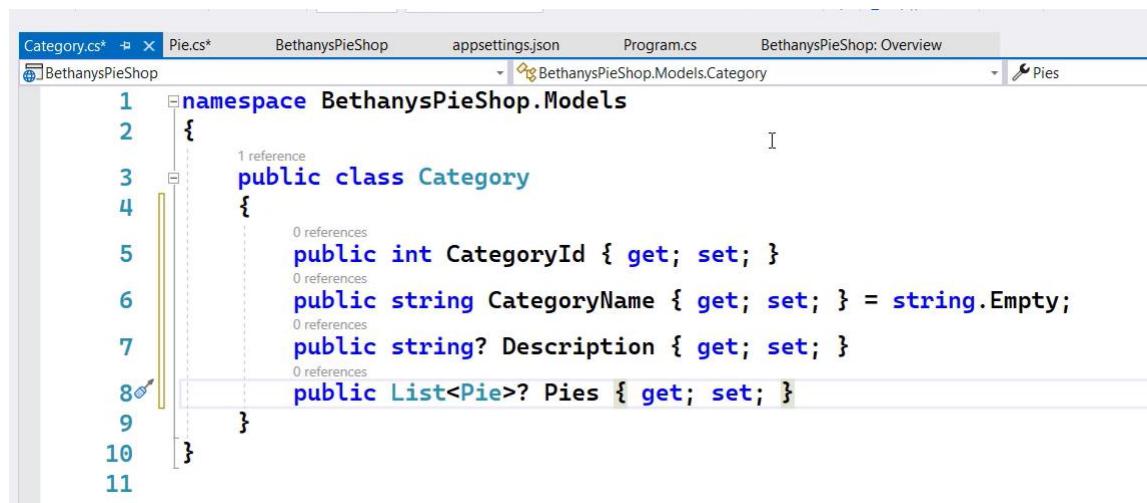


The screenshot shows the Visual Studio IDE interface with the file 'Pie.cs' open. The code defines a class 'Pie' with various properties and a constructor. A circled 'X' icon is located in the top right corner of the code editor window.

```
1 namespace BethanysPieShop.Models
2 {
3     public class Pie
4     {
5         public int PieId { get; set; }
6         public string Name { get; set; } = string.Empty;
7         public string? ShortDescription { get; set; }
8         public string? LongDescription { get; set; }
9         public string? AllergyInformation { get; set; }
10        public decimal Price { get; set; }
11        public string? ImageUrl { get; set; }
12        public string? ImageThumbnailUrl { get; set; }
13        public bool IsPieOfTheWeek { get; set; }
14        public bool InStock { get; set; }
15        public int CategoryId { get; set; }
16        public Category Category { get; set; } = default!;
17    }
18}
19
```

Osservazioni:

- E' stato creato un PieId;
- Alcune proprietà sono nullable (sulle proprietà che non richiedono un valore, in quanto nel DB potrebbero essere NULL)
- Name è settato a vuoto,
- Category ha default! "Questo operatore perdonava i valori null" , per dichiarare che quella proprietà non dovrebbe essere nulla, ora aggiungiamo la classe Category



The screenshot shows the Visual Studio IDE interface with the file 'Category.cs' open. The code defines a class 'Category' with properties and a collection of 'Pies'. A circled 'X' icon is located in the top right corner of the code editor window.

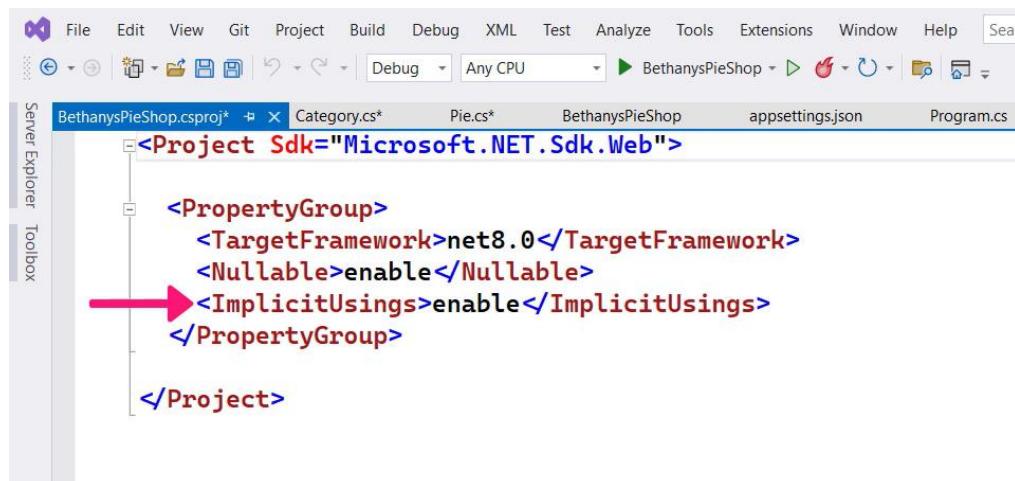
```
1 namespace BethanysPieShop.Models
2 {
3     public class Category
4     {
5         public int CategoryId { get; set; }
6         public string CategoryName { get; set; } = string.Empty;
7         public string? Description { get; set; }
8         public List<Pie>? Pies { get; set; }
9     }
10}
11
```

Osservazioni:

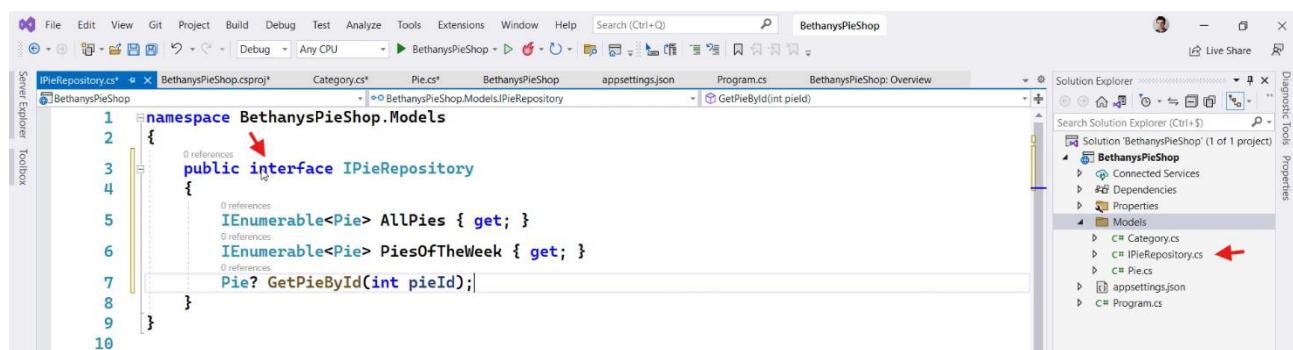
`public List<Pie>? Pies { get; set; }` potrebbe avere una categoria ma non un elenco di torte

Home

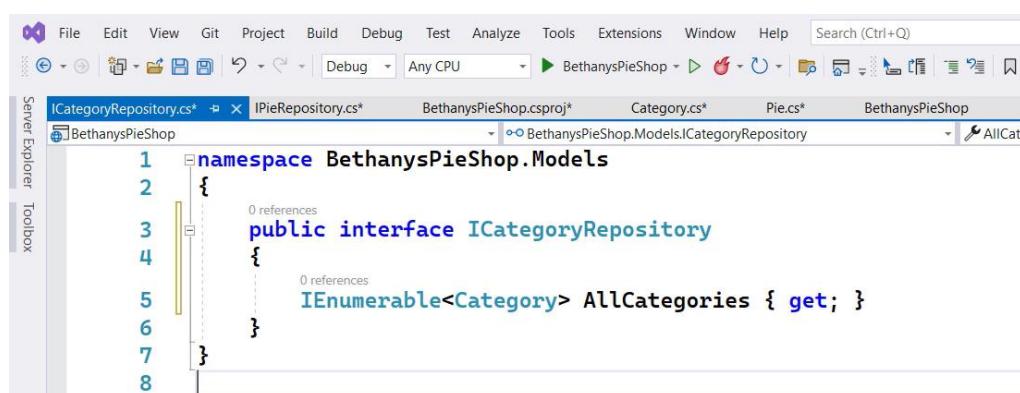
Possiamo notare che non sono presenti using all'interno delle varie classi, il motivo è questo:



Ora creiamo IPieRepository.cs che verrà utilizzato per racchiudere tutta la logica di interazione dei dati e assicurarsi che le classi che utilizzano nella mia applicazione stiano semplicemente comunicando con questo repository:

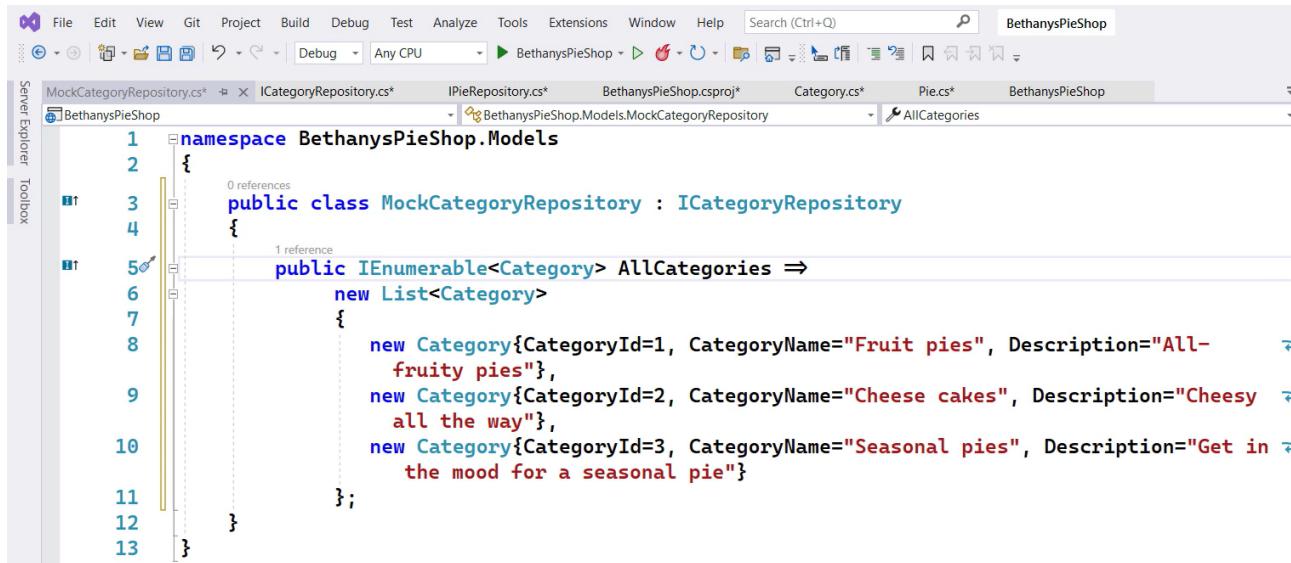


Procediamo in egual modo anche per l'interfaccia Category, creando ICategoryRepository.cs:



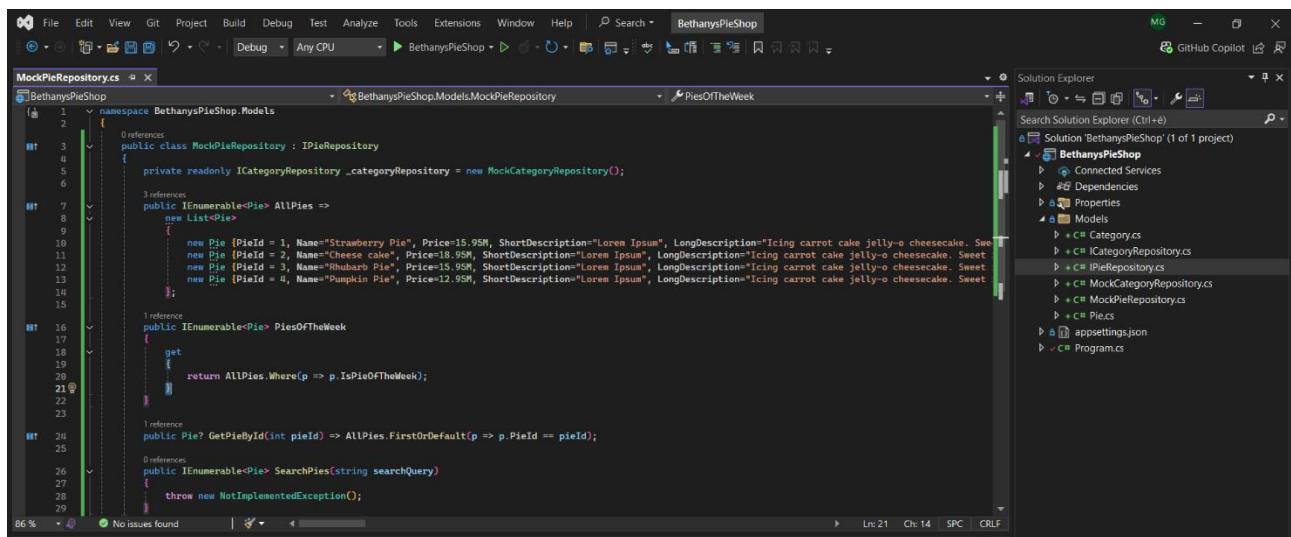
Home

Ora utilizzeremo una mock class per poter simulare il DB, ovvero i dati che recuperiamo dal database. Quindi creiamo MockPieCategory.cs che implementerà ICategoryRepository:



```
namespace BethanysPieShop.Models
{
    public class MockCategoryRepository : ICategoryRepository
    {
        public IEnumerable<Category> AllCategories =>
            new List<Category>
            {
                new Category{CategoryId=1, CategoryName="Fruit pies", Description="All-fruity pies"},
                new Category{CategoryId=2, CategoryName="Cheese cakes", Description="Cheesy all the way"},
                new Category{CategoryId=3, CategoryName="Seasonal pies", Description="Get in the mood for a seasonal pie"}
            };
    }
}
```

Procediamo in egual modo anche per il mock di Pie, creando quindi MockPieRepository.cs



```
namespace BethanysPieShop.Models
{
    public class MockPieRepository : IPieRepository
    {
        private readonly ICategoryRepository _categoryRepository = new MockCategoryRepository();

        public IEnumerable<Pie> AllPies =>
            new List<Pie>
            {
                new Pie {PieId = 1, Name="Strawberry Pie", Price=15.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet", Sweet=true},
                new Pie {PieId = 2, Name="Cheese cake", Price=18.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet", Sweet=false},
                new Pie {PieId = 3, Name="Rhubarb Pie", Price=15.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet", Sweet=false},
                new Pie {PieId = 4, Name="Pumpkin Pie", Price=12.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet", Sweet=true}
            };

        public IEnumerable<Pie> PiesOfTheWeek
        {
            get
            {
                return AllPies.Where(p => p.IsPieOfTheWeek);
            }
        }

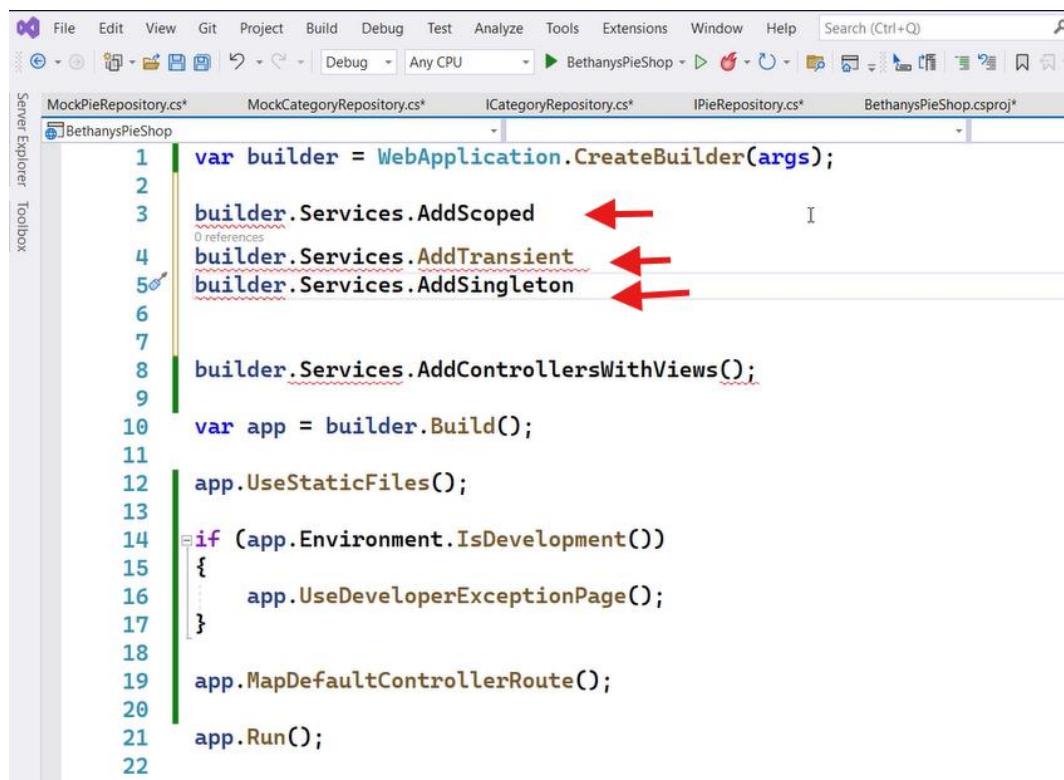
        public Pie? GetPieById(int pieId) => AllPies.FirstOrDefault(p => p.PieId == pieId);

        public IEnumerable<Pie> SearchPies(string searchQuery)
        {
            throw new NotImplementedException();
        }
    }
}
```

Fatto ciò, dobbiamo comunicarli alla nostra applicazione in modo da poterli utilizzare in combinazione con l'inserimento delle dipendenze. Dobbiamo quindi registrare queste classi mentre sono servizi con la raccolta Services, quindi ritorniamo in program.cs:

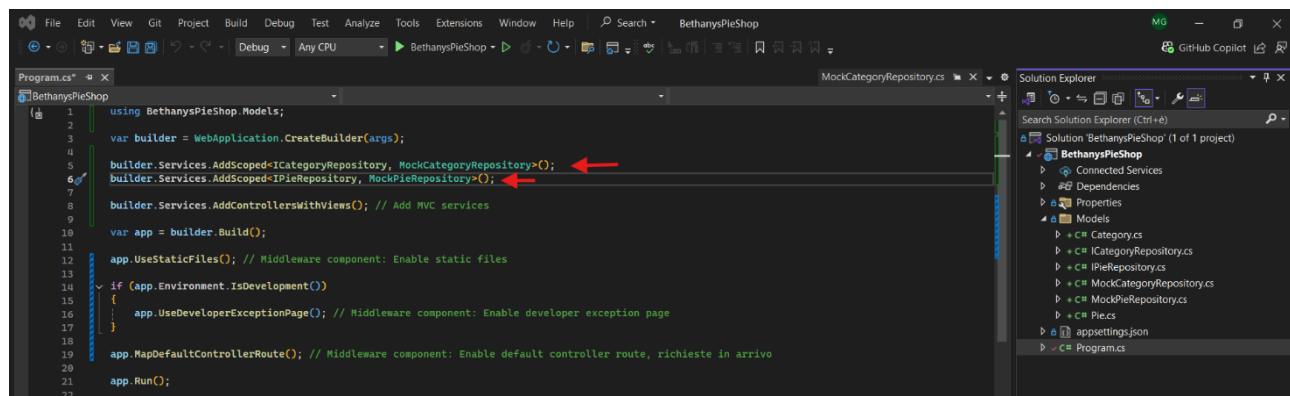
Ricordandoci che possiamo utilizzare tre operazioni che permettono l'aggiunta di un servizio:

Home



```
1 var builder = WebApplication.CreateBuilder(args);
2
3 builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>(); ←
4 builder.Services.AddTransient<IPieRepository, MockPieRepository>(); ←
5 builder.Services.AddSingleton<IPieRepository, MockPieRepository>(); ←
6
7
8 builder.Services.AddControllersWithViews();
9
10 var app = builder.Build();
11
12 app.UseStaticFiles();
13
14 if (app.Environment.IsDevelopment())
15 {
16     app.UseDeveloperExceptionPage();
17 }
18
19 app.MapDefaultControllerRoute();
20
21 app.Run();
22
```

In questo caso utilizzeremo **AddScoped**:



```
1 using BethanyPieShop.Models;
2
3 var builder = WebApplication.CreateBuilder(args);
4
5 builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>(); ←
6 builder.Services.AddScoped<IPieRepository, MockPieRepository>(); ←
7
8
9 builder.Services.AddControllersWithViews(); // Add MVC services
10
11 var app = builder.Build();
12
13 app.UseStaticFiles(); // Middleware component: Enable static files
14
15 if (app.Environment.IsDevelopment())
16 {
17     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
18 }
19
20 app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
21
22 app.Run();
23
```

Creating the Controller

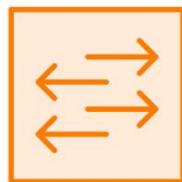
The Controller



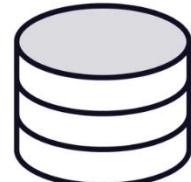
Central role



Respond to user interaction



Interact with model and select view



No knowledge of data persistence

A Basic Controller

```
public class PieController : Controller
{
    public ViewResult Index() <----- Action
    {
        return View();
    }
}
```

A Real Controller

```
public class PieController : Controller
{
    private readonly IPieRepository _pieRepository;

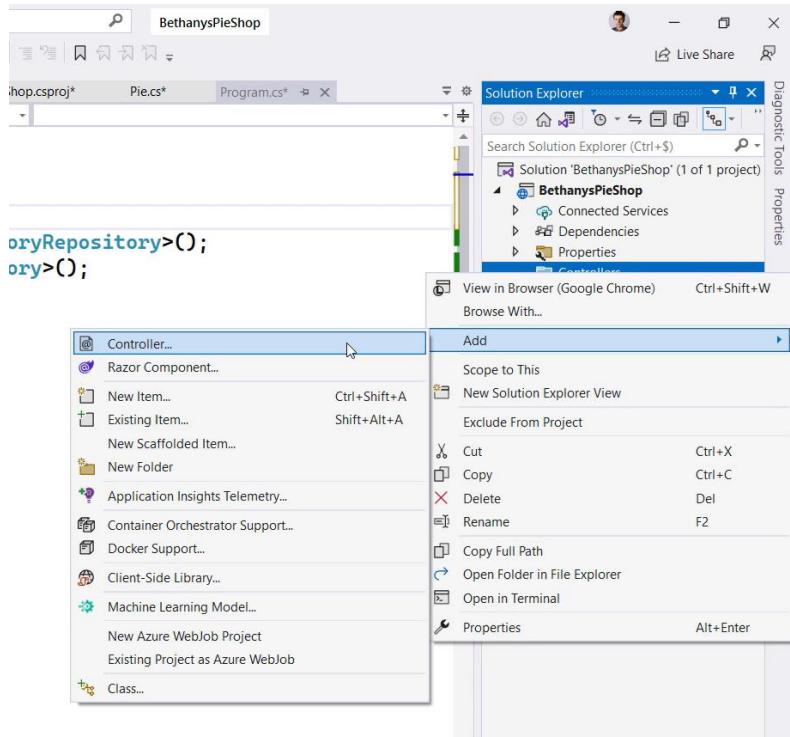
    public PieController(IPieRepository pieRepository)
    {
        _pieRepository = pieRepository;
    }

    public ViewResult List()
    {
        return View(_pieRepository.Pies);
    }
}
```

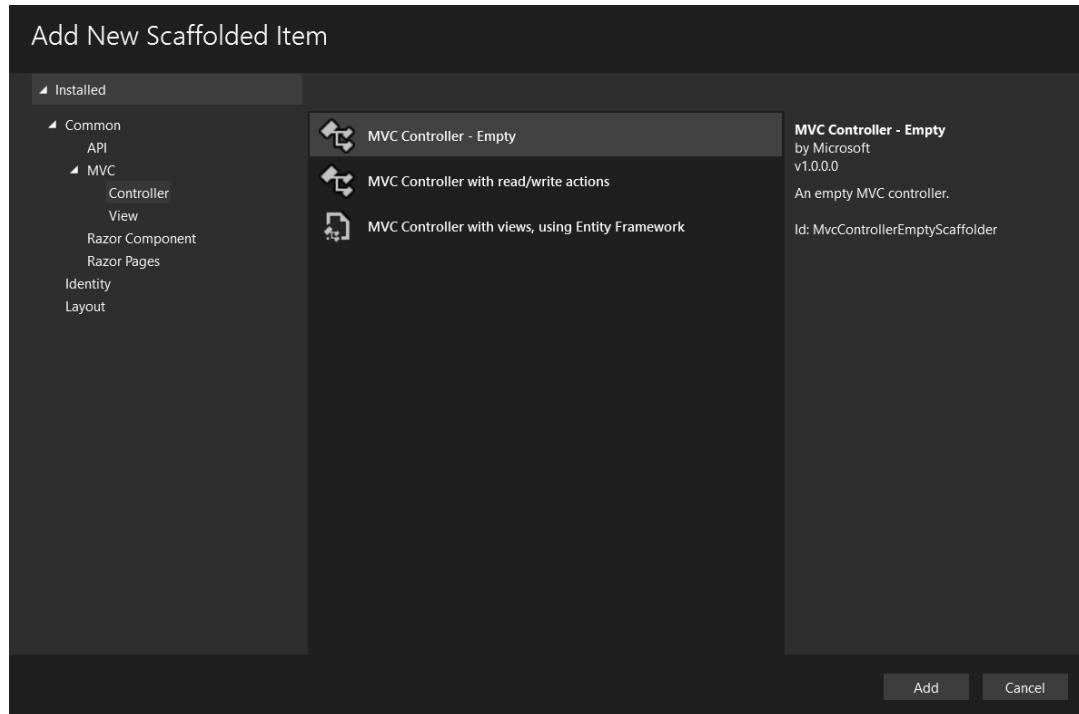
un metodo di azione reale su un controller in genere farà molto

Demo: Adding the Controller

Creiamo una cartella chiamata Controllers, in essa aggiungiamo un Controller:



[Home](#)



E lo chiamiamo PieController

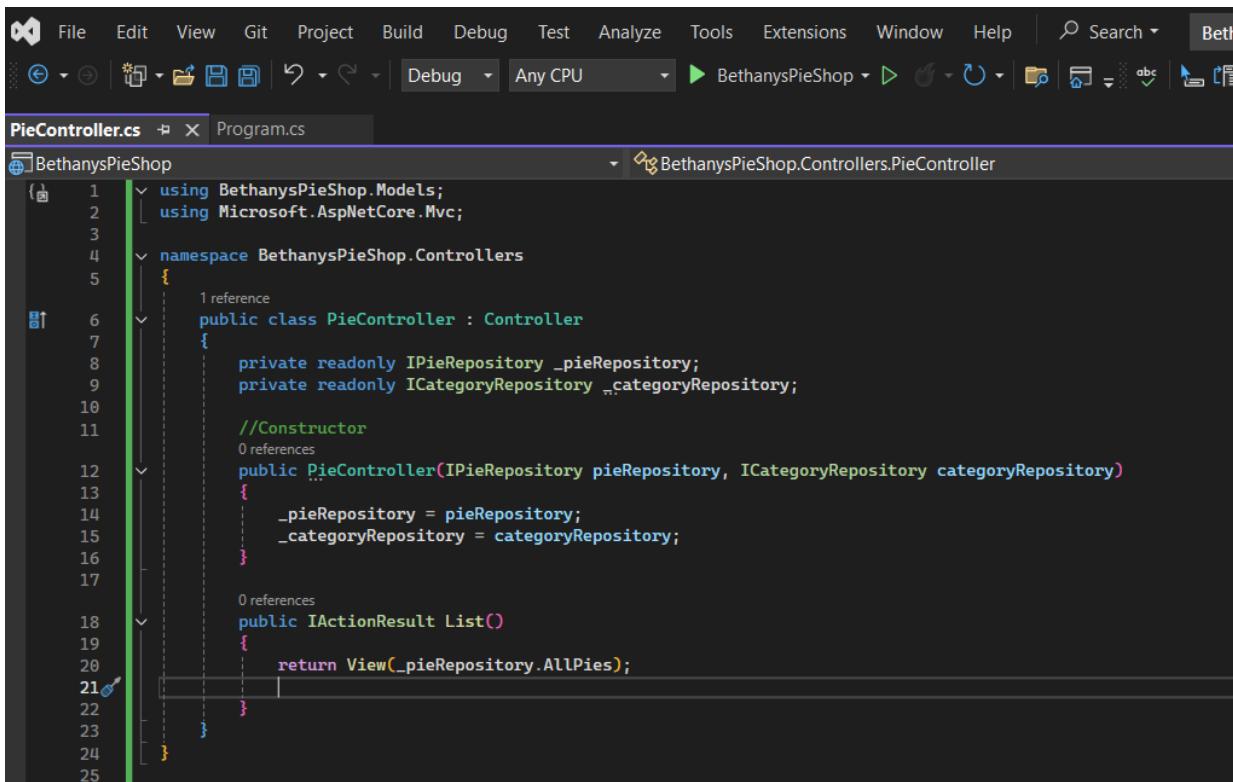
```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace BethanysPieShop.Controllers
4  {
5      public class PieController : Controller
6      {
7          public IActionResult Index()
8          {
9              return View();
10         }
11     }
12 }
13
```

Osservazioni: Notiamo che eredita da Controller.

[Home](#)

Modifichiamo lo script, in quanto ci serve un metodo/azione List che andrà al repository e chiederà la restituzione di tutte le torte, successivamente richiederò la disponibilità dei repository nel mio controller.

Possiamo quindi creare un'istanza di sola lettura:



The screenshot shows the Visual Studio IDE interface with the 'PieController.cs' file open. The code defines a controller named 'PieController' that injects two repositories: 'IPieRepository' and 'ICategoryRepository'. It contains a single action method 'List()' which returns a view of all pies from the repository.

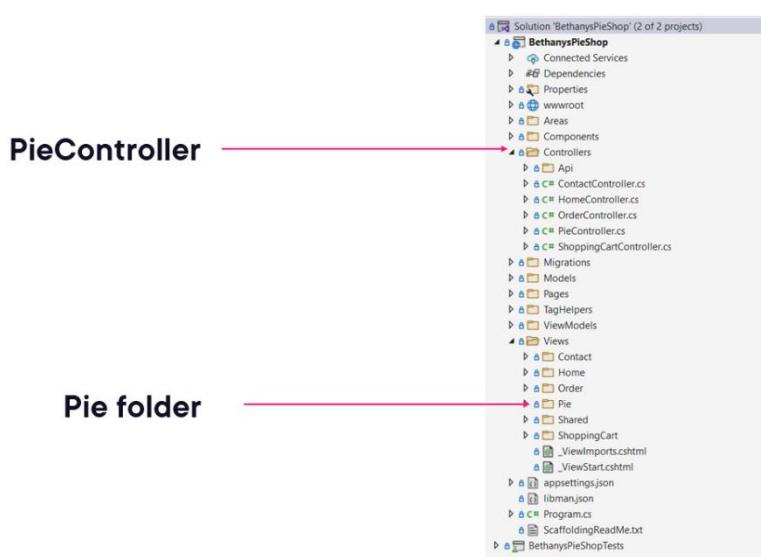
```
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace BethanysPieShop.Controllers
5  {
6      public class PieController : Controller
7      {
8          private readonly IPieRepository _pieRepository;
9          private readonly ICategoryRepository _categoryRepository;
10
11         //Constructor
12         public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
13         {
14             _pieRepository = pieRepository;
15             _categoryRepository = categoryRepository;
16         }
17
18         public IActionResult List()
19         {
20             return View(_pieRepository.AllPies);
21         }
22     }
23 }
24
25 }
```

Ora ci manca solo la vista.

Adding the View

Osservazione: Inserire la logica (C#) necessaria alla View, senza esagerare.

Matching the Controller and Its Views



Quindi per ogni controller dobbiamo avere una sottocartella in cui risiedono i file cshtml, come fa il controller a sapere quale vista verrà renderizzata? Consideriamo questo esempio:

Matching the Action With the View

Convention-based approach

```
public class PieController : Controller
{
    public ViewResult Index() //..... Action
    {
        return View(); //..... View to show: Index.cshtml
    }
}
```

Regular View

```
<!DOCTYPE html>

<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <div>
      Welcome to Bethany's Pie Shop
    </div>
  </body>
</html>
```

```
public class PieController : Controller
{
    public ViewResult Index()
    {
        ViewBag.Message = "Welcome to Bethany's Pie Shop";
        return View();
    }
}
```

Using ViewBag from the Controller

ViewBag, un oggetto dinamico a cui possiamo aggiungere dati dal controller .

Inoltre anche la nostra vista può accedere a questi dati:

Dynamic Content Using ViewBag

```
<!DOCTYPE html>

<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <div>
      @ViewBag.Message
    </div>
  </body>
</html>
```

L'immagine precedente è una vista dinamica, con la '@' utilizziamo Razor, dove indichiamo che il codice C# è in arrivo.

```
@ViewBag.Message

<p>@DateTime.Now</p>

@{
    var message = "Welcome to Bethany's
    Pie Shop";
}

<h3>@message</h3>
```

◀ Using ViewBag in Razor

◀ Displaying a date in Razor code

◀ Using a code block

Dato che nel nostro controller ritorniamo una vista, dobbiamo prepararla in cshtml:

A Strongly-typed View

```
@model IEnumerable<Pie>
<html>
  <body>
    <div>
      @foreach (var pie in Model)
      {
        <div>
          <h2>@pie.Name</h2>
          <h3>@pie.Price.ToString("c")</h3>
          <h4>@pie.Category.CategoryName</h4>
        </div>
      }
    </div>
  </body>
</html>
```

Osservazione: pie viene utilizzato in piccolo, perché una volta che è stata riconosciuta grazie alla dichiarazione del Model, è trattata come una variabile nel foreach.

Demo: Creating first View

Creiamo la cartella Views nel nostro progetto, al suo interno creiamo la cartella Pie per riflettere ciò che avviene nel controller, come è stato in precedenza:



```
namespace BethanysPieShop.Controllers
{
    public class PieController : Controller
    {
        private readonly IPieRepository _pieRepository;
        private readonly ICategoryRepository _categoryRepository;

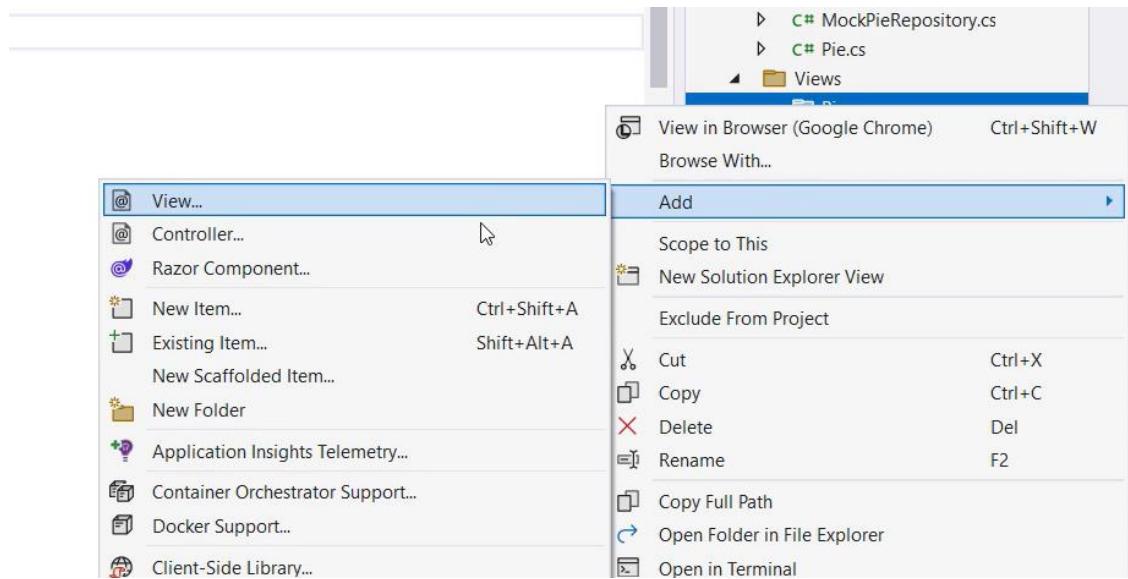
        public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
        {
            _pieRepository = pieRepository;
            _categoryRepository = categoryRepository;
        }

        public IActionResult List()
        {
            return View(_pieRepository.AllPies);
        }
    }
}
```

Così il framework ora cercherà una vista a chiamata Elenco all'interno della cartella Pie.

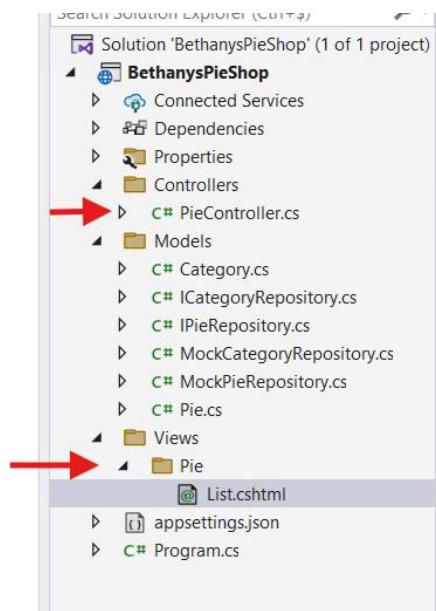
[Home](#)

Aggiungiamo quindi quel file di visualizzazione:



Empty Razor view > Chiamato nel nostro caso List

Cancelliamo tutto il contenuto, ricordiamoci quindi la convenzione dove il controller name



Osservazione: List.cshtml è un file .cshtml ed ha una icona con la '@'

Scriviamo le basi per la nostra view:

[Home](#)

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop'</title>
</head>
<body>
</body>
</html>
```

Per ora abbiamo un semplice html statico, dobbiamo renderlo dinamico.

Ricordando che in PieController.cs abbiamo aggiunto un ViewBag che è dinamico e posso aggiunger qualunque proprietà desidero, possiamo dire che è condivisa tra il mio controller e la mia vista:

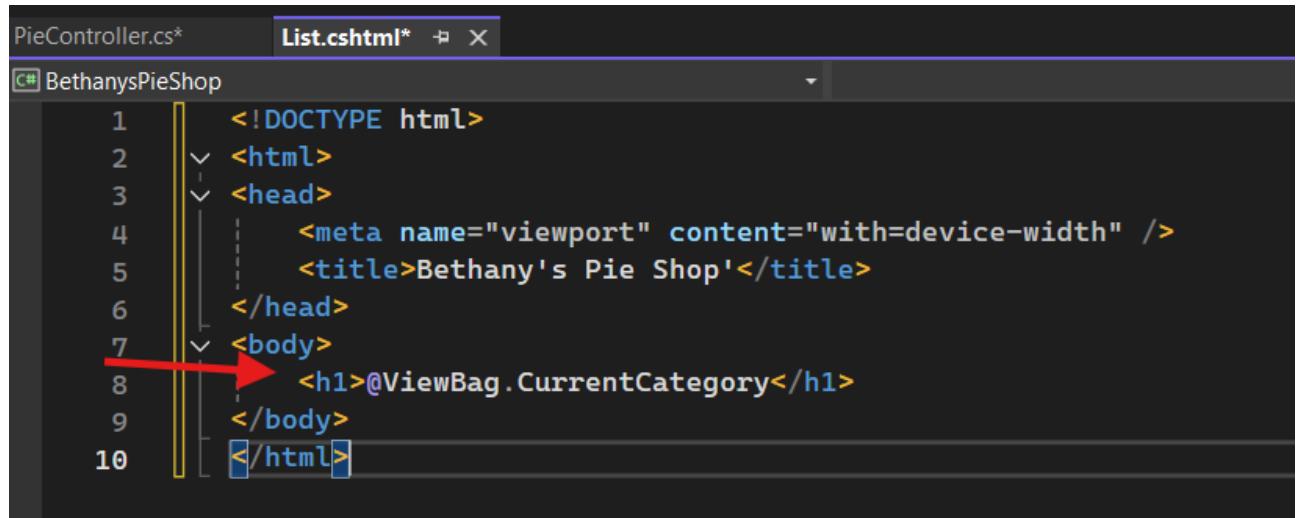
```
namespace BethanysPieShop.Controllers
{
    public class PieController : Controller
    {
        private readonly IPieRepository _pieRepository;
        private readonly ICategoryRepository _categoryRepos
        public PieController(IPieRepository pieRepository,
            categoryRepository)
        {
            _pieRepository = pieRepository;
            _categoryRepository = categoryRepository;
        }

        public IActionResult List()
        {
            ViewBag.CurrentCategory = "Cheese cakes";
            return View(_pieRepository.AllPies);
        }
    }
}
```

La view sarà in grado di accedere alle proprietà all'interno di ViewBag.

Quindi torniamo a List.cshtm e inseriamo esattamente il valore della categoria da visualizzare:

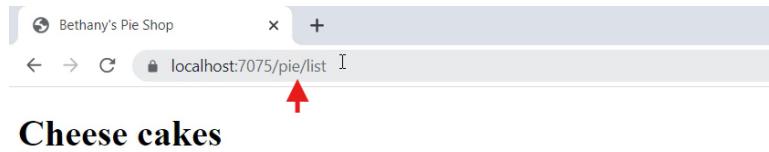
[Home](#)



```
PieController.cs* List.cshtml* ✎ ×
C# BethanysPieShop
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta name="viewport" content="width=device-width" />
5      <title>Bethany's Pie Shop'</title>
6  </head>
7  <body>
8      <h1>@ViewBag.CurrentCategory</h1>
9  </body>
10 </html>
```

Osservazione: Il metodo *CurrentCategory* funziona in run time.

Non siamo ancora pronti per poter visualizzare la nostra vista, è necessaria aggiungere la **route**, a meno che non digitiamo manualmente l'url, esempio:



Tornando al progetto, possiamo passare alla nostra View il modello intero, la vista scaverà in queste proprietà. Consideriamo:

Home

The screenshot shows the Visual Studio IDE with the PieController.cs file open. The code defines a PieController class that implements the Controller interface. It has two private readonly properties: _pieRepository and _categoryRepository. The constructor takes IPieRepository and ICategoryRepository as parameters and initializes the instance variables. The List() method sets ViewBag.CurrentCategory to "Cheese cakes" and returns a view that takes _pieRepository.AllPies as a parameter. A tooltip on the right indicates that AllPies is not null here.

```
4  namespace BethanyPieShop.Controllers
5  {
6      public class PieController : Controller
7      {
8          private readonly IPieRepository _pieRepository;
9          private readonly ICategoryRepository _categoryRepository;
10
11         public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
12         {
13             _pieRepository = pieRepository;
14             _categoryRepository = categoryRepository;
15         }
16
17         public IActionResult List()
18         {
19             ViewBag.CurrentCategory = "Cheese cakes";
20             return View(_pieRepository.AllPies);
21         }
22     }
23 }
24
```

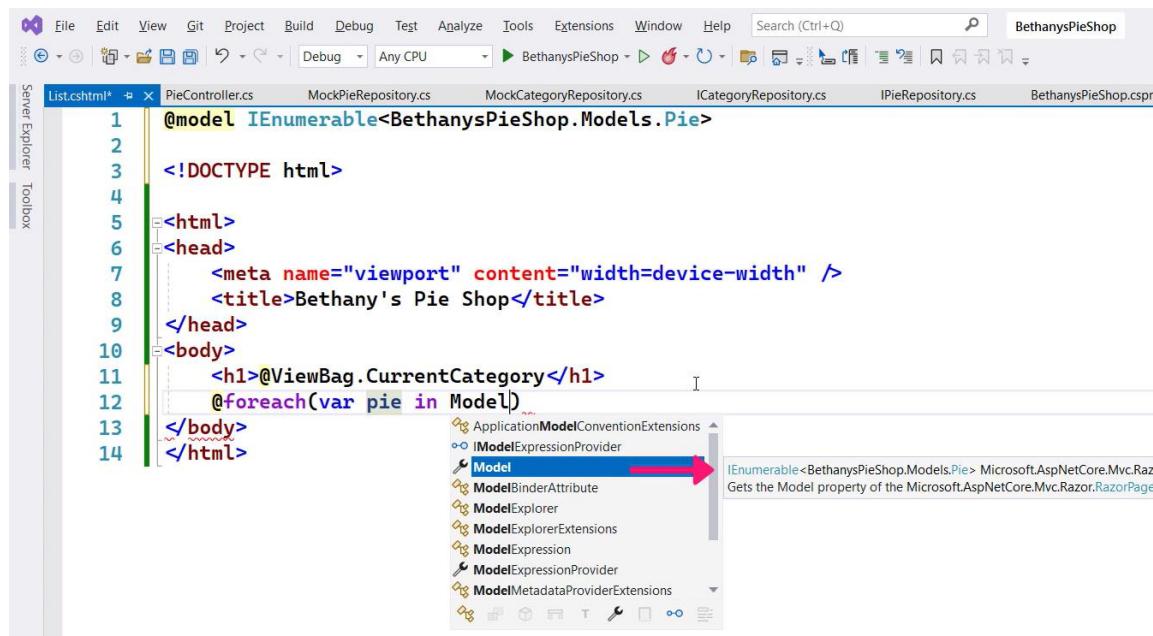
Torno su List.cshtml e aggiungo direttamente il modello IEnumerable:

The screenshot shows the List.cshtml file in Visual Studio. The first line contains the @model directive followed by the type IEnumarable<BethanyPieShop.Models.Pie>. An arrow points from the text "Ora effettuerò il loop su tutte le torte che mi sono state passate in questo Ienumerable e le visualizzerò alcuni valori per ciascuno di questi, e potremmo farlo con un @ per ciascuno:" to the start of the @model directive.

```
1  @model IEnumerable<BethanyPieShop.Models.Pie>
2
3  <!DOCTYPE html>
4
5  <html>
6      <head>
7          <meta name="viewport" content="width=device-width" />
8          <title>Bethany's Pie Shop</title>
9      </head>
10     <body>
11         <h1>@ViewBag.CurrentCategory</h1>
12     </body>
13     </html>
```

Ora effettuerò il loop su tutte le torte che mi sono state passate in questo Ienumerable e le visualizzerò alcuni valori per ciascuno di questi, e potremmo farlo con un @ per ciascuno:

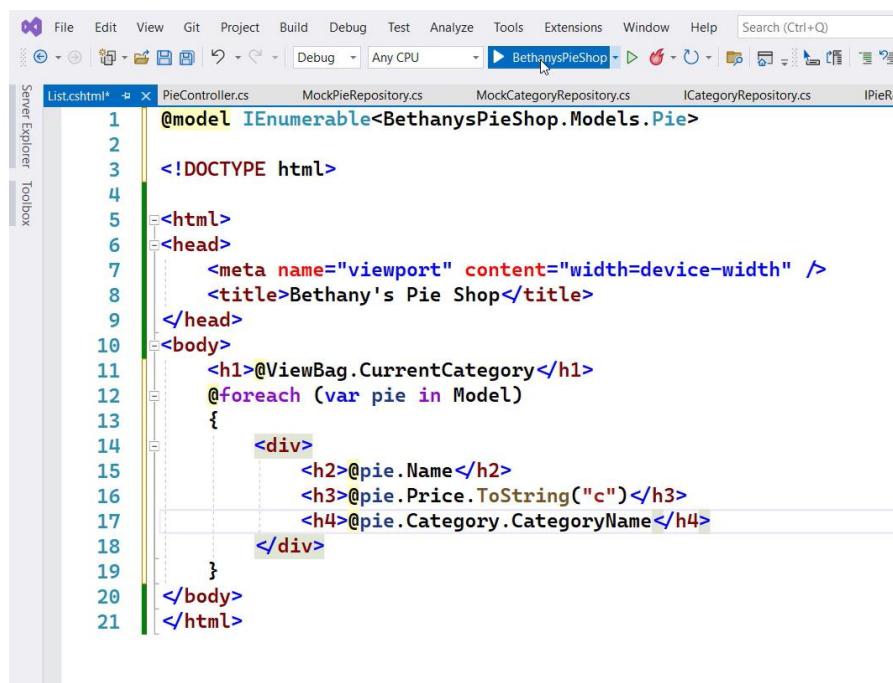
Home



The screenshot shows the Visual Studio IDE with the List.cshtml file open. The code is as follows:

```
@model IEnumerable<BethanysPieShop.Models.Pie>
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop</title>
</head>
<body>
    <h1>@ViewBag.CurrentCategory</h1>
    @foreach(var pie in Model)
    {
        <div>
            <h2>@pie.Name</h2>
            <h3>@pie.Price.ToString("c")</h3>
            <h4>@pie.Category.CategoryName</h4>
        </div>
    }
</body>
</html>
```

A tooltip from the Intellisense dropdown is shown, pointing to the 'Model' entry in the list. The tooltip text is: 'IEnumerable<BethanysPieShop.Models.Pie> Microsoft.AspNetCore.Mvc.Razor Gets the Model property of the Microsoft.AspNetCore.Mvc.Razor.RazorPage'. A red arrow points from the text to the 'Model' entry in the list.



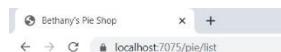
The screenshot shows the Visual Studio IDE with the List.cshtml file open. The code is now fully completed:

```
@model IEnumerable<BethanysPieShop.Models.Pie>
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop</title>
</head>
<body>
    <h1>@ViewBag.CurrentCategory</h1>
    @foreach (var pie in Model)
    {
        <div>
            <h2>@pie.Name</h2>
            <h3>@pie.Price.ToString("c")</h3>
            <h4>@pie.Category.CategoryName</h4>
        </div>
    }
</body>
</html>
```

Osservazione: All'interno delle parentesi graffe non è necessario ripetere la @, a meno che non è necessario passare codice negli elementi html.

Ora dovremmo essere in grado di visualizzare (sempre inserendo l' url manualmente) la lista delle torte presenti in quella categoria, dove per ognuna è elencato nome, prezzo e nome della categoria:

[Home](#)



Cheese cakes

Strawberry Pie

15,95 €

Fruit pies

Cheese cake

18,95 €

Cheese cakes

Rhubarb Pie

15,95 €

Fruit pies

Pumpkin Pie

12,95 €

Seasonal pies

Demo: Using a View Model

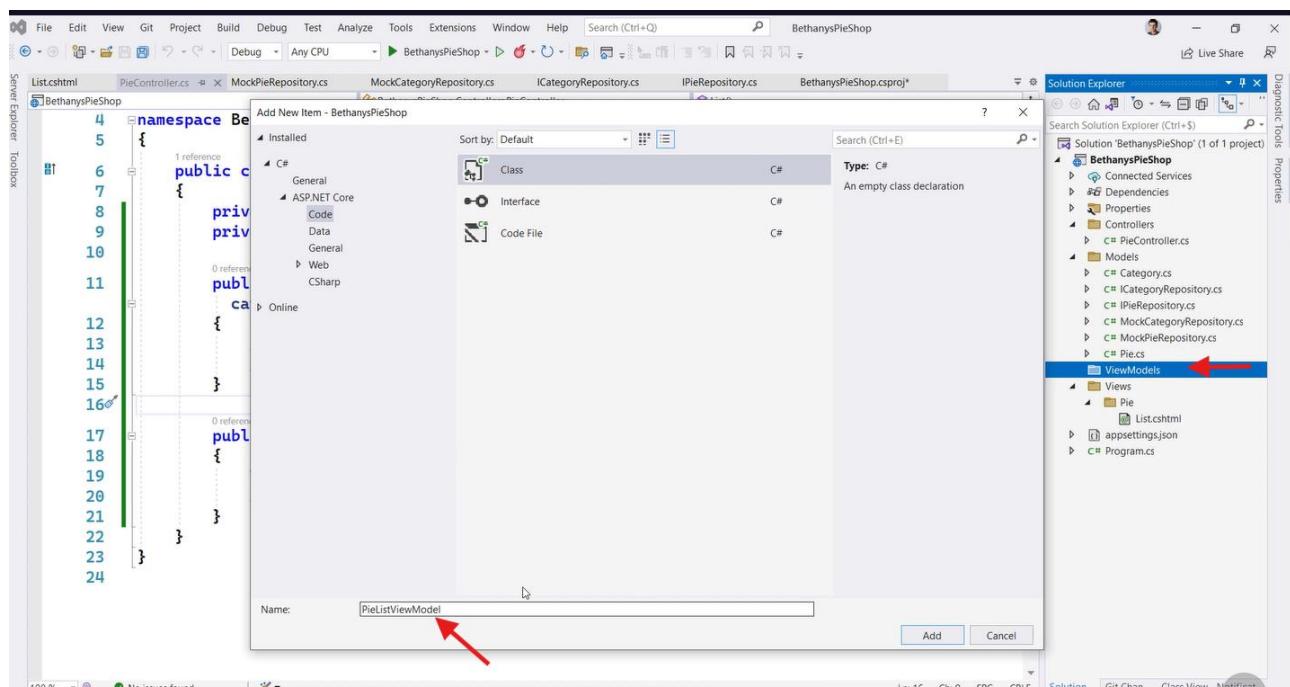
```
public class PieListViewModel
{
    public IEnumerable<Pie>? Pies { get; set; }
    public string? CurrentCategory { get; set; }
}
```

Introducing the View Model

In molti casi vedremo che il modello che vogliamo passare alla vista non è MAPPATO a un modello di dominio. E' buona abitudine utilizzare un modello di visualizzazione, ovvero una classe che racchiude più proprietà in modo che la visualizzazione possa accedere a tutti i dati di cui ha bisogno.

Creiamo quindi una nuova cartella nel nostro progetto "**ViewModels**", **conterrà delle utility class non vere e proprie entità del DB.**

Home



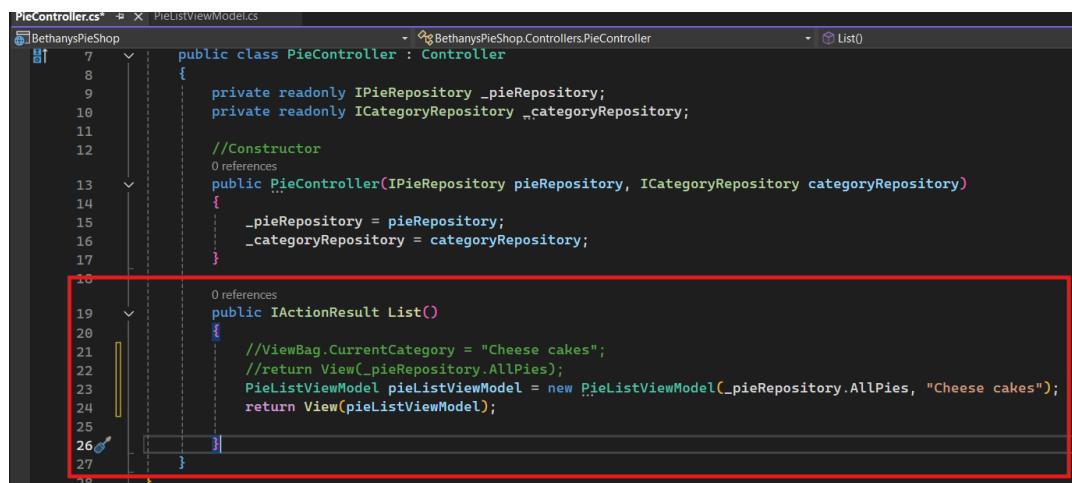
Scriviamo il codice:

```
using BethanysPieShop.Models;
namespace BethanysPieShop.ViewModels
{
    public class PieListViewModel
    {
        public IEnumerable<Pie> Pies { get; }
        public string? CurrentCategory { get; }

        public PieListViewModel(IEnumerable<Pie> pies, string? currentCategory)
        {
            Pies = pies;
            CurrentCategory = currentCategory;
        }
    }
}
```

Fatto ciò, torniamo al controller e lo modifichiamo, in quanto ora possiamo utilizzare in PieController.cs il costruttore PieListViewModel appena creato:

Home



```
PieController.cs*  X  PieListViewModel.cs
BethanysPieShop
public class PieController : Controller
{
    private readonly IPieRepository _pieRepository;
    private readonly ICategoryRepository _categoryRepository;

    //Constructor
    public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
    {
        _pieRepository = pieRepository;
        _categoryRepository = categoryRepository;
    }

    public IActionResult List()
    {
        ViewBag.CurrentCategory = "Cheese cakes";
        return View(_pieRepository.AllPies);
        PieListViewModel pieListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
        return View(pieListViewModel);
    }
}
```

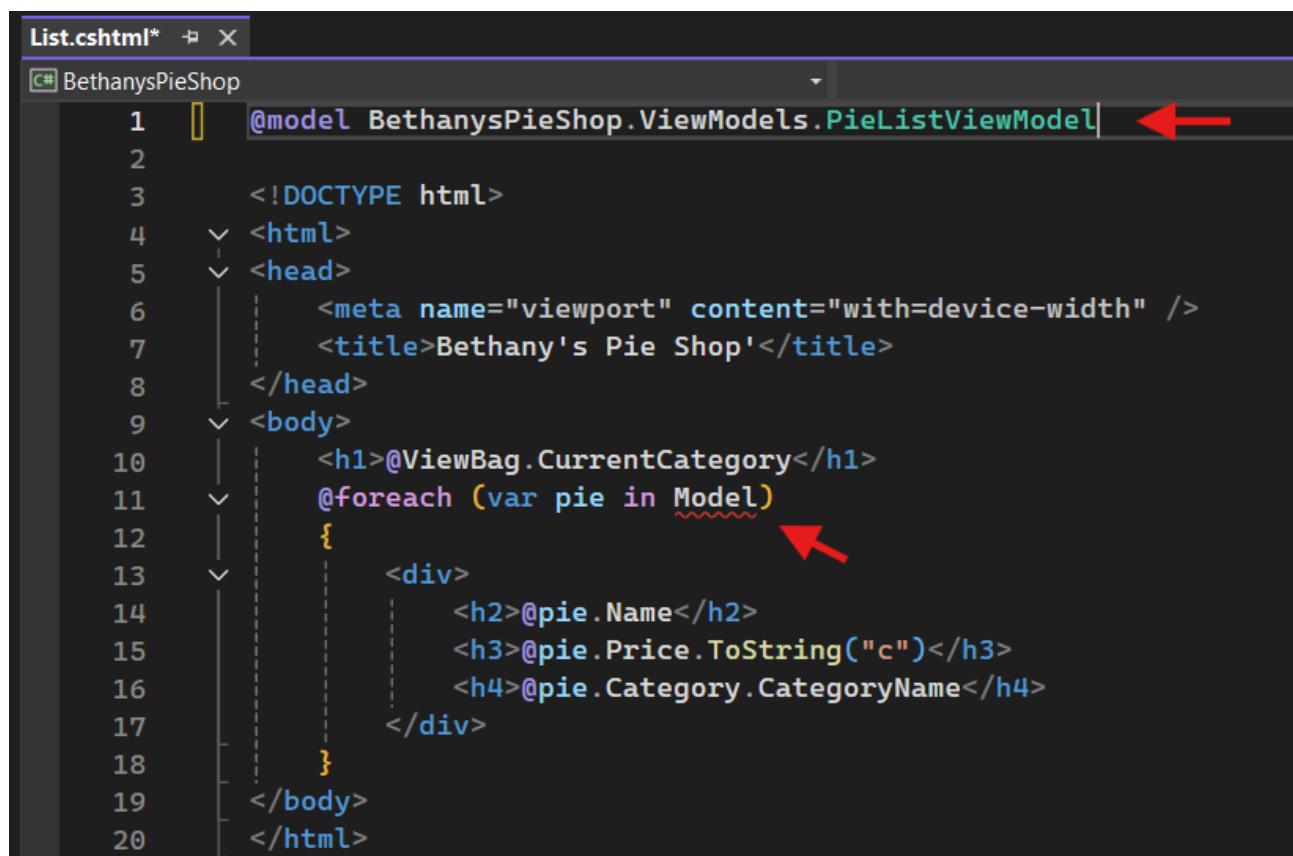
Dobbiamo riflettere tali modifiche anche per la View, quindi ci rechiamo in List.cshtml e modifichiamo la riga del model, in quanto ora utilizziamo la cartella ViewModels e PieListViewModel.

Prima:



```
List.cshtml*  X  PieController.cs      MockPieRepository.cs      MockCategoryRepository.cs      ICategoryRepositor
1  @model IEnumerable<BethanysPieShop.Models.Pie>
```

Dopo:



```
List.cshtml*  X
BethanysPieShop
1  @model BethanysPieShop.ViewModels.PieListViewModel
2
3  <!DOCTYPE html>
4  <html>
5  <head>
6      <meta name="viewport" content="width=device-width" />
7      <title>Bethany's Pie Shop'</title>
8  </head>
9  <body>
10     <h1>@ViewBag.CurrentCategory</h1>
11     <foreach var pie in Model>
12     {
13         <div>
14             <h2>@pie.Name</h2>
15             <h3>@pie.Price.ToString("c")</h3>
16             <h4>@pie.Category.CategoryName</h4>
17         </div>
18     }
19     </body>
20     </html>
```

L'errore a riga 11, indica che model non è più un lenumerables in Pies, quindi possiamo non enumerarlo, però ora abbiamo Pies, quindi scriviamo Model.Pies:

```
@model BethanysPieShop.ViewModels.PieListViewModel

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop'</title>
</head>
<body>
    <h1>@ViewBag.CurrentCategory</h1>
    @foreach (var pie in Model.Pies)
    {
        <div>
            <h2>@pie.Name</h2>
            <h3>@pie.Price.ToString("c")</h3>
    
```



ViewBag non è più utilizzato, diventa Model:

```
<h1>Model.CurrentCategory</h1>
```

Ora abbiamo un codice più pulito con lo stesso risultato

Adding Extra View Files

Using the _Layout.cshtml

Template

Lives in Shared
folder
Searched by default

One or more
View can specify

Esempio di modello di layout:

_Layout.cshtml

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bethany's Pie Shop</title>
    </head>
    <body>
        <div>
            @RenderBody()
        </div>
    </body>
</html>
```

```
@{
    Layout = "_Layout";
}

@model PieListViewModel
```

View Can Specify the Layout

Osservazione: possiamo farlo individualmente in ogni visione

[Home](#)

```
@{  
    Layout = "_Layout";  
}
```

_ViewStart.cshtml

Contains logic shared by set of views
Executed automatically

Questo è un altro file CSHTML che verrà cercato automaticamente da ASP.NET

Osservazione: in _ViewStart.cshtml in genere inseriamo il codice che vogliamo eseguire quando viene chiamata una qualsiasi vista

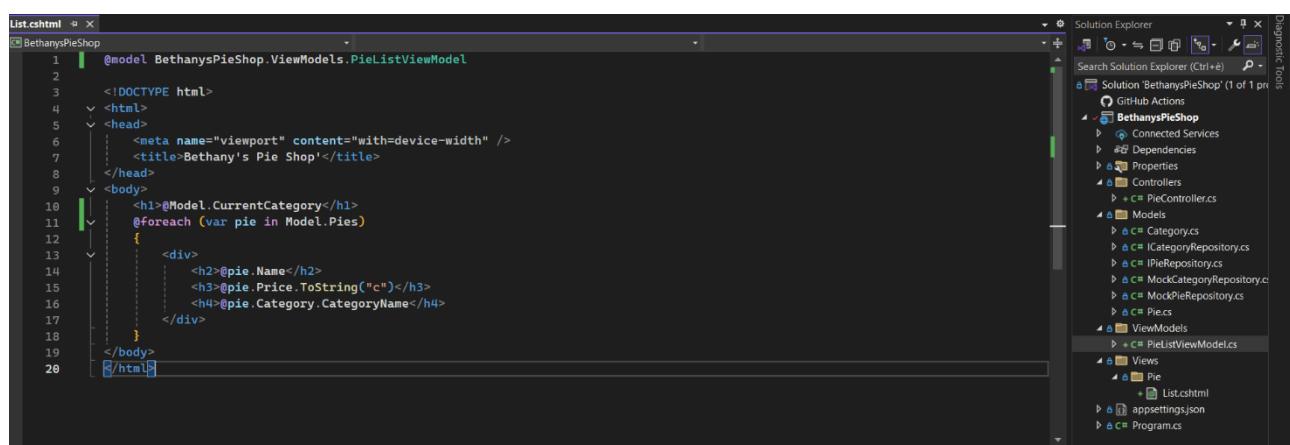
```
@using BethanysPieShop.Models
```

View Imports

Group commonly used using statements

Demo: Adding Extra View Files

Ciò che faremo è "Adding a layout template", "Creating the ViewStart file", Adding the ViewImports file"



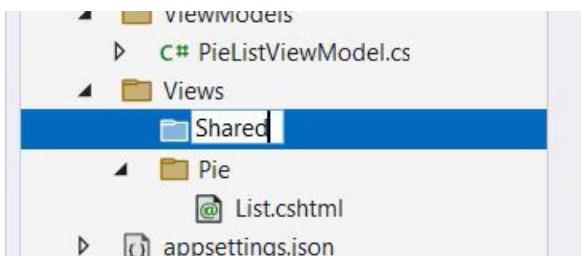
Attualmente il nostro progetto contiene una sola pagina.

Views/Shared è la cartella condivisa tra tutte le visualizzazioni di tutti i controllori, conosciuta automaticamente

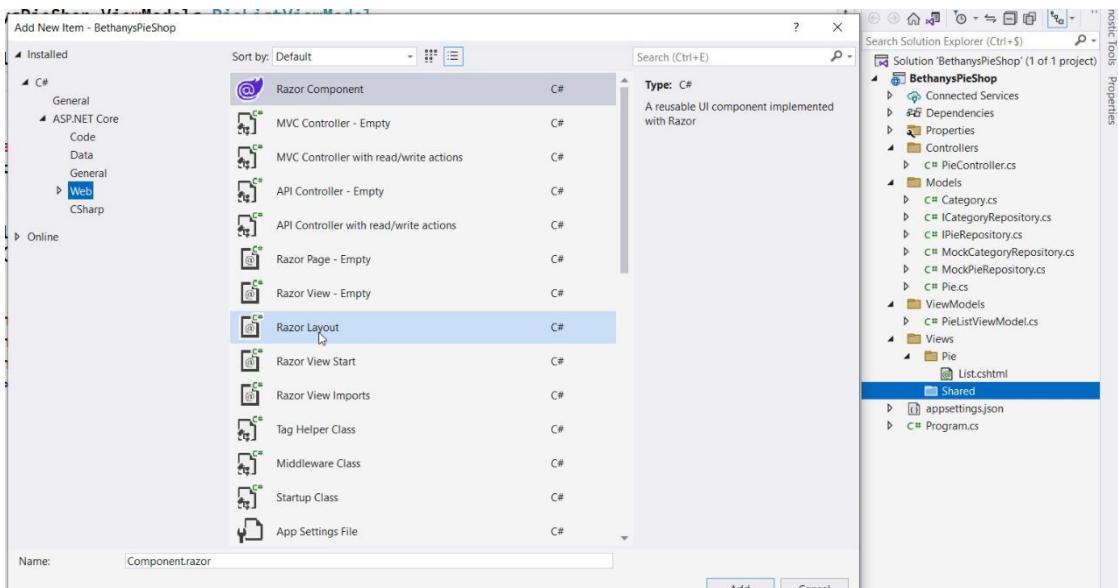
[Home](#)

da ASP.NET Core.

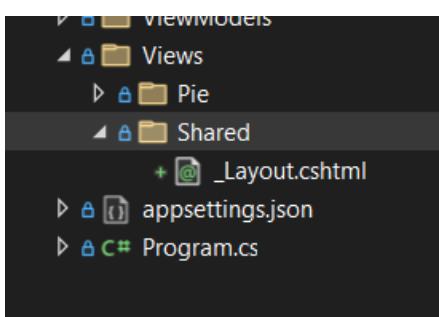
Quindi la creiamo:



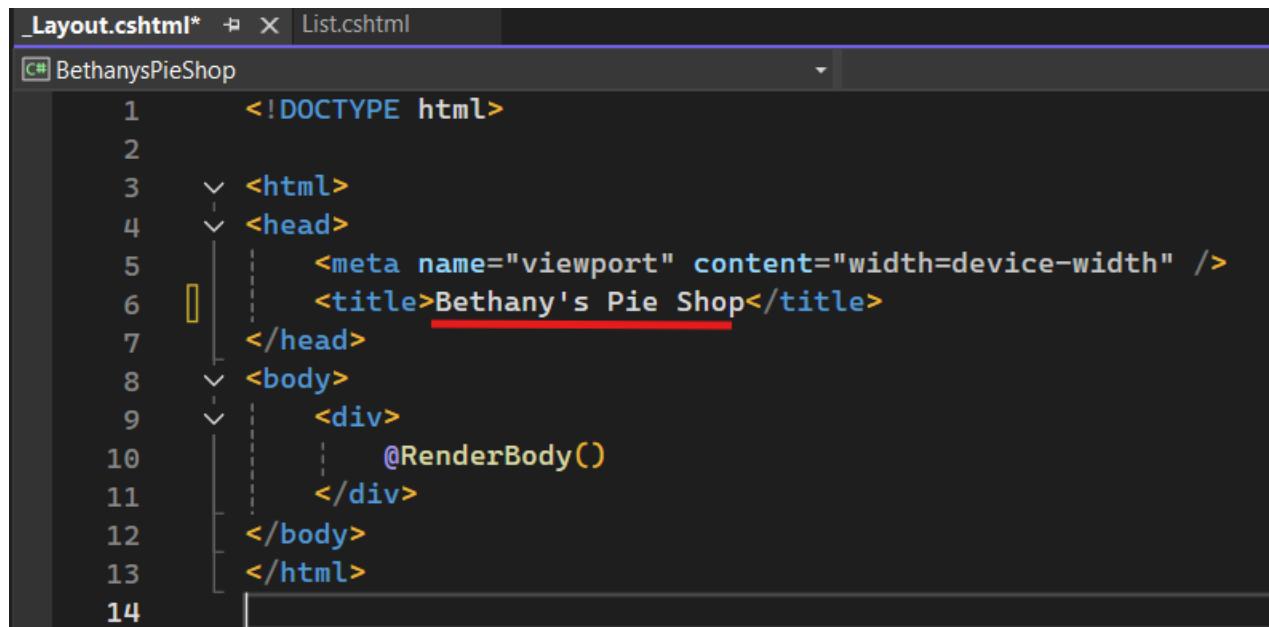
All'interno della cartella creiamo il nostro layout con Add New Item > Class > Web > Razor Layout:



Lasciamo il default name per ora



[Home](#)



```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop</title>
</head>
<body>
    @RenderBody()
</body>
</html>
```

Ora torniamo nel nostro List.cshtml per rimuovere la parte statica (quindi anche i due elementi finali "/body" e "/html"):



```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Bethany's Pie Shop</title>
</head>
<body>
    <h1>@Model.CurrentCategory</h1>
    @foreach (var pie in Model.Pies)
    {
        <div>
            <h2>@pie.Name</h2>
            <h3>@pie.Price.ToString("c")</h3>
            <h4>@pie.Category.CategoryName</h4>
        </div>
    }
</body>
</html>
```

Ed aggiungiamo il nostro layout a List.cshtml:

Home

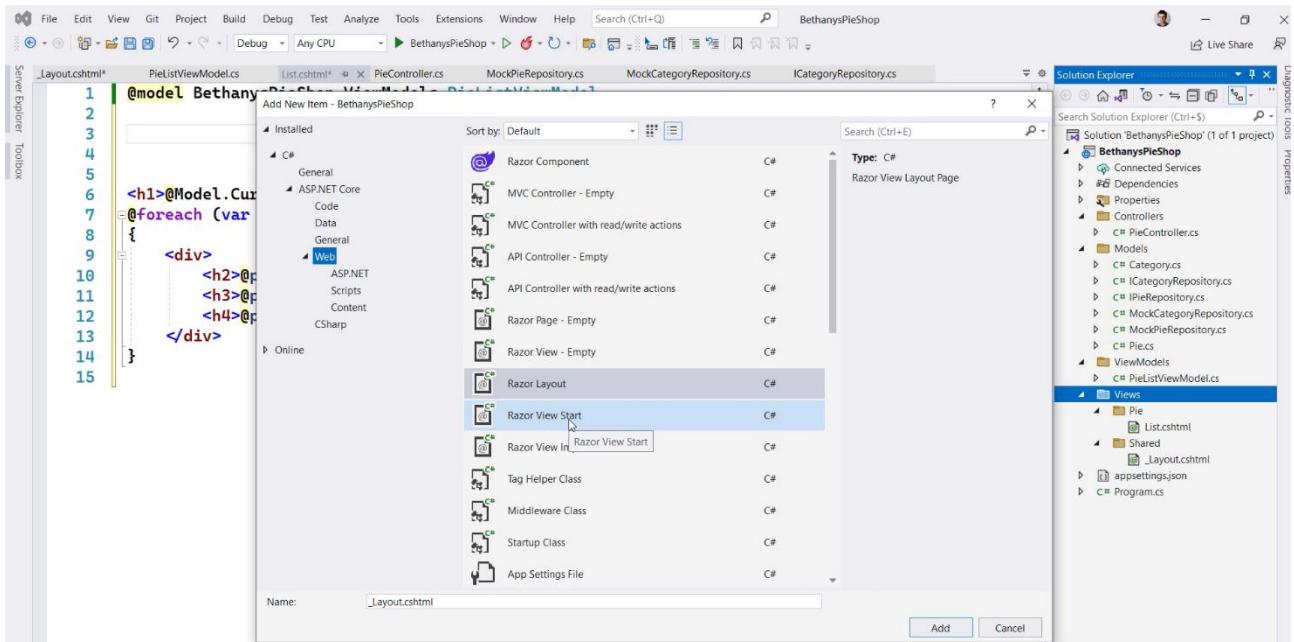
```
1 @model BethanyPieShop.ViewModels.PieListViewModel
2
3 @{
4     Layout = "_Layout";
5 }
6
7
8 <h1>@Model.CurrentCategory</h1>
9 @foreach (var pie in Model.Pies)
10 {
11     <div>
12         <h2>@pie.Name</h2>
13         <h3>@pie.Price.ToString("c")</h3>
14         <h4>@pie.Category.CategoryName</h4>
15     </div>
16 }
```

Possiamo quindi includere quel layout in ogni file, ma in realtà è possibile rimuoverlo di nuovo e creare quello che è noto

come file ViewStart, il quale verrà eseguito ogni volta che viene eseguito il rendering di una vista e li inseriamo il codice che deve

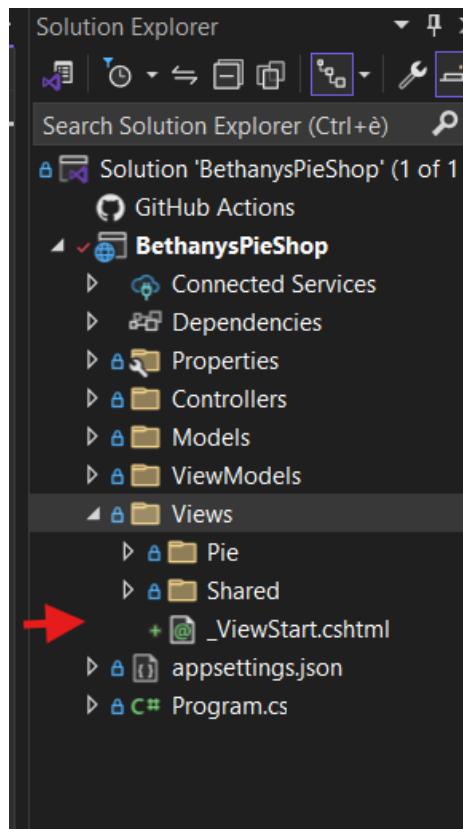
essere eseguito ogni volta

Quindi cancelliamo la porzione di codice presente nell'immagine precedente e aggiungiamo Razor View Start:

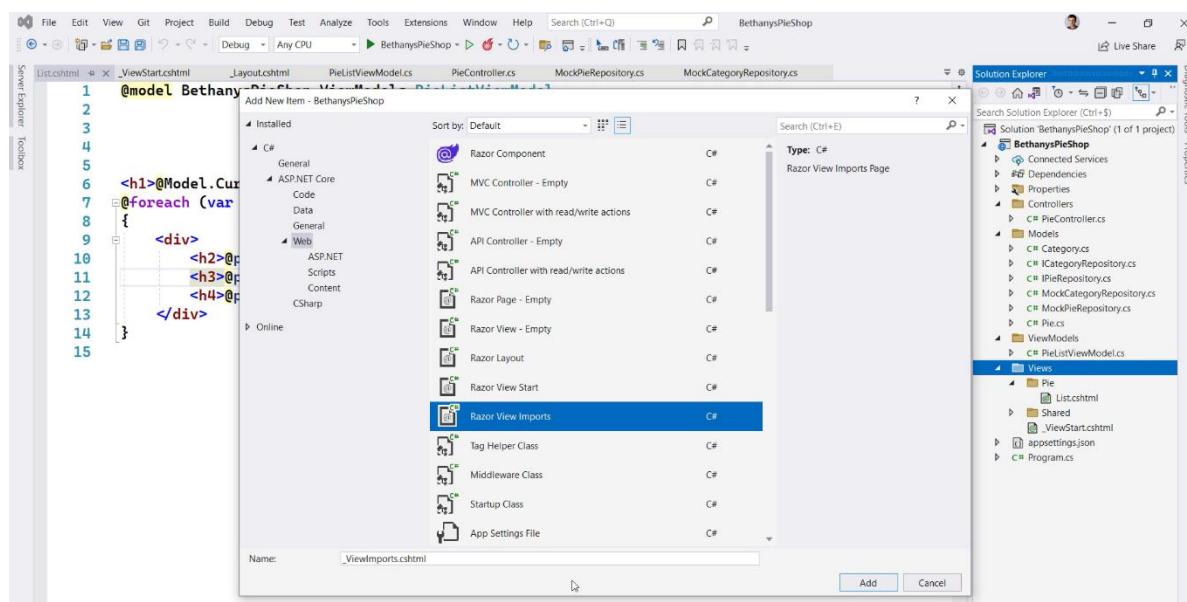


Deve essere presente nella root di View:

Home



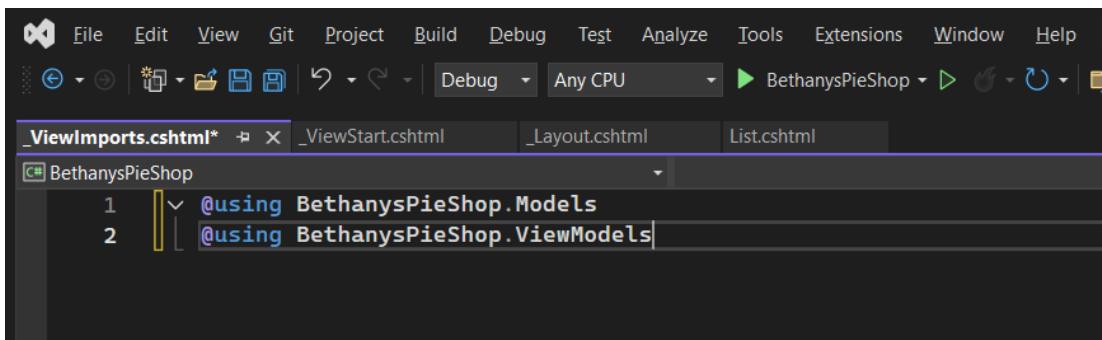
Sempre in Views aggiungiamo Razor _ViewImports.cshtml:



_ViewImports.cshtml sarà utilizzato per inserire altre istruzioni using globali

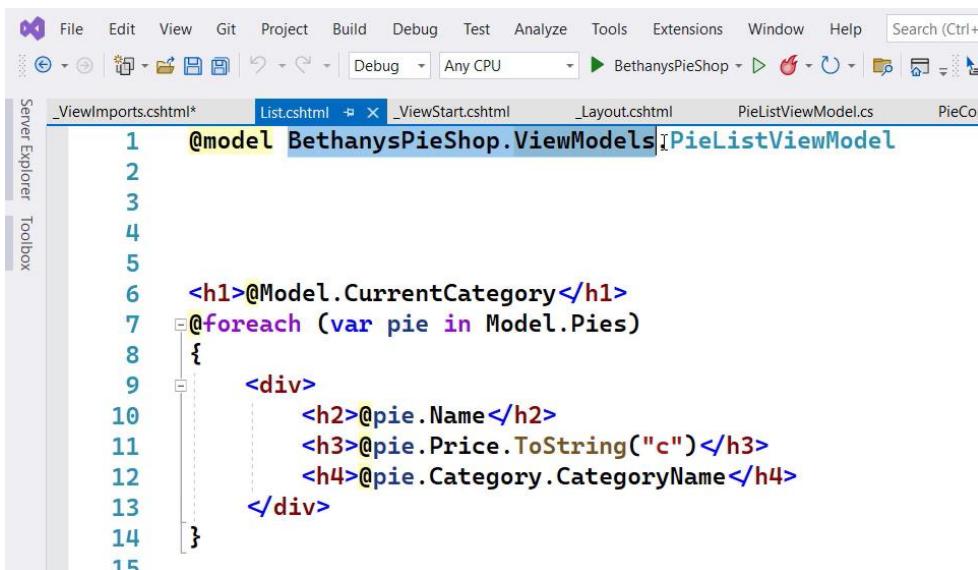
Quindi inseriamo:

Home



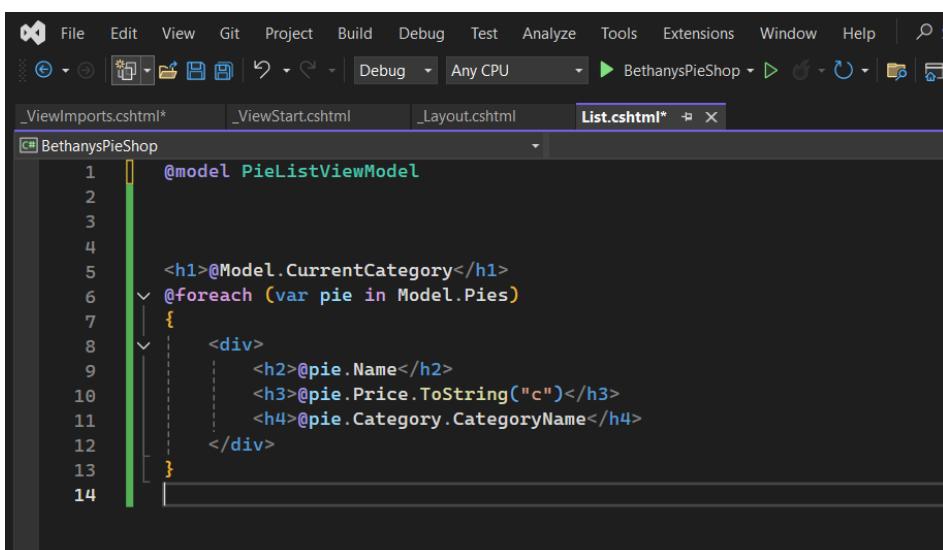
```
_ViewImports.cshtml*  _ViewStart.cshtml  _Layout.cshtml  List.cshtml
C# BethanysPieShop
1  @using BethanysPieShop.Models
2  @using BethanysPieShop.ViewModels
```

Questo ci permette di poter cancellare da List.cshtml la stringa *BethanysPieShop.ViewModels*:



```
_ViewImports.cshtml*  List.cshtml  _ViewStart.cshtml  _Layout.cshtml  PieListViewModel.cs  PieCor
1  @model BethanysPieShop.ViewModels|PieListViewModel
2
3
4
5
6  <h1>@Model.CurrentCategory</h1>
7  @foreach (var pie in Model.Pies)
8  {
9    <div>
10      <h2>@pie.Name</h2>
11      <h3>@pie.Price.ToString("c")</h3>
12      <h4>@pie.Category.CategoryName</h4>
13    </div>
14  }
15
```

Ottenendo:



```
_ViewImports.cshtml*  _ViewStart.cshtml  _Layout.cshtml  List.cshtml*
C# BethanysPieShop
1  @model PieListViewModel
2
3
4
5  <h1>@Model.CurrentCategory</h1>
6  @foreach (var pie in Model.Pies)
7  {
8    <div>
9      <h2>@pie.Name</h2>
10     <h3>@pie.Price.ToString("c")</h3>
11     <h4>@pie.Category.CategoryName</h4>
12   </div>
13 }
14
```

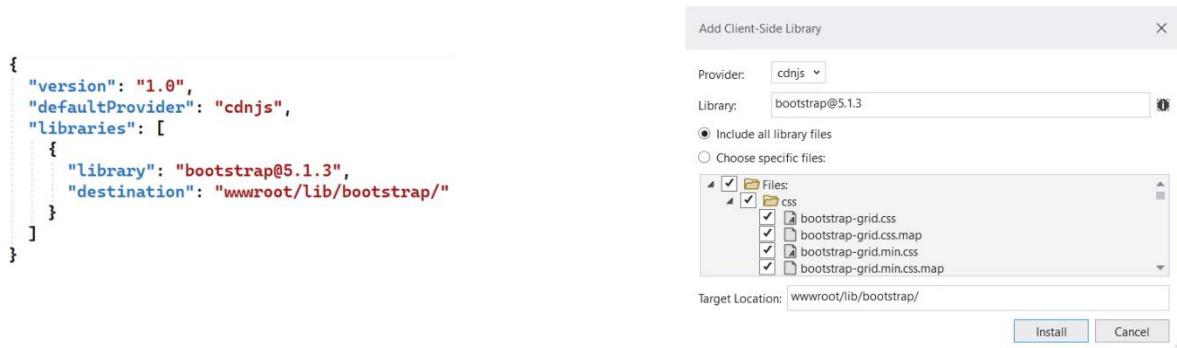
Styling the view

Utilizzeremo Bootstrap (una libreria CSS)

Inseriamo questa libreria Client-Side tramite il gestore delle librerie di Visual Studio.

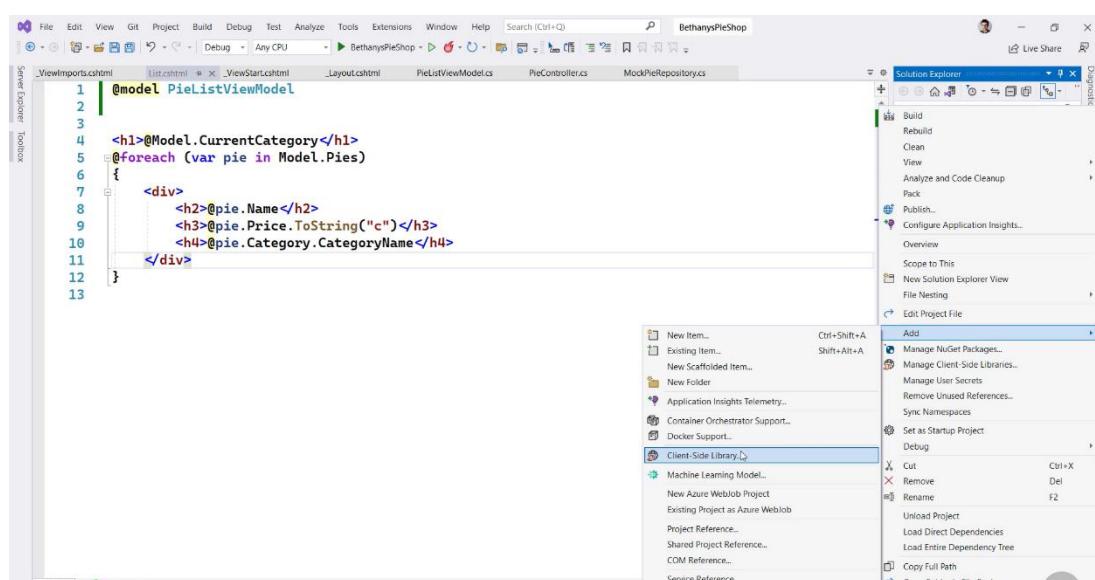
Il tutto sarà presente in un file .json, che possiamo modificare a nostro piacimento per inserire ulteriori dipendenze.

