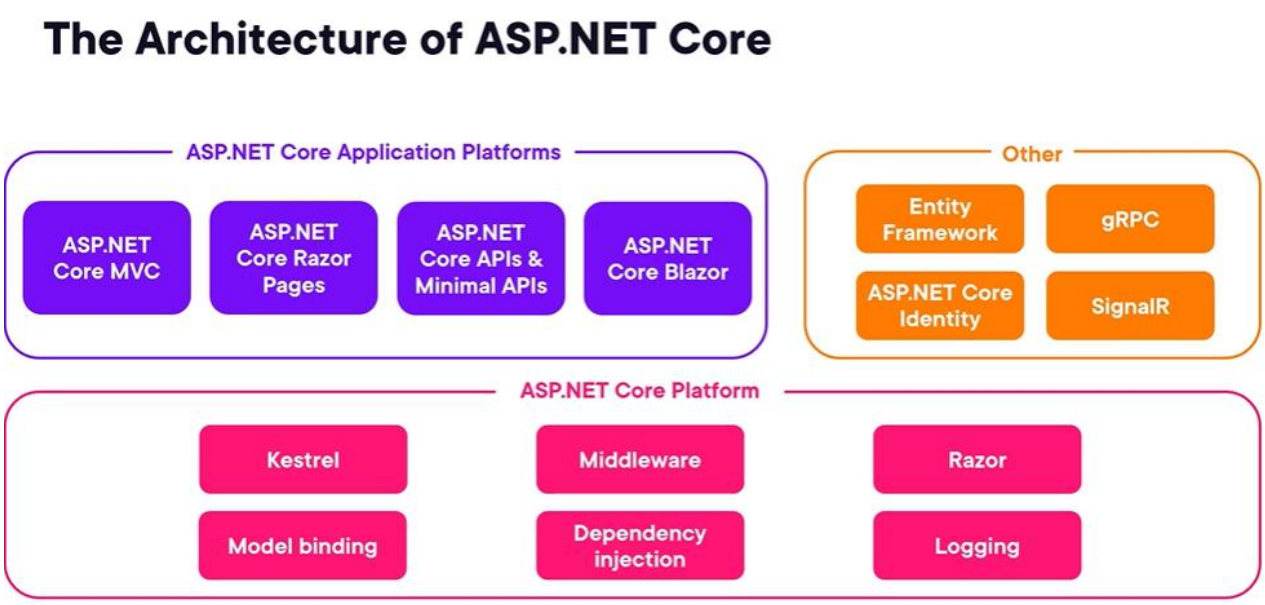
# Guida scritta da Marco Guerriero

# Riferimenti: www.pluralsight.com, by Gill Cleeren

## Indice

# Understanding ASP.NET Core and ASP.NET Core MVC




**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**:

- **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).**

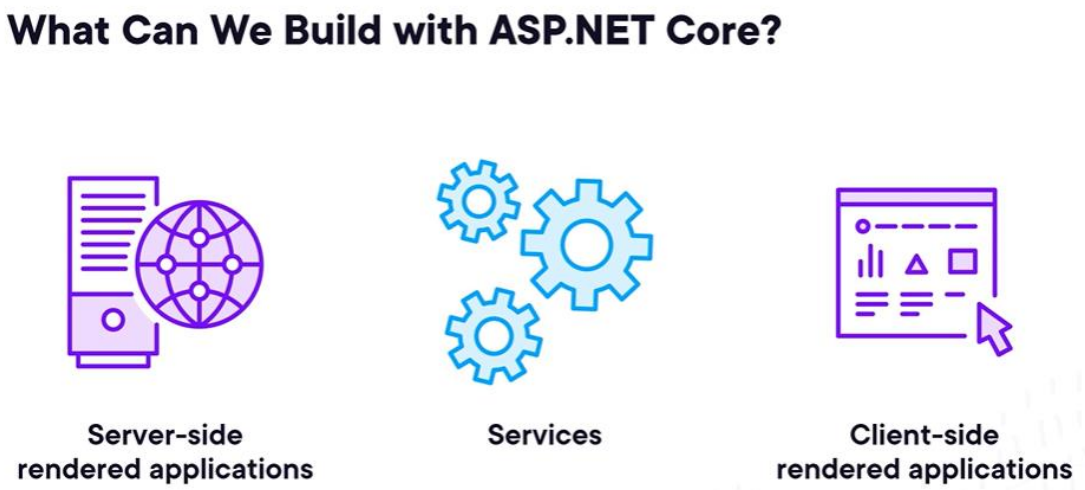
2. **Routing**:
   - **ASP.NET utilizza un sistema di routing per mappare le URL alle azioni del Controller o alle pagine specifiche.**
   - **Questo permette di definire URL amichevoli e organizzare meglio le risorse dell'applicazione.**