Adding Client-side Libraries

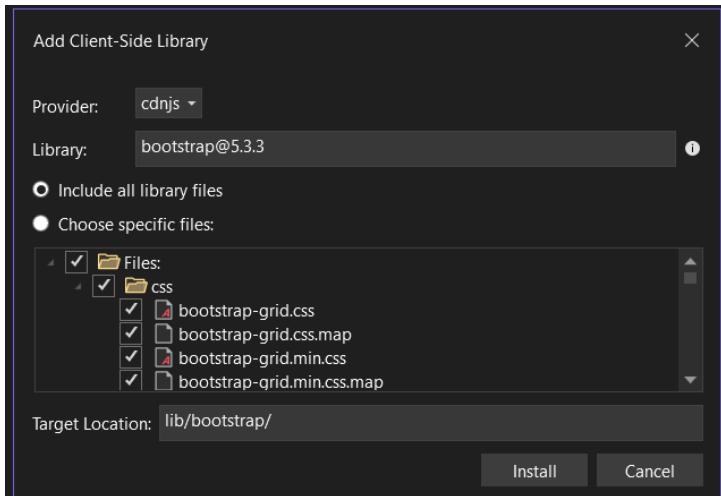


Demo: Styling the view

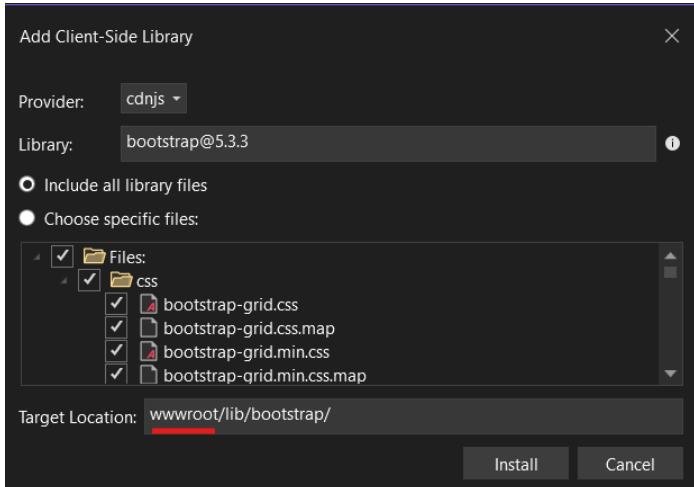
Click destro sul nostro progetto *Add > Client-Side-Library*



[Home](#)

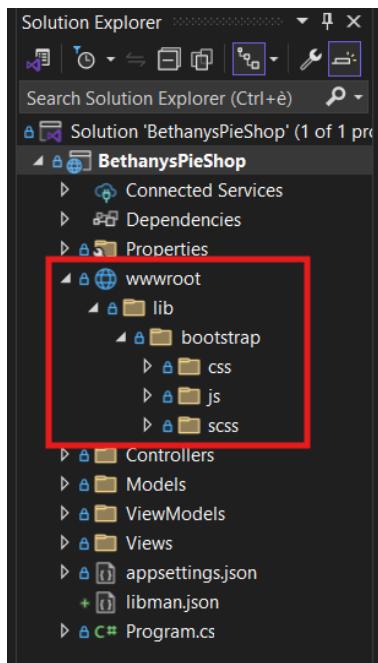


Dopo aver cercato Bootstrap l'editor ci consiglia l'ultima versione presente, ora ci tocca modificare il target location, in:

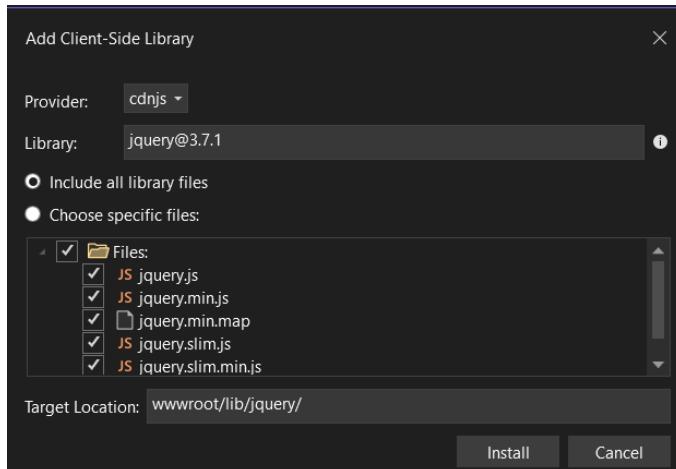


Ora abbiamo la nostra libreria:

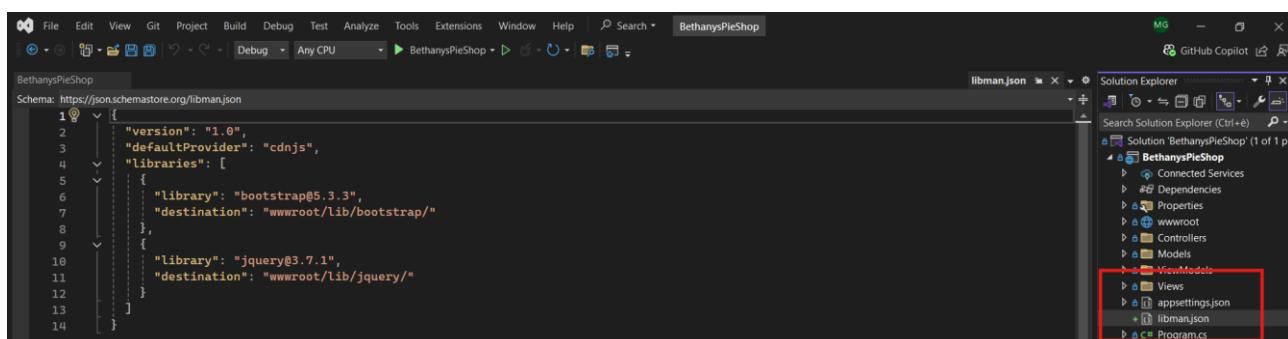
Home



Procediamo in maniera analoga con l'aggiunta di Jquery:



Possiamo notare le varie configurazioni nel file libman.json:

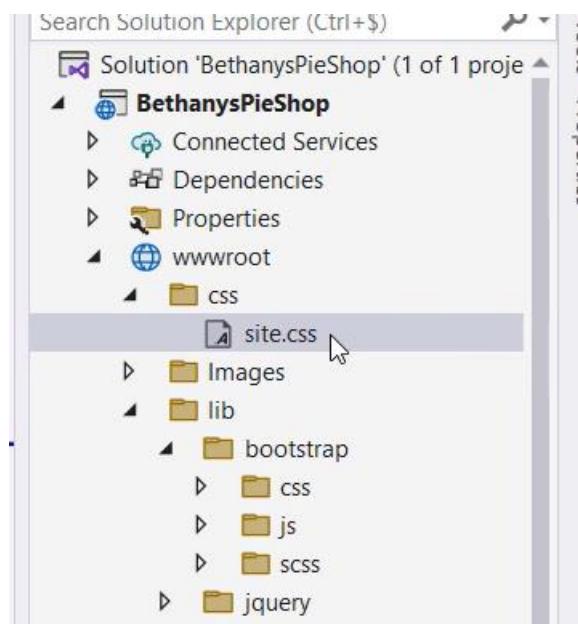


[Home](#)

Ora copiamo i seguenti static files presenti nel nostro File System:

Nome	Tipo	Dimensione compre...	Protetto da...	Dimensione	Proporzione	Ultima modifica
Images	Cartella di file					08/01/2024 04:12
site.css	File di origine CSS	28 KB	No	236 KB 89%		05/02/2024 04:03

E li incolliamo in wwwroot:



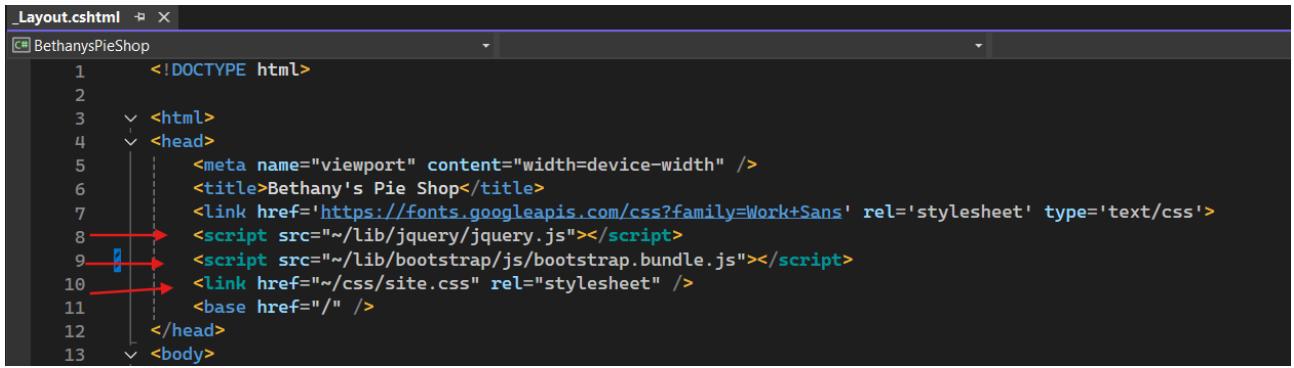
Ora torniamo al nostro _layout.cshtml dove nell' head aggiungeremo tutti i metadati di cui abbiamo bisogno, come jquery, bootstrap e il nostro site.css:

Prima:

```
_Layout.cshtml
1 <!DOCTYPE html>
2
3 <html>
4 <head>
5   <meta name="viewport" content="width=device-width" />
6   <title>Bethany's Pie Shop</title>
7 </head>
8 <body>
9   <div>
10    @RenderBody()
11   </div>
12 </body>
13 </html>
```

Dopo:

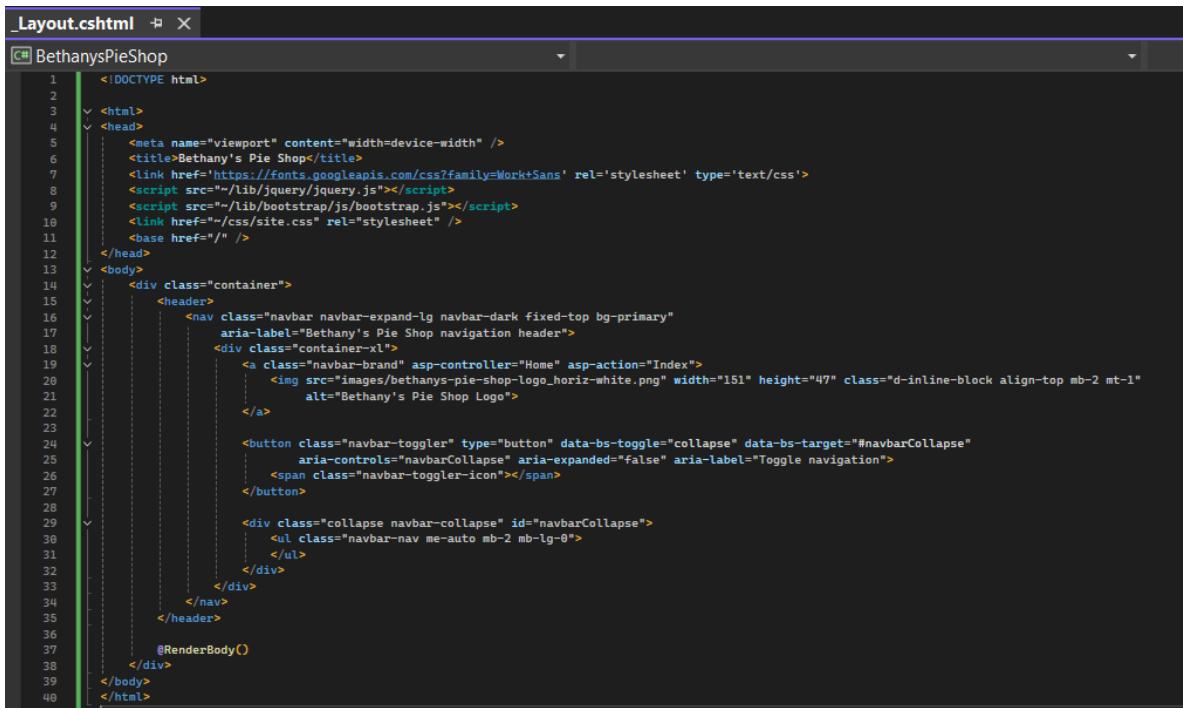
[Home](#)



```
<!DOCTYPE html>
<html>
    <head>
        <meta name="viewport" content="width=device-width" />
        <title>Bethany's Pie Shop</title>
        <link href='https://fonts.googleapis.com/css?family=Work+Sans' rel='stylesheet' type='text/css'>
        <script src='~/lib/jquery/jquery.js'></script>
        <script src='~/lib/bootstrap/js/bootstrap.bundle.js'></script>
        <link href='~/css/site.css' rel='stylesheet' />
        <base href="/" />
    </head>
    <body>
```

Consiglio:Riga 9, per essere certi che tutto funzioni correttamente, consiglio di utilizzare *bootstrap.bundle.js*

Ora prima di @RenderBody() dovremmo aggiungere il nostro container al <div>, nel seguente modo:



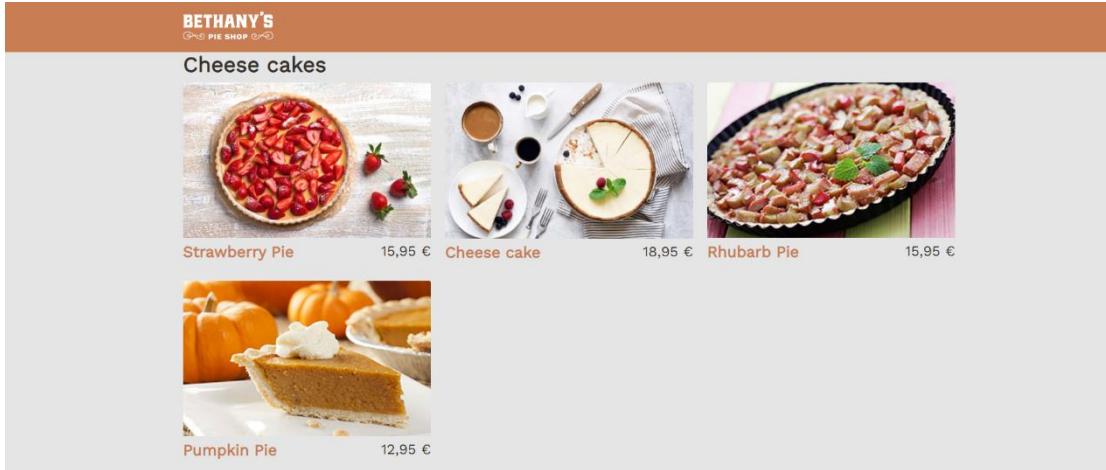
```
<!DOCTYPE html>
<html>
    <head>
        <meta name="viewport" content="width=device-width" />
        <title>Bethany's Pie Shop</title>
        <link href='https://fonts.googleapis.com/css?family=Work+Sans' rel='stylesheet' type='text/css'>
        <script src='~/lib/jquery/jquery.js'></script>
        <script src='~/lib/bootstrap/js/bootstrap.js'></script>
        <link href='~/css/site.css' rel='stylesheet' />
        <base href="/" />
    </head>
    <body>
        <div class="container">
            <header>
                <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary" aria-label="Bethany's Pie Shop navigation header">
                    <div class="container-xl">
                        <a class="navbar-brand" asp-controller="Home" asp-action="Index">
                            
                        </a>
                        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse" aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
                            <span class="navbar-toggler-icon"></span>
                        </button>
                        <div class="collapse navbar-collapse" id="navbarCollapse">
                            <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                            </ul>
                        </div>
                    </div>
                </nav>
            </header>
            @RenderBody()
        </div>
    </body>
</html>
```

Ci rechiamo nel List.cshtml e lo modifichiamo nel seguente modo:

[Home](#)

```
List.cshtml ✘ ×
BethanyPieShop
1  @model PieListViewModel
2
3  <h1>@Model.CurrentCategory</h1>
4
5  <div class="row row-cols-1 row-cols-md-3 g-4">
6
7      @foreach (var pie in Model.Pies)
8      {
9          <div class="col">
10             <div class="card pie-card">
11                 
12                 <div class="card-body pie-button">
13                     <h4 class="d-grid">
14                         </h4>
15
16                     <div class="d-flex justify-content-between mt-2">
17                         <h2 class="text-start">
18                             <a class="pie-link" href="#">@pie.Name</a>
19                         </h2>
20                         <h5 class="textnowrap">
21                             @pie.Price.ToString("c")
22                         </h5>
23                     </div>
24                 </div>
25             </div>
26         </div>
27     }
28
29 </div>
```

Eseguiamo per verificare che il tutto sia stato integrato correttamente (naturalmente dovremmo impostare manualmente la route nella barra degli indirizzi in quanto non è stata ancora configurata, nel nostro caso: <http://localhost:5000/Pie/List>)



Working with Real Data Using Entity Framework Core

- Introducing Entity Framework Core
- Adding EF Core to the Application
- Demo: Adding EF Core to the Application
- Creating the Repository
- Demo: Creating the Repository
- Using Migrations
- Demo: Creating the Database Using Migrations
- Demo Adding Seed Data

Introducing Entity Framework Core



**Nearly all web applications
you build will need data from a database.**



While we can use low-level ADO.NET combined with SQL statements, we will use Entity Framework Core.

ADO.NET, we can bring in raw SQL statements or invoke a stored

Introducing Entity Framework Core



Osservazione: ORM "Object relational mapper"

EF Core



What EF Core Does for You

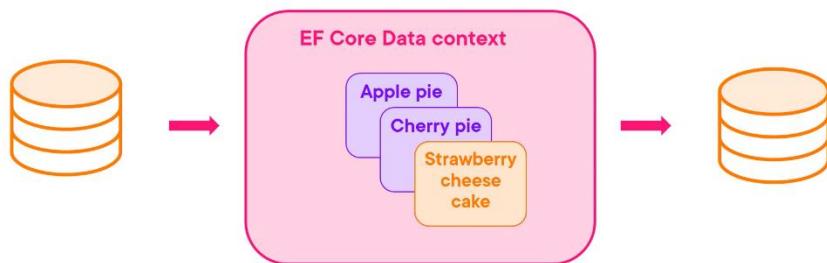
Class

```
public class Pie
{
    public int PieId { get; set; }
    public string? Name { get; set; }
    public string? Description { get; set; }
}
```

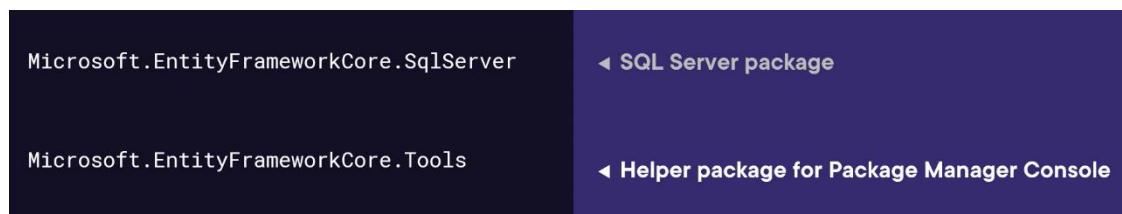
Table

Field	Type
Name	String
Description	string

The EF Core Change Tracker



Adding EF Core to the Application



Considerando il nostro progetto Bethany's Pie Shop:

Creating the Mapping



Table name and column name



Field will become primary key



CategoryId will become foreign key



Column types used in database

The Database Context

```
public class BethanysPieShopDbContext : DbContext
{
    public BethanysPieShopDbContext
        (DbContextOptions<BethanysPieShopDbContext> options)
        : base(options)
    {
    }

    public DbSet<Pie> Pies { get; set; }
}
```

Adding EF Core to the Application

Packages

Domain classes

Database context

Application configuration

we'll need to perform some additional configuration

```
{
  "ConnectionStrings": {
    "BethanysPieShopDbContextConnection": {
      "Server=(localdb)\\mssqllocaldb",
      "Database=BethanysPieShop",
      "Trusted_Connection=True",
      "MultipleActiveResultSets=true"
    }
  }
}
```

Adding the Connection String

appSettings.json

Read automatically by default

We'll need a way to let our code know how we can connect

```
builder.Services.AddDbContext<BethanysPieShopDbContext>(
    options => {
        options.UseSqlServer(
            builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
    }
);
```

Registering the Database Context

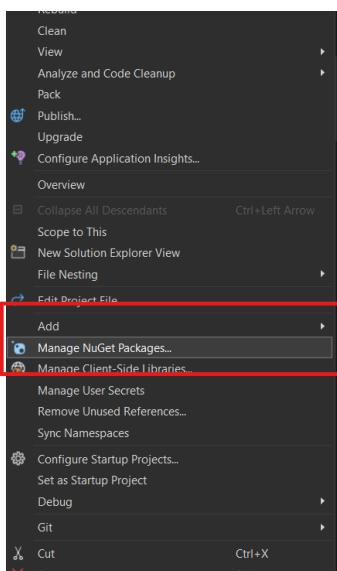
AddDbContext is an extension method

We also will need to add some configuration code in the program.cs. We have

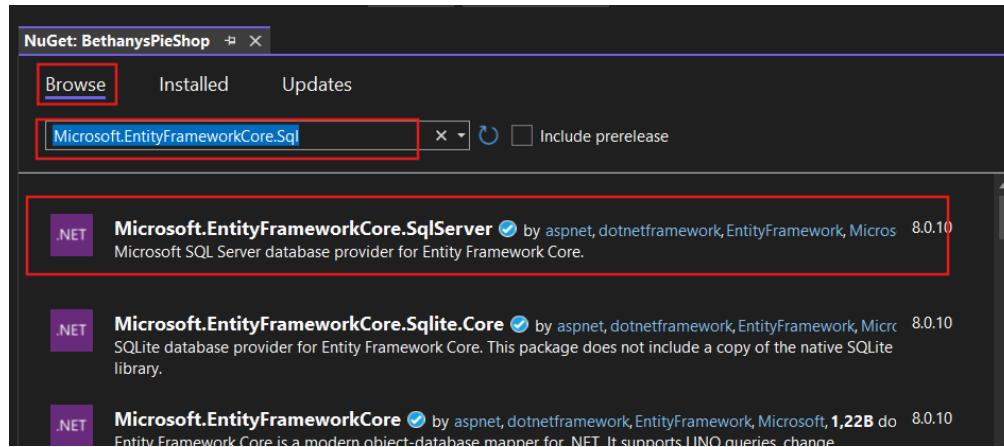
Demo: Adding EF Core to the Application

Inseriamo i NuGet packages per lavorare con EF Core.

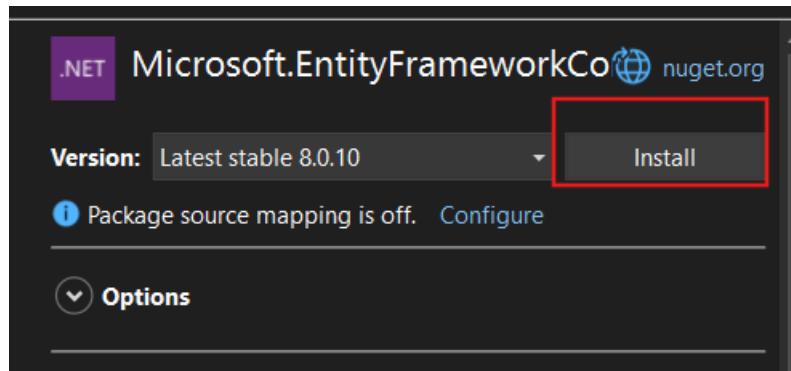
Click destro sul progetto > manage NuGet packages...



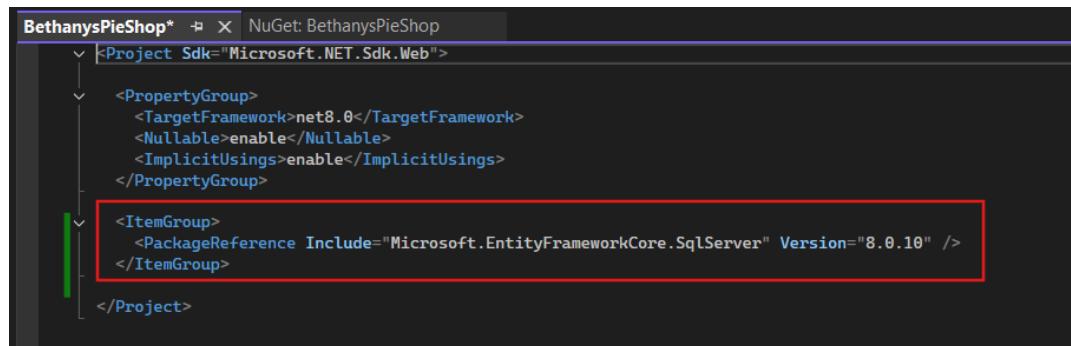
Home



Click destro > Installa

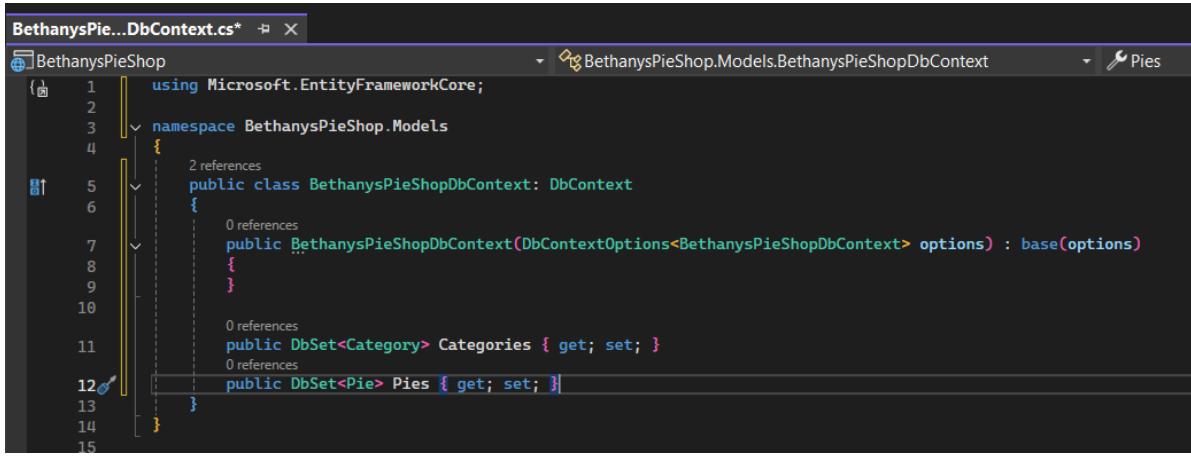


Nel nostro csproj noteremo:



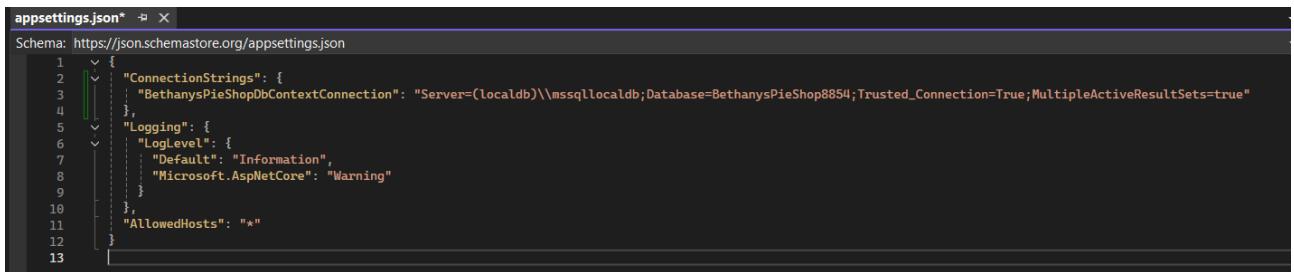
Ora creiamo nella cartella Models la classe BethanyPieShopDbContext.cs:

Home



```
BethanysPie...DbContext.cs* □ X
BethanysPieShop
1  using Microsoft.EntityFrameworkCore;
2
3  namespace BethanysPieShop.Models
4  {
5      public class BethanysPieShopDbContext : DbContext
6      {
7          public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
8          {
9          }
10
11         public DbSet<Category> Categories { get; set; }
12         public DbSet<Pie> Pies { get; set; }
13     }
14 }
15
```

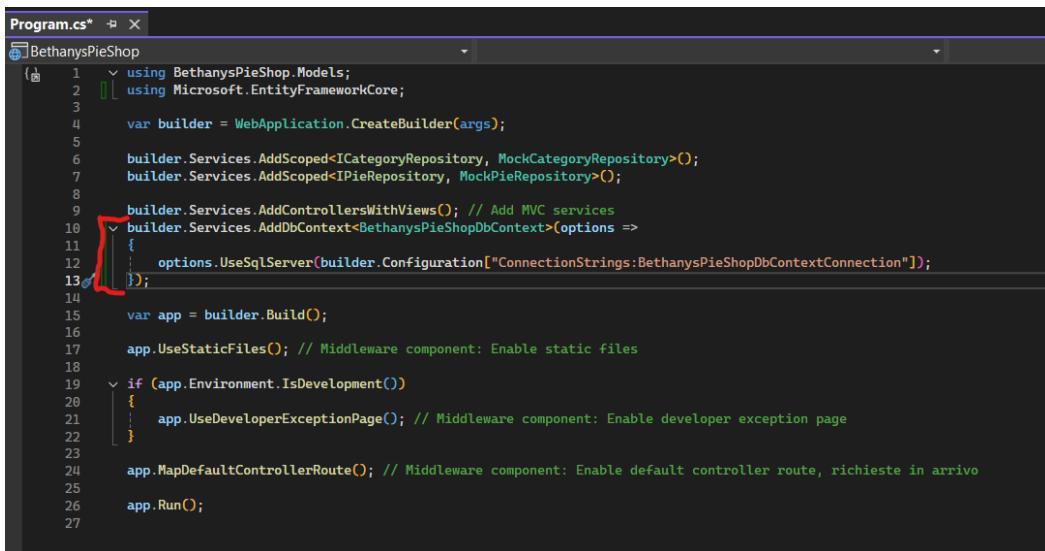
Modifichiamo nel seguente modo il file appsettings.json, in modo da poter avere la connectionStrings del DbContextConnection:



```
appsettings.json* □ X
Schema: https://json.schemastore.org/appsettings.json
1  {
2      "ConnectionStrings": {
3          "BethanysPieShopDbContextConnection": "Server=(localdb)\\mssqllocaldb;Database=BethanysPieShop8854;Trusted_Connection=True;MultipleActiveResultSets=true"
4      },
5      "Logging": {
6          "LogLevel": {
7              "Default": "Information",
8              "Microsoft.AspNetCore": "Warning"
9          }
10     },
11     "AllowedHosts": "*"
12 }
```

Se dovessi usare SQL Server Management studio 2022 il ConnectionStrings sarà differente

Aggiungiamo il servizio in Program.cs:



```
Program.cs* □ X
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, MockPieRepository>();
8
9  builder.Services.AddControllersWithViews(); // Add MVC services
10 builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
11 {
12     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
13 });
14
15 var app = builder.Build();
16
17 app.UseStaticFiles(); // Middleware component: Enable static files
18
19 if (app.Environment.IsDevelopment())
20 {
21     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22 }
23
24 app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
25
26 app.Run();
27
```

Creating the Repository

```
_bethanysPieShopDbContext.Pies.Include(c => c.Category).Where(p => p.IsPieOfTheWeek);
```

Querying for Data Using LINQ

Adding New Items

```
foreach (ShoppingCartItem? shoppingCartItem in shoppingCartItems)
{
    var orderDetail = new OrderDetail
    {
        Amount = shoppingCartItem.Amount,
        PieId = shoppingCartItem.Pie?.PieId,
        Price = shoppingCartItem.Pie?.Price
    };

    order.OrderDetails.Add(orderDetail);
}

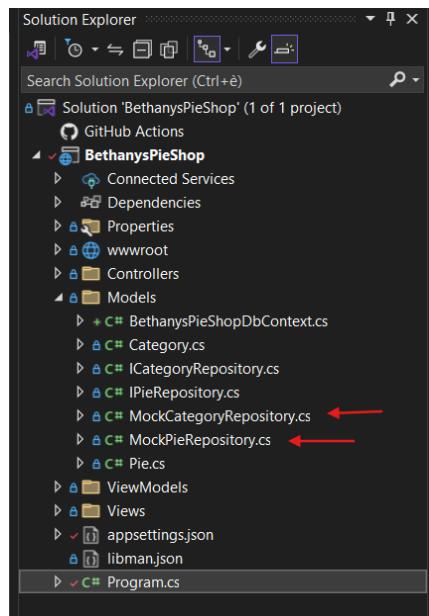
_bethanysPieShopDbContext.Orders.Add(order);

_bethanysPieShopDbContext.SaveChanges();
```

Demo: Creating the Repository

Inizialmente abbiamo usato delle classi mock per poter simulare l'utilizzo dei dati estratti dal database:

Home



Adesso utilizzeremo una PieRepository.cs reale, quindi creiamo questa classe nella cartella Models:

```
namespace BethanysPieShop.Models
{
    public class PieRepository
    {
    }
}
```

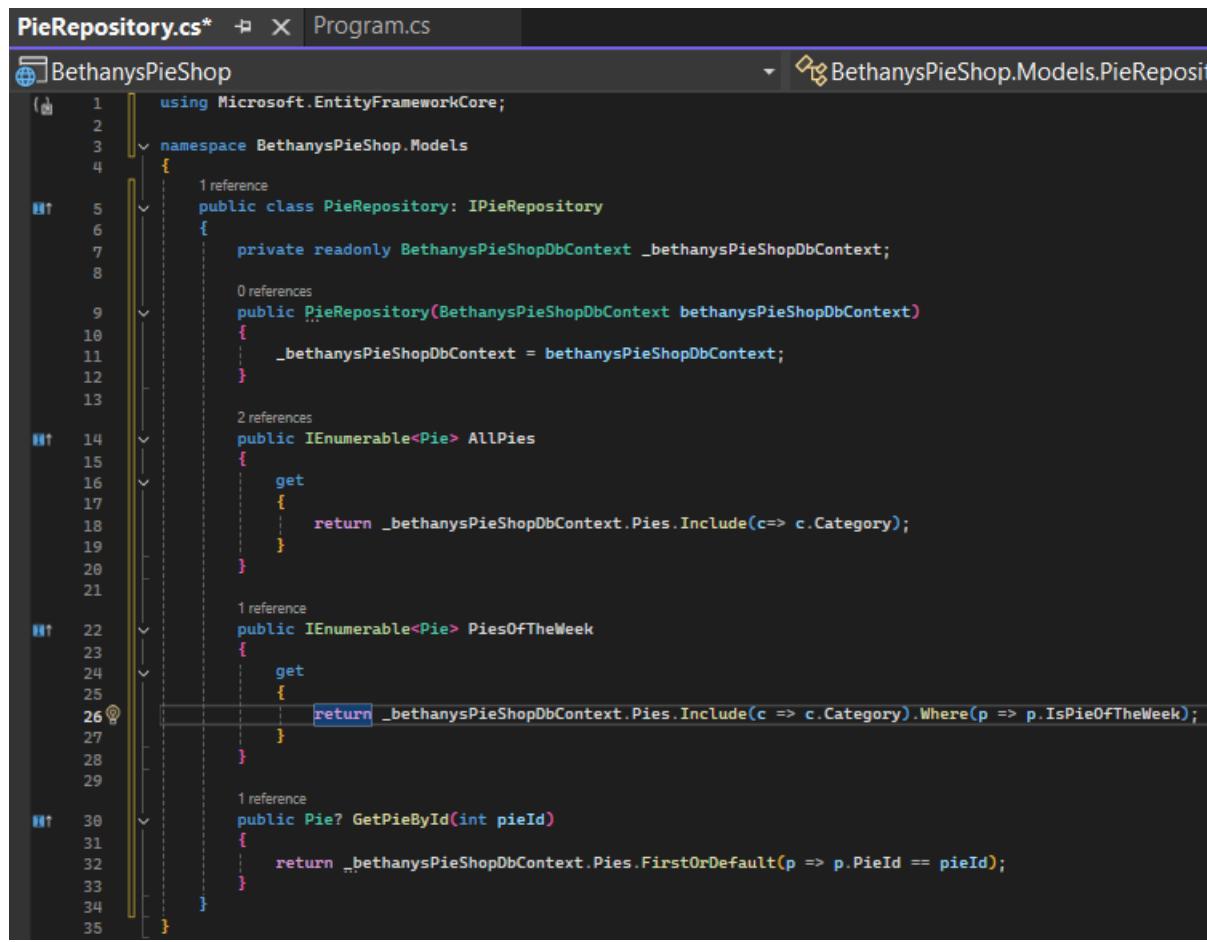
Modifichiamo:

```
namespace BethanysPieShop.Models
{
    public class PieRepository: IPieRepository
    {
        private readonly BethanysPieShopDbContext bethanysPieShopDbContext;
    }
}
```

Dato che abbiamo aggiunto il servizio del DbContext possiamo effettuare usare la DI (riga 5)

Continuiamo la modifica del file nel seguente modo:

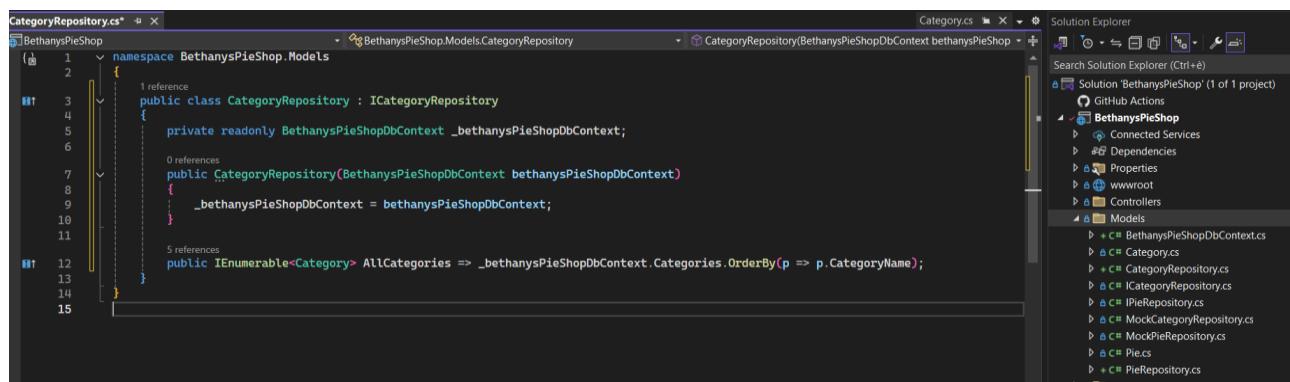
Home



The screenshot shows the Visual Studio code editor with the file `PieRepository.cs` open. The code implements a repository pattern for managing pie data. It includes methods to get all pies, get pies of the week, and get a pie by its ID. The code uses Entity Framework Core to interact with a database context.

```
1  using Microsoft.EntityFrameworkCore;
2
3  namespace BethanysPieShop.Models
4  {
5      public class PieRepository : IPieRepository
6      {
7          private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;
8
9          public PieRepository(BethanysPieShopDbContext bethanysPieShopDbContext)
10         {
11             _bethanysPieShopDbContext = bethanysPieShopDbContext;
12         }
13
14         public IEnumerable<Pie> AllPies
15         {
16             get
17             {
18                 return _bethanysPieShopDbContext.Pies.Include(c => c.Category);
19             }
20         }
21
22         public IEnumerable<Pie> PiesOfTheWeek
23         {
24             get
25             {
26                 return _bethanysPieShopDbContext.Pies.Include(c => c.Category).Where(p => p.IsPieOfTheWeek);
27             }
28         }
29
30         public Pie? GetPieById(int pieId)
31         {
32             return _bethanysPieShopDbContext.Pies.FirstOrDefault(p => p.PieId == pieId);
33         }
34     }
35 }
```

Procediamo in egual modo nella creazione di `CategoryRepository.cs` e implementiamolo il seguente codice:

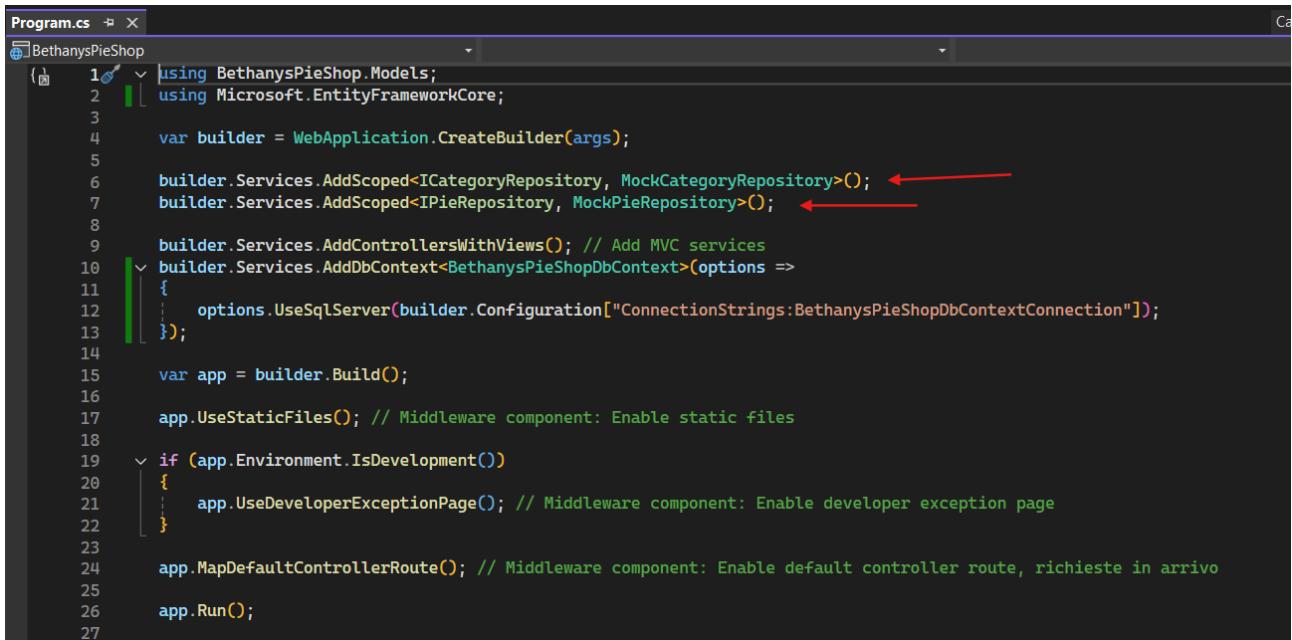


The screenshot shows the Visual Studio code editor with the file `CategoryRepository.cs` open. The code implements a repository pattern for managing category data. It includes a method to get all categories, ordered by name. The code uses Entity Framework Core to interact with a database context.

```
1  using Microsoft.EntityFrameworkCore;
2
3  namespace BethanysPieShop.Models
4  {
5      public class CategoryRepository : ICategoryRepository
6      {
7          private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;
8
9          public CategoryRepository(BethanysPieShopDbContext bethanysPieShopDbContext)
10         {
11             _bethanysPieShopDbContext = bethanysPieShopDbContext;
12         }
13
14         public IEnumerable<Category> AllCategories => _bethanysPieShopDbContext.Categories.OrderBy(p => p.CategoryName);
15     }
16 }
```

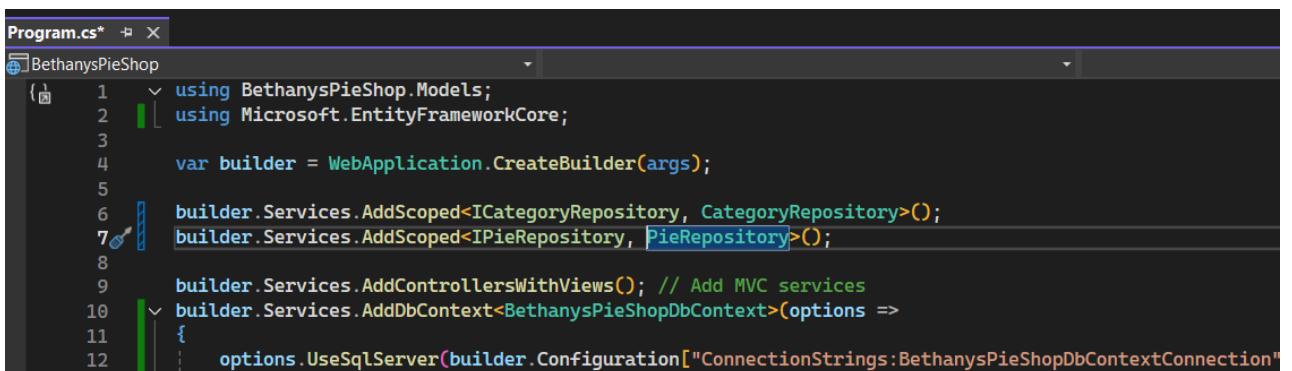
Ritorniamo al `Program.cs`, dove avevamo configurato i vecchi Mock:

Home



```
Program.cs
BethanyPieShop
1 using BethanyPieShop.Models;
2 using Microsoft.EntityFrameworkCore;
3
4 var builder = WebApplication.CreateBuilder(args);
5
6 builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>(); ←
7 builder.Services.AddScoped<IPieRepository, MockPieRepository>(); ←
8
9 builder.Services.AddControllersWithViews(); // Add MVC services
10 builder.Services.AddDbContext<BethanyPieShopDbContext>(options =>
11 {
12     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanyPieShopDbContextConnection"]);
13 });
14
15 var app = builder.Build();
16
17 app.UseStaticFiles(); // Middleware component: Enable static files
18
19 if (app.Environment.IsDevelopment())
20 {
21     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22 }
23
24 app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
25
26 app.Run();
27
```

Cancello semplicemente la parola Mock:



```
Program.cs*
BethanyPieShop
1 using BethanyPieShop.Models;
2 using Microsoft.EntityFrameworkCore;
3
4 var builder = WebApplication.CreateBuilder(args);
5
6 builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
7 builder.Services.AddScoped<IPieRepository, PieRepository>();
8
9 builder.Services.AddControllersWithViews(); // Add MVC services
10 builder.Services.AddDbContext<BethanyPieShopDbContext>(options =>
11 {
12     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanyPieShopDbContextConnection"]);
13 })
```

Osservazione: Ciò ci consente di evitare di modificare altri componenti come il Controller.

Se proviamo ad avviare ora il nostro programma e giungiamo all'url solito, noteremo questo output sul nostro browser:

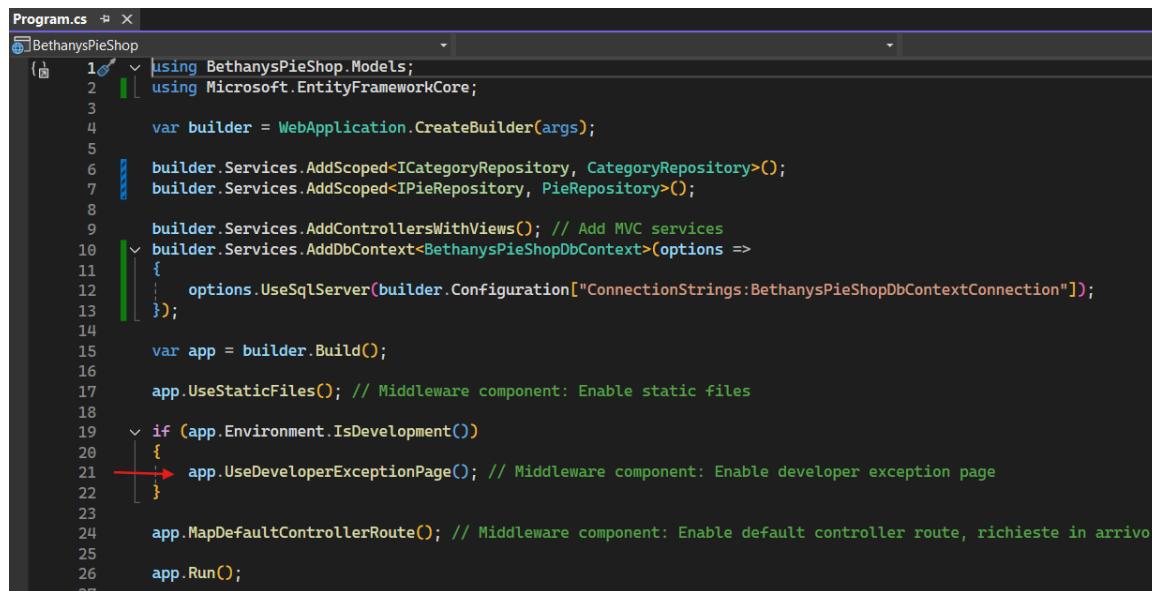
An unhandled exception occurred while processing the request.

```
SqlException: Cannot open database "BethanyPieShop8854" requested by the login. The login failed.  
Login failed for user [REDACTED]  
Microsoft.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, bool breakConnection, Action<Action> wrapCloseInAction)  
  
InvalidOperationException: An exception has been raised that is likely due to a transient failure. Consider enabling transient error resiliency by adding 'EnableRetryOnFailure' to the 'UseSqlServer' call.  
Microsoft.EntityFrameworkCore.SqlServer.Storage.Internal.SqlServerExecutionStrategy.Execute<TState, TResult>(TState state, Func<DbContext, TState, TResult> operation, Func<DbContext, TState, ExecutionResult<TResult>> verifySucceeded)  
  
Stack Query Cookies Headers Routing  
  
SqlException: Cannot open database "BethanyPieShop8854" requested by the login. The login failed. Login failed for user [REDACTED]  
Microsoft.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, bool breakConnection, Action<Action> wrapCloseInAction)
```

[Home](#)

Ovvero, il nostro DB identificato nella ConnectionString non è stato trovato.
Inoltre ha fallito il log in, è del tutto normale, non abbiamo ancora un DB.

Inoltre, questo out siamo riusciti a ricavarlo grazie al comando aggiunto nel nostro Program.cs, il messaggio inoltre sarà mostrato solo in fase di sviluppo, non quando il nostro programma è in produzione o release:



```
Program.cs
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, PieRepository>();
8
9  builder.Services.AddControllersWithViews(); // Add MVC services
10 builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
11 {
12     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
13 });
14
15 var app = builder.Build();
16
17 app.UseStaticFiles(); // Middleware component: Enable static files
18
19 if (app.Environment.IsDevelopment())
20 {
21     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22 }
23
24 app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
25
26 app.Run();
27
```

Using Migrations

La migrazione è un codice che permette di poter sincronizzare il database al modello

La migrazione necessita di due comandi del package Manager Console:

```
>add-migrations <MigrationName>
>update-database
```

Demo: Creating Database using migrations

Ricordandoci di aver installato i seguenti packages nel nostro progetto:

[Home](#)

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

Ora utilizzeremo i comandi di migrazione, quindi ci rechiamo in
View > Other Windows > Package Manager Console

Digitiamo quindi il seguente codice:



PM> add-migration InitialMigration

(Se non viene riconosciuto il comando `add-migration <name of migration>` che verrà inserito a breve è perché non è stato installato correttamente il seguente pacchetto dal PM:

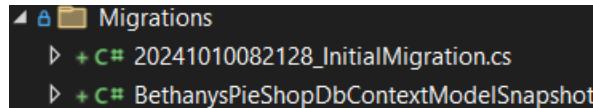
Install-Package Microsoft.EntityFrameworkCore.Tools)

Output:

```
Package Manager Console
Package source: All | Default project: BethanysPieShop
PM> add-migration InitialMigration
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
  No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
  No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
  No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
  To undo this action, use Remove-Migration.
PM>
```

[Home](#)

Possiamo notare nel Solution Explorer che è stata generata una cartella Migrations:



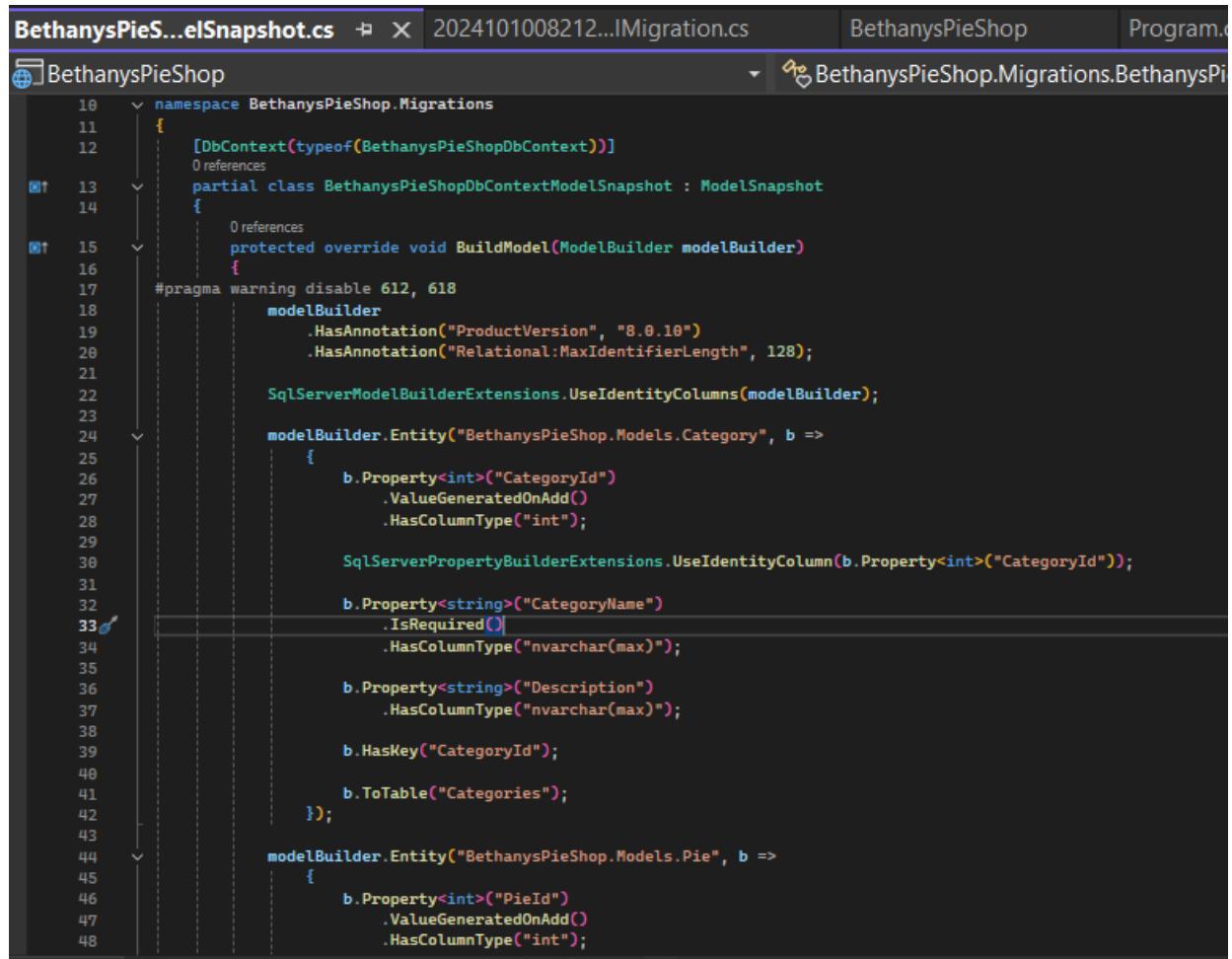
Prima parte di 2024101008218_InitialMigration.cs:

```
20241010082...Migration.cs  ✘ X  BethanysPieShop  Program.cs
BethanysPieShop
  ↴ 1  using Microsoft.EntityFrameworkCore.Migrations;
  2
  3  #nullable disable
  4
  5  namespace BethanysPieShop.Migrations
  6  {
  7      /// <inheritdoc />
  8      public partial class InitialMigration : Migration
  9      {
 10          /// <inheritdoc />
 11          protected override void Up(MigrationBuilder migrationBuilder) ←
 12          {
 13              migrationBuilder.CreateTable(
 14                  name: "Categories",
 15                  columns: table => new
 16                  {
 17                      CategoryId = table.Column<int>(type: "int", nullable: false)
 18                          .Annotation("SqlServer:Identity", "1, 1"),
 19                      CategoryName = table.Column<string>(type: "nvarchar(max)", nullable: false),
 20                      Description = table.Column<string>(type: "nvarchar(max)", nullable: true)
 21                  },
 22                  constraints: table =>
 23                  {
 24                      table.PrimaryKey("PK_Categories", x => x.CategoryId);
 25                  });
 26
 27              migrationBuilder.CreateTable(
 28                  name: "Pies",
 29                  columns: table => new
 30                  {
 31                      PieId = table.Column<int>(type: "int", nullable: false)
 32                          .Annotation("SqlServer:Identity", "1, 1"),
 33                      Name = table.Column<string>(type: "nvarchar(max)", nullable: false),
 34                      ShortDescription = table.Column<string>(type: "nvarchar(max)", nullable: true),
 35                      LongDescription = table.Column<string>(type: "nvarchar(max)", nullable: true),
 36                      AllergyInformation = table.Column<string>(type: "nvarchar(max)", nullable: true),
 37                      Price = table.Column<decimal>(type: "decimal(18,2)", nullable: false),
 38                      ImageUrl = table.Column<string>(type: "nvarchar(max)", nullable: true),
 39                      ImageThumbnailUrl = table.Column<string>(type: "nvarchar(max)", nullable: true),
 40                  });
 41          }
 42      }
 43  }
```

Possiamo notare due metodi, il metodo **Up** conterrà il codice per sincronizzare il database, per questo sarà necessario creare la tabella delle categorie con questi campi, così come la tabella delle torte con altri campi e vincoli, come le chiavi primarie ed esterne, ottenute da EF Core conventions.

L'altro file generato, BethanysPieShopDbContextModelSnapshot.cs serve per tenere traccia di dove ci troviamo a questo punto è uno snapshot:

[Home](#)



The screenshot shows a code editor with the file `BethanysPieShopSnapshot.cs` open. The code is part of the `BethanysPieShop.Migrations` namespace and defines a `BethanysPieShopDbContextModelSnapshot` class that implements `ModelSnapshot`. The class contains logic for building database models for the `Category` and `Pie` entities. The `Category` entity has properties for `CategoryId`, `CategoryName`, and `Description`, with `CategoryId` being the primary key. The `Pie` entity has a property for `PieId`.

```
10  namespace BethanysPieShop.Migrations
11  {
12      [DbContext(typeof(BethanysPieShopDbContext))]
13      partial class BethanysPieShopDbContextModelSnapshot : ModelSnapshot
14      {
15          protected override void BuildModel(ModelBuilder modelBuilder)
16          {
17              #pragma warning disable 612, 618
18              modelBuilder
19                  .HasAnnotation("ProductVersion", "8.0.10")
20                  .HasAnnotation("Relational:MaxIdentifierLength", 128);
21
22              SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder);
23
24              modelBuilder.Entity("BethanysPieShop.Models.Category", b =>
25              {
26                  b.Property<int>("CategoryId")
27                      .ValueGeneratedOnAdd()
28                      .HasColumnType("int");
29
30                  SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("CategoryId"));
31
32                  b.Property<string>("CategoryName")
33                      .IsRequired()
34                      .HasColumnType("nvarchar(max)");
35
36                  b.Property<string>("Description")
37                      .HasColumnType("nvarchar(max)");
38
39                  b.HasKey("CategoryId");
40
41                  b.ToTable("Categories");
42              });
43
44              modelBuilder.Entity("BethanysPieShop.Models.Pie", b =>
45              {
46                  b.Property<int>("PieId")
47                      .ValueGeneratedOnAdd()
48                      .HasColumnType("int");
49              });
50          }
51      }
52  }
```

Ora aggiorniamo il Database, quindi ritorniamo al nostro Package Manager Console e digitiamo **updatedatabase**

Output:

[Home](#)

```
To undo this action, use REMOVE-MIGRATION.
PM> update-database ←
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
    No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will
precision and scale. Explicitly specify the SQL server column type that can accommodate all the val
scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
    No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will
precision and scale. Explicitly specify the SQL server column type that can accommodate all the val
scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause
precision and scale. Explicitly specify the SQL server column type that can accommodate all the val
scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (187ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
        CREATE DATABASE [BethanyPieShop8854];
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (51ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
        IF SERVERPROPERTY('EngineEdition') <> 5
            RECONFIGURE;
%
```

```
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
        CREATE TABLE [Pies] (
            [PieId] int NOT NULL IDENTITY,
            [Name] nvarchar(max) NOT NULL,
            [ShortDescription] nvarchar(max) NULL,
            [LongDescription] nvarchar(max) NULL,
            [AllergyInformation] nvarchar(max) NULL,
            [Price] decimal(18,2) NOT NULL,
            [ImageUrl] nvarchar(max) NULL,
            [ImageThumbnailUrl] nvarchar(max) NULL,
            [IsPieOfTheWeek] bit NOT NULL,
            [InStock] bit NOT NULL,
            [CategoryId] int NOT NULL,
            CONSTRAINT [PK_Pies] PRIMARY KEY ([PieId]),
            CONSTRAINT [FK_Pies_Categories_CategoryId] FOREIGN KEY ([CategoryId]) REFERENCES [Categories] ([CategoryId]) ON DELETE CASCADE
        );
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
        CREATE INDEX [IX_Pies_CategoryId] ON [Pies] ([CategoryId]);
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
        INSERT INTO [_EFMigrationsHistory] ([MigrationId], [ProductVersion])
        VALUES (N'20241010082128_InitialMigration', N'8.0.10');
Done.
PM> |
```

Il nostro database è stato generato, in appsetting.json possiamo verificare il suo nome:

```
Schema: https://json.schemastore.org/appsettings.json
1  {
2      "ConnectionStrings": {
3          "BethanyPieShopDbContextConnection": "Server=(localdb)\\mssqllocaldb;Database=BethanyPieShop8854;Trusted_Connection=True;MultipleActiveResultSets=True"
4      },
5      "Logging": {
6          "LogLevel": {
7              "Default": "Information",
8              "Microsoft.AspNetCore": "Warning"
9          }
10     },
11     "AllowedHosts": "*"
12   }
```

[Home](#)

Ora apriamo il pannello per poter visualizzare SQL Server Object Explorer (View > SQL Server Object Explorer)

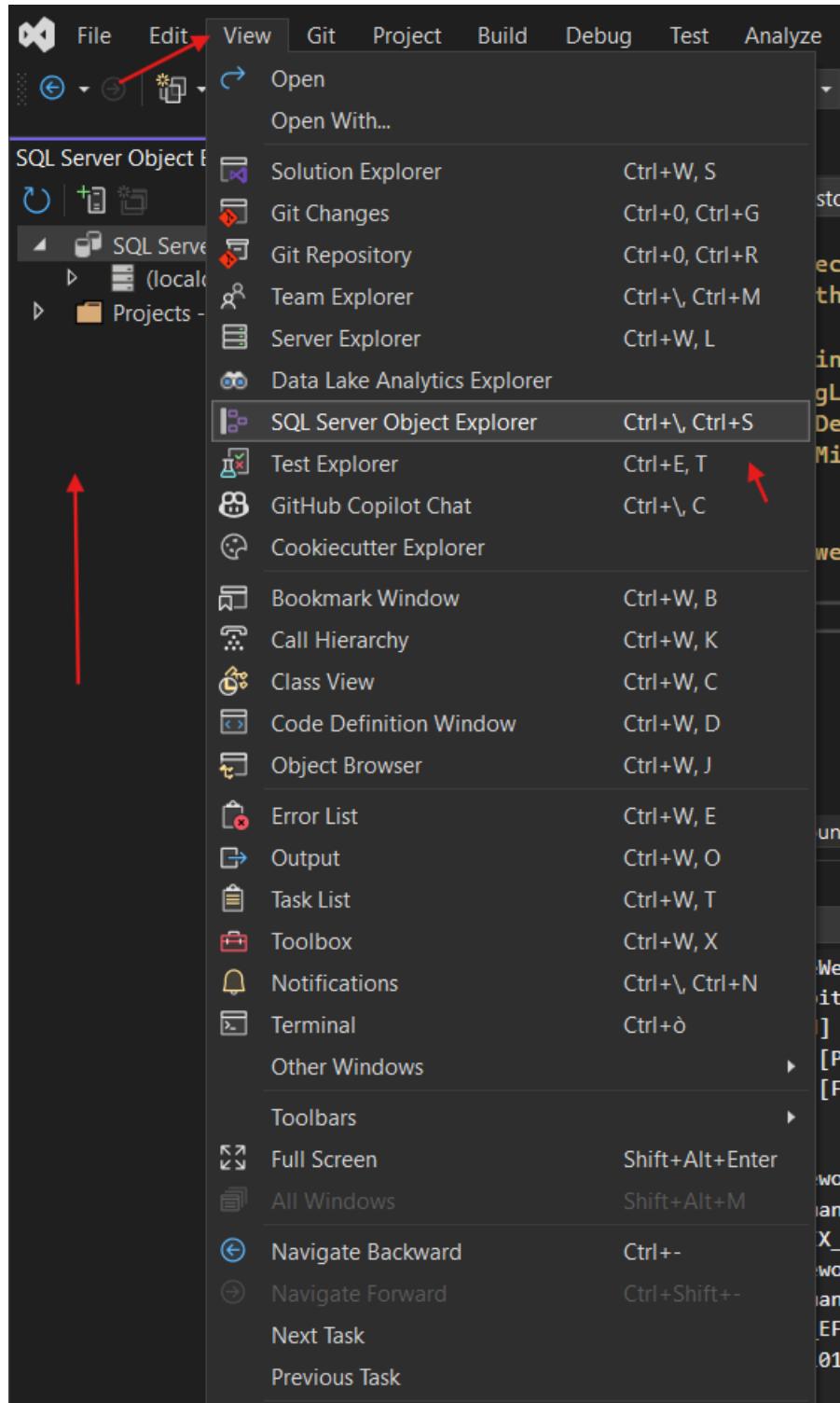
NB: E' possibile poterlo visualizzare se nel Visual Studio Installer è stato installato, è possibile farlo mediante queste istruzioni:

Assicurati di aver installato il componente **Data storage and processing** o **SQL Server Data Tools (SSDT)**, che ti forniscono accesso a SQL Server Object Explorer.

- **Passi:**

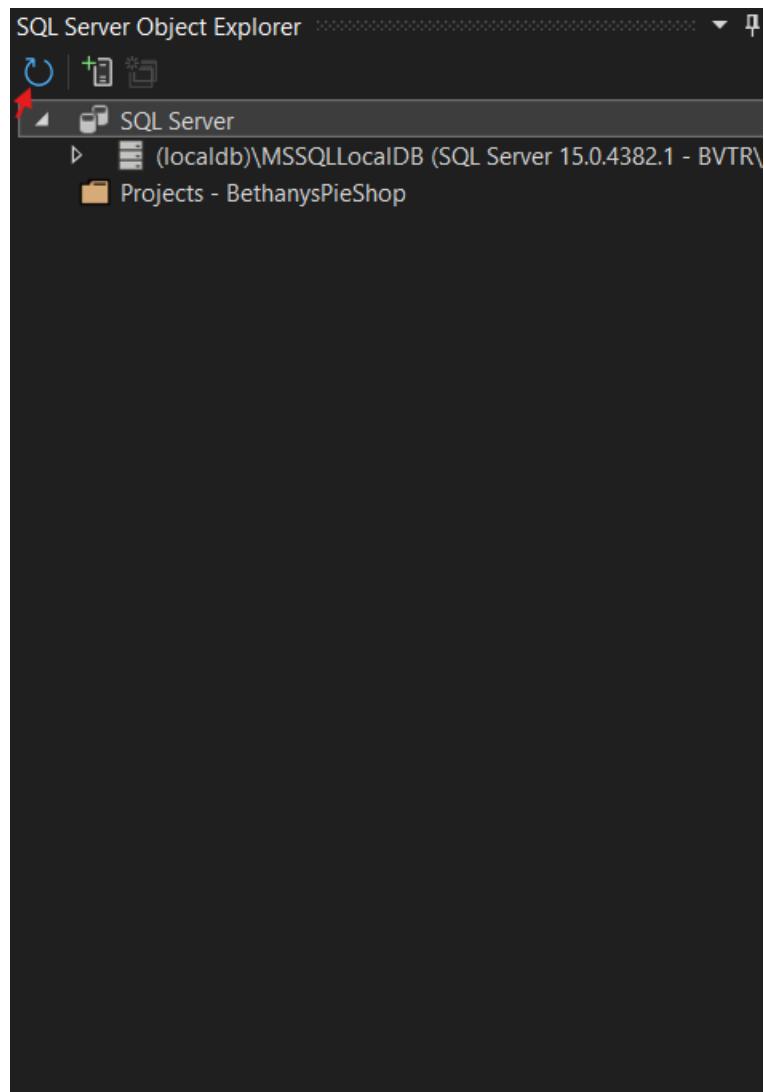
1. Vai su **Visual Studio Installer**.
2. Seleziona **Modifica** sulla tua installazione di Visual Studio.
3. Nella sezione dei **workloads**, assicurati che sia selezionato **Data storage and processing**.
4. Se non è già installato, seleziona il componente e clicca su **Modifica** per installarlo.

[Home](#)

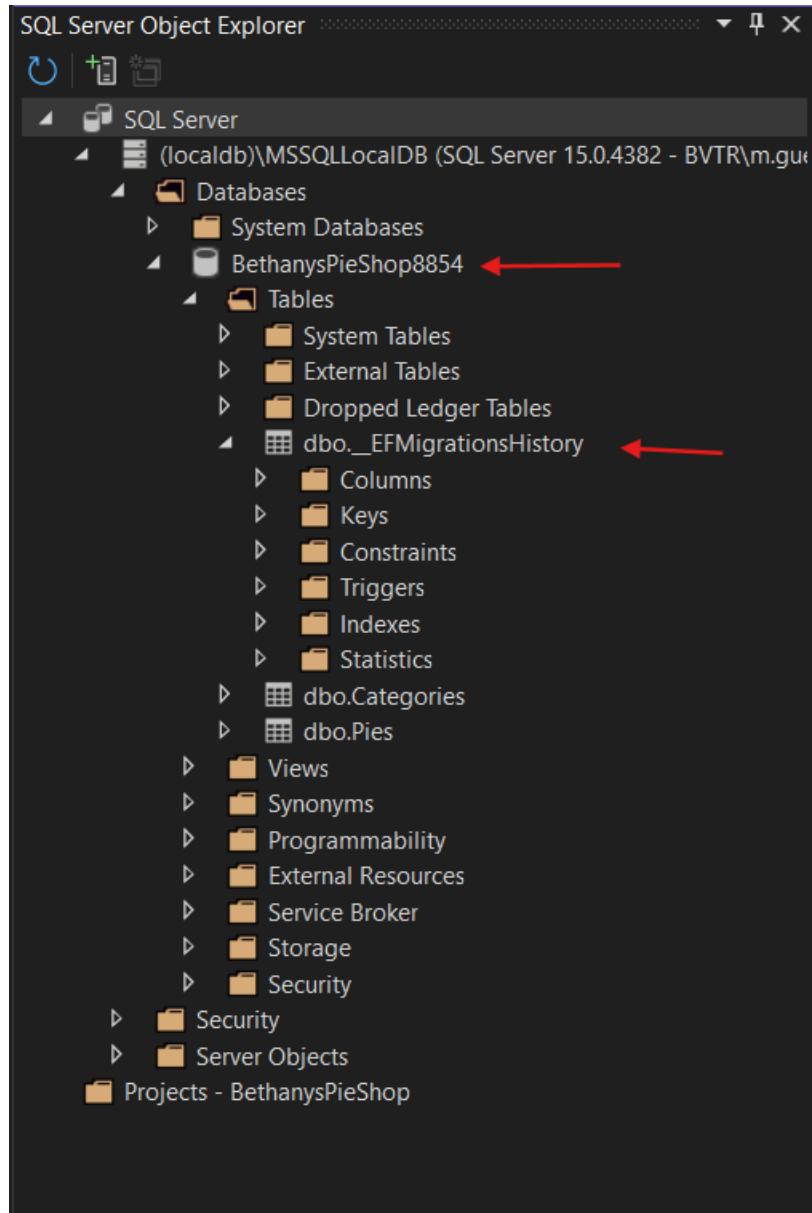


Quindi avremo questa visualizzazione dopo aver cliccato il refresh (freccia blu interno in SQ S.O.E.):

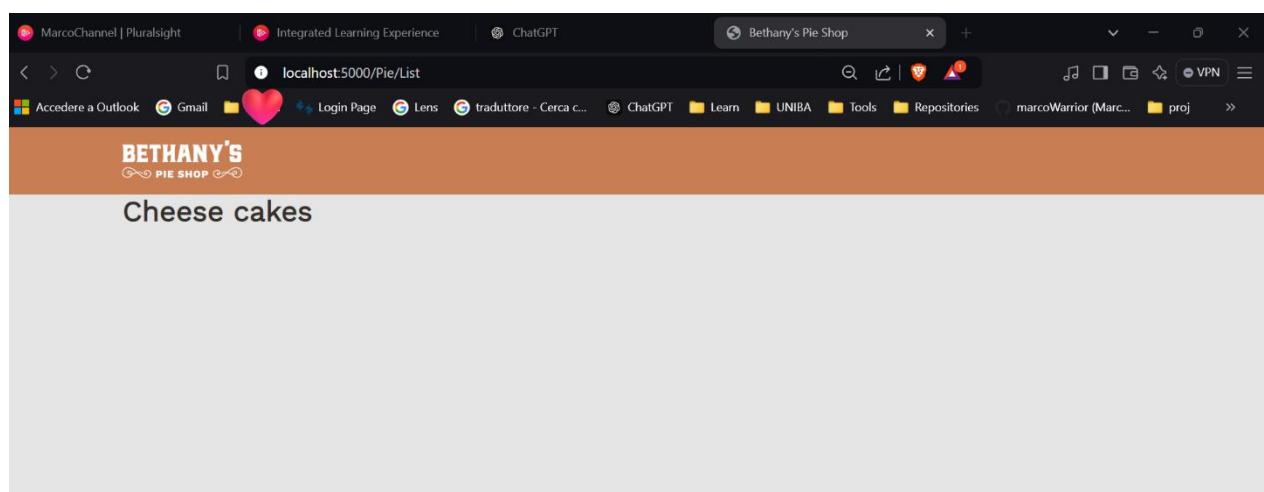
[Home](#)



[Home](#)



Eseguiamo il nostro programma, dove dobbiamo indicare manualmente la classica route e avremo ciò:



[Home](#)

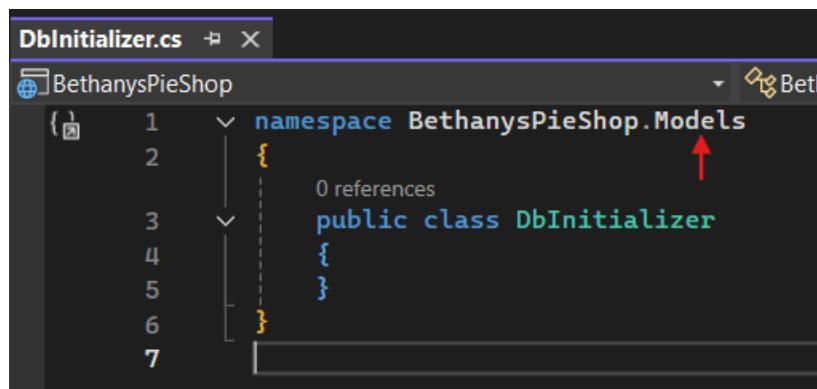
Ciò è normale, in quanto abbiamo creato il Database ma non abbiamo aggiunto ancora alcun dato.

Demo: Adding Seed Data

Questa operazione va fatta quando avviamo un progetto nuovo, inoltre è possibile utilizzare EF Core in combinazione con un database già esistente.

Nel nostro caso abbiamo lasciato che EF Core lo generasse e quindi non disponiamo ancora di dati, quindi ora creiamo alcuni dati seed che verranno caricati nel database.

Utilizzeremo una classe statica che contiene un metodo statico, quindi nel nostro modello creiamo la classe **DbInitializer.cs**



Al suo interno inseriremo una grande quantità di codice

The screenshot shows the 'DbInitializer.cs' code editor with the entire seed data implementation. The code defines a static class 'DbInitializer' within the 'BethanysPieShop.Models' namespace. It includes a 'Seed' method that creates a 'BethanysPieShopDbContext' and adds ranges of 'Category' and 'Pie' entities to the database. The 'Categories' and 'Pies' lists contain numerous entries with names, prices, short descriptions, and long descriptions.

```
namespace BethanysPieShop.Models
{
    public static class DbInitializer
    {
        public static void Seed(IApplicationBuilder applicationBuilder)
        {
            BethanysPieShopDbContext context = applicationBuilder.ApplicationServices.CreateScope().ServiceProvider.GetRequiredService<BethanysPieShopDbContext>();
            if (context.Categories.Any())
            {
                context.Categories.AddRange(Categories.Select(c => c.Value));
            }
            if (context.Pies.Any())
            {
                context.AddRange(Pies);
            }
            context.SaveChanges();
        }
    }
}

private static Dictionary<string, Category> Categories;
private static Dictionary<string, Pie> Pies;

public static void Seed(IApplicationBuilder applicationBuilder)
{
    BethanysPieShopDbContext context = applicationBuilder.ApplicationServices.CreateScope().ServiceProvider.GetRequiredService<BethanysPieShopDbContext>();
    if (context.Categories.Any())
    {
        context.Categories.AddRange(Categories.Select(c => c.Value));
    }
    if (context.Pies.Any())
    {
        context.AddRange(Pies);
    }
    context.SaveChanges();
}

private static Dictionary<string, Category> Categories;
private static Dictionary<string, Pie> Pies;

public static void Seed(IApplicationBuilder applicationBuilder)
{
    BethanysPieShopDbContext context = applicationBuilder.ApplicationServices.CreateScope().ServiceProvider.GetRequiredService<BethanysPieShopDbContext>();
    if (context.Categories.Any())
    {
        context.Categories.AddRange(Categories.Select(c => c.Value));
    }
    if (context.Pies.Any())
    {
        context.AddRange(Pies);
    }
    context.SaveChanges();
}
```

Nella parte iniziale del codice otteniamo l'accesso al DbContext:

Home

```
namespace BethanysPieShop.Models
{
    0 references
    public static class DbInitializer
    {
        0 references
        public static void Seed(IApplicationBuilder applicationBuilder)
        {
            BethanysPieShopDbContext context = applicationBuilder.ApplicationServices.CreateScope()
                .ServiceProvider.GetRequiredService<BethanysPieShopDbContext>();

            if (!context.Categories.Any())
            {
                context.Categories.AddRange(Categories.Select(c => c.Value));
            }

            if (!context.Pies.Any())
            {

```

Osservazione: Non posso utilizzare il DI qui, quindi otterrò il DbContext da applicationBuilder.

Ricordandoci che applicationBuilder contiene l'accesso ai servizi, quindi inizierò a caricare dati e controllerò se ci sono categorie nel mio db (ciclo if presente nell'immagine sopra).

Difatti se stiamo lavorando con un db già esistente l'if non viene preso in considerazione.

(Stesso comportamento avviene nella parte successiva dove si considerano le varie torte)

The screenshot shows the `DbInitializer.cs` file in a code editor. The `Seed` method is highlighted. A red box surrounds the first `if` statement that checks for categories. A red arrow points to the `SaveChanges` call at the bottom of the method. The code lists various pie items with their names, prices, short descriptions, and long descriptions.

```
17    new Pie { Name = "Caramel Popcorn Cheese Cake", Price = 22.95M, ShortDescription = "The ultimate cheese cake", LongDescription = "Icing carrot cake with caramel popcorn and a hint of cheese." },
18    new Pie { Name = "Chocolate Cheese Cake", Price = 19.95M, ShortDescription = "The chocolate lover's dream", LongDescription = "Icing carrot cake with rich chocolate flavor and a hint of cheese." },
19    new Pie { Name = "Pistachio Cheese Cake", Price = 21.95M, ShortDescription = "We're going nuts over this one!", LongDescription = "Icing carrot cake with pistachios and a hint of cheese." },
20    new Pie { Name = "Pecan Pie", Price = 21.95M, ShortDescription = "More pecan than you can handle!", LongDescription = "Icing carrot cake with pecans and a hint of cheese." },
21    new Pie { Name = "Birthday Pie", Price = 29.95M, ShortDescription = "A Happy Birthday with this pie!", LongDescription = "Icing carrot cake with colorful sprinkles and a hint of cheese." },
22    new Pie { Name = "Apple Pie", Price = 12.95M, ShortDescription = "Our famous apple pies!", LongDescription = "Icing carrot cake with fresh apples and a hint of cheese." },
23    new Pie { Name = "Blueberry Cheese Cake", Price = 18.95M, ShortDescription = "You'll love it!", LongDescription = "Icing carrot cake with blueberries and a hint of cheese." },
24    new Pie { Name = "Cheese Cake", Price = 18.95M, ShortDescription = "Plain cheese cake. Plain pleasure.", LongDescription = "Icing carrot cake with cream cheese and a hint of cheese." },
25    new Pie { Name = "Cherry Pie", Price = 15.95M, ShortDescription = "A summer classic!", LongDescription = "Icing carrot cake jelly-o cherries and a hint of cheese." },
26    new Pie { Name = "Christmas Apple Pie", Price = 13.95M, ShortDescription = "Happy holidays with this pie!", LongDescription = "Icing carrot cake with cinnamon and a hint of cheese." },
27    new Pie { Name = "Cranberry Pie", Price = 17.95M, ShortDescription = "A Christmas favorite", LongDescription = "Icing carrot cake jelly-o cranberries and a hint of cheese." },
28    new Pie { Name = "Peach Pie", Price = 15.95M, ShortDescription = "Sweet as peach", LongDescription = "Icing carrot cake jelly-o peaches and a hint of cheese." },
29    new Pie { Name = "Pumpkin Pie", Price = 12.95M, ShortDescription = "Our Halloween favorite", LongDescription = "Icing carrot cake jelly-o pumpkins and a hint of cheese." },
30    new Pie { Name = "Rhubarb Pie", Price = 15.95M, ShortDescription = "My God, so sweet!", LongDescription = "Icing carrot cake jelly-o rhubarb and a hint of cheese." },
31    new Pie { Name = "Strawberry Pie", Price = 15.95M, ShortDescription = "Our delicious strawberry pie!", LongDescription = "Icing carrot cake jelly-o strawberries and a hint of cheese." },
32    new Pie { Name = "Strawberry Cheese Cake", Price = 18.95M, ShortDescription = "You'll love it!", LongDescription = "Icing carrot cake with strawberries and a hint of cheese." },
33    new Pie { Name = "Vanilla Bean Cheesecake", Price = 24.95M, ShortDescription = "A classic vanilla bean cheesecake", LongDescription = "Icing carrot cake with vanilla beans and a hint of cheese." },
34    );
35}
36}
37    context.SaveChanges(); ←
38}
39}
40}
41}
42}
43}

private static Dictionary<string, Category>? categories;
{

```

`context.SaveChanges();` è anche un tracker delle modifiche, quindi aggiunge effettivamente categorie e torte ai set di dati in memoria nel contesto del database.

Quando viene chiamato esaminerà le raccolte in memoria e vedrà che a tutte queste categorie e torte è stato aggiunto lo stato.

Osservando la parte finale del codice:

Home

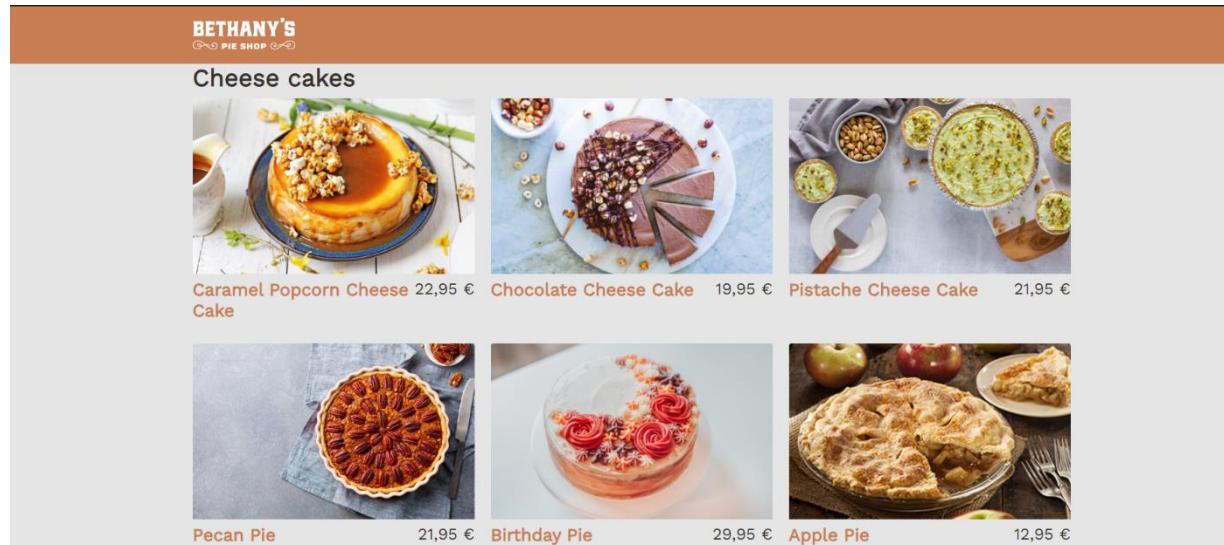
```
38     }
39 
40     private static Dictionary<string, Category>? categories;
41 
42     public static Dictionary<string, Category> Categories
43     {
44         get
45         {
46             if (categories == null)
47             {
48                 var genresList = new Category[]
49                 {
50                     new Category { CategoryName = "Fruit pies" },
51                     new Category { CategoryName = "Cheese cakes" },
52                     new Category { CategoryName = "Seasonal pies" }
53                 };
54 
55                 categories = new Dictionary<string, Category>();
56 
57                 foreach (Category genre in genresList)
58                 {
59                     categories.Add(genre.CategoryName, genre);
60                 }
61             }
62 
63             return categories;
64         }
65     }
66 }
67 }
```

Possiamo notare un Dizionario chiamato Categories, dove ad ogni chiave CategoryName è assegnato il suo valore.

Ora ci rechiamo in **Program.cs** dove chiamerò il DBInitializer.Seed e devo passare app come argomento:

```
8     builder.Services.AddControllersWithViews(); // Add MVC services
9 
10    builder.Services.AddDbContext<BethanyPieShopDbContext>(options =>
11    {
12        options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanyPieShop"]);
13    });
14 
15    var app = builder.Build();
16 
17    app.UseStaticFiles(); // Middleware component: Enable static files
18 
19    if (app.Environment.IsDevelopment())
20    {
21        app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22    }
23 
24    app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route
25    app.UseDbInitializer.Seed(app); // Seed the database ←
26 
27    app.Run();
28 }
```

Salviamo tutto ed eseguiamo ora il nostro programma:



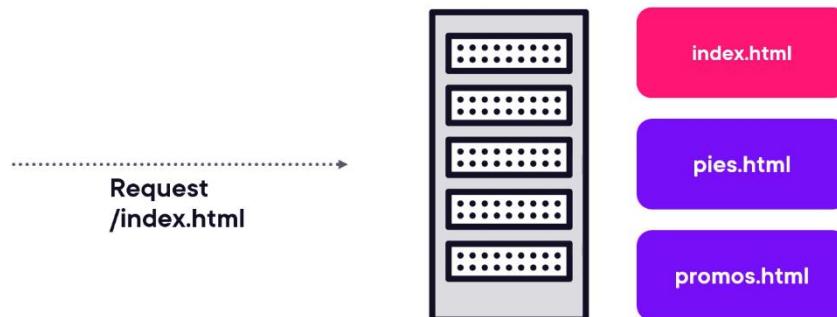
Finalmente siamo in grado di vedere i nostri dati, proveniente da un vero e proprio database.

Adding Routes and Navigation

- Adding Routes and Navigation
- Demo: Adding Routes to the Application
- Configuring Routes
- Navigating with Tag Helpers
- Demo: Adding Navigation to the Site

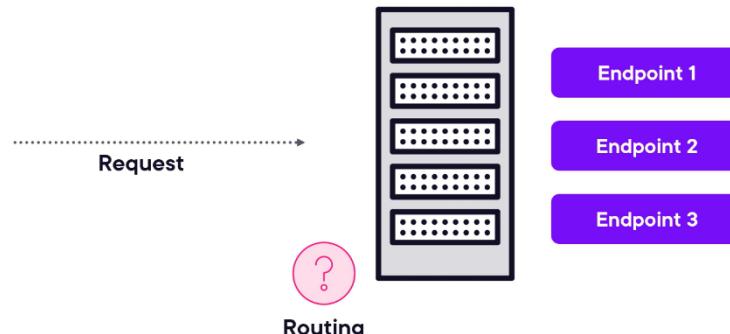
Pensando a un Server Web classico:

Serving Files from a Server



Quando viene effettuata una richiesta per index.html (per esempio), il server web cercherà quel file e un volta trovato verrà spedito al client .
Il file esiste fisicamente sul disco.

Handling Requests in ASP.NET Core



Con MVC, le richieste vengono gestite da metodi di azione del suo controller, che mappano e ritornano il tutto alle View.

L'url ha effettivamente come target un metodo di azione, non il file .cshtml.

Dato che la richiesta non contiene l'endpoint, nell'MVC è presente il Routing che è fornito con ASP.NET Core.

Il routing è un processo che mapperà la richiesta HTTP in entrata su uno degli endpoint dell'applicazione.

L'endpoint esegue del codice e invia tale risposta che verrà inviata al Client.

Gli endpoint vengono configurati all'avvio dell'applicazione in Program.cs

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

Routing in ASP.NET Core

Osservazione: Metodo HTTP GET che viene inviato all'URL root.

In questo esempio abbiamo creato un singolo endpoint utilizzando **MapGet**, funzionerà solo se viene inviata una richiesta GET, non quando un altro verbo HTTP viene inviato a questo URL.