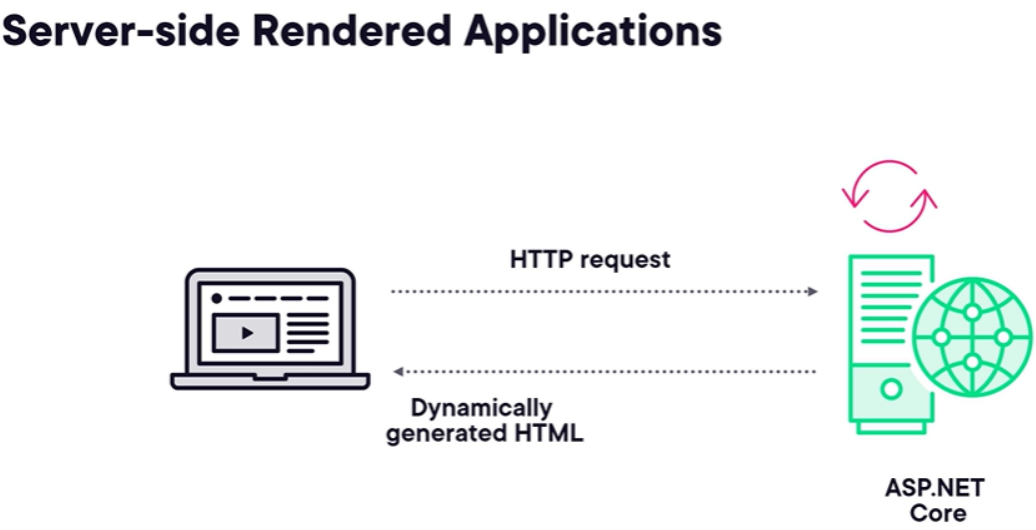
3. **Sicurezza**:
   - **Supporta vari metodi di autenticazione (OAuth, JWT, cookie-based authentication, ecc.) e autorizzazione, permettendo agli sviluppatori di proteggere le risorse.**
   - **Integrato con sistemi di sicurezza come Identity per gestire utenti, ruoli e permessi.**

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.**

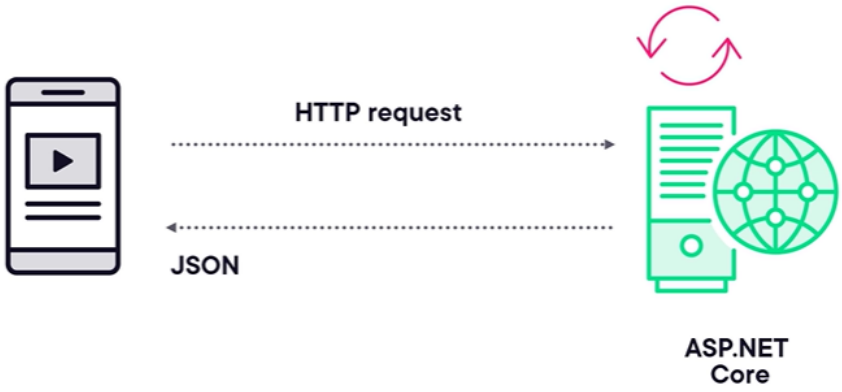
**Perché usare ASP.NET:**

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.

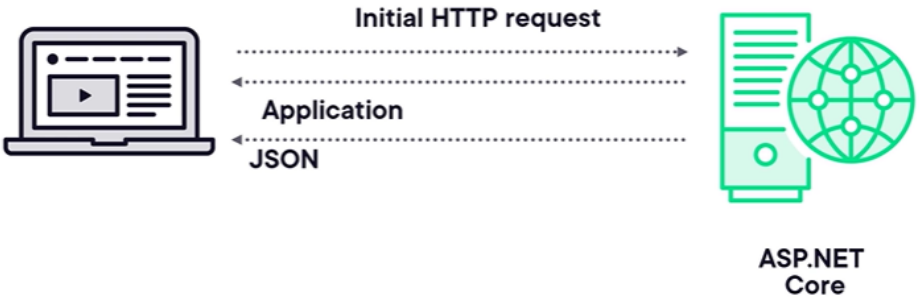


## ASP.NET Core Services



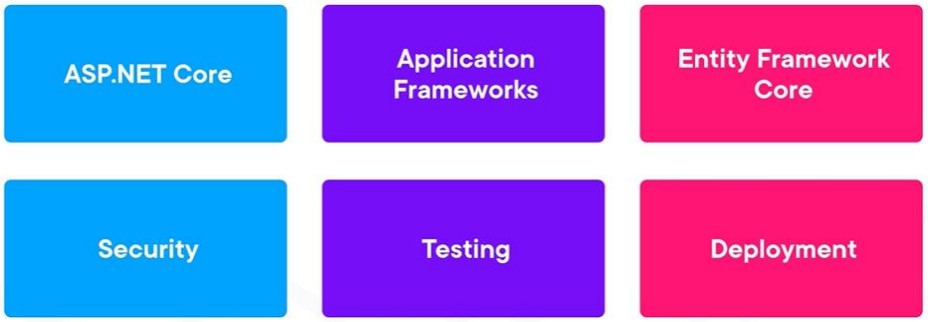

## Client-side Rendered Applications



# Setting up ASP.NET Core MVC Application

- Apprendimento
- Obbiettivo
- 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

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


**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**.

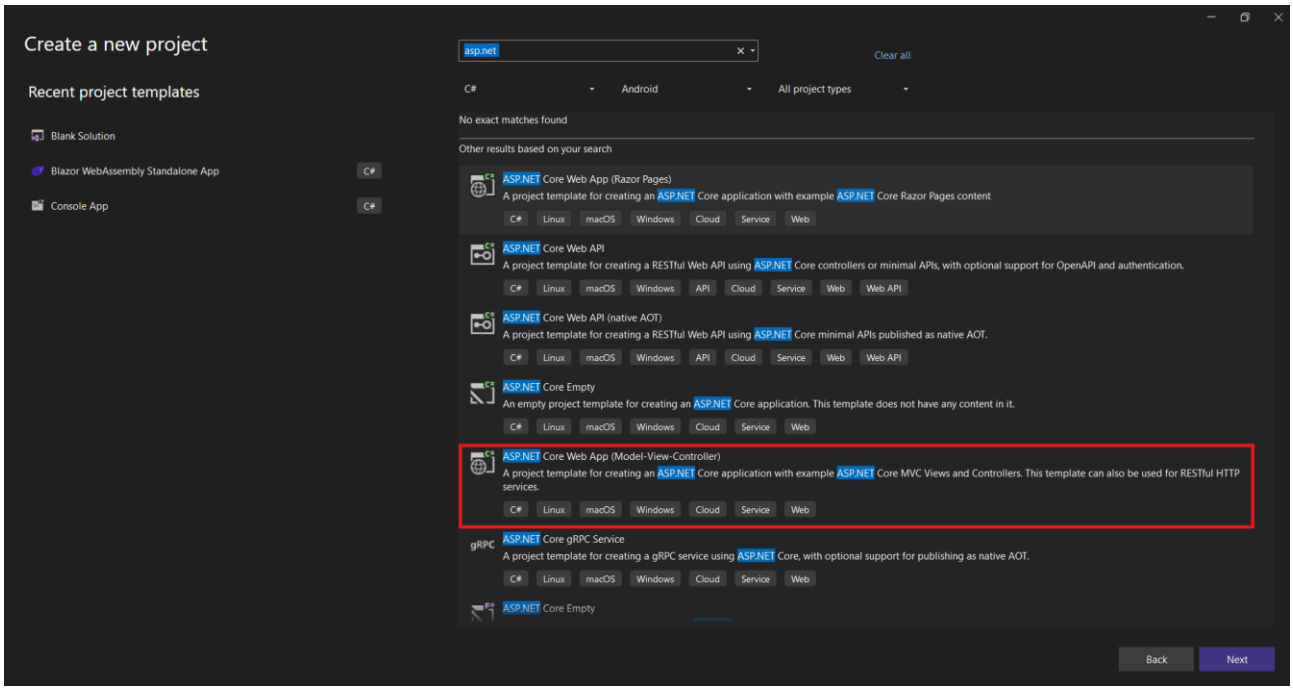
**Getting your machine ready**:
Visual Studio 2002
.NET 8

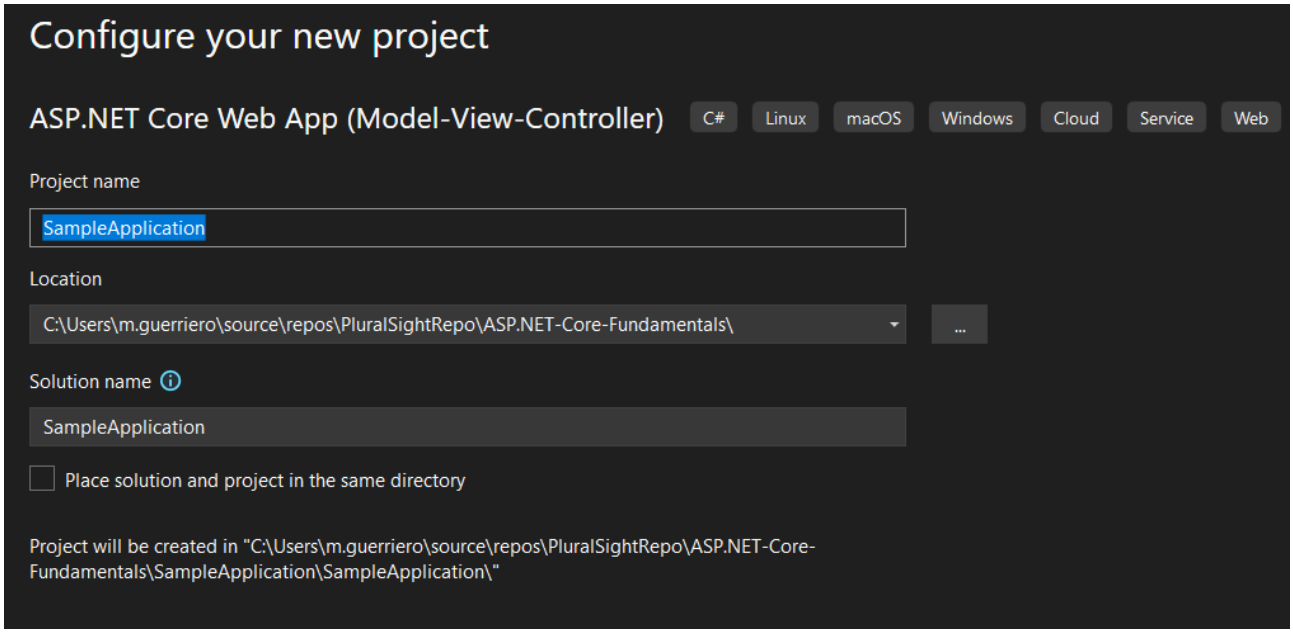
**Setting up an ASP.NET Core MVC Application**