Endpoints in ASP.NET Core



Executable units that can handle a request



Defined in app using configuration (Program)



Matching process gives access to values from URL of request

Middleware for Routing

`UseRouting()`

`UseEndpoints()`

UseRouting: introduce funzionalità di corrispondenza del percorso nella pipeline

UseEndpoints: responsabile dell'aggiunta dell'esecuzione dell'endpoint

Routing in ASP.NET Core MVC



Match URL of request with template



Defined in Program.cs



Mapped to action on controller



Also used to generate outgoing links

Types of Routing

Convention-based

Attribute-based

Convention-based Routing



{Controller}/{Action}

```
app.MapControllerRoute(name: "default", pattern: "{controller}/{action}");
```

Routing to the Action Method

```
public class PieController : Controller
{
    public ViewResult List()
    {
        ...
    }
}
```

Se dovessimo aver bisogno di più segmenti, immaginiamo di voler visualizzare i dettagli di una torta con ID 1.
L'URL della richiesta potrebbe essere:

Convention-based Routing



```
app.MapControllerRoute( name: "default", pattern: "{controller}/{action}/{id}");
```

In tal caso la richiesta potrebbe essere:

Passing a Value

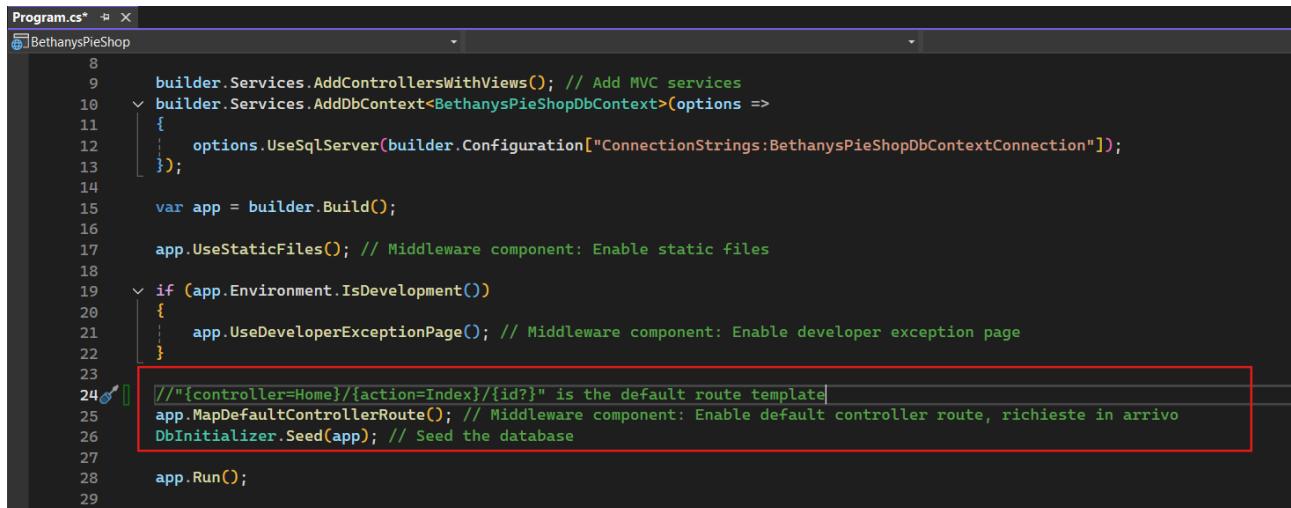
```
public class PieController : Controller
{
    public ViewResult Details(int id)
    {
        //Do something
    }
}
```

```
app.MapDefaultControllerRoute();
```

Using MapDefaultControllerRoute

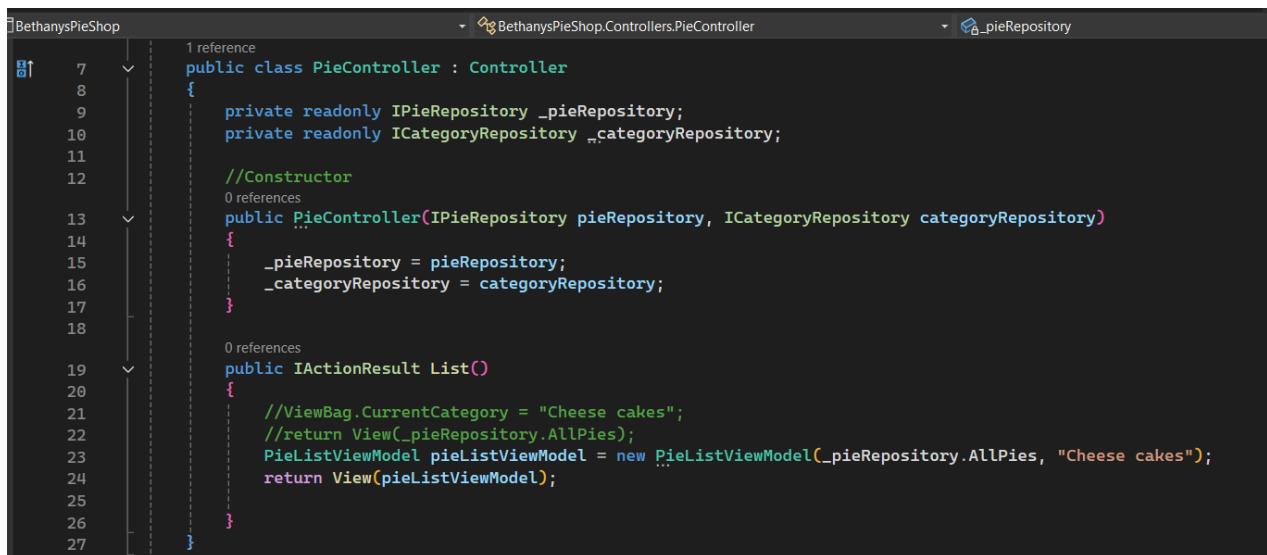
Demo: Adding Routes to the Application

Fino ad ora siamo riusciti comunque a visualizzare le pagine inserendo manualmente l'URL, ciò + stato possibile proprio grazie a tale comando:



```
Program.cs*  ➔ X
BethanyPieShop
8
9     builder.Services.AddControllersWithViews(); // Add MVC services
10    builder.Services.AddDbContext<BethanyPieShopDbContext>(options =>
11    {
12        options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanyPieShopDbContextConnection"]);
13    });
14
15    var app = builder.Build();
16
17    app.UseStaticFiles(); // Middleware component: Enable static files
18
19    if (app.Environment.IsDevelopment())
20    {
21        app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22    }
23
24    //"{controller=Home}/{action=Index}/{id?}" is the default route template
25    app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
26    DbInitializer.Seed(app); // Seed the database
27
28    app.Run();
29
```

Difatti il nostro controller è PieController.cs il quale contiene una action:



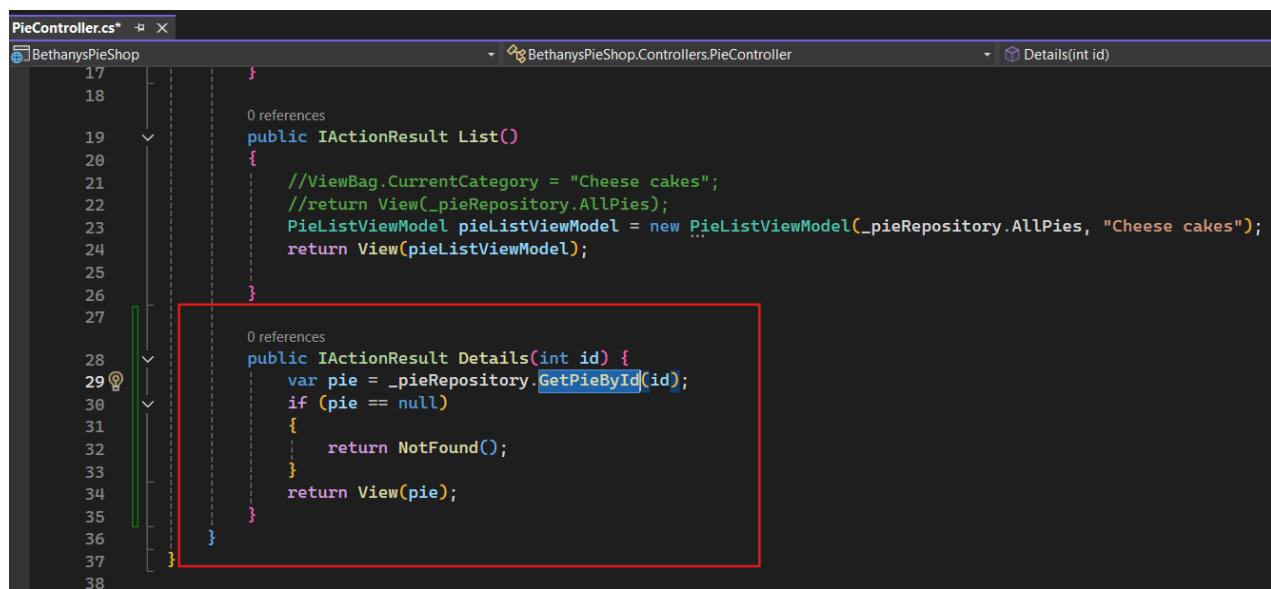
```
BethanyPieShop
BethanyPieShop.Controllers.PieController
pieRepository
1 reference
public class PieController : Controller
{
    private readonly IPieRepository _pieRepository;
    private readonly ICategoryRepository _categoryRepository;

    //Constructor
    0 references
    public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
    {
        _pieRepository = pieRepository;
        _categoryRepository = categoryRepository;
    }

    0 references
    public IActionResult List()
    {
        //ViewBag.CurrentCategory = "Cheese cakes";
        //return View(_pieRepository.AllPies);
        PieListViewModel pieListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
        return View(pieListViewModel);
    }
}
```

[Home](#)

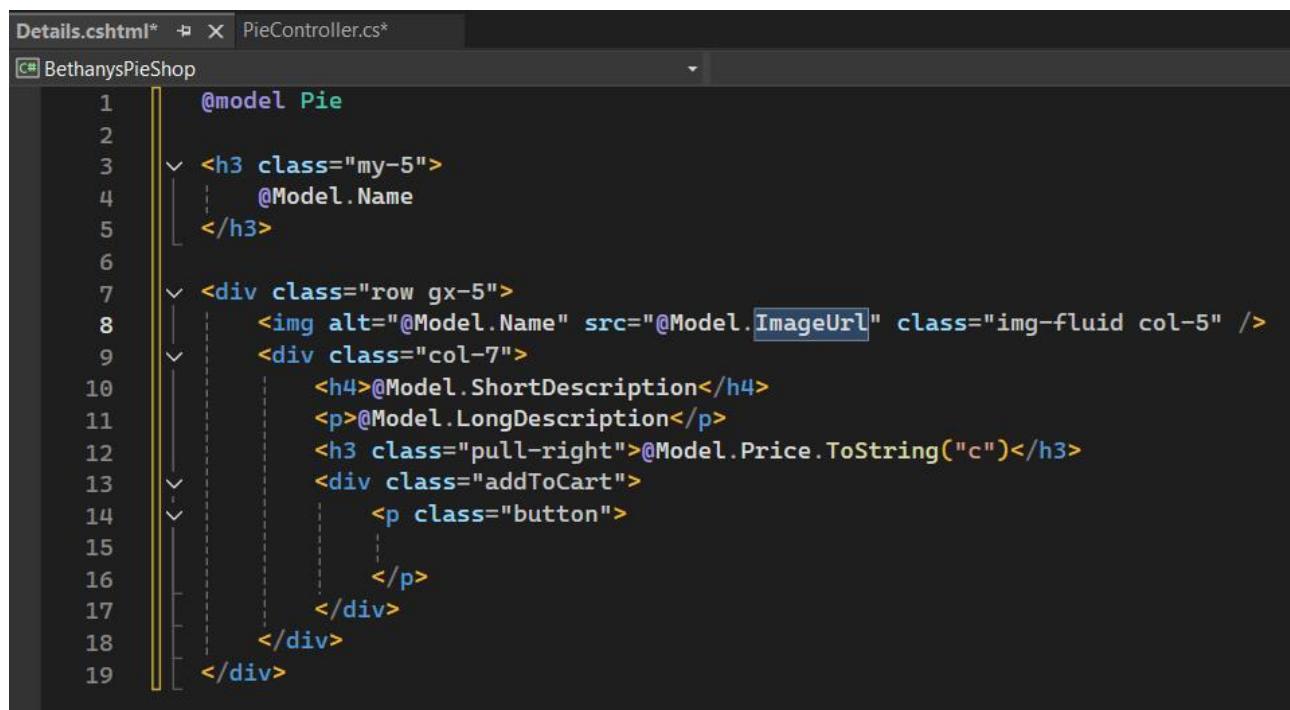
Quindi aggiungiamo ora in PieController il metodo **IActionResult Details ()**:



```
PieController.cs*  ↗ X
BethanyPieShop
17
18
19     }
20
21     public IActionResult List()
22     {
23         //ViewBag.CurrentCategory = "Cheese cakes";
24         //return View(_pieRepository.AllPies);
25         PieListViewModel pieListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
26         return View(pieListViewModel);
27     }
28
29     public IActionResult Details(int id)
30     {
31         var pie = _pieRepository.GetPieById(id);
32         if (pie == null)
33         {
34             return NotFound();
35         }
36         return View(pie);
37     }
38 }
```

Dopodiché, creiamo una nuova vista Razor empty chiamata **Details** in Views > Pie

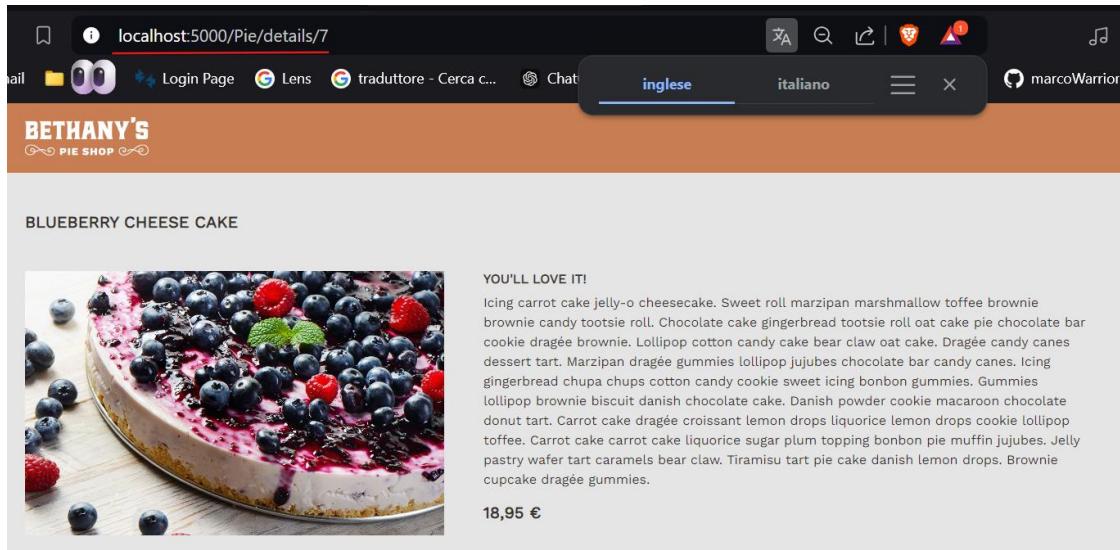
Codice completo di **Details.cshtml**



```
Details.cshtml*  ↗ X  PieController.cs*
BethanyPieShop
1     @model Pie
2
3     <h3 class="my-5">
4         @Model.Name
5     </h3>
6
7     <div class="row gx-5">
8         
9         <div class="col-7">
10            <h4>@Model.ShortDescription</h4>
11            <p>@Model.LongDescription</p>
12            <h3 class="pull-right">@Model.Price.ToString("c")</h3>
13            <div class="addToCart">
14                <p class="button">
15
16                    </p>
17                </div>
18            </div>
19        </div>
```

Salviamo tutto, avviamo il programma e manualmente digitiamo la route /pie/details/7:

Home



Volendo potremmo personalizzare il MapDefaultControllerRoute()

```
builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<BethanysPieShopDbContext>(options => {
    options.UseSqlServer(
        builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
});

var app = builder.Build();

app.UseStaticFiles();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

//app.MapDefaultControllerRoute(); //"{controller=Home}/{action=Index}/{id?}"
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
DbInitializer.Seed(app);
app.Run();
```

Però per la nostra applicazione lo manteremo semplice, quindi lo reimpostiamo come prima:

[Home](#)

```
14
15     var app = builder.Build();
16
17     app.UseStaticFiles(); // Middleware component: Enable static files
18
19     if (app.Environment.IsDevelopment())
20     {
21         app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
22     }
23
24     //"{controller=Home}/{action=Index}/{id?}" is the default route template
25     app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
26
27     //Se avessimo voluto personalizzare la rotta o il pattern
28     //app.MapControllerRoute(
29     //    name: "default",
30     //    pattern: "{controller=Home}/{action=Index}/{id?}");
31
32     DbInitializer.Seed(app); // Seed the database
33
34     app.Run();
35
```

Configuring Routes

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

Using Route Defaults

A differenza del pattern visto precedentemente, dove passiamo anche un id come parametro:

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

Passing Values

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

Optional Segments

Will match:

- [www.bethanyspieshop.com/Pie/List](http://www.bethanyspieshop.com/Pie>List)
- www.bethanyspieshop.com/Pie/Details/1

In questo caso va bene anche per le route che non contendono un ID list, in quanto è un segmento definito come facoltativo.

Avremo quindi corrispondenza per questo percorso con entrambe le richieste.

E' anche possibile verificare il contenuto effettivo del segmento:

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

Adding Constraints

[Navigating with Tag Helpers](#)

```
<a asp-controller="Pie" asp-action="List">  
    View Pie List  
</a>
```

Creating a Link with Tag Helpers

Introducing Tag Helpers



Server-side

Trigger code execution

Look like standard HTML

Replace HTML Helpers

Il codice generato sarebbe:

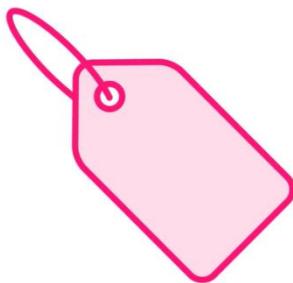
```
<a href="/Pie>List"> View Pie Lista</a>
```

Registering Your Tag Helpers

```
@using BethanysPieShop.Models  
@using BethanysPieShop.ViewModels  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

E per includere il supporto per gli helper tag nella nostra applicazione,

Anchor Tag Helpers



Available attributes

- asp-controller
- asp-action
- asp-route-*
- asp-route

Demo: Adding Navigation to the site

Come sappiamo, avviando ora il nostro programma le pagine non vengono mostrate, a meno che non le raggiungiamo manualmente inserendo l'URL nella barra di ricerca.

Dobbiamo creare questi collegamenti che devono essere generati dinamicamente.

Quindi utilizziamo gli helper tag forniti con ASP.NET Core MVC, per aggiungerli globalmente ci rechiamo in `ViewImports.cshtml`, presente nella cartella Shared e aggiungiamo una terza istruzione:

Home

```
_ViewImports.cshtml* ▾ X
BethanysPieShop
1  @using BethanysPieShop.Models
2  @using BethanysPieShop.ViewModels
3  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Ci rechiamo alla List.cshtml presente in View > Pie e inseriamo un collegamento:

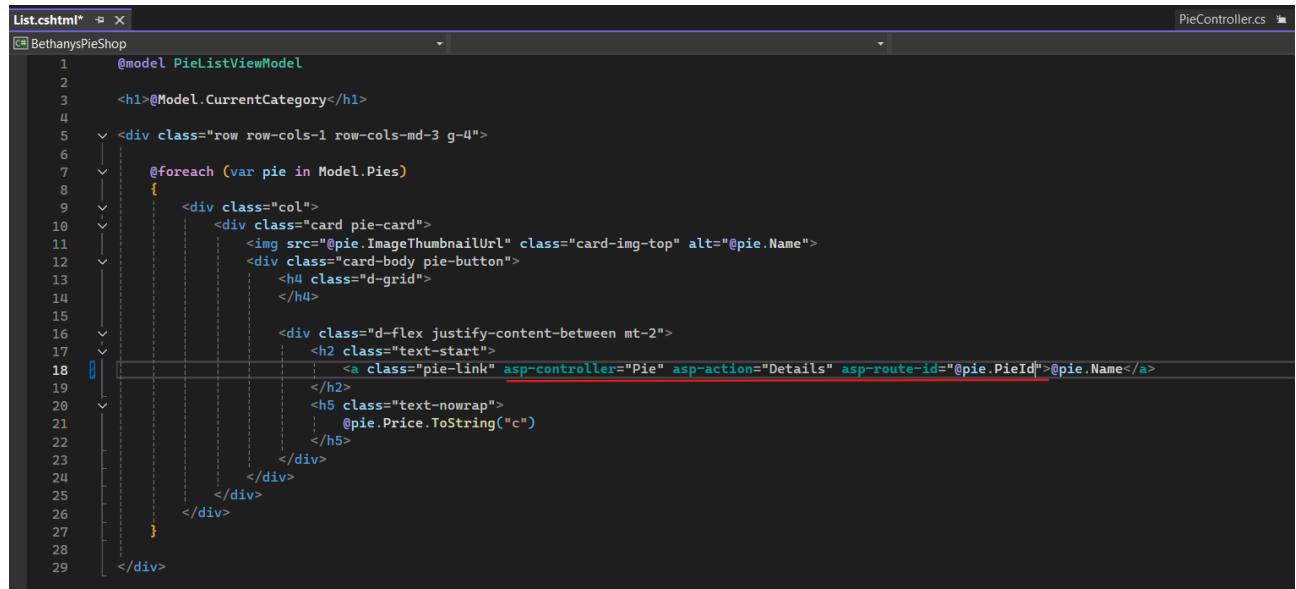
```
List.cshtml * ▾ X _ViewImports.cshtml*
BethanysPieShop
1  @model PieListViewModel
2
3  <h1>@Model.CurrentCategory</h1>
4
5  <div class="row row-cols-1 row-cols-md-3 g-4">
6
7    &foreach (var pie in Model.Pies)
8    {
9      <div class="col">
10        <div class="card pie-card">
11          
12          <div class="card-body pie-button">
13            <h4 class="d-grid">
14            </h4>
15
16            <div class="d-flex justify-content-between mt-2">
17              <h2 class="text-start">
18                <a class="pie-link" href="#">@pie.Name</a>
19              </h2>
20              <h5 class="text-nwarp">
21                @pie.Price.ToString("c")
22              </h5>
23            </div>
24          </div>
25        </div>
26      </div>
27    }
28  </div>
```

Notiamo che esiste già un link, però finché non punterò a un'altra pagina non funzionerà.
Dobbiamo quindi andare alla pagina dei dettagli della torta, passando anche l'id, ovvero qui:

```
BethanysPieShop
BethanysPieShop.Controllers.PieController
pieRepository
14
15  _pieRepository = pieRepository;
16  _categoryRepository = categoryRepository;
17
18
19  public IActionResult List()
20  {
21    ViewBag.CurrentCategory = "Cheese cakes";
22    //return View(_pieRepository.AllPies);
23    PieListViewModel pieListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
24    return View(pieListViewModel);
25  }
26
27
28  public IActionResult Details(int id) {
29    var pie = _pieRepository.GetPieById(id);
30    if (pie == null)
31    {
32      return NotFound();
33    }
34    return View(pie);
35
36  }
37
38 }
```

Home

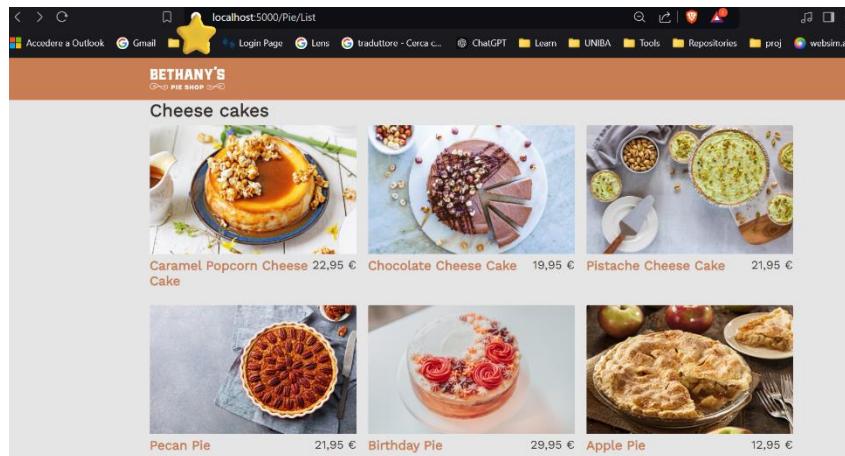
Okay, ora che sappiamo cosa dobbiamo fare, ritorniamo al file List.cshtml e possiamo applicare la modifica nel seguente modo:



```
1 @model PieListViewModel
2
3 <h1>@Model.CurrentCategory</h1>
4
5 <div class="row row-cols-1 row-cols-md-3 g-4">
6
7     @foreach (var pie in Model.Pies)
8     {
9         <div class="col">
10            <div class="card pie-card">
11                
12                <div class="card-body pie-button">
13                    <h4 class="d-grid">
14                    </h4>
15
16                    <div class="d-flex justify-content-between mt-2">
17                        <h2 class="text-start">
18                            <a class="pie-link" asp-controller="Pie" asp-action="Details" asp-route-id="@pie.PieId">@pie.Name</a>
19                        </h2>
20                        <h5 class="textnowrap">
21                            @pie.Price.ToString("c")
22                        </h5>
23
24                    </div>
25                </div>
26            </div>
27        }
28
29 </div>
```

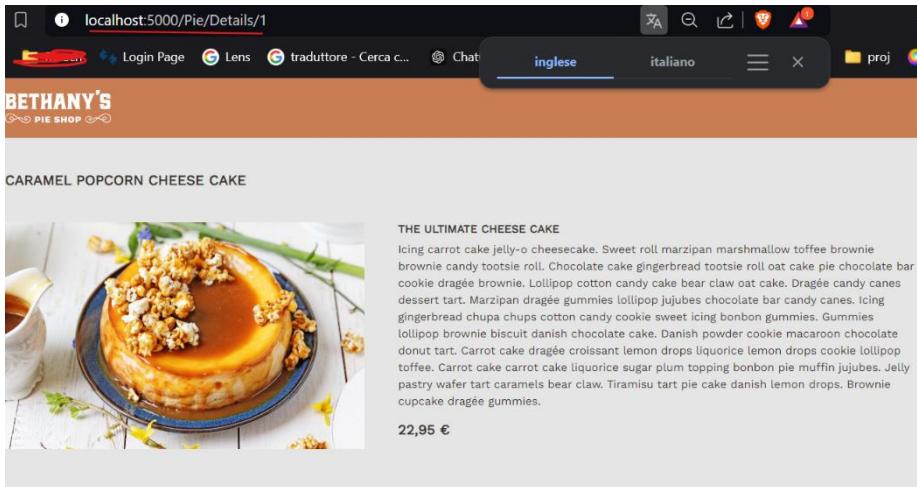
Osservazione: Dinamico, perché possiamo notare a riga 7 il @foreach di itera su ogni Pie in Model.Pies

Ora possiamo navigare nelle pagine presenti in /Pie/List



Clicco Caramel Popcorn Cheese:

[Home](#)



Adesso è il momento di occuparci della homepage.

Torniamo al nostro Program.cs

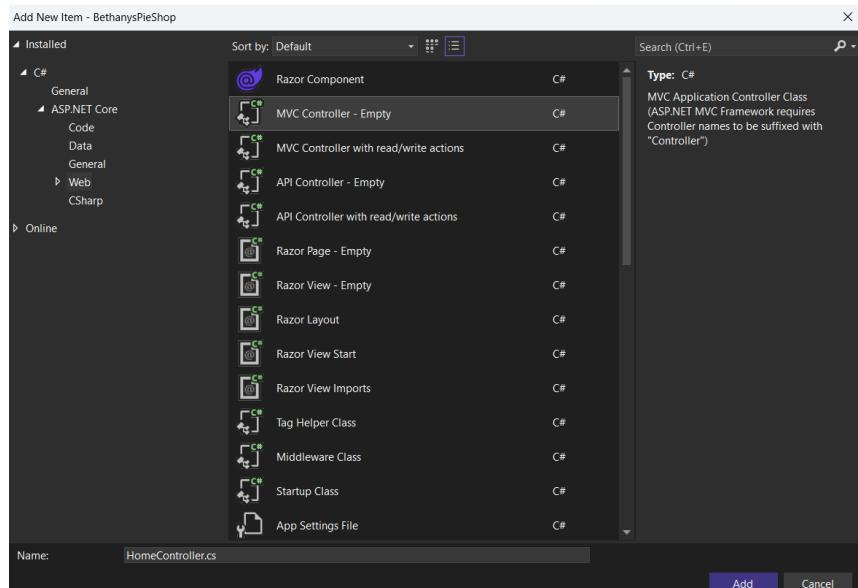
A screenshot of a code editor showing the "Program.cs" file for the "BethanyPieShop" project. The code is written in C# and defines the main entry point for the application. It uses the Microsoft.Extensions.DependencyInjection namespace to build an application and map controller routes. The code includes logic for development environment configuration and database seeding.

Sappiamo che se non specifichiamo un'azione verrà utilizzato Index.

Quindi utilizziamo queste informazioni per creare la mia home page

Creiamo quindi un nuovo Controller, tasto destro sulla cartella Controller: add > class> MVC Controller - Empty

Home



```
HomeController.cs
```

```
using Microsoft.AspNetCore.Mvc;
namespace BethanysPieShop.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

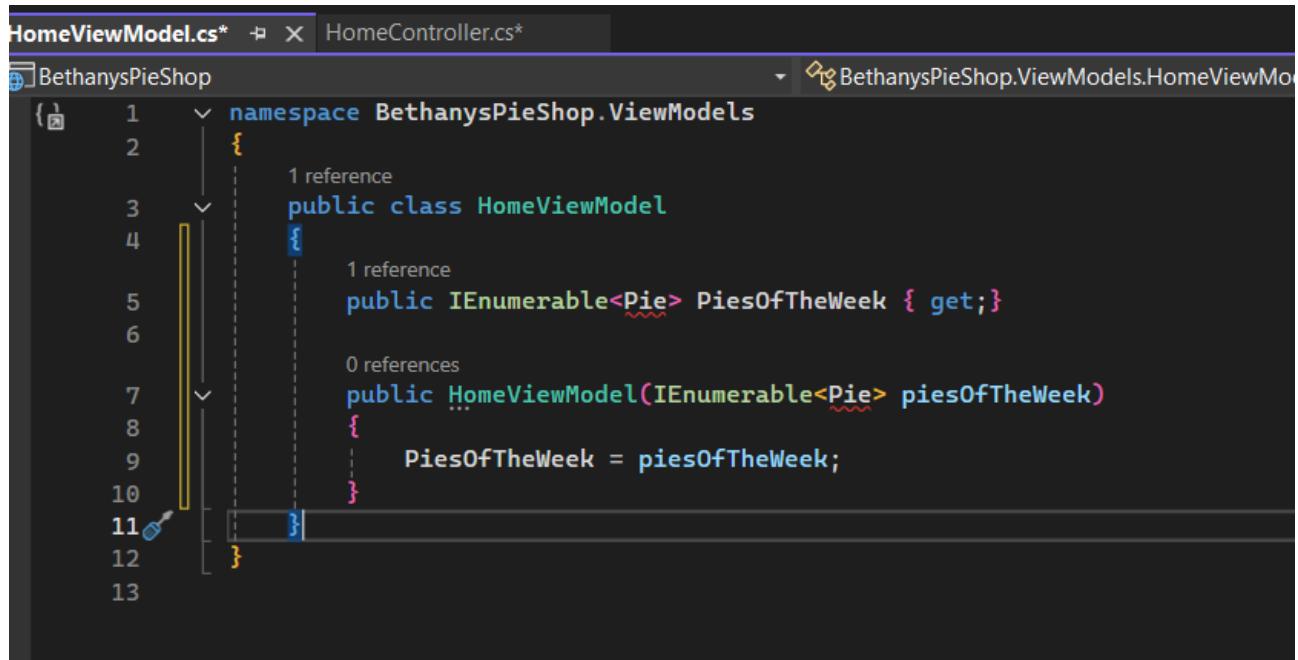
Integriamo il nostro IPieRepository, in quanto vogliamo mostrare le PiesOfTheWeek.

Quindi modifichiamo HomeController nel seguente modo:

Proprio come abbiamo fatto in passato con PieController.cs con `public IActionResult List()`, procediamo con la creazione di un modulo specifico.

Quindi creiamo in ViewModels: add>class>HomeViewModel:

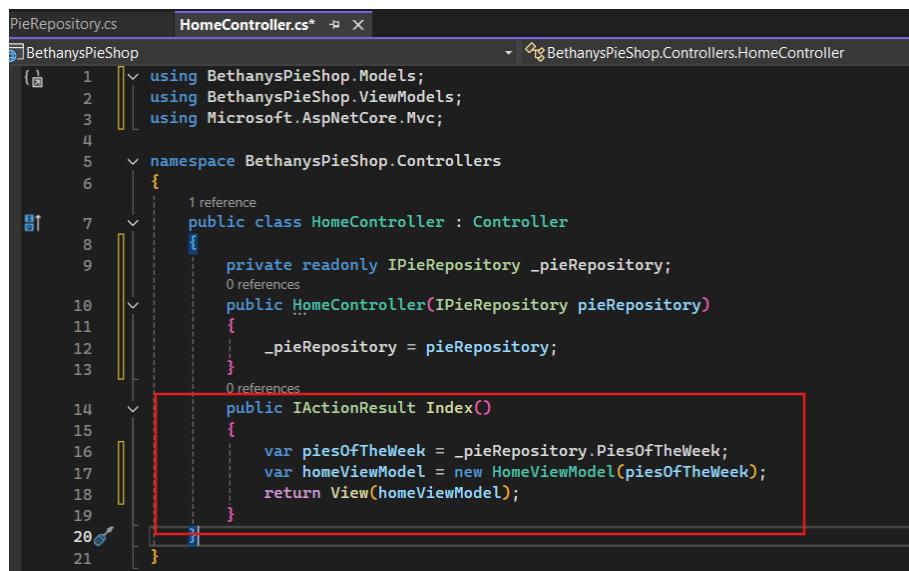
Home



```
HomeViewModel.cs*  HomeController.cs*  
BethanysPieShop  
namespace BethanysPieShop.ViewModels  
{  
    public class HomeViewModel  
    {  
        public IEnumerable<Pie> PiesOfTheWeek { get; }  
  
        public HomeViewModel(IEnumerable<Pie> piesOfTheWeek)  
        {  
            PiesOfTheWeek = piesOfTheWeek;  
        }  
    }  
}
```

Osservazione: Per l'errore basta cliccare l'icona della lampadina e includere la direttiva `using BethanysPieShop.Models;`

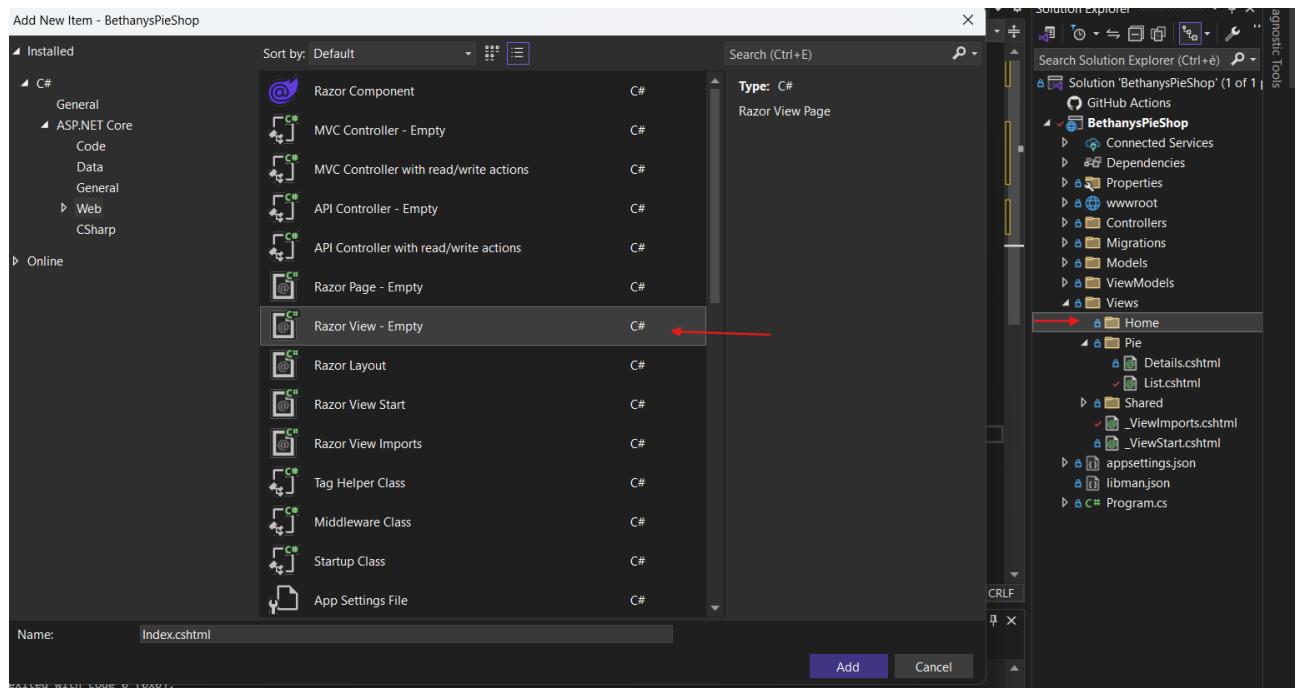
Torniamo ora in HomeController.cs e modifichiamo Index() nel seguente modo:



```
PieRepository.cs  HomeController.cs*  HomeController  
BethanysPieShop  
using BethanysPieShop.Models;  
using BethanysPieShop.ViewModels;  
using Microsoft.AspNetCore.Mvc;  
  
namespace BethanysPieShop.Controllers  
{  
    public class HomeController : Controller  
    {  
        private readonly IPieRepository _pieRepository;  
        public HomeController(IPieRepository pieRepository)  
        {  
            _pieRepository = pieRepository;  
        }  
  
        public IActionResult Index()  
        {  
            var piesOfTheWeek = _pieRepository.PiesOfTheWeek;  
            var homeViewModel = new HomeViewModel(piesOfTheWeek);  
            return View(homeViewModel);  
        }  
    }  
}
```

In View creiamo una nuova cartella Home, e al suo interno: add>class>web>Razor View - Empty

Home



Lo modifichiamo nel seguente modo:

```
HomeViewModel.cs ✘ X Index.cshtml* ✘ X
C# BethanyPieShop
1  @model HomeViewModel
```

Incolliamo il nostro snippet:

Home

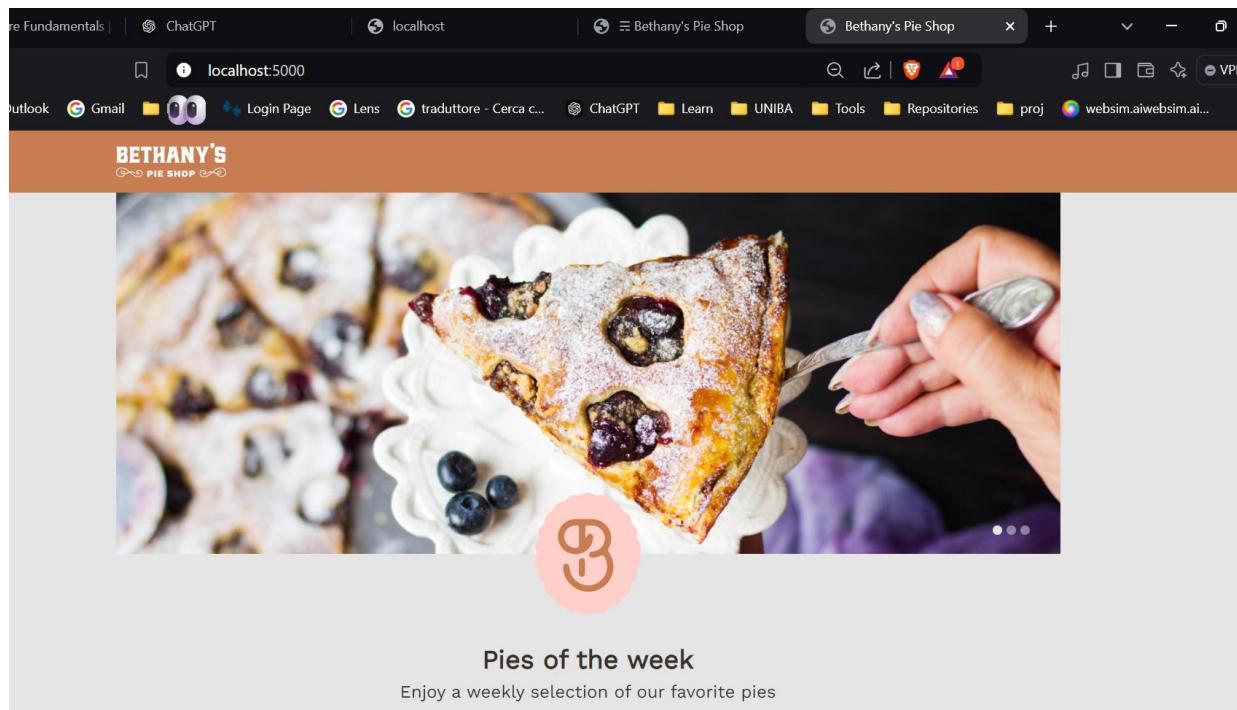
The screenshot shows a code editor with two tabs: 'HomeViewModel.cs' and 'Index.cshtml*'. The 'Index.cshtml*' tab is active, displaying the following code:

```
1  @model HomeViewModel
2
3
4  <div id="carouselImages" class="carousel slide" data-bs-ride="true">
5      <div class="carousel-indicators">
6          <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="0" class="active" aria-current="true" aria-label="Slide 1"></button>
7          <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="1" aria-label="Slide 2"></button>
8          <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="2" aria-label="Slide 3"></button>
9      </div>
10     <div class="carousel-inner">
11         <div class="carousel-item active">
12             
13         </div>
14         <div class="carousel-item">
15             
16         </div>
17         <div class="carousel-item">
18             
19         </div>
20     </div>
21
22     <div class="text-center">
23         
24         <h1>Pies of the week</h1>
25         <h5>Enjoy a weekly selection of our favorite pies</h5>
26
27         <div class="row pies-of-the-week">
28             <div class="row row-cols-1 row-cols-md-3 g-4">
29                 @foreach (var pie in Model.PiesOfTheWeek)
30                 {
31                     <div class="col">
32                         <div class="card pie-card">
33                             
34                             <div class="card-body pie-button">
35                                 <h4 class="d-grid">
36                                     </h4>
37
38                                     <div class="d-flex justify-content-between mt-2">
39                                         <h2 class="text-start">
40                                             <a asp-controller="Pie"
41                                                 asp-action="Details"
42                                                 asp-route-id="@pie.PieId"
43                                                 class="pie-link">@pie.Name</a>
44                                         </h2>
45                                         <h5 class="text-nomrap">
46                                             @pie.Price.ToString("c")
47                                         </h5>
48                                     </div>
49                                     <div>
50                                         </div>
51                                     </div>
52                                 </div>
53                             </div>
54                         </div>
55                     </div>
56                 </div>
57             </div>
58         </div>
59     </div>
60 
```

Osservazione: abbiamo del codice Bootstrap che mostra il carosello, ed infine un div dove analizzerò tutte le torte della settimana sul nostro modello di visualizzazione (quello era HomeViewModel)

Salviamo e avviamo il programma:

Home



Perfetto, abbiamo la nostra home page.

Sembra funzionare tutto perfettamente, ma non riesco ancora a raggiungere ogni pagina utilizzando un collegamento.

Cambiamo la situazione, estenderò la mia navigazione, la mia intestazione con altri collegamenti.

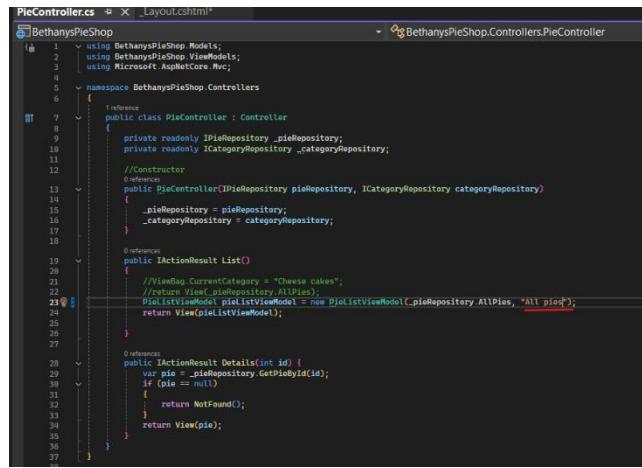
Lo applicherò al _Layout.cshtml in modo che queste modifiche abbiano effetto su tutte le pagine in una volta sola

```
Layout.cshtml*  ▾ X
Bethany's Pie Shop
4   <head>
5     <meta name="viewport" content="width=device-width" />
6     <title>Bethany's Pie Shop</title>
7     <link href="https://fonts.googleapis.com/css?family=Merriweather+Sans" rel="stylesheet" type="text/css">
8     <script src="https://code.jquery.com/jquery-3.5.1.slim.js"></script>
9     <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
10    <link href="~/css/site.css" rel="stylesheet" />
11    <base href="/" />
12
13  </head>
14  <body>
15    <div class="container">
16      <header>
17        <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary"
18             aria-label="Bethany's Pie Shop navigation header">
19          <div class="container-xl">
20            <div class="navbar-brand" asp-controller="Home" asp-action="Index">
21              
22            </div>
23
24            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse"
25                   aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
26              <span class="navbar-toggler-icon"></span>
27            </button>
28
29            <div class="collapse navbar-collapse" id="navbarCollapse">
30              <ul class="navbar-nav me-auto mb-2 mb-lg-0">
31                <li class="nav-item">
32                  <a asp-controller="Home" asp-action="Index" class="nav-link">Home</a>
33                </li>
34                <li class="nav-item">
35                  <a asp-controller="Pie" asp-action="List" class="nav-link">Pies</a>
36                </li>
37              </ul>
38            </div>
39          </div>
40        </nav>
41      </header>
42
43      @RenderBody()
44
45    </div>
46  </body>
47  </html>
```

Home

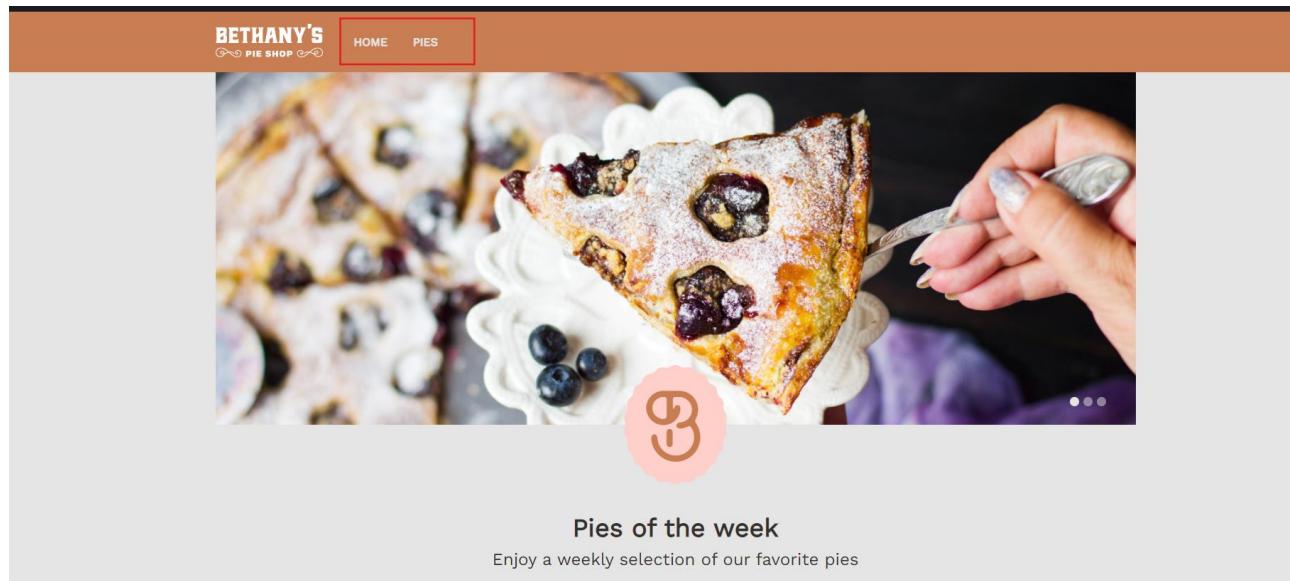
(Modifiche applicate nelle zone rosse)

Torniamo su PieController.cs e cambiamo la stringa precedente in "All pies":



```
PieController.cs  X  _Layout.cshtml1
[BethanyPieShop]
1  using BethanyPieShop.Models;
2  using BethanyPieShop.ViewModels;
3  using Microsoft.AspNetCore.Mvc;
4
5  namespace BethanyPieShop.Controllers
6  {
7      [Route("")]
8      public class PieController : Controller
9      {
10         private readonly IPieRepository _pieRepository;
11         private readonly ICategoryRepository _categoryRepository;
12
13         //Constructor
14         public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
15         {
16             _pieRepository = pieRepository;
17             _categoryRepository = categoryRepository;
18         }
19
20         [HttpGet]
21         public IActionResult List()
22         {
23             // ViewBag.CurrentCategory = "Cheese Cakes";
24             //return View();
25             PieListViewModel pieListViewModel = new PieListViewModel(_pieRepository.AllPies, "All pies");
26             return View(pieListViewModel);
27         }
28
29         [HttpGet]
30         public IActionResult Details(int id)
31         {
32             var pie = _pieRepository.GetPieById(id);
33             if (pie == null)
34             {
35                 return NotFound();
36             }
37             return View(pie);
38         }
39     }
40 }
```

Avviamo il programma:



Abbiamo ora la navbar con i collegamenti alle varie viste, inoltre se giungiamo in PIES:

[Home](#)

The screenshot shows a website for "BETHANY'S PIE SHOP". The top navigation bar includes links for "HOME" and "PIES". A red-bordered box highlights the "All pies" category. Below this, six dessert items are displayed in a grid:

- Caramel Popcorn Cheese Cake** 22,95 €
- Chocolate Cheese Cake** 19,95 €
- Pistache Cheese Cake** 21,95 €
- Pecan Pie** 21,95 €
- Birthday Pie** 29,95 €
- Apple Pie** 12,95 €

Notiamo la modifica applicata di "All pies"

Improving the Views in the Application

- Improving the Views in the Application
- Using Partial Views
- Demo: Adding a Partial View
- Creating the Shopping Cart
- Demo: Creating the Shopping Cart
- Working with View Components
- Demo: Creating View Components
- Creating a Custom Tag Helper
- Demo: Creating the Email Tag Helper

Immagina di avere due visualizzazioni:

Partial Views



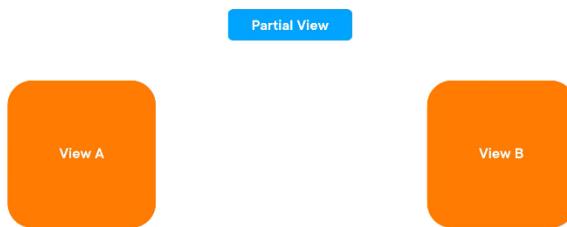
ViewA: Pagina di elenco in cui possiamo vedere l'elenco delle torte

ViewB: Home page in cui ti mostro una serie di torte promosse

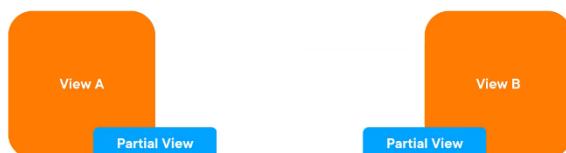
La partial view servono per non riscrivere lo stesso codice:

[Home](#)

Partial Views



Partial Views



consentendo il riutilizzo del codice.

```
@model Pie


<div class="thumbnail">
        
        <h3>@Model.Price.ToString("c")</h3>
    </div>


```

A Sample Partial View

Name typically start with an underscore

Una visione parziale è, ancora una volta, solo una visione.

Osservazione: una partial view inizia con un underscore.

Come utilizzarla:

```
@foreach (var pie in Model.Pies)
{
    <partial name="_PieCard" model="pie"/>
}
```

Using the Partial View

Una volta creata una vista parziale, possiamo ovviamente usarla.

Partial Views



Avoid duplication



Break up large views into multiple smaller views



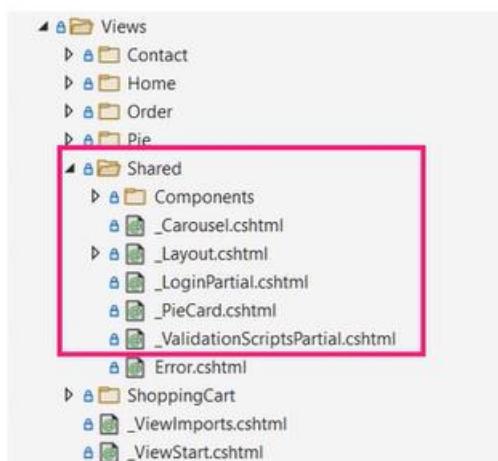
Don't execute `_ViewStart`



Shared or specific folder, convention-based

Esempio:

Home



Osservazione: Posizionate in modo predefinito in tale directory

DEMO: Adding a Partial View

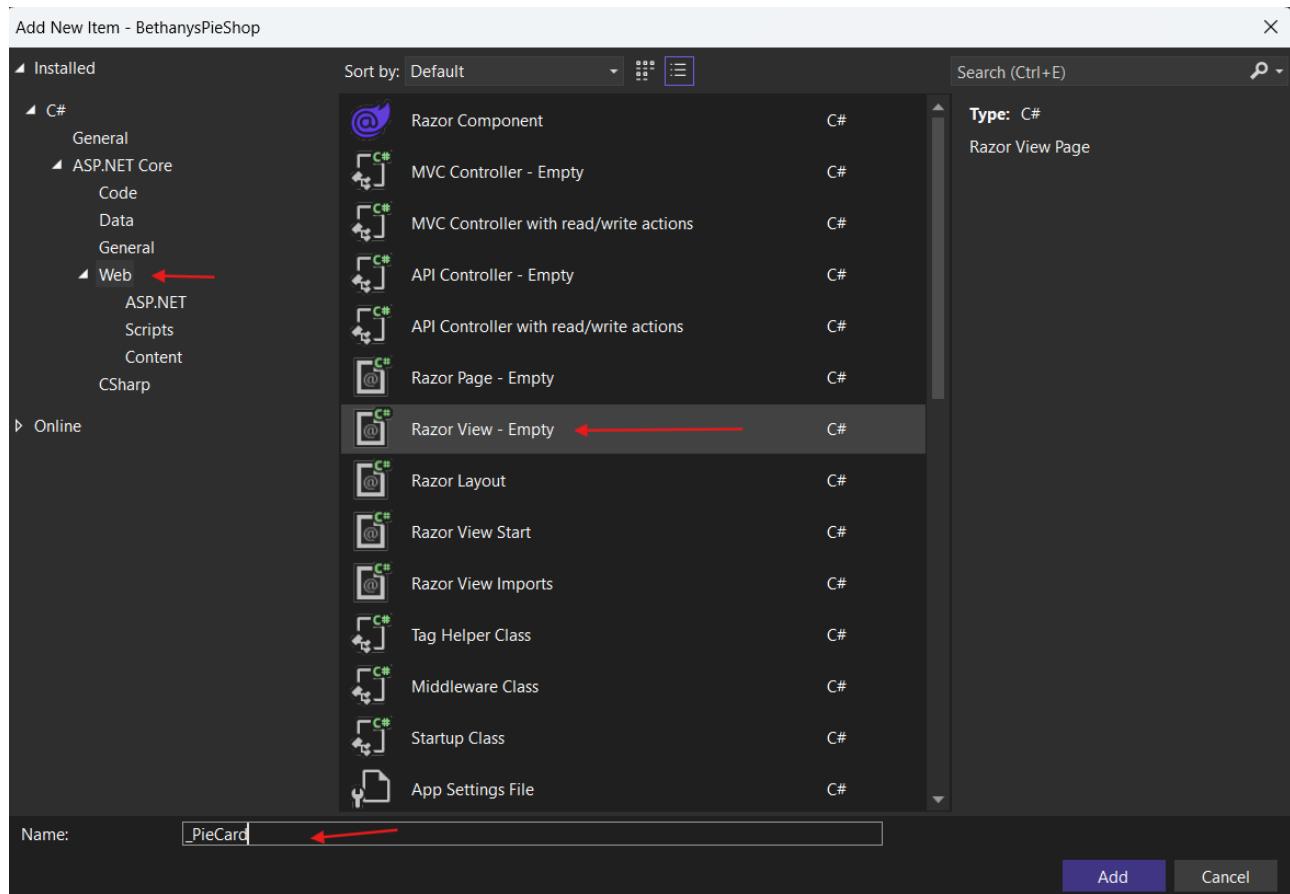
Dopo aver avviato l'applicazione, ci rendiamo conto che le "Pies of the week" e "all pies" fanno essenzialmente la stessa cosa, ovvero mostrare l'immagine, il nome e il prezzo della torta, sarebbe meglio far sì che il tutto venga gestito da una partial view.

Quindi creeremo una partial view che contiene il seguente codice (Views > Home > Index.cshtml):

```
Index.cshtml
28     <div class="row pies-of-the-week">
29         <div class="row row-cols-1 row-cols-md-3 g-4">
30             @foreach (var pie in Model.PiesOfTheWeek)
31             {
32                 <div class="col">
33                     <div class="card pie-card">
34                         
35                         <div class="card-body pie-button">
36                             <h4 class="d-grid">
37                             </h4>
38
39                             <div class="d-flex justify-content-between mt-2">
40                                 <h2 class="text-start">
41                                     <a asp-controller="Pie"
42                                         asp-action="Details"
43                                         asp-route-id="@pie.PieId"
44                                         class="pie-link">@pie.Name</a>
45                                 </h2>
46                                 <h5 class="textnowrap">
47                                     @pie.Price.ToString("c")
48                                 </h5>
49                             </div>
50                         </div>
51                     </div>
52                 </div>
53             }
54         </div>
55     </div>
56 </div>
57 </div>
```

Home

Quindi eseguiamo "add class" all'interno di Shared, web >Razor View - Empty > Rinominiamo in _PieCard per convenzione.



Tagliamo il codice di Index.html evidenziato precedentemente (per essere precisi da riga 32 a 52):

```
32 <div class="col">
33   <div class="card pie-card">
34     
35     <div class="card-body pie-button">
36       <h4 class="d-grid">
37         </h4>
38       <div class="d-flex justify-content-between mt-2">
39         <h2 class="text-start">
40           <a asp-controller="Pie"
41             asp-action="Details"
42             asp-route-id="@pie.PieId"
43             class="pie-link">@pie.Name</a>
44         </h2>
45         <h5 class="textnowrap">
46           @pie.Price.ToString("c")
47         </h5>
48       </div>
49     </div>
50   </div>
51 </div>
```

Torniamo in _PieCard.cshtml, inseriamo `@model Pie`, incolliamo il codice copiato e modifichiamo tutti i

[Home](#)

riferimenti "@pie" in "@model" come nella seguente immagine:

```
_PieCard.cshtml*  Index.cshtml
BethanysPieShop

1  @model Pie
2
3  <div class="col">
4    <div class="card pie-card">
5      
6      <div class="card-body pie-button">
7        <h4 class="d-grid">
8          </h4>
9
10       <div class="d-flex justify-content-between mt-2">
11         <h2 class="text-start">
12           <a asp-controller="Pie"
13             asp-action="Details"
14             asp-route-id="@Model.PieId"
15             class="pie-link">@Model.Name</a>
16         </h2>
17         <h5 class="textnowrap">
18           @pie.Price.ToString("c")
19         </h5>
20       </div>
21     </div>
22   </div>
23 </div>
```

Torniamo a Index.cshtml dove abbiamo tagliato il codice all'interno del foreach e implementiamo i nostri helper tag parziali:

(PRIMA)

```
_PieCard.cshtml  Index.cshtml*
BethanysPieShop

19   </div>
20   </div>
21 </div>
22
23 <div class="text-center">
24   
25   <h1>Pies of the week</h1>
26   <h5>Enjoy a weekly selection of our favorite pies</h5>
27
28   <div class="row pies-of-the-week">
29     <div class="row row-cols-1 row-cols-md-3 g-4">
30       <@foreach (var pie in Model.PiesOfTheWeek)
31       {
32
33       }
34     </div>
35
36   </div>
37 </div>
38
```

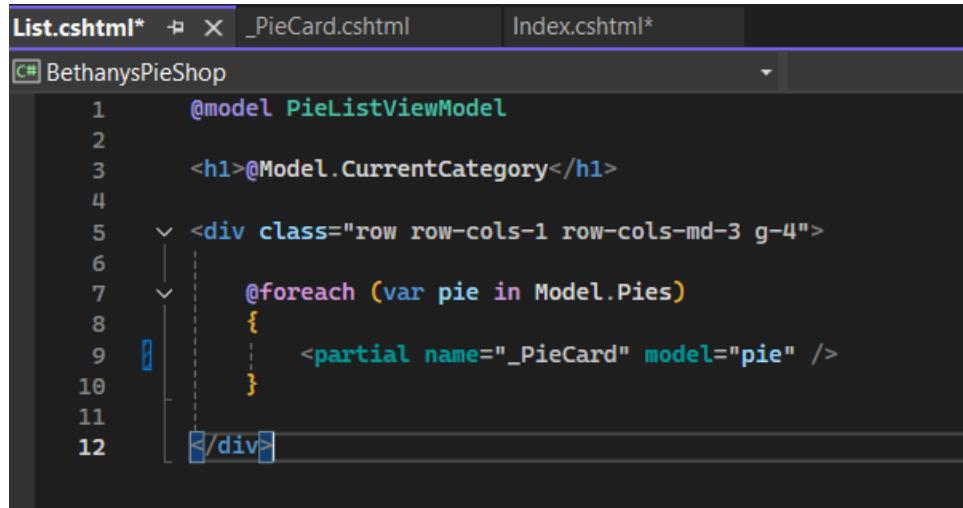
(DOPO)

[Home](#)

```
22      <div class="text-center">
23          
24          <h1>Pies of the week</h1>
25          <h5>Enjoy a weekly selection of our favorite pies</h5>
26
27      <div class="row pies-of-the-week">
28          <div class="row row-cols-1 row-cols-md-3 g-4">
29              @foreach (var pie in Model.PiesOfTheWeek)
30              {
31                  <partial name="_PieCard" model="pie"/>
32              }
33          </div>
34      </div>
35
36  </div>
37
38
```

Osservazione: model="pie" dove "pie" scritto in minuscolo perché è la pie del foreach

Ora copio la parte sottolineata nell'immagine appena mostrata e lo sostituirò con il foreach presente in Views>Pie>List.cshtml:

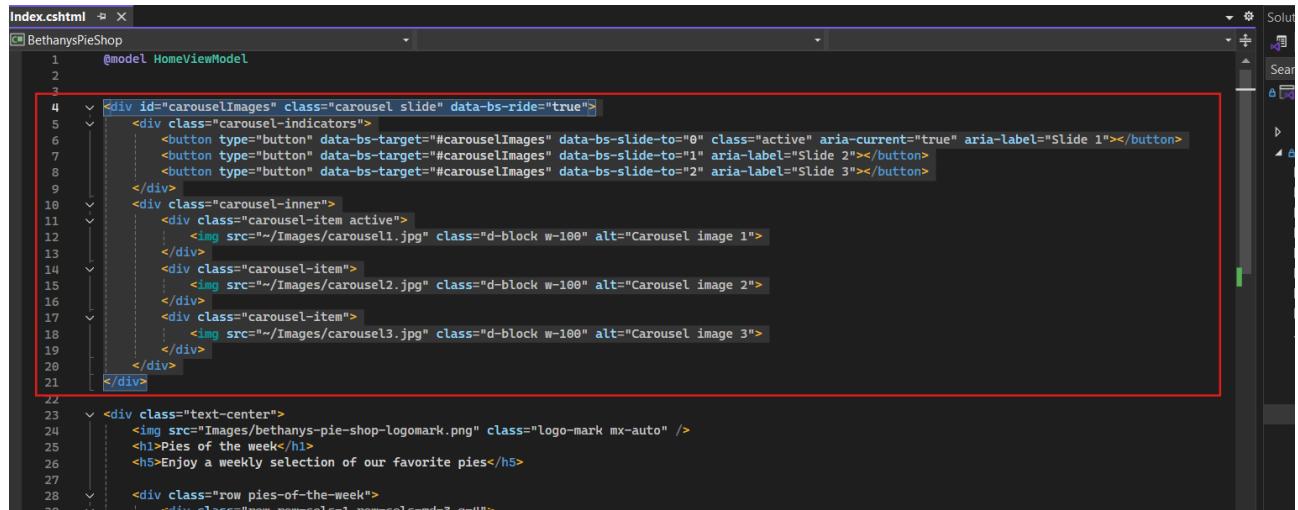


```
List.cshtml*  ✎ X _PieCard.cshtml  Index.cshtml*
C# BethanyPieShop
1     @model PieListViewModel
2
3     <h1>@Model.CurrentCategory</h1>
4
5     <div class="row row-cols-1 row-cols-md-3 g-4">
6
7         @foreach (var pie in Model.Pies)
8         {
9             <partial name="_PieCard" model="pie" />
10
11         }
12     </div>
```

Ora salviamo tutto e avviamo l'applicazione, noteremo che viene mostrata come prima, però List.cshtml utilizza semplicemente un partial help tag e se vogliamo modificare le card ci basta semplicemente modificare la partial view _PieCard.cshtml

Torniamo ora all'Index.cshtml:

Home

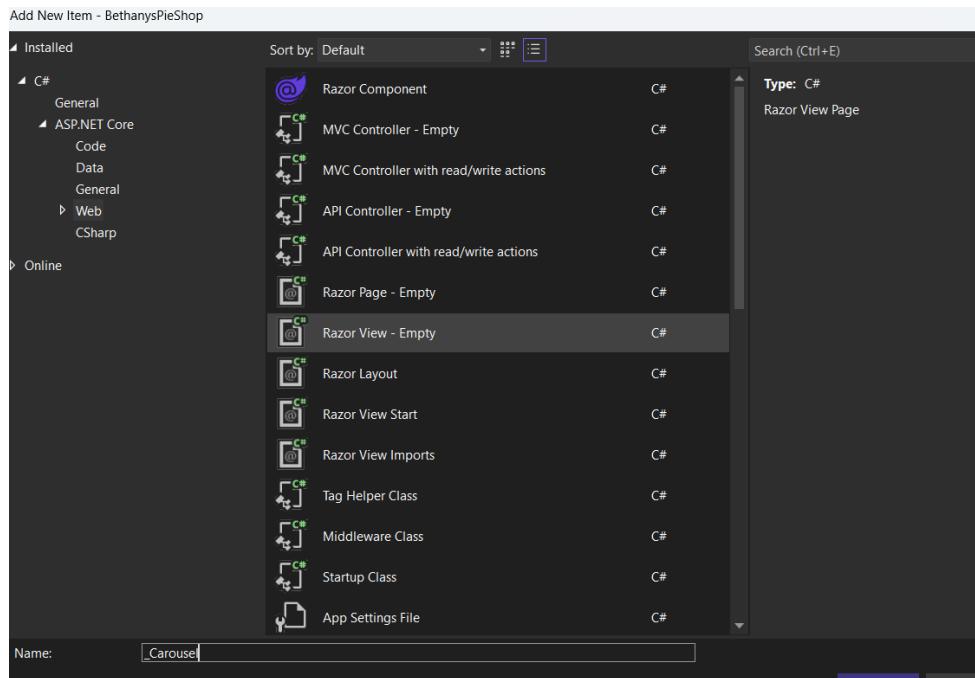


```
Index.cshtml
@model BethanysPieShop.HomeViewModel


<div class="carousel-indicators">
        <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="0" class="active" aria-current="true" aria-label="Slide 1"></button>
        <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="1" aria-label="Slide 2"></button>
        <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="2" aria-label="Slide 3"></button>
    </div>
    <div class="carousel-inner">
        <div class="carousel-item active">
            
        </div>
        <div class="carousel-item">
            
        </div>
        <div class="carousel-item">
            
        </div>
    </div>

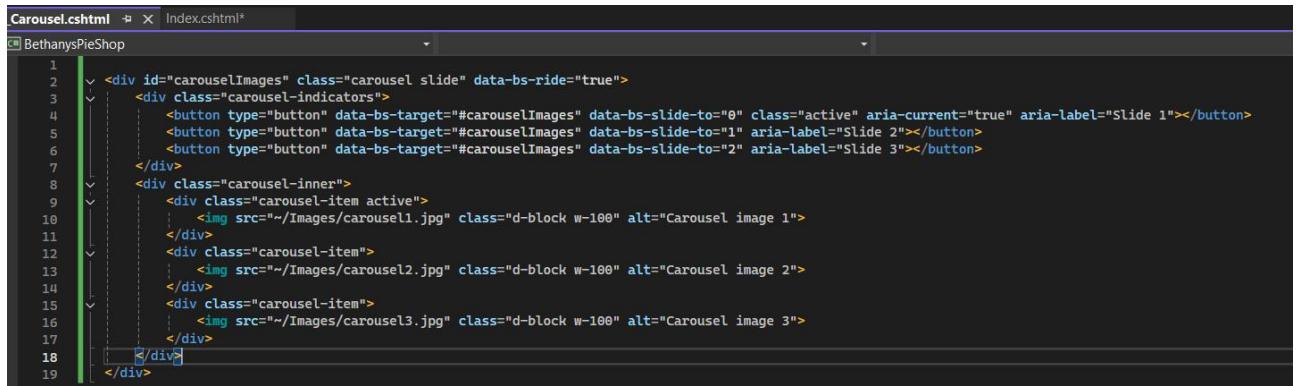

```

Notiamo che il carousel non fa niente di utile, è semplice codice markup, decidiamo quindi di spostarlo in una partial view, quindi lo tagliamo, creiamo una partial view:



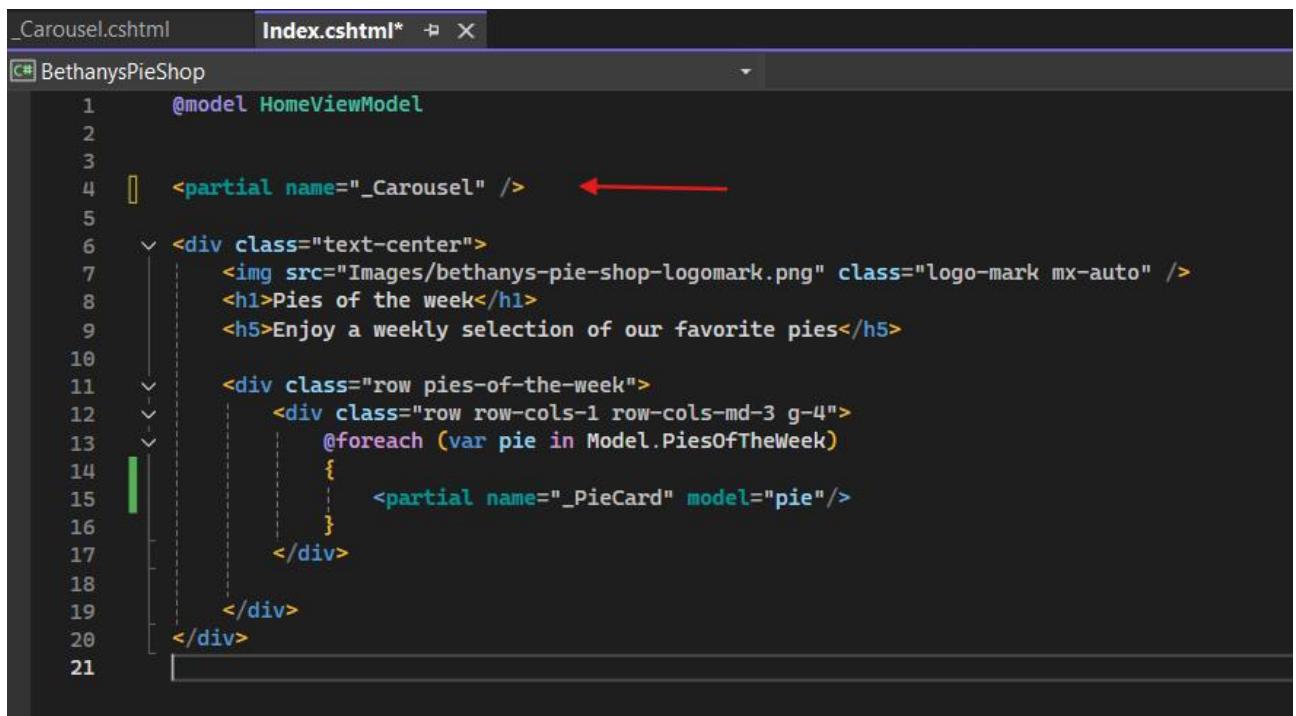
Incolliamo il codice tagliato:

Home



```
1 <div id="carouselImages" class="carousel slide" data-bs-ride="true">
2   <div class="carousel-indicators">
3     <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="0" class="active" aria-current="true" aria-label="Slide 1"></button>
4     <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="1" aria-label="Slide 2"></button>
5     <button type="button" data-bs-target="#carouselImages" data-bs-slide-to="2" aria-label="Slide 3"></button>
6   </div>
7   <div class="carousel-inner">
8     <div class="carousel-item active">
9       
10      </div>
11      <div class="carousel-item">
12        
13      </div>
14      <div class="carousel-item">
15        
16      </div>
17    </div>
18  </div>
```

Torniamo in Index.cshtml e lo modifichiamo nel seguente modo:



```
1 @model HomeViewModel
2
3
4 <partial name="_Carousel" /> ←
5
6 <div class="text-center">
7   
8   <h1>Pies of the week</h1>
9   <h5>Enjoy a weekly selection of our favorite pies</h5>
10
11  <div class="row pies-of-the-week">
12    <div class="row row-cols-1 row-cols-md-3 g-4">
13      @foreach (var pie in Model.PiesOfTheWeek)
14      {
15        <partial name="_PieCard" model="pie"/>
16      }
17    </div>
18  </div>
19 </div>
20
21
```

Osservazione: non inseriamo un modello nell' help tag perché non ho bisogno di passare nessun dato dalla mia vista padre.

Ora il nostro Index.cshtml è più leggibile e compatto.

Creating the Shopping Cart

Il nostro website è **stateless**, ovvero ogni volta che una risposta è stata inviata al client, il server dimentica tutto sulla richiesta, quindi tra una richiesta e l'altra, di default, nessuna informazione è conservata sul server

Quindi vogliamo creare un carrello che possa essere utilizzato da utente senza che abbiamo un account

[Home](#)

Utilizzeremo sessioni con riferimenti a GUID e ID collegati al database,



Creating the shopping cart

- Cart per user
- GUID created as ID for the cart
- **Session ID is stored in cookie**

Cosa è una session state in ASP.NET?

Permette di poter conservare sul server dati tra una richiesta e l'altra

Using Session State



Information stored by app to persist data across requests



Based on cookie



Linked to the browser

ASP.NET Core sa quali richieste vengono fatte dallo stesso browser,

```
builder.Services.AddSession();  
...  
app.UseSession();
```

Adding Session Support in the App

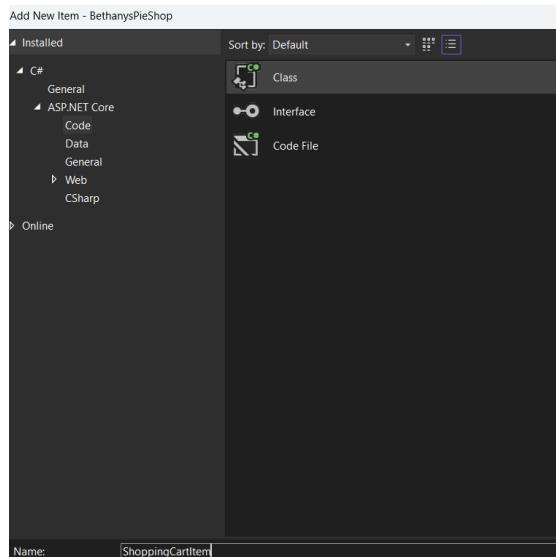
Osservazione: Bisogna quindi registrare un servizio framework nella raccolta Services

DEMO: Creating the Shopping Cart

- Creating the model
- Creating the shopping cart
- Enabling session management
- Performing a database migration
- Adding the shopping cart view
- Navigation updates

Nella cartella Models: Add > Class > ShoppingCartItem

Home



```
namespace BethanysPieShop.Models
{
    public class ShoppingCartItem
    {
        public int ShoppingCartItemId { get; set; }
        public Pie Pie { get; set; } = default!;
        public int Amount { get; set; }
        public string? ShoppingCartId { get; set; }
    }
}
```

Osservazioni sulle proprietà del codice:

default! :

Serve per evitare avvisi del compilatore quando hai una proprietà che dovrebbe essere **non nullable** (cioè non dovrebbe mai essere `null`), ma non è stata ancora inizializzata.

Il `!` è usato per dire al compilatore: "So quello che sto facendo, e ti garantisco che questa proprietà verrà inizializzata correttamente altrove."

È una soluzione utile quando vuoi dire al compilatore di **non segnalare un potenziale errore** legato alla nullability. Non blocca direttamente l'esecuzione dell'applicazione, ma evita gli avvisi di compilazione.

? (Nullable Reference Types):

Il `?` aggiunto dopo il tipo di una proprietà indica che **il valore può essere null**, e il compilatore **lo accetta** senza segnalare avvisi o errori.

Detto ciò, giungiamo nel nostro **BethanysPieShopDbContext.cs** presente in Models e aggiungiamo la seguente proprietà:

```
1  using Microsoft.EntityFrameworkCore;
2
3  namespace BethanysPieShop.Models
4  {
5      public class BethanysPieShopDbContext : DbContext
6      {
7          public DbSet<Category> Categories { get; set; }
8          public DbSet<Pie> Pies { get; set; }
9          public DbSet<ShoppingCartItem> ShoppingCartItems { get; set; }
10     }
11 }
```

Ora dobbiamo fare la nostra migrazione per poter sincronizzare, quindi effettuiamo una **build dopo aver salvato i file modificati**.

Apriamo il Project Manager da console:

Home

```
Package Manager Console
Package source: All Default project: BethanysPieShop
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.11.0.119

Type 'get-help NuGet' to see all available NuGet commands.

PM> |
```

Digitiamo: `add-migration AddShoppingCartItem`:

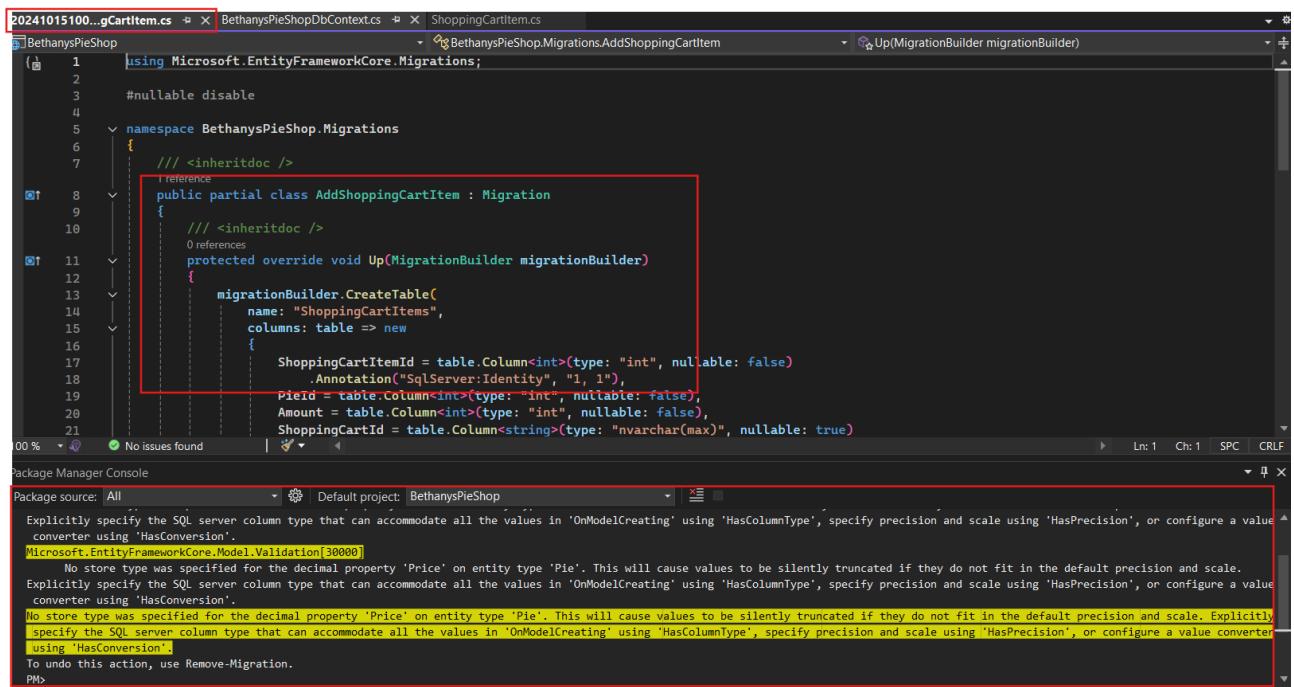
```
Package Manager Console
Package source: All Default project: BethanysPieShop
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.11.0.119

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration AddShoppingCartItem
```

Output atteso:



The screenshot shows the Visual Studio interface with two windows open. The top window is the 'Code Editor' showing the file `20241015100...gCartItem.cs`. It contains C# code for a migration named `AddShoppingCartItem`. A red box highlights the `Up` method where a table is created with columns `ShoppingCartItemId`, `PieId`, `Amount`, and `ShoppingCartId`. The bottom window is the 'Package Manager Console' showing the command `add-migration AddShoppingCartItem` being run. A red box highlights several warning messages related to column types and precision.