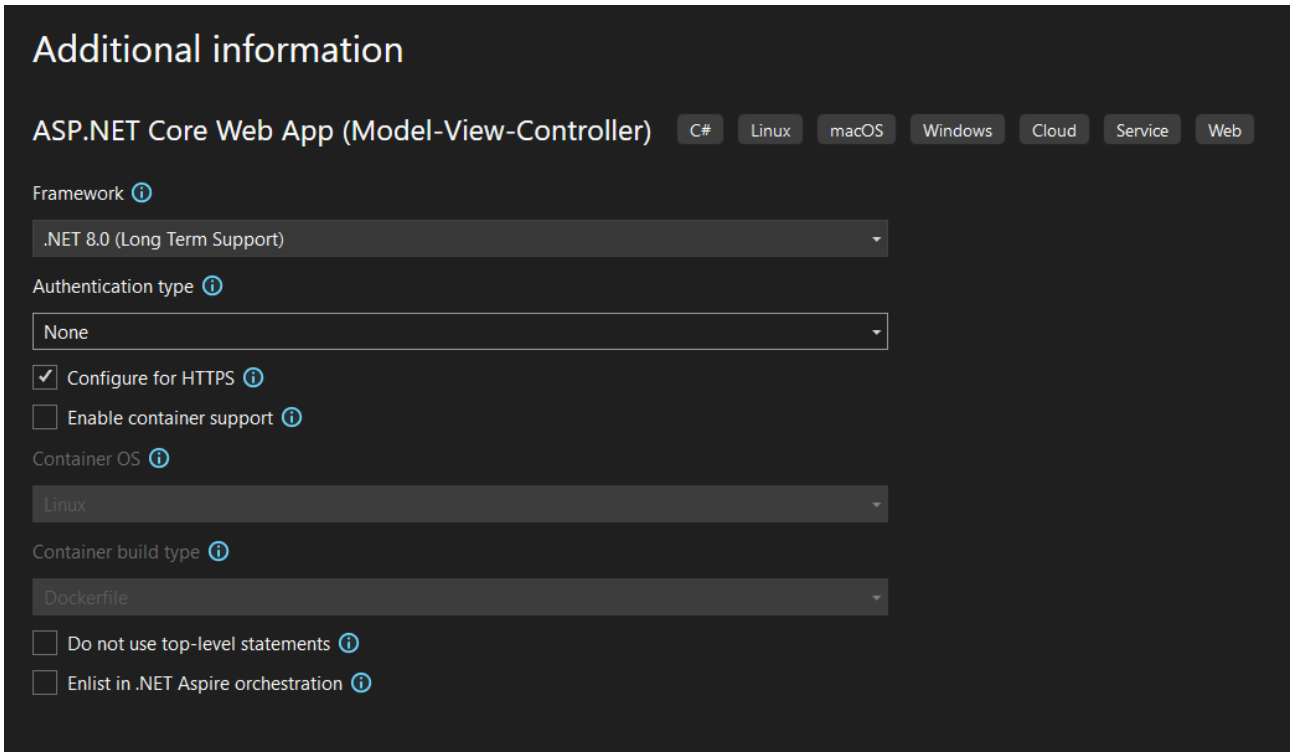
- Creating a new project
- Exploring the generated files
- Configuring the site
- How ASP.NET Core handles a request