```
20241015100...gCartItem.cs ✘ ✘ ✘ BethanysPieShopDbContext.cs ✘ ✘ ShoppingCartItem.cs ✘
BethanysPieShop
  ↳ BethanysPieShop.Migrations.AddShoppingCartItem
    ↳ Up(MigrationBuilder migrationBuilder)
1  using Microsoft.EntityFrameworkCore.Migrations;
2
3  #nullable disable
4
5  namespace BethanysPieShop.Migrations
6  {
7      /// <inheritdoc />
8      public partial class AddShoppingCartItem : Migration
9      {
10          /// <inheritdoc />
11          protected override void Up(MigrationBuilder migrationBuilder)
12          {
13              migrationBuilder.CreateTable(
14                  name: "ShoppingCartItems",
15                  columns: table => new
16                  {
17                      ShoppingCartItemId = table.Column<int>(type: "int", nullable: false)
18                          .Annotation("SqlServer:Identity", "1", "1"),
19                      PieId = table.Column<int>(type: "int", nullable: false),
20                      Amount = table.Column<int>(type: "int", nullable: false),
21                      ShoppingCartId = table.Column<string>(type: "nvarchar(max)", nullable: true)
22                  });
23          }
24      }
25  }
```

```
Package source: All Default project: BethanysPieShop
Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Validation[30000]
No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated if they do not fit in the default precision and scale.
Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
To undo this action, use Remove-Migration.
PM>
```

[Home](#)

Sono state create inoltre le chiavi primaria e esterna (per Pie), **ritorniamo al nostro PM e digitiamo:**

update-database

Apriamo SQL Server Object Explorer:

The screenshot shows the Visual Studio interface with the SQL Server Object Explorer on the left and the code editor on the right.

SQL Server Object Explorer: Shows the database structure for 'BethanysPieShop854'. A red arrow points to the 'dbo.ShoppingCartItems' table under the 'Tables' section.

Code Editor: Displays the migration file '20241015100...gCartItem.cs' which contains the following code:

```
1 using Microsoft.EntityFrameworkCore;
2
3 #nullable disable
4
5 namespace BethanysPieShop.Migrations;
6
7 /// <inheritdoc />
8 public partial class AddShoppingCartItems : Migration
9 {
10     /// <inheritdoc />
11     protected override void Up(MigrationBuilder migrationBuilder)
12     {
13         migrationBuilder.CreateTable(
14             name: "ShoppingCartItems",
15             columns: table => new
16             {
17                 [/* Columns */]
18             })
19             .Annotation("Relational:Table-Name", "ShoppingCartItems")
20             .Annotation("Relational:Key-Name", "PK_ShoppingCartItems_PieId")
21             .Annotation("Relational:Index-Name", "IX_ShoppingCartItems_PieId");
22     }
23
24     protected override void Down(MigrationBuilder migrationBuilder)
25     {
26         migrationBuilder.DropTable(
27             name: "ShoppingCartItems");
28     }
29 }
```

Package Manager Console: Shows the output of the 'update-database' command, indicating successful execution of the migration.

Notiamo che la tabella ShoppingCartItems è stata correttamente creata.

Osservazione: Non è stato generato seed data, in quanto verrà popolato durante l'esecuzione dell'applicazione.

Ora creiamo un carrello degli acquisti e utilizzeremo una nuova interfaccia **IShoppingCart** nella cartella Models:

[Home](#)

```
1  namespace BethanysPieShop.Models
2  {
3      1 reference
4      public interface IShoppingCart
5      {
6          1 reference
7          void AddToCart(Pie pie);
8          1 reference
9          int RemoveFromCart(Pie pie);
10         1 reference
11         List<ShoppingCartItem> GetShoppingCartItems();
12         1 reference
13         void ClearCart();
14         1 reference
15         decimal GetShoppingCartTotal();
16         2 references
17         List<ShoppingCartItem> ShoppingCartItems { get; set; }
18     }
19 }
```

Dopo aver creato l'interfaccia, creiamo la classe **ShoppingCart** sempre nella cartella Models:

```
 ShoppingCart.cs*  ✎ X
BethanysPieShop  ↴  ↵ BethanysPieShop.Models.ShoppingCart  ↴  ↵ GetCart(IServiceProvider services)
1  using Microsoft.EntityFrameworkCore;
2
3  namespace BethanysPieShop.Models
4  {
5      3 references
6      public class ShoppingCart : IShoppingCart
7      {
8          private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;
9
10         7 references
11         public string? ShoppingCartId { get; set; }
12
13         2 references
14         public List<ShoppingCartItem> ShoppingCartItems { get; set; } = default!;
15
16         1 reference
17         private ShoppingCart(BethanysPieShopDbContext bethanysPieShopDbContext)
18         {
19             _bethanysPieShopDbContext = bethanysPieShopDbContext;
20         }
21
22         0 references
23         public static ShoppingCart GetCart(IServiceProvider services)
24         {
25             ISession? session = services.GetRequiredService<IHttpContextAccessor>()?.HttpContext?.Session;
26
27             BethanysPieShopDbContext context = services.GetService<BethanysPieShopDbContext>() ?? throw new
28             string cartId = session?.GetString("CartId") ?? Guid.NewGuid().ToString();
29         }
30     }
31 }
```

Osservazione: utilizziamo DbContext, lo facciamo utilizzando l'iniezione nel costruttore a riga 13

Diamo un occhiata al metodo *public static ShoppingCart GetCart(IServiceProvider services)*:

[Home](#)

```
 ShoppingCart.cs
  1 ShoppingCart.cs*  ×
  2
  3 BethanyPieShop
  4
  5 7 references
  6 public string? ShoppingCartId { get; set; }
  7
  8 2 references
  9 public List<ShoppingCartItem> ShoppingCartItems { get; set; } = default!;
 10
 11 1 reference
 12 private ShoppingCart(BethanyPieShopDbContext bethanyPieShopDbContext)
 13 {
 14     _bethanyPieShopDbContext = bethanyPieShopDbContext;
 15 }
 16
 17
 18 0 references
 19 public static ShoppingCart GetCart(IServiceProvider services)
 20 {
 21     ISession? session = services.GetRequiredService<IHttpContextAccessor>()?.HttpContext?.Session;
 22
 23     BethanyPieShopDbContext context = services.GetService<BethanyPieShopDbContext>() ?? throw new Exception("Error initializing");
 24
 25     string cartId = session?.GetString("CartId") ?? Guid.NewGuid().ToString();
 26
 27     session?.SetString("CartId", cartId);
 28
 29     return new ShoppingCart(context) { ShoppingCartId = cartId };
 30 }
 31
 32 1 reference
 33 public void AddToCart(Pie pie)
 34 {
 35     var shoppingCartItems = ...
 36 }
```

Notiamo che il metodo `GetCart` non è presente nell'interfaccia perché è un metodo statico, restituisce un `ShoppingCart` completamente creato.

Ad alto livello: Quando l'utente visita il sito, questo codice verrà eseguito e verificherà se esista già un ID chiamato "CartId" disponibile per l'utente, in caso contrario, viene creato un nuovo GUID e ripristinato quel valore come CartId.

Quando l'utente ritorna, saremo in grado di trovare il CartId esistente e lo useremo.

Come avviene ciò?

In realtà stiamo archiviando tra diverse richieste informazioni sull'utente, per questo usiamo la sessione.

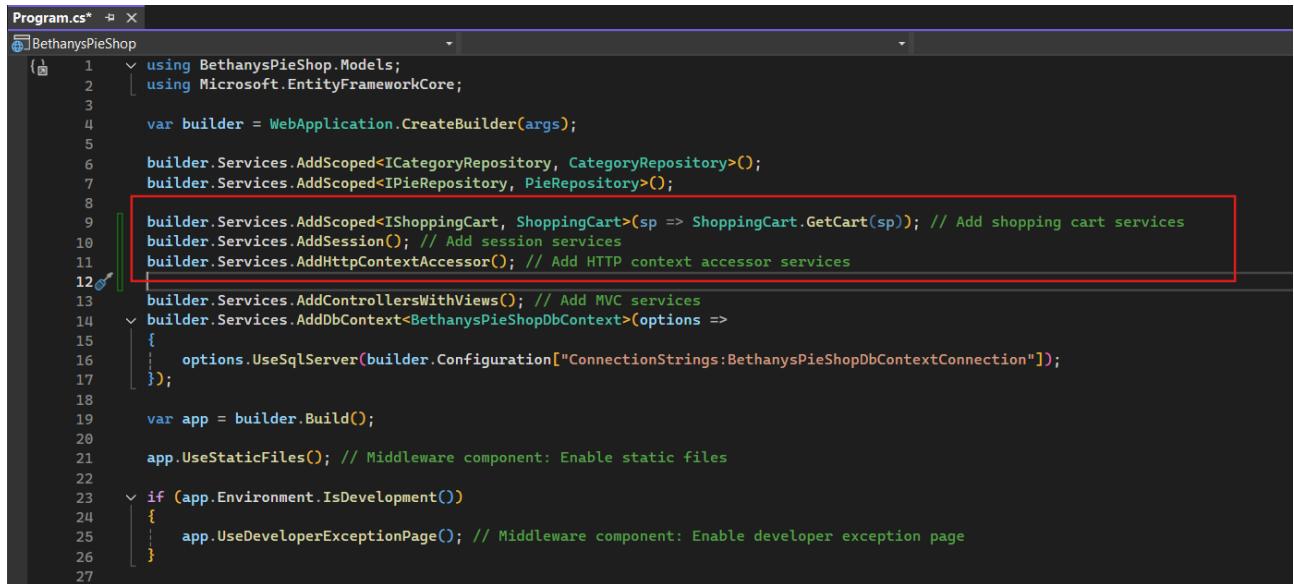
A riga 20 stiamo cercando di accedere alla sessione, è avviabile e di fatti utilizziamo come parametro (*IServiceProvider services*).

Quindi, ricapitolando, abbiamo accesso alla sessione, al DbContext e successivamente verifichiamo in base alla sessione se esiste già il valore chiamato CartId per l'utente in arrivo.

ASP.NET Core utilizza un cookie per associare diverse richieste dello stesso utente, della stessa macchina.

Dopo aver studiato il funzionamento del carrello, torniamo in **Program.cs** per aggiungere 3 servizi:

Home



```
Program.cs*  X
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, PieRepository>();
8
9  builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp)); // Add shopping cart services
10 builder.Services.AddSession(); // Add session services
11 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
12
13 builder.Services.AddControllersWithViews(); // Add MVC services
14 builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
15 {
16     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
17 });
18
19 var app = builder.Build();
20
21 app.UseStaticFiles(); // Middleware component: Enable static files
22
23 if (app.Environment.IsDevelopment())
24 {
25     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
26 }
27
```

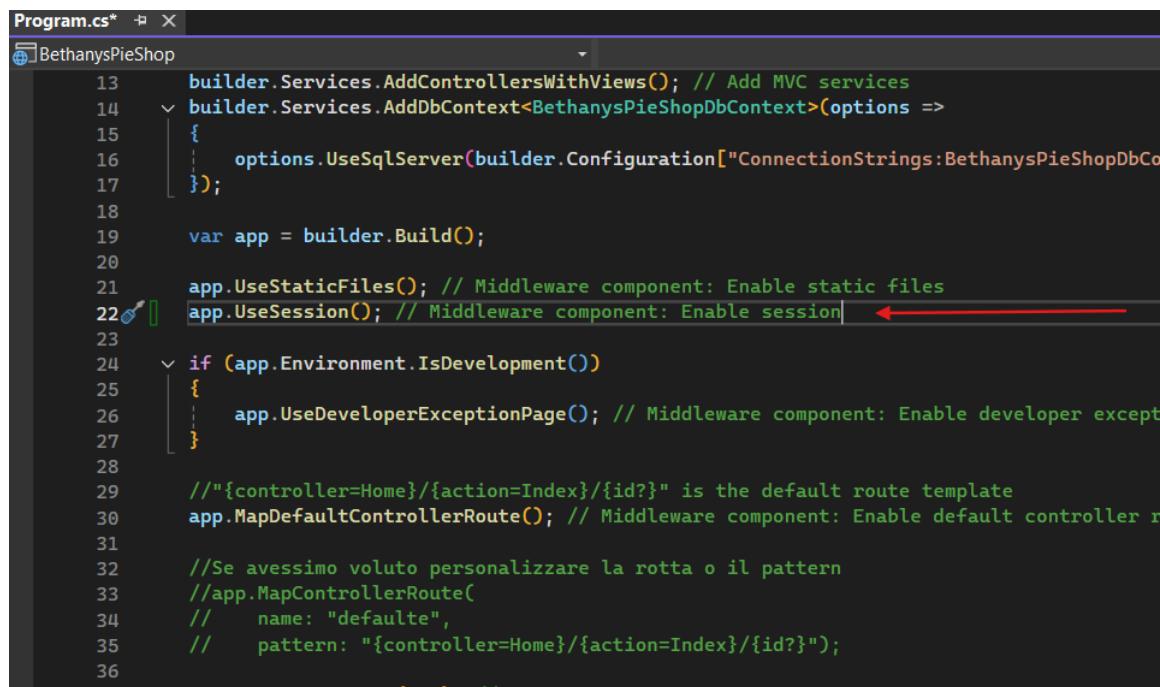
Osservazioni:

- Riga 9, invoca GetCart, **sp** è ServiceProvider necessario come parametro, utilizziamo AddScoped che creerà uno ShoppingCart per la richiesta, quindi tutti i luoghi all'interno della richiesta che hanno accesso a ShoppingCart utilizzeranno lo stesso ShoppingCart che viene istanziato con il metodo getCart()
- Riga 10, AddSession perché utilizziamo le sessioni nel Carrello della spesa
- Riga 11, AddHttpContextAccessor per poter fare questa parte qui di ShoppingCart.cs:

```
1 reference
public static ShoppingCart GetCart(IServiceProvider services)
{
    ISession? session = services.GetRequiredService<IHttpContextAccessor>()?.HttpContext?.Session;
```

Ora, sempre in **Program.cs** ci rechiamo nel middleware, per poter supportare le sessioni e aggiungiamo:

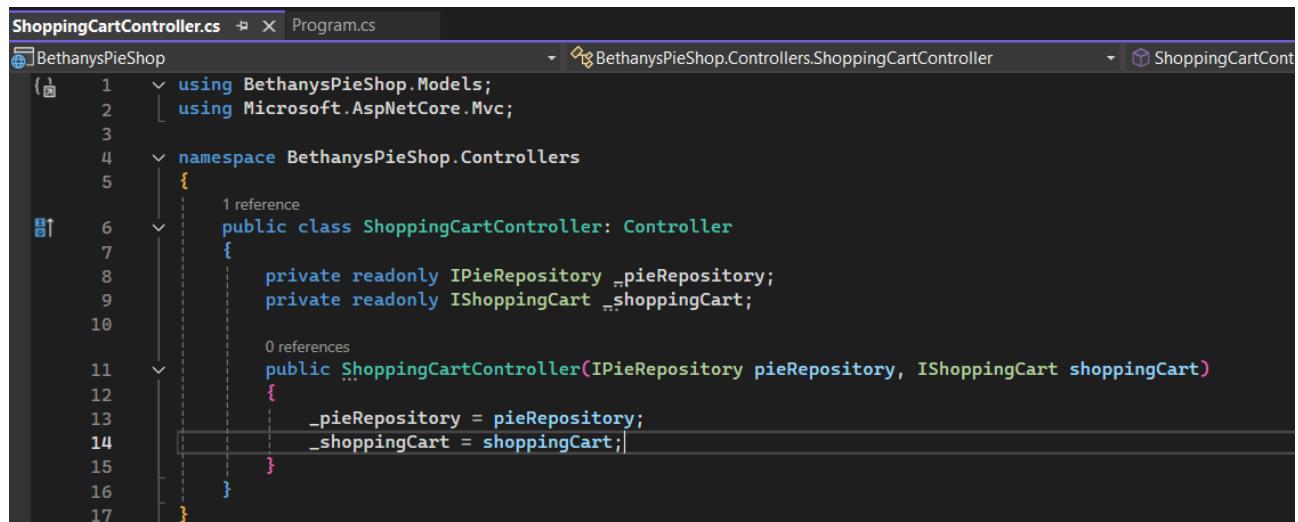
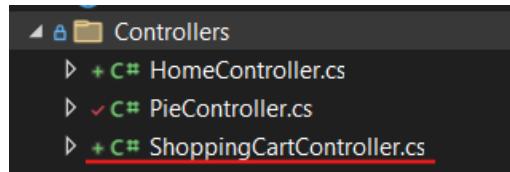
Home



```
Program.cs*  X
BethanysPieShop
13     builder.Services.AddControllersWithViews(); // Add MVC services
14     builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
15     {
16         options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbCo
17     });
18
19     var app = builder.Build();
20
21     app.UseStaticFiles(); // Middleware component: Enable static files
22     app.UseSession(); // Middleware component: Enable session ←
23
24     if (app.Environment.IsDevelopment())
25     {
26         app.UseDeveloperExceptionPage(); // Middleware component: Enable developer except
27     }
28
29     //"{controller=Home}/{action=Index}/{id?}" is the default route template
30     app.MapDefaultControllerRoute(); // Middleware component: Enable default controller r
31
32     //Se avessimo voluto personalizzare la rotta o il pattern
33     //app.MapControllerRoute(
34     //    name: "default",
35     //    pattern: "{controller=Home}/{action=Index}/{id?}");
36
37     public static void Main(string[] args)
38     {
39         CreateHostBuilder(args).Build().Run();
40     }
41 }
```

Salviamo e buildiamo e procediamo come sempre nella creazione di un controller e le viste necessarie.

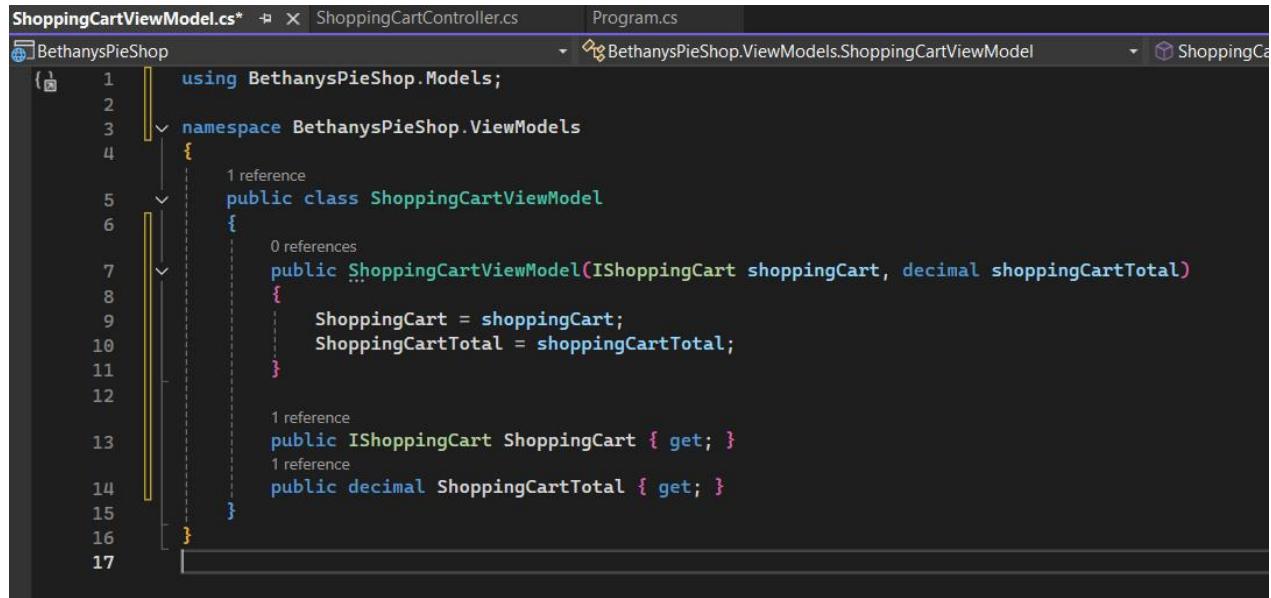
Creiamo quindi ShoppingCartController.cs



```
ShoppingCartController.cs  X  Program.cs
BethanysPieShop  BethanysPieShop.Controllers.ShoppingCartController  ShoppingCartCont
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace BethanysPieShop.Controllers
5  {
6      public class ShoppingCartController: Controller
7      {
8          private readonly IPieRepository _pieRepository;
9          private readonly IShoppingCart _shoppingCart;
10
11         public ShoppingCartController(IPieRepository pieRepository, IShoppingCart shoppingCart)
12         {
13             _pieRepository = pieRepository;
14             _shoppingCart = shoppingCart;
15         }
16     }
17 }
```

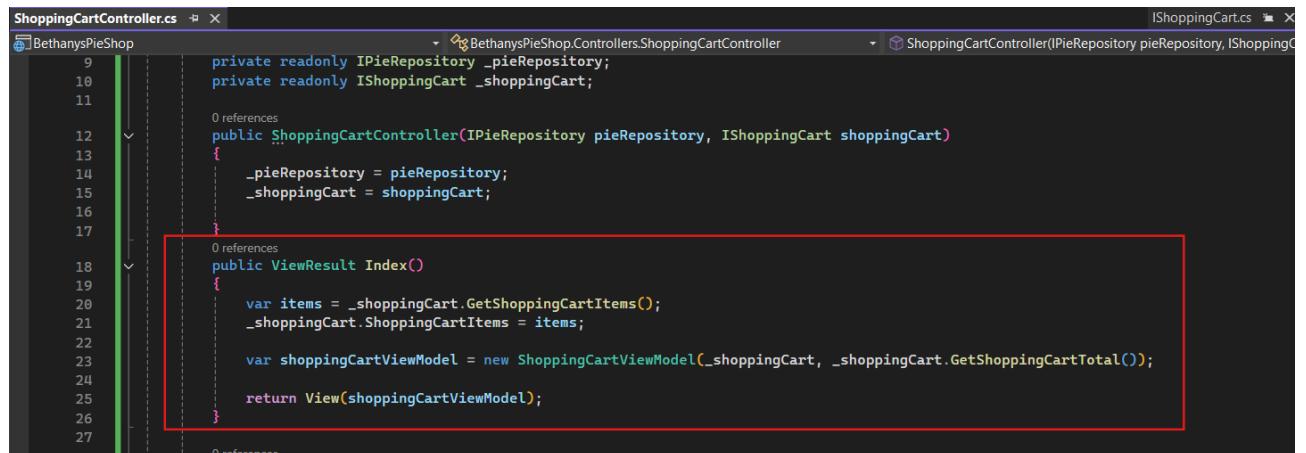
[Home](#)

Ora creiamo una **ShoppingCartViewModel** nella cartella **ViewModels**:



```
 ShoppingCartViewModel.cs*  ✎ X | ShoppingCartController.cs | Program.cs
BethanyPieShop  ↴ BethanyPieShop.ViewModels.ShoppingCartViewModel  ↴ ShoppingCart
  1  using BethanyPieShop.Models;
  2
  3  namespace BethanyPieShop.ViewModels
  4  {
  5      1 reference
  6      public class ShoppingCartViewModel
  7      {
  8          0 references
  9          public ShoppingCartViewModel(IShoppingCart shoppingCart, decimal shoppingCartTotal)
 10          {
 11              ShoppingCart = shoppingCart;
 12              ShoppingCartTotal = shoppingCartTotal;
 13          }
 14
 15          1 reference
 16          public IShoppingCart ShoppingCart { get; }
 17          1 reference
 18          public decimal ShoppingCartTotal { get; }
 19      }
 20  }
```

Ora torniamo in ShoppingCartController.cs e aggiungiamo i vari metodi:



```
 ShoppingCartController.cs  ✎ X | IShoppingCart.cs  ✎ X
BethanyPieShop  ↴ BethanyPieShop.Controllers.ShoppingCartController  ↴ ShoppingCartController(IPieRepository pieRepository, IShoppingC
  9
 10
 11
 12  private readonly IPieRepository _pieRepository;
 13  private readonly IShoppingCart _shoppingCart;
 14
 15
 16
 17
 18  0 references
 19  public ShoppingCartController(IPieRepository pieRepository, IShoppingCart shoppingCart)
 20  {
 21      _pieRepository = pieRepository;
 22      _shoppingCart = shoppingCart;
 23  }
 24
 25
 26  0 references
 27  public ViewResult Index()
 28  {
 29      var items = _shoppingCart.GetShoppingCartItems();
 30      _shoppingCart.ShoppingCartItems = items;
 31
 32      var shoppingCartViewModel = new ShoppingCartViewModel(_shoppingCart, _shoppingCart.GetShoppingCartTotal());
 33
 34      return View(shoppingCartViewModel);
 35  }
 36
 37  0 references
```

Osservazione: Chiederò al cartello tutti gli items, dopo creo un nuovo ShoppingCartViewModel, passandogli gli articoli oltre al totale ed infine ritorno la vista.

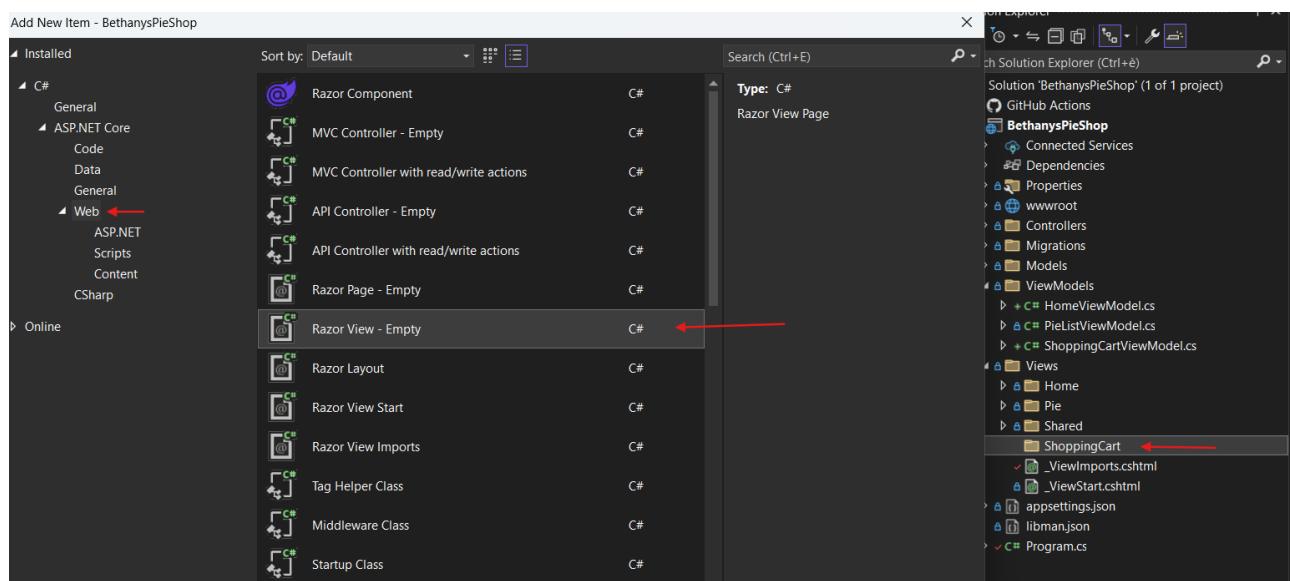
Home

```
27
28     public RedirectToActionResult AddToShoppingCart(int pieId)
29     {
30         var selectedPie = _pieRepository.AllPies.FirstOrDefault(p => p.PieId == pieId);
31
32         if (selectedPie != null)
33         {
34             _shoppingCart.AddToCart(selectedPie);
35         }
36         return RedirectToAction("Index");
37     }
38
39
40     public RedirectToActionResult RemoveFromShoppingCart(int pieId)
41     {
42         var selectedPie = _pieRepository.AllPies.FirstOrDefault(p => p.PieId == pieId);
43
44         if (selectedPie != null)
45         {
46             _shoppingCart.RemoveFromCart(selectedPie);
47         }
48         return RedirectToAction("Index");
49     }
50
51 }
```

Osservazione: Due metodi, aggiungono e rimuovono il carrello, utilizzando un Id come parametro.

Inoltre non ritorniamo una vista!

Creiamo ora una cartella ShoppingCart in Views perché il nostro controller si chiama ShoppingCart



Al suo interno creiamo index.cshtml:

[Home](#)

```
Index.cshtml*  ShoppingCartController.cs
C# BethanysPieShop
1  @model ShoppingCartViewModel
2
3  <h3 class="my-5">
4      Shopping cart
5  </h3>
6
7
8  <div class="row gx-3">
9      <div class="col-8">
10         @foreach (var line in Model.ShoppingCart.ShoppingCartItems)
11         {
12             <div class="card shopping-cart-card mb-2">
13                 <div class="row">
14                     <div class="col-md-4">
15                         
16                     </div>
17                     <div class="col-md-8">
18                         <div class="card-body">
19                             <h5 class="card-text">@line.Amount x @line.Pie.Name</h5>
20                             <div class="d-flex justify-content-between">
21                                 <h6>@line.Pie.ShortDescription</h6>
22                                 <h2>@line.Pie.Price.ToString("c")</h2>
23                             </div>
24                         </div>
25                     </div>
26                 </div>
27             </div>
28         }
29     </div>
30     <div class="col-4">
31         <div class="card shopping-cart-card p-3">
32             <div class="row">
33                 <h4 class="col">Total:</h4>
34                 <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
35             </div>
36             <hr />
37             <div class="text-center d-grid">
38             </div>
39         </div>
40     </div>
41 </div>
```

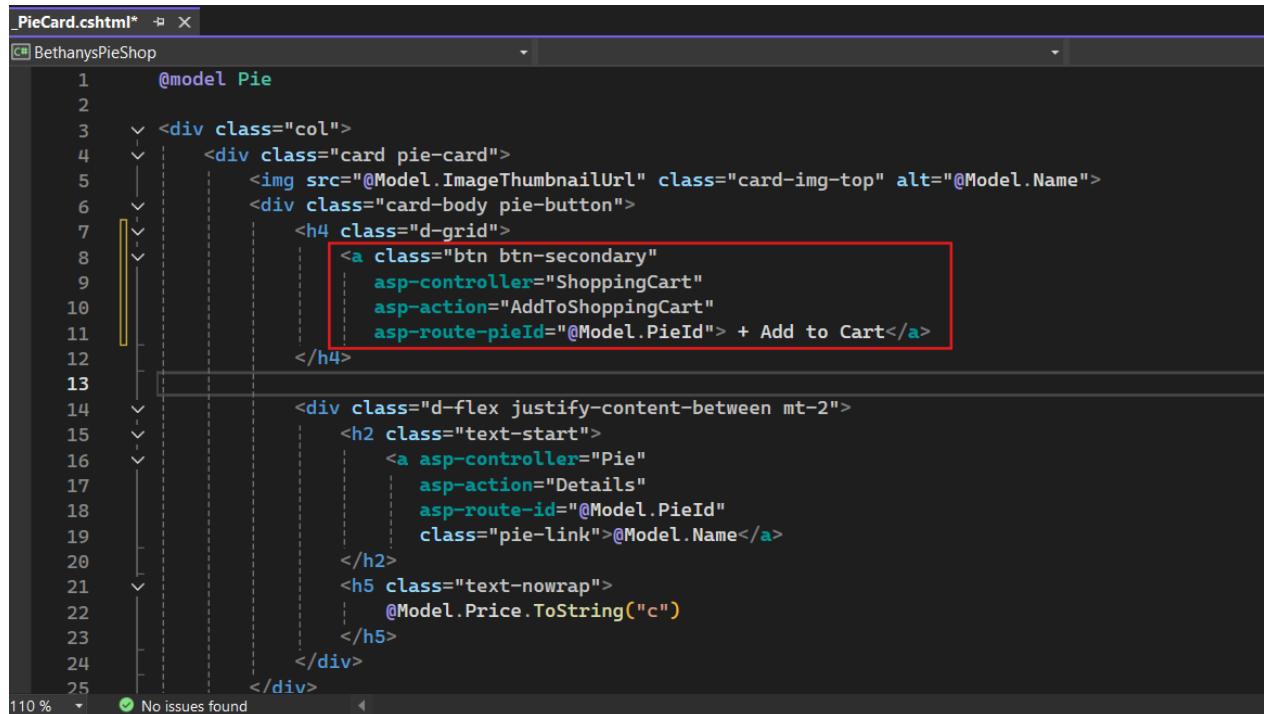
Osservazione:

Possiamo notare il totale del carrello, la stringa "c" passata come parametro del ToString() nel contesto ASP.NET Core serve per formattare un numero come valuta

```
28
29
30     </div>
31     <div class="col-4">
32         <div class="card shopping-cart-card p-3">
33             <div class="row">
34                 <h4 class="col">Total:</h4>
35                 <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
36             </div>
37             <hr />
38             <div class="text-center d-grid">
39             </div>
40         </div>
41     </div>
```

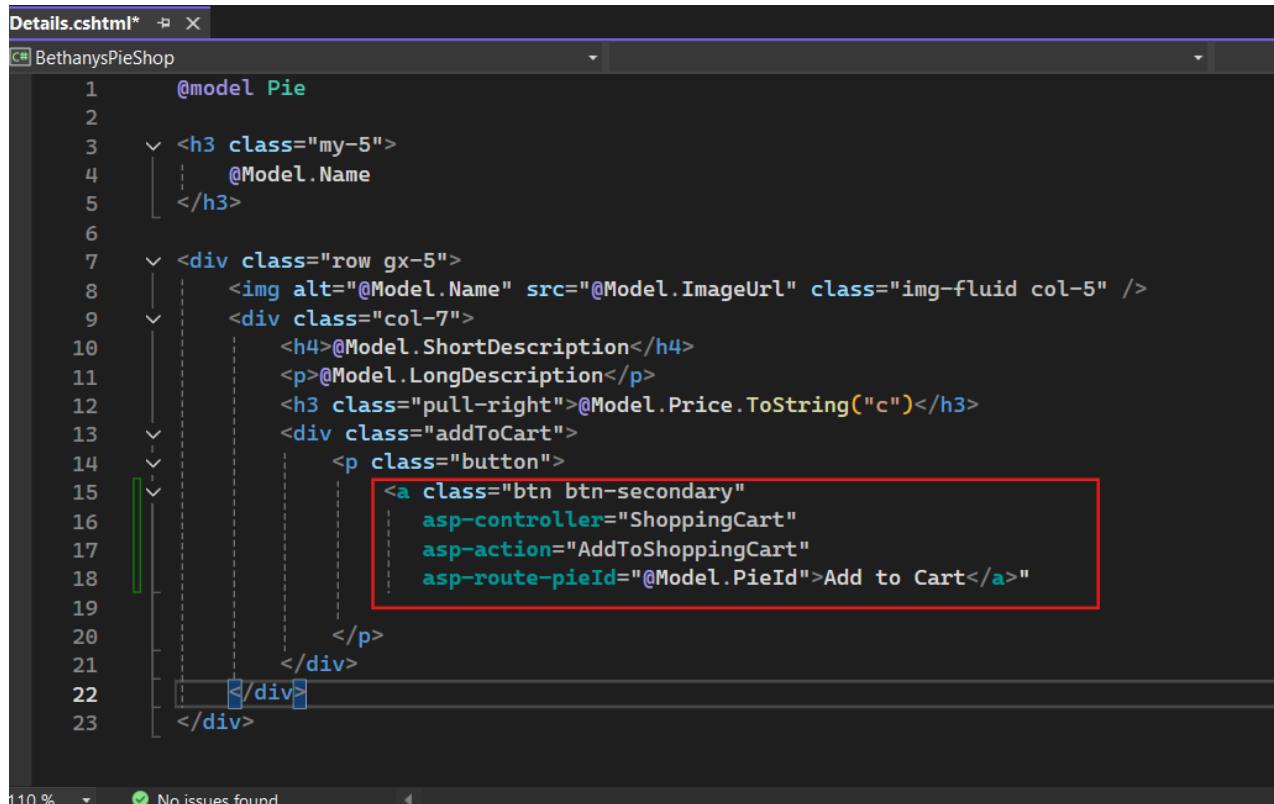
Ora dobbiamo aggiungere il button per aggiungere una torta al carrello, quindi andiamo _PieCard.cshtml presente nella cartella Shared e implementiamo il seguente codice:

Home



```
_PieCard.cshtml*
1 @model Pie
2
3 <div class="col">
4   <div class="card pie-card">
5     
6     <div class="card-body pie-button">
7       <h4 class="d-grid">
8         <a class="btn btn-secondary"
9             asp-controller="ShoppingCart"
10            asp-action="AddToShoppingCart"
11            asp-route-pieId="@Model.PieId"> + Add to Cart</a>
12       </h4>
13
14     <div class="d-flex justify-content-between mt-2">
15       <h2 class="text-start">
16         <a asp-controller="Pie"
17             asp-action="Details"
18             asp-route-id="@Model.PieId"
19             class="pie-link">@Model.Name</a>
20       </h2>
21       <h5 class="textnowrap">
22         @Model.Price.ToString("c")
23       </h5>
24     </div>
25   </div>
110 % ✓ No issues found
```

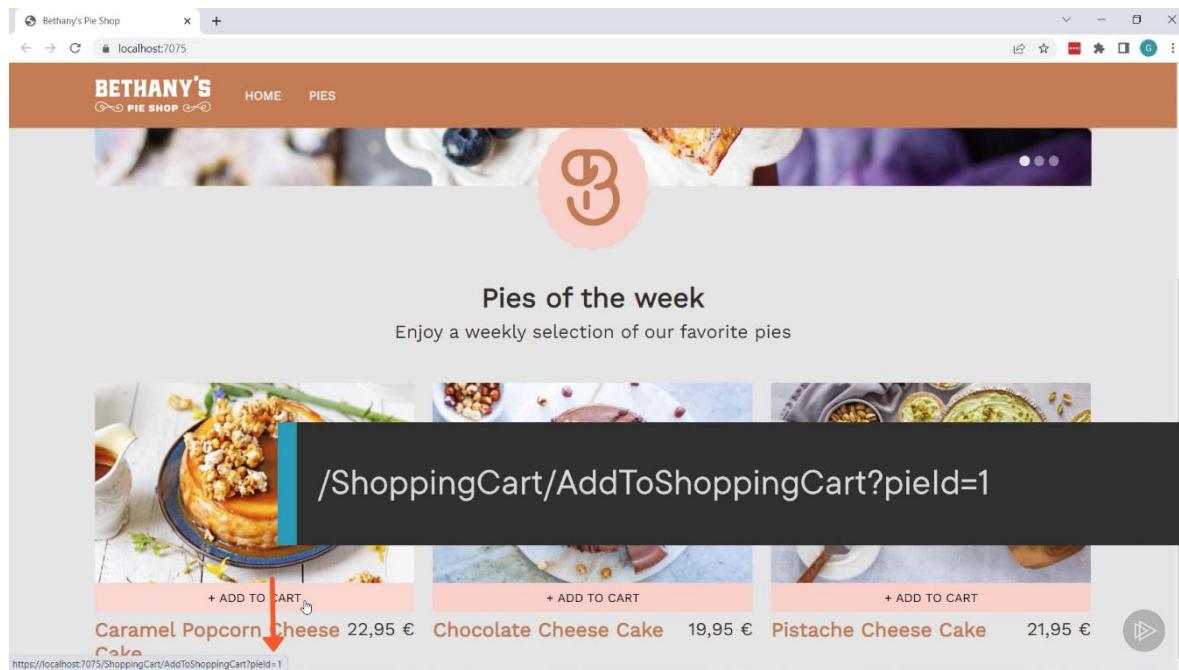
Adoperiamo in egual modo in Details.cshtml:



```
Details.cshtml*
1 @model Pie
2
3 <h3 class="my-5">
4   @Model.Name
5 </h3>
6
7 <div class="row gx-5">
8   
9   <div class="col-7">
10    <h4>@Model.ShortDescription</h4>
11    <p>@Model.LongDescription</p>
12    <h3 class="pull-right">@Model.Price.ToString("c")</h3>
13    <div class="addToCart">
14      <p class="button">
15        <a class="btn btn-secondary"
16            asp-controller="ShoppingCart"
17            asp-action="AddToShoppingCart"
18            asp-route-pieId="@Model.PieId">Add to Cart</a>
19      </p>
20    </div>
21  </div>
22</div>
23
110 % ✓ No issues found
```

Salviamo ed eseguiamo l'applicazione.

[Home](#)



The screenshot shows the 'SHOPPING CART' page. It displays two items: '1 x Caramel Popcorn Cheese Cake' and '2 x Chocolate Cheese Cake'. The total price is shown as '62,85 €'. The 'Chocolate Cheese Cake' entry has a red box around the quantity '2 x'. The URL in the browser address bar is <https://localhost:7075/ShoppingCart/AddToShoppingCart?pieId=1>.

Working with View Components

Ora utilizzeremo una componente View per mostrare su ogni pagina quanti elementi abbiamo nel nostro carrello.

```
@model Pie
<div>
  <div>
    
    <h3>@Model.Price.ToString("c")</h3>
    <p>@Model.ShortDescription</p>
```

Partial View Limitations

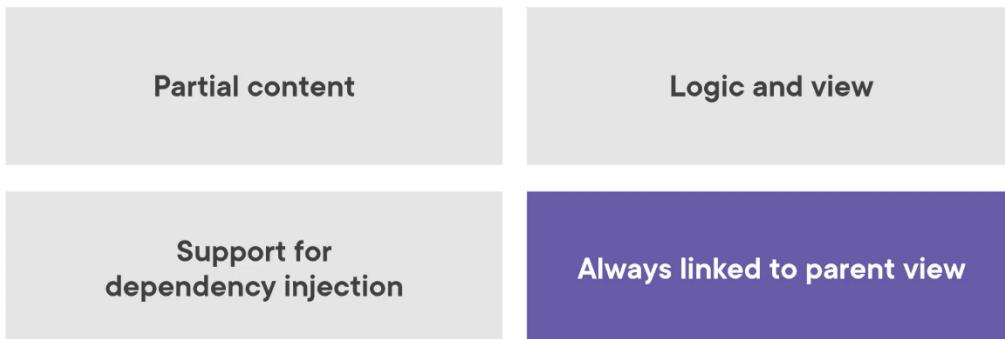
All'inizio di questo modulo abbiamo esaminato le viste parziali.

Osservazione: Le partial view funzionano con i dati che gli vengono passati dalla vista chiamante, nel nostro caso era l'elenco della panoramica della torta.

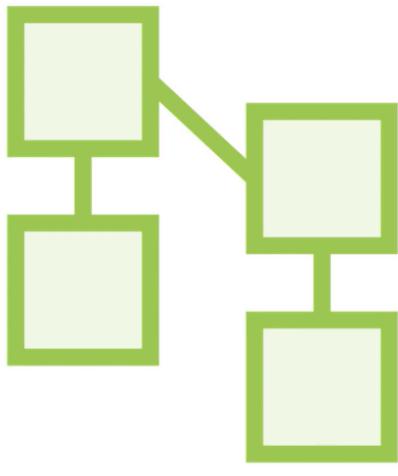
ASP.NET core fornisce una funzionalità che ci aiuterà a risolvere questo problema, visualizzare i componenti

Il View Component:

Introducing View Components



Hanno senso per componenti standalone in cui la logica si trova dietro.



View Component usage

- Login panel
- Dynamic navigation
- Shopping cart

Creating a View Component



Derive from ViewComponent



[ViewComponent]



Suffix with ViewComponent

Useremo l'approccio della classe base ViewComponent, quindi la prima opzione.

```
public class ShoppingCartSummary : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        return View(model);
    }
}
```

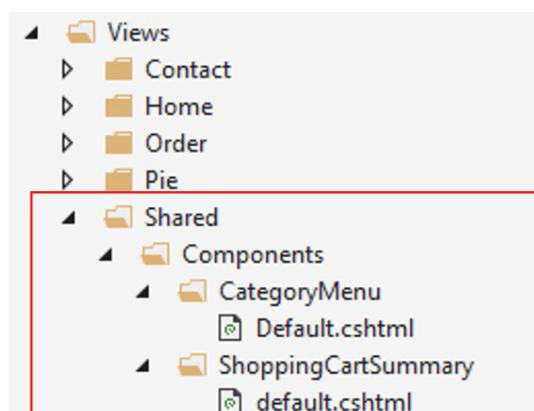
Creating a View Component

Deve essere pubblica, NON astratta e NON nidificata, proprio come i controller.

Il nostro component non deve necessariamente restituire una vista, può anche restituire una stringa.

Osservazione: Questo non è un override

View Components Search Path



Quando vogliamo richiamare un componente della vista da una vista o un layout utilizziamo la seguente espressione:

```
@await Component.InvokeAsync("ShoppingCartSummary")
```

Using the View Component

In alternativa un tag helper (ciò che faremo nel progetto):

```
<vc:shopping-cart-summary></vc:shopping-cart-summary>
```

Invoking a View Component with a Tag Helper

dobbiamo quindi specificare il nome del componente di visualizzazione.

Demo: Creating View Components

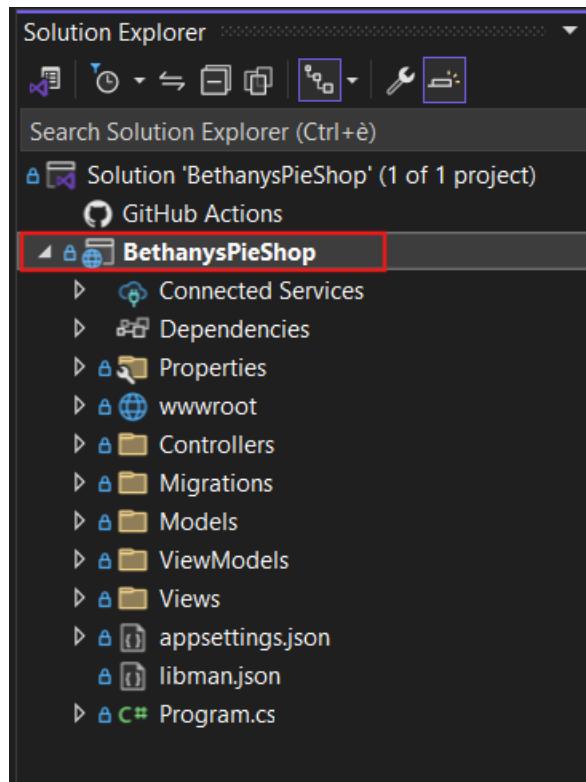
- Creating a navigation view component
- Creating a shopping cart view component

[Home](#)

Vorrei vedere qui in alto quanti articoli ho nel mio carrello,

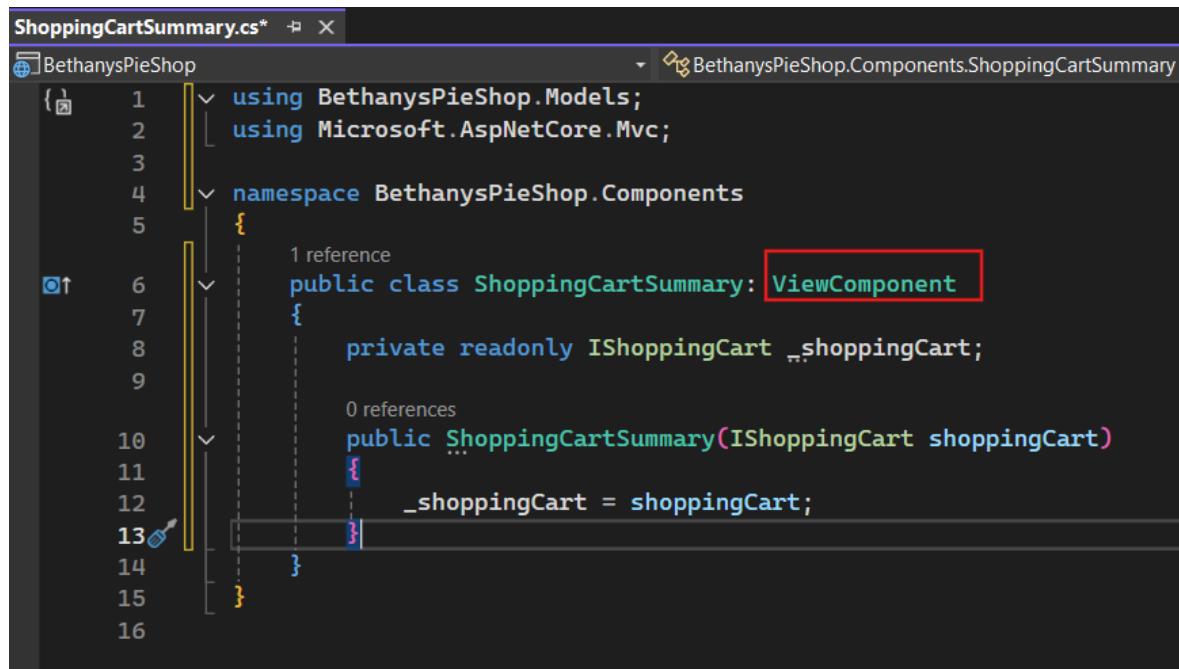
Come già detto una PartialView è più limitata rispetto a un ComponentView, in quanto dipende dalla vista padre (esempio _PieCard.cshtml)

Creiamo la cartella Components all'interno del progetto BethanysPieShop



[Home](#)

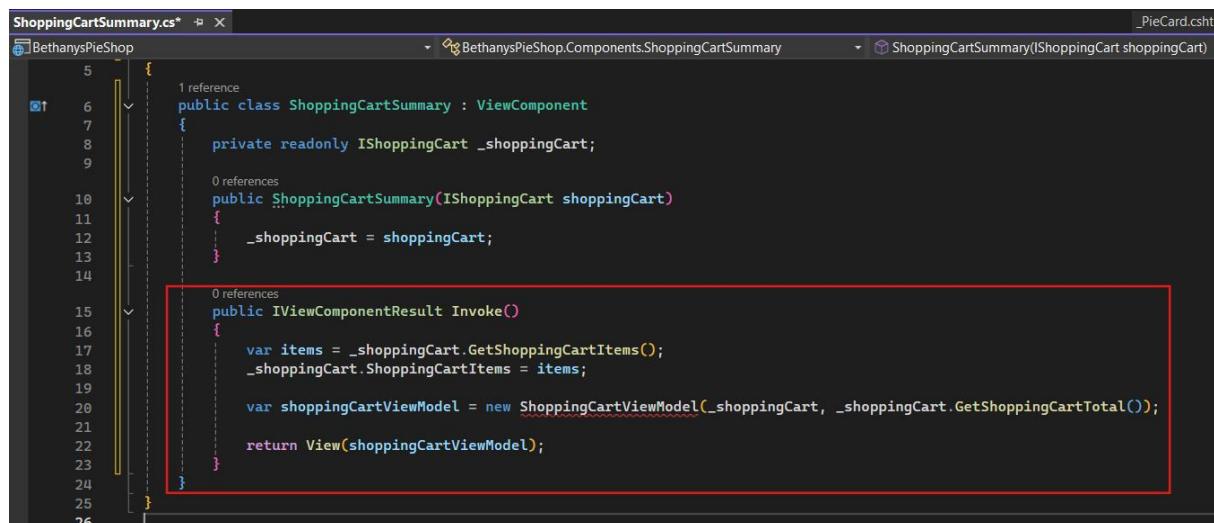
Al suo interno creiamo la classe ShoppingCartSummary.cs:



```
 ShoppingCartSummary.cs*  □ X
BethanyPieShop          ▼ BethanyPieShop.Components.ShoppingCartSummary
1  using BethanyPieShop.Models;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace BethanyPieShop.Components
5  {
6      1 reference
7      public class ShoppingCartSummary : ViewComponent
8      {
9          private readonly IShoppingCart _shoppingCart;
10
11         0 references
12         public ShoppingCartSummary(IShoppingCart shoppingCart)
13         {
14             _shoppingCart = shoppingCart;
15         }
16     }
}
```

Ci serve IShoppingCart se no come potrei contare quanti articoli ci sono nel carrello?
Questa è una istanza-scope quindi non accederemo sempre al database

Aggiungiamo il metodo Invoke():

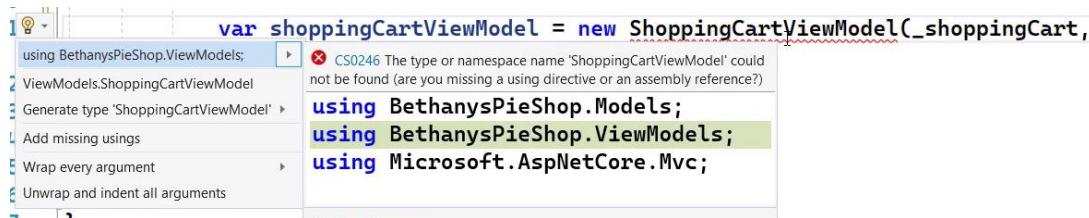


```
 ShoppingCartSummary.cs*  □ X
BethanyPieShop          ▼ BethanyPieShop.Components.ShoppingCartSummary          ▼ ShoppingCartSummary(IShoppingCart shoppingCart)
1  {
2      1 reference
3      public class ShoppingCartSummary : ViewComponent
4      {
5          private readonly IShoppingCart _shoppingCart;
6
7          0 references
8          public ShoppingCartSummary(IShoppingCart shoppingCart)
9          {
10             _shoppingCart = shoppingCart;
11         }
12
13         0 references
14         public IViewComponentResult Invoke()
15         {
16             var items = _shoppingCart.GetShoppingCartItems();
17             _shoppingCart.ShoppingCartItems = items;
18
19             var shoppingCartViewModel = new ShoppingCartViewModel(_shoppingCart, _shoppingCart.GetShoppingCartTotal());
20
21             return View(shoppingCartViewModel);
22         }
23     }
24 }
25
26 }
```

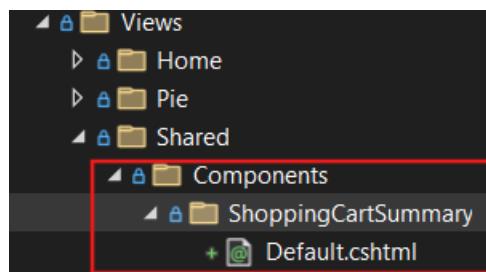
Osservazione: Simile a ciò che abbiamo fatto nel controller, però non specifichiamo un metodo di azione specifico

[Home](#)

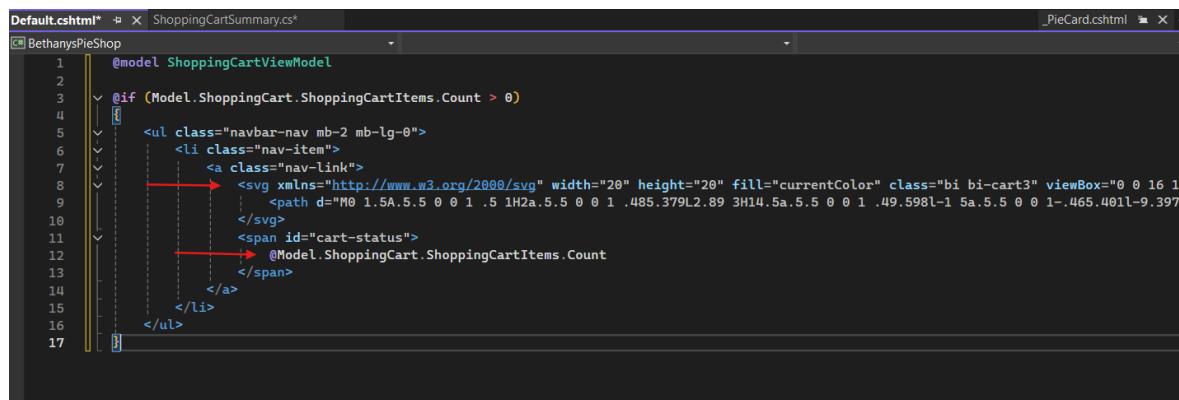
(L'errore lo risolviamo sempre con la lampadina:)



Fatto ciò, creiamo le seguenti cartelle e file:



All'interno di ShoppingCartSummary creiamo la view **Default.cshtml**



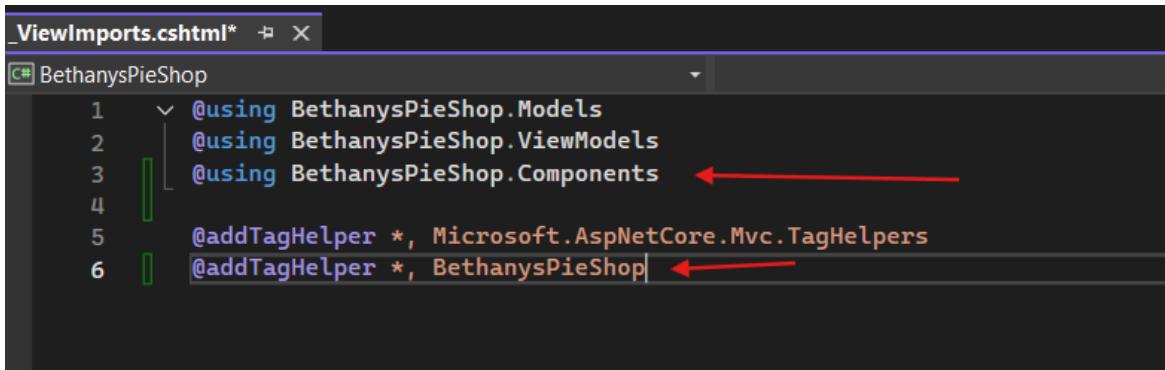
Osservazione:

Riga 8: SVG per disegnare il carrello

Riga 12: mostriamo il conteggio degli articoli

E' necessario ora aggiungere in **_ViewImports.cshtml**:

Home



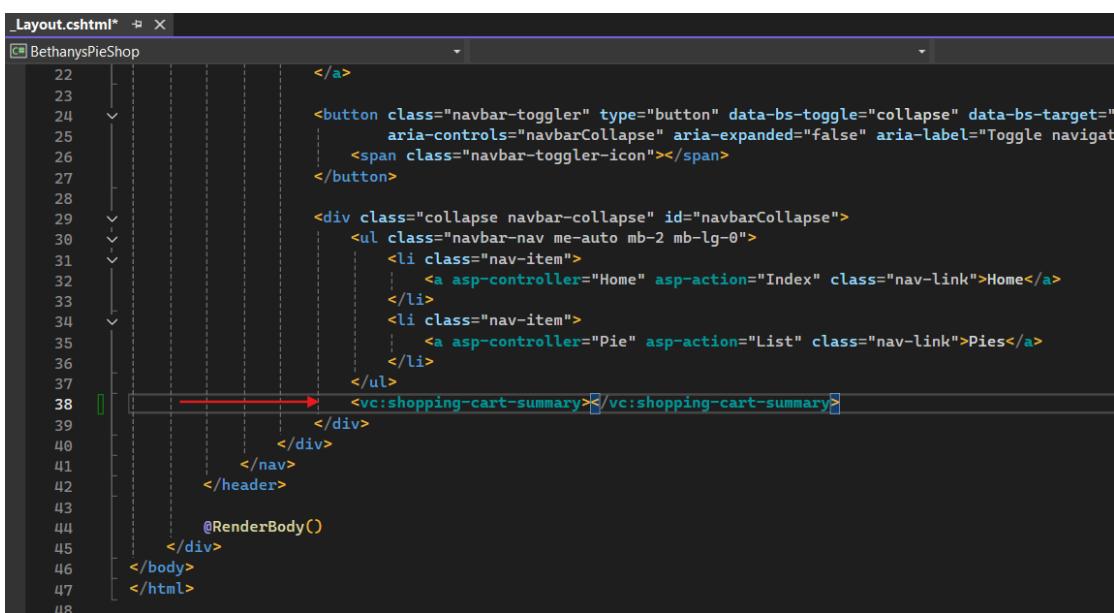
```
_ViewImports.cshtml*  ✎ X
C# BethanysPieShop
1  @using BethanysPieShop.Models
2  @using BethanysPieShop.ViewModels
3  @using BethanysPieShop.Components ←
4
5  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
6  @addTagHelper *, BethanysPieShop ←
```

The screenshot shows the _ViewImports.cshtml file in a code editor. It contains the following code:

```
1  @using BethanysPieShop.Models
2  @using BethanysPieShop.ViewModels
3  @using BethanysPieShop.Components ←
4
5  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
6  @addTagHelper *, BethanysPieShop ←
```

Two red arrows point to the last two lines of code, indicating where the custom tag helper was registered.

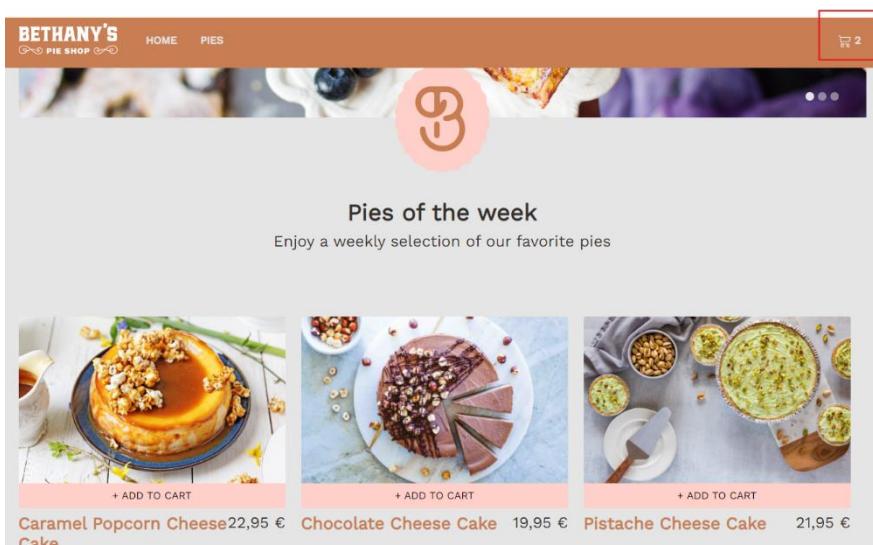
Andiamo in _Layout.cshtml per aggiungere un altro Tag Helper vc:



```
_Layout.cshtml*  ✎ X
C# BethanysPieShop
22
23
24  <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse">
25    
26  </button>
27
28  <div class="collapse navbar-collapse" id="navbarCollapse">
29    <ul class="navbar-nav me-auto mb-2 mb-lg-0">
30      <li class="nav-item">
31        <a asp-controller="Home" asp-action="Index" class="nav-link">Home</a>
32      </li>
33      <li class="nav-item">
34        <a asp-controller="Pie" asp-action="List" class="nav-link">Pies</a>
35      </li>
36    </ul>
37
38  ← <vc:shopping-cart-summary></vc:shopping-cart-summary> ←
39
40  </div>
41  </nav>
42  </header>
43
44  @RenderBody()
45
46  </body>
47  </html>
```

The screenshot shows the _Layout.cshtml file in a code editor. A red arrow points to the line containing the custom tag helper: <vc:shopping-cart-summary></vc:shopping-cart-summary>. This line is highlighted with a red background.

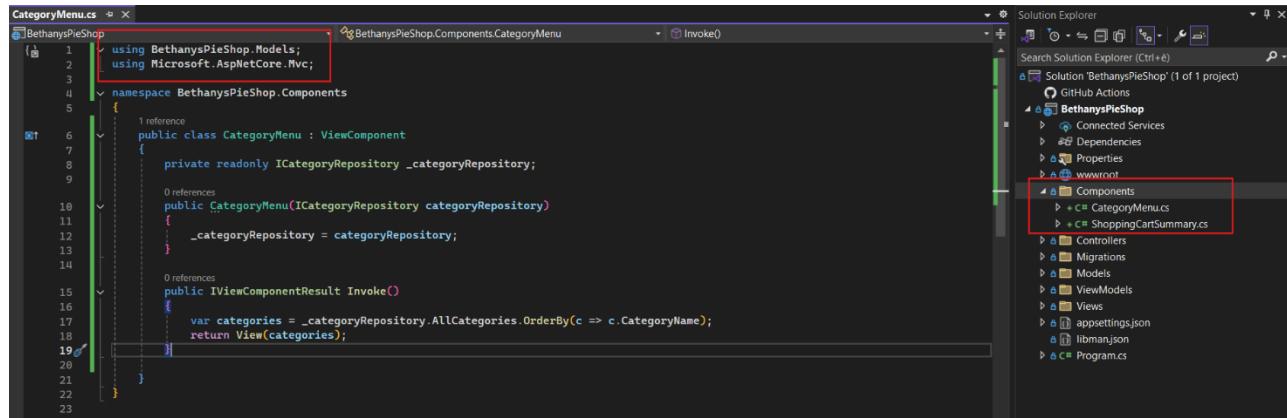
Salva, builda e dovresti ottenere il seguente output dopo aver aggiunto una torna nel carrello:



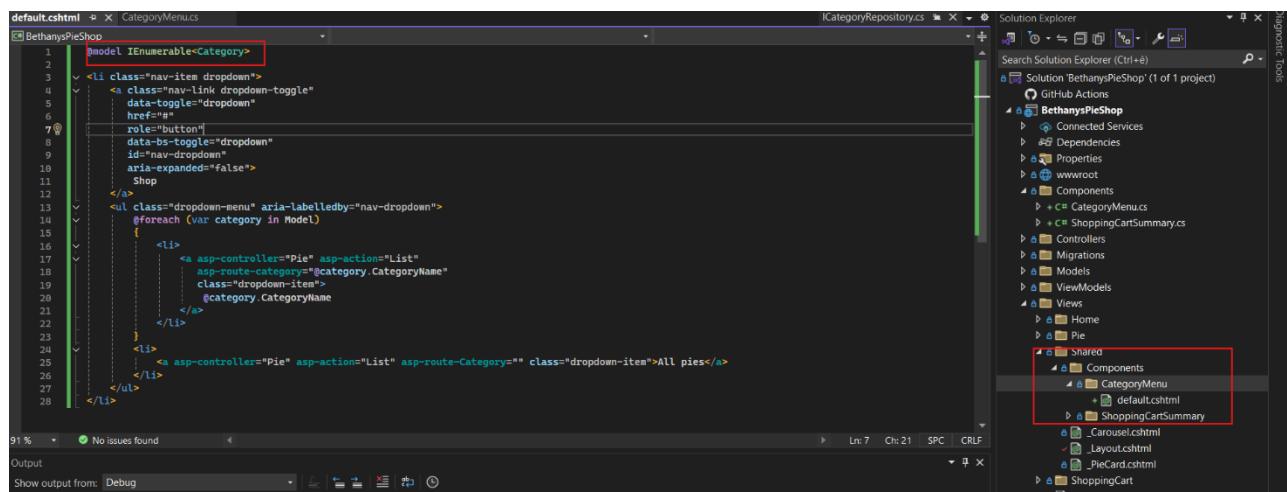
Home

Ora aggiungiamo un altro component "SHOP" al posto di "HOME" e "PIES" per poter filtrare le varie categorie.

Creiamo quindi CategoryMenu.cs in Components:



Ora, ci rechiamo in **Shared > Components** e adoperiamo nello stesso modo di ShoppingCartSummary:



Osservazioni:

Recuperiamo le categorie grazie a riga 1 e al foreach di riga 14, il bootstrap si occupa della creazione del dropdown-menu.

Possiamo notare come ogni figlio del ha collegamenti che passano all'azione Elenco controller torta, a riga 18 passiamo un nuovo parametro di categoria **però prima non lo facevamo?!**

Difatti è stato modificato anche PieController.cs nel seguente modo:

Home

```
17 //public IActionResult List()
18 //{
19 //    //ViewBag.CurrentCategory = "Cheese cakes";
20 //
21 //    //return View(_pieRepository.AllPies);
22 //
23 //    PieListViewModel piesListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
24 //    return View(piesListViewModel);
25 //}
26 //
27
28 public ViewResult List(string category)
29 {
30     IEnumerable<Pie> pies;
31     string? currentCategory;
32
33     if (string.IsNullOrEmpty(category))
34     {
35         pies = _pieRepository.AllPies.OrderBy(p => p.PieId);
36         currentCategory = "All pies";
37     }
38     else
39     {
40         pies = _pieRepository.AllPies.Where(p => p.Category.CategoryName == category)
41             .OrderBy(p => p.PieId);
42         currentCategory = _categoryRepository.AllCategories.FirstOrDefault(c => c.CategoryName == category)?.CategoryName;
43     }
44
45     return View(new PieListViewModel(pies, currentCategory));
46 }
```

Abbiamo modificato anche _Layout.cshtml nel seguente modo:

```
14 <div class="container">
15     <header>
16         <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary"
17             aria-label="Bethany's Pie Shop navigation header">
18             <div class="container-xl">
19                 <a class="navbar-brand" asp-controller="Home" asp-action="Index">
20                     
22
23                 <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse"
24                     aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
25                     <span class="navbar-toggler-icon"></span>
26                 </button>
27
28                 <div class="collapse navbar-collapse" id="navbarCollapse">
29                     <ul class="navbar-nav me-auto mb-2 mb-lg-0">
30                         <li>
31                             <vc:category-menu></vc:category-menu>
32                         <li class="nav-item">
33                             <a asp-controller="Contact" asp-action="Index" class="nav-link">Contact</a>
34                         </li>
35                         <vc:shopping-cart-summary></vc:shopping-cart-summary>
36                     </ul>
37                 </div>
38             </div>
39         </nav>
40     </header>
41     @RenderBody()
42 </div>
43 </body>
44 </html>
```

Creating a Custom Tag Helper



Tag Helpers enable server-side C# code to participate in creating and rendering HTML elements in Razor files.

Gli helper tag contengono codice C# e quel codice C# genererà HTML.

Creating a Custom Tag Helper



Inherit from base **TagHelper** class



<Name>TagHelper



Override **Process/ProcessAsync**

Creating a Custom Tag Helper

```
public class EmailTagHelper : TagHelper
{
    public override void Process(
        TagHelperContext context, TagHelperOutput output)
    {
        ...
    }
}
```

```
<email  
    address="info@bethanyspieshop.com"  
    content="Contact us">  
</email>
```

Using the Tag Helper

Dopo aver creato l'helper tag nel codice Razor,

Registering the Custom Tag Helper

```
@using BethanysPieShop.Models  
@using BethanysPieShop.ViewModels  
@addTagHelper BethanysPieShop.TagHelpers.*, BethanysPieShop  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Ora, anche se possiamo posizionare i nostri tag helper dove vogliamo,

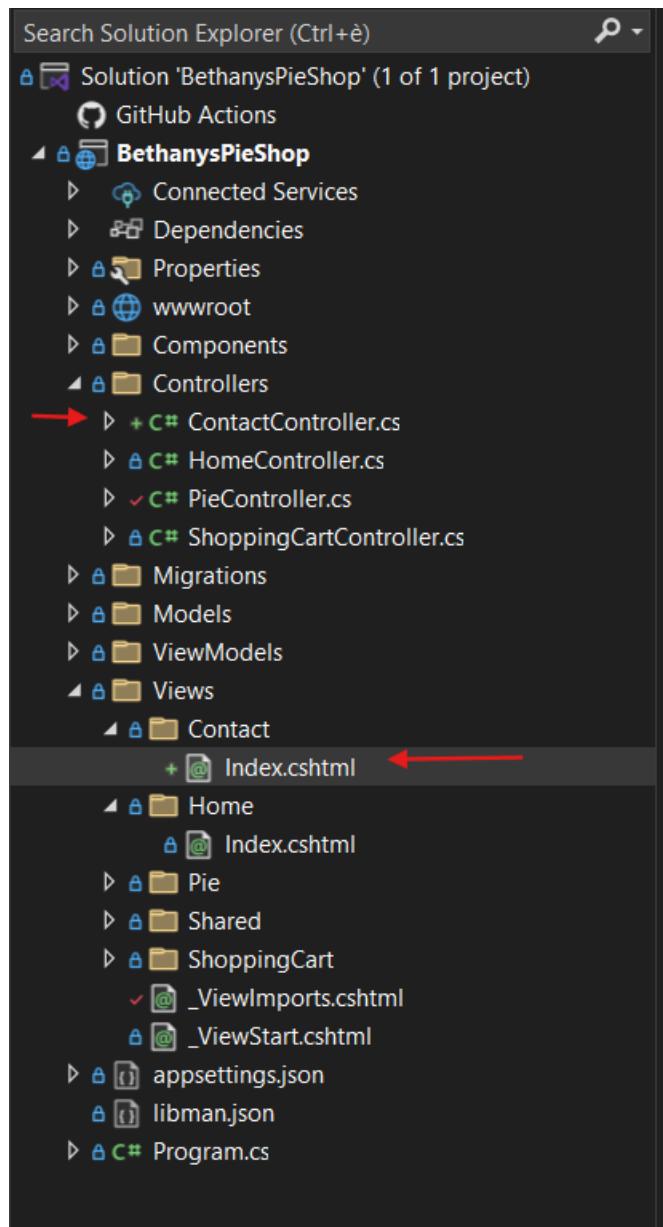
Demo: Creating the Email Tag Helper

- Creating the contact page
- Creating a tag helper
- Adding the tag helper to the contact page

Il tag helper che creeremo ci consentirà di creare più facilmente un collegamento e-mail.

Quindi ci serve creare un file **ContactController.cs** e un file **index.cshtml** in View > Contact, come segue:

Home



The screenshot shows the Visual Studio code editor with two files open:

- Index.cshtml**:

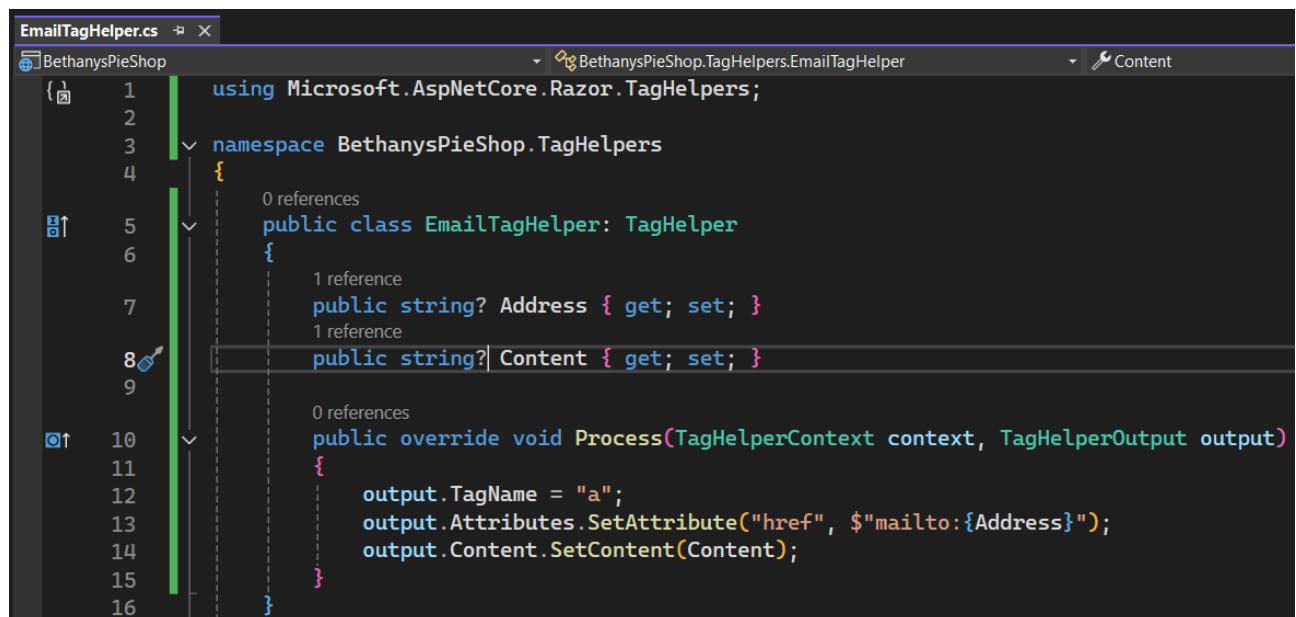
```
<h3 class="my-5">
    Contact us
</h3>

<div class="row gx-5">
    
    <div class="col-7">
        <h1>We'd Love to Hear from You!</h1>
        <h5>Please contact us by sending an email using the button below</h5>
        <email address="info@bethanyspieshop.com" content="Contact us" class="btn btn-secondary"></email>
    </div>
</div>
```
- ContactController.cs**:

```
using Microsoft.AspNetCore.Mvc;
namespace BethanysPieShop.Controllers
public class ContactController : Controller
{
    // GET: <controller>/
    public IActionResult Index()
    {
        return View();
    }
}
```

Ora passiamo alla creazione del TagHelpers, creiamo una cartella TagHelpers all'interno del progetto. Al suo interno creiamo una classe chiamata EmailTagHelper.cs

[Home](#)

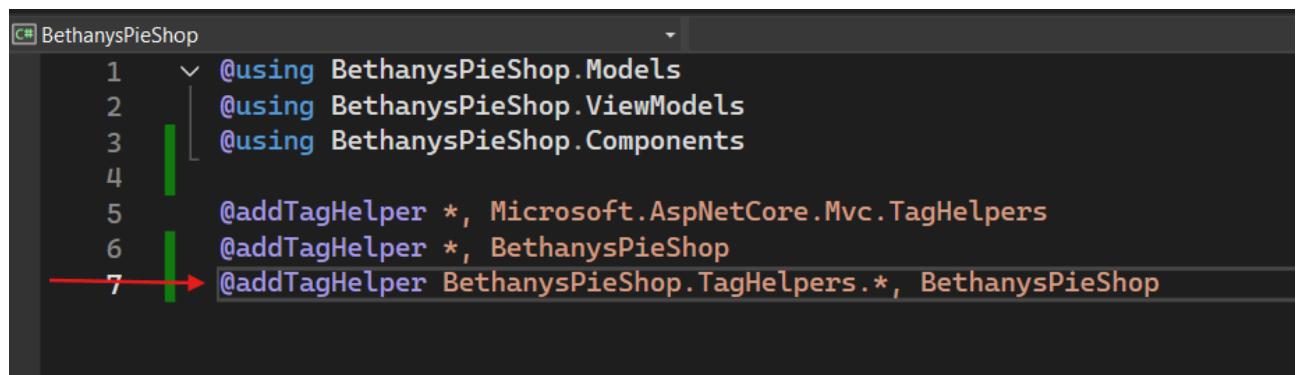


The screenshot shows the Visual Studio code editor with the file `EmailTagHelper.cs` open. The code defines a custom tag helper named `EmailTagHelper`. It includes properties for `Address` and `Content`, and a `Process` method that sets the `TagName` to "a", adds an `href` attribute with the value `mailto:{Address}`, and sets the `Content`.

```
using Microsoft.AspNetCore.Razor.TagHelpers;
namespace BethanysPieShop.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public string? Address { get; set; }
        public string? Content { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";
            output.Attributes.SetAttribute("href", $"mailto:{Address}");
            output.Content.SetContent(Content);
        }
    }
}
```

Aggiungiamo in `_ViewImports.cshtml` lo spazio dei nomi per il tag helper personalizzato:



The screenshot shows the `_ViewImports.cshtml` file in the Visual Studio code editor. A red arrow points to the line where the `@addTagHelper` directive is added to include the custom tag helper namespace.

```
@using BethanysPieShop.Models
@using BethanysPieShop.ViewModels
@using BethanysPieShop.Components
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, BethanysPieShop
@addTagHelper BethanysPieShop.TagHelpers.*, BethanysPieShop
```

Salviamo e buildiamo.

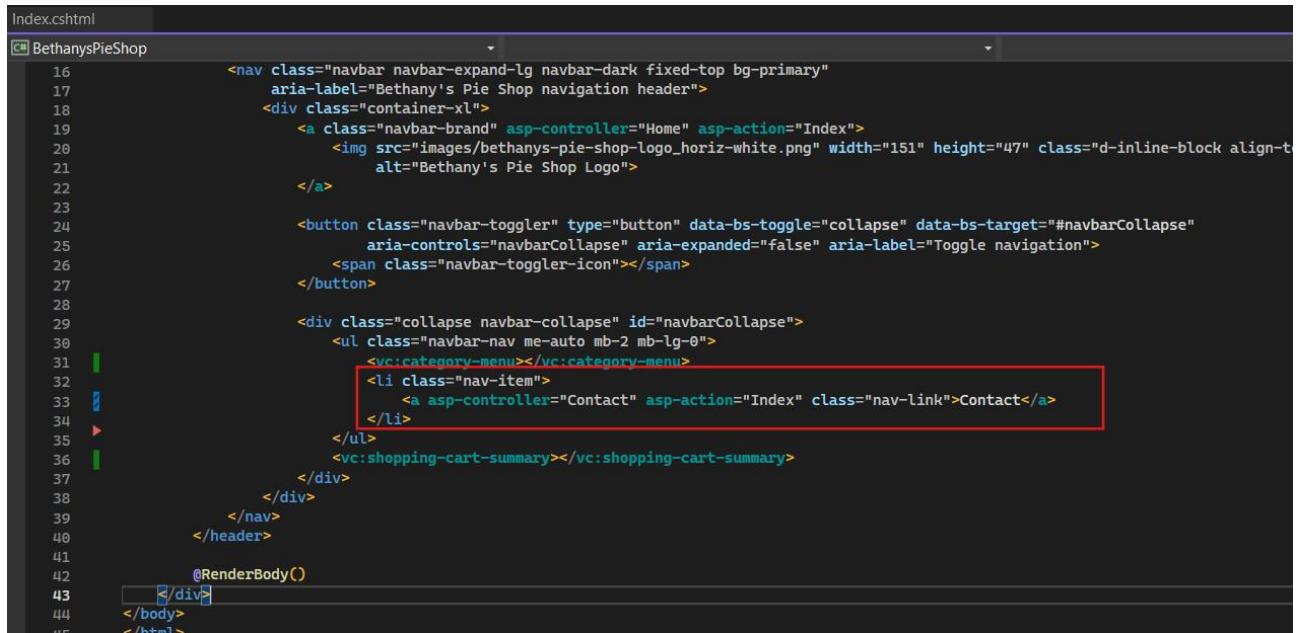
Torniamo in `Index.cshtml` di `Views > Contact` e ora possiamo utilizzare il nostro tag helper.

Noi lo abbiamo già fatto implementato scrivendo il seguente elemento:

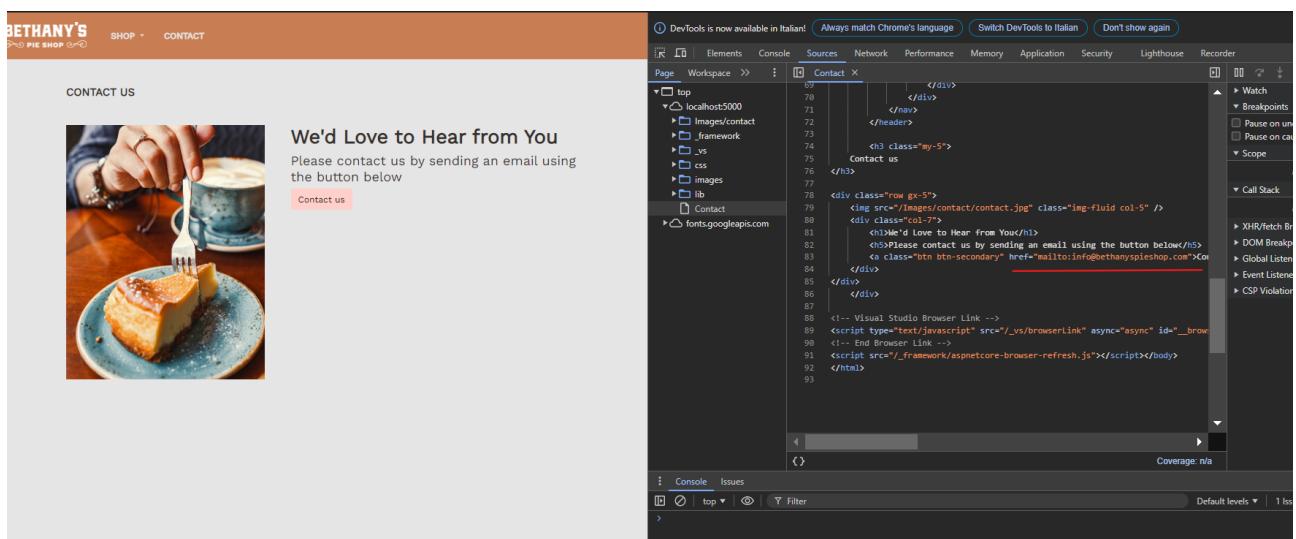
```
<email address="info@@bethanypieshop.com" content="Contact us" class="btn btn-secondary"></email>
```

Dopo aver verificato la sua presenza, torniamo in `_Layout.cshtml` per poter inserire un link in più per andare all'indice dei contatti

Home



```
Index.cshtml
BethanyPieShop
16     <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary"
17         aria-label="Bethany's Pie Shop navigation header">
18         <div class="container-xl">
19             <a class="navbar-brand" asp-controller="Home" asp-action="Index">
20                 
22             </a>
23
24             <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse"
25                 aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
26                 <span class="navbar-toggler-icon"></span>
27             </button>
28
29             <div class="collapse navbar-collapse" id="navbarCollapse">
30                 <ul class="navbar-nav me-auto mb-2 mb-lg-0">
31                     <vc:category-menu></vc:category-menu>
32                     <li class="nav-item">
33                         <a asp-controller="Contact" asp-action="Index" class="nav-link">Contact</a>
34                     </li>
35                 </ul>
36                 <vc:shopping-cart-summary></vc:shopping-cart-summary>
37             </div>
38         </div>
39     </nav>
40 </header>
41
42     @RenderBody()
43 </div>
44 </body>
45 </html>
```



BETHANY'S
PIE SHOP

SHOP · CONTACT

CONTACT US

We'd Love to Hear from You

Please contact us by sending an email using the button below

Contact us

The screenshot shows a browser window with the Bethany's Pie Shop website. The 'CONTACT' menu item is selected. On the left, there's a photo of a slice of pie on a plate. Below it is a contact form with a large button labeled 'Contact us'. To the right, the DevTools sidebar is open, showing the DOM structure. A red box highlights the 'Contact' link in the navigation bar's dropdown menu. The DevTools sidebar also shows the page structure, including the 'Contact' section and the 'Contact us' button.

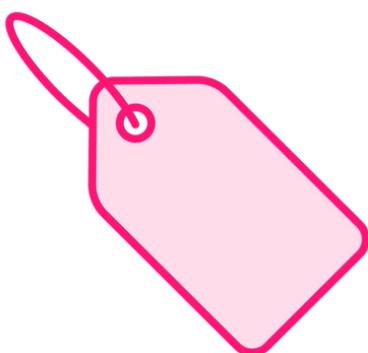
Osservazione: Una volta salvato e avviato l'applicazione, giungendo nella vista Contact possiamo notare con il DevTools che è la risorsa <mailto:infor@bethanypieshop.com> è presente ed è un link che una volta cliccato aprirà sul nostro client il servizio di posta elettronica configurato sulla nostra macchina, prova! Clicca su "Contact us" 😊👍

Working with Forms and Model Binding

- Creating a Form Using Tag Helpers
- Demo: Creating the Order Form
- Understanding Model Binding
- Demo: Accessing Posted Data Using Model Binding
- Adding Validation
- Demo: Adding Validation
- Demo: Adding Client-side Validation
- Understanding Razor Pages
- Demo: Recreating the Form Using Razor Pages

Creating a Form Using Tag Helpers

Built-in Tag Helpers in Forms



Form tag helper

Form action tag helper

Input tag helper

Label tag helper

Textarea tag helper

Select tag helper

Validation tag helpers

ASP.NET Core viene fornito con una serie di helper tag integrati che ci

Osserviamo esempi di label tag helper:

```
<label asp-for="FirstName">  
</label>  
  
<label for="FirstName">  
  FirstName  
</label>  
  
<label for="FirstName">  
  First name  
</label>  
  
<label for="FirstName"  
      class="SomeClass">  
  First name  
</label>
```

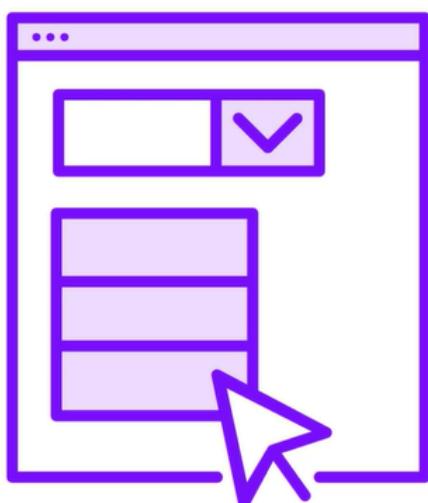
◀ Label Tag Helper

◀ Resulting HTML

◀ Attributes on Model

◀ Other HTML attributes

Form Tag Helpers



asp-controller

asp-action

asp-route-*

asp-route

asp-antiforgery

```
<form asp-action="Checkout" method="post"  
role="form">  
    ...  
</form>
```

Form Tag Helper

Generates <form> action attribute

Generates hidden token against Cross-site request forgery

Osservazione: Viene generato un Token per proteggere il modulo e il sito da richieste cross-site di attacchi di falsificazione

DEMO: Creating the Order Form

- Adding support for the Order creation
- Creating the Order Form
- Navigating to the Order form

Creeremo la possibilità di effettuare l'ordine, **gli ordini vivranno nel database.**

Creiamo i Model **Order.cs, OrderDetails.cs, IOrderRepository.cs, OrderRepository.cs**:

Home

Order.cs

```
namespace BethanysPieShop.Models
{
    public class Order
    {
        public int OrderId { get; set; }

        public List<OrderDetail>? OrderDetails { get; set; }

        public string FirstName { get; set; } = string.Empty;

        public string LastName { get; set; } = string.Empty;

        public string AddressLine1 { get; set; } = string.Empty;

        public string? AddressLine2 { get; set; }

        public string ZipCode { get; set; } = string.Empty;
    }
}
```

OrderDetail.cs

```
namespace BethanysPieShop.Models
{
    public class OrderDetail
    {
        public int OrderDetailId { get; set; }

        public int OrderId { get; set; }

        public int PieId { get; set; }

        public int Amount { get; set; }

        public decimal Price { get; set; }

        public Pie Pie { get; set; } = default!;

        public Order Order { get; set; } = default!;
    }
}
```

[Home](#)

IOrderRepository.cs OrderDetail.cs Order.cs

```
namespace BethanysPieShop.Models
{
    public interface IOrderRepository
    {
        void CreateOrder(Order order);
    }
}
```

OrderRepository.cs* IOrderRepository.cs OrderDetail.cs Order.cs

BethanysPieShop

BethanysPieShop.Models.OrderRepository

CreateOrder()

```
namespace BethanysPieShop.Models
{
    public class OrderRepository : IOrderRepository
    {
        private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;
        private readonly IShoppingCart _shoppingCart;

        public OrderRepository(BethanysPieShopDbContext bethanysPieShopDbContext, IShoppingCart shoppingCart)
        {
            _bethanysPieShopDbContext = bethanysPieShopDbContext;
            _shoppingCart = shoppingCart;
        }

        public void CreateOrder(Order order)
        {
            order.OrderPlaced = DateTime.Now;

            List<ShoppingCartItem>? shoppingCartItems = _shoppingCart.ShoppingCartItems;
            order.OrderTotal = _shoppingCart.GetShoppingCartTotal();

            order.OrderDetails = new List<OrderDetail>();

            //adding the order with its details
            foreach (ShoppingCartItem? shoppingCartItem in shoppingCartItems)
            {
                var orderDetail = new OrderDetail
                {
                    ...
                };
                ...
            }
        }
    }
}
```

Osservazioni OrderRepository:

- Necessita del DbContext, nonché di un nuovo carrello della spesa che inseriremo tramite DI.
- In CreateOrder (firma dell'interfaccia di IOrderRepository) forniamo l'implementazione, riceve un ordine che in pratica deve essere salvato nel db.
- Riga16: Aggiorniamo l'orario
- Riga18: Cerchiamo nel carrello degli acquisti gli articoli del carrello degli acquisti, chiediamo l'ordine del totale e creeremo i dettagli dell'ordine tramite il foreach di riga 25.

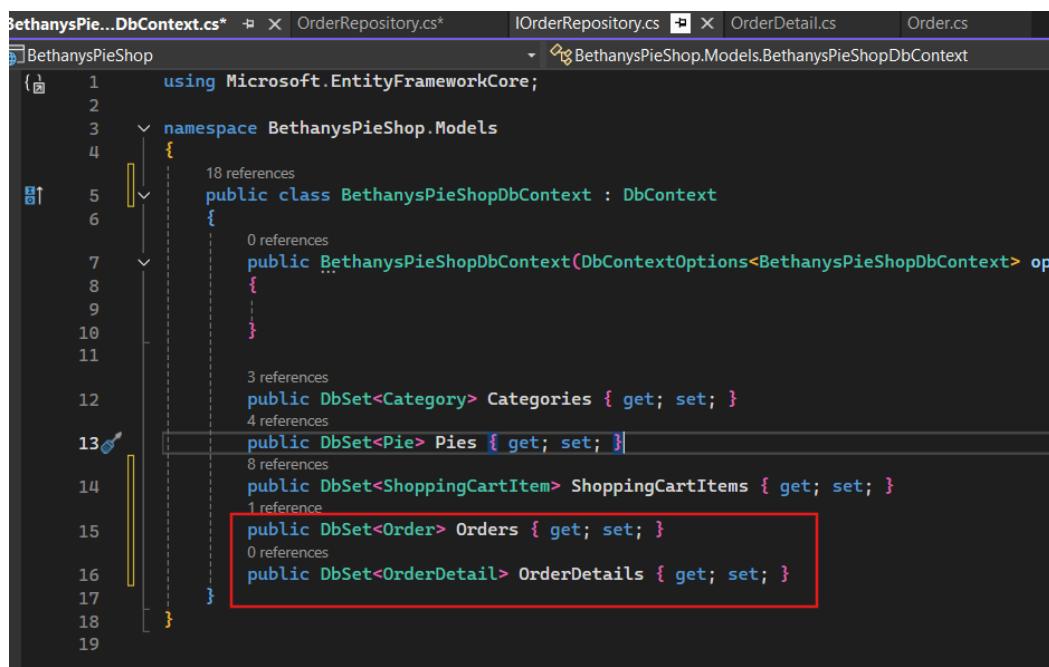
Esplorano la parte successiva del codice notiamo le operazioni di aggiunta, salvataggio e tracking del db:

[Home](#)

```
1 reference
14     public void CreateOrder(Order order)
15     {
16         order.OrderPlaced = DateTime.Now;
17
18         List<ShoppingCartItem>? shoppingCartItems = _shoppingCart.ShoppingCartItems;
19         order.OrderTotal = _shoppingCart.GetShoppingCartTotal();
20
21         order.OrderDetails = new List<OrderDetail>();
22
23         //adding the order with its details
24
25         foreach (ShoppingCartItem? shoppingCartItem in shoppingCartItems)
26         {
27             var orderDetail = new OrderDetail
28             {
29                 Amount = shoppingCartItem.Amount,
30                 PieId = shoppingCartItem.Pie.PieId,
31                 Price = shoppingCartItem.Pie.Price
32             };
33
34             order.OrderDetails.Add(orderDetail);
35
36         }
37
38         _bethanysPieShopDbContext.Orders.Add(order);
39
40         _bethanysPieShopDbContext.SaveChanges();
41
42     }
```

Notiamo l'errore a riga 37 perché nel DbContext di Bethany's Pie Shop devo ancora aggiungere un DbSet per Orders e OrderDetails.

Quindi ci rechiamo nel model **BethanysPieShopDbContext** e aggiungiamo:

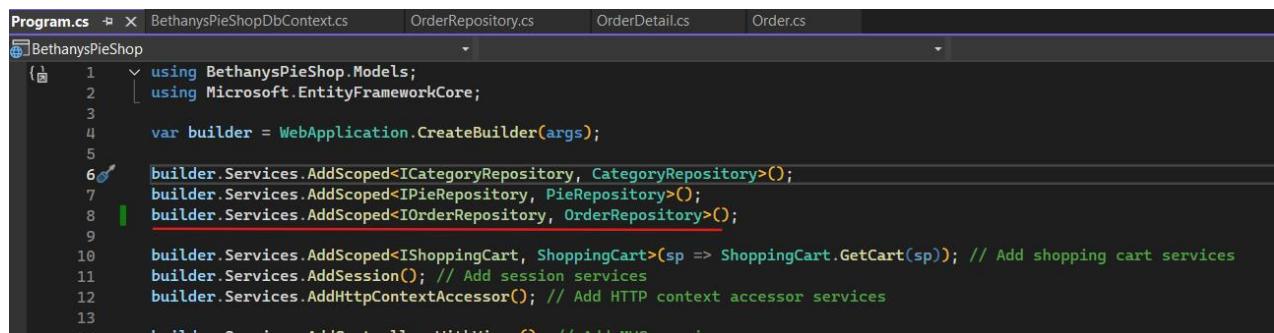


```
BethanysPie...DbContext.cs* + X OrderRepository.cs* IOrderRepository.cs + X OrderDetail.cs Order.cs
BethanysPieShop
1 using Microsoft.EntityFrameworkCore;
2
3 namespace BethanysPieShop.Models
4 {
5     public class BethanysPieShopDbContext : DbContext
6     {
7         public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
8         {
9         }
10
11
12         public DbSet<Category> Categories { get; set; }
13         public DbSet<Pie> Pies { get; set; }
14         public DbSet<ShoppingCartItem> ShoppingCartItems { get; set; }
15         public DbSet<Order> Orders { get; set; } // This line is highlighted with a red box
16         public DbSet<OrderDetail> OrderDetails { get; set; }
17
18     }
19 }
```

[Home](#)

Così possono essere salvati nel DataBase, per ShoppingCart non era necessario.

Ci rechiamo in **Program.cs** per aggiungere i servizi:



```
Program.cs  ✘ ✗ BethanysPieShopDbContext.cs OrderRepository.cs OrderDetail.cs Order.cs
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, PieRepository>();
8  builder.Services.AddScoped<IOrderRepository, OrderRepository>();
9
10 builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp)); // Add shopping cart services
11 builder.Services.AddSession(); // Add session services
12 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
13
```

Eseguiamo una build.

Ora, come sappiamo dobbiamo effettuare le migrazioni, in quanto abbiamo modificato il DbContext e aggiunto due DbSet

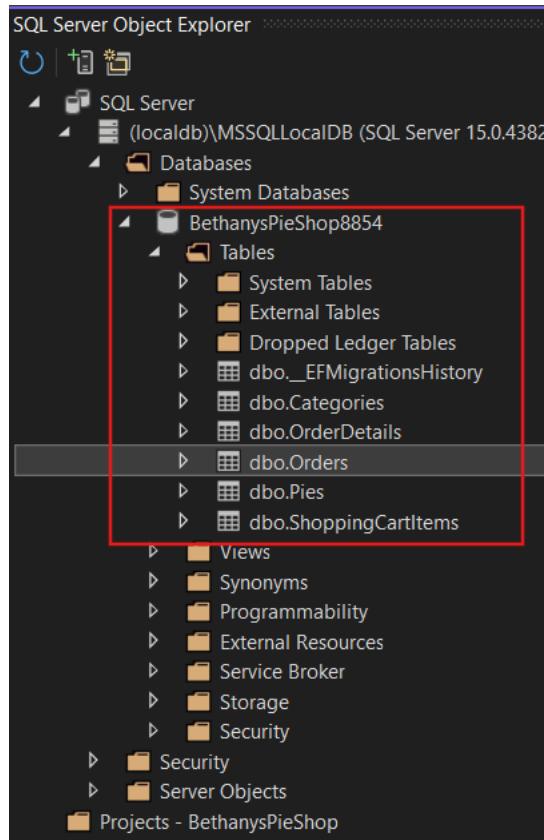
Apriamo il **PM Console e digitiamo:**

add-migration OrderAdded

Dopo l'aggiunta digitiamo:

update-database

[Home](#)



Ora possiamo procedere con la creazione del Modulo Ordine

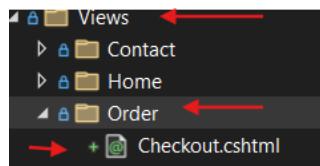
Creiamo un OrderController:

The screenshot shows the code editor for 'OrderController.cs'. The code defines a controller named 'OrderController' that inherits from 'Controller'. It has two private readonly properties: '_orderRepository' and '_shoppingCart'. The constructor takes two parameters: 'IOrderRepository' and 'IShoppingCart', and initializes the properties. The 'Checkout' method returns a 'View' result. The code is part of the 'BethanysPieShop.Controllers' namespace.

```
OrderController.cs
-----
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace BethanysPieShop.Controllers
5  {
6      public class OrderController : Controller
7      {
8          private readonly IOrderRepository _orderRepository;
9          private readonly IShoppingCart _shoppingCart;
10
11         public OrderController(IOrderRepository orderRepository, IShoppingCart shoppingCart)
12         {
13             _orderRepository = orderRepository;
14             _shoppingCart = shoppingCart;
15         }
16
17         public IActionResult Checkout()
18         {
19             return View();
20         }
21     }
22 }
```

[Home](#)

Ora ci serve la View per il metodo checkout, creiamo la cartella e la vista:



```
Checkout.cshtml*  OrderController.cs*  
BethanyPieShop  
1  @model Order  
2  
3  <form asp-action="Checkout" method="post" role="form"> ←  
4  <h3 class="my-5">  
5    You're just one step away from your delicious pies.  
6  </h3>  
7  
8  <div asp-validation-summary="All" class="text-danger"></div> ←  
9  
10 <div class="col-6">  
11   <div class="row g-2">  
12     <div class="col-12">  
13       <label asp-for="FirstName" class="form-label"></label>  
14       <input asp-for="FirstName" class="form-control" />  
15       <span asp-validation-for="FirstName" class="text-danger"></span>  
16     </div>
```

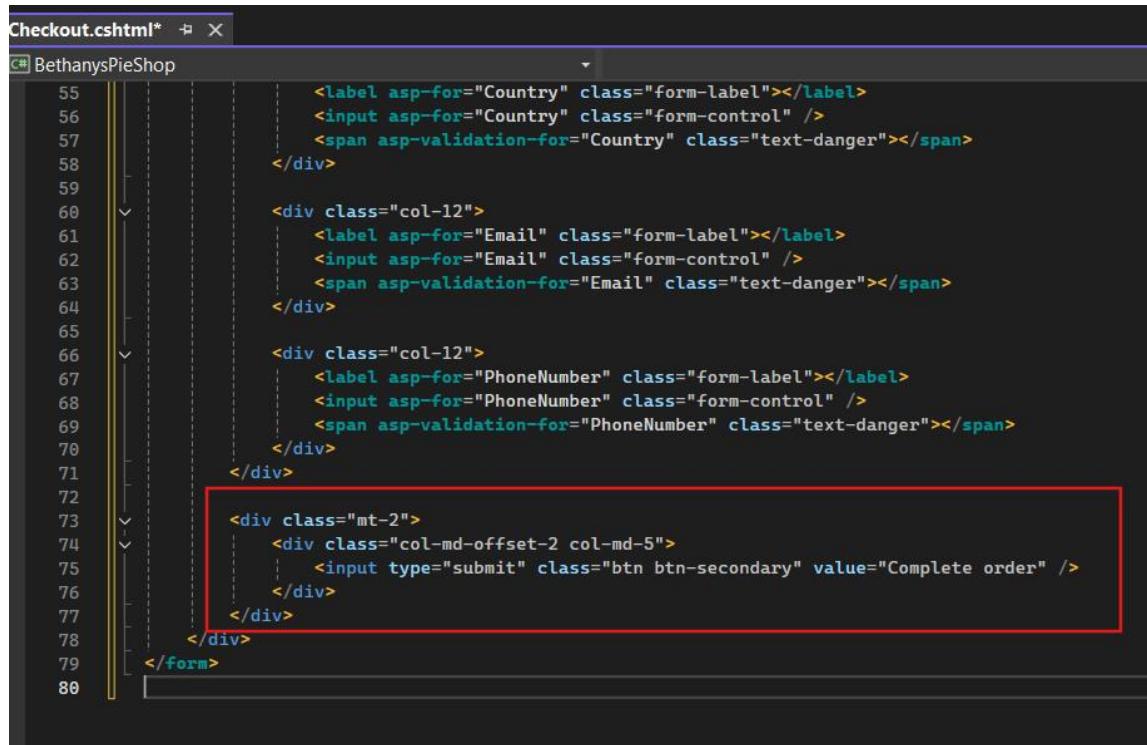
Osservazioni: Riga 13 utilizziamo a questo scopo un'etichetta, che utilizzerà l'helper tag `asp-for` che punta alla proprietà `FirstName` sull'ordine, che è il nostro modello, l'etichetta mostrerà semplicemente `FirstName`, cioè il nome della proprietà.