**Creating a new project**

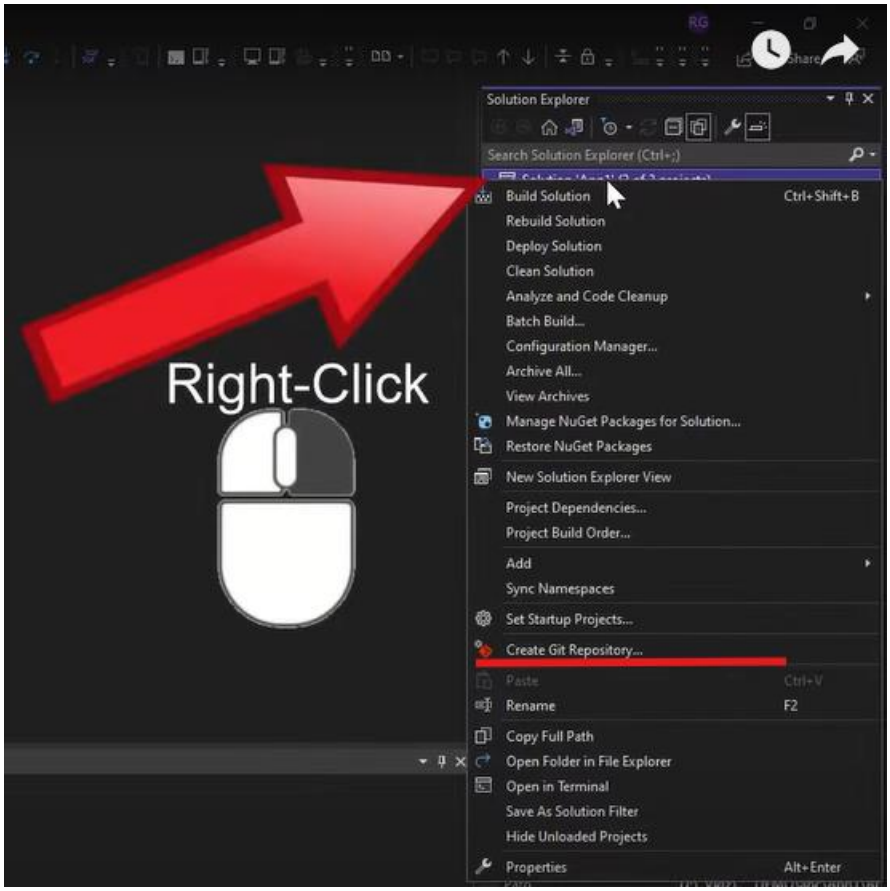
**TRAMITE INTERFACCIA VISUAL STUDIO (Windows)**


Create a new project
Recent project templates
Blank Solution
Blazor WebAssembly Standalone App
Console App
asp.net
Clear all
C#
Android
All project types
No exact matches found
Other results based on your search
ASP.NET Core Web App (Razor Pages)
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content
ASP.NET Core Web API
A project template for creating a RESTful Web API using ASP.NET Core controllers or minimal APIs, with optional support for OpenAPI and authentication.
ASP.NET Core Web API (native AOT)
A project template for creating a RESTful Web API using ASP.NET Core minimal APIs published as native AOT.
ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
ASP.NET Core gRPC Service
A project template for creating a gRPC service using ASP.NET Core, with optional support for publishing as native AOT.
Back
Next

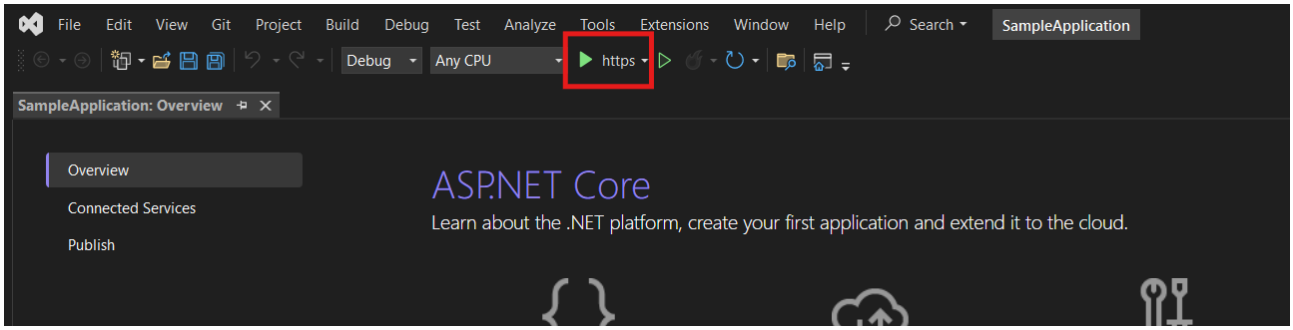

Configure your new project
ASP.NET Core Web App (Model-View-Controller)
C#
Linux
macOS
Windows
Cloud
Service
Web
Project name
SampleApplication
Location
C:\Users\m.guerriero\source\repos\PluralSightRepo\ASP.NET-Core-Fundamentals\
Solution name
SampleApplication
Place solution and project in the same directory
Project will be created in "C:\Users\m.guerriero\source\repos\PluralSightRepo\ASP.NET-Core-Fundamentals\SampleApplication\SampleApplication\"