Quindi creo un input e li specifico anche l'asp per essere `FirstName`, questo collegherà quindi l'input `FirstName` alla proprietà `FirstName` sull'ordine quindi quando l'utente inserisce un valore per l'input `FirstName` tale valore arriverà successivamente all'interno della proprietà `FirstName` dell'oggetto `Order`.

Proseguiamo completando il codice con il resto degli input che sono quelli che compongono il nostro model `Order`.

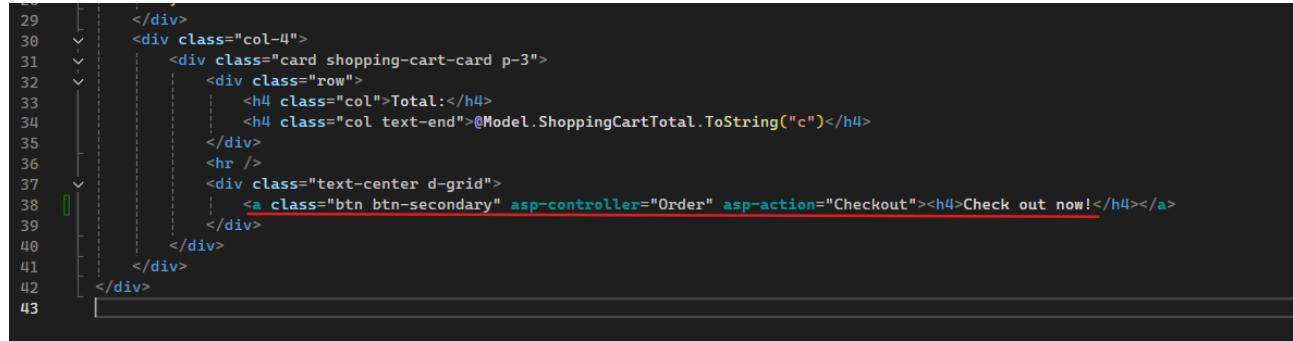
Dobbiamo anche essere in grado di inviare il modulo e si baserà su HTML standard:

[Home](#)



```
Checkout.cshtml*  X
C# BethanyPieShop
55      <label asp-for="Country" class="form-label"></label>
56      <input asp-for="Country" class="form-control" />
57      <span asp-validation-for="Country" class="text-danger"></span>
58    </div>
59
60    <div class="col-12">
61      <label asp-for="Email" class="form-label"></label>
62      <input asp-for="Email" class="form-control" />
63      <span asp-validation-for="Email" class="text-danger"></span>
64    </div>
65
66    <div class="col-12">
67      <label asp-for="PhoneNumber" class="form-label"></label>
68      <input asp-for="PhoneNumber" class="form-control" />
69      <span asp-validation-for="PhoneNumber" class="text-danger"></span>
70    </div>
71  </div>
72
73  <div class="mt-2">
74    <div class="col-md-offset-2 col-md-5">
75      <input type="submit" class="btn btn-secondary" value="Complete order" />
76    </div>
77  </div>
78</div>
79</form>
80
```

Adesso ci assicuriamo che sia possibile navigare sullo schermo dal carrello, quindi ci rechiamo in Index.cshtml presente nella cartella ShoppingCart e aggiungiamo il collegamento:



```
29      </div>
30    <div class="col-4">
31      <div class="card shopping-cart-card p-3">
32        <div class="row">
33          <h4 class="col">Total:</h4>
34          <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
35        </div>
36        <hr />
37        <div class="text-center d-grid">
38          <a class="btn btn-secondary" asp-controller="Order" asp-action="Checkout"><h4>Check out now!</h4></a>
39        </div>
40      </div>
41    </div>
42  </div>
43
```

Ora siamo in grado di poter visualizzare il Checkout del nostro Carrello e poter visionare il form per l'acquisto, naturalmente il submit non inolterà ancora nessun dato.

Understanding Model Binding

I dati inviati con il form (modulo) sono presenti nel corpo, la richiesta sarà di tipo post.

Per evitare di riscrivere codice ridondante e ottenere maggiore scalabilità, ASP.NET ci viene incontro con il model binding, il quale ci permette di ottenere l'accesso ai dati in arrivo con la richiesta.

Model Binding



Immagina di avere un metodo di azione `public ViewResult Detail(int id)`, che richiede un id di tipo int come parametro, ASP.NET core necessita che sia presente un valore per poter chiamare questo metodo, l'associazione del modello tenterà di trovare il valore per questo parametro id, quando eseguiamo una richiesta per `/Pie/Detail/1` in realtà vogliamo che l'1 vada nel parametro id.

L'associazione del modello farà questo per noi e a questo scopo, utilizza i model binders (raccoglitori di modelli), sono componenti che aiutano a fornire i valori da una determinata posizione nella richiesta, esempio esiste un m.b. che cercherà nel dati del modulo della richiesta, un altro mb cercherà la variabile di percorso e il terzo cercherà nella stringa di query, il tutto avverrà nell'ordine dell'immagine seguente, non appena viene trovata la corrispondenza, il processo si interromperà e passerà il valore al nostro metodo di azione

Model Binding



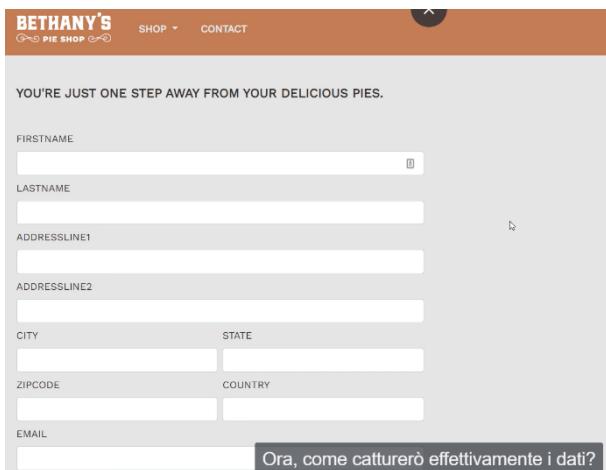
Come accennato, non funziona solo per i tipi semplici, ma anche con tipi complessi, il motore di associazione del modello esaminerà le proprietà:

[Home](#)

Binding to Complex Types



Demo: Accessing Posted Data Using Model Binding



Torniamo al nostro Controller:

[Home](#)

```
using BethanysPieShop.Models;
using Microsoft.AspNetCore.Mvc;

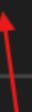
namespace BethanysPieShop.Controllers
{
    1 reference
    public class OrderController : Controller
    {
        private readonly IOrderRepository _orderRepository;
        private readonly IShoppingCart _shoppingCart;

        0 references
        public OrderController(IOrderRepository orderRepository, IShoppingCart shoppingCart)
        {
            _orderRepository = orderRepository;
            _shoppingCart = shoppingCart;
        }
        0 references
        public IActionResult Checkout()
        {
            return View();
        }
    }
}
```

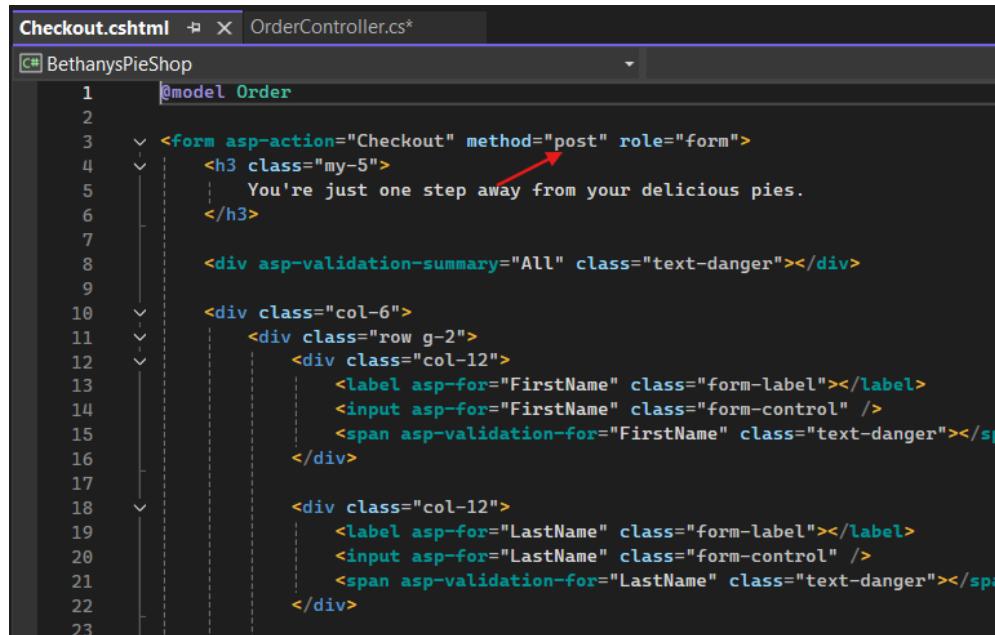
Il metodo IActionResult viene invocato quando la pagina viene caricata, quando il browser effettua la richiesta di ordine/checkout.

Verrà richiamato quando viene ricevuta una richiesta GET:

```
0 references
public IActionResult Checkout() //GET
{
    return View();
}
{
    return View();
}
```



Home

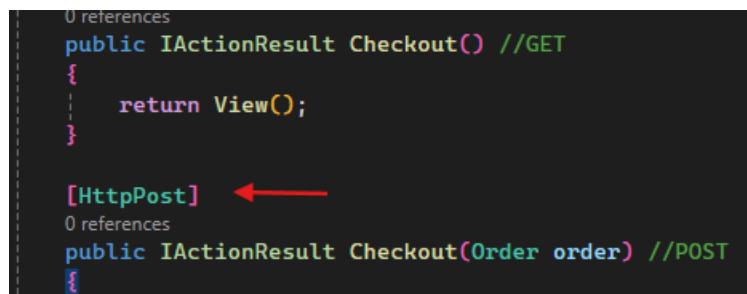


```
Checkout.cshtml  OrderController.cs*
@model Order
<form asp-action="Checkout" method="post" role="form">
    <h3 class="my-5">
        You're just one step away from your delicious pies.
    </h3>
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="col-6">
        <div class="row g-2">
            <div class="col-12">
                <label asp-for="FirstName" class="form-label"></label>
                <input asp-for="FirstName" class="form-control" />
                <span asp-validation-for="FirstName" class="text-danger"></span>
            </div>
            <div class="col-12">
                <label asp-for="LastName" class="form-label"></label>
                <input asp-for="LastName" class="form-control" />
                <span asp-validation-for="LastName" class="text-danger"></span>
            </div>
        </div>
    </div>
```

In Checkout.cshtml abbiamo un post, devo creare un altro metodo di azione che verrà richiamato quando viene ricevuto un post

Torniamo in OrderController.cs e implementiamo:

Iniziamo col definire l'attributo:



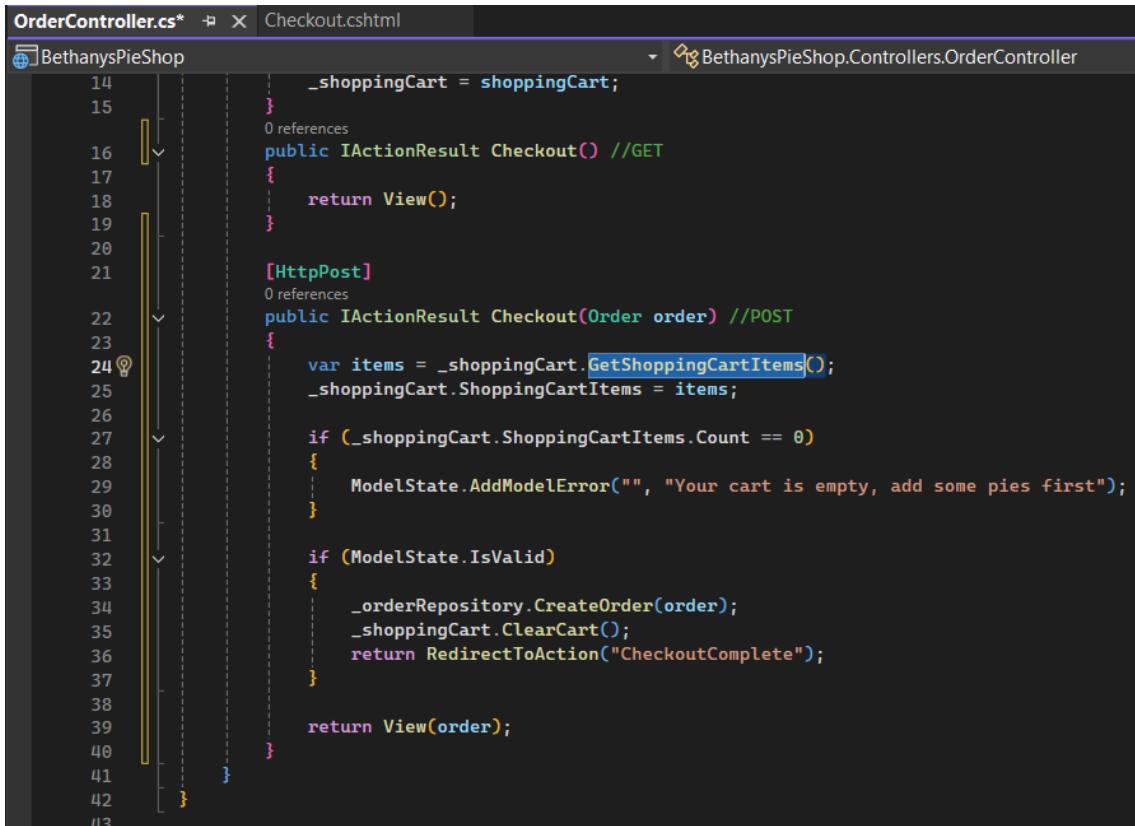
```
0 references
public IActionResult Checkout() //GET
{
    return View();
}

[HttpPost] ←
0 references
public IActionResult Checkout(Order order) //POST
{
```

Osservazione: Nel GET non abbiamo scritto l'attributo perché di default è un GET

Implementazione del post completa:

[Home](#)



The screenshot shows a code editor window for a C# file named OrderController.cs. The file is part of a project called BethanyPieShop. The code implements a controller with two actions: Checkout (GET) and Checkout (POST). The POST action checks if the shopping cart is empty, adds an error message if it is, and then creates a new order if the model state is valid. If the ModelState is invalid, it returns the same view. A yellow vertical bar on the left indicates a breakpoint is set on line 24.

```
OrderController.cs*  X  Checkout.cshtml
BethanyPieShop
14     _shoppingCart = _shoppingCart;
15 }
16 }
17 }
18 }
19 }
20 }
21 [HttpPost]
22 public IActionResult Checkout(Order order) //POST
23 {
24     var items = _shoppingCart.GetShoppingCartItems();
25     _shoppingCart.ShoppingCartItems = items;
26
27     if (_shoppingCart.ShoppingCartItems.Count == 0)
28     {
29         ModelState.AddModelError("", "Your cart is empty, add some pies first");
30     }
31
32     if (ModelState.IsValid)
33     {
34         _orderRepository.CreateOrder(order);
35         _shoppingCart.ClearCart();
36         return RedirectToAction("CheckoutComplete");
37     }
38
39     return View(order);
40 }
41 }
42
/3
```

Osservazioni:

Verifichiamo prima che l'utente abbia effettivamente articoli nel carrello.

Se così non fosse, l'if di riga 27 da un avvertimento con un **ModelState**, è un prodotto secondario, collaterale dell'associazione del modello, si occupa di mostrare errori.

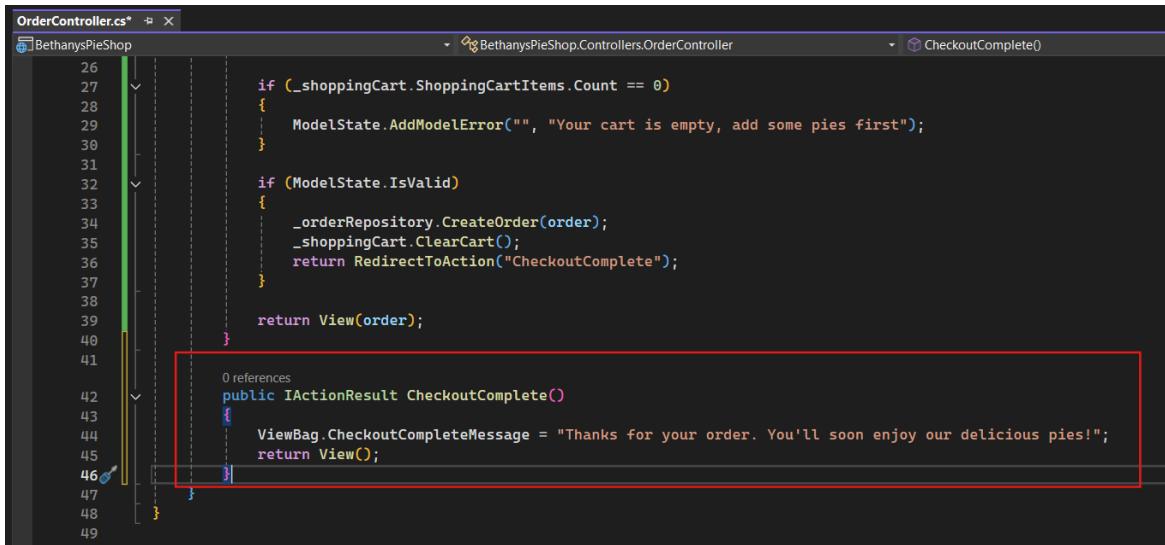
Il ModelState ha anche una proprietà di Validità (riga 32), è valida se l'associazione del modello è andata a buon fine e non stati aggiunti ulteriori errori.

In caso positivo, viene creato l'ordine con `_orderRepository` e quindi nel db, viene cancellata la cart, ed effettuiamo un reindirizzamento a un'altra azione (Riga 37: che non abbiamo ancora visto).

Se il ModelState non era valido, ritorniamo la stessa vista(riga 39).

Non abbiamo ancora `CheckoutComplete`, la creiamo subito:

Home



```
OrderController.cs*  x
BethanyPieShop  BethanyPieShop.Controllers.OrderController  CheckoutComplete()

26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

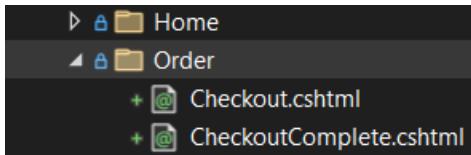
    if (_shoppingCart.ShoppingCartItems.Count == 0)
    {
        ModelState.AddModelError("", "Your cart is empty, add some pies first");
    }

    if (ModelState.IsValid)
    {
        _orderRepository.CreateOrder(order);
        _shoppingCart.ClearCart();
        return RedirectToAction("CheckoutComplete");
    }

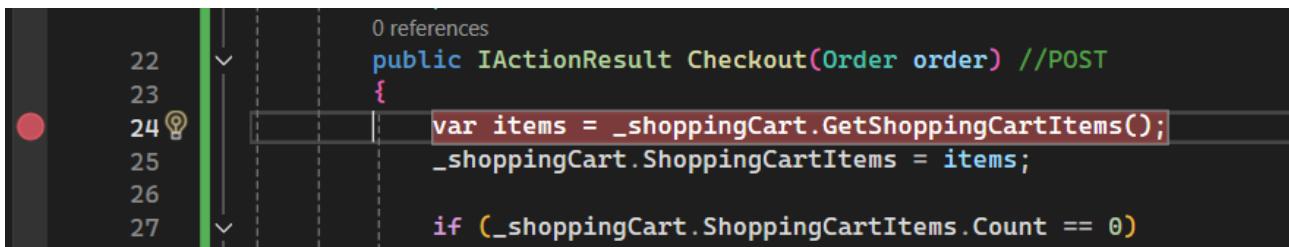
    return View(order);
}

0 references
public IActionResult CheckoutComplete()
{
    ViewBag.CheckoutCompleteMessage = "Thanks for your order. You'll soon enjoy our delicious pies!";
    return View();
}
```

Creiamo la vista:



Inseriamo un breakpoint a riga 24 di OrderController.cs e avviamo:



```
OrderController.cs*  x
BethanyPieShop  BethanyPieShop.Controllers.OrderController  Checkout()

22
23
24 var items = _shoppingCart.GetShoppingCartItems();
25 _shoppingCart.ShoppingCartItems = items;
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

Di default avremo la vista del checkout senza dati, inseriamo dei dati manualmente:

Home

YOU'RE JUST ONE STEP AWAY FROM YOUR DELICIOUS PIES.

FIRSTNAME
Marco

LASTNAME
Power

ADDRESSLINE1
Some Street

ADDRESSLINE2

CITY STATE
Amsterdam null

ZIPCODE COUNTRY
1011 Netherlands

EMAIL
markfake@gmail.com

PHONENUMBER
123456789

Complete order

Click su *Complete order* questo attiverà il trigger dell'invio del modulo, quindi tornando nel nostro IDE con il debug attivo, noteremo che l'oggetto `ordern` ha effettuato con successo il model binding:

```
0 references
public OrderController(IOrderRepository orderRepository, ShoppingCart shoppingCart)
{
    _orderRepository = orderRepository;
    _shoppingCart = shoppingCart;
}

0 references
public IActionResult Checkout() //GET
{
    return View();
}

[HttpPost]
0 references
public IActionResult Checkout(Order order)
{
    var items = _shoppingCart.GetShoppingCartItems();
    _shoppingCart.ShoppingCartItems = items;

    if (_shoppingCart.ShoppingCartItems.Count == 0)
        return RedirectToAction("Index");
}

0 references
public void Dispose()
{
    _orderRepository?.Dispose();
    _shoppingCart?.Dispose();
}
```

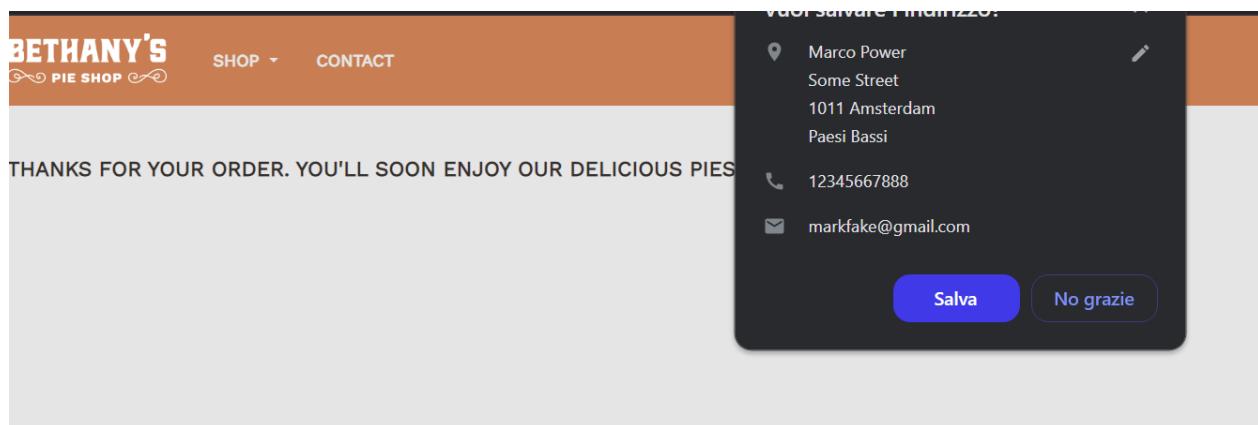
The screenshot shows a debugger tooltip for the variable `order` of type `BethanysPieShop.Models.Order`. The tooltip displays the following properties and their values:

Property	Type	Value
AddressLine1	View	"Some Street"
AddressLine2	View	null
City	View	"Amsterdam"
Country	View	"Netherlands"
Email	View	"markfake@gmail.com"
FirstName	View	"Marco"
LastName	View	"Power"
OrderDetails	View	null
OrderId	View	0
OrderPlaced	View	{01/01/0001 00:00:00}
OrderTotal	View	0
PhoneNumber	View	"123456789"
State	View	null
ZipCode	View	"1011"

Proseguendo alle istruzioni successive, noteremo che `isValid` è true se tutti i campi obbligatori sono stati inseriti:

Home

```
ethanysPieShop OrderController.cs
22 public IActionResult Checkout(Order order) //POST
23 {
24     var items = _shoppingCart.GetShoppingCartItems();
25     _shoppingCart.ShoppingCartItems = items;
26
27     if (_shoppingCart.ShoppingCartItems.Count == 0)
28     {
29         ModelState.AddModelError("", "Your cart is empty");
30     }
31
32     if (ModelState.IsValid)
33     {
34         _orderRepository.CreateOrder(order);
35         _shoppingCart.ClearCart();
36         return RedirectToAction("CheckoutComplete");
37     }
38
39     return View(order);
40 }
41
42 public IActionResult CheckoutComplete()
43 {
```



Adding Validation

Cosa accade se l'utente inserisce dati non corretti?

The Need for Validation



```
if (ModelState.IsValid)
{
    _orderRepository.CreateOrder(order);
    return RedirectToAction("CheckoutComplete");
}
else
{
    return View();
}
```

Validation

ModelState contains binding and validation errors

Solo quando il modello sarà valido salverà l'ordine

E' possibile eseguire una convalida del modello sull'intera istanza di un modello in una sola volta utilizzando ModelState.IsValid, GetValidationState accetta una singola proprietà ed infine AddModelError

ModelState Properties

IsValid

GetValidationState

AddModelError

Validation

Attributes on the model classes

Constraints, required, regex patterns...

Custom attributes

(Non verrà fatto in questo corso)

Validation Attributes



Required
StringLength
Range
RegularExpression
EmailAddress
Phone

More Validation Attributes



Introduced with .NET 8

- Length
- AllowedValues
- DeniedValues

Adding Validation Attributes

```
public class Order
{
    [Required(ErrorMessage = "Enter your first name")]
    [StringLength(50)]
    public string FirstName { get; set; }
}
```

```
[Length(3, 30)]
public required string Name { get; set; }

[DeniedValues("Broccoli", "Fish", "Sprouts")]
public string PieIngredient { get; set; }
```

New Data Validations

Added with .NET 8

```
<form asp-action="Checkout" method="post">
    <div asp-validation-summary="All" class="text-danger">
    </div>
</form>
```

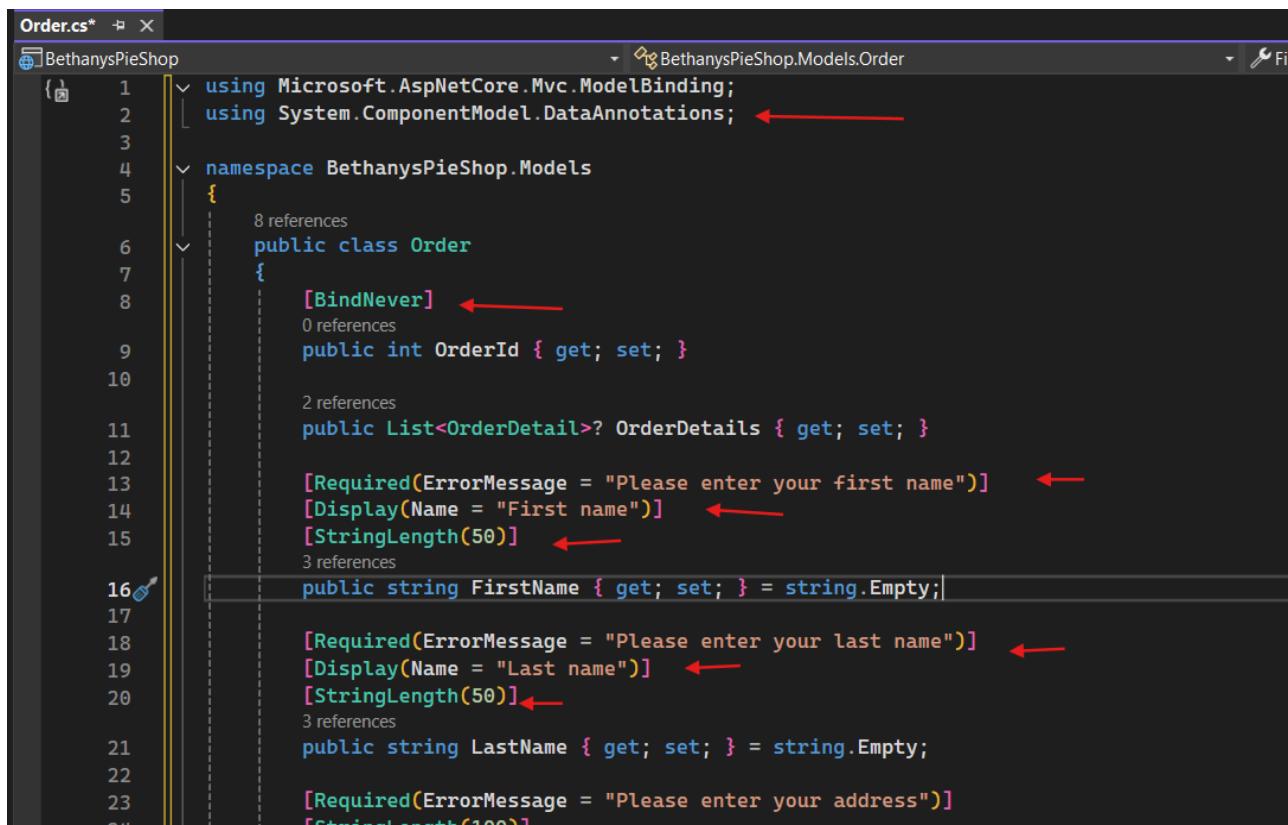
Displaying a Validation Summary

Osservazione: In questo caso la classe dietro helper tag che è la classe dell'helper tag validation-summary cercherà l'attributo asp-validation-summary su un div, eventuali errori rilevati nel metodo di azione verranno visualizzati nel riepilogo della convalida.

Possiamo vedere l'attributo asp-validation-summary posizionato su un div come valore specificato All, che mostrerà in modo abbastanza logico tutti gli errori di convalida che si sono verificati.

Demo: Adding Validation

Aggiungiamo una logica di convalida, quindi andiamo in Order.cs e aggiungiamo i vari attributi:



The screenshot shows the Order.cs file in the Visual Studio code editor. The file contains the following code:

```
Order.cs* ▾ X
BethanysPieShop
  1  using Microsoft.AspNetCore.Mvc.ModelBinding;
  2  using System.ComponentModel.DataAnnotations; ←
  3
  4  namespace BethanysPieShop.Models
  5  {
  6      public class Order
  7      {
  8          [BindNever] ←
  9          public int OrderId { get; set; }
 10
 11         public List<OrderDetail>? OrderDetails { get; set; }
 12
 13         [Required(ErrorMessage = "Please enter your first name")]
 14         [Display(Name = "First name")]
 15         [StringLength(50)] ←
 16         public string FirstName { get; set; } = string.Empty;
 17
 18         [Required(ErrorMessage = "Please enter your last name")]
 19         [Display(Name = "Last name")]
 20         [StringLength(50)] ←
 21         public string LastName { get; set; } = string.Empty;
 22
 23         [Required(ErrorMessage = "Please enter your address")]
 24         [StringLength(100)]
```

Annotations are highlighted with red arrows pointing to them from the left margin:

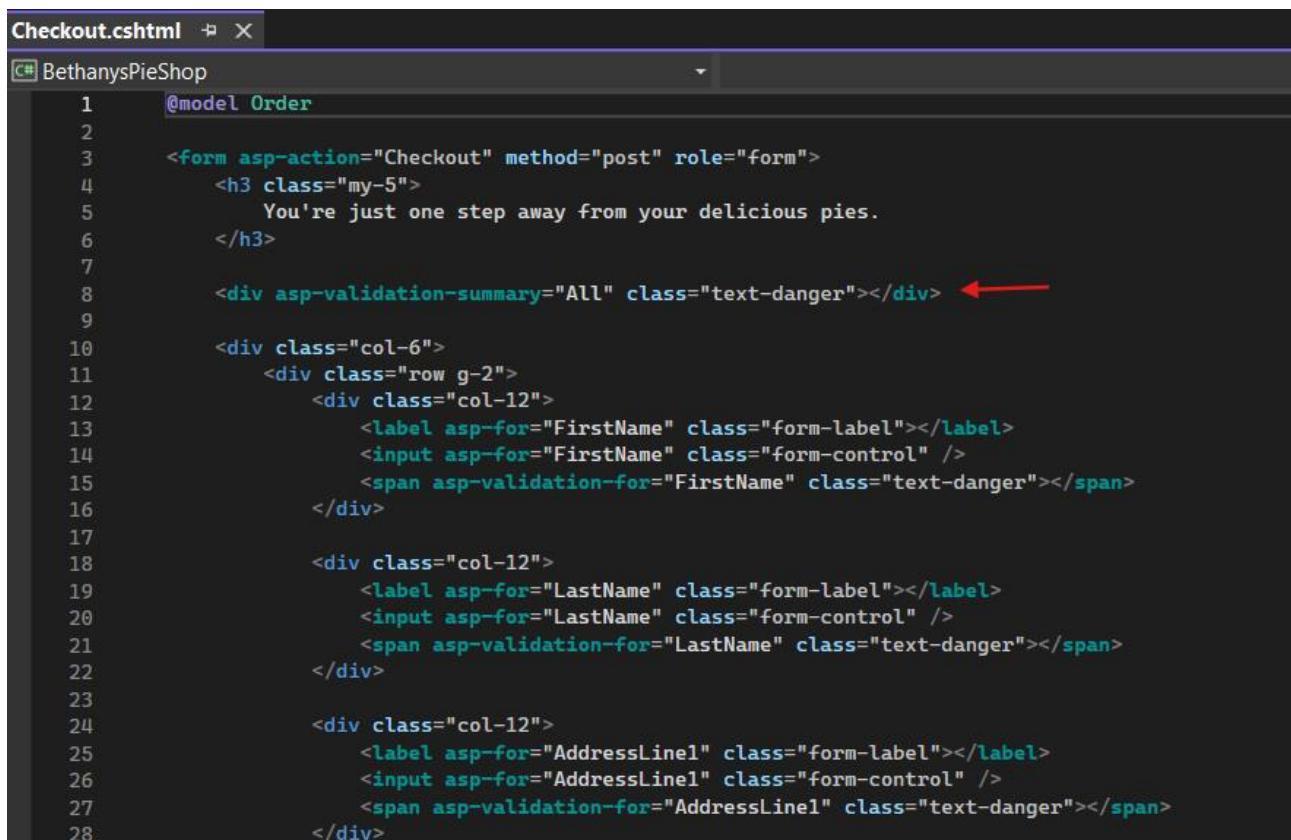
- A red arrow points to the `using System.ComponentModel.DataAnnotations;` line.
- A red arrow points to the `[BindNever]` attribute on line 8.
- A red arrow points to the `[Required(ErrorMessage = "Please enter your first name")]` annotation on line 13.
- A red arrow points to the `[Display(Name = "First name")]` annotation on line 14.
- A red arrow points to the `[StringLength(50)]` annotation on line 15.
- A red arrow points to the `[Required(ErrorMessage = "Please enter your last name")]` annotation on line 18.
- A red arrow points to the `[Display(Name = "Last name")]` annotation on line 19.
- A red arrow points to the `[StringLength(50)]` annotation on line 20.
- A red arrow points to the `[Required(ErrorMessage = "Please enter your address")]` annotation on line 23.
- A red arrow points to the `[StringLength(100)]` annotation on line 24.

[Home](#)

Regular expression applicato sulla mail:

```
[Required]
[StringLength(50)]
[DataType(DataType.EmailAddress)]
[RegularExpression(@"(?:[a-z0-9!#$%&'*+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+=?^_`{|}~-]+)*|""(?:[a-z0-9](?:[a-z0-9]*[a-z0-9])?\.){3}(?:25[0-5]|2[0-4][0-9]|01|[0-9][0-9]?){2}\.){3}(?:25[0-5]|2[0-4][0-9]|01|[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f])*""))@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.){3}(?:25[0-5]|2[0-4][0-9]|01|[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])",
    ErrorMessage = "The email address is not entered in a correct format")]
```

Andando in Checkout.cshtml



```
Checkout.cshtml ✘ X
C# BethanysPieShop
1 @model Order
2
3 <form asp-action="Checkout" method="post" role="form">
4     <h3 class="my-5">
5         You're just one step away from your delicious pies.
6     </h3>
7
8     <div asp-validation-summary="All" class="text-danger"></div> ←
9
10    <div class="col-6">
11        <div class="row g-2">
12            <div class="col-12">
13                <label asp-for="FirstName" class="form-label"></label>
14                <input asp-for="FirstName" class="form-control" />
15                <span asp-validation-for="FirstName" class="text-danger"></span>
16            </div>
17
18            <div class="col-12">
19                <label asp-for="LastName" class="form-label"></label>
20                <input asp-for="LastName" class="form-control" />
21                <span asp-validation-for="LastName" class="text-danger"></span>
22            </div>
23
24            <div class="col-12">
25                <label asp-for="AddressLine1" class="form-label"></label>
26                <input asp-for="AddressLine1" class="form-control" />
27                <span asp-validation-for="AddressLine1" class="text-danger"></span>
28            </div>
```

possiamo notare che a riga 8 abbiamo un helper tag che conterrà tutti gli errori di convalida che si verificano nel modulo.

Inoltre sono presenti span che mostrano l'errore specifico utilizzando la class="text-danger"

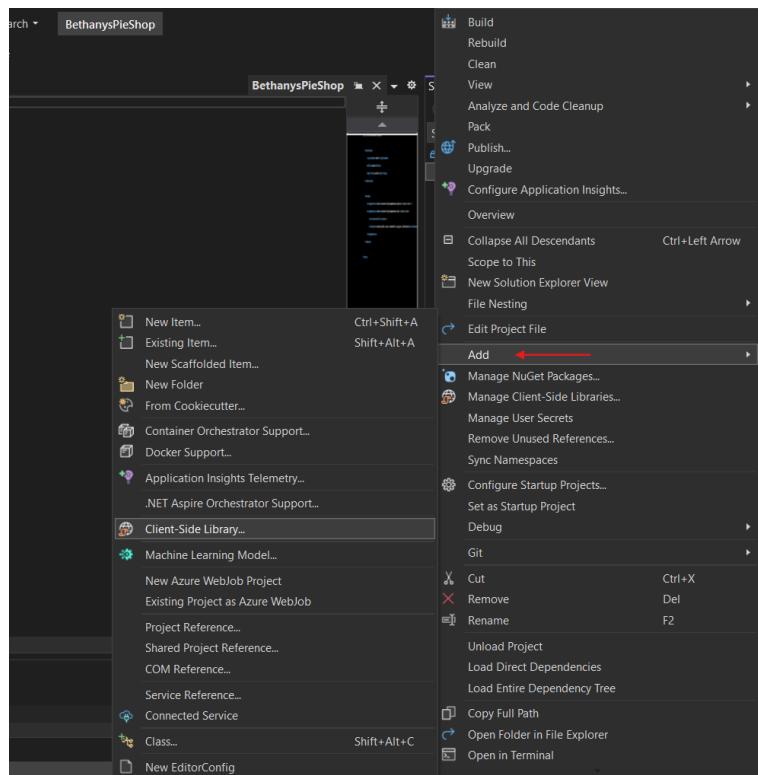
Difatti, se proviamo a cliccare su Complete Order del form ci compaiono errori di convalida che verranno visualizzati 2 volte.

Questi errori sono lato server, però possiamo farlo anche lato client che può offrire all'utente un'esperienza migliore, vediamo come farlo.

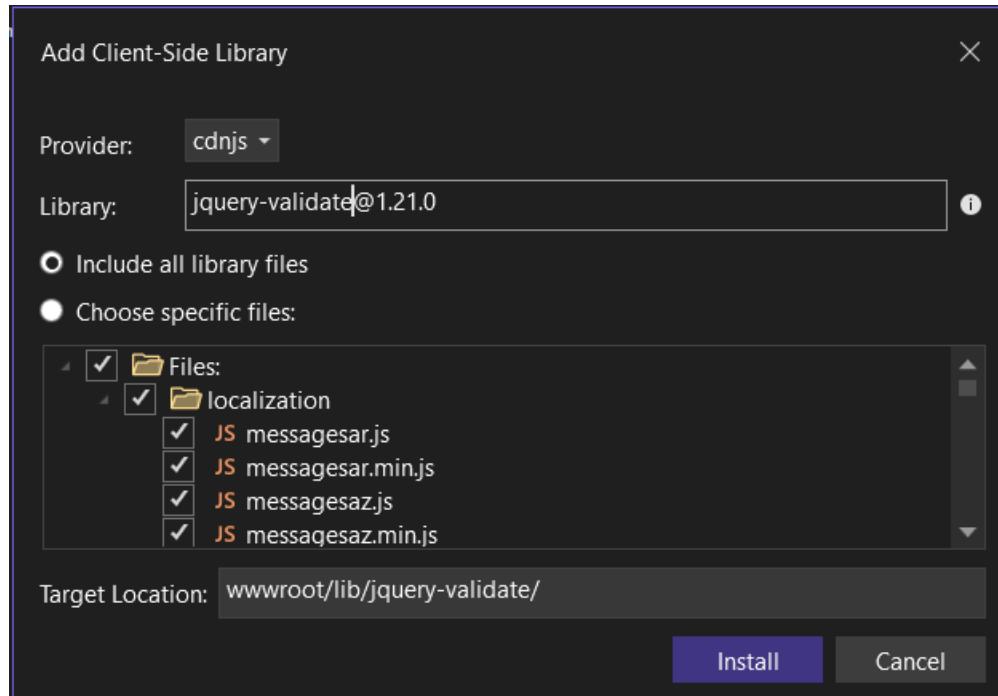
Demo: Adding Client-side Validation

Introduciamo ora la convalida Jquery.

Click destro sul nome del progetto > Aggiungiamo altri ClientSide Library:



Home



Ora in libman.json ho anche Jquery e aggiungiamo sotto una librerie di estensione/supporto a jquery-validate

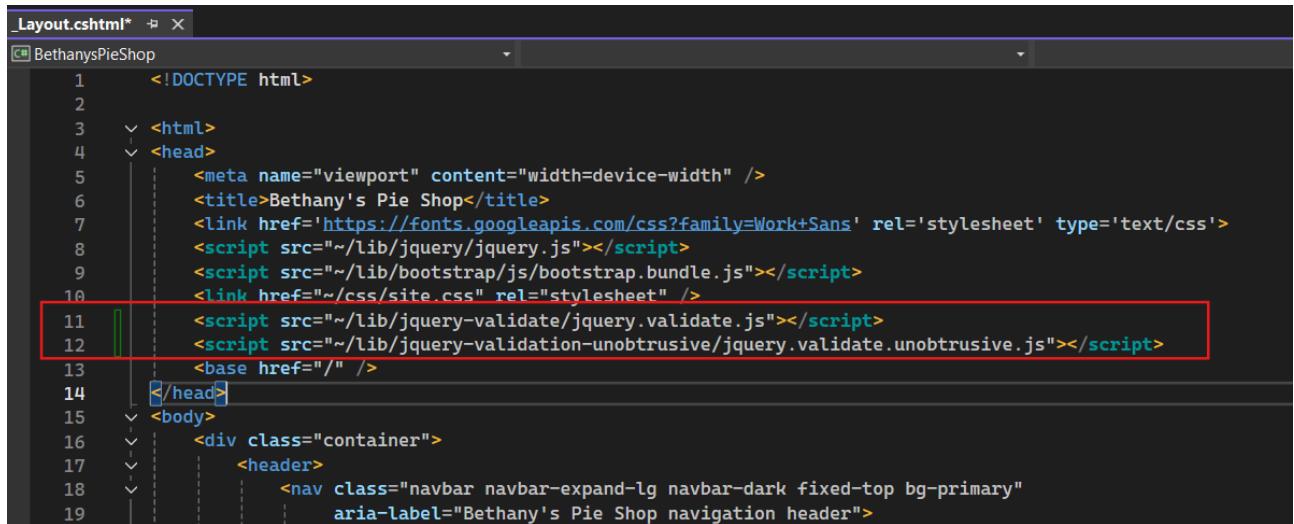
The screenshot shows the 'libman.json*' file in a code editor. The schema is defined as 'Schema: https://json.schemastore.org/libman.json'. The JSON content is as follows:

```
1  {
2    "version": "1.0",
3    "defaultProvider": "cdnjs",
4    "libraries": [
5      {
6        "library": "bootstrap@5.3.3",
7        "destination": "wwwroot/lib/bootstrap/"
8      },
9      {
10        "library": "jquery@3.7.1",
11        "destination": "wwwroot/lib/jquery/"
12      },
13      {
14        "library": "jquery-validate@1.21.0", ← Red arrow
15        "destination": "wwwroot/lib/jquery-validate/" ← Red arrow
16      },
17      {
18        "library": "jquery-validation-unobtrusive@3.2.12", ← Blue arrow
19        "destination": "wwwroot/lib/jquery-validation-unobtrusive/" ← Blue arrow
20      }
21    ]
22 }
```

Annotations: A red arrow points to the 'library' entry for 'jquery-validate@1.21.0'. Another red arrow points to the 'library' entry for 'jquery-validation-unobtrusive@3.2.12'. A blue arrow points to the 'library' entry for 'jquery-validation-unobtrusive@3.2.12'.

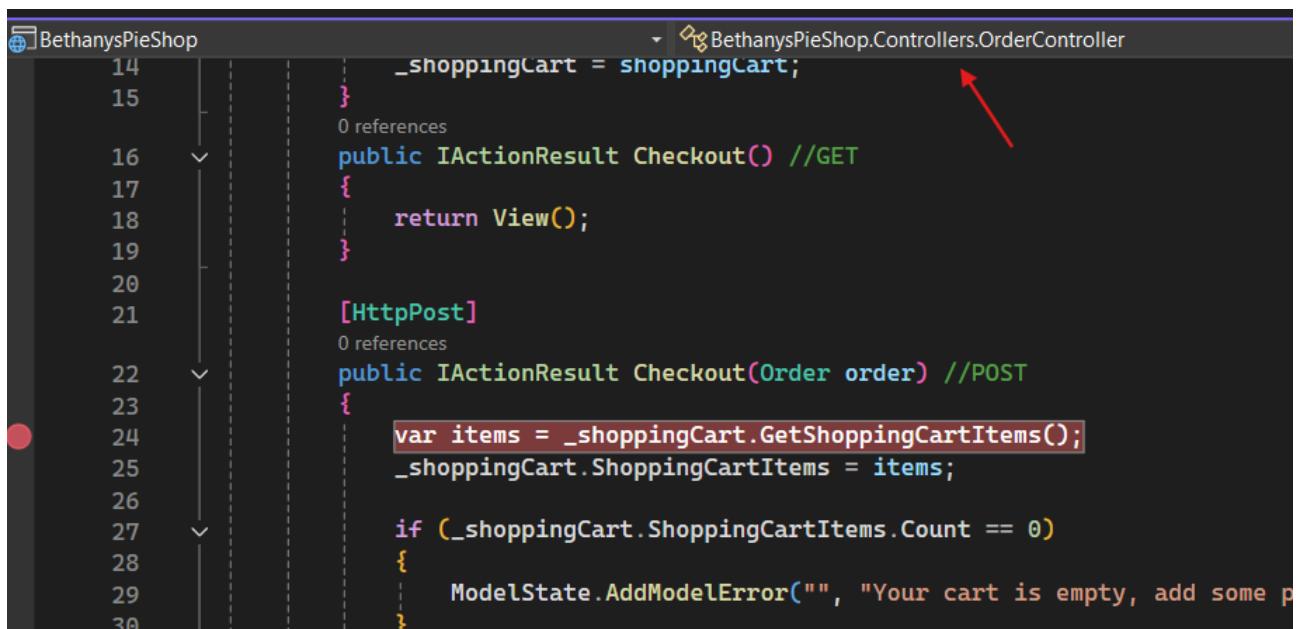
[Home](#)

Ora dobbiamo utilizzarle all'interno della nostra applicazione, quindi in Shared > _Layout.cshtml:



```
<!DOCTYPE html>
<html>
    <head>
        <meta name="viewport" content="width=device-width" />
        <title>Bethany's Pie Shop</title>
        <link href='https://fonts.googleapis.com/css?family=Work+Sans' rel='stylesheet' type='text/css'>
        <script src="~/lib/jquery/jquery.js"></script>
        <script src="~/lib/bootstrap/js/bootstrap.bundle.js"></script>
        <link href="/css/site.css" rel="stylesheet" />
        <script src="~/lib/jquery-validation/jquery.validate.js"></script>
        <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
        <base href="/" />
    </head>
    <body>
        <div class="container">
            <header>
                <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary" aria-label="Bethany's Pie Shop navigation header">
```

Salviamo ed effettuiamo un piccolo debug inserendo un breakpoint qui:



```
_shoppingCart = shoppingCart;
}
0 references
public IActionResult Checkout() //GET
{
    return View();
}

[HttpPost]
0 references
public IActionResult Checkout(Order order) //POST
{
    var items = _shoppingCart.GetShoppingCartItems();
    _shoppingCart.ShoppingCartItems = items;
    if (_shoppingCart.ShoppingCartItems.Count == 0)
    {
        ModelState.AddModelError("", "Your cart is empty, add some p
```

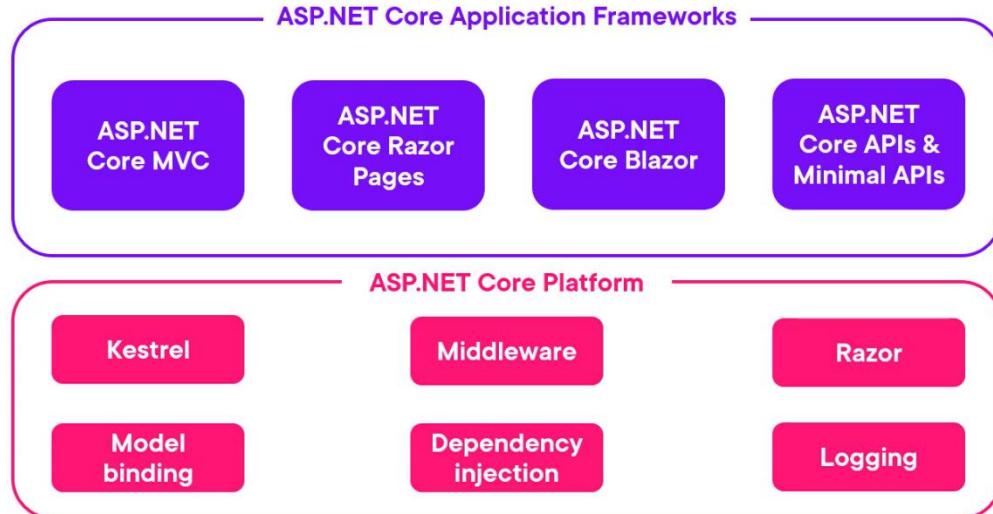
Ora i messaggi di errore saranno mostrati lato Client.

Recreating the Form Using a Razor Page

Finora abbiamo utilizzato ASP.NET Core MVC, quindi l'approccio basato su MVC, per creare le pagine del sito.

Adesso cambieremo quindi marcia e ricreeremo la pagina degli ordini ma utilizzando Razor Pages (un approccio differente che può essere anche combinato)

ASP.NET Core Application Frameworks



Introducing Razor Pages



Simpler form of creating pages



Based on PageModel and code-behind



Can co-exist in single application with MVC



Most concepts apply

Quindi cose come componenti di visualizzazione, helper

[Home](#)

```
builder.Services.AddRazorPages();  
app.MapRazorPages();
```

Configuring the Application for Razor Pages

Changes to Program.cs

Dopodichè creiamo una RazorPage:

```
@page  
  
<h1>Welcome to Bethany's Pie Shop</h1>  
<h2>The time is now @DateTime.Now</h2>
```

A First Razor Page

@page directive
Must be first directive

Osservazione: Non sono necessari Controiller o metodi di azione per eseguire questa pagina
Il file ha sempre una estensione cshtml.

Colleghiamo la pagina a un model:

```
@page
@model CheckoutPageModel

<p>
    @Model.Order.Total.ToString()
</p>
```

Using the @model Directive

Page model class

Links to class with same name as Page itself

@Model gives access to methods and properties

Una classe paragonabile a un file code-behind

Osservazione: Model scritto con la 'M' maiuscola.

```
public class CheckoutPageModel : PageModel
{
    public Order? Order { get; private set; }
```

The PageModel Class

Order defines a Total property

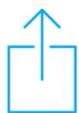
Page Handler Methods



Allows for separating of page and its data



OnGet()



OnPost()



Async versions exist too

```
public Order? Order { get; set; }

public IActionResult OnGet()
{
    Order = dbContext.Orders.First();
    return Page();
}
```

Using OnGet()

Most Concepts Are the Same

Dependency injection

Layout page

Partial views

Tag Helpers

View Components

Cosa è leggermente diverso:

Routing to Pages

WRONG WAY

Routing is simpler



Pages folder as root



Filename and location

Esempio:

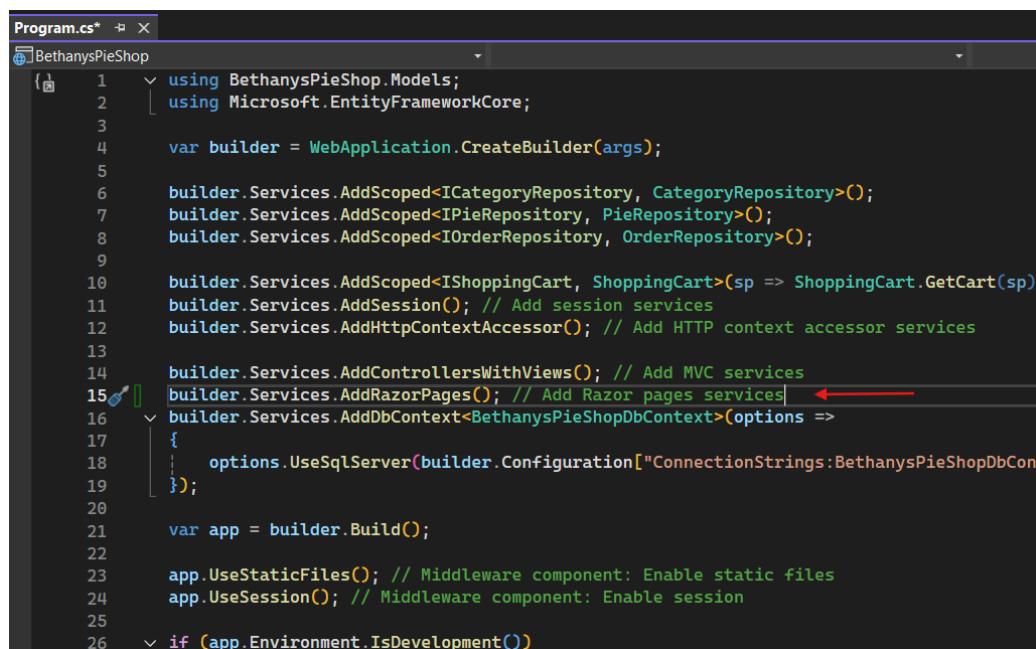
Routing to Razor Pages

Route	Page file
/	/Pages/index.cshtml
/Contact	/Pages/contact.cshtml
/Store/Pies	/Pages/Store/Pies.cshtml
/Store/Pies/3	/Pages/Store/Pies.cshtml @page "{id:int?}"

Demo: Recreating the Form Using Razor Pages

Per prima cosa abiliterò Razor Pages all'interno dell'applicazione.

Quindi andiamo in program.cs:



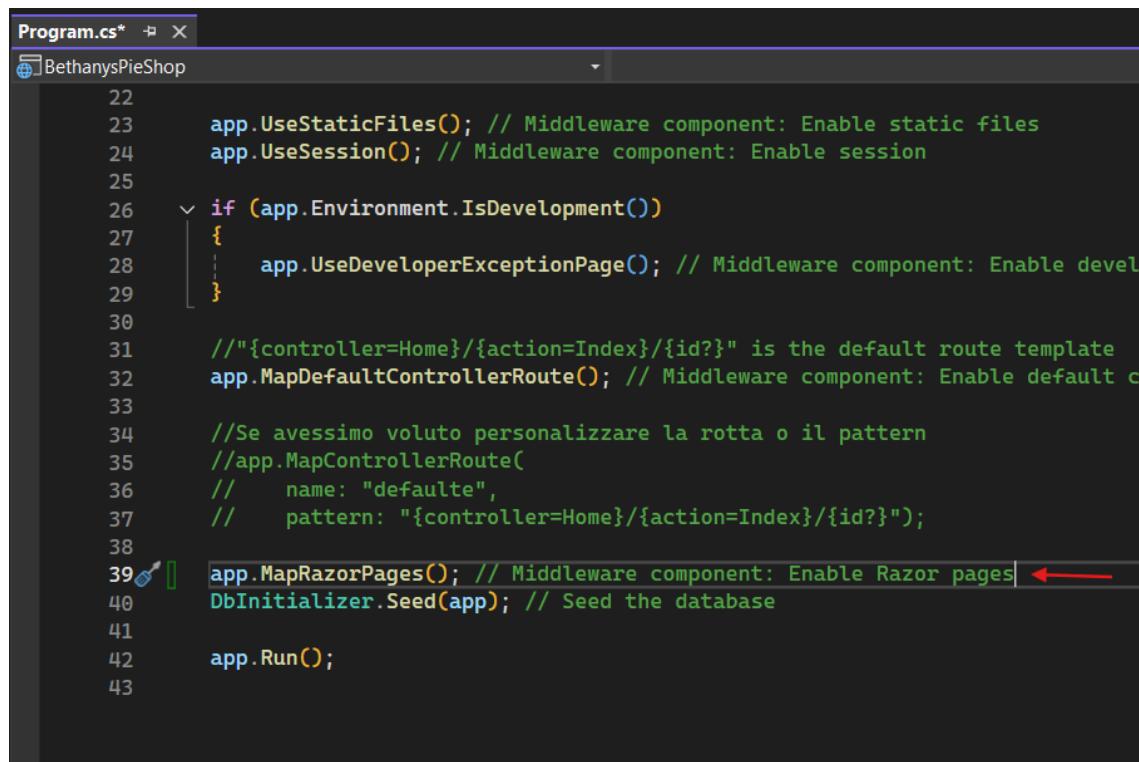
```
Program.cs*  ▾ X
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, PieRepository>();
8  builder.Services.AddScoped<IOrderRepository, OrderRepository>();
9
10 builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp));
11 builder.Services.AddSession(); // Add session services
12 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
13
14 builder.Services.AddControllersWithViews(); // Add MVC services
15 builder.Services.AddRazorPages(); // Add Razor pages services| -----^
16 builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
17 {
18     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContext"]);
19 });
20
21 var app = builder.Build();
22
23 app.UseStaticFiles(); // Middleware component: Enable static files
24 app.UseSession(); // Middleware component: Enable session
25
26 if (app.Environment.IsDevelopment())

```

Osservazione: l'ordine in cui aggiungiamo i servizi non ha importanza, nel Middleware si.

Aggiungiamo ora il middleware che permette di abilitare il modello Razor Page.:

Home

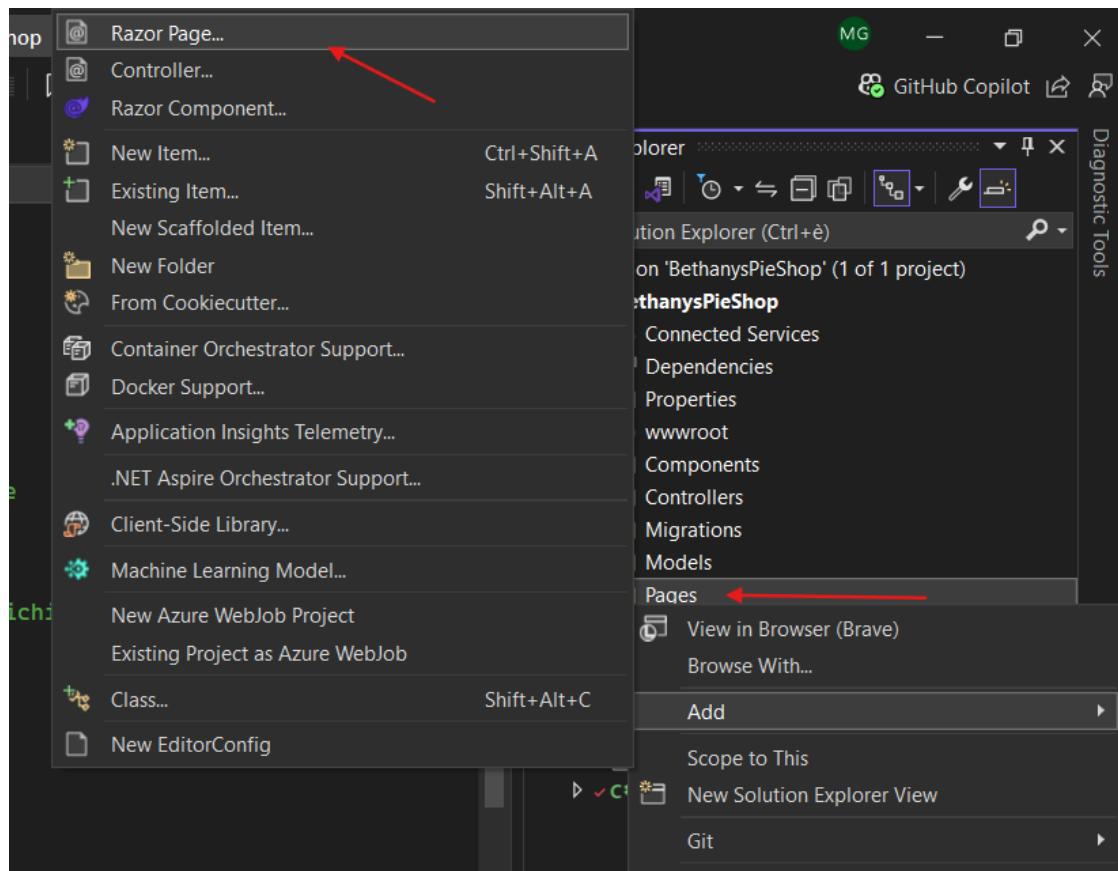


```
Program.cs* ✘ ×
BethanyPieShop

22
23     app.UseStaticFiles(); // Middleware component: Enable static files
24     app.UseSession(); // Middleware component: Enable session
25
26     if (app.Environment.IsDevelopment())
27     {
28         app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
29     }
30
31     //"{controller=Home}/{action=Index}/{id?}" is the default route template
32     app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route
33
34     //Se avessimo voluto personalizzare la rotta o il pattern
35     //app.MapControllerRoute(
36     //    name: "default",
37     //    pattern: "{controller=Home}/{action=Index}/{id?}");
38
39    app.MapRazorPages(); // Middleware component: Enable Razor pages ←
40    DbInitializer.Seed(app); // Seed the database
41
42    app.Run();
43
```

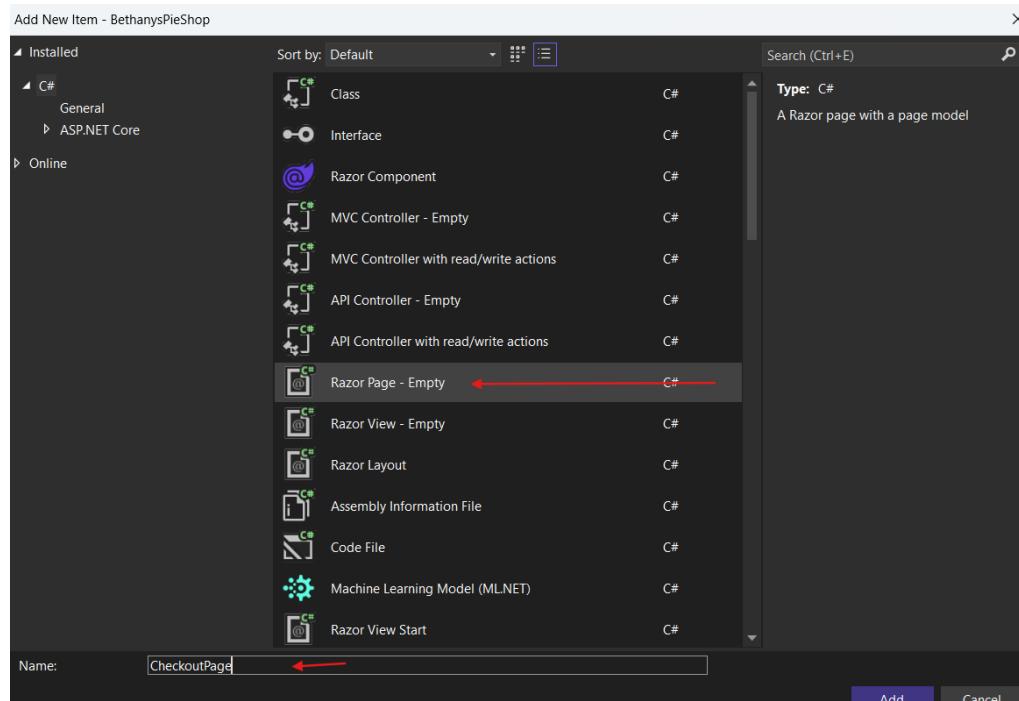
Ora, click destro sul progetto e creiamo una nuova cartella "Pages"

La cartella sarà a livello della root, fatto ciò creiamo una Razor pages al suo interno:

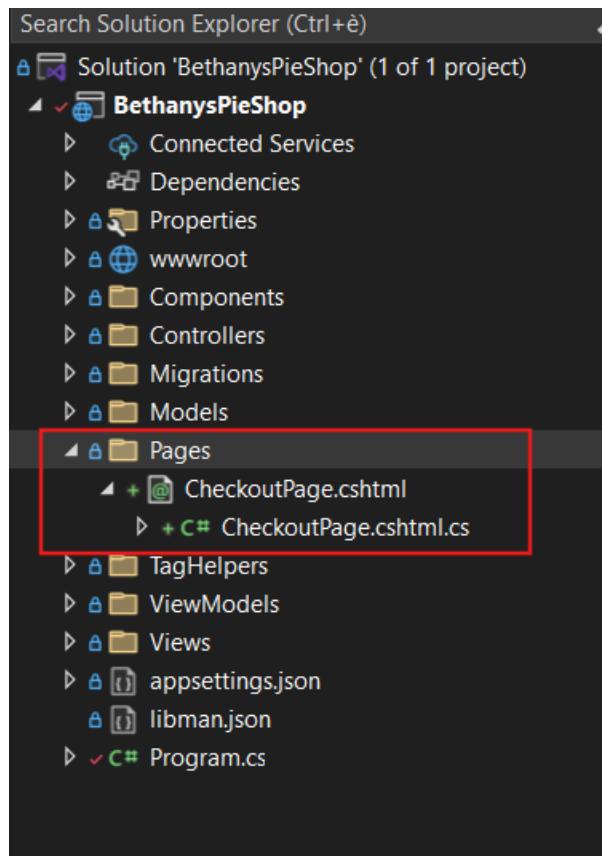


Home

Ci assicuriamo di scegliere una Razor Page - Empty e la chiamiamo CheckoutPage:



Home



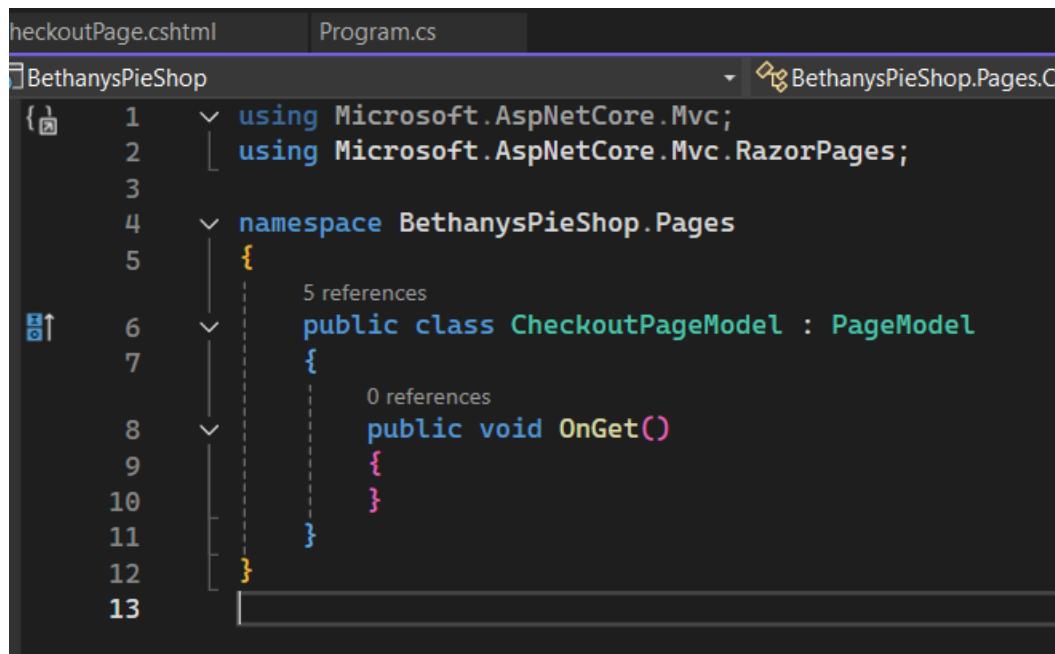
Osservazione: una Razor Page contiene un file .cshtml.cs "Code Behind"

```
CheckoutPage.cshtml  X Program.cs
1  @page
2  @model BethanysPieShop.Pages.CheckoutPageModel
3  @{
4      }
5
```

Osservazione: Notiamo che riga due punta esattamente al file citato nell'osservazione precedente

Apriamolo:

[Home](#)



```
heckoutPage.cshtml      Program.cs
BethanysPieShop          BethanysPieShop.Pages.C
{
    1     using Microsoft.AspNetCore.Mvc;
    2     using Microsoft.AspNetCore.Mvc.RazorPages;
    3
    4     namespace BethanysPieShop.Pages
    5     {
    6         public class CheckoutPageModel : PageModel
    7         {
    8             public void OnGet()
    9             {
    10            }
    11        }
    12    }
    13}
```

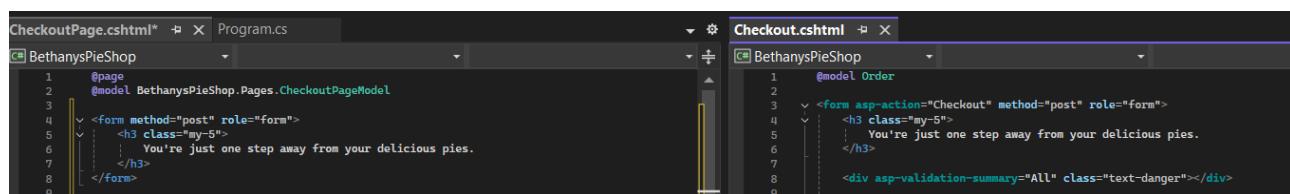
La convenzione vuole che ottiene lo stesso nome ma con suffisso .cs
Eredita da PageModel.

E' stato già generato OnGet()

Questo verrà eseguito quando viene ricevuta una richiesta get su questa pagina, ritornerà semplicemente la View.

Ora torniamo a **CheckoutPage.cshtml**:

Aggiungiamo una piccola parte, come sarebbe stato fatto nel Checkout di Views, immagine che mostra la differenza:



```
CheckoutPage.cshtml*  Program.cs
BethanysPieShop          BethanysPieShop.Pages.C
1  @page
2  @model BethanysPieShop.Pages.CheckoutPageModel
3
4  <form method="post" role="form">
5      <h3 class="my-5">
6          You're just one step away from your delicious pies.
7      </h3>
8  </form>
9

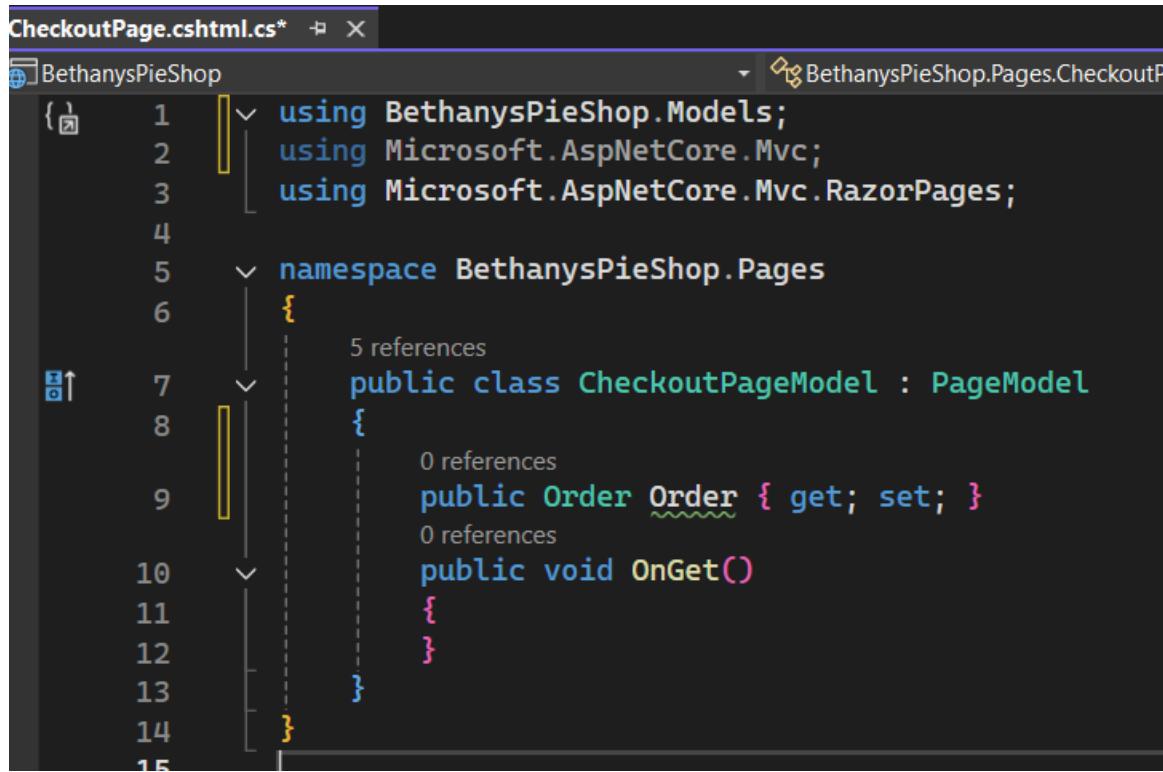
Checkout.cshtml*  Program.cs
BethanysPieShop          BethanysPieShop.Pages.C
1  @model Order
2
3  <form asp-action="Checkout" method="post" role="form">
4      <h3 class="my-5">
5          You're just one step away from your delicious pies.
6      </h3>
7
8      <div asp-validation-summary="All" class="text-danger"></div>
9
```

Possiamo notare che per impostazione predefinita. lo modulo verrà pubblicato sulla stessa pagina.

Non ho bisogno di specificare un azione, il modello della nostra pagina è CheckoutPageModel

Ora nel nostro CheckoutPageModel.cshtml.cs creiamo una proprietà di tipo Order:

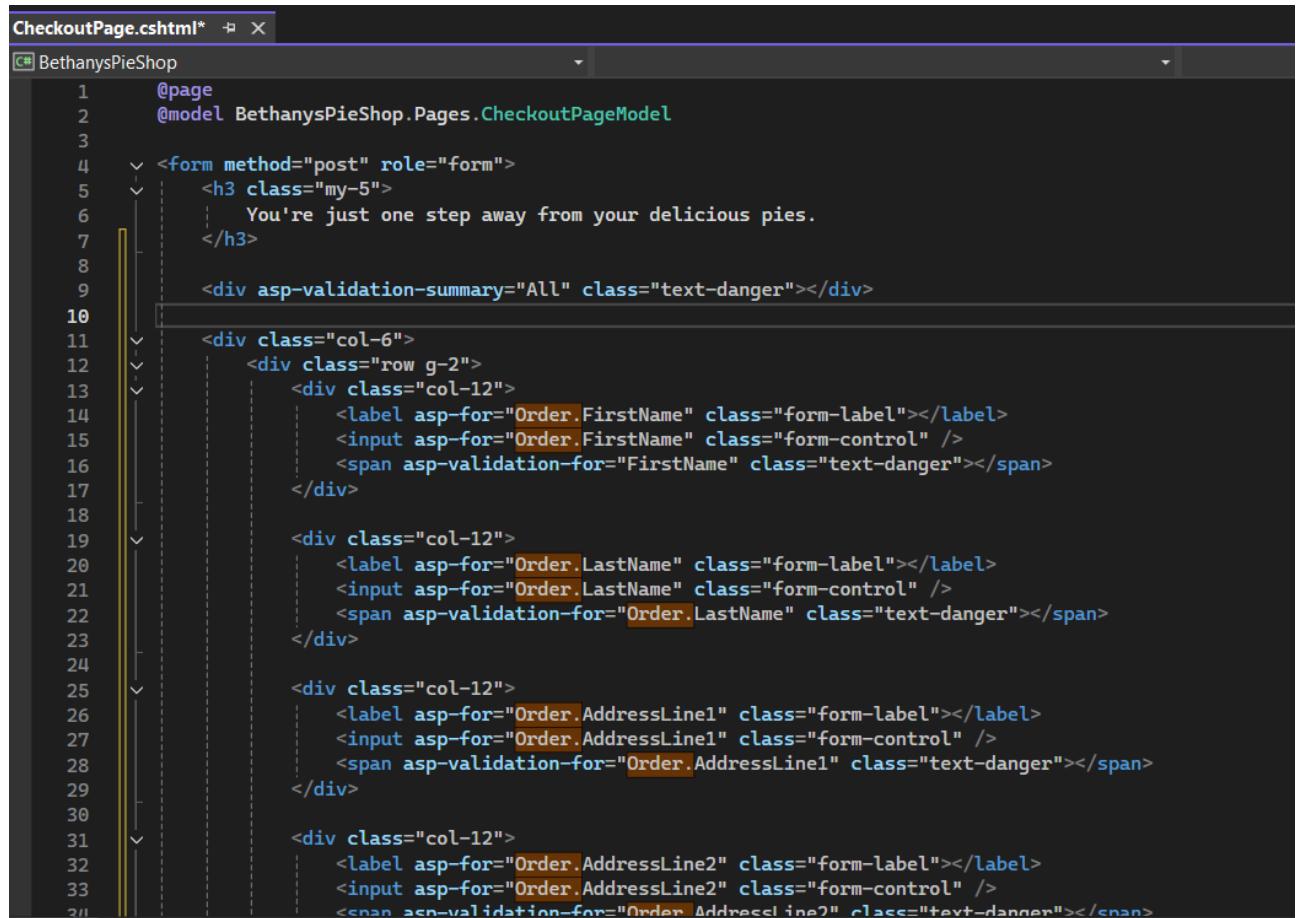
[Home](#)



```
CheckoutPage.cshtml.cs*  ✎ X
BethanysPieShop  ▾  BethanysPieShop.Pages.CheckoutP
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Mvc;
3  using Microsoft.AspNetCore.Mvc.RazorPages;
4
5  namespace BethanysPieShop.Pages
6  {
7      public class CheckoutPageModel : PageModel
8      {
9          public Order Order { get; set; }
10         public void OnGet()
11         {
12         }
13     }
14 }
15 }
```

Torniamo alla nostra CheckoutPage.cshtml e la completiamo, notiamo delle differenze rispetto alla vista create in View > Checkout:

Home



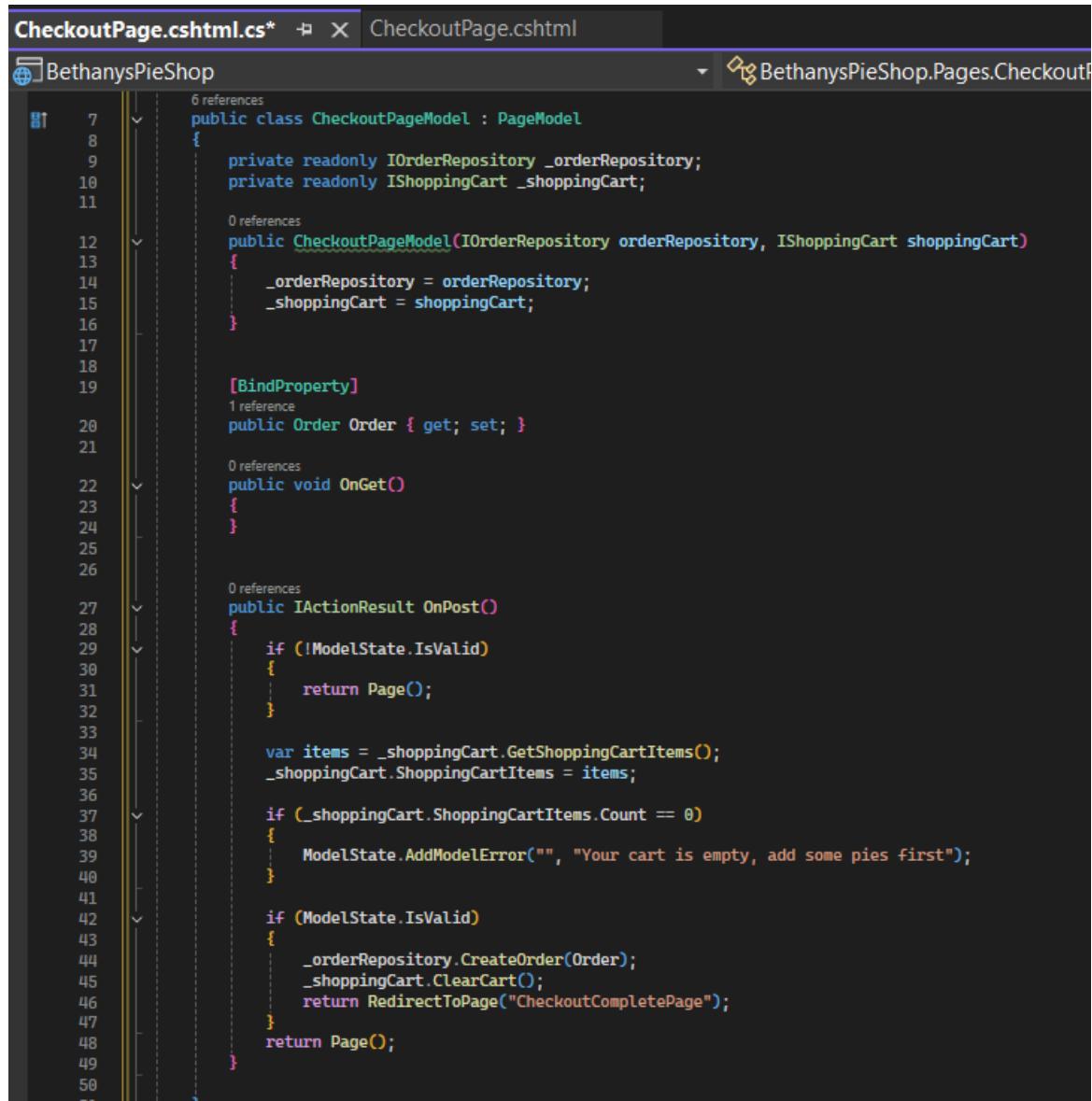
```
CheckoutPage.cshtml*  X
C# BethanyPieShop
1  @page
2  @model BethanyPieShop.Pages.CheckoutPageModel
3
4  <form method="post" role="form">
5      <h3 class="my-5">
6          You're just one step away from your delicious pies.
7      </h3>
8
9      <div asp-validation-summary="All" class="text-danger"></div>
10
11     <div class="col-6">
12         <div class="row g-2">
13             <div class="col-12">
14                 <label asp-for="Order.FirstName" class="form-label"></label>
15                 <input asp-for="Order.FirstName" class="form-control" />
16                 <span asp-validation-for="FirstName" class="text-danger"></span>
17             </div>
18
19             <div class="col-12">
20                 <label asp-for="Order.LastName" class="form-label"></label>
21                 <input asp-for="Order.LastName" class="form-control" />
22                 <span asp-validation-for="Order.LastName" class="text-danger"></span>
23             </div>
24
25             <div class="col-12">
26                 <label asp-for="Order.AddressLine1" class="form-label"></label>
27                 <input asp-for="Order.AddressLine1" class="form-control" />
28                 <span asp-validation-for="Order.AddressLine1" class="text-danger"></span>
29             </div>
30
31             <div class="col-12">
32                 <label asp-for="Order.AddressLine2" class="form-label"></label>
33                 <input asp-for="Order.AddressLine2" class="form-control" />
34                 <span asp-validation-for="Order.AddressLine2" class="text-danger"></span>
35             </div>
36
37         </div>
38     </div>
39
40     <div class="col-6">
41         <div class="row g-2">
42             <div class="col-12">
43                 <label asp-for="Order.PhoneNumber" class="form-label"></label>
44                 <input asp-for="Order.PhoneNumber" class="form-control" />
45                 <span asp-validation-for="Order.PhoneNumber" class="text-danger"></span>
46             </div>
47
48             <div class="col-12">
49                 <label asp-for="Order.Email" class="form-label"></label>
50                 <input asp-for="Order.Email" class="form-control" />
51                 <span asp-validation-for="Order.Email" class="text-danger"></span>
52             </div>
53
54         </div>
55     </div>
56
57     <div class="col-12">
58         <button type="submit" class="btn btn-primary w-100" value="Submit">Submit</button>
59     </div>
60
61 </form>
```

Osservazione: Specificamo "Order." in quanto è una proprietà su CheckoutPageModel

Capiamo così quanto siano utili le help tager e quanto il codice sia rimasto praticamente lo stesso.

Torniamo a CheckoutPageModel e lo completiamo:

Home



The screenshot shows the code editor for the `CheckoutPage.cshtml.cs` file. The code defines a `CheoutPageModel` class that implements `PageModel`. It has properties for `IOrderRepository` and `IShoppingCart`, and a constructor that takes these dependencies. The `OnGet()` method handles the GET request for the page. The `OnPost()` method handles the POST request, checking if the model state is valid. If it's not, it returns the current page. If it is, it creates an order using the `IOrderRepository`, clears the shopping cart, and then redirects to the `CheckoutCompletePage`.