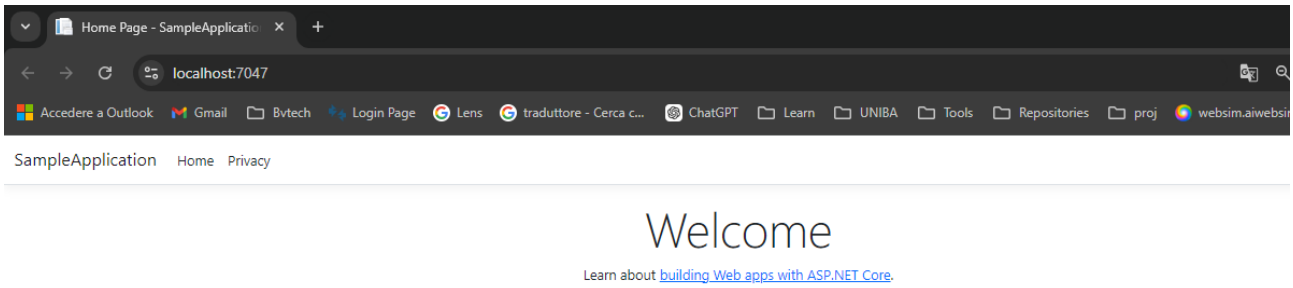



**Consiglio: Creare una repository GitHub da Visual Studio nel seguente modo:**



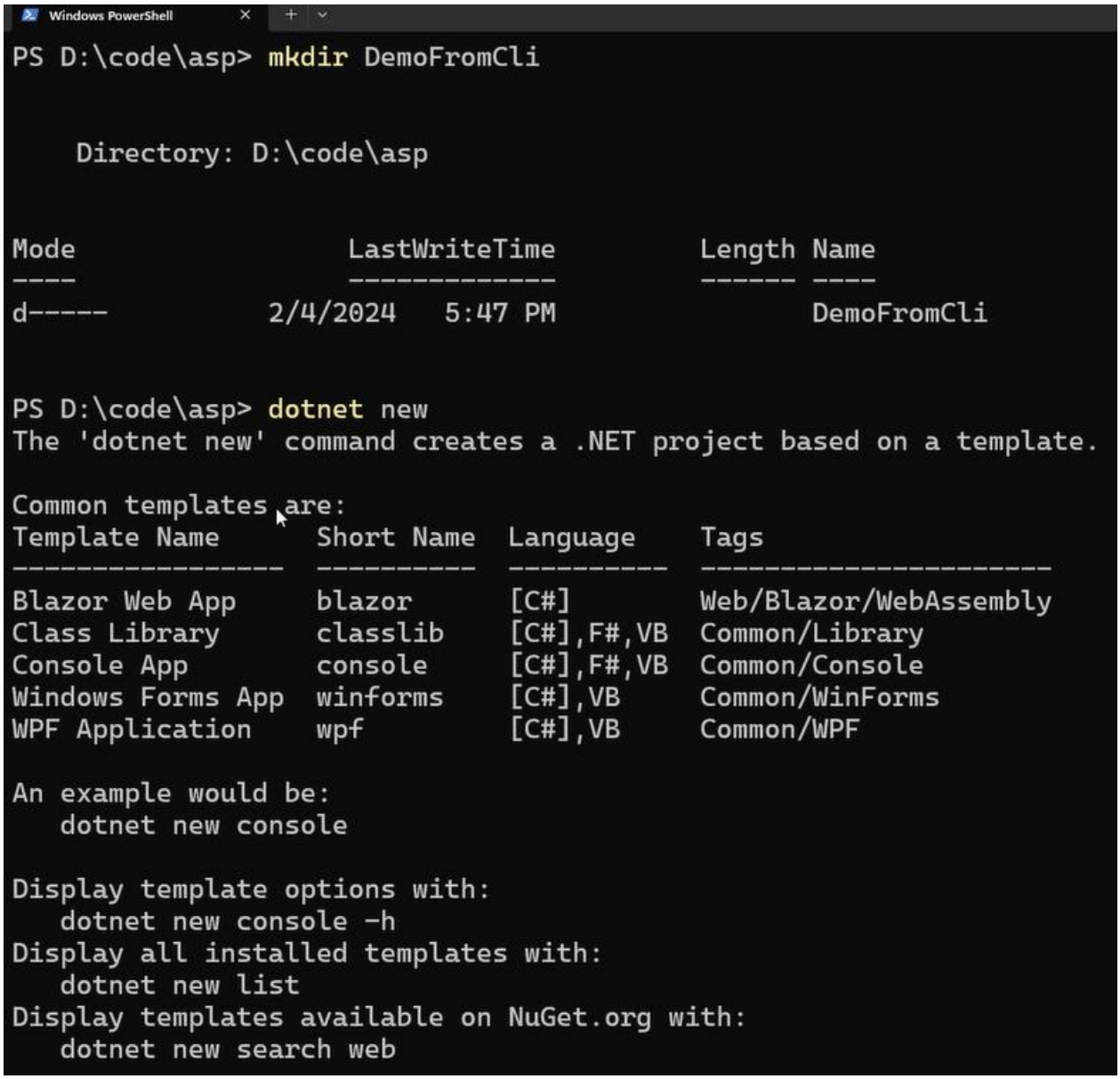
La nostra applicazione è già pronta per essere eseguita:

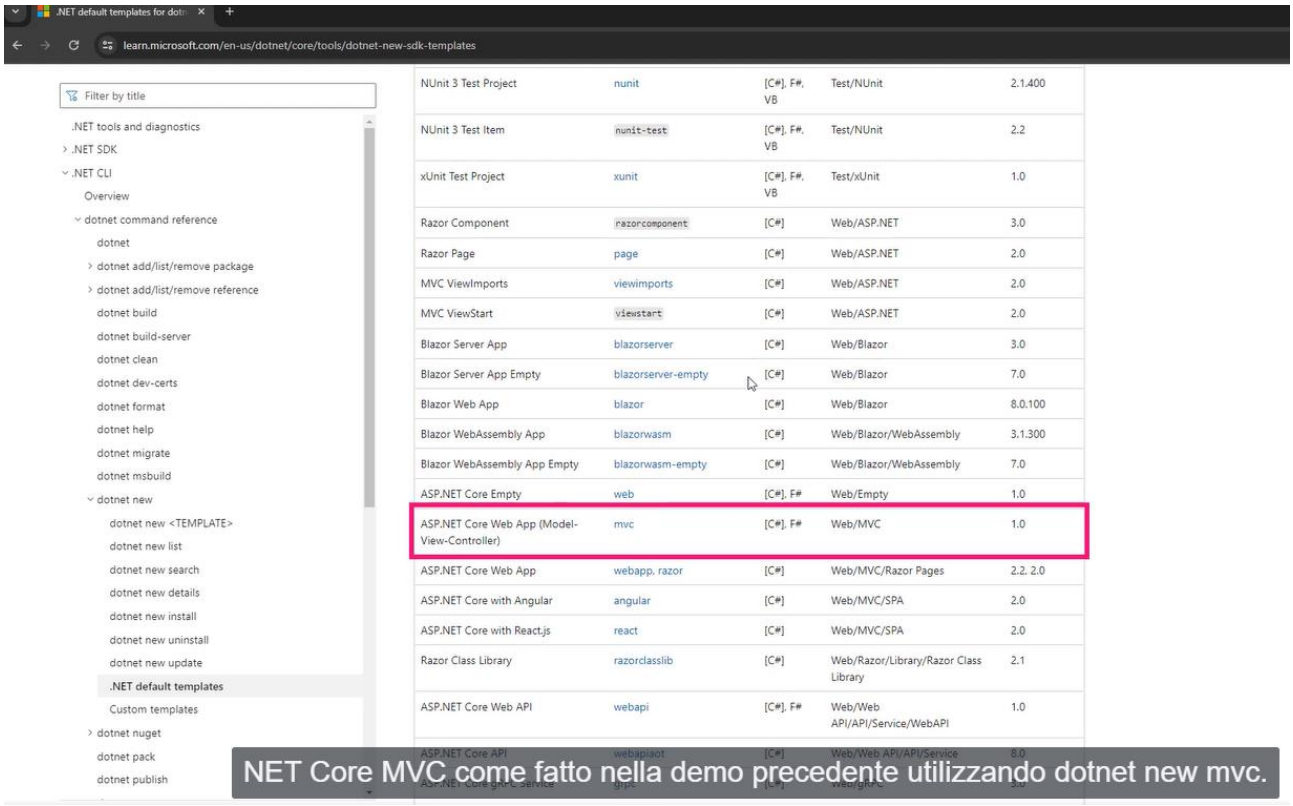


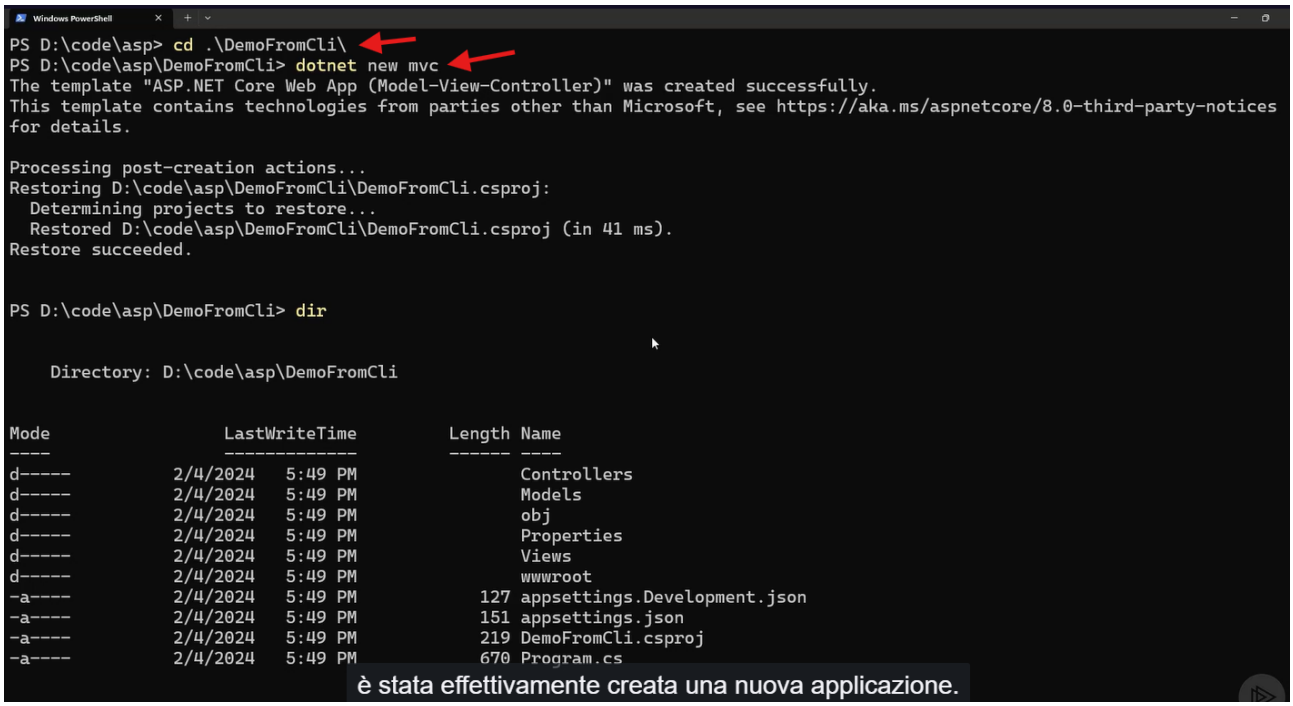

Inoltre è già responsive.

SE AVESSI VOLUTO CREARE TALE PROGETTO DA CLI:


Windows PowerShell