```
public class CheckoutPageModel : PageModel
{
    private readonly IOrderRepository _orderRepository;
    private readonly IShoppingCart _shoppingCart;

    public CheckoutPageModel(IOrderRepository orderRepository, IShoppingCart shoppingCart)
    {
        _orderRepository = orderRepository;
        _shoppingCart = shoppingCart;
    }

    [BindProperty]
    public Order Order { get; set; }

    public void OnGet()
    {
    }

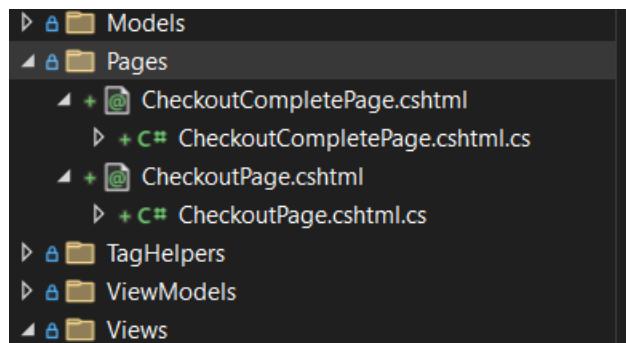
    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var items = _shoppingCart.GetShoppingCartItems();
        _shoppingCart.ShoppingCartItems = items;

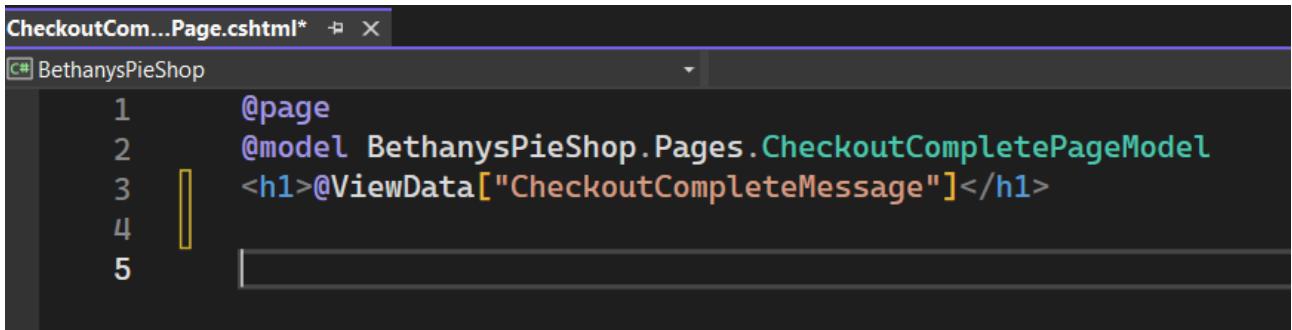
        if (_shoppingCart.ShoppingCartItems.Count == 0)
        {
            ModelState.AddModelError("", "Your cart is empty, add some pies first");
        }

        if (ModelState.IsValid)
        {
            _orderRepository.CreateOrder(Order);
            _shoppingCart.ClearCart();
            return RedirectToPage("CheckoutCompletePage");
        }
        return Page();
    }
}
```

Aggiungiamo ora una Razor Page `CheckoutCompletePage`:



[Home](#)

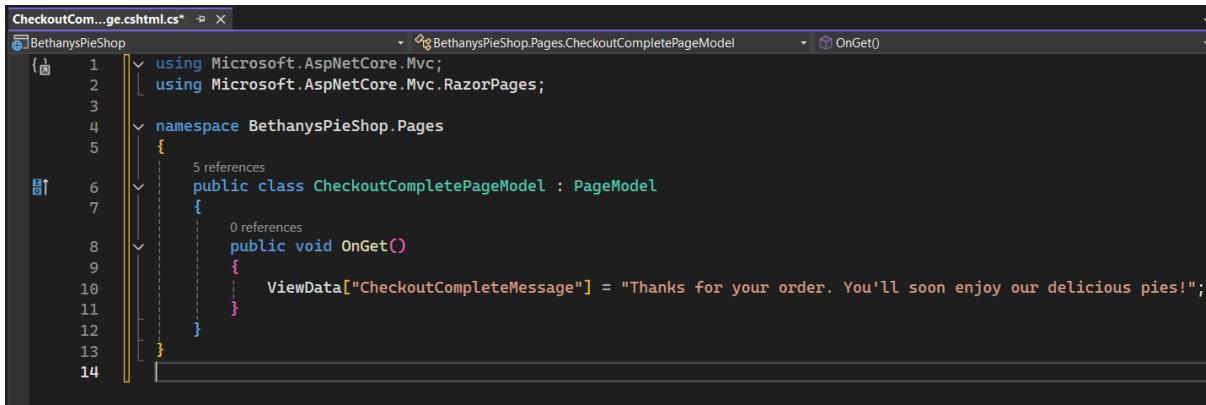


The screenshot shows a code editor window for a Razor page named "CheckoutCompletePage.cshtml". The code is as follows:

```
1 @page
2 @model BethanysPieShop.Pages.CheckoutCompletePageModel
3 <h1>@ViewData["CheckoutCompleteMessage"]</h1>
4
5 
```

Osservazione: al suo interno usiamo un ViewData Dizionario.

Il suo .cs sarà così:



The screenshot shows a code editor window for a C# class named "CheckoutCompletePageModel". The code is as follows:

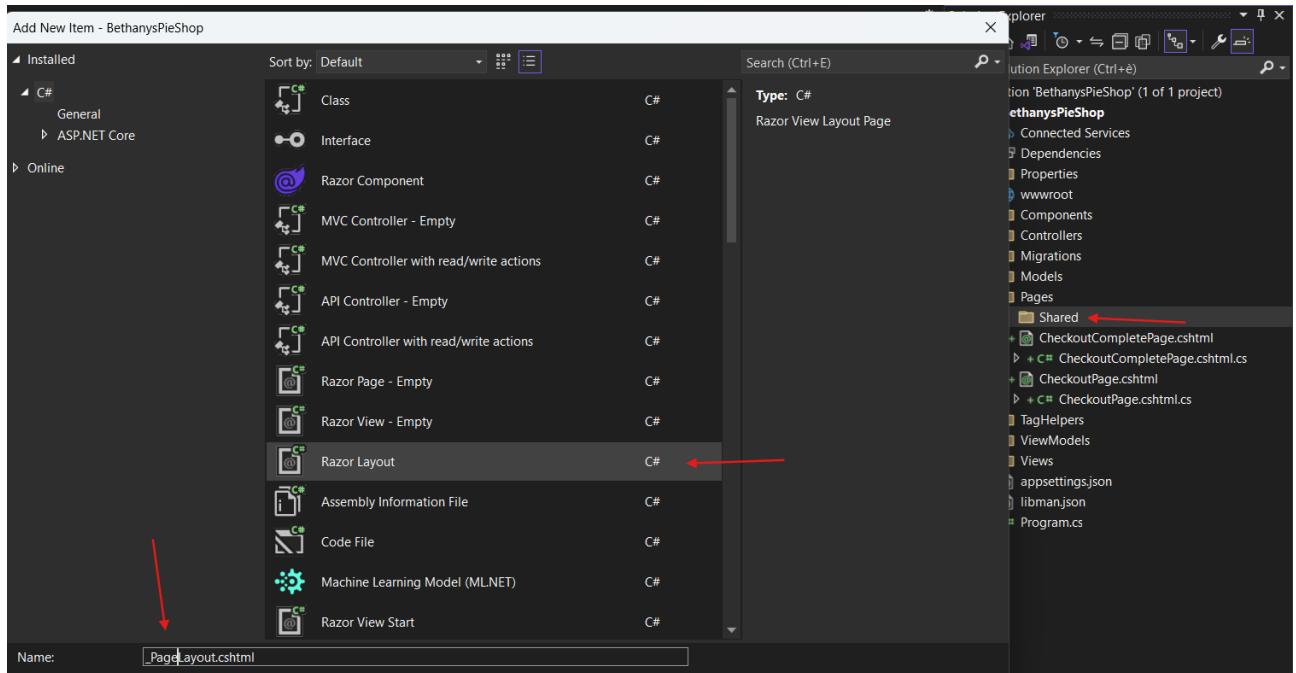
```
1 using Microsoft.AspNetCore.Mvc;
2 using Microsoft.AspNetCore.Mvc.RazorPages;
3
4 namespace BethanysPieShop.Pages
5 {
6     public class CheckoutCompletePageModel : PageModel
7     {
8         public void OnGet()
9         {
10             ViewData["CheckoutCompleteMessage"] = "Thanks for your order. You'll soon enjoy our delicious pies!";
11         }
12     }
13 }
14 
```

I tag helpers non sono riconosciuti da VS per le nostre Razor Pages, non ne è a conoscenza.

Però noi li abbiamo in `_ViewImports.cshtml`.

Quindi creiamo un Layout separato e quindi una nuova cartella condivisa

Home



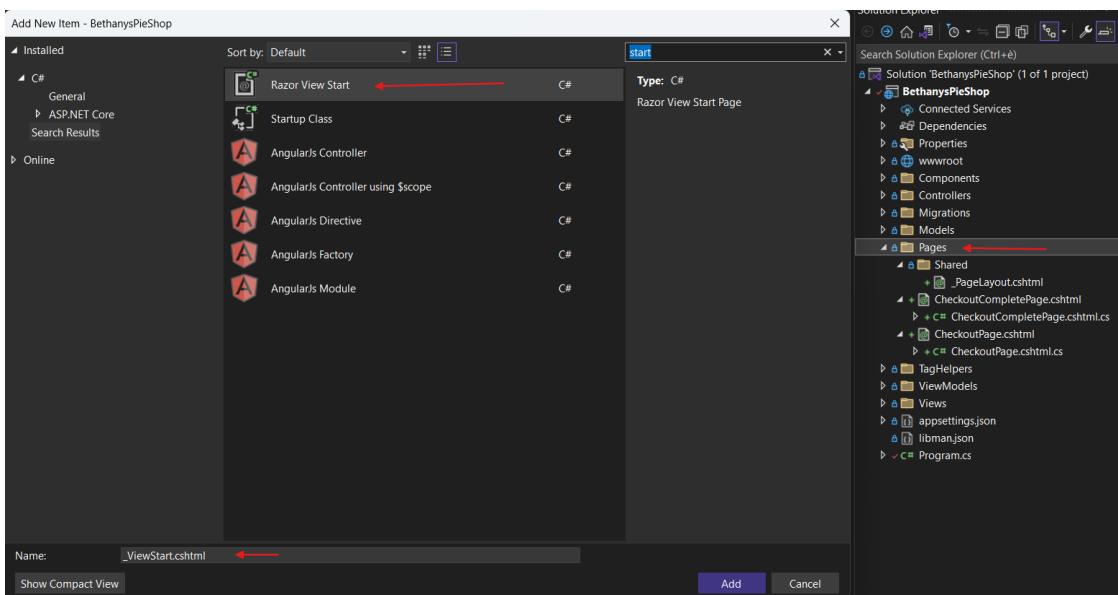
Inseriamo il _Layout di Views > Shared:

```
<!DOCTYPE html>
<html>
    <head>
        <meta name="viewport" content="width=device-width" />
        <title>Bethany's Pie Shop</title>
        <link href='https://fonts.googleapis.com/css?family=Work+Sans' rel='stylesheet' type='text/css'>
        <script src="~/lib/jquery/jquery.js"></script>
        <script src="~/lib/bootstrap/js/bootstrap.bundle.js"></script>
        <link href="~/css/site.css" rel="stylesheet" />
        <script src="~/lib/jquery-validation/jquery.validate.js"></script>
        <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
        <base href="/" />
    </head>
    <body>
        <div class="container">
            <header>
                <nav class="navbar navbar-expand-lg navbar-dark fixed-top bg-primary" aria-label="Bethany's Pie Shop navigation header">
                    <div class="container-xl">
                        <a class="navbar-brand" asp-controller="Home" asp-action="Index">
                            
                        </a>
                        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse" aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
                            <span class="navbar-toggler-icon"></span>
                        </button>
                        <div class="collapse navbar-collapse" id="navbarCollapse">
                            <ul class="navbar-nav me-auto mb-2 mb-lg-0">
```

Le pagine devono sapere di questo _PageLayout.cshtml quindi creiamo nella Root che nel nostro caso è Pages altri files:

_ViewStart:

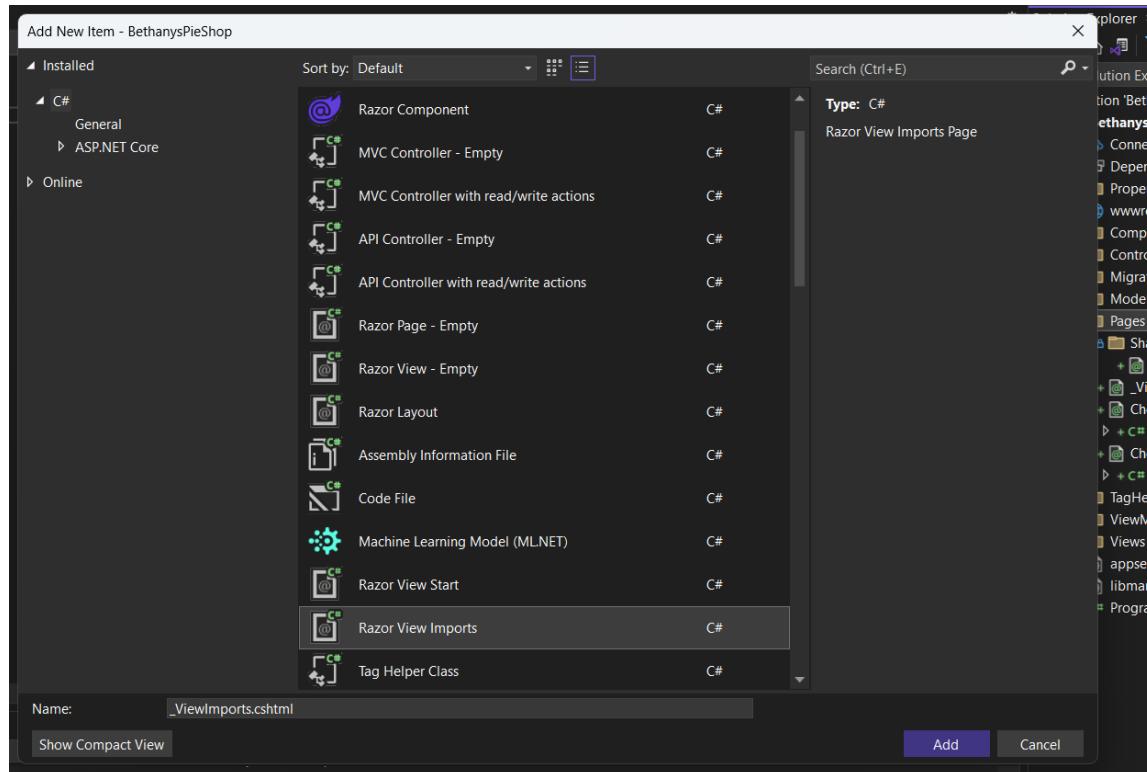
Home



```
_ViewStart.cshtml*  ⇧ X
C# BethanysPieShop
1  ↴ @{
2    | Layout = "_PageLayout";
3  }
4
```

Ora inseriamo un _ViewImports:

Home



```
_ViewImports.cshtml
```

```
C# BethanyPieShop
```

```
1 @using BethanyPieShop
2 @using BethanyPieShop.Models
3 @using BethanyPieShop.ViewModels
4 @using BethanyPieShop.Components
5 @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
6 @addTagHelper *, BethanyPieShop
7 @addTagHelper BethanyPieShop.TagHelpers.*, BethanyPieShop
```

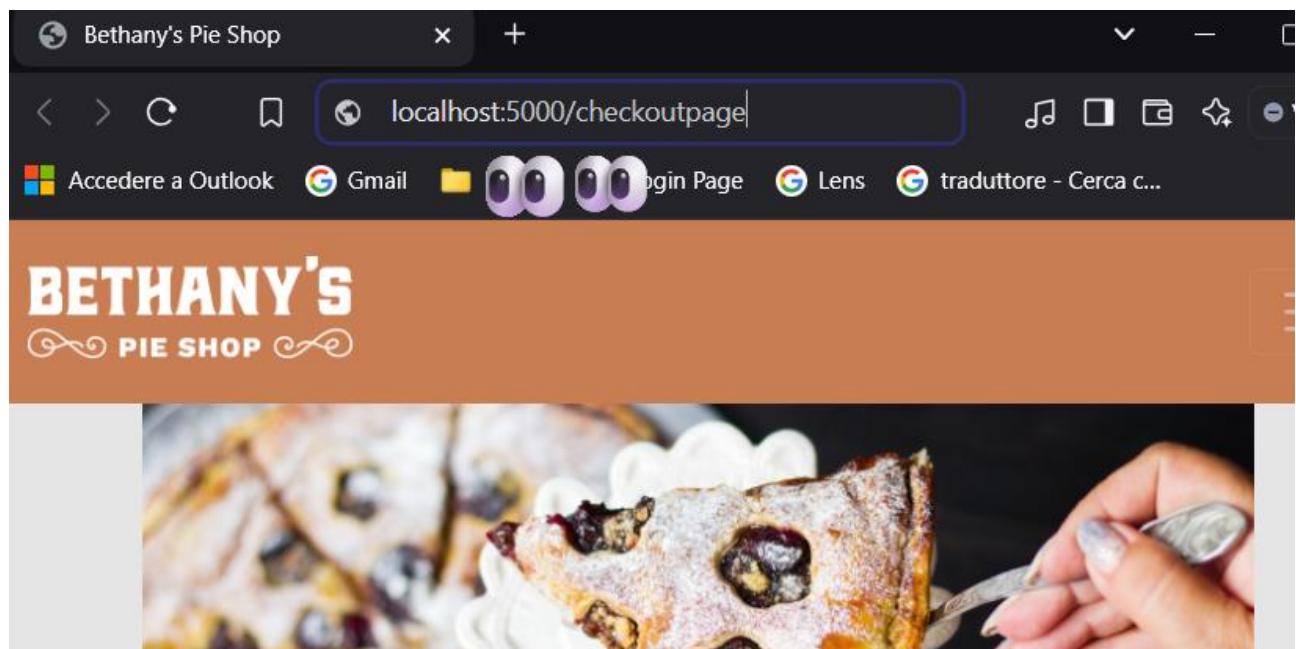
Anche qui abbiamo copiato dall'altro import che avevamo.

Ora buildiamo e dopo averlo fatto noteremo che anche questi file riconoscono i tag helpers:

[Home](#)

```
CheckoutPage.cshtml* ✎ X
C# BethanyPieShop
1  @page
2  @model BethanyPieShop.Pages.CheckoutPageModel
3
4  <form method="post" role="form">
5      <h3 class="my-5">
6          You're just one step away from your delicious pies.
7      </h3>
8
9      <div asp-validation-summary="All" class="text-danger"></div>
10
11     <div class="col-6">
12         <div class="row g-2">
13             <div class="col-12">
14                 <label asp-for="Order.FirstName" class="form-label"></label>
15                 <input asp-for="Order.FirstName" class="form-control" />
16                 <span asp-validation-for="Order.FirstName" class="text-danger"></span>
17             </div>
18
19             <div class="col-12">
20                 <label asp-for="Order.LastName" class="form-label"></label>
21                 <input asp-for="Order.LastName" class="form-control" />
22                 <span asp-validation-for="Order.LastName" class="text-danger"></span>
23             </div>
24
25         <div class="col-12">
26             <label asp-for="Order.AddressLine1" class="form-label"></label>
```

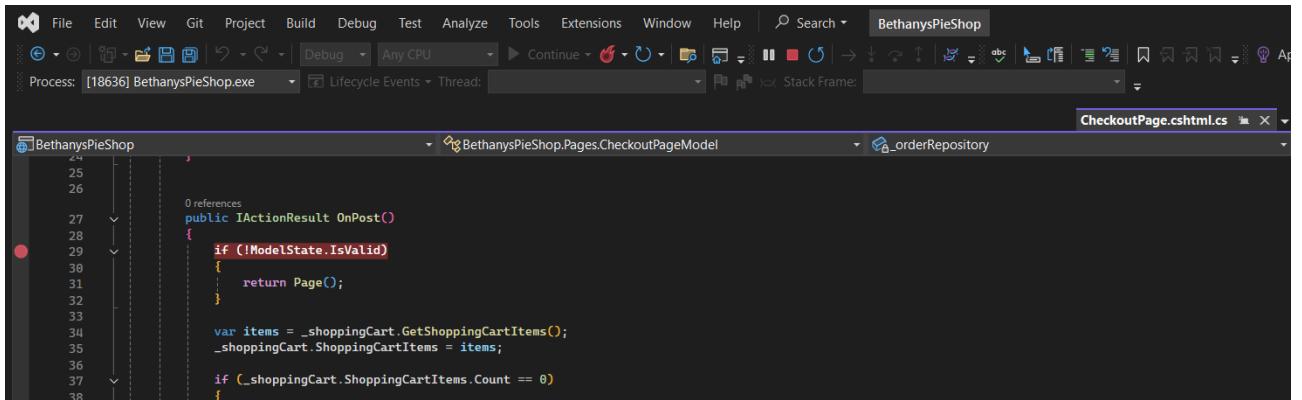
Ora eseguiamo:



Osservazione: Ci basta semplicemente inserire checkoutpage per giungere alla pagina di checkout senza specificare la cartella Pages, questa è l'impostazione predefinita, questa è la radice.

Se inseriamo un breakpoint qui:

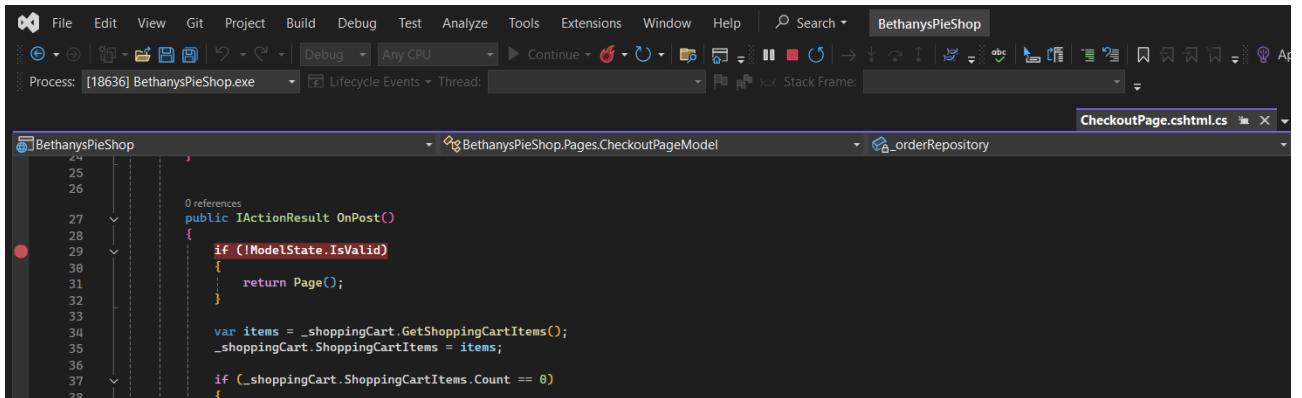
Home



```
24
25
26
27 public IActionResult OnPost()
28 {
29     if (!ModelState.IsValid)
30     {
31         return Page();
32     }
33
34     var items = _shoppingCart.GetShoppingCartItems();
35     _shoppingCart.ShoppingCartItems = items;
36
37     if (_shoppingCart.ShoppingCartItems.Count == 0)
38     {
39 
```

Ed inseriamo i dati di compilazione ed inviamo, noteremo che sembrerà nullo, questo perché non è stato implementato il model binding.

Ci basterà aggiungere l'attributo:



```
24
25
26
27 public IActionResult OnPost()
28 {
29     if (!ModelState.IsValid)
30     {
31         return Page();
32     }
33
34     var items = _shoppingCart.GetShoppingCartItems();
35     _shoppingCart.ShoppingCartItems = items;
36
37     if (_shoppingCart.ShoppingCartItems.Count == 0)
38     {
39 
```

Una volta fatto possiamo vedere le associazioni:

[Home](#)

}

[BindProperty]

31 references

```
public Order Order { get; set; }
```

0 references

```
public void OnGet()
```

{

}

0 references

```
public IActionResult
```

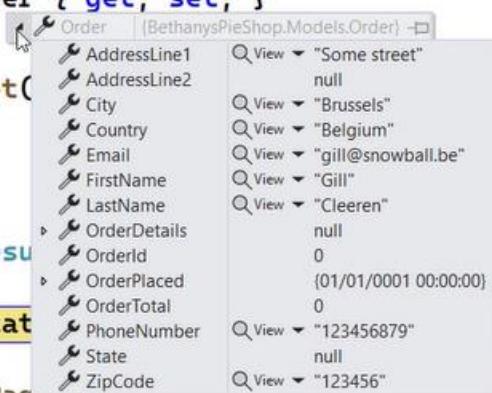
{

```
if (!ModelState
```

{

```
    return Page<
```

}

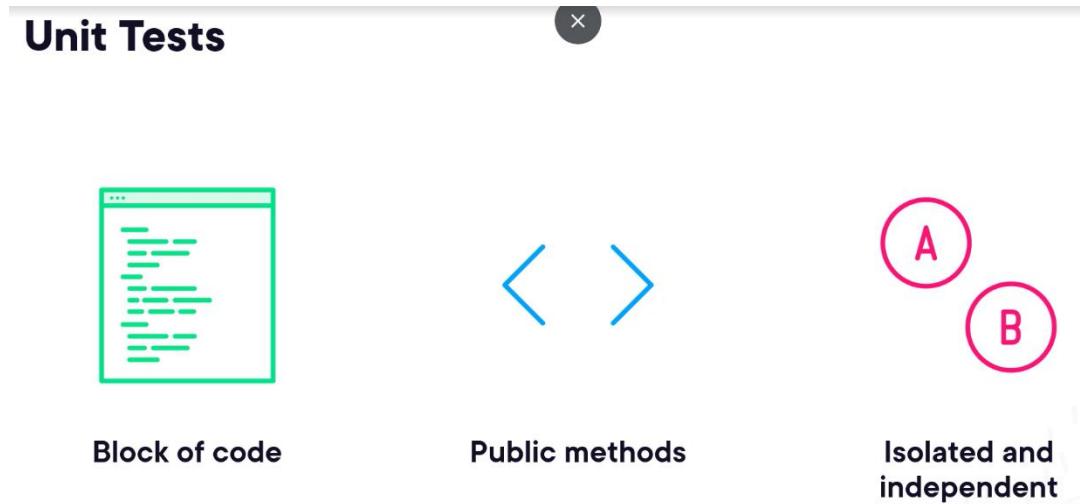


Questo è tutto sull'approccio con Razor Pages.

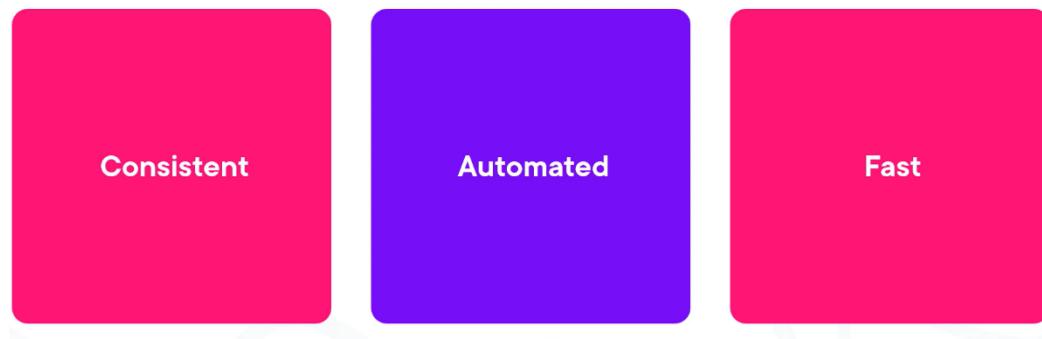
Testing the Application Components

- Understanding Unit Tests
- Writing Unit Tests
- Demo: Testing the Controllers
- Demo: Testing a Tag Helper

A unit test is an automated piece of code that invokes a unit of work in the system and then checks single assumption about behavior of that unit of work



Using Unit Tests



Why Do We Need Unit Tests?



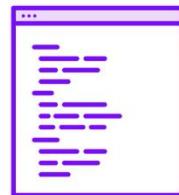
Find bugs



Change without
fear of breaking
something



Improve quality



Documentation
of code

Writing Unit Tests

- Arrange: è il blocco che si occupa della configurazione/set-up
- Act: esegui il blocco
- Assert: per verificare il risultato del codice di prova, dove specifichiamo i risultati attesi.

Used Framework

- xUnit -> Permette di creare test unitari su classi Mock
- Moq

Esempio:

A Simple Unit Test

```
public void CanUpdatePiePrice()
{
    // Arrange
    var pie =
        new Pie { Name = "Sample pie", Price = 12.95M };

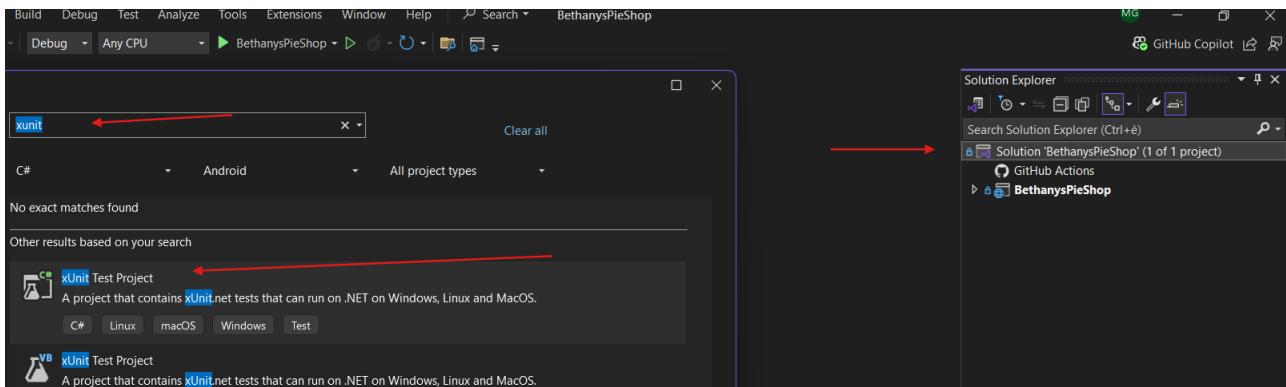
    // Act
    pie.Price = 20M;

    //Assert
    Assert.Equal(20M, pie.Price);
}
```

Demo: Testing the Controllers

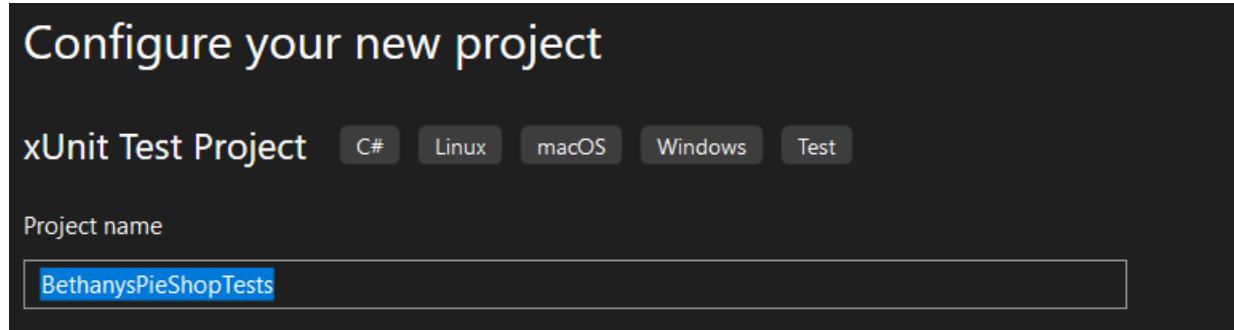
Aggiungiamo un progetto di test unitario alla nostra soluzione.

Click destro sulla soluzione > New project > digitiamo xUnit Test Project

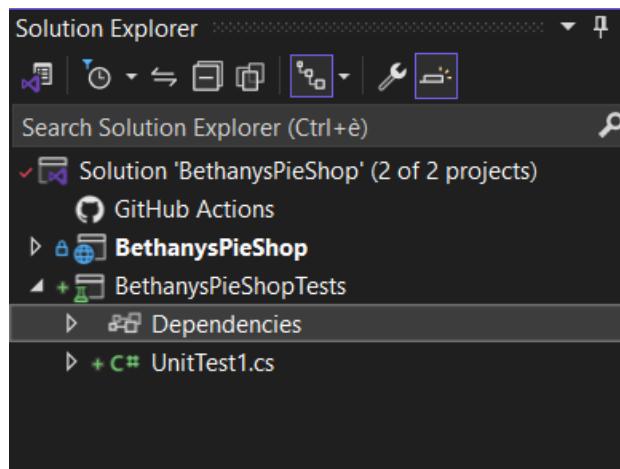


Lo chiamiamo nel seguente modo:

[Home](#)



Dopo aver scelto Dot Net 8, di default avremo ciò:

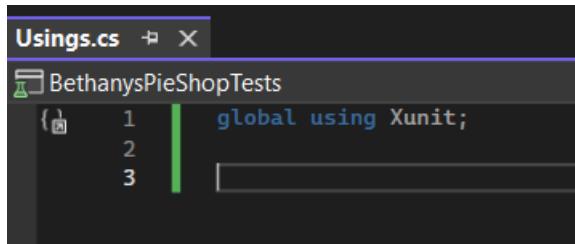


```
1  namespace BethanysPieShopTests
2  {
3      public class UnitTest1
4      {
5          [Fact]
6          public void Test1()
7          {
8          }
9      }
10 }
11 }
```

The screenshot shows the code editor with the file 'UnitTest1.cs' open. The code defines a class 'UnitTest1' with a single test method 'Test1' annotated with the '[Fact]' attribute. The code is numbered from 1 to 11.

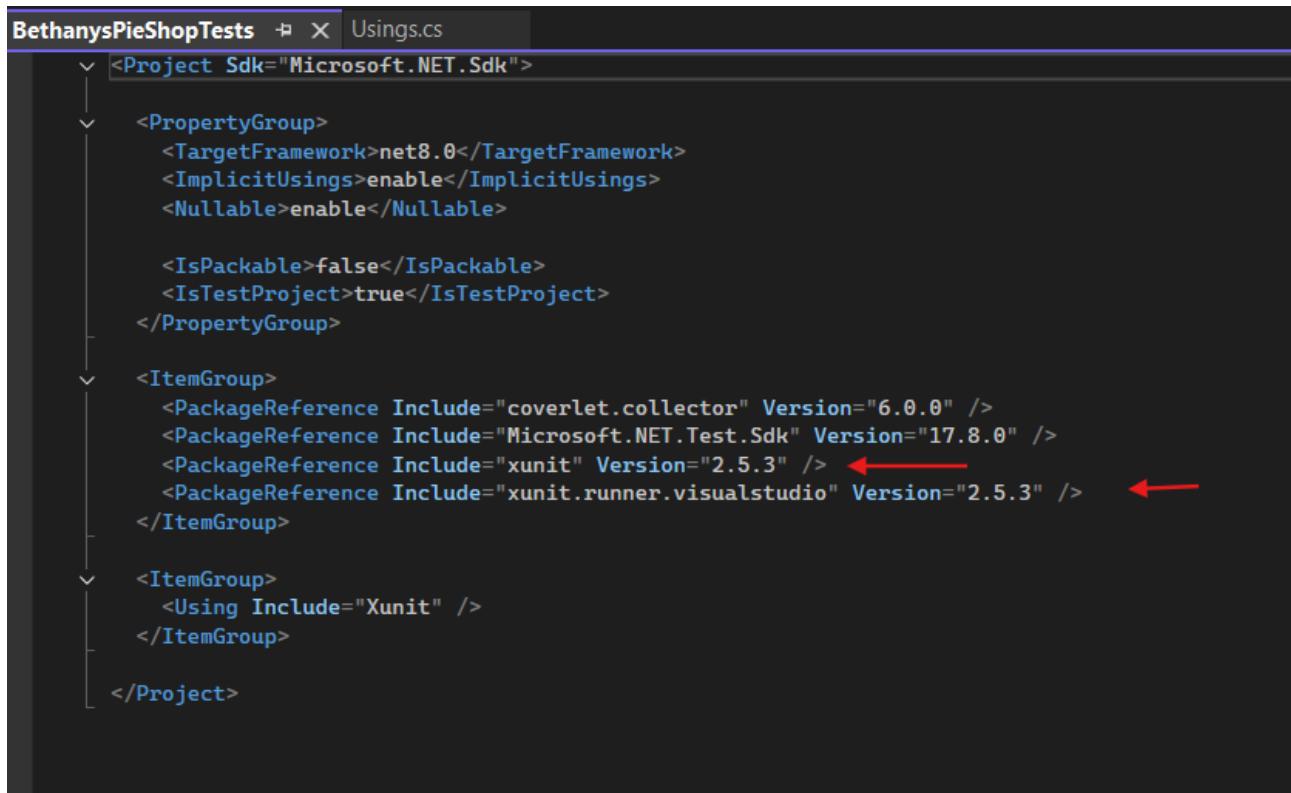
Cancelliamo questa classe e creiamo Usings.cs:

[Home](#)



```
Usings.cs ✎ X
BethanysPieShopTests
1 global using Xunit;
2
3
```

Possiamo notare che nel proj del test abbiamo i riferimenti a xunit:



```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>

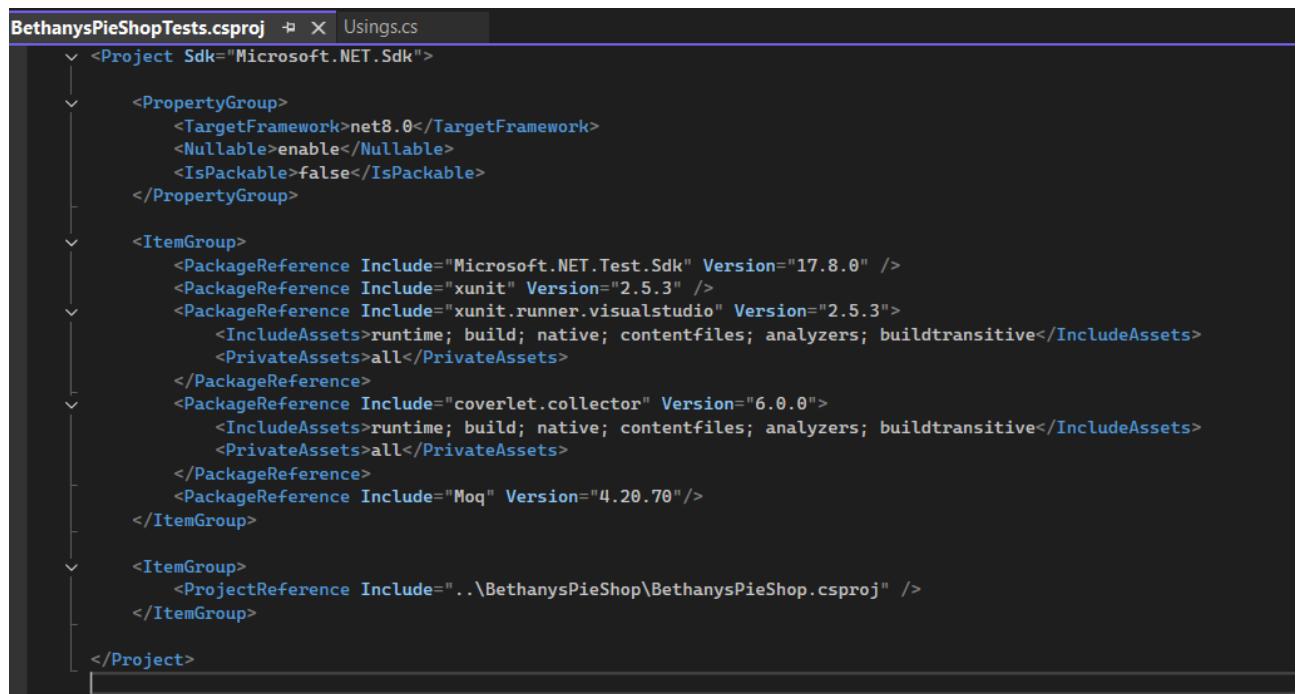
    <IsPackable>false</IsPackable>
    <IsTestProject>true</IsTestProject>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="coverlet.collector" Version="6.0.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
    <PackageReference Include="xunit" Version="2.5.3" /> ←
    <PackageReference Include="xunit.runner.visualstudio" Version="2.5.3" /> ←
  </ItemGroup>

  <ItemGroup>
    <Using Include="Xunit" />
  </ItemGroup>
</Project>
```

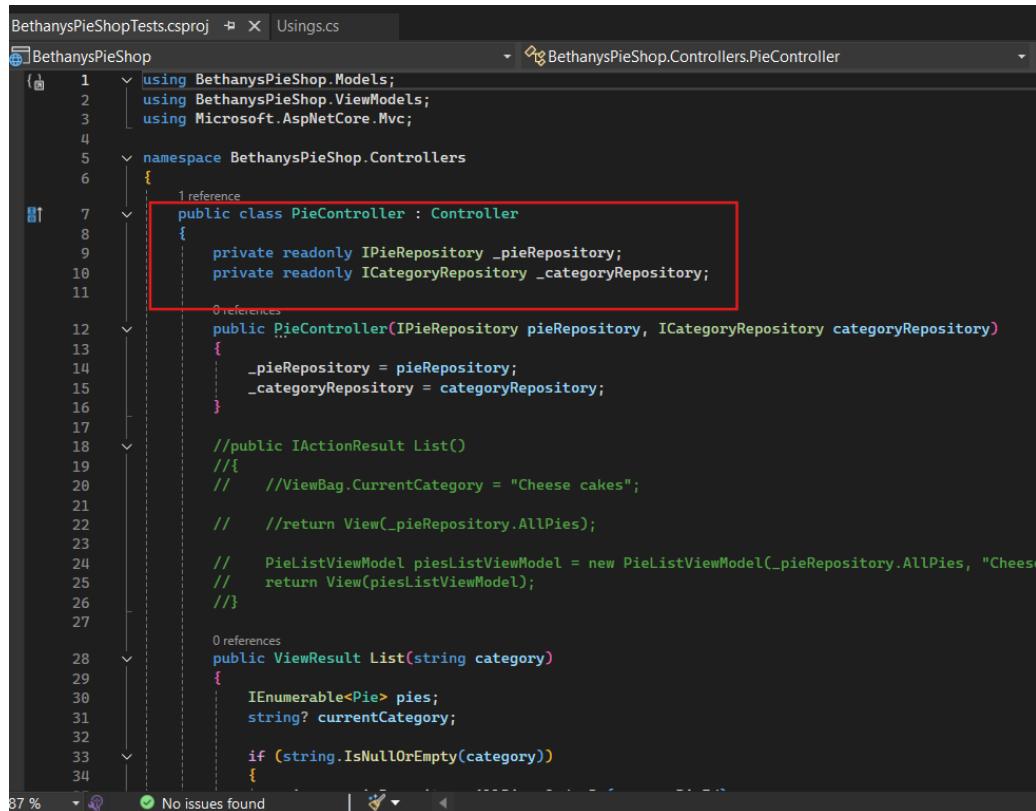
Lo modifichiamo nel seguente modo:

Home



```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
    <PackageReference Include="xunit" Version="2.5.3" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.5.3" />
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="coverlet.collector" Version="6.0.0" />
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Moq" Version="4.20.70" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include=".\\BethanyPieShop\\BethanyPieShop.csproj" />
  </ItemGroup>
</Project>
```

Okay, ora possiamo iniziare con PieController, possiamo notare che ha due dipendenze IPieRepository e ICategoryRepository:



```
using BethanyPieShop.Models;
using BethanyPieShop.ViewModels;
using Microsoft.AspNetCore.Mvc;

namespace BethanyPieShop.Controllers
{
    public class PieController : Controller
    {
        private readonly IPieRepository _pieRepository;
        private readonly ICategoryRepository _categoryRepository;
    }

    public PieController(IPieRepository pieRepository, ICategoryRepository categoryRepository)
    {
        _pieRepository = pieRepository;
        _categoryRepository = categoryRepository;
    }

    //public IActionResult List()
    //{
    //    ViewBag.CurrentCategory = "Cheese cakes";
    //    return View(_pieRepository.AllPies);
    //}

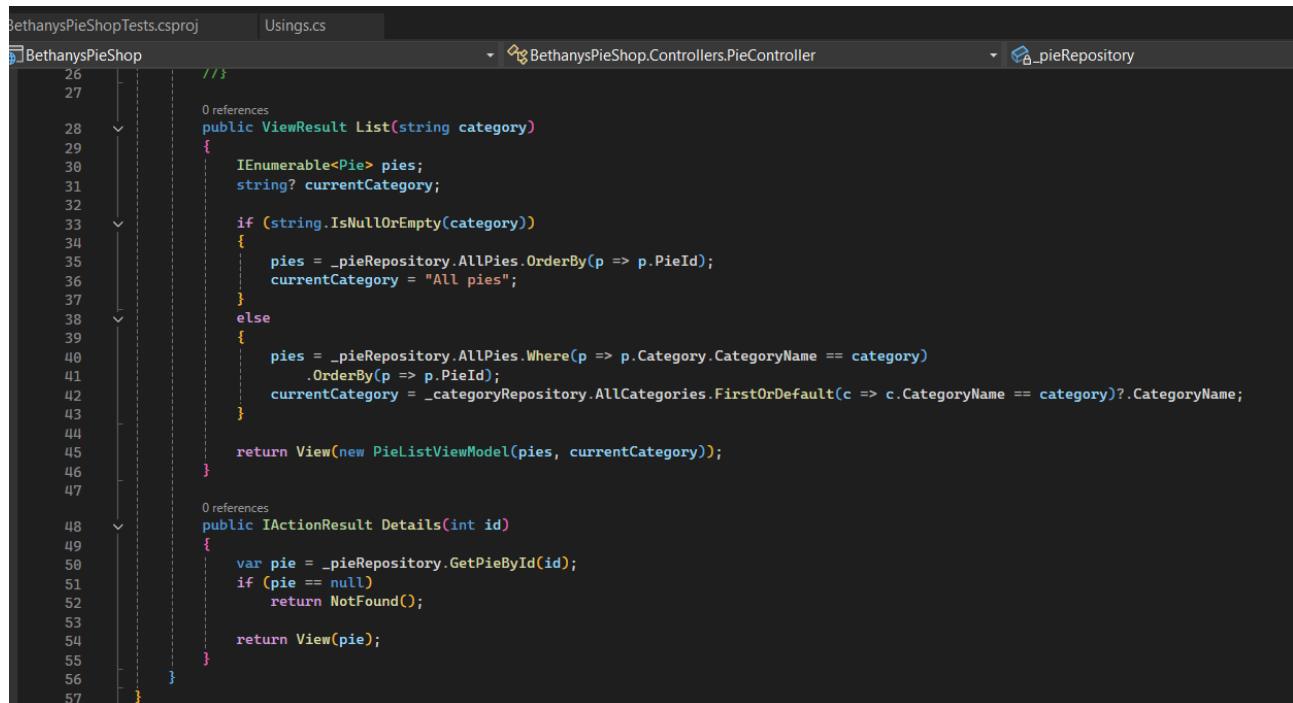
    // PieListViewModel piesListViewModel = new PieListViewModel(_pieRepository.AllPies, "Cheese cakes");
    // return View(piesListViewModel);
    //}

    public ViewResult List(string category)
    {
        IEnumerable<Pie> pies;
        string? currentCategory;
        if (string.IsNullOrEmpty(category))
        {
```

[Home](#)

Ora creeremo una versione simulata di questi e li initteremo nel PieController quando vogliamo testarlo.

Testeremo quindi il metodo List e ricapitoliamo cosa fa questo metodo:

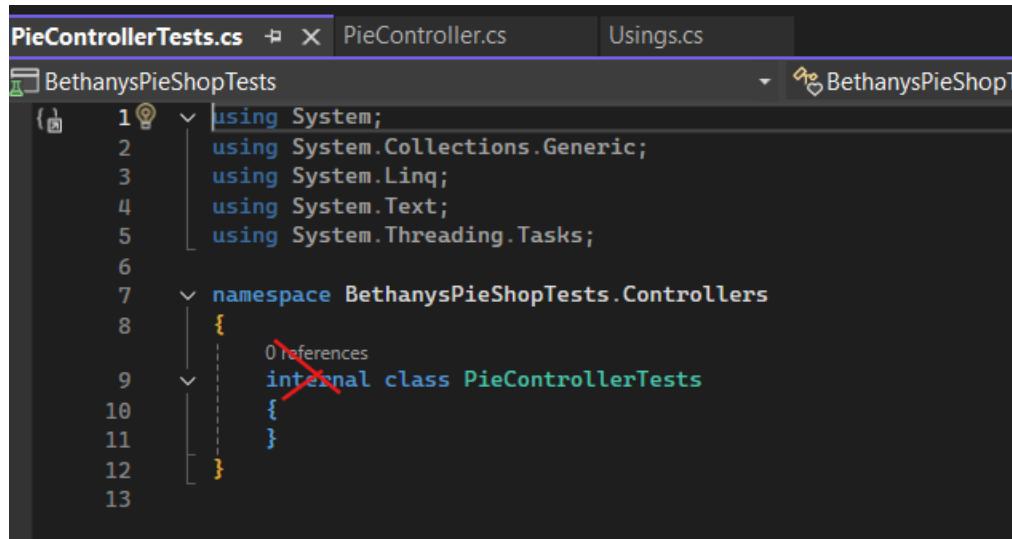


The screenshot shows the code for the `PieController.cs` file in Visual Studio. The code defines two methods: `List` and `Details`. The `List` method takes a `category` parameter and returns a `ViewResult`. It checks if the category is null or empty, in which case it retrieves all pies from the repository and sets the current category to "All pies". Otherwise, it filters the pies by category and retrieves the corresponding category name from the repository. The `Details` method takes an `id` parameter, retrieves the pie by ID from the repository, and returns a `ViewResult` containing the pie object. The code uses `IEnumerable<Pie>` for the pies and `IRepository<Pie>` for the pie repository.

```
26     // ...
27     0 references
28     public ViewResult List(string category)
29     {
30         I Enumerable<Pie> pies;
31         string? currentCategory;
32
33         if (string.IsNullOrEmpty(category))
34         {
35             pies = _pieRepository.AllPies.OrderBy(p => p.PieId);
36             currentCategory = "All pies";
37         }
38         else
39         {
40             pies = _pieRepository.AllPies.Where(p => p.Category.CategoryName == category)
41                 .OrderBy(p => p.PieId);
42             currentCategory = _categoryRepository.AllCategories.FirstOrDefault(c => c.CategoryName == category)?.CategoryName;
43         }
44
45         return View(new PieListViewModel(pies, currentCategory));
46     }
47
48     0 references
49     public IActionResult Details(int id)
50     {
51         var pie = _pieRepository.GetPieById(id);
52         if (pie == null)
53             return NotFound();
54
55         return View(pie);
56     }
57 }
```

Dopo averlo esaminato andiamo nel nostro progetto di test "BethanysPieShopTests" e creiamo la cartella Controllers

Al suo interno creiamo `PieControllerTests.cs`



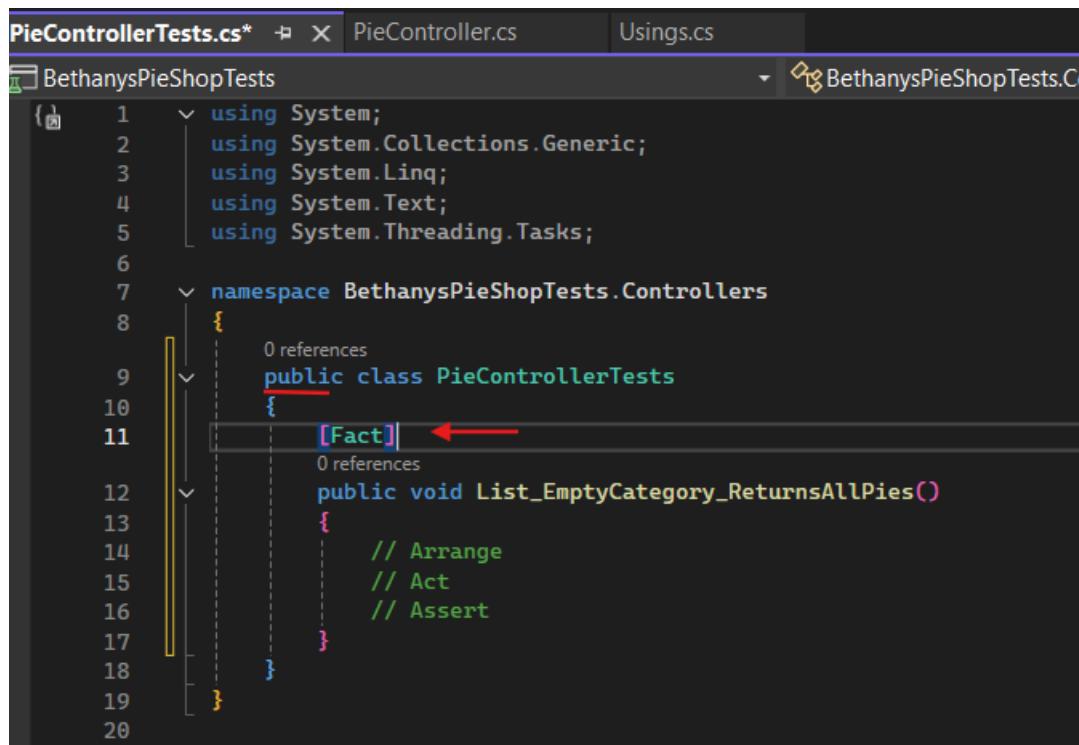
The screenshot shows the code for the `PieControllerTests.cs` file in Visual Studio. The code starts with several using statements for System, Collections.Generic, Linq, Text, and Threading.Tasks. It then defines a namespace `BethanysPieShopTests.Controllers` and an internal class `PieControllerTests`. The class contains a single constructor that takes a `PieController` parameter and a `TestOutputHelper` parameter. A large red X is drawn over the entire class definition.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace BethanysPieShopTests.Controllers
8 {
9     0 references
10    internal class PieControllerTests
11    {
12    }
13 }
```

Eliminiamo internal e scriviamo public

[Home](#)

Per prima cosa testiamo il metodo List che simula il caso in cui non passiamo nessuna Categoria come parametro, il metodo del test deve essere parlante.



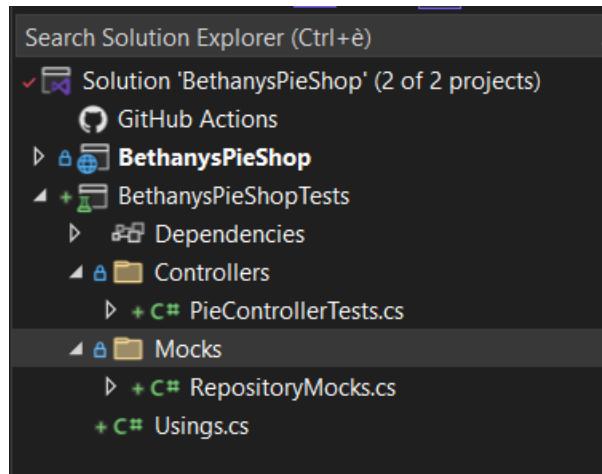
```
PieControllerTests.cs*  X  PieController.cs  Usings.cs
BethanysPieShopTests
1  1  using System;
2  2  using System.Collections.Generic;
3  3  using System.Linq;
4  4  using System.Text;
5  5  using System.Threading.Tasks;
6
7  7  namespace BethanysPieShopTests.Controllers
8  8  {
9  9      public class PieControllerTests
10 10     {
11 11         [Fact] ←
12 12         public void List_EmptyCategory_ReturnsAllPies()
13 13         {
14 14             // Arrange
15 15             // Act
16 16             // Assert
17 17         }
18 18     }
19 19  }
20 20  }
```

Osservazione: Aggiungiamo l'attributo Fact per far sì che questo metodo è un test unitario.

Ci ricordiamo che ci servono un PieRepository e un CategoryRepository.

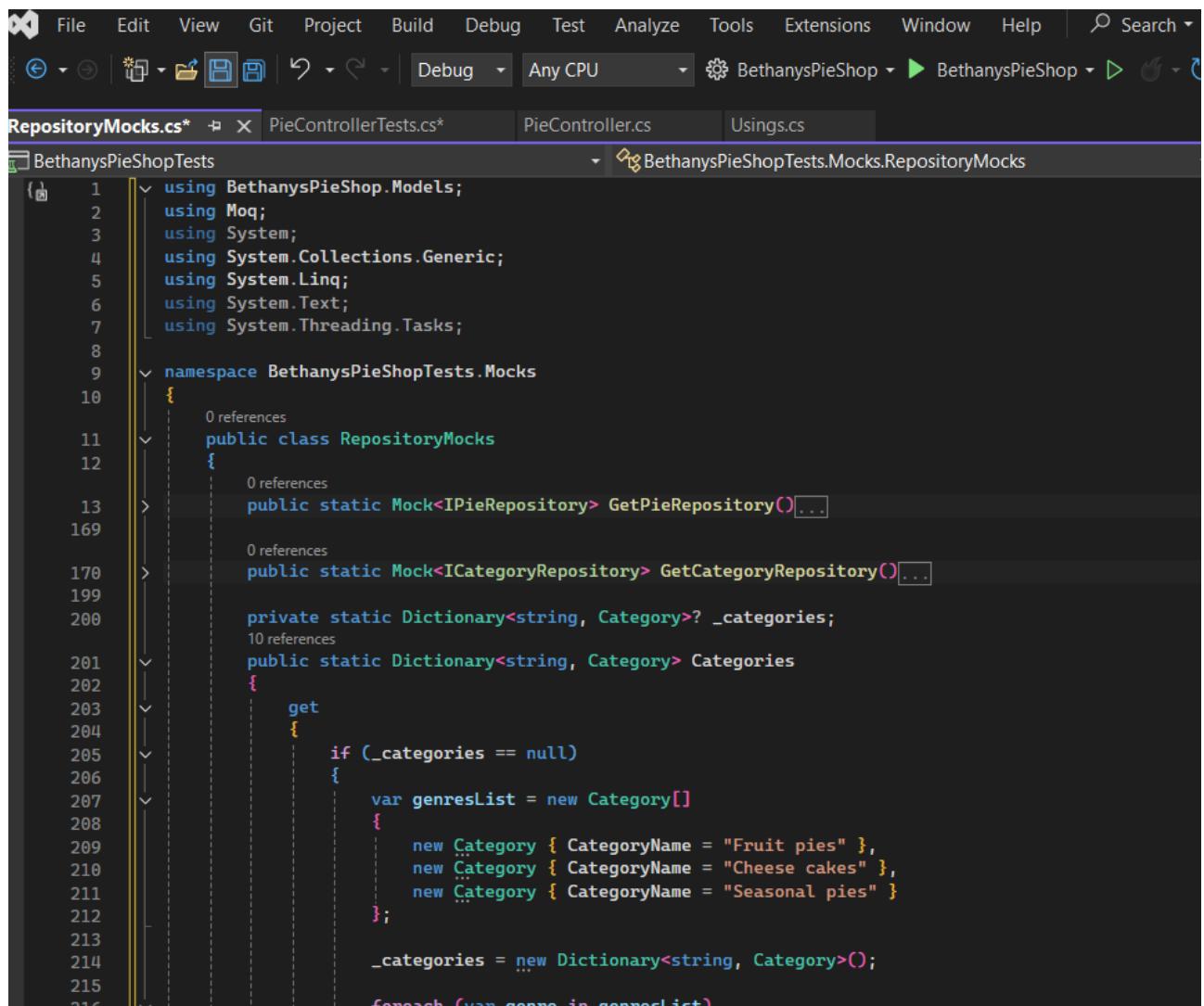
Non usiamo quelli veri ma una versione fittizia un mock.

Quindi nel progetto di test unitario aggiungiamo una cartella chiamata Mocks e al suo intero una classe chiamata RepositoryMocks:



[Home](#)

Incolliamo i nostri snippets:

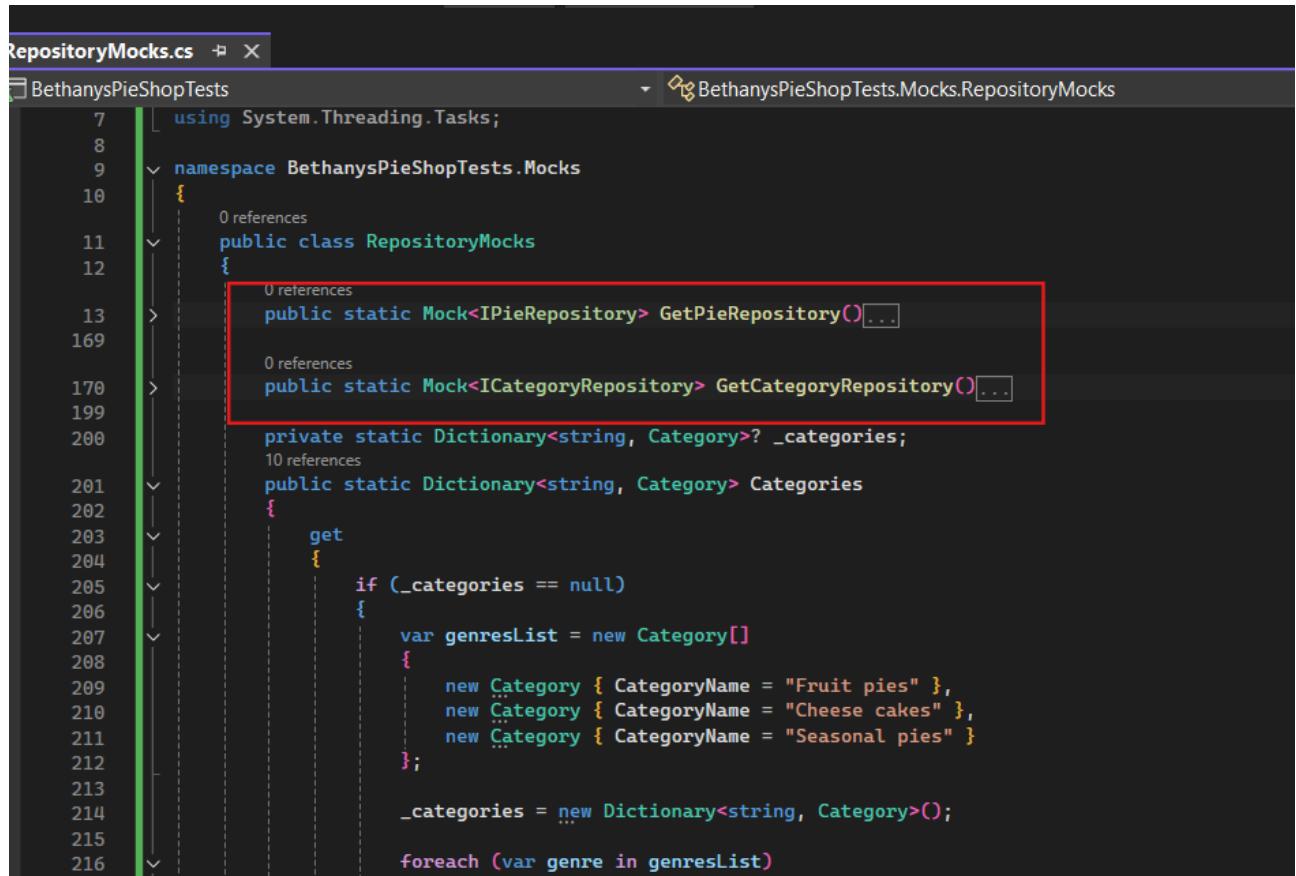


The screenshot shows the Visual Studio IDE interface with the following details:

- File Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbox:** Standard icons for file operations.
- Project Explorer:** Shows the solution structure with "BethanyPieShop" and its sub-project "BethanyPieShopTests".
- Task List:** Shows "BethanyPieShopTests.Mocks.RepositoryMocks".
- Code Editor:** Displays the `RepositoryMocks.cs` file content.

```
1  using BethanyPieShop.Models;
2  using Moq;
3  using System;
4  using System.Collections.Generic;
5  using System.Linq;
6  using System.Text;
7  using System.Threading.Tasks;
8
9  namespace BethanyPieShopTests.Mocks
10 {
11     public class RepositoryMocks
12     {
13         public static Mock<IPieRepository> GetPieRepository()
14         {
15             return new Mock<IPieRepository>();
16         }
17
18         public static Mock<ICategoryRepository> GetCategoryRepository()
19         {
20             return new Mock<ICategoryRepository>();
21         }
22
23         private static Dictionary<string, Category>? _categories;
24
25         public static Dictionary<string, Category> Categories
26         {
27             get
28             {
29                 if (_categories == null)
30                 {
31                     var genresList = new Category[]
32                     {
33                         new Category { CategoryName = "Fruit pies" },
34                         new Category { CategoryName = "Cheese cakes" },
35                         new Category { CategoryName = "Seasonal pies" }
36                     };
37
38                     _categories = new Dictionary<string, Category>();
39                     foreach (var genre in genresList)
40                         _categories.Add(genre.CategoryName, genre);
41                 }
42             }
43         }
44     }
45 }
```

[Home](#)

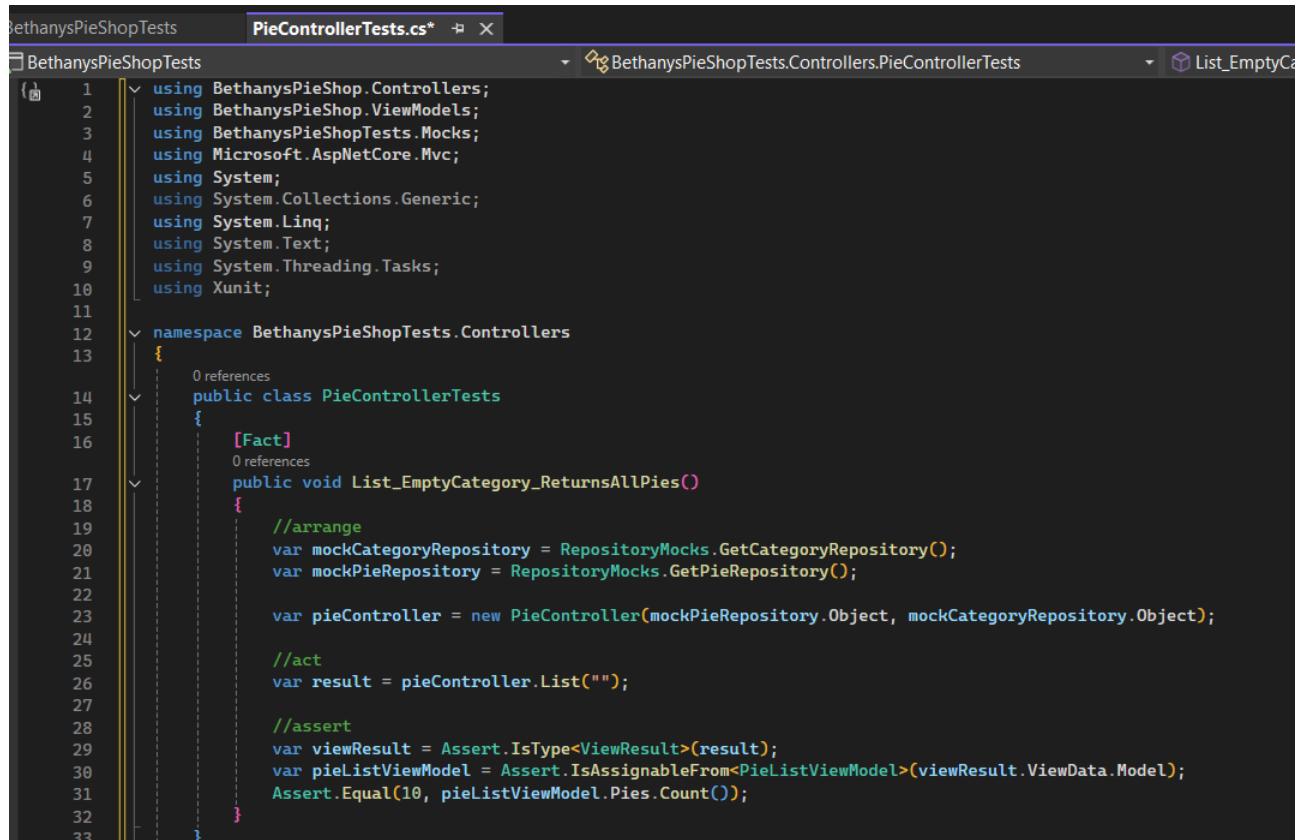


```
RepositoryMocks.cs  ✘ X
BethanysPieShopTests  ✘ BethanysPieShopTests.Mocks.RepositoryMocks
7   using System.Threading.Tasks;
8
9   namespace BethanysPieShopTests.Mocks
10  {
11      public class RepositoryMocks
12      {
13          public static Mock<IPieRepository> GetPieRepository()
14          {
15              return null;
16          }
17
18          public static Mock<ICategoryRepository> GetCategoryRepository()
19          {
20              return null;
21          }
22
23          private static Dictionary<string, Category>? _categories;
24
25          public static Dictionary<string, Category> Categories
26          {
27              get
28              {
29                  if (_categories == null)
30                  {
31                      var genresList = new Category[]
32                      {
33                          new Category { CategoryName = "Fruit pies" },
34                          new Category { CategoryName = "Cheese cakes" },
35                          new Category { CategoryName = "Seasonal pies" }
36                      };
37
38                      _categories = new Dictionary<string, Category>();
39
40                      foreach (var genre in genresList)
41                          _categories.Add(genre.CategoryName, genre);
42
43                  }
44
45                  return _categories;
46              }
47          }
48
49      }
50
51  }
```

Osservazione: abbiamo due metodi Get che restituiscono una versione fittizia delle due IRepository.
Tutto ciò è possibile grazie al framework mock

Torniamo al nostro PieControllerTest.cs e lo completiamo nel seguente modo:

[Home](#)



The screenshot shows the Visual Studio code editor with the file `PieControllerTests.cs` open. The code is a unit test for the `PieController`. It uses XUnit and Moq for testing. The test method `List_EmptyCategory_ReturnsAllPies` checks if the controller returns all pies when listing an empty category.

```
1  using BethanysPieShop.Controllers;
2  using BethanysPieShop.ViewModels;
3  using BethanysPieShopTests.Mocks;
4  using Microsoft.AspNetCore.Mvc;
5  using System;
6  using System.Collections.Generic;
7  using System.Linq;
8  using System.Text;
9  using System.Threading.Tasks;
10 using Xunit;
11
12 namespace BethanysPieShopTests.Controllers
13 {
14     public class PieControllerTests
15     {
16         [Fact]
17         public void List_EmptyCategory_ReturnsAllPies()
18         {
19             //arrange
20             var mockCategoryRepository = RepositoryMocks.GetCategoryRepository();
21             var mockPieRepository = RepositoryMocks.GetPieRepository();
22
23             var pieController = new PieController(mockPieRepository.Object, mockCategoryRepository.Object);
24
25             //act
26             var result = pieController.List("");
27
28             //assert
29             var viewResult = Assert.IsType<ViewResult>(result);
30             var pieListViewModel = Assert.IsAssignableFrom<PieListViewModel>(viewResult.ViewData.Model);
31             Assert.Equal(10, pieListViewModel.Pies.Count());
32         }
33     }
}
```

Osservazione sugli assert:

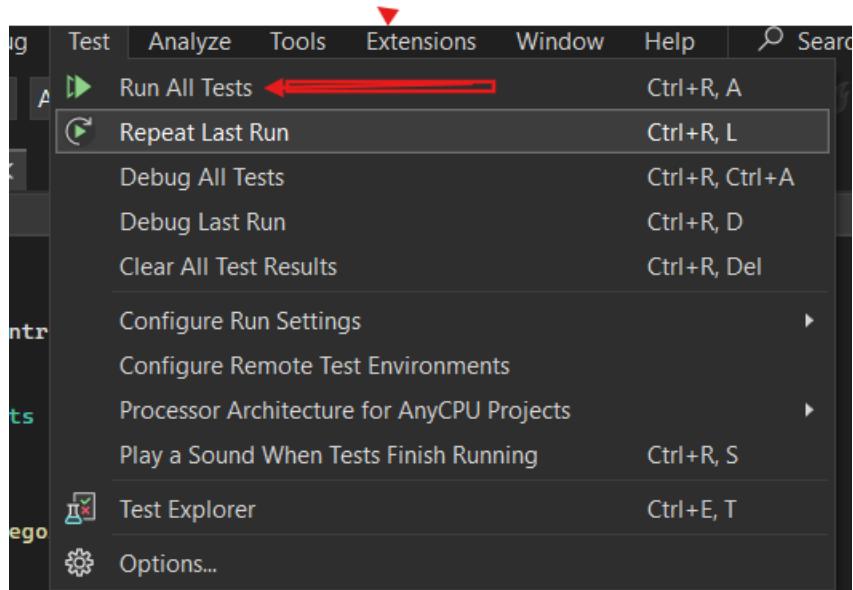
riga 29: Verifichiamo che quanto restituito sia di tipo `viewResult` (in quanto in `PieController.cs` ritorniamo una vista)

riga 30: Verifichiamo che `ViewRsult.ViewData.Model` quindi il modello che abbiamo passato (ovvero il parametro passato al return `View` del metodo `List` di `PieController.cs`)

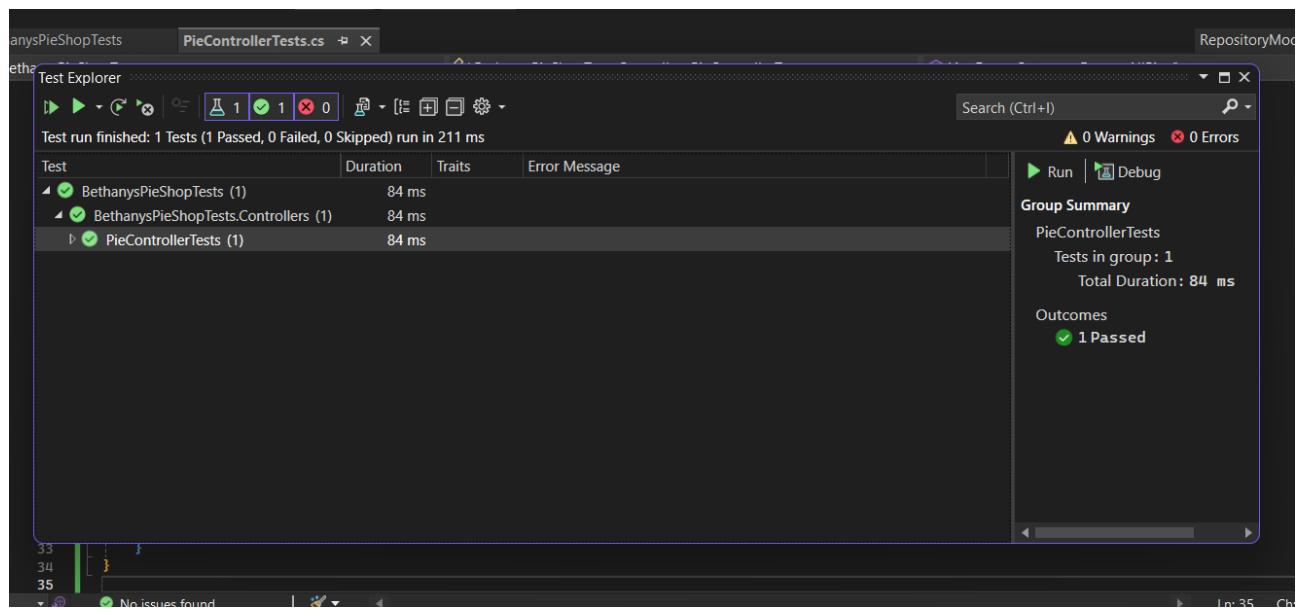
Riga31: Ora verifichiamo se sono presenti 10 torte che abbiamo aggiunto effettivamente nella Mock

Ora in V.S. andiamo in `Test > Run All Tests`

[Home](#)



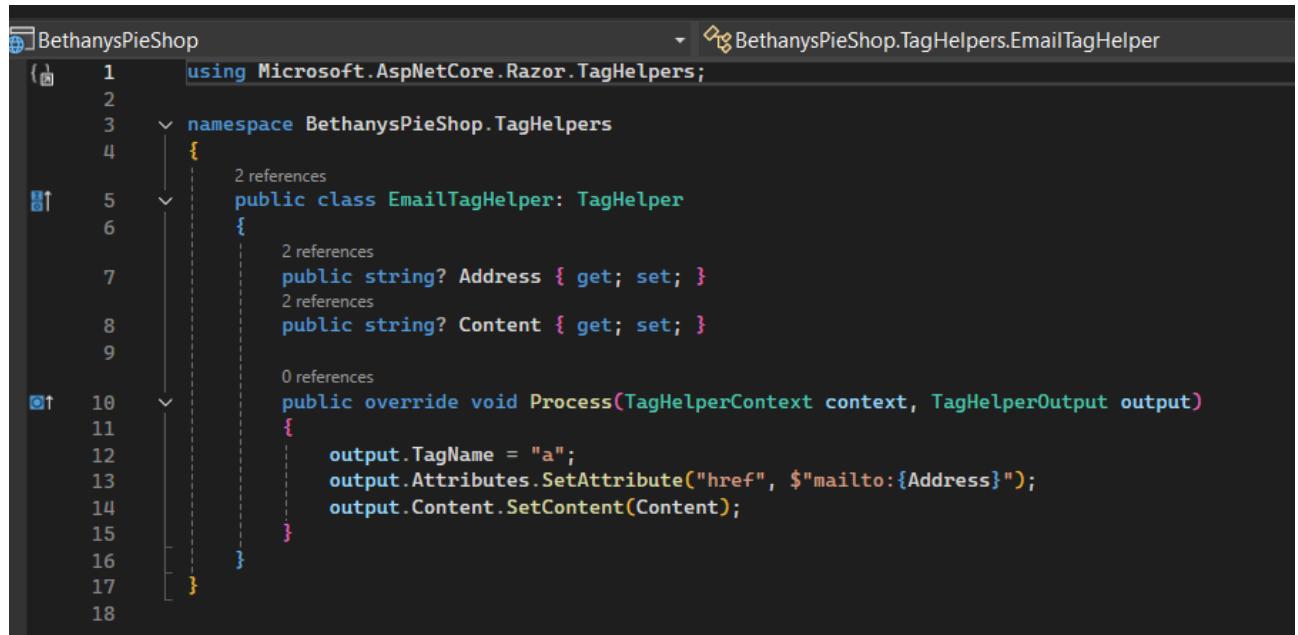
Output:



Demo: Testing Tag Helper

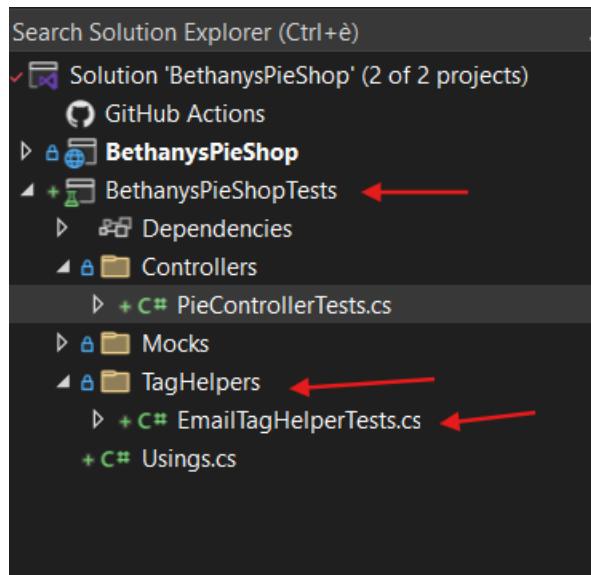
Ricordandoci EmailTagHelper presente nella cartella TagHelpers

[Home](#)



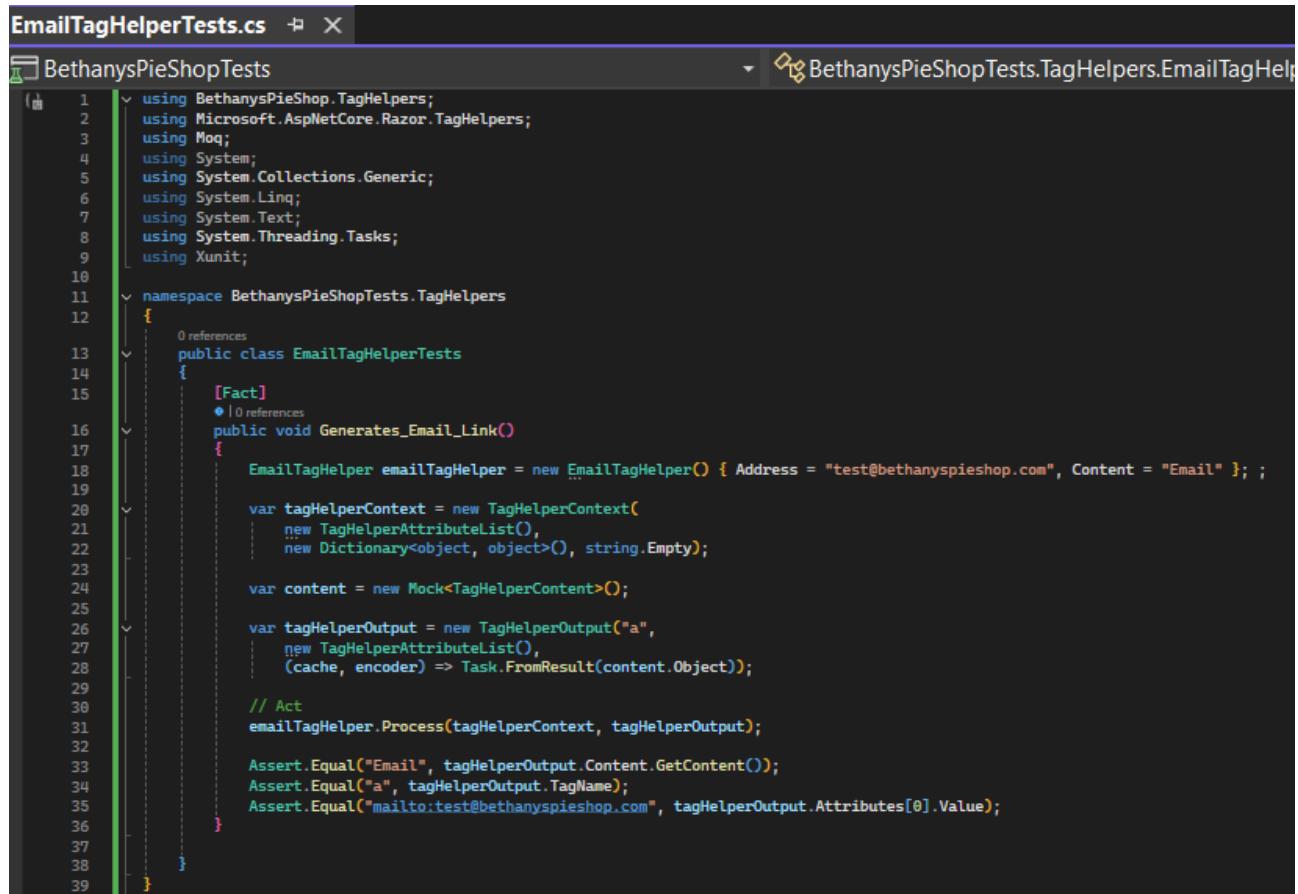
```
1  using Microsoft.AspNetCore.Razor.TagHelpers;
2
3  namespace BethanysPieShop.TagHelpers
4  {
5      public class EmailTagHelper : TagHelper
6      {
7          public string? Address { get; set; }
8          public string? Content { get; set; }
9
10     public override void Process(TagHelperContext context, TagHelperOutput output)
11     {
12         output.TagName = "a";
13         output.Attributes.SetAttribute("href", $"mailto:{Address}");
14         output.Content.SetContent(Content);
15     }
16 }
17
18 }
```

Creiamo quindi la cartella e il file per testare EmailTagHelper:



Completiamo lo script nel seguente modo:

[Home](#)



The screenshot shows a code editor window with the title "EmailTagHelperTests.cs". The code is written in C# and defines a unit test for the `EmailTagHelper`. The test class is named `EmailTagHelperTests` and contains a single test method named `Generates_Email_Link`. The test uses Moq to mock the `TagHelperContext` and `TagHelperOutput` classes. It creates a new instance of `EmailTagHelper` with an address and content, then calls its `Process` method with the mocked context and output. Finally, it asserts that the output content is "Email", the tag name is "a", and the attribute value is "mailto:test@bethanyspieshop.com".

```
1  using BethanysPieShop.TagHelpers;
2  using Microsoft.AspNetCore.Razor.TagHelpers;
3  using Moq;
4  using System;
5  using System.Collections.Generic;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using Xunit;
10
11 namespace BethanysPieShopTests.TagHelpers
12 {
13     [Fact]
14     public class EmailTagHelperTests
15     {
16         [Fact]
17         public void Generates_Email_Link()
18         {
19             EmailTagHelper emailTagHelper = new EmailTagHelper() { Address = "test@bethanyspieshop.com", Content = "Email" };
20
21             var tagHelperContext = new TagHelperContext(
22                 new TagHelperAttributeList(),
23                 new Dictionary<object, object>(),
24                 string.Empty);
25
26             var content = new Mock<TagHelperContent>();
27
28             var tagHelperOutput = new TagHelperOutput("a",
29                 new TagHelperAttributeList(),
30                 (cache, encoder) => Task.FromResult(content.Object));
31
32             // Act
33             emailTagHelper.Process(tagHelperContext, tagHelperOutput);
34
35             Assert.Equal("Email", tagHelperOutput.Content.GetContent());
36             Assert.Equal("a", tagHelperOutput.TagName);
37             Assert.Equal("mailto:test@bethanyspieshop.com", tagHelperOutput.Attributes[0].Value);
38         }
39     }
}
```

Osservazioni:

Anche qui abbiamo gli arrange, inizialmente creo un'istanza di `EmailTagHelper`, passando un valore per l'indirizzo e il contenuto, poi abbiamo ACT che chiama il metodo `process` che passa i due parametri per il processo (come avviene nella classe originale).

In `Assert` verifichiamo quindi il contenuto, 'a', e il "mailto"

Questo dimostra come possiamo utilizzare test unitari anche per cose diverse dai controller

Making the Site Interactive

- Searching Using JavaScript and an ASP.NET Core API
- Creating an ASP.NET Core RESTful API
- Demo: Setting up the API
- Creating the API Responses
- Demo: Completing the API
- Adding jQuery and Ajax
- Demo: Creating the Search Page with Ajax and the API
- Introducing ASP.NET Core Blazor
- Demo: Exploring a New Blazor Project
- Demo: Creating the Search Page Using Blazor

Searching Using JavaScript and an ASP.NET Core API

Creating the Search Page



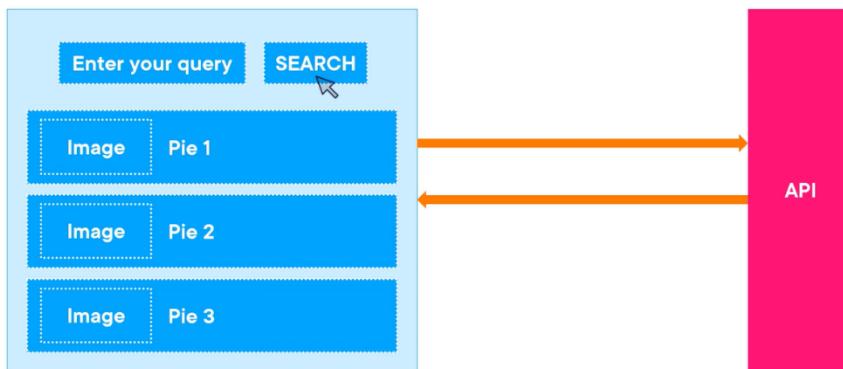
Però questo approccio presenta alcuni svantaggi:

Disadvantages of this Approach

-  Full page needs to refresh
-  More data over the wire
-  Slower
-  “Data” is not accessible for third-party

Quindi creeremo un'API, fondamentalmente un servizio, utilizzando ASP.NET Core e che non restituirà HTML, ma solo i dati:

Updating Parts of the Page



Utilizzando JavaScript quindi codice lato client, questo comunicherà con l'API per attivare l'esecuzione della query di ricerca sul server.

L'Api restituirà quindi semplicemente i dati.

Effettueremo un aggiornamento parziale della pagina invece di eseguire nuovamente il rendering dell'intera pagina (grazie a JQuery)

Quindi quello che faremo è:

1. Creare un API che restituisca solo i dati
2. Includere JQuery per visualizzare i risultati

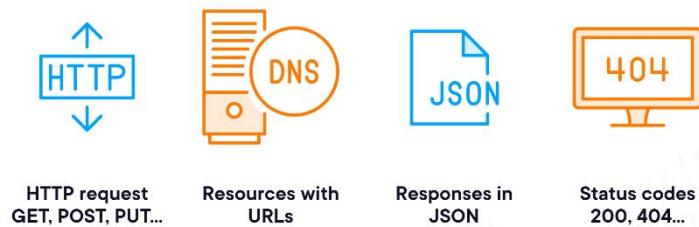
Creating an ASP.NET Core RESTful API

[Home](#)

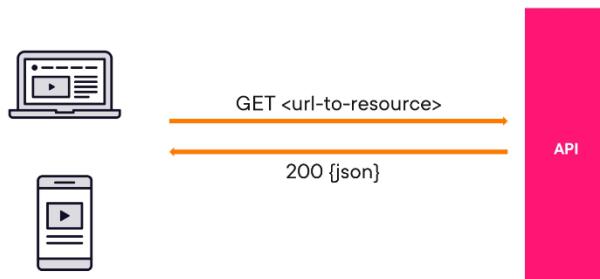
Creating an API

-  Uses “just” the data
-  JSON or XML
-  Open for many types of clients
-  Can also be built using ASP.NET Core and MVC approach

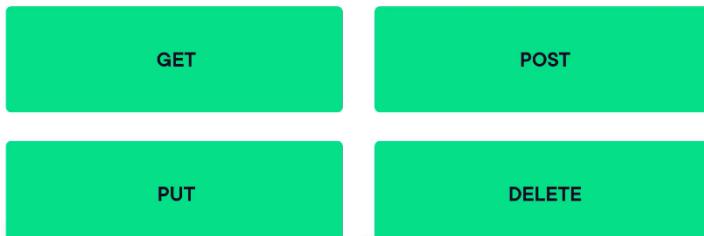
Creating a RESTful API



Creating a RESTful API



HTTP Verbs



JSON Response

```
[{"productId": 1, "name": "Apple Pie", "shortDescription": "Our famous apple pies!", "longDescription": "", "allergenInformation": "", "price": 12.95, "imageURL": "files/applepie.jpg", "imageThumbnailURL": "files/applepiesmall.jpg", "isItemOfTheWeek": true, "inStock": true, "categoryId": 1, "category": { "categoryId": 1, "categoryName": "Fruit pies", "description": null, "pies": [ null ] } }, {"productId": 2, "name": "Blueberry Cheese Cake", "shortDescription": "You'll love it!", "longDescription": "It's", "allergenInformation": "", "price": 18.95, "imageURL": "files/blueberrycheesecake.jpg", "imageThumbnailURL": "files/blueberrycheesecakesmall.jpg", "isItemOfTheWeek": false, "inStock": true, "categoryId": 2, "category": { "categoryId": 2, "categoryName": "Cheese cakes", "description": null, "pies": [ null ] } }]
```

Creating an API with ASP.NET Core

Controller-based APIs

Similar way of working to MVC

Most complete approach

Minimal APIs

Based on endpoints defined in Program.cs

Not all features supported

Creating an API with ASP.NET Core Controllers



Controller that returns data



JsonResult



Attribute-based routing



Other concepts are identical

```
builder.Services.AddControllers();  
app.MapControllers();
```

Configuring the Application

Program.cs

Not needed separately if already done for regular MVC

```
public class PieController : ControllerBase  
{  
}
```

Creating a Controller

ControllerBase adds support for access to HttpContext, Request...

Routing Options in ASP.NET Core

Convention-based routing

Attribute-based routing

```
[Route("api/[controller]")]
public class PieController : ControllerBase
{  
}
```

Using the Route Attribute

Accessible via /api/search

```
[Route("api/[controller]")]
public class PieController : ControllerBase
{  
}
```

Using the Route Attribute

Accessible via /api/search

Reaching the Action Methods

```
[Route("api/[controller]")]
public class PieController : ControllerBase
{
    private readonly IPieRepository _pieRepository;

    [HttpGet]
    public IActionResult GetAll()
    {
        ...
    }
}
```

corrisponde a una vista denominata index.cs HTML in una

```
[Route("api/[controller]")]
public class PieController : ControllerBase
{
    private readonly IPieRepository _pieRepository;

    [HttpGet("{id}")]
    public IActionResult GetById(int id)
    {
        ...
    }
}
```

Passing a Parameter

Uses model binding again
Can work with complex types too

Routing to an API Action Method



Se, dal client, invii una richiesta `HttpGet` ad `api/pie/3`.



Demo: Setting up the API

Ora includiamo una API.

Home

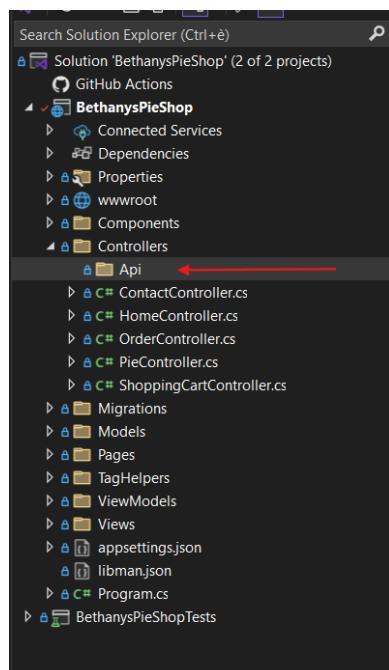
Come dimostrato nella teoria, dovremmo aggiungere il servizio Controller, nel nostro caso non è necessario in quanto ne abbiamo già uno che include le viste, inoltre abbiamo già anche un supporto per il Routing:

```
Program.cs*  • X
[BethanyPieShop]
1  using System;
2  using Microsoft.EntityFrameworkCore;
3  var builder = WebApplication.CreateBuilder(args);
4
5  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
6  builder.Services.AddScoped<IPieRepository, PieRepository>();
7  builder.Services.AddScoped<OrderRepository, OrderRepository>();
8
9  builder.Services.AddScoped<ShoppingCart, ShoppingCart>(sp => ShoppingCart.Create(sp));
10 builder.Services.AddSession(); // Add session services
11 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
12
13 builder.Services.AddControllersWithViews(); // Add MVC services
14 builder.Services.AddRazorPages(); // Add Razor pages services
15
16 builder.Services.AddDbContext<BethanyPieShopContext>(options =>
17 {
18     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanyPieShop"]);
19 });
20
21 builder.Services.AddMvc(); // Add MVC services
22
23 var app = builder.Build();
```

```
Program.cs*  • X
[BethanyPieShop]
22 var app = builder.Build();
23
24 app.UseStaticFiles(); // Middleware component: Enable static files
25 app.UseSession(); // Middleware component: Enable session
26
27 if (app.Environment.IsDevelopment())
28 {
29     app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
30 }
31
32 // "{controller=Home}/{action=Index}/{id?}" is the default route
33 app.MapDefaultControllerRoute(); // Middleware component: Map default controller route
34
35 // Se avessimo voluto personalizzare la rotta o il pattern
36 // app.MapControllerRoute(
37 //     name: "default",
38 //     pattern: "{controller=Home}/{action=Index}/{id?}")
39
40 app.MapRazorPages(); // Middleware component: Enable Razor Pages
41
42 DBInitializer.Seed(app); // Seed the database
43
44 app.Run();
```

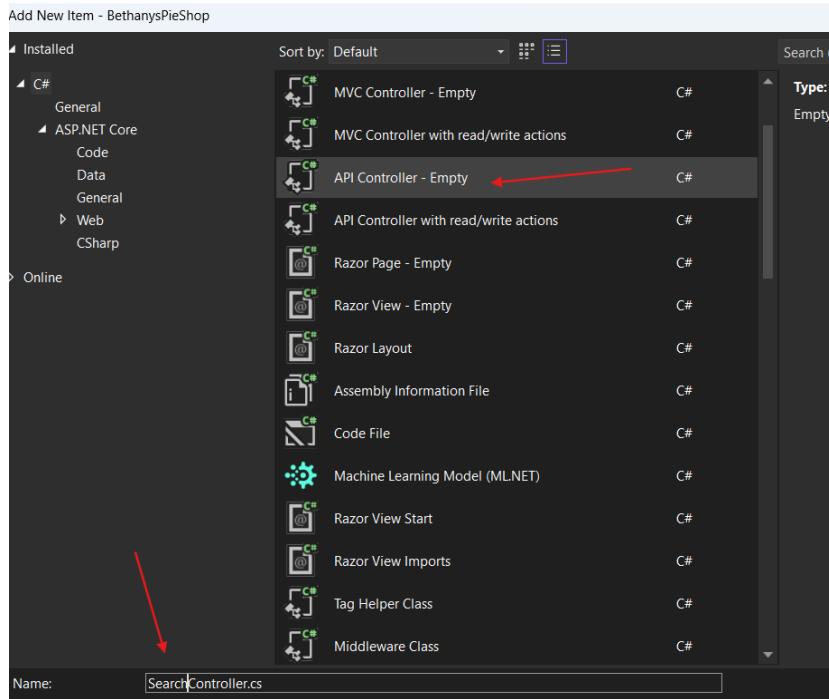
Dobbiamo quindi aggiungere un Controller che sia differente da quello MVC, quindi un controller che si occupi solo della logica API.

Quindi nella cartella Controllers creiamo una NewFolder che chiameremo Api



All'interno inseriamo un API Controller - Empty e lo chiamiamo SearchController
Che ci servirà per ricercare le torte

[Home](#)



```
1 using Microsoft.AspNetCore.Http;
2 using Microsoft.AspNetCore.Mvc;
3
4 namespace BethanysPieShop.Controllers.Api
5 {
6     [Route("api/[controller]")]
7     [ApiController]
8     public class SearchController : ControllerBase
9     {
10     }
11 }
12 }
```

Riga 6: indica che stiamo utilizzando un percorso specifico per instradare le richieste a questo controller.

Le API in genere utilizzano il routing basato sugli attributi, quindi inseriamo la route direttamente nel controller.

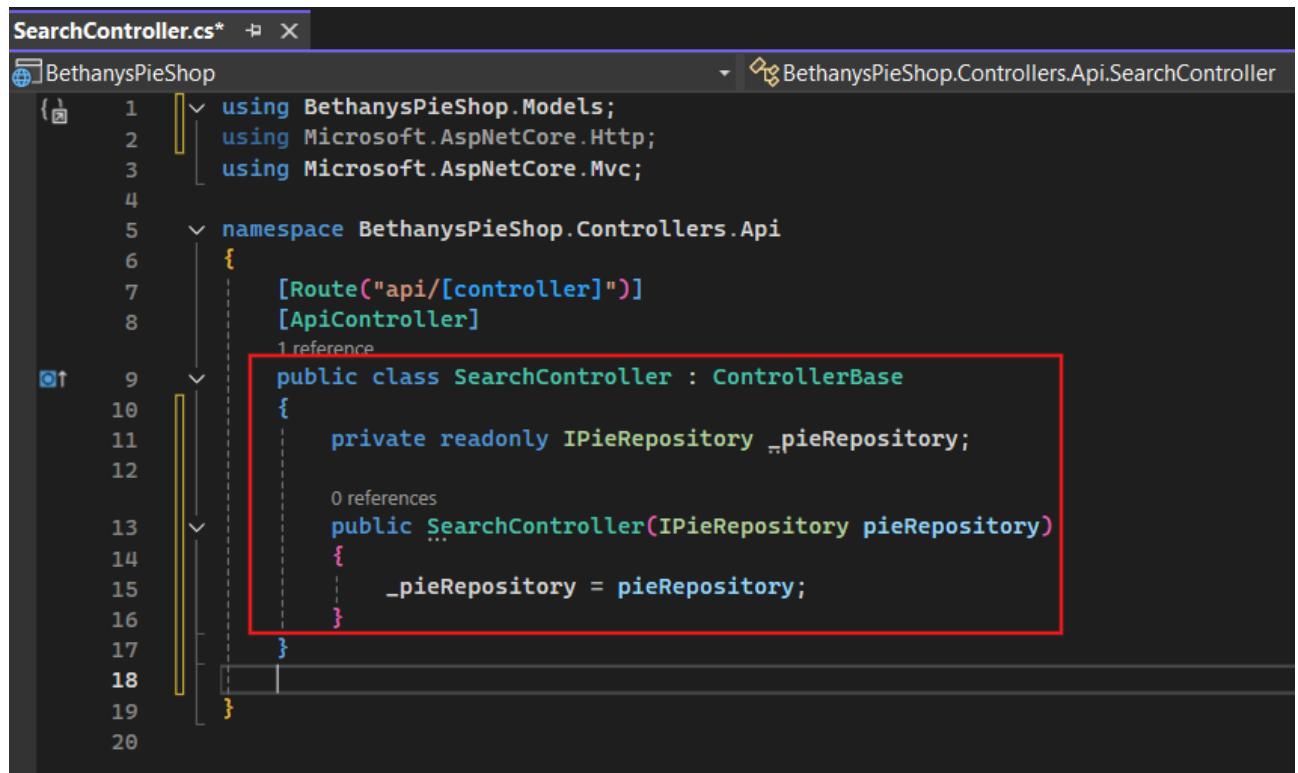
All'avvio Asp.Net crea una table di tutte le rotte.

[controller] è il segnaposto per il nome del controller, avremmo potuto scrivere per esempio
[Route("api/SearchController")]

Riga7: L'uso di tale attributo è facoltativo, aggiunge solo alcuni comportamenti specifici dell'API.

[Home](#)

Detto ciò, inseriamo le D.I.:



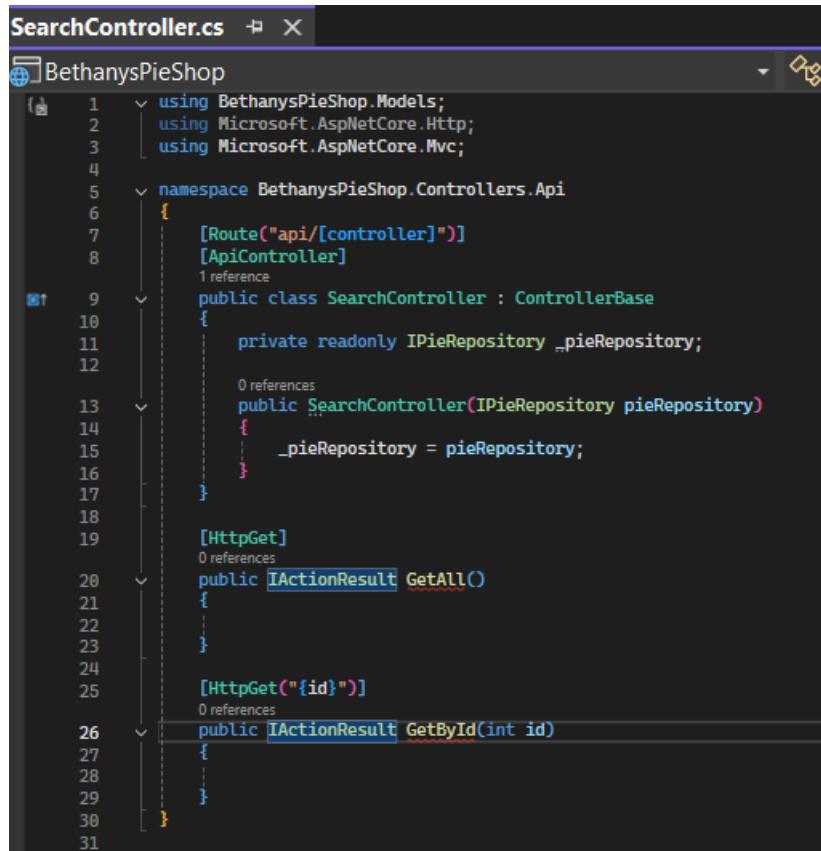
```
SearchController.cs*  ✎ X
BethanyPieShop
  1  using BethanyPieShop.Models;
  2  using Microsoft.AspNetCore.Http;
  3  using Microsoft.AspNetCore.Mvc;
  4
  5  namespace BethanyPieShop.Controllers.Api
  6  {
  7      [Route("api/[controller]")]
  8      [ApiController]
  9      public class SearchController : ControllerBase
 10     {
 11         private readonly IPieRepository _pieRepository;
 12
 13         public SearchController(IPieRepository pieRepository)
 14         {
 15             _pieRepository = pieRepository;
 16         }
 17     }
 18 }
 19
 20
```

The code editor shows the `SearchController.cs` file. A red box highlights the constructor injection code:

```
public SearchController(IPieRepository pieRepository)
```

Ed ora l'action method:

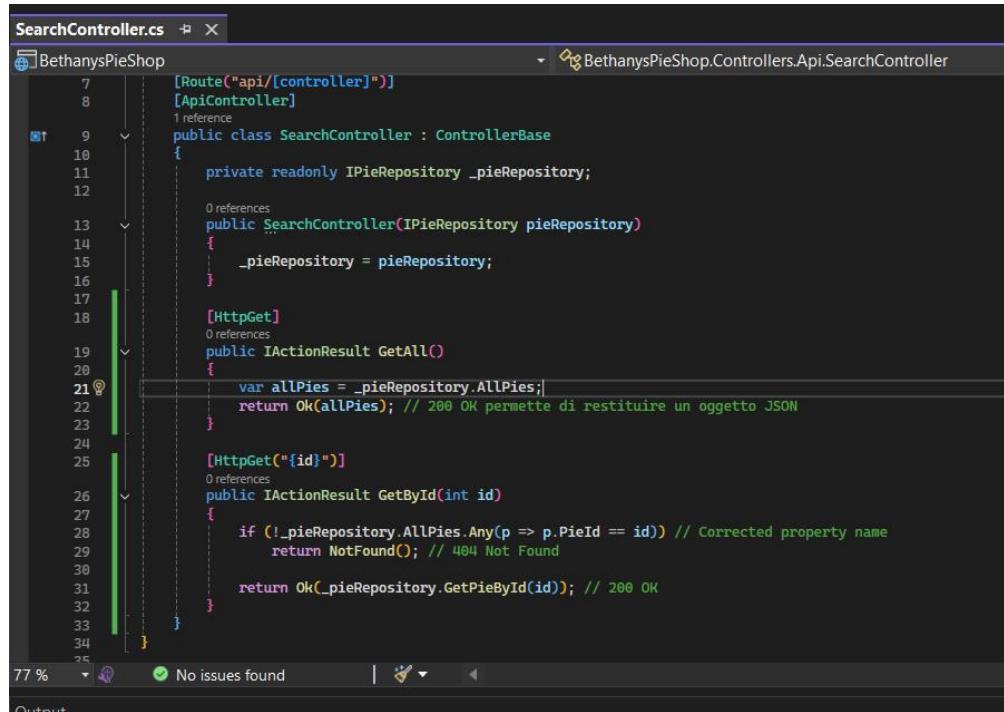
[Home](#)



```
SearchController.cs  ✎ X
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Http;
3  using Microsoft.AspNetCore.Mvc;
4
5  namespace BethanysPieShop.Controllers.Api
6  {
7      [Route("api/[controller]")]
8      [ApiController]
9      public class SearchController : ControllerBase
10     {
11         private readonly IPieRepository _pieRepository;
12
13         public SearchController(IPieRepository pieRepository)
14         {
15             _pieRepository = pieRepository;
16         }
17
18         [HttpGet]
19         public IActionResult GetAll()
20         {
21         }
22
23         [HttpGet("{id}")]
24         public IActionResult GetById(int id)
25         {
26         }
27
28
29
30     }
31 }
```

(Lasciamo questo script così momentaneamente)

Creating the API Responses



```
SearchController.cs  ✎ X
BethanysPieShop  ✎ X
BethanysPieShop.Controllers.Api.SearchController
7  [Route("api/[controller]")]
8  [ApiController]
9  public class SearchController : ControllerBase
10 {
11     private readonly IPieRepository _pieRepository;
12
13     public SearchController(IPieRepository pieRepository)
14     {
15         _pieRepository = pieRepository;
16     }
17
18     [HttpGet]
19     public IActionResult GetAll()
20     {
21         var allPies = _pieRepository.AllPies();
22         return Ok(allPies); // 200 OK permette di restituire un oggetto JSON
23     }
24
25     [HttpGet("{id}")]
26     public IActionResult GetById(int id)
27     {
28         if (!_pieRepository.AllPies.Any(p => p.PieId == id)) // Corrected property name
29             return NotFound(); // 404 Not Found
30
31         return Ok(_pieRepository.GetPieById(id)); // 200 OK
32     }
33
34 }
```

[Home](#)

Ricorda, una torta fa riferimento a una categoria e una categoria fa riferimento a un elenco di torte, durante la serializzazione delle torte, ASP.NET Core si confonderà perché diciamo di entrare in un ciclo infinito in cui si hanno categorie di riferimento alle torte, un altro riferimento alle torte e così via..

Dobbiamo dire a ASP.NET Core che durante la serializzazione dovrebbe sostanzialmente ignorare questi cicli, per far ciò torniamo alla classe Program.cs e inseriamo le seguenti istruzioni:

```
Program.cs*  ✎ X
BethanyPieShop
1  using BethanyPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3  using System.Text.Json.Serialization;
4
5  var builder = WebApplication.CreateBuilder(args);
6
7  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
8  builder.Services.AddScoped<IPieRepository, PieRepository>();
9  builder.Services.AddScoped<IOrderRepository, OrderRepository>();
10
11 builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp)); // Add shopping
12 builder.Services.AddSession(); // Add session services
13 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
14
15 builder.Services.AddControllersWithViews() // Add MVC services
16     .AddJsonOptions(options =>
17     {
18         options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
19     });
20
21
22 builder.Services.AddRazorPages(); // Add Razor pages services
23 builder.Services.AddDbContext<BethanyPieShopDbContext>(options =>
24 {
    ...
});
```

Okay, salviamo, avviamo e ora giungiamo all'url <http://localhost:5000/api/search>

Verrà mostrato il Json:

```
[{"pieId": 1, "name": "Caramel Popcorn Cheese Cake", "shortDescription": "The ultimate cheese cake", "longDescription": "Icing carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy tootsie roll. Chocolate cake gingerbread tootsie roll oat cake pie chocolate bar cookie drag\u00e9e brownie. Lollipop cotton candy cake bear claw oat cake. Drag\u00e9e candy canes dessert tart. Marzipan drag\u00e9e gummies lollipop jujubes chocolate bar candy canes. Icing gingerbread chupa chups cotton candy cookie sweet icing bonbon gummies. Gummies lollipop brownie biscuit danish chocolate cake. Danish powder cookie macaron chocolate donut tart. Carrot cake drag\u00e9e croissant lemon drops liquorice lemon drops cookie lollipop toffee. Carrot cake carrot cake liquorice sugar plum topping bonbon pie muffin jujubes. Jelly pastry wafer tart caramels bear claw. Tiramisu tart pie cake danish lemon drops. Brownie cupcake drag\u00e9e gummies.", "allergyInformation": "", "price": 22.95, "imageUrl": "https://gillcleerenpluralsight.blob.core.windows.net/files/bethanypieshop/cheesecakes/caramelpopcorncheesecake.jpg", "imageThumbnailUrl": "https://gillcleerenpluralsight.blob.core.windows.net/files/bethanypieshop/cheesecakes/caramelpopcorncheesecakesmall.jpg", "isPieOfTheWeek": true, "inStock": true, "categoryId": 2, "category": { "categoryId": 2, "categoryName": "Cheese cakes", "description": null, "pies": [null] } }, {"pieId": 2, "name": "Chocolate Cheese Cake", "shortDescription": "The chocolate lover's dream", "longDescription": "Icing carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy tootsie roll. Chocolate cake gingerbread tootsie roll oat cake pie chocolate bar cookie drag\u00e9e brownie. Lollipop cotton candy cake bear claw oat cake. Drag\u00e9e candy canes dessert tart. Marzipan drag\u00e9e gummies lollipop jujubes chocolate bar candy canes. Icing gingerbread chupa chups cotton candy cookie sweet icing bonbon gummies. Gummies lollipop brownie biscuit danish chocolate cake. Danish powder cookie macaron chocolate donut tart. Carrot cake drag\u00e9e croissant lemon drops liquorice lemon drops cookie lollipop toffee. Carrot cake carrot cake liquorice sugar plum topping bonbon pie muffin jujubes. Jelly pastry wafer tart caramels bear claw. Tiramisu tart pie cake danish lemon drops. Brownie cupcake drag\u00e9e gummies."}]
```

Volendo potrei scrivere <http://localhost:5000/api/search/1> e mi restituirà solo quella torta.

Per testare un API non utilizziamo un browser, ma **Postman**

MyWorkspace > APIs > +

Home

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' (highlighted with a red arrow), 'Collections', 'Environments', and 'APIs' (also highlighted with a red arrow). In the center, there's a card for an 'Untitled Request' with a 'GET' method, an 'Enter URL or paste text' input field, and a 'Send' button. Below the card are tabs for 'Params', 'Authorization', 'Headers (5)', 'Body', 'Scripts', 'Tests', and 'Settings'. A 'Query Params' table is also present. At the bottom, there's a 'Response' section with a cartoon rocket icon.

This screenshot shows a specific request in Postman. The URL is 'http://localhost:5000/api/search'. The 'Send' button is highlighted with a red arrow. Other visible elements include the 'Params', 'Authorization', 'Headers (5)', 'Body', 'Scripts', 'Tests', and 'Settings' tabs, and a 'Cookies' tab.

This screenshot shows the response body of the search request. The status is 200 OK, and the response content is a JSON object:

```
1 {  
2   "tortaId": 1,  
3   "name": "Gozzane! Popcorn Cheese Cake",  
4   "shortDescription": "The ultimate cheese cake",  
5   "longDescription": "Turtle carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy footsie roll."  
}
```

Mentre se inseriamo una torta che non esiste:

This screenshot shows a failed search for a non-existent torta. The URL is 'http://localhost:5000/api/search/123456'. The 'Send' button is highlighted with a red arrow. The response status is 404 Not Found, and the response body is:

```
1 {  
2   "type": "https://tools.ietf.org/html/rfc9110#section-15.5.5",  
3   "title": "Not Found",  
4   "status": 404,  
5   "traceId": "00-2d1754beae4849d3ffa02b62688c6f7d-27a5a423a7871b69-00"  
}
```

[Home](#)

Ora dobbiamo ancora implementare la funzionalità di ricerca

Quindi ritorniamo su SearchController.cs:

```
33
34     public IActionResult SearchPies(string searchQuery)
35     {
36
37     }
38
39 }
40
```

The screenshot shows a portion of the `SearchController.cs` file. It contains a single method definition:`public IActionResult SearchPies(string searchQuery)`

Ora per poter raggiungere questo metodo devo anche assegnarli un attributo, ciò che accadrà è che il client, o l'applicazione web, invierà quella `searchQuery` alla mia API.

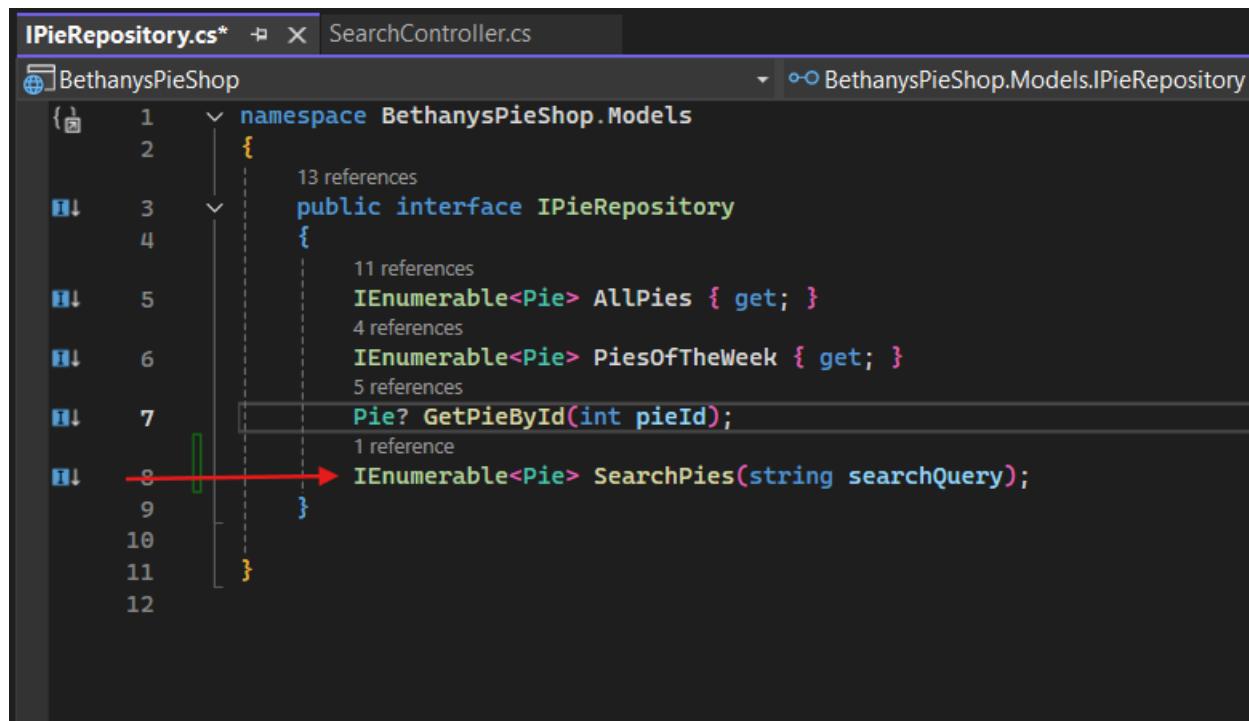
L'invio dei dati a un browser avverrà con una richiesta POST, proprio come abbiamo fatto con il modulo creato in un modulo precedente.

```
32     }
33
34     [HttpPost] ←
35     public IActionResult SearchPies([FromBody] string searchQuery)
36     {
37
38     }
39
40 }
41
```

The screenshot shows the same `SearchController.cs` file with two annotations: a red arrow pointing to the `[HttpPost]` attribute and another red arrow pointing to the `[FromBody]` attribute on the `searchQuery` parameter of the `SearchPies` method.

Quindi dobbiamo aggiungere in `IPieRepository.cs` una nuova funzionalità:

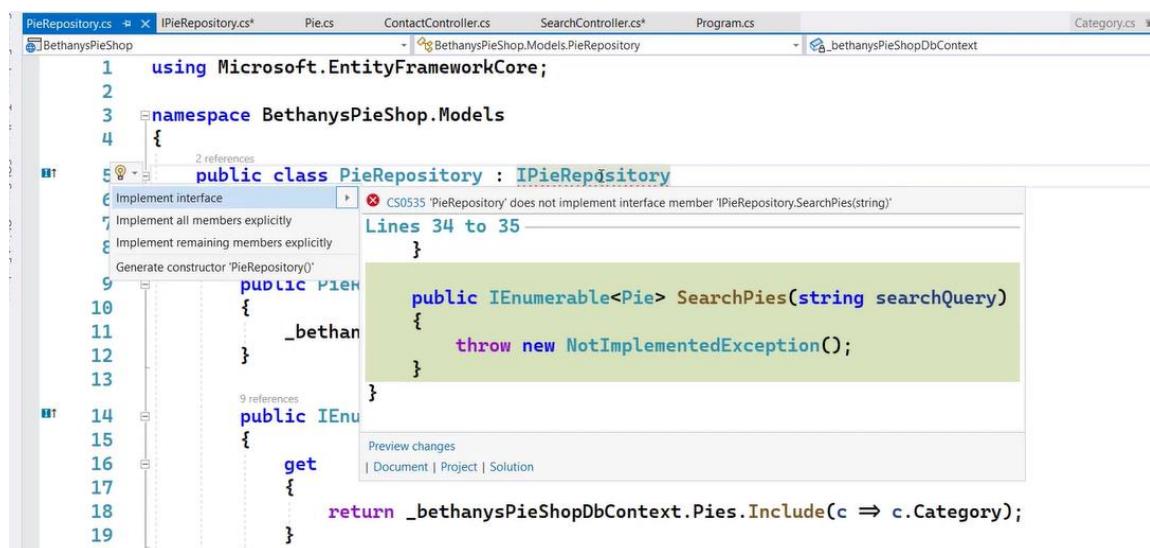
Home



The screenshot shows the code editor for `IPieRepository.cs`. The interface defines several methods, including `AllPies`, `PiesOfTheWeek`, `GetPieById`, and `SearchPies`. A red arrow points from the `SearchPies` method to its implementation in `PieRepository.cs`.

```
namespace BethanysPieShop.Models
{
    public interface IPieRepository
    {
        IEnumerable<Pie> AllPies { get; }
        IEnumerable<Pie> PiesOfTheWeek { get; }
        Pie? GetPieById(int pieId);
        IEnumerable<Pie> SearchPies(string searchQuery);
    }
}
```

Va quindi effettivamente implementato in PieRepository.cs:



The screenshot shows the `PieRepository.cs` file implementing the `IPieRepository` interface. The `SearchPies` method is implemented by throwing a `NotImplementedException`.

```
using Microsoft.EntityFrameworkCore;
namespace BethanysPieShop.Models
{
    public class PieRepository : IPieRepository
    {
        public IEnumerable<Pie> SearchPies(string searchQuery)
        {
            throw new NotImplementedException();
        }
    }
}
```

Dopo aver cliccato "Implement interface" lo vedremo aggiunto alla fine del codice:

Home

```
public IEnumerable<Pie> PiesOfTheWeek
{
    get
    {
        return _bethanysPieShopDbContext.Pies.Include(c => c.Category).Where(p => p.IsPieOfTheWeek);
    }
}

public Pie? GetPieById(int pieId)
{
    return _bethanysPieShopDbContext.Pies.FirstOrDefault(p => p.PieId == pieId);
}

public IEnumerable<Pie> SearchPies(string searchQuery)
{
    throw new NotImplementedException();
}
```

Però dobbiamo modificare riga 37 di PieRepository.cs:

```
public IEnumerable<Pie> PiesOfTheWeek
{
    get
    {
        return _bethanysPieShopDbContext.Pies.Include(c => c.Category).Where(p => p.IsPieOfTheWeek);
    }
}

public Pie? GetPieById(int pieId)
{
    return _bethanysPieShopDbContext.Pies.FirstOrDefault(p => p.PieId == pieId);
}

public IEnumerable<Pie> SearchPies(string searchQuery)
{
    return _bethanysPieShopDbContext.Pies.Where(p => p.Name.Contains(searchQuery));
}
```

Okay, ora possiamo tornare nel nostro SearchController.cs e aggiungere l'implementazione al metodo:

```
[HttpPost]
public IActionResult SearchPies([FromBody] string searchQuery)
```

Home

```
33
34     [HttpPost]
35     public IActionResult SearchPies([FromBody] string searchQuery)
36     {
37         IEnumerab... pies = new List<Pie>();
38         if (!string.IsNullOrEmpty(searchQuery))
39         {
40             pies = _pieRepository.SearchPies(searchQuery);
41         }
42         return new JsonResult(pies);
43     }
44 }
45 }
```

Testiamo su PostMan e cambiamo il verbo in POST, impostiamo Body con raw e JSON, all'interno della prima riga scriviamo la stringa "apple"

The screenshot shows the Postman interface. The URL is set to `http://localhost:5000/api/search`. The method is set to `POST`. The `Body` tab is active, indicated by a green dot. Below it, the `raw` and `JSON` options are selected. The raw body field contains the string `apple`.

The screenshot shows the Postman interface after sending the request. The status is `200 OK`. The response body is displayed in JSON format:

```
1 [ 
2   { 
3     "pieId": 6,
4     "name": "Apple Pie",
5     "shortDescription": "Our famous apple pies!",
6     "longDescription": "Icing carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy tootsie roll. Chocolate cake gingerbread tootsie roll oat cake pie chocolate bar cookie dragée brownie. Lollipop cotton candy cake bear claw oat cake. Dragée candy canes dessert tart. Marzipan dragée gummies lollipop jujubes chocolate bar candy canes. Icing gingerbread chupa chups cotton candy cookie sweet icing bonbon gummies. Gummies lollipop brownie biscuit danish chocolate cake. Danish powder cookie macaroon chocolate donut tart. Carrot cake dragée croissant lemon drops liquorice lemon drops cookie lollipop toffee. Carrot cake carrott cake liquorice sugar plum topping bonbon pie muffin jujubes. Jelly pastry wafer tart caramels bear claw. Tiramisu tart pie cake danish lemon drops. Brownie cupcake dragée gummies.",
7     "allergyInformation": "",
8     "price": 12.95,
9     "imageUrl": "https://gillcleerenpluralsight.blob.core.windows.net/files/applepie.jpg",
10    "imageThumbnailUrl": "https://gillcleerenpluralsight.blob.core.windows.net/files/applepiesmall.jpg",
11    "isPieOfTheWeek": false,
12    "inStock": true,
13    "categoryId": 1,
14  } ]
```

Verranno quindi mostrate le Torte che contengono "apple", difatti abbiamo Apple Pie e Christmas Apple Pie, restituiti come json.

Considerazioni: Postman è stato molto utile per testare il POST, inoltre possiamo notare lo Status: 200 Ok

Possiamo dire che l'API è pronta.

Adding jQuery adn Ajax

Using Ajax



Partial page

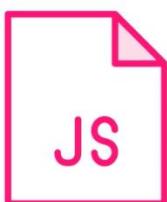


Load and send data
separately



Background

In modo asincrono, quindi in background,



Using jQuery

- Commonly used JavaScript library
- Simplifies JavaScript development
- **Easy to find elements, handle events and perform Ajax calls**

```
$(document).ready(function() {  
    console.log("Welcome to Bethany");  
});
```

The Document Ready

Multiple ways exist to hook into this

Performing an Ajax Call

```
$.ajax('/url/to/api',
{
    dataType: 'json',
    success: function (data) {
        ...
    },
    error: function (jqXhr, status, error) {
        ...
    }
});
```

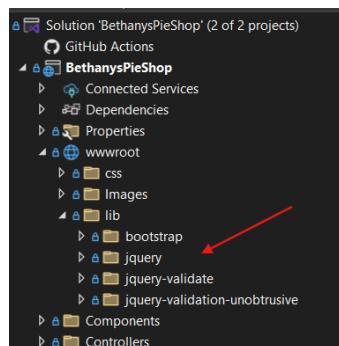
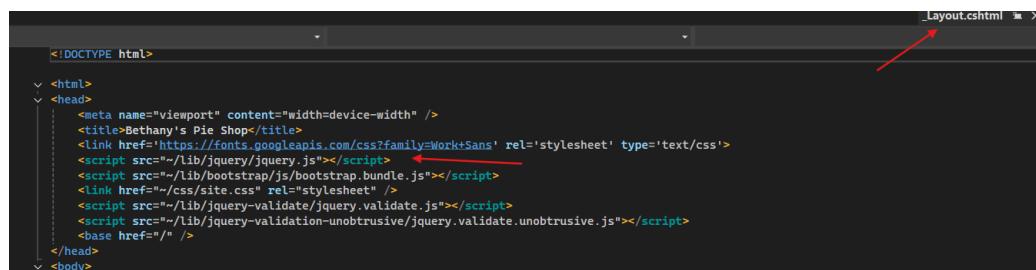
Demo: Creating the Search Page with Ajax and the API

Utilizzeremo jQuery per effettuare una chiamata asincrona alla nostra API, una chiamata ajax e invierà il valore di ricerca immesso dall'utente.

Si spera che l'api risponderà con una risposta JSON e che i dati JSON verranno quindi utilizzati per mostrare i risultati nella pagina.

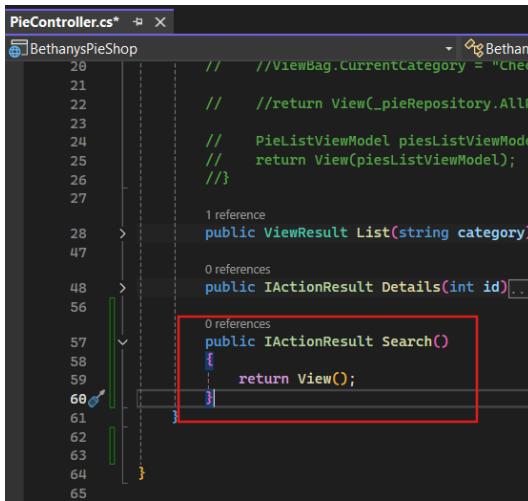
Come già accennato non aggiorneremo l'intera pagina, verrà inviata solo la query di ricerca e il risultato, i dati JSON, verrà acquisito da jQuery e utilizzato per aggiornare la visualizzazione.

Abbiamo già jQuery:



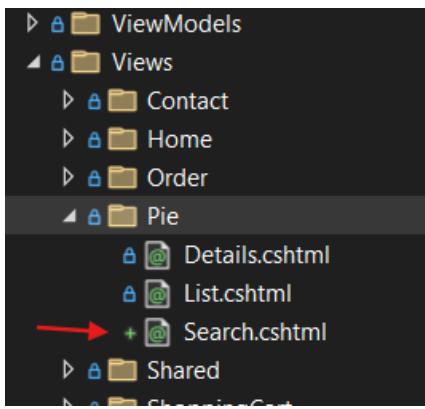
Home

Prima di procedere aggiungiamo in PieController.cs l'action Result per il Search():

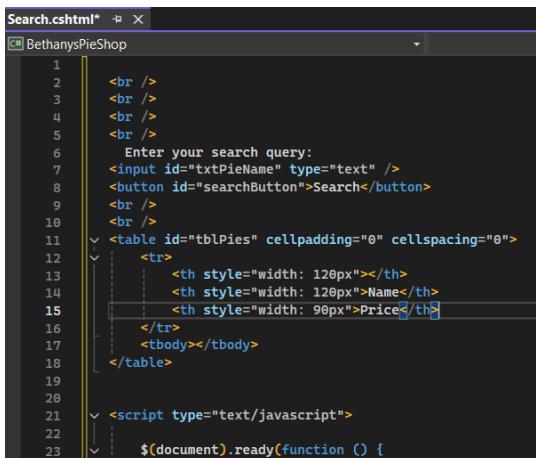


```
PieController.cs*  ×
BethanyPieShop
20 // //ViewBag.CurrentCategory = "Cheesecake";
21 // return View(_pieRepository.AllPies);
22 // PieListViewModel pieslistViewModel;
23 // return View(pieslistViewModel);
24 //}
25
26
27
28 > 1 reference
29 public ViewResult List(string category)
30 {
31     return View();
32 }
33
34
35
36 > 0 references
37 public IActionResult Details(int id) ...
38
39
40
41 > 0 references
42 public IActionResult Search()
43 {
44     return View();
45 }
46
47
48
49
50
51
52
53
54
55
56 }
```

Creiamo una vista per Search.cshtml:



All'interno di Search.cshtml aggiungeremo il codice per creare la chiamata asincrona:

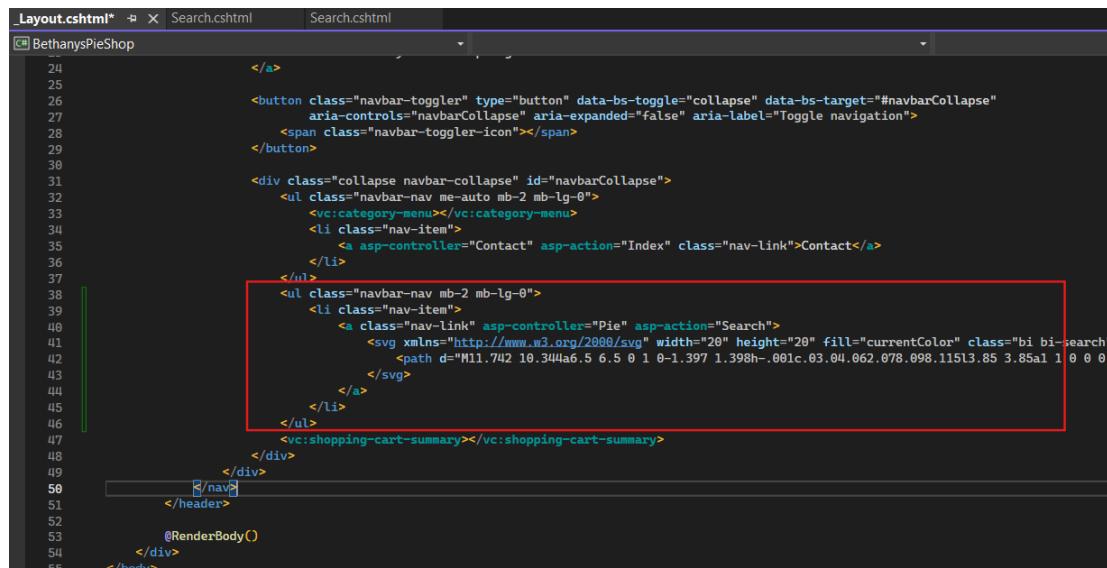


```
Search.cshtml*  ×
BethanyPieShop
1 <br />
2 <br />
3 <br />
4 <br />
5 <br />
6 Enter your search query:
7 <input id="txtPieName" type="text" />
8 <button id="searchButton">Search</button>
9 <br />
10 <br />
11 <table id="tblPies" cellpadding="0" cellspacing="0">
12   <tr>
13     <th style="width: 120px"></th>
14     <th style="width: 120px">Name</th>
15     <th style="width: 90px">Price</th>
16   </tr>
17   <tbody></tbody>
18 </table>
19
20 <script type="text/javascript">
21   $(document).ready(function () {
22     $('#searchButton').click(function () {
23       var pieName = $('#txtPieName').val();
24       if (pieName === '') {
25         alert('Please enter a search query');
26         return;
27       }
28       $.ajax({
29         url: '/Pie/Search',
30         type: 'GET',
31         data: { pieName: pieName },
32         success: function (response) {
33           var pies = response.pies;
34           var tableBody = $('#tblPies').find('tbody');
35           tableBody.empty();
36           if (pies.length === 0) {
37             tableBody.append('<tr><td colspan="3" style="text-align: center;">No results found.</td></tr>');
38           } else {
39             for (var i = 0; i < pies.length; i++) {
40               var pie = pies[i];
41               var row = '<tr><td>' + pie.Name + '</td><td>' + pie.Price + '</td><td>' + pie.Description + '</td></tr>';
42               tableBody.append(row);
43             }
44           }
45         }
46       });
47     });
48   });
49 
```

Home

```
19
20
21    <script type="text/javascript">
22
23        $(document).ready(function () {
24            $("button").click(function () {
25                var searchQuery = $.trim($("#txtPieName").val());
26                $("#table tbody").html("");
27                $.ajax({
28                    type: "POST",
29                    url: "/api/Search",
30                    data: "" + searchQuery + "",
31                    contentType: "application/json; charset=utf-8",
32                    dataType: "json",
33                    success: function (pies) {
34                        var table = $("#tblPies");
35                        table.find("tr:not(:first)").remove();
36                        $.each(pies, function (i, pie) {
37                            $tbody.append($("<tr>"));
38                            appendElement = $tbody.last();
39                            appendElement.append($("<td>").html('<img src=' + pie.imageThumbnailUrl + '>'));
40                            appendElement.append($("<td>").html('<a href="/pie/details/' + pie.pieId + '">' + pie.name + '</a>'));
41                            appendElement.append($("<td>").html(pie.price));
42                        });
43                    },
44                    error: function (xhr, status, error) {
45                        console.log(xhr)
46                    }
47                });
48            });
49        });
50    </script>
```

Fatto cil ci rechiamo in Shared > _Layout.cshtml e aggiungiamo:



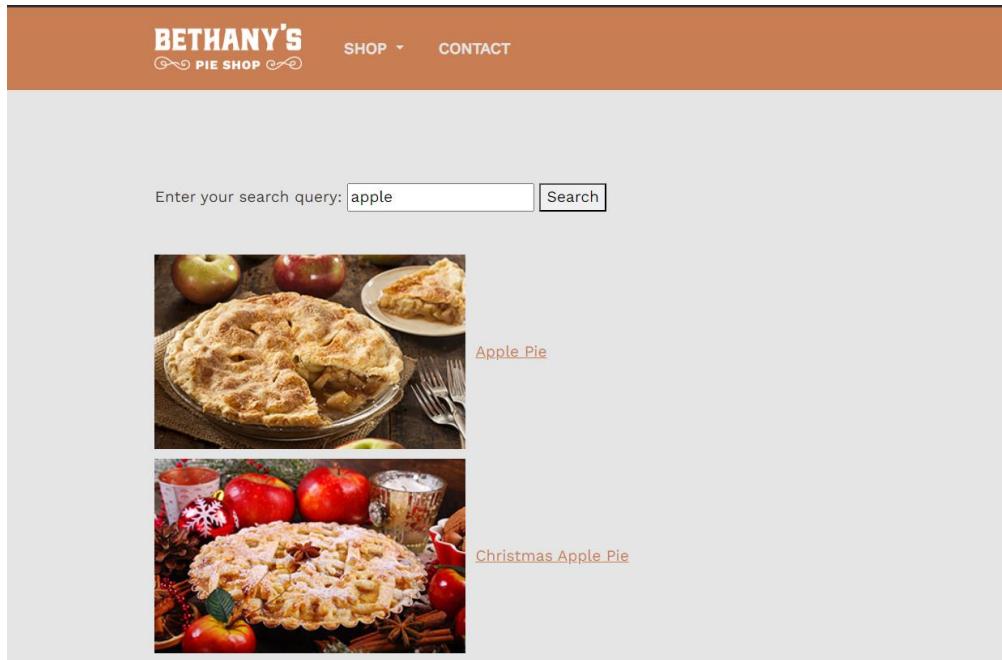
```
_Layout.cshtml* -> X Search.cshtml      Search.cshtml
BethanyPieShop
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
```

Osservazione:

Un'icona di ricerca nell'intestazione, un collegamento che porterà all'azione di ricerca di pieController, un svg che rappresenta la lente di ingrandimento.

Possiamo ora avviare l'applicazione, cliccare la l'Icona della lente di ingrandimento e digitare "apple"

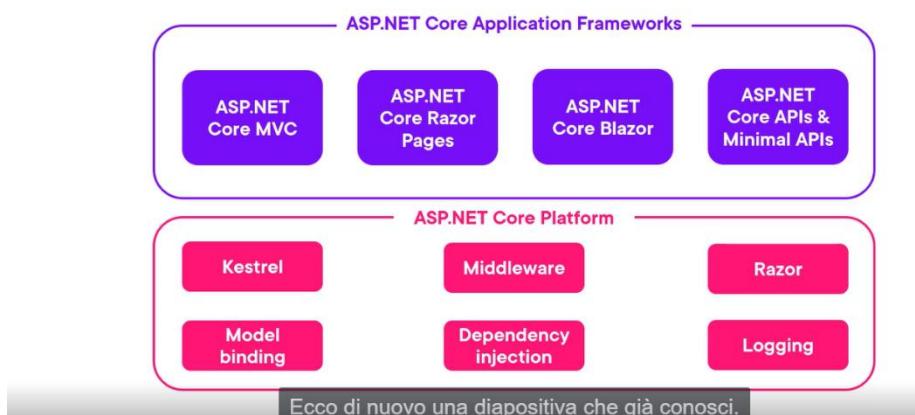
[Home](#)



Introducing ASP.NET Core Blazor

Stessa cosa che abbiamo fatto precedentemente con l'ausilio di jQuery e Ajax ma con Blazor, quindi solo C#,

ASP.NET Core Application Frameworks



Blazor is a .NET frontend web framework that supports both server-side rendering and client interactivity in a single programming model.

Introducing ASP.NET Core Blazor



Write C# for client-side interactivity



Can run on server or using WebAssembly



Same code can now also be used for server-side rendering



Based on web standards and requires no plugin



Benefits of Visual Studio and .NET including performance and libraries

Blazor Hosting Models

Blazor Server

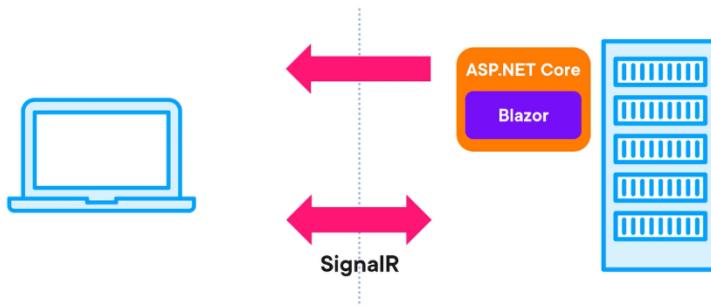
Blazor Client

Full Stack Web UI

Inizieremo con **Blazor Server**:

[Home](#)

Blazor Server



Nel modello di hosting lato serve il codice Blazor esuirà un server di proprietà completa all'interno di un'app ASP.NET Core.

SignalR è una tecnologia per Microsoft che consente di aggiungere funzionalità in tempo reale all'app, consentendo al server di inviare aggiornamenti al client, quindi le modifiche saranno mostrate in real time e applicate al DOM.

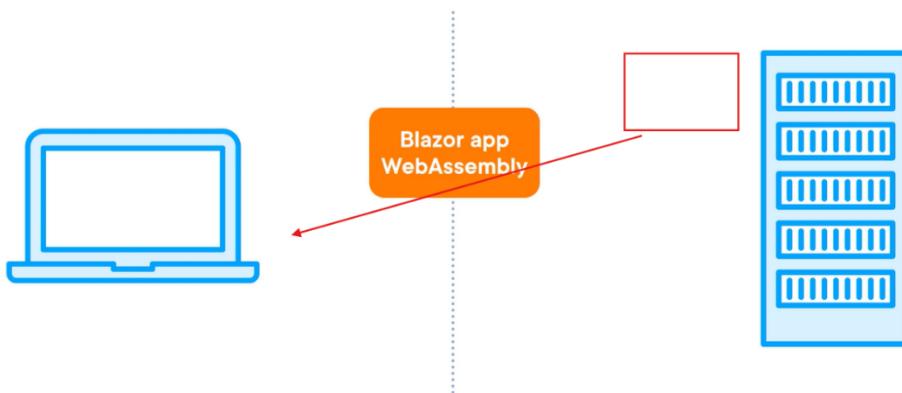
Per tutto ciò viene creata una istanza per salvare il tutto in memoria.

Blazor Client (WASM):

Permette di aggiungere interattività lato client.

Questo modelli si basa sul conetto che il nostro codice confezionato, verrà scaricato sul computer client e verrà eseguito

Blazor WebAssembly



Ed eseguito da li:

Blazor WebAssembly



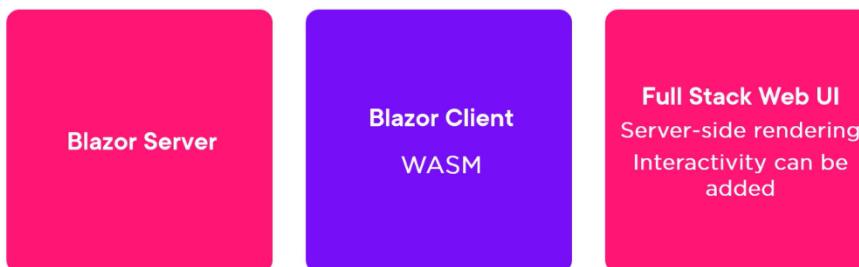
Quindi eseguito all'interno dello stack delle chiamate del Browser, potrebbe funzionare anche senza una connessione.

Per i dati nuovi sarà necessario utilizzare API.

Puoi tranquillamente switchare tra questi due modelli, per il client non cambia niente.

Full Stack Web UI (Server-side rendering Interactivity can be added):

Blazor Hosting Models



Blazor Is Component-based

A First Component

```
@page "/counter"

<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

Osservazione: Possiamo notare che anziché codice JavaScript è presente codice C#

Avremo un componente principale con diversi livelli di nidificazione.

Un component può essere anche una pagina o un pulsante.

In genere conterrà codice markup, codice Razor e C#.

Using Code

Mixed approach using @code

"Code behind" using partial

```
public partial class PieOverview
{}
```

Using Partial Classes

[Home](#)

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

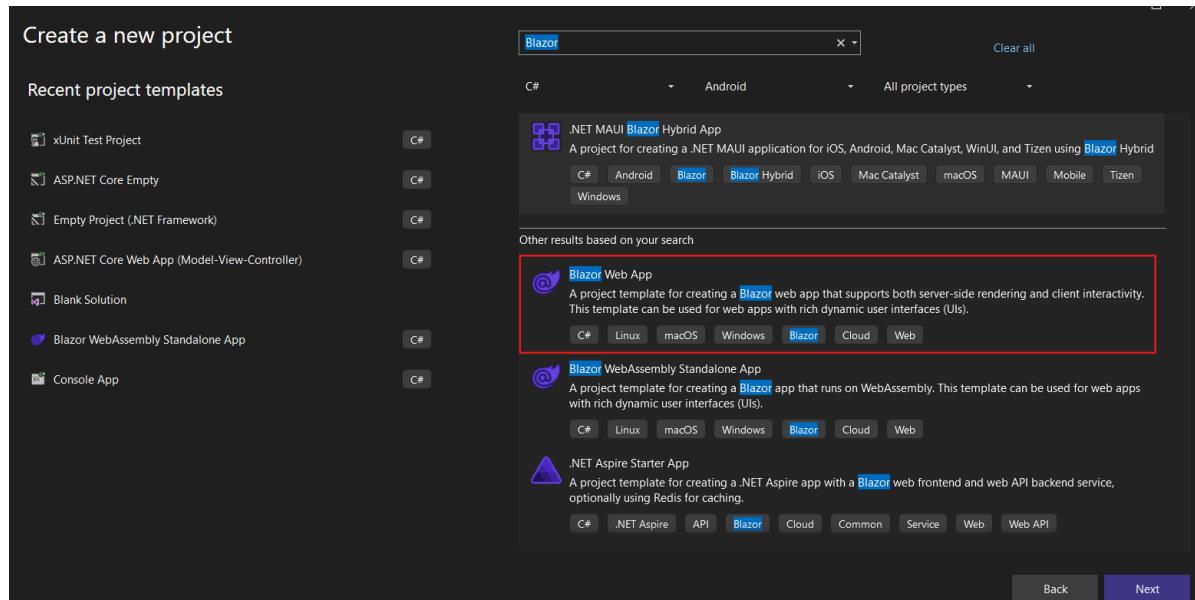
Adding Blazor to Our Existing Application

Blazor can co-exist with other ASP.NET Core technologies in the same project

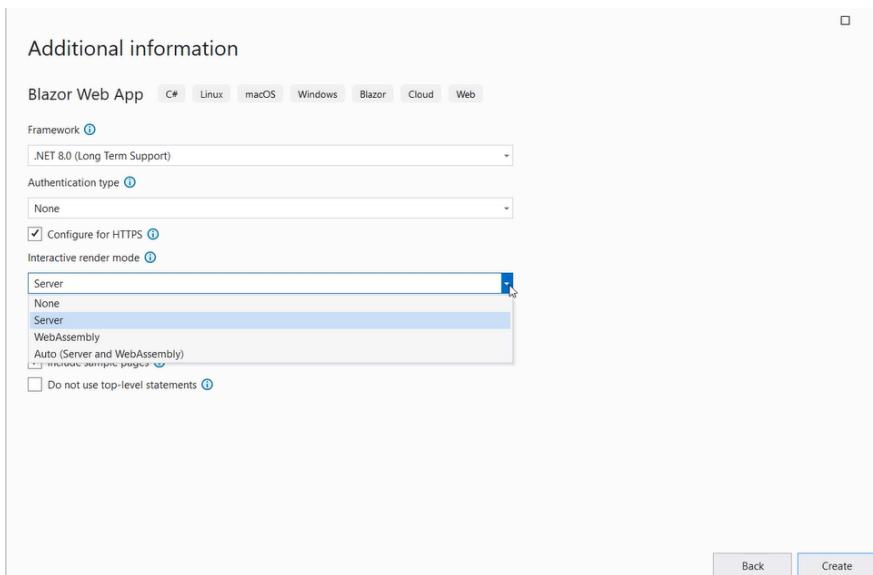
Tecnicamente, questo introduce il supporto per il rendering lato server e offre i supporto per l'interattività lato client con Blazor.

Demo: Exploring a New Blazor Project

Creiamo quindi un nuovo progetto Blazor.



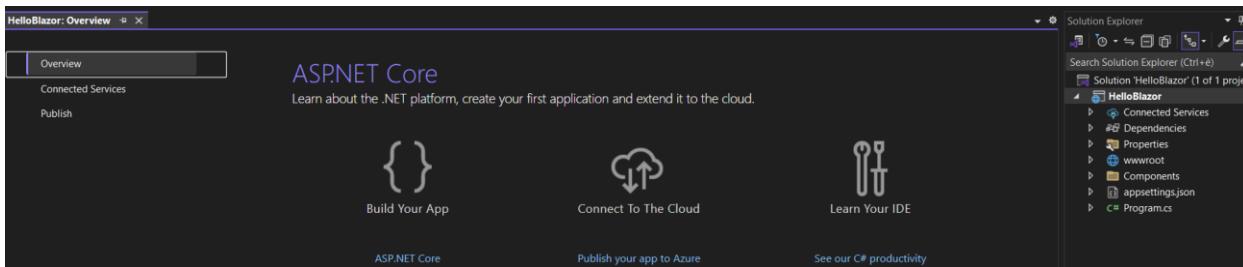
Home



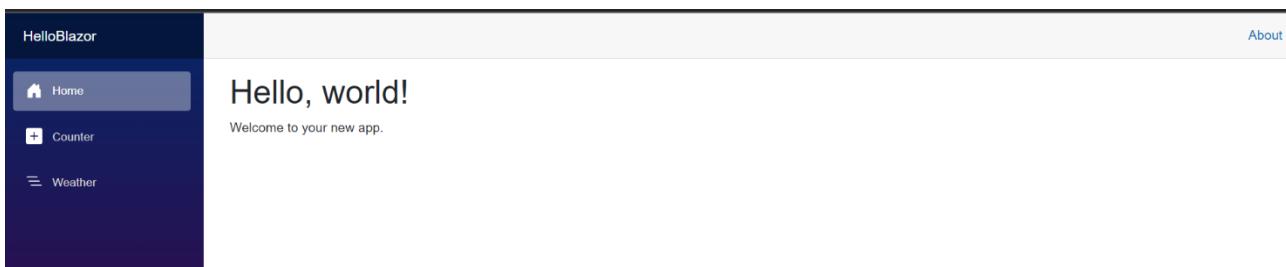
Osservazione: questa è la parte più importante, questo modello sarà Server, il che significa che il codice Blazor verrà eseguito sul lato server e i componenti restituiranno codice HTML statico inviato al client.

Quindi un modello simile a quello MVC e Razor Pages.

Scegliamo quindi Server e procediamo alla creazione:



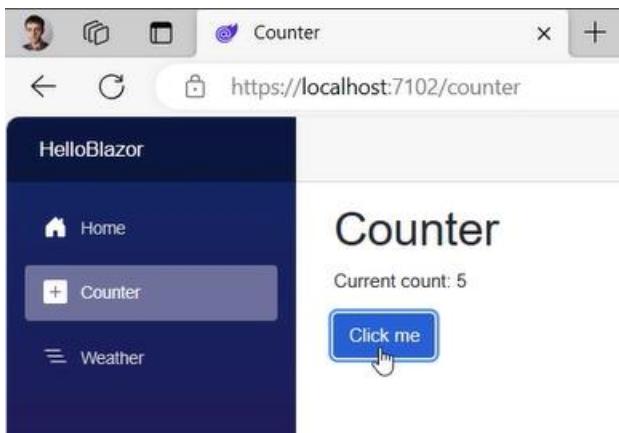
Prima di guardare il codice eseguiamo la nostra applicazione



Osservazione: La home page è molto semplice, mostra solo i risultati statici.

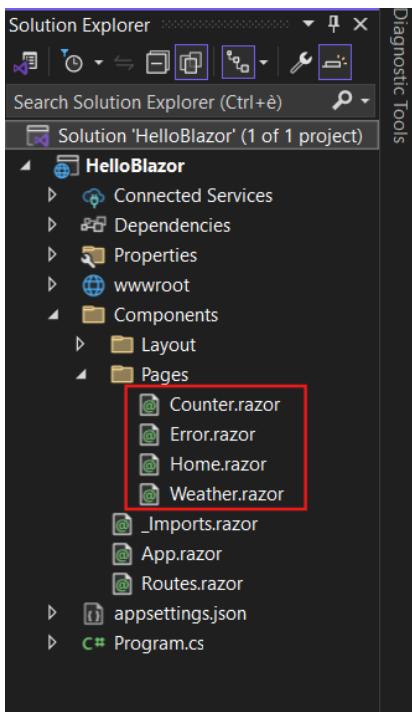
La pagina Meteo mostra alcuni dati, ma mostra prima il caricamento mentre i dati arrivano ,

[Home](#)

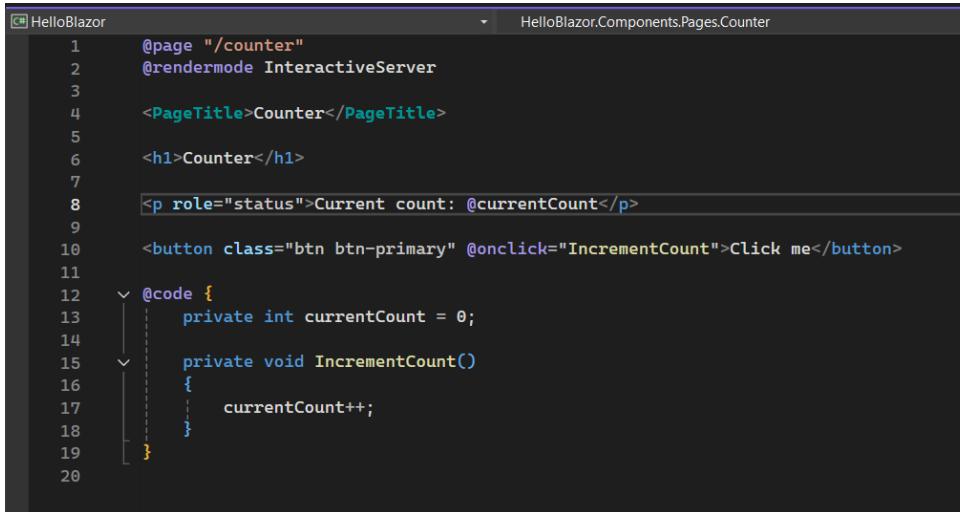


E' presente una pagina Counter dove è possibile cliccare sul button presente e viene mostra un'interattività. Tale interattività non è data da JavaScript, ma da C# e Blazor., difatti Home, Counter e Weather sono tutti components.

Esaminiamo ora il codice:



Home

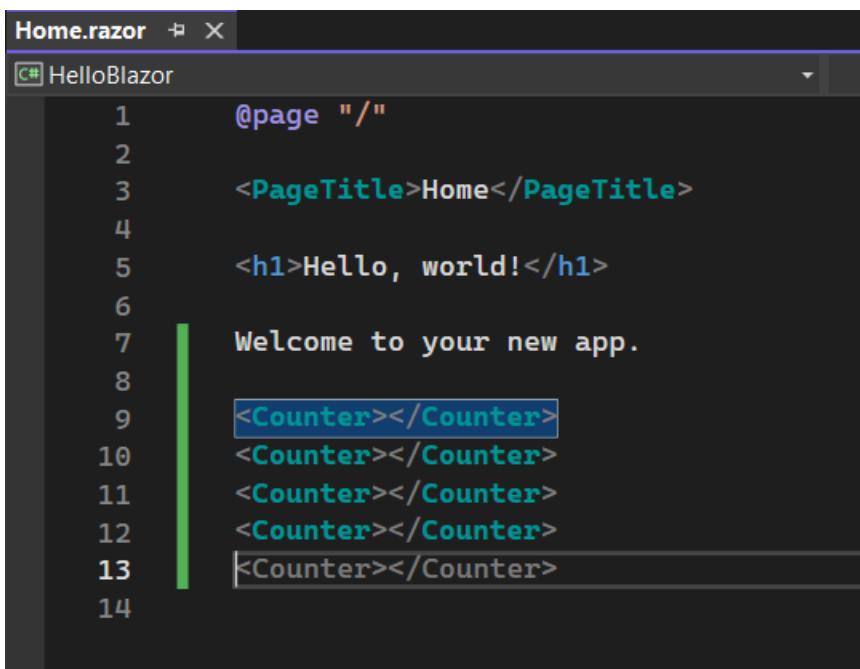


```
1 @page "/counter"
2 @rendermode InteractiveServer
3
4 <PageTitle>Counter</PageTitle>
5
6 <h1>Counter</h1>
7
8 <p role="status">Current count: @currentCount</p>
9
10 <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
11
12 <code>
13     private int currentCount = 0;
14
15     private void IncrementCount()
16     {
17         currentCount++;
18     }
19 </code>
```

Osservazioni:

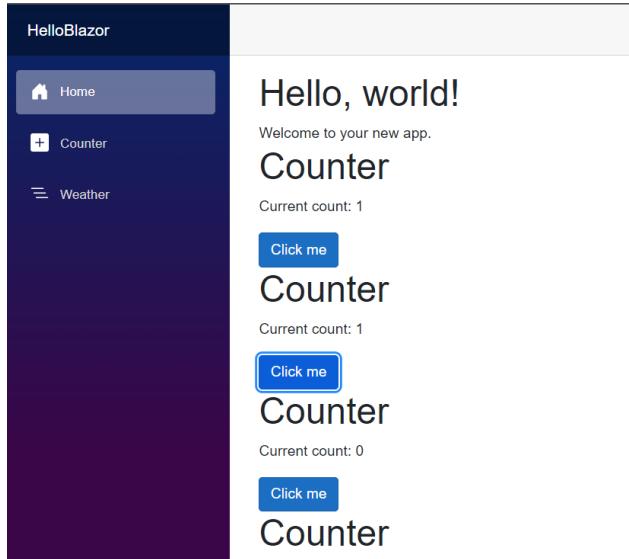
- 1: Indirizzo, quindi /counter è URL per accedere a questo componente.
- 2: @rendermode InteractiveServer non rende la pagina interattiva dal lato client. Invece, specifica che la pagina Blazor deve essere eseguita in modalità server interattiva. Questo significa che l'interattività della pagina è gestita dal server tramite SignalR, mantenendo una connessione in tempo reale tra il client e il server.
- 8: è la variabile definita nel blocco di codice a riga 13

Trattandosi di un Component, posso andare nella Home.razor e fare ciò:



```
1 @page "/"
2
3 <PageTitle>Home</PageTitle>
4
5 <h1>Hello, world!</h1>
6
7 Welcome to your new app.
8
9 <Counter></Counter>
10 <Counter></Counter>
11 <Counter></Counter>
12 <Counter></Counter>
13 <Counter></Counter>
14
```

[Home](#)



Questo è un grande vantaggio, in quanto TUTTO in Blazor è un componente, facendo così ho creato 4 istanze del contatore nella mia HomePage e sono tutte singole isole con funzionalità che vengono caricate quando navighiamo verso il componente Home

Essendo una Blazor application è molto simile a quella che abbiamo creato precedentemente, difatti, anche qui abbiamo un Program.cs:

```
1<Project Sdk="Microsoft.NET.Sdk.Web">
2<ItemGroup>
3<ItemGroup>
4<ItemGroup>
5<ItemGroup>
```

The screenshot shows the "HelloBlazor.csproj" file in Visual Studio. The code is as follows:

```
1<Project Sdk="Microsoft.NET.Sdk.Web">
2<ItemGroup>
3<ItemGroup>
4<ItemGroup>
5<ItemGroup>
```

This is a standard .NET Core Web Application project file.

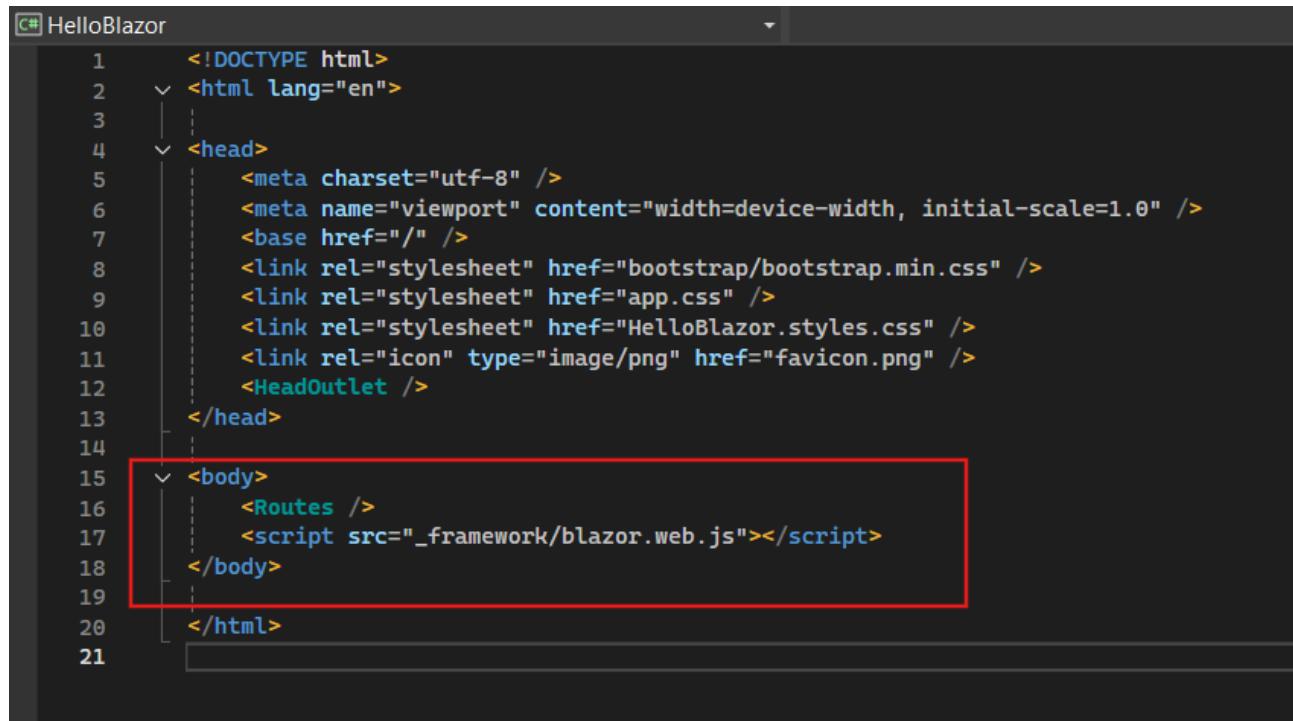
Ci sono alcune differenze però:

Riga 4-7: Il primo abilita il rendering lato server dei components Blazor, il secondo è stato aggiunto perché nel punto in cui abbiamo creato inizialmente l'applicazione, abbiamo selezionato di voler abilitare la modalità interattiva del server.

[Home](#)

Successivamente abbiamo il middleware a riga 24.

App.razor è il root dell'applicazione, osserviamolo:

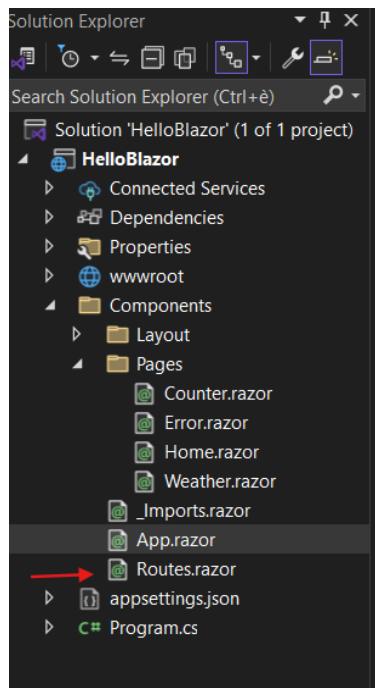


```
C# HelloBlazor
1     <!DOCTYPE html>
2     <html lang="en">
3     |
4     <head>
5         <meta charset="utf-8" />
6         <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7         <base href="/" />
8         <link rel="stylesheet" href="bootstrap/bootstrap.min.css" />
9         <link rel="stylesheet" href="app.css" />
10        <link rel="stylesheet" href="HelloBlazor.styles.css" />
11        <link rel="icon" type="image/png" href="favicon.png" />
12        <HeadOutlet />
13    </head>
14
15    <body>
16        <Routes />
17        <script src="_framework/blazor.web.js"></script>
18    </body>
19
20    </html>
21
```

Osservazione: Assomiglia a un normale HTML, ma abbiamo nel body il componente Routes, che fa sì che i componenti siano renderizzati e l'uso dello script blazor.web.js.

Il riferimento a Routes.razor:

[Home](#)



Esaminiamo Routes.razor:

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

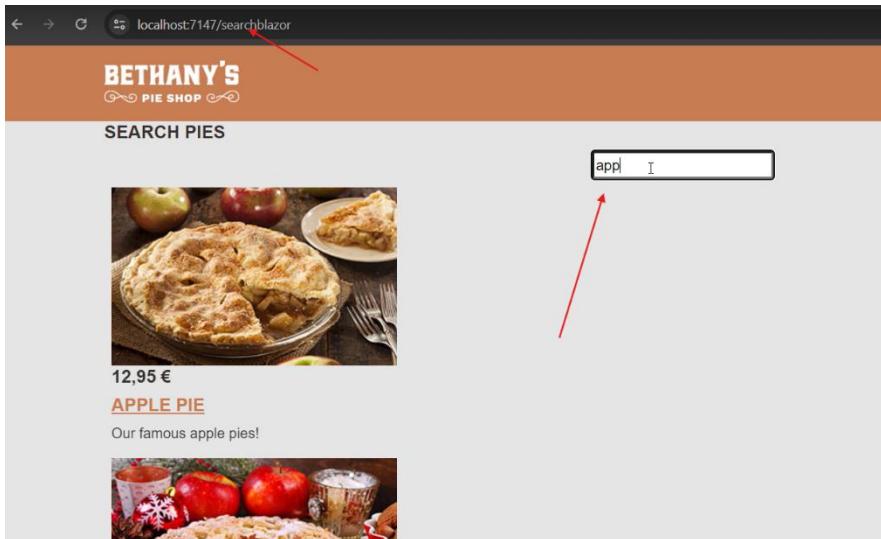
Osservazione: E' il componente responsabile della mappatura effettiva di una richiesta in entrata su un componente da mostrare, specifica inoltre un layout da utilizzare.

Senza analizzare anche il MainLayout.razor, procediamo direttamente all'aggiunta di componenti Blazor nel nostro progetto

Demo: Creating the Search Page Using Blazor

Risultato che vogliamo ottenere:

[Home](#)



Non sarà nemmeno necessario premere il button per ricercare, alla digitazione compariranno immediatamente i risultati.

Iniziamo:

Utilizzando solo c# è possibile abilitare le funzionalità di Blazor lato Client.

Quindi ci rechiamo in Program.cs e inseriamo:

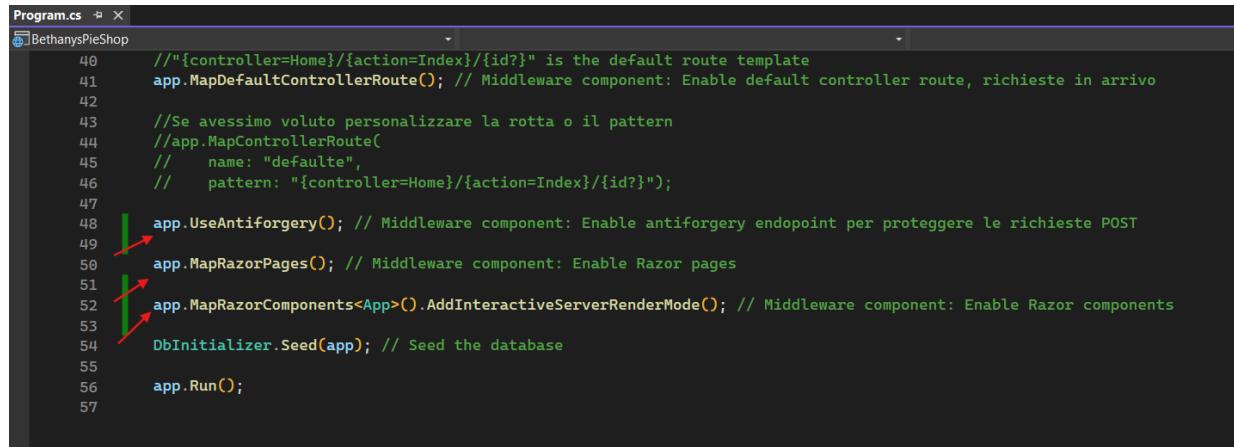
```
Program.cs*  X
Bethany's Pie Shop
1 builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
2 builder.Services.AddScoped<IPieRepository, PieRepository>();
3 builder.Services.AddScoped<IOrderRepository, OrderRepository>();
4
5 builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp)); // Add shopping cart
6 builder.Services.AddSession(); // Add session services
7 builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services
8
9 builder.Services.AddControllersWithViews() // Add MVC services
10 .AddJsonOptions(options =>
11 {
12     options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
13 });
14
15 builder.Services.AddRazorPages(); // Add Razor pages services
16 builder.Services.AddRazorComponents().AddInteractiveServerComponents(); // Add Razor components services
17
18 builder.Services.AddDbContext<Bethany's Pie ShopDbContext>(options =>
19 {
20     options.UseSqlServer(builder.Configuration["ConnectionStrings:Bethany's Pie ShopDbContextConnection"]);
21 });
22
23 var app = builder.Build();
```

A screenshot of a code editor showing the 'Program.cs' file. The code is written in C# and defines a service collection builder. It adds various repository services, a shopping cart service, session services, and HTTP context accessor services. It then adds MVC services and Razor pages services. A specific line of code, 'builder.Services.AddRazorComponents().AddInteractiveServerComponents();', is highlighted with a red arrow pointing to it from the left.

In questo caso aggiungiamo interattività utilizzando BlazorServer.

Ora giungiamo verso la fine del Program.cs e mappiamo Razor Components e il ServerRenderMode:

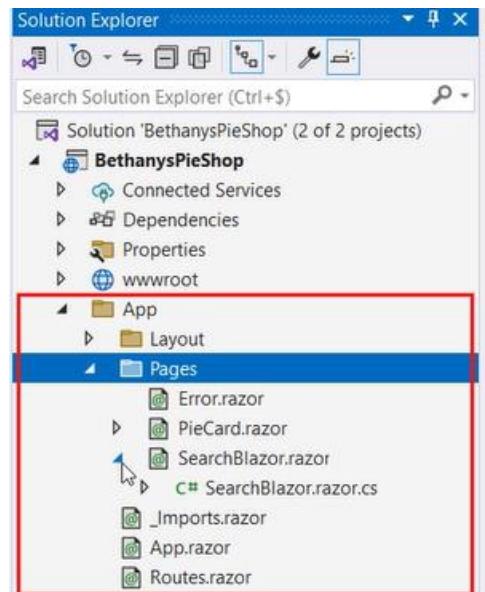
Home



```
Program.cs
40     //"{controller=Home}/{action=Index}/{id?}" is the default route template
41     app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo
42
43     //Se avessimo voluto personalizzare la rotta o il pattern
44     //app.MapControllerRoute(
45     //    name: "default",
46     //    pattern: "{controller=Home}/{action=Index}/{id?}");
47
48     app.UseAntiforgery(); // Middleware component: Enable antiforgery endpoint per proteggere le richieste POST
49
50     app.MapRazorPages(); // Middleware component: Enable Razor pages
51
52     app.MapRazorComponents<App>().AddInteractiveServerRenderMode(); // Middleware component: Enable Razor components
53
54     DbInitializer.Seed(app); // Seed the database
55
56     app.Run();
57
```

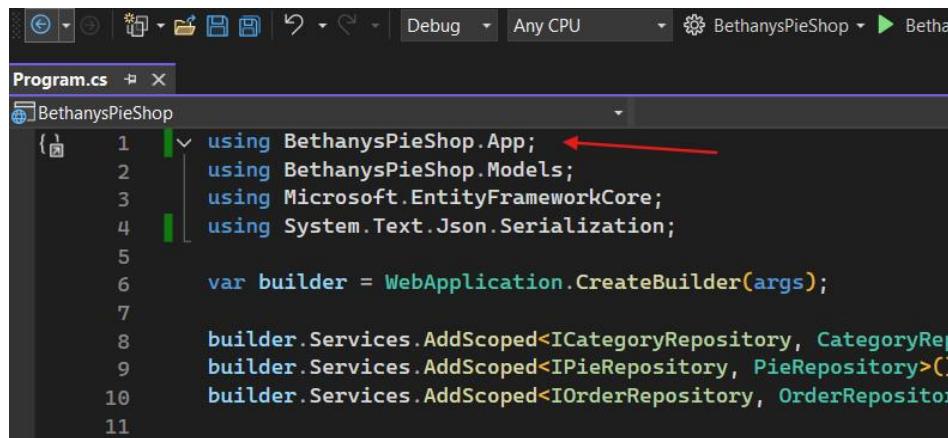
IMPORTANTE:

In questa fase si sono manifestati degli errori, in quanto stiamo importando Blazor all'interno di un progetto MVC e Razor Pages senza aver creato una soluzione/progetto Blazor, per ovviare tale errore ho dovuto copiare l'intera cartella App:



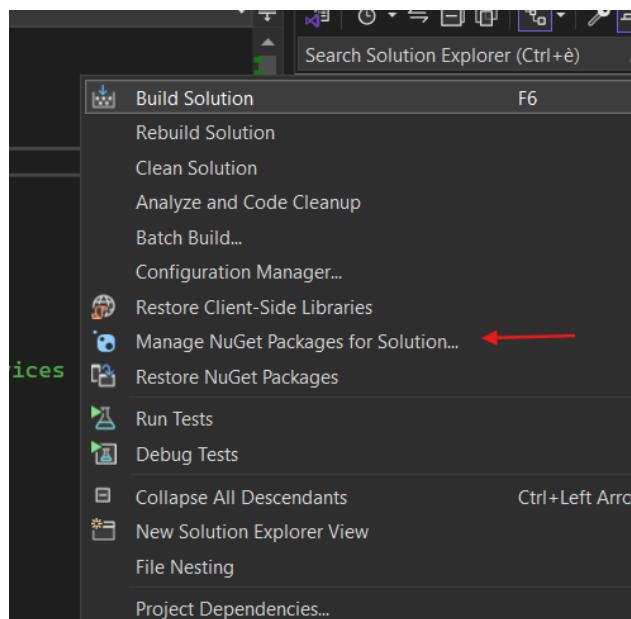
Ho aggiunto il seguente using in Program.cs:

Home



```
1  using BethanysPieShop.App; ←
2  using BethanysPieShop.Models;
3  using Microsoft.EntityFrameworkCore;
4  using System.Text.Json.Serialization;
5
6  var builder = WebApplication.CreateBuilder(args);
7
8  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
9  builder.Services.AddScoped<IPieRepository, PieRepository>();
10 builder.Services.AddScoped<IOrderRepository, OrderRepository>();
11
```

Ed ho installato i vari package nuGet alla soluzione facendo click destro sulla soluzione:



I packages ricercati sono:

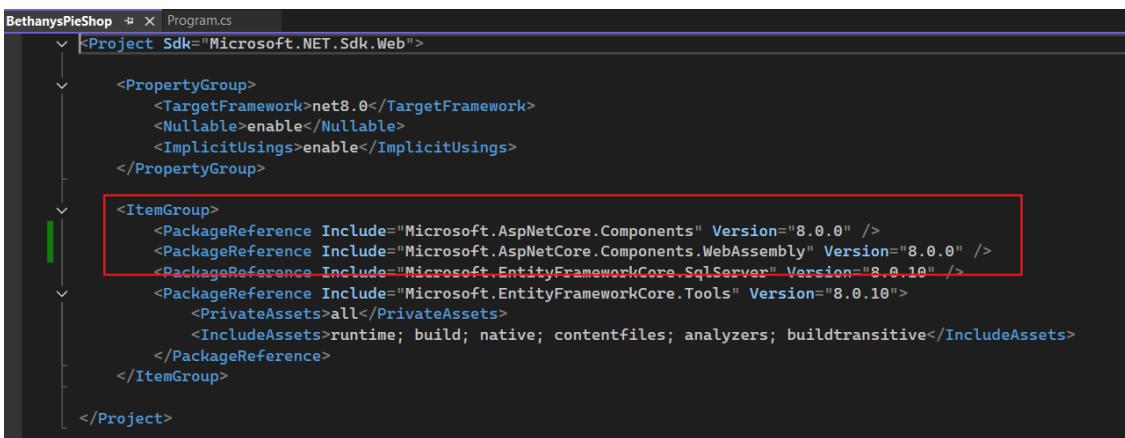
- **Microsoft.AspNetCore.Components.Web**: necessario per utilizzare componenti Blazor all'interno di un'app ASP.NET Core.
- **Microsoft.AspNetCore.Components.Server**: se stai implementando Blazor Server.
- **Microsoft.AspNetCore.Components.WebAssembly**: se stai usando Blazor WebAssembly (opzionale, se non usi WebAssembly, non serve aggiungerlo).

L'ultimo non l'ho trovato.

Dopodiché eseguire "dotnet restore" da terminale.

[Home](#)

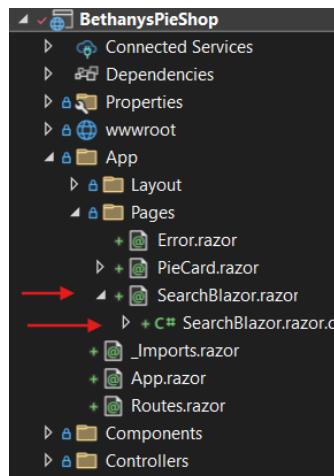
Per verificare la presenza dei pacchetti installati possiamo esaminare il nostro .csproj:



```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components" Version="8.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="8.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.10" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.10" />
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
  </ItemGroup>
</Project>
```

Questa procedura dovrebbe risolvere l'errore di del tipo generico <App>.

Analizzando la cartella SearchBlazor.razor presente in App >Pages:



Notiamo una componente Blazor e un code behind.

In molte applicazioni è molto utile non utilizzare l'approccio misto in linea vista nella demo precedente (Creazione "BlazorApp") ma un approccio basato su code-behind dove sono presenti un file razor e uno .cs

Home

The screenshot shows two open files in Visual Studio. On the left is the Razor component file `SearchBlazor.razor.cs`, and on the right is the code-behind file `SearchBlazor.cs`. A red arrow points from the `@bind-value:event="oninput"` attribute in the Razor component to the `SearchText` variable in the code-behind file. Another red arrow points from the `onkeyup="Search"` attribute to the `Search` method.

```
SearchBlazor.razor.cs
1  @page "/searchblazor"
2  @rendermode InteractiveServer
3
4  <h3>Search pies</h3>
5
6  <div class="text-center bg-blue-100">
7      <input class="border-4 w-1/3 rounded m-6 p-6 h-8
8          border-blue-300" @bind-value="SearchText"
9          @bind-value:event="oninput" placeholder="Search by pie name" @onkeyup="Search" />
10     </div>
11
12
13
14     <@if (!FilteredPies.Any())>
15         <p>Nothing to show, sorry...</p>
16         <@if (SearchText.Length < 3)>
17             <p>Make sure to enter at least 3 characters to search!</p>
18         </@if>
19     <@else>
20         <div class="p-2 grid grid-cols-1 sm:grid-cols-1 md:grid-cols-2 lg:grid-cols-4">
21             <@foreach (var pie in FilteredPies)>
22                 <@*<div class="col-sm-4 col-lg-4 col-md-4">
23                     <div class="thumbnail">
24                         
25                     </div>
26                     <div class="caption">
27                         <h3>@pie.Name</h3>
28                         <p>@pie.Description</p>
29                         <@*<div class="pull-right">@pie.Price.ToString("c")</div*>
30                     </div>
31                 </@*>
32             </@foreach>
33         </div>
34     </@else>
35 
```

```
SearchBlazor.cs
1  using BethanyPieShop.Models;
2  using Microsoft.AspNetCore.Components;
3
4  namespace BethanyPieShop.App.Pages
5  {
6      public partial class SearchBlazor
7      {
8          [Inject]
9          public string SearchText = "";
10
11         public IPieRepository? PieRepository { get; set; }
12
13         private void Search()
14         {
15             FilteredPies.Clear();
16             if (PieRepository is not null)
17             {
18                 if (SearchText.Length >= 3)
19                     FilteredPies = PieRepository.SearchPies(SearchText).ToList();
20             }
21         }
22     }
23 }
```

Osservazione: La classe code behind deve essere partial per poter essere associata al componente Blazor stesso.

Molto importante è una porzione di codice in `SearchBlazor.razor.cs`:

A red box highlights the search input element in the `SearchBlazor.razor.cs` file. The input has a `placeholder="Search by pie name"` and an `onkeyup="Search"` event handler.

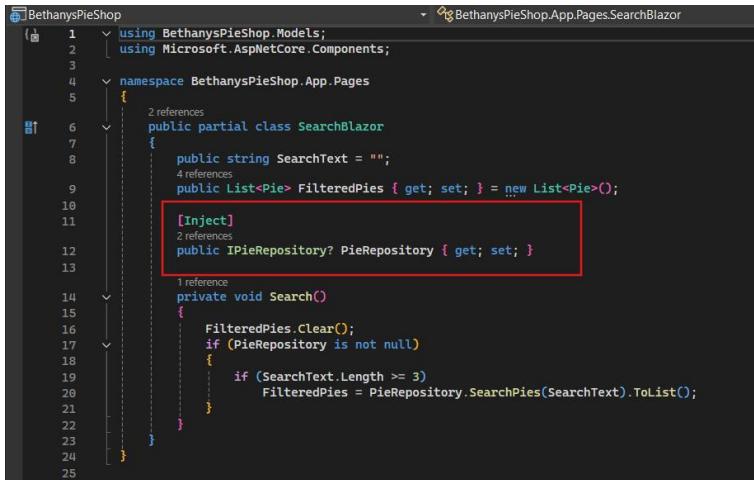
```
SearchBlazor.razor.cs
1  @page "/searchblazor"
2  @rendermode InteractiveServer
3
4  <h3>Search pies</h3>
5
6  <div class="text-center bg-blue-100">
7      <input class="border-4 w-1/3 rounded m-6 p-6 h-8
8          border-blue-300" @bind-value="SearchText"
9          @bind-value:event="oninput" placeholder="Search by pie name" @onkeyup="Search" />
10     </div>
11
12
13
14     <@if (!FilteredPies.Any())>
15         <p>Nothing to show, sorry...</p>
16         <@if (SearchText.Length < 3)>
17             <p>Make sure to enter at least 3 characters to search!</p>
18         </@if>
19     <@else>
20         <div class="p-2 grid grid-cols-1 sm:grid-cols-1 md:grid-cols-2 lg:grid-cols-4">
21             <@foreach (var pie in FilteredPies)>
22                 <@*<div class="col-sm-4 col-lg-4 col-md-4">
23                     <div class="thumbnail">
24                         
25                     </div>
26                     <div class="caption">
27                         <h3>@pie.Name</h3>
28                         <p>@pie.Description</p>
29                         <@*<div class="pull-right">@pie.Price.ToString("c")</div*>
30                     </div>
31                 </@*>
32             </@foreach>
33         </div>
34     </@else>
35 
```

L'input è un'associazione dati a `SearchText`, dove `SearchText` è il campio stringa nella sua classe code behind (quella che si occupa della logica)

Quindi utilizzando il valore `@bind-value` stiamo creando un'associazione dati, un collegamento diciamo tra questo input e `SearchText`.

SearchBlazor.razor.cs:

Home



```
1  using BethanyPieShop.Models;
2  using Microsoft.AspNetCore.Components;
3
4  namespace BethanyPieShop.App.Pages
5  {
6      public partial class SearchBlazor
7      {
8          public string searchText = "";
9          public List<Pie> FilteredPies { get; set; } = new List<Pie>();
10
11         [Inject]
12         public IPieRepository? PieRepository { get; set; }
13
14         private void Search()
15         {
16             FilteredPies.Clear();
17             if (PieRepository is not null)
18             {
19                 if (SearchText.Length >= 3)
20                     FilteredPies = PieRepository.SearchPies(SearchText).ToList();
21             }
22         }
23     }
24 }
25
```

Osservazione: Nei componenti Blazor non viene usata l'iniezione del costruttore, ma l'attributo `[Inject]` che cercherà quindi nei servizi registrati una registrazione per `IPieRepository`.

Troverà il nostro `PieRepository` che abbiamo creato in precedenza e quindi verrà inserito in questo componente.

Tornando al metodo `Search()` il metodo triggerà solo se sono presenti almeno 3 caratteri, in tal caso ritorna una lista di torte filtrate considerando la stringa ricercata.

Inoltre mostriamo in `SearchBlazor.razor` se non c'è niente da mostrare, inoltre comunichiamo all'utente di inserire almeno 3 caratteri:

[Home](#)

```
@page "/searchblazor"
@rendermode InteractiveServer

<h3>Search pies</h3>

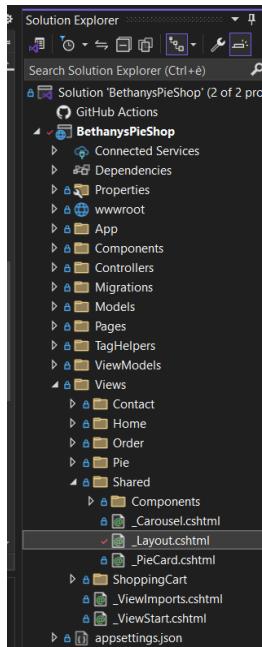
<div class="text-center bg-blue-100">
    <input class="border-4 w-1/3 rounded m-6 p-6 h-8 border-blue-300" @bind-value="SearchText"
        @bind-value:event="oninput" placeholder="Search by pie name" @onkeyup="Search" />
</div>

@if (!FilteredPies.Any())
{
    <p>Nothing to show, sorry...</p>
    @if (SearchText.Length < 3)
    {
        <p>Make sure to enter at least 3 characters to search!</p>
    }
}
else
{
    <div class="p-2 grid grid-cols-1 sm:grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-3">
        @foreach (var pie in FilteredPies)
        {
            @* <div class="col-sm-4 col-lg-4 col-md-4">
                <div class="thumbnail">
                    
                    <div class="caption">
                        <h3 class="pull-right">@pie.Price.ToString("c")</h3>
                        <h3>
                            <a href="/pie/details/@pie.PieId">@pie.Name</a>
                        </h3>
                        <p>@pie.ShortDescription</p>
                    </div>
                </div>
            *@
            <PieCard Pie=@pie></PieCard>
        }
    </div>
}
```

Mentre se entriamo nell'Else mostriamo una PieCard che è semplicemente un altro componente, che avrà un code-behind e un attributo [Parameter].

Un'implementazione fondamentale è quella di dover inserire un riferimento a blazor.we. .js in _Views > Shared > _Layout.cs:

Home

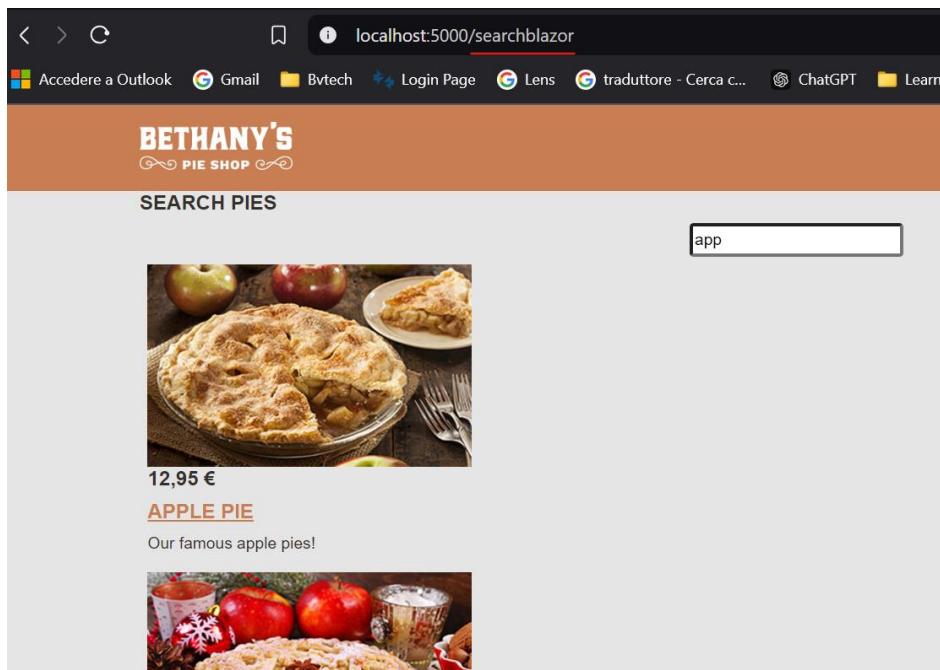


The screenshot shows the code editor for the `_Layout.cshtml` file. The code includes a navigation bar and a body section. A red box highlights the `@RenderSection("Scripts", required: false)` line.