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

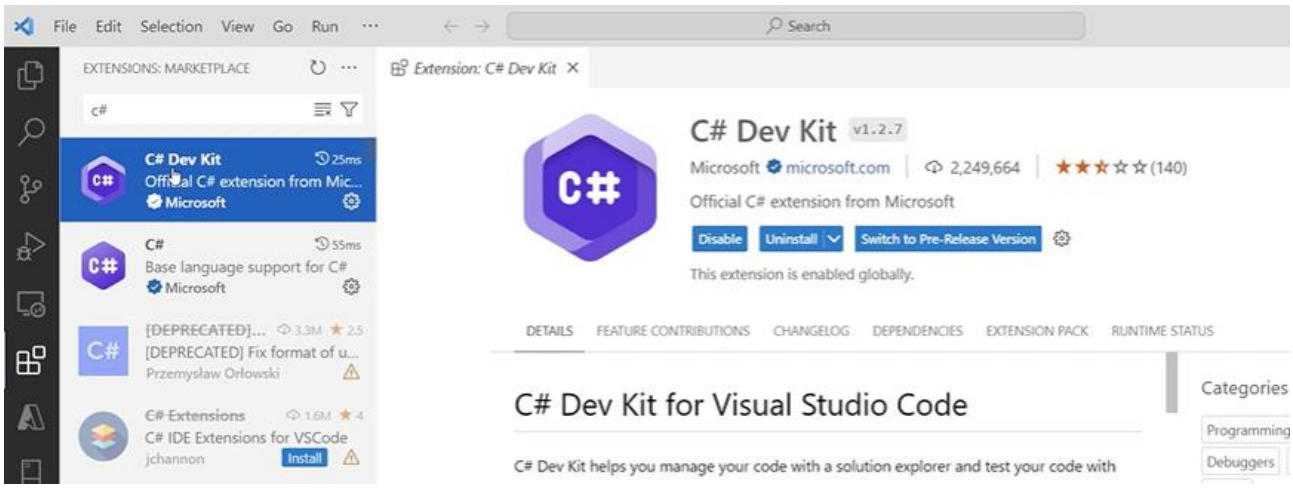


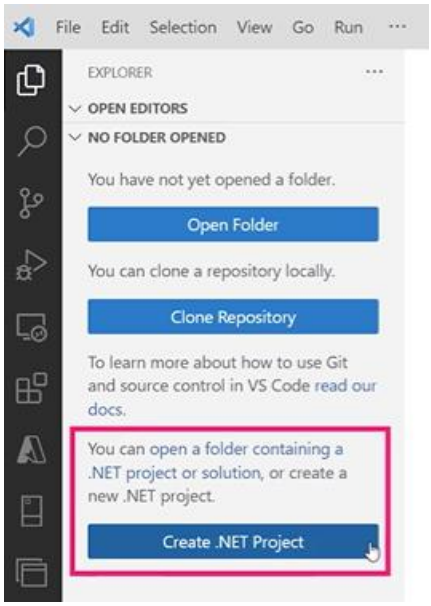

Dopo aver ricercato il tipo di architettura, nel nostro caso **mvc:**