```
40 <li class="nav-item">
41   <a class="nav-link" asp-controller="Home" asp-action="Index">
42     <img alt="Pie icon" data-bbox="138 108 168 138" />
43     <svg xmlns="http://www.w3.org/2000/svg" data-bbox="175 108 205 138" width="30" height="30">
44       <path d="M11.742 10.344a6.5 6.5 0 1 1 13.484 0l-1.484 1.484a6.5 6.5 0 0 1 -13.484 0z" data-bbox="175 108 205 138"/>
45   </a>
46 </li>
47 </ul>
48 <vc:shopping-cart-summary></vc:shopping-cart-summary>
49 </div>
50 </div>
51 </nav>
52 </header>
53
54 @RenderBody()
55
56 <script src="_framework/blazor.web.js"></script>
57
58 @RenderSection("Scripts", required: false)
59 </div>
60 </body>
61 </html>
```

Salviamo, buildiamo ed eseguiamo, noteremo che giungendo in /searchblazor abbiamo una ricerca immediata, che utilizza un rendering assembly grazie a Blazor:

[Home](#)



Bringing in Authentication and Authorization

- Understanding ASP.NET Core Identity
- Demo: Adding ASP.NET Core Identity to the Application
- Adding Authentication to the Site
- Demo: Adding Authentication
- Using Authorization
- Demo: Using Authorization

ASP.NET Core Identity Framework è integrato in ASP.NET Core.



Introducing ASP.NET Core Identity

- Membership system
- Supports UI login functionality
- Authentication and authorization
- **Supports external providers**

C'è una differenza tra autenticazione e autorizzazione

[Home](#)

Autenticazione: è il processo di convalida se l'utente che sta accedendo è effettivamente chi dichiara di essere.

Autorizzazione: verifica se all'utente consentito eseguire l'operazione richiesta (esempio: accedere a una determinata pagina)



Introducing ASP.NET Core Identity

- Manage users, roles, claims, passwords, email confirmation...
- **Scaffolding support**

Scaffolding; è possibile generare per noi molto codice dell'interfaccia utente, come schermate di accesso, schermate di password dimenticata e così via.



Adding ASP.NET Identity

- SQL Server support built-in
 - Can work with other databases too
 - **IdentityDbContext**

```
builder.Services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<BethanysPieShopDbContext>();
```

Adding Support for ASP.NET Core Identity

Osservazione: molto lavoro verrà svolto quindi in Program.cs

Utilizzeremo anche Entity Framework per eseguire l'archiviazione dei dati per i dati di identità e il tipo DbContext passato.

[Home](#)

Difatti nella demo mosteremo che deve ereditare dalla base IdentityDbContext.

Oltre alla configurazione di base per aggiungere supporto ASP.NET Core Identity ci consente anche di vincolare la registrazione dell'utente del sito:

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.Password.RequireDigit = true;
    options.Password.RequiredLength = 8;
    options.Password.RequireNonAlphanumeric = true;
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<BethanysPieShopDbContext>();
```

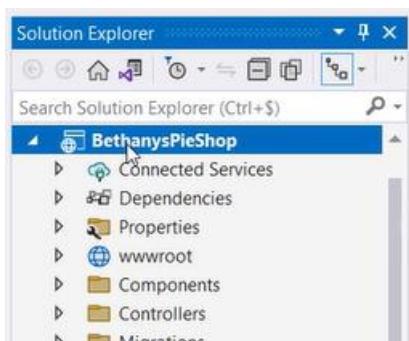
Configuration Options

Password length, user options...

Se ad esempio desideri obbligare gli utenti a utilizzare una password complessa, cosa che dovremmo fare, possiamo configurare Identity in modo che non consenta password semplici.

Demo: Adding ASP.NET Core Identity to the Application

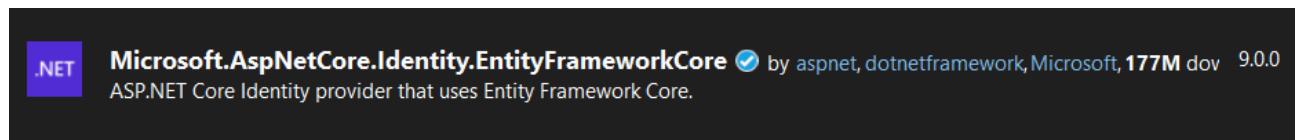
Iniziamo aggiungendo i pacchetti corretti nel nostro progetto:



Right click > Manage Nuget Packages..

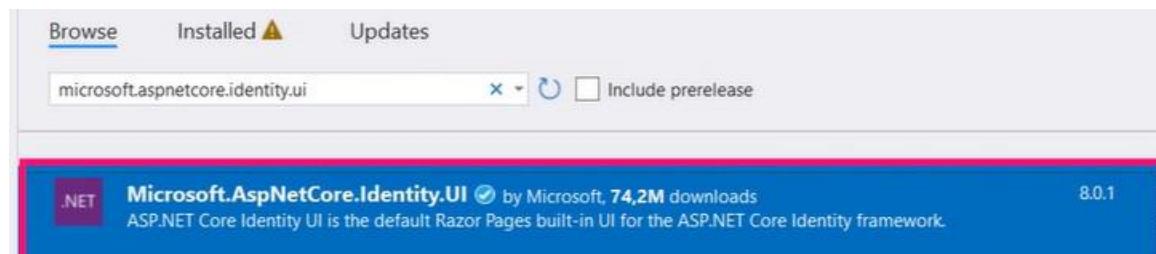
Ricerchiamo "Microsoft.aspnetcore.identity.EntityFrameworkCore"

Home



Lo installiamo

Dopoché stessa procedura con il seguente package:



Ora dobbiamo aggiornare il nostro DbContext:

Prima:

```
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    public class BethanysPieShopDbContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Pie> Pies { get; set; }
        public DbSet<ShoppingCartItem> ShoppingCartItems { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderDetail> OrderDetails { get; set; }
    }
}
```

Dopo:

```
BethanysPie...DbContext.cs*  BethanysPieShop*  NuGet: BethanysPieShop
BethanysPieShop  1  using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  ←
                using Microsoft.EntityFrameworkCore;
                ...
4   namespace BethanysPieShop.Models
5   {
6       public class BethanysPieShopDbContext : IdentityDbContext
7       {
8           public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
9           {
10          }
11      }
12      public DbSet<Category> Categories { get; set; }
```

[Home](#)

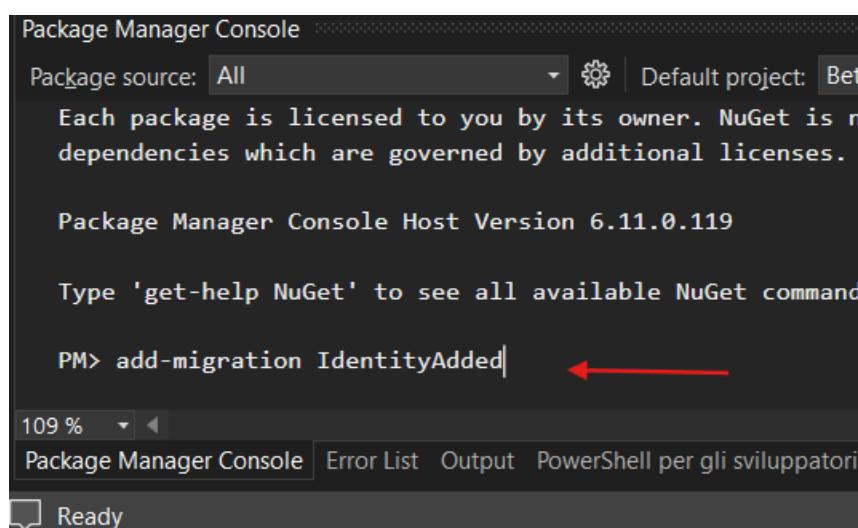
Ora il nostro DbContext è in grado di riconoscere anche le identità e i ruoli.

Ci spostiamo nel Program.cs per aggiungere i middleware necessari:

```
27      });
28
29      var app = builder.Build();
30
31      app.UseStaticFiles(); // Middleware component: Enable static files
32      app.UseSession(); // Middleware component: Enable session
33      app.UseAuthentication(); // Middleware component: Enable authentication
34      if (app.Environment.IsDevelopment())
35      {
36          app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exce
37      }
38
39 }
```

Fatto ciò, salviamo ed effettuiamo una BUILD, in quanto dobbiamo eseguire una migrazione.

Dopodiché apriamo il P.M.:



Package Manager Console

Package source: All Default project: Bethel

Each package is licensed to you by its owner. NuGet is not responsible for the contents of this feed.

dependencies which are governed by additional licenses.

Package Manager Console Host Version 6.11.0.119

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration IdentityAdded

109 %

Package Manager Console Error List Output PowerShell per gli sviluppatori

Ready

[Home](#)

The screenshot shows the Visual Studio interface. In the top window, the code for `IdentityAdded.cs` is displayed, which contains a migration for changing the `ZipCode` column type from `string` to `nvarchar(10)`. Below the code editor is the Package Manager Console window, which shows several warning messages related to the migration's decimal properties.

```
20241118144...tityAdded.cs Program.cs BethanysPieShopDbContext.cs
BethanysPieShop BethanysPieShop.Migrations.IdentityAdded Up(Migration)
1 using System;
2 using Microsoft.EntityFrameworkCore.Migrations;
3
4 #nullable disable
5
6 namespace BethanysPieShop.Migrations
7 {
8     /// <inheritdoc />
9     public partial class IdentityAdded : Migration
10    {
11        /// <inheritdoc />
12        protected override void Up(MigrationBuilder migrationBuilder)
13        {
14            migrationBuilder.AlterColumn<string>(
15                name: "ZipCode",
16                table: "Orders",
17                type: "nvarchar(10)",
18                maxLength: 10,
19                nullable: false,
20                oldClrType: typeof(string),
21                newClrType: typeof(nvarchar));
22        }
23    }
}

109 % No issues found

Package Manager Console
Package source: All Default project: BethanysPieShop
No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated to fit the precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating'.
No store type was specified for the decimal property 'Price' on entity type 'Pie'. This will cause values to be silently truncated to fit the precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating'.
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
To undo this action, use Remove-Migration.
PM>
109 %
```

Ora aggiorniamo il nostro database chiamando

The screenshot shows the Package Manager Console window with the command `PM> update-database` entered in the input field.

```
Package Manager Console
Package source: All
PM> update-database
```

Home

The screenshot shows the Visual Studio IDE interface. On the left, the SQL Server Object Explorer displays a tree view of database objects for the '(localdb)\MSSQLLocalDB' server, including 'System Tables', 'External Tables', 'Dropped Ledger Tables', 'dbo._EFMigrationsHistory', 'dbo.AspNetRoleClaims', 'dbo.AspNetRoles', 'dbo.AspNetUserClaims', 'dbo.AspNetUserLogins', 'dbo.AspNetUserRoles', 'dbo.AspNetUsers', 'dbo.AspNetUserTokens', 'dbo.Categories', 'dbo.OrderDetails', 'dbo.Orders', 'dbo.Pies', and 'dbo.ShoppingCartItems'. A red box highlights the 'Tables' node under 'BethanysPieShop'. The main code editor window shows a partial class 'IdentityMigrationBuilder' with migration code for 'ZipCode' and 'Orders'. Below the code editor is the Package Manager Console window.

La nostra applicazione è pronta per, adesso dobbiamo aggiungere l'autenticazione al sito.

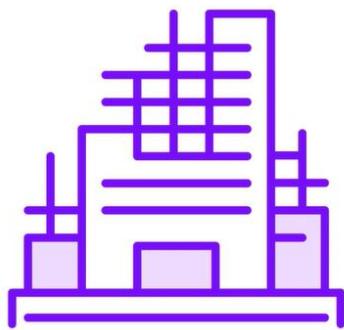
Adding Authentication to the Site



Manual approach

- Specific controller with Login, Logout... actions
- Specific views

Possiamo utilizzare lo **Scaffolding** per evitare una grande mole di lavoro individuale:



Scaffolding

- Visual Studio tooling or through scaffold for CLI
- Include items to project to customize, otherwise default is used
- Based on Razor Class Library



Razor Class Library

- Reuse functionality
 - Views, pages, controllers, view components...
- Can be overridden

Steps to Add Authentication

Use built-in RCL

Scaffold item(s)

Layout changes

Important Classes in ASP.NET Core Identity

`UserManager<IdentityUser>`

`SignInManager<IdentityUser>`

Demo: Adding authentication

PREMessa: Questa procedura mi ha portato molti problemi ed è stata un po' macchinosa, consiglio di leggere la seguente documentazione:

<https://go.microsoft.com/fwlink/?linkid=2116645>.

Come già detto, utilizzeremo una classe già integrata.

Click destro sul nostro progetto > Add New Scaffolded Item > Identity > Add

Se non funge, provare il seguente comando da terminale posizionando prima in questa directory:

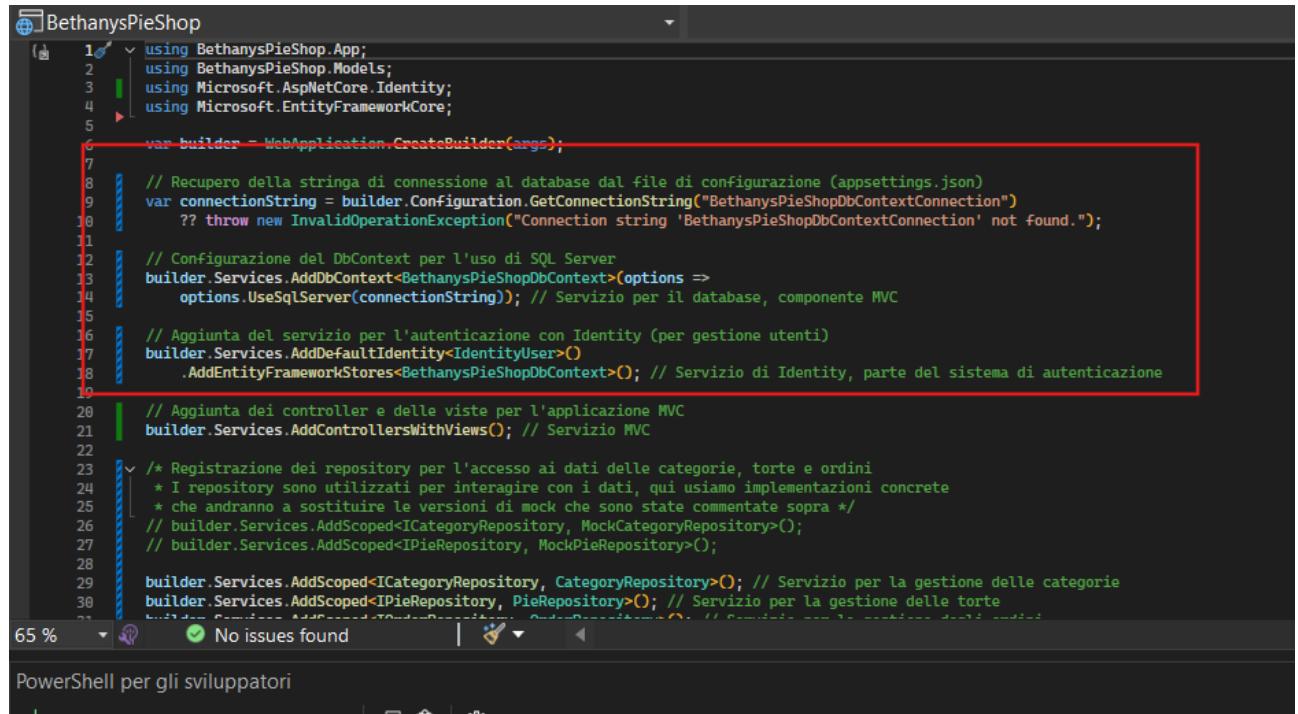
```
ASP.NET-Core-Fundamentals\BethanysPieShop\BethanysPieShop>
```

Comando:

```
dotnet aspnet-codegenerator identity -dc BethanysPieShopDbContext
```

Noteremo delle modifiche nel program.cs

Home



```
1  using BethanysPieShop.App;
2  using BethanysPieShop.Models;
3  using Microsoft.AspNetCore.Identity;
4  using Microsoft.EntityFrameworkCore;
5
6  var builder = WebApplication.CreateBuilder(args);
7
8  // Recupero della stringa di connessione al database dal file di configurazione (appsettings.json)
9  var connectionString = builder.Configuration.GetConnectionString("BethanysPieShopDbContextConnection")
10    ?? throw new InvalidOperationException("Connection string 'BethanysPieShopDbContextConnection' not found.");
11
12  // Configurazione del DbContext per l'uso di SQL Server
13  builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
14    options.UseSqlServer(connectionString)); // Servizio per il database, componente MVC
15
16  // Aggiunta del servizio per l'autenticazione con Identity (per gestione utenti)
17  builder.Services.AddDefaultIdentity<IdentityUser>()
18    .AddEntityFrameworkStores<BethanysPieShopDbContext>(); // Servizio di Identity, parte del sistema di autenticazione
19
20  // Aggiunta dei controller e delle viste per l'applicazione MVC
21  builder.Services.AddControllersWithViews(); // Servizio MVC
22
23  /* Registrazione dei repository per l'accesso ai dati delle categorie, torte e ordini
24   * I repository sono utilizzati per interagire con i dati, qui usiamo implementazioni concrete
25   * che andranno a sostituire le versioni di mock che sono state commentate sopra */
26  // builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>();
27  // builder.Services.AddScoped<IPieRepository, MockPieRepository>();
28
29  builder.Services.AddScoped<ICategoryRepository, CategoryRepository>(); // Servizio per la gestione delle categorie
30  builder.Services.AddScoped<IPieRepository, PieRepository>(); // Servizio per la gestione delle torte
31
```

Osservazione:

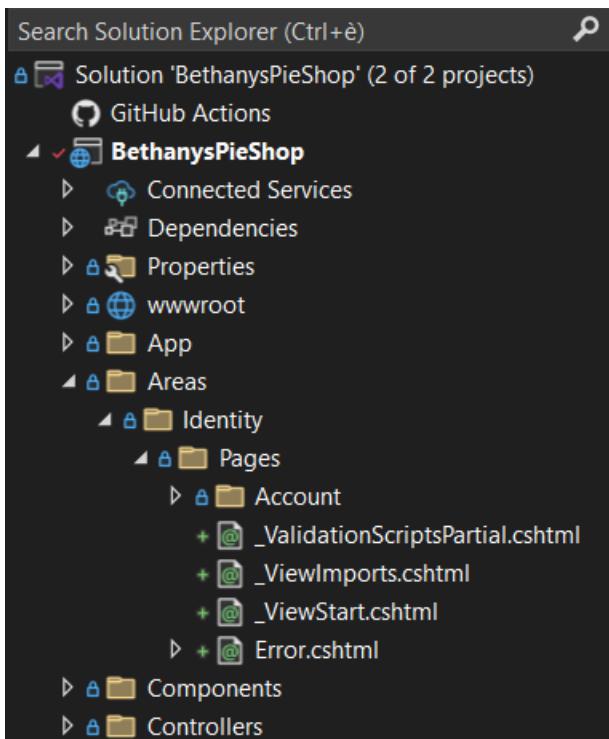
Prima riga 16 era così:

```
11
12  builder.Services.AddDefaultIdentity<IdentityUser>(options =>
13    options.SignIn.RequireConfirmedAccount = true)
14      .AddEntityFrameworkStores<BethanysPieShopDbContext>();
```

L'ho modificata per rendere i test più semplici.

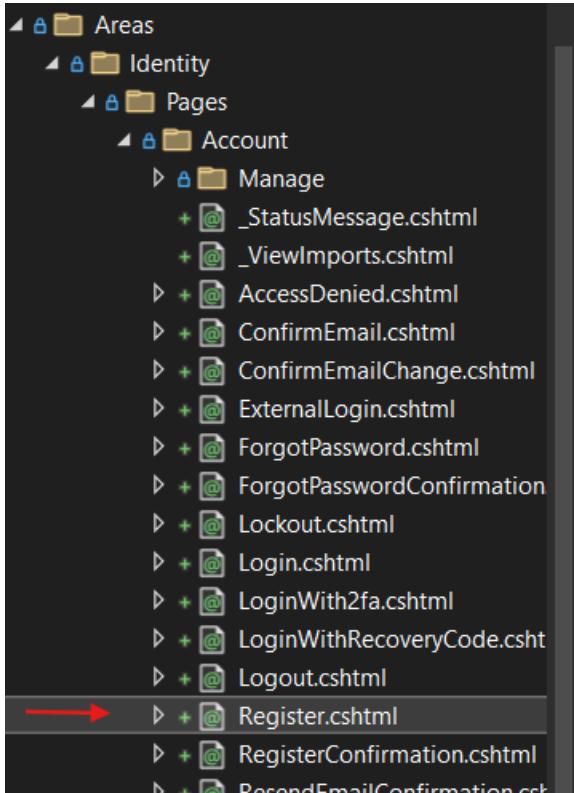
Abbiamo una nuova cartella generata dallo Scaffolding, la cartella **Areas**:

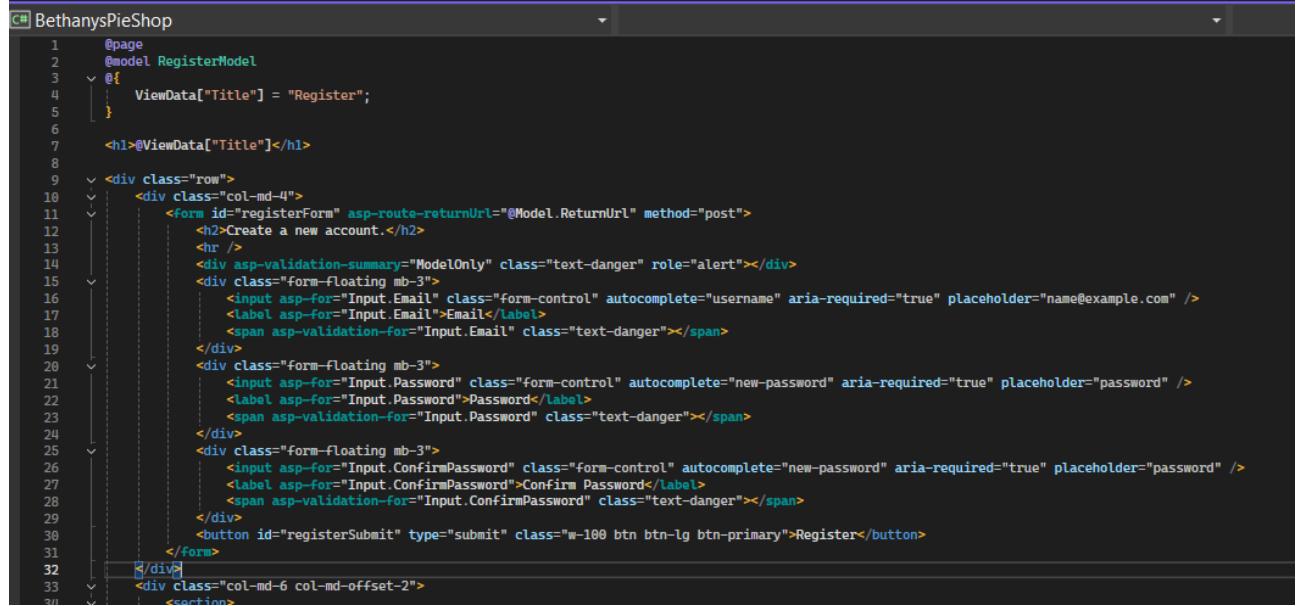
[Home](#)



Utilizzata per organizzare le funzionalità correlate alle identità.

Giungiamo in Register.cshtml:





The screenshot shows a code editor window with the title "Bethany'sPieShop". The file being edited is "Register.cshtml". The code is written in ASP.NET MVC Razor syntax. It starts with a page directive (@page) and a model directive (@model RegisterModel). A ViewData entry for "Title" is set to "Register". The view contains an H1 tag with the title. Below it is a form with a row class. The form has a col-md-4 column containing an h2 tag with the text "Create a new account.". The form includes validation summary and floating labels for email, password, and confirm password fields. Each field has an asp-for attribute pointing to its respective Input model property. The password field also has an autocomplete="new-password" attribute. The form ends with a button labeled "Register".

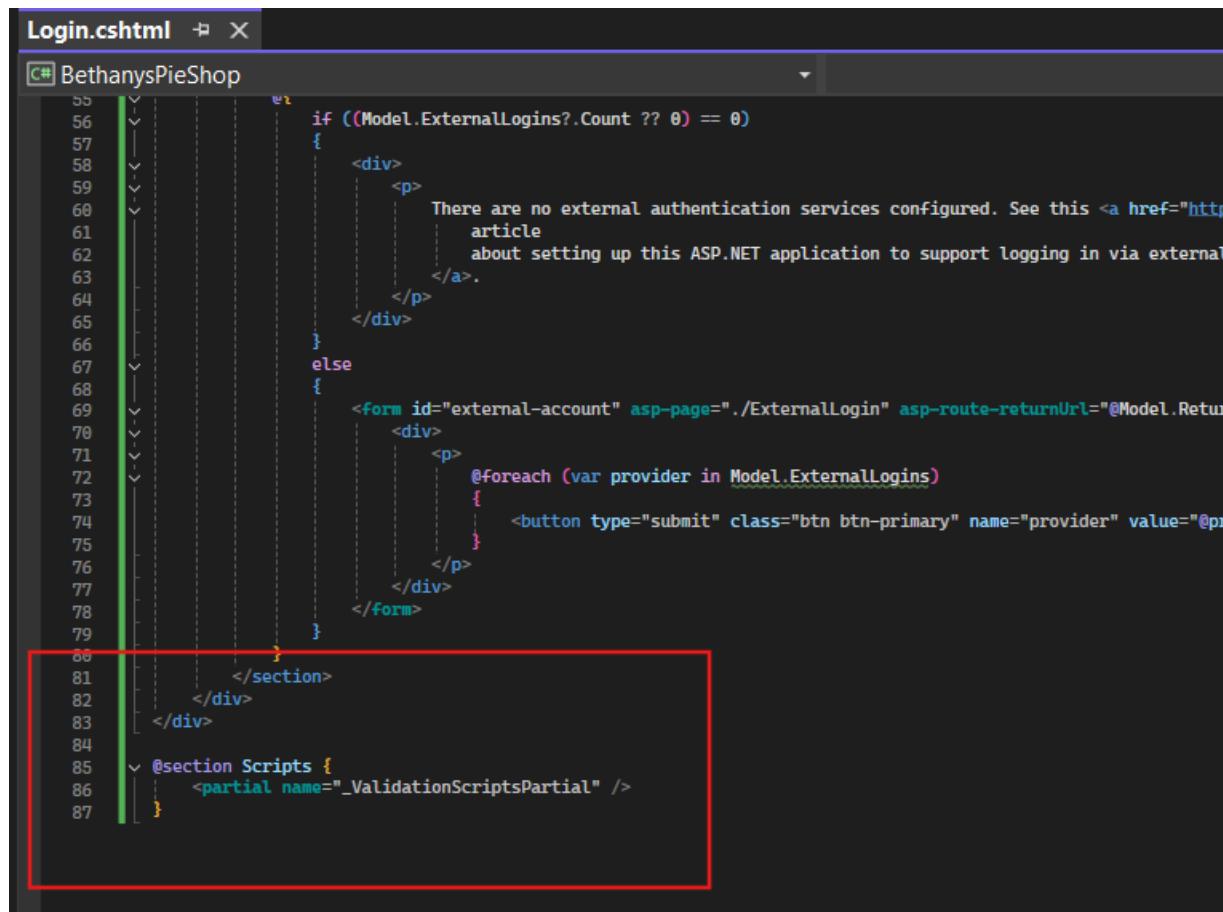
```
1  @page
2  @model RegisterModel
3  @{
4      ViewData["Title"] = "Register";
5  }
6
7  <h1>@ViewData["Title"]</h1>
8
9  <div class="row">
10 <div class="col-md-4">
11     <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl" method="post">
12         <h2>Create a new account.</h2>
13         <br />
14         <div asp-validation-summary="ModelOnly" class="text-danger" role="alert"></div>
15         <div class="form-floating mb-3">
16             <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" placeholder="name@example.com" />
17             <label asp-for="Input.Email">Email</label>
18             <span asp-validation-for="Input.Email" class="text-danger"></span>
19         </div>
20         <div class="form-floating mb-3">
21             <input asp-for="Input.Password" class="form-control" autocomplete="new-password" aria-required="true" placeholder="password" />
22             <label asp-for="Input.Password">Password</label>
23             <span asp-validation-for="Input.Password" class="text-danger"></span>
24         </div>
25         <div class="form-floating mb-3">
26             <input asp-for="Input.ConfirmPassword" class="form-control" autocomplete="new-password" aria-required="true" placeholder="password" />
27             <label asp-for="Input.ConfirmPassword">Confirm Password</label>
28             <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
29         </div>
30         <button id="registerSubmit" type="submit" class="w-100 btn btn-lg btn-primary">Register</button>
31     </form>
32 </div>
33 <div class="col-md-6 col-md-offset-2">
34     <section>
```

In breve Microsoft ci fornisce molto codice utile per Identity.

Quello che faremo è sovrascrivere il codice dell'interfaccia utente con uno snippet (il nostro codice avrà 40 righe)

Facciamo lo stesso con login.cshtml, in modo che corrispondano al resto della nostra applicazione.

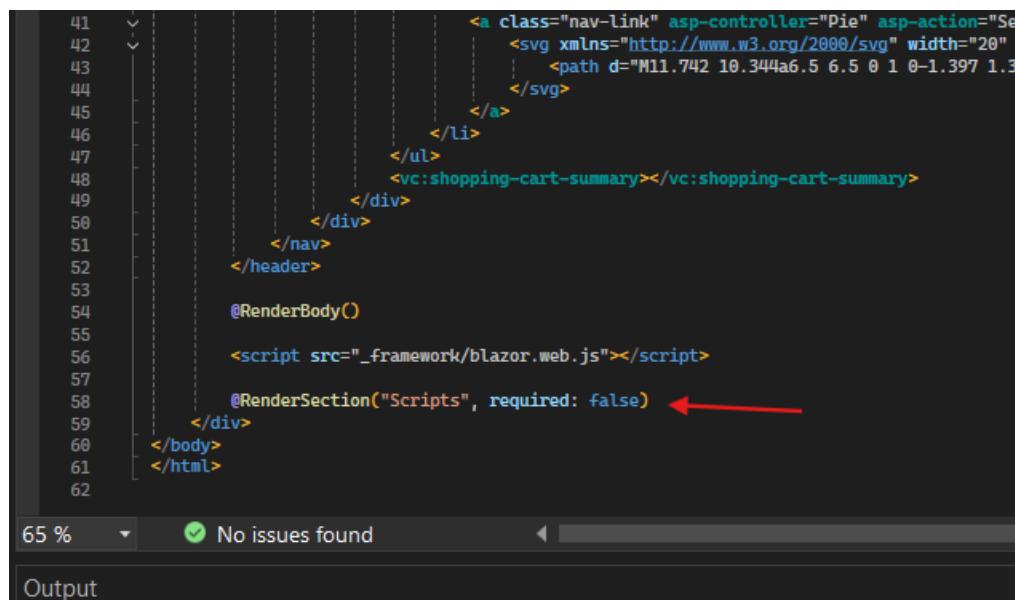
Home



```
55 if ((Model.ExternalLogins?.Count ?? 0) == 0)
56 {
57     <div>
58         <p>
59             There are no external authentication services configured. See this <a href="http://go.microsoft.com/fwlink/?LinkID=532165">article</a> about setting up this ASP.NET application to support logging in via external
60             providers.
61         </p>
62     </div>
63 }
64 else
65 {
66     <form id="external-account" asp-page=".ExternalLogin" asp-route-returnUrl="@Model.ReturnUrl">
67         <div>
68             <p>
69                 @foreach (var provider in Model.ExternalLogins)
70                 {
71                     <button type="submit" class="btn btn-primary" name="provider" value="@provider.Name">@provider.Name</button>
72                 }
73             </p>
74         </div>
75     </form>
76 }
77
78 </div>
79 </div>
80 </div>
81 </div>
82 </div>
83 </div>
84 <@section Scripts {
85     <partial name="_ValidationScriptsPartial" />
86 }>
```

Osservazione: Riga 85 si occupa del rendering, in modo che i vari script possono essere richiamati dal nostro layout.

Quindi andiamo in _Layout.cshtml, presente in Views > Shared e aggiungiamo ciò:



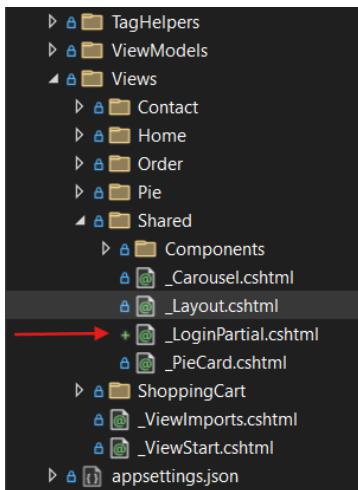
```
41 <a class="nav-link" asp-controller="Pie" asp-action="Search">
42     <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20">
43         <path d="M11.742 10.344a6.5 6.5 0 1 0-1.397 1.397c-0.78 0.78-1.397 1.397-1.397 1.397s-1.397-0.78-1.397-1.397c0-0.78 0.78-1.397 1.397-1.397l1.397-1.397c0.78-0.78 1.397-1.397 1.397-1.397z" />
44     </svg>
45 </a>
46 </li>
47 </ul>
48 <vc:shopping-cart-summary></vc:shopping-cart-summary>
49 </div>
50 </div>
51 </nav>
52 </header>
53
54 @RenderBody()
55
56 <script src="_framework/blazor.web.js"></script>
57
58 @RenderSection("Scripts", required: false) ←
59 </div>
60 </body>
61 </html>
```

Farà in modo che pagine come login e registro che definiscono una sezione aggiuntiva chiamata Scripts possano

Home

includerla in questo posto nel layout

Andiamo in un altro file che è stato generato, sempre in shared, chiamato _LoginPartial.cshtml:



```
1 @using Microsoft.AspNetCore.Identity
2
3 @inject SignInManager<IdentityUser> SignInManager
4 @inject UserManager<IdentityUser> UserManager
5
6 <ul class="navbar-nav">
7   @if (SignInManager.IsSignedIn(User))
8   {
9     <li class="nav-item">
10       <a id="manage" class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @UserManager.GetUserName(User)!</a>
11     </li>
12   }
13   <li class="nav-item">
14     <form id="logoutForm" class="form-inline" asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })">
15       <button id="Logout" type="submit" class="nav-link btn btn-link text-dark border-0">Logout</button>
16     </form>
17   </li>
18 }
19 <else
20 {
21   <li class="nav-item">
22     <a class="nav-link text-dark" id="register" asp-area="Identity" asp-page="/Account/Register">Register</a>
23   </li>
24   <li class="nav-item">
25     <a class="nav-link text-dark" id="login" asp-area="Identity" asp-page="/Account/Login">Login</a>
26   </li>
27 </ul>
```

Osservazioni:

Utilizzato per visualizzare il nome dello user che ha effettuato l'accesso o un pulsante Registrati e Accedi se non ha effettuato l'accesso.

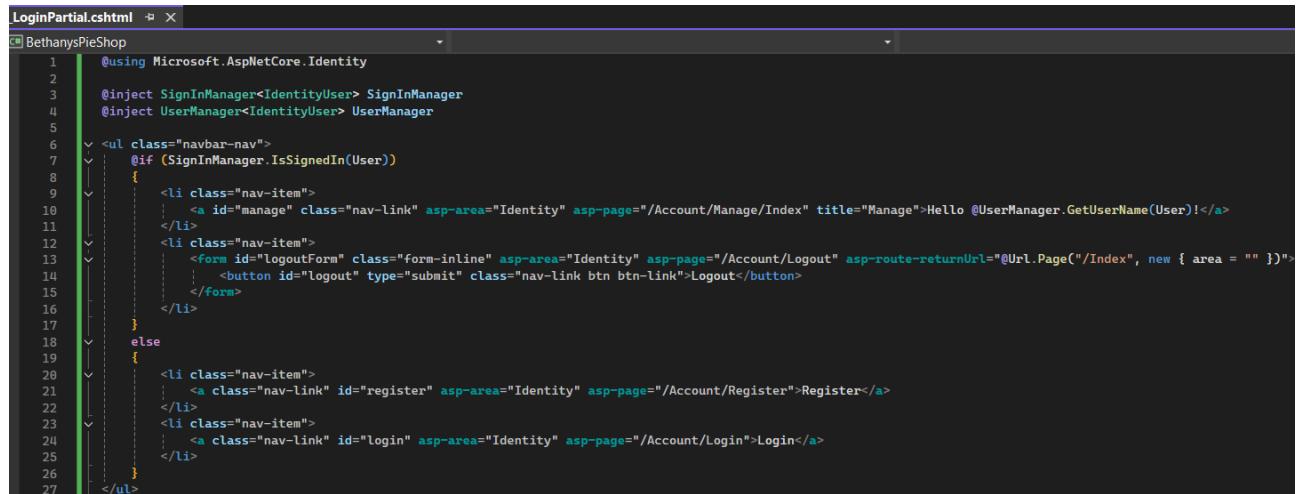
If-else per controllare se l'utente corrente ha effettuato l'accesso, infatti utilizziamo una delle classi Core Identity ASP.NET chiamata SignInManager e fa appello al metodo IsSignedIn, se ha effettuato l'accesso il nome viene visualizzato a riga 10 con @UserManager.GetUserName(User), insieme a un collegamento per gestire l'account .

Riga 13 presenta il pulsante di disconnessione.

Se entriamo nell'else mostreremo due collegamenti alle pagine di registrazione e accesso.

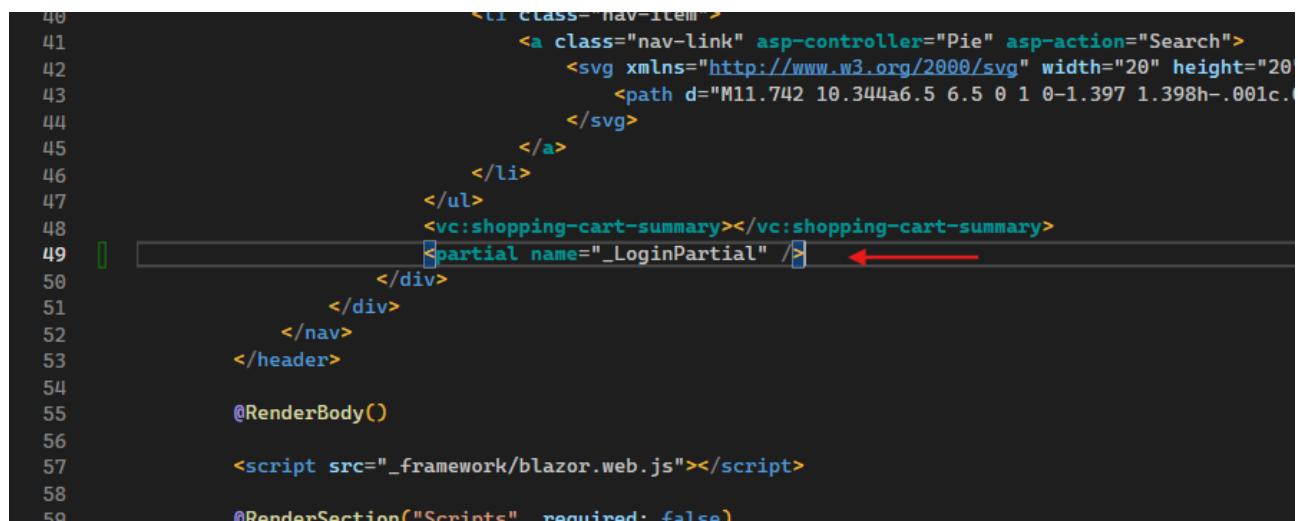
Nel nostro caso rimuoviamo text-dar a riga 21 e 24 per essere coerenti con il Layout di BethanPieShop:

Home



```
1  @using Microsoft.AspNetCore.Identity
2
3  @inject SignInManager<IdentityUser> SignInManager
4  @inject UserManager<IdentityUser> UserManager
5
6  <ul class="navbar-nav">
7      @if (SignInManager.IsSignedIn(User))
8      {
9          <li class="nav-item">
10             <a id="manage" class="nav-link" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @UserManager.GetUserName(User)!</a>
11         </li>
12         <li class="nav-item">
13             <form id="logoutForm" class="form-inline" asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })">
14                 <button id="logout" type="submit" class="nav-link btn btn-link">Logout</button>
15             </form>
16         </li>
17     }
18     else
19     {
20         <li class="nav-item">
21             <a class="nav-link" id="register" asp-area="Identity" asp-page="/Account/Register">Register</a>
22         </li>
23         <li class="nav-item">
24             <a class="nav-link" id="login" asp-area="Identity" asp-page="/Account/Login">Login</a>
25         </li>
26     }
27 </ul>
```

Torniamo al nostro _Layout.cshtml e aggiungiamo LoginPartial:

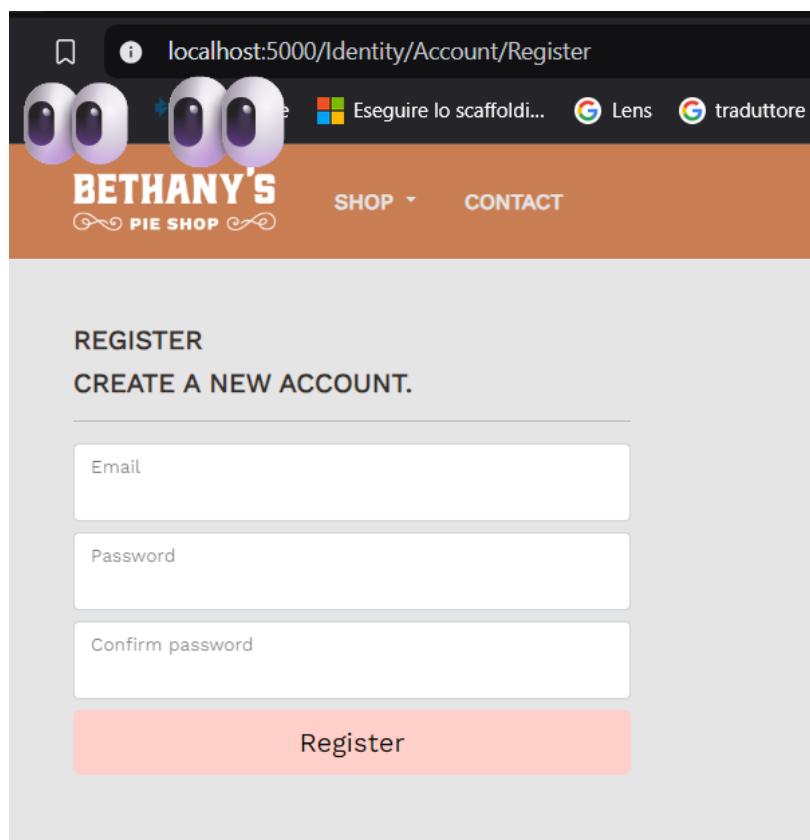
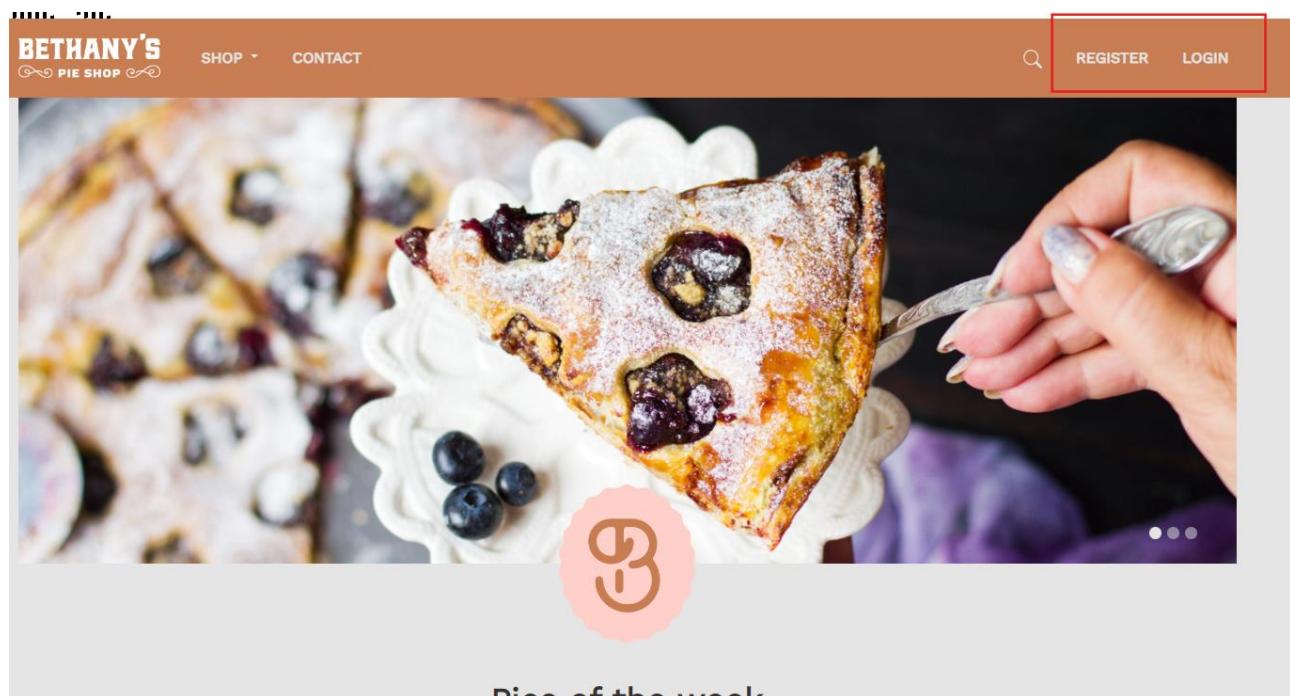


```
40
41
42
43
44
45
46
47
48
49     [partial name="_LoginPartial" /] ←
50
51
52
53
54
55
56
57
58
59
```

Osservazione: Riga 57 si può cancellare se decidiamo di non utilizzare la componente Blazor nel nostro progetto.

Salviamo, buildiamo ed eseguiamo:

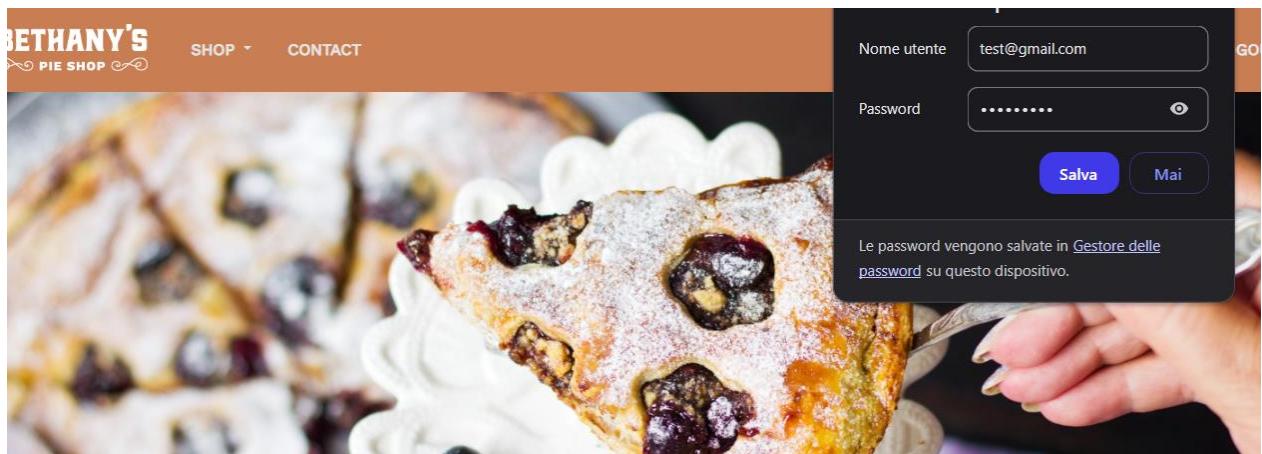
[Home](#)



Possiamo notare l'url della pagina Register che inizia con Identity/Account.

Proviamo a registrarci:

[Home](#)



Using Authorization

```
[Authorize]  
public class OrderController : Controller  
{  
}
```

Authorizing Users

Anche se ora abbiamo consentito a un utente di registrarsi nell'applicazione,

Non stiamo ancora utilizzando le informazioni di cui disponiamo.

Fondamentalmente abbiamo utilizzato solo l'autenticazione, dobbiamo quindi permettere di accedere a una determinata risorsa solo agli utenti che hanno l'autorizzazione.

Quindi verificheremo che l'utente che vuole effettuare un ordine sia registrato, altrimenti la richiesta sarà rifiutata.

Possiamo anche avere un controllo più preciso su ciò che l'utente può fare, utilizzando l'attributo Autorizza:

```
public class OrderController : Controller
{
    [Authorize]
    public IActionResult Checkout()
    {
    }
}
```

Action Level

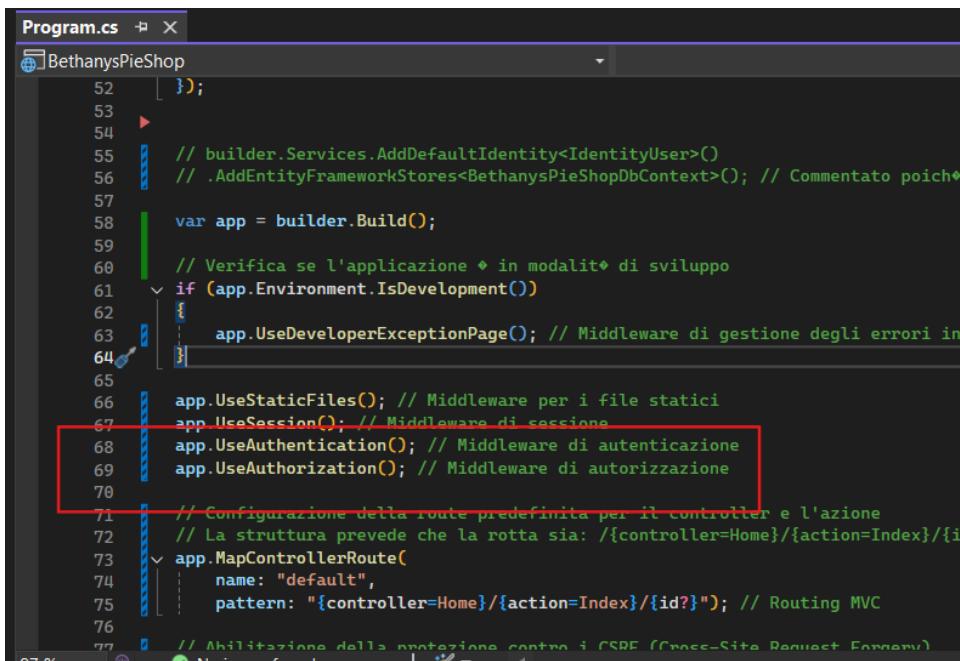
```
[Authorize(Roles = "Administrator")]
public class OrderController : Controller
{
}
```

Combining with Roles

Demo: Using Authorization

Andiamo in Program.cs e aggiungiamo i seguenti Middleware:

[Home](#)

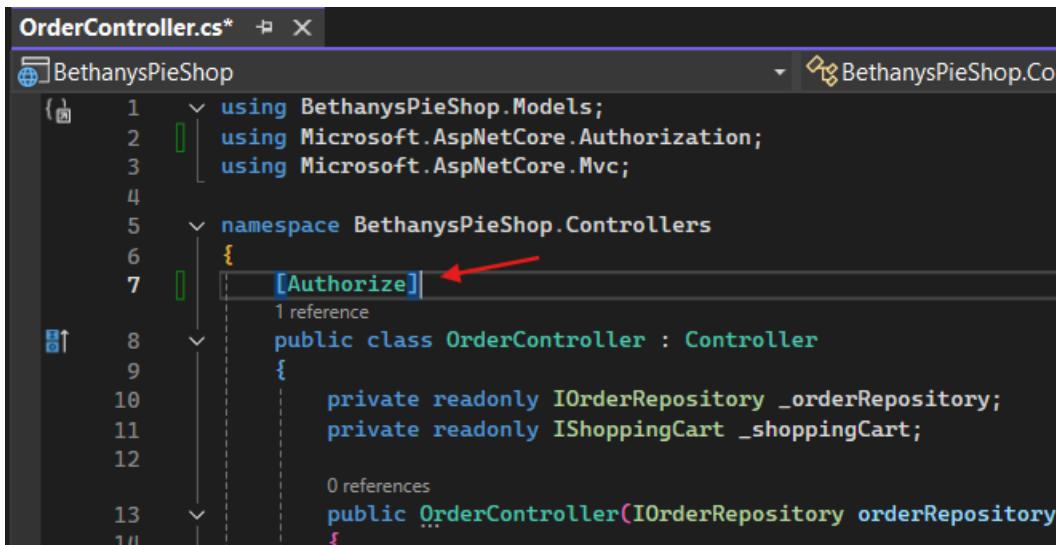


The screenshot shows the `Program.cs` file in a code editor. The code is part of the `BethanysPieShop` project. It defines a `builder` variable and uses it to build an `IApplicationBuilder`. The application checks if it's in development mode and then applies developer exception pages. It then adds static files, sessions, authentication, and authorization middleware. A red box highlights the `app.UseAuthentication()` and `app.UseAuthorization()` calls. The code then maps a controller route named `default` with a pattern of `{controller=Home}/{action=Index}/{id?}`. Finally, it enables CORS protection.

```
52     });
53
54
55     // builder.Services.AddDefaultIdentity<IdentityUser>()
56     // .AddEntityFrameworkStores<BethanysPieShopDbContext>(); // Commentato poiché non viene più utilizzato
57
58     var app = builder.Build();
59
60     // Verifica se l'applicazione è in modalità di sviluppo
61     if (app.Environment.IsDevelopment())
62     {
63         app.UseDeveloperExceptionPage(); // Middleware di gestione degli errori in sviluppo
64     }
65
66     app.UseStaticFiles(); // Middleware per i file statici
67     app.UseSession(); // Middleware di sessione
68     app.UseAuthentication(); // Middleware di autenticazione
69     app.UseAuthorization(); // Middleware di autorizzazione
70
71     // Configurazione della route predefinita per il controller e l'azione
72     // La struttura prevede che la rotta sia: /{controller=Home}/{action=Index}/{id?}
73     app.MapControllerRoute(
74         name: "default",
75         pattern: "{controller=Home}/{action=Index}/{id?}"); // Routing MVC
76
77     // Abilitazione delle protezioni contro i CSRF (Cross-Site Request Forgery)
78 }
```

Il nostro scopo è impedire ad un utente non registrato di poter accedere al carrello di un prodotto e poter effettuare un ordine.

Quindi giungiamo in `OrderController.cs` e aggiungiamo l'attributo necessario:

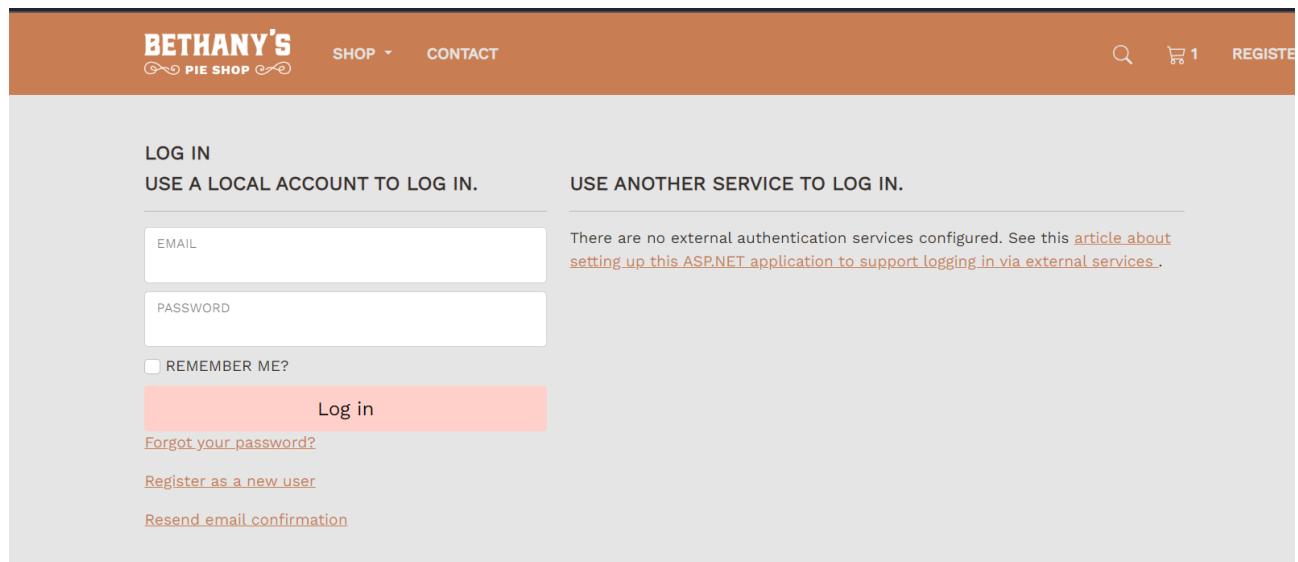


The screenshot shows the `OrderController.cs` file in a code editor. The code defines a `OrderController` class that inherits from `Controller`. It has a constructor that takes `IOrderRepository` and `IShoppingCart` as parameters. An arrow points to the `[Authorize]` attribute on the first line of the controller's body.

```
1  using BethanysPieShop.Models;
2  using Microsoft.AspNetCore.Authorization;
3  using Microsoft.AspNetCore.Mvc;
4
5  namespace BethanysPieShop.Controllers
6  {
7      [Authorize] // Arrow points here
8      public class OrderController : Controller
9      {
10         private readonly IOrderRepository _orderRepository;
11         private readonly IShoppingCart _shoppingCart;
12
13         public OrderController(IOrderRepository orderRepository,
14             IShoppingCart shoppingCart)
```

Fatto ciò, noteremo che avviene un redirect alla pagina di registrazione:

Home



Naturalmente possiamo accedere con i dati con i quali ci siamo registrati precedentemente

Deploying the Site to an Azure App Service

- Understanding the Azure App Service
- Demo: Looking at the Azure Portal
- Deploying the Application to an Azure App Service
- Demo: Deploying the Site

Premessa: Non ho effettuato il deploying in quanto momentaneamente non possiedo un Tenant (inoltre dovrebbe richiedere un pagamento mensile).

Ora che il sito è stato approvato dal nostro cliente non ci resta che distribuirlo in rete, ovvero effettuare il Deploying.

Effettueremo il Deploying usando Azure App Service

Understanding the Azure App Service



Azure Portal

A screenshot of the Microsoft Azure portal. The top navigation bar includes links for "Create a resource", "SQL databases", "Resource groups", "Spatial Anchors", "App Service plans", "Subscriptions", "All resources", "App Service Certificates", and "More services". Below this is a "Resources" section with a "Recent" tab selected, showing a list of resources with their names, types, and last viewed times. The list includes various App Services, Resource groups, and SQL servers. At the bottom, there is a "Navigate" section with links for "Subscriptions", "Resource groups", "All resources", and "Dashboard".

Azure Services



Compute services



Messaging



Identity



Data services



Storage

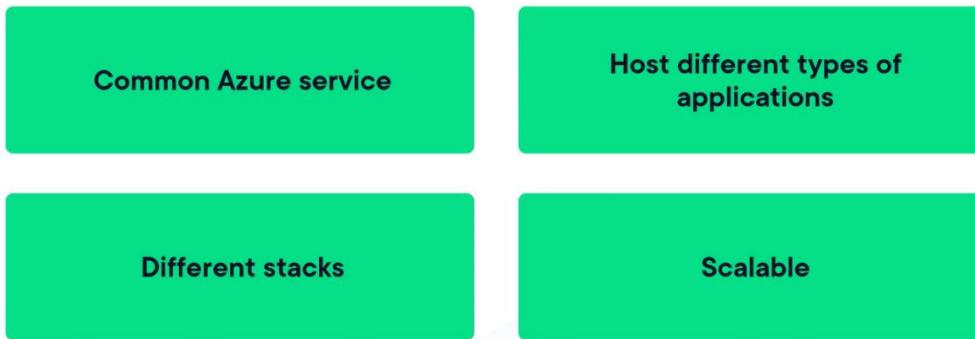


AI

[Home](#)

Noi utilizzeremo un app service:

Introducing Azure App Service



Dobbiamo depolare anche il nostro D.B.:

Demo: Looking at the Azure Portal

Un concetto fondamentale sono i gruppi di risorse, all'interno di un gruppo di risorse possiamo trovare tipi di risorse come App Service, App Service plan, SQL database, SQL server. (Esempio)

The screenshot shows the Azure Portal interface for the "BethanyPieShop" resource group. The left sidebar lists various resource types: Overview, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Deployments, Security, Policies, Properties, Locks, Cost Management, Cost analysis, and Cost alerts (preview). The main content area displays the "Essentials" section with information about the subscription (MSDN Snowball BVBA), deployment status (2 succeeded), and location (West Europe). Below this, the "Resources" section shows a list of four items:

Name	Type	Location
BethanyPieShop20220618190113	App Service	West Europe
BethanyPieShop20220618190113Plan	App Service plan	West Europe
BethanyPieShop_db (bethanypieshopdbserver123456/BethanyPieShop_db)	SQL database	West Europe
bethanypieshopobserver123456	SQL server	West Europe

A red box highlights the list of resources. A red banner at the bottom of the table says "Ne creeremo uno anche noi tra un minuto," which translates to "We will create one for us in a minute."

App Service è la più comune:

Home

The screenshot shows the Microsoft Azure portal interface for an App Service named 'BethanysPieShop20220618190113'. The main content area displays the 'Essentials' section with details such as Resource group (BethanysPieShop), Status (Running), Location (West Europe), and Subscription (MSDN Snowball BVBA). A red arrow points to the 'URL' field, which contains the value <https://bethanyPieShop20220618190113.azurewebsites.net>. Below this, there are sections for 'Diagnose and solve problems' and 'Application Insights'. At the bottom, there are three cards: 'Http 5xx', 'Data In', and 'Data Out'. A callout box highlights the URL with the text 'Ad esempio, questo è l'URL che possiamo utilizzare per navigare nel sito.'

Osservazione: App Service Plan è il server che ospiterà il nostro servizio di app.

Attualmente il piano è un S1, cliccandolo vediamo il piano di servizio, dove viene mostrato che un'app è in esecuzione

The screenshot shows the Microsoft Azure portal interface for an App Service Plan named 'BethanysPieShop20220618190113Plan'. The main content area displays the 'Essentials' section with details such as Resource group (BethanysPieShop), Status (Ready), Location (West Europe), and Subscription (MSDN Snowball BVBA). A red arrow points to the 'App(s) / Slots' field, which contains the value 1/0. Below this, there are three performance monitoring charts: 'CPU Percentage', 'Memory Percentage', and 'Data In'. The 'CPU Percentage' chart shows usage around 60%. The 'Memory Percentage' chart shows usage around 50%. The 'Data In' chart shows traffic levels between 1.5MB and 3.5MB.

Cliccando Scale Up, nel pannello di scorrimento a sinistra, possiamo constatare il traffico:

Home

The screenshot shows the Azure portal interface for managing an App Service plan. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events (preview), Settings, Apps, File system storage, Networking, Scale up (App Service plan) (which is selected and highlighted with a red arrow), Scale out (App Service plan), Properties, and Locks.

The main content area displays the 'BethanysPieShop20220618190113Plan | Scale up (App Service plan)' page. It features three tabs: Dev / Test (selected), Production, and Isolated. Under 'Recommended pricing tiers', the S1 tier is highlighted with a red box. Other tiers shown include P1V2, P2V2, P3V2, P1V3, P2V3, and P3V3. Each tier provides details on total ACU, memory, and price. Below the tiers, there are sections for 'Included features' (Custom domains / SSL) and 'Included hardware' (Azure Compute Units (ACU)).

Iniziamo quindi con la creazione di un nuovo gruppo di risorse in cui inseriremo quindi tutte le risorse correlate:

The screenshot shows the Azure services and resources management interface. The top navigation bar includes 'Create a resource', 'Resource groups' (highlighted with a red box), 'Azure Database for PostgreSQL...', 'Cost Management ...', 'SQL databases', 'Subscriptions', 'Cognitive Services', 'All resources', 'Virtual machines', and 'More services'. The 'Recent' tab in the 'Resources' section is selected, displaying a list of resources with columns for Name, Type, and Last Viewed.

Name	Type	Last Viewed
BethanysPieShop	Resource group	a few seconds ago
TechoramaProjection	App Service	29 minutes ago
BlazorApps	Resource group	36 minutes ago
Default1	App Service plan	a month ago

[Home](#)

Resource groups - Microsoft Azure + portal.azure.com/#view/HubsExtension/BrowseResourceGroups

Microsoft Azure Search resources, services, and docs (G+/)

Home > Resource groups ⚡ ...

Ordina

+ Create Manage view Refresh Export to CSV Open query Assign t

Create Filter for any field... Subscription == all Location == all Add filter

0 Unsecure resources

<input type="checkbox"/> Name ↑↓
<input type="checkbox"/> [?] BethanysPieShop
<input type="checkbox"/> [?] BlazorApps
<input type="checkbox"/> [?] cloud-shell-storage-westeurope
<input type="checkbox"/> [?] Default-ApplicationInsights-EastUS
<input type="checkbox"/> [?] View details

[Home](#)

← → ⌛ 🔒 portal.azure.com/#create/Microsoft.ResourceGroup

Microsoft Azure Search resources, services, and docs (G+)

Home > Resource groups >

Create a resource group

Basics Tags Review + create

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. [Learn more](#)

Project details

Subscription * ⓘ

Resource group * ⓘ

Resource details

Region * ⓘ

(Europe) West Europe

Quindi faccio clic su Crea e riv

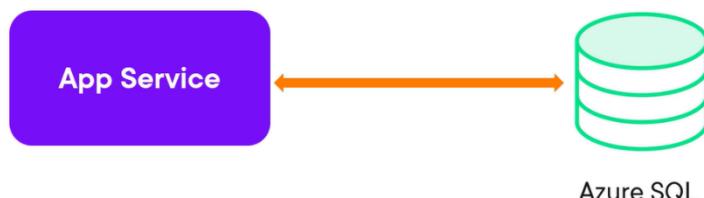
Review + create Review + create Next: Tags >

Verrà mostrato un messaggio di conferma.

Deploying the Application to an Azure App Service

Ora siamo pronti per deployare

Our App in Azure

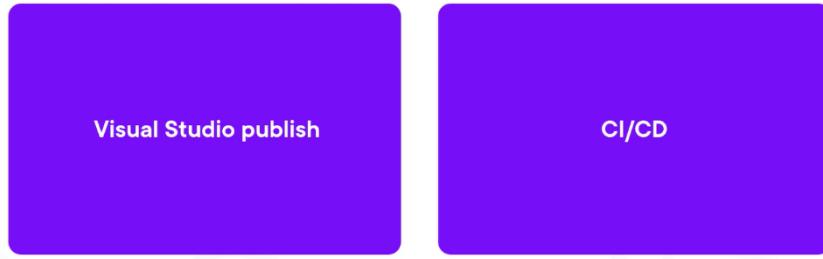


[Home](#)

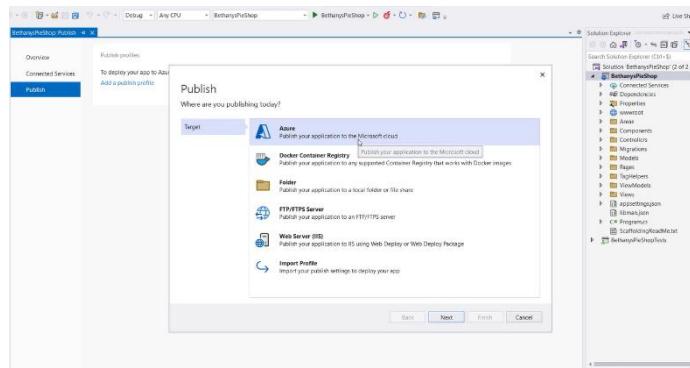
La soluzione/opzione più semplice è Visual Studio publish, click destro del mouse sul progetto in Solution Explorer.

Oppure utilizzando i CI/CD "Continous Integration" "Continuous Deployment".

Deployment Options

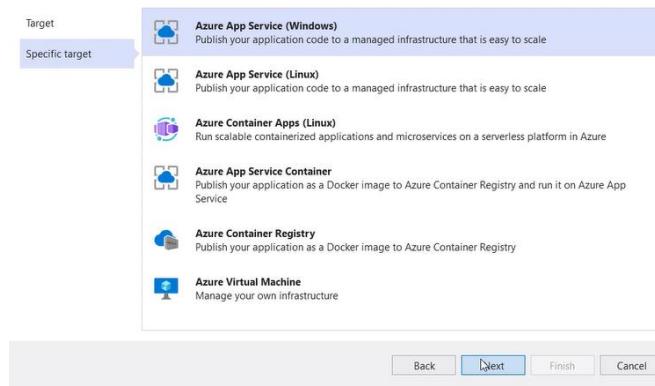


Demo: Deploying the Site

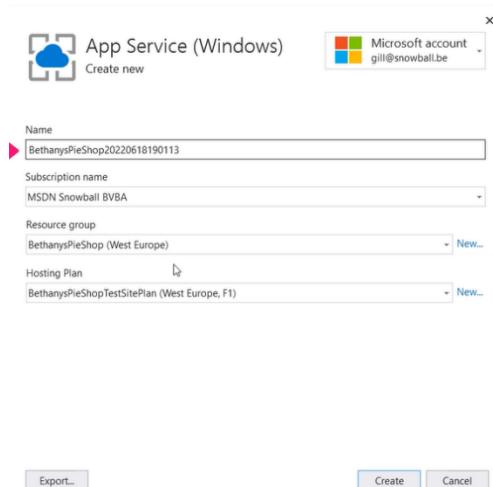
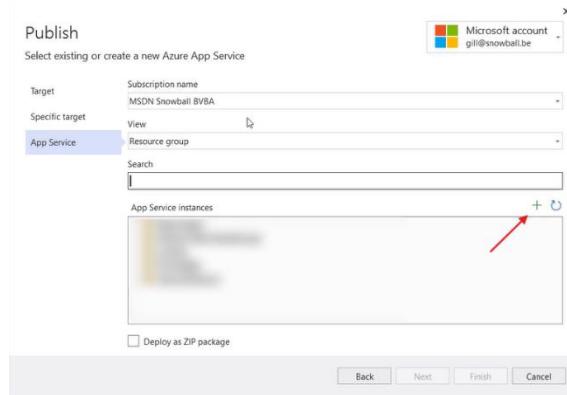


Publish

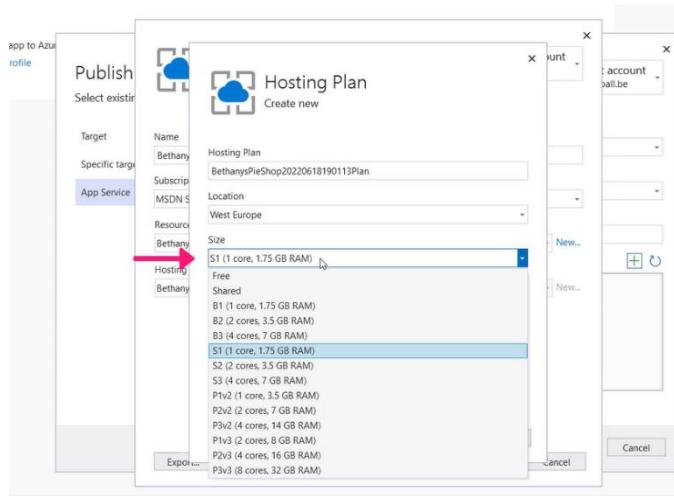
Which Azure service would you like to use to host your application?



Home



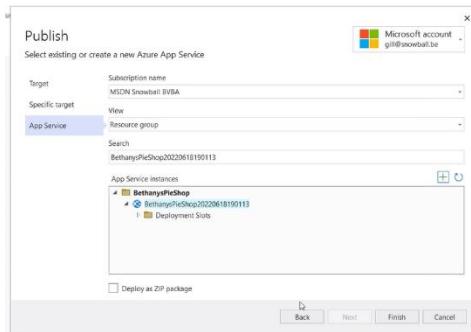
Creiamo un nuovo Hostin Plan cliccando su "New..."



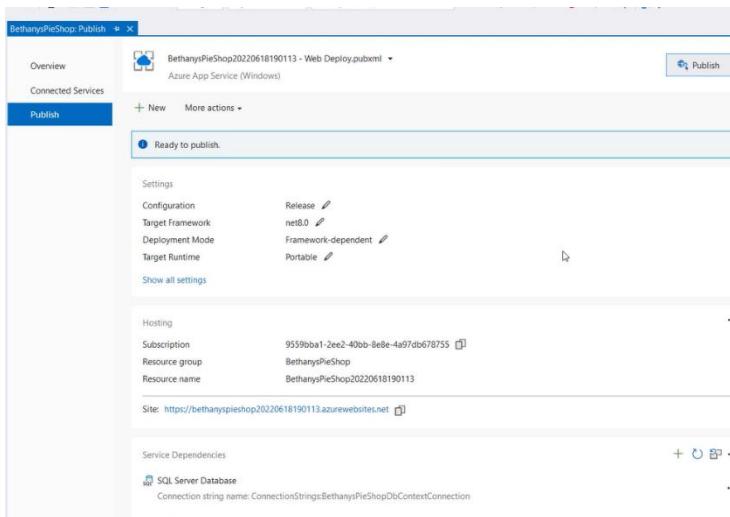
Già size S1 ha un costo mensile, assicurati di eliminare le risorse al termine di questa demo

Okay > Create

Home

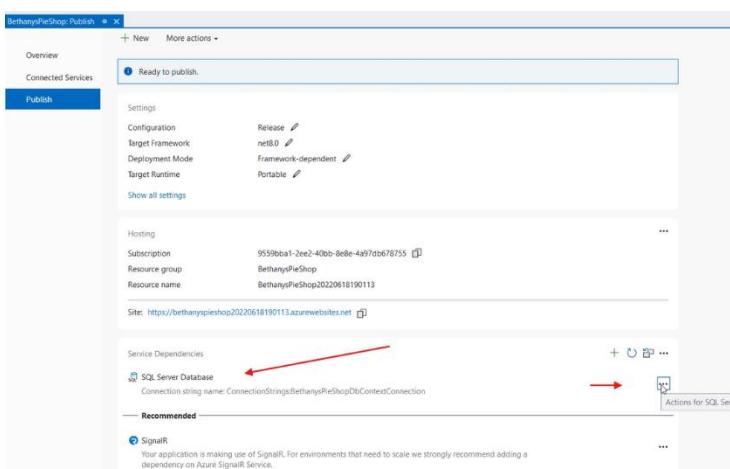


Finish > Close



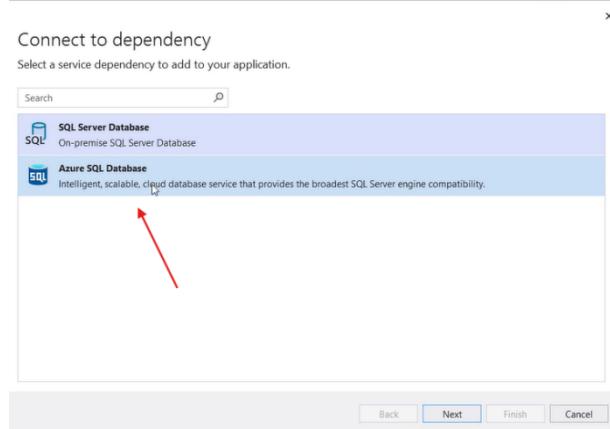
Sarà necessario cliccare **Publish** in alto a destra per ultimare del tutto la pubblicazione

Inoltre possiamo notare che Visual Studio ha rilevato una Service Dependencies SQL Server Database.

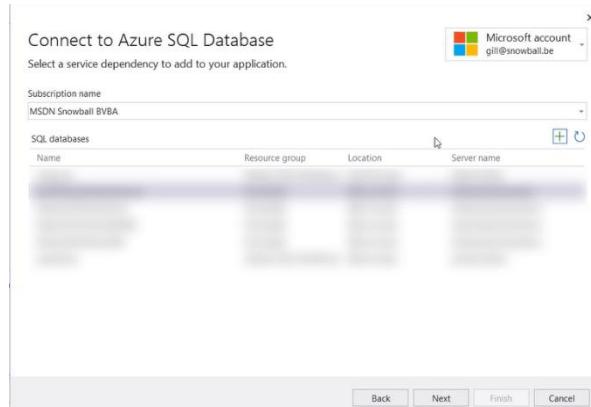


Home

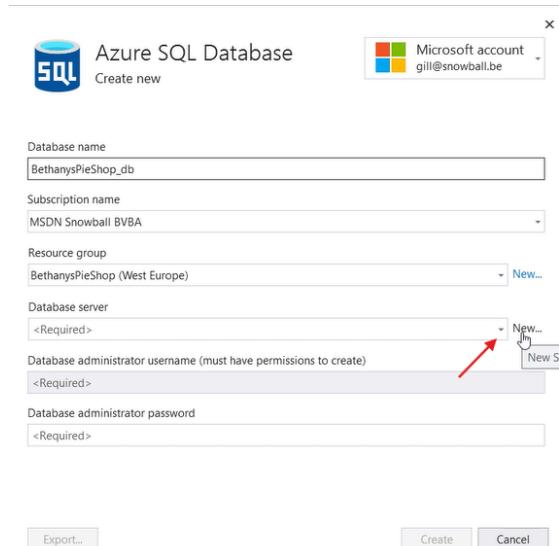
Difatti ci serve un DB, quindi clicchiamo e successivamente selezioniamo "Connect".



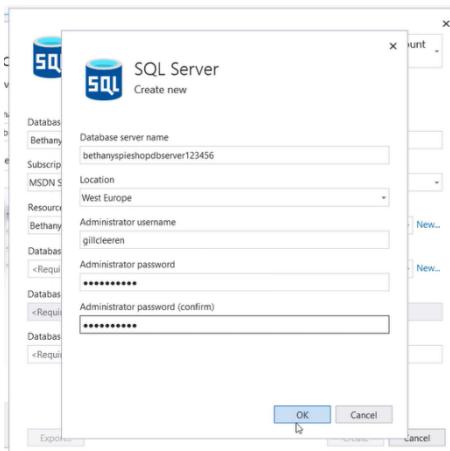
Verranno mostrati i DB esistenti:



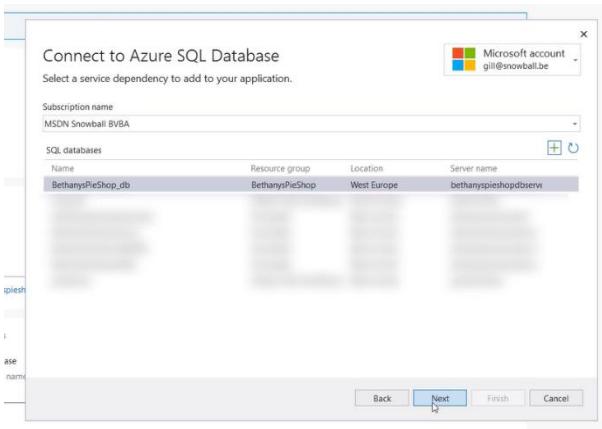
Ma noi creeremo uno nuovo cliccando sul simbolo del '+'



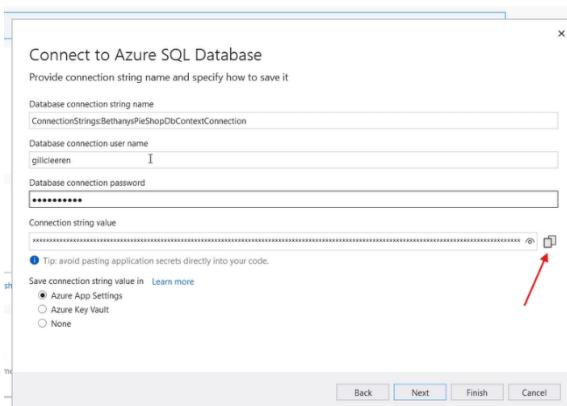
Home



Okay > Create e vedremo il DB creato:



Clicchiamo su "Next" e dovremo reinserire i dati e password per la connessione al DB:

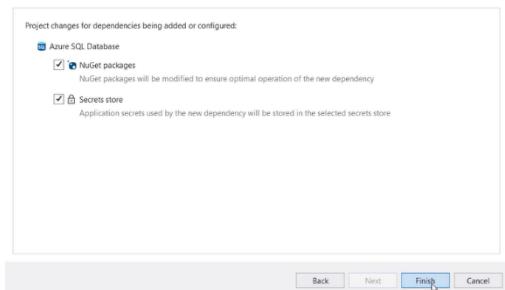


Dopo aver inserito i dati delle nostre credenziali di accesso, copiamo il connection string value.

[Home](#)

Click su "Next"

Summary of changes



Ora dobbiamo assicurarci che il nostro db schema venga implementato in quel database.

Quindi andiamo in appsettings.json:

e sostituiamo l'attuale DbContextConnection string con quello copiato nel clipboard:

```
appsettings.json * X BethanysPieShop.Publish
Schema: https://json.schemastore.org/appsettings.json
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    }
8    "AllowedHosts": "*",
9    "ConnectionStrings": {
10      "BethanysPieShopDbContextConnection": "Data Source=tcp:bethanypieshopdbserver123456.database.windows.net;Initial Catalog=BethanyPieShop;User ID=bethany;Password=MySecurePass;MultipleActiveResultSets=true"
11    }
12  }
13
```

Ora dobbiamo aggiornare il db per far sì che avvenga la migrazione sul database in Azure.

Dobbiamo però prima tornare al portare e aggiornare il firewall, perché per impostazione di default non saremo in grado di accederci da VS.

Home

The screenshot shows the Microsoft Azure portal interface for the 'BethanysPieShop' resource group. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Settings (Deployments, Security, Policies, Properties, Locks), Cost Management (Cost analysis, Cost alerts (preview), Budgets), and Power Platform (Power BI, Power Apps, Power Automate). The main content area displays the 'Essentials' section with information about the subscription (move to MSDN Snowball BVBA, ID 9559bba1-2ee2-40bb-8e8e-4a97db678755, Location West Europe) and tags (Click here to add tags). Below this is a 'Resources' section with a filter bar (Type == all, Location == all, Add filter). It lists four records: 'BethanysPieShop20220618190113' (App Service, West Europe), 'BethanysPieShop20220618190113Plan' (App Service plan, West Europe), 'BethanysPieShop_db (bethanyspieshopdbserver123456/BethanysPieShop_db)' (SQL database, West Europe), and 'bethanyspieshopdbserver123456' (SQL server, West Europe). A red arrow points to the 'BethanysPieShop_db' entry.

Osservazione: Manca il nostro DB Schema, selezioniamo la risorsa mostrata nell'immagine.

The screenshot shows the Microsoft Azure portal for the 'BethanysPieShop_db' database. The left sidebar includes links for Overview, Activity log, Tags, Diagnose and solve problems, Getting started, Query editor (preview), Power Platform (Power BI, Power Apps, Power Automate), and Settings (Compute + storage). The main page displays the database's 'Essentials' section with details like Resource group (BethanysPieShop), Status (Online), Location (West Europe), Subscription (MSDN Snowball BVBA), and Tags (Click here to add tags). A red arrow points to the 'Set server firewall' button in the top toolbar. Below this is a 'Compute utilization' chart and a 'Database data storage' section showing 0.49% usage.

Scrolliamo in basso fino a Firewall rules e aggiungiamo l'IP del nostro client corrente.

Home

Virtual networks
Allow virtual networks to connect to your resource using service endpoints. [Learn more](#)

+ Add a virtual network rule

Rule	Virtual network	Subnet	Address range	Endpoint status	Resource group	Subscription
AllowAllAzureIPs			0.0.0.0	Open	BethanysPieShop	Microsoft Azure

Firewall rules
Allow certain public internet IP addresses to access your resource. [Learn more](#)

+ Add your client IPv4 address (81.164.235.174) + Add a firewall rule

Rule name	Start IPv4 address	End IPv4 address
AllowAllAzureIPs	0.0.0.0	0.0.0.0
ClientIPAddress_2022-6-18_19-13-57	81.164.235.174	81.164.235.174

Exceptions

Verrà mostrato:

Firewall rules
Allow certain public internet IP addresses to access your resource. [Learn more](#)

+ Add your client IPv4 address (81.164.235.174) + Add a firewall rule

Rule name	Start IPv4 address	End IPv4 address
AllowAllAzureIPs	0.0.0.0	0.0.0.0
ClientIPAddress_2022-6-18_19-13-57	81.164.235.174	81.164.235.174

Save | **Discard** | **ClientIPAddress_2022-6-18_19-13-57** è consentito attraverso il firewall e potrà quindi accedere alla risorsa.

Clicchiamo su "Save".

Ora torniamo in VS e apriamo il Package Manager Console:

appsettings.json BethanysPieShop: Publish

Schema: <https://json.schemastore.org/appsettings.json>

```
1  "logging": {
2    "LogLevel": {
3      "Default": "Information",
4      "Microsoft.AspNetCore": "Warning"
5    }
6  }
7
8  "allowedHosts": "*",
9  "connectionStrings": {
10   "BethanysPieShopDbContextConnection": "Data Source=tcp:bethanypie.com,1433;Initial Catalog=BethanysPieShop;User Id=bethany;Password=Pa$$w0rd;MultipleActiveResultSets=true"
11 }
12
13 }
```

Package Manager Console

Package source: All Default project: BethanysPieShop

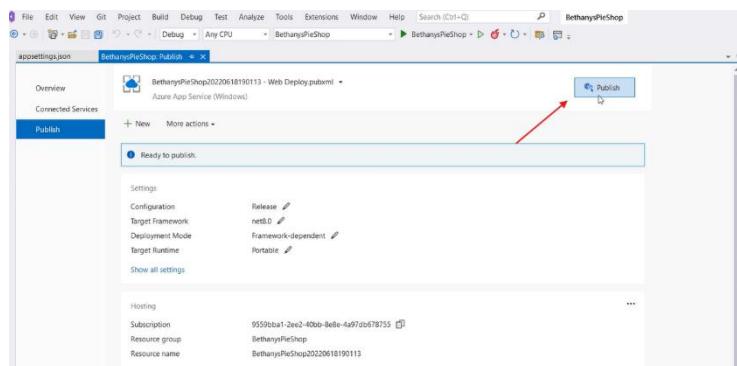
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.2.0.146

Type 'get-help NuGet' to see all available NuGet commands.

PM> update-database

Home



Finish! Il nostro sito è stato pubblicato con successo su Azure:

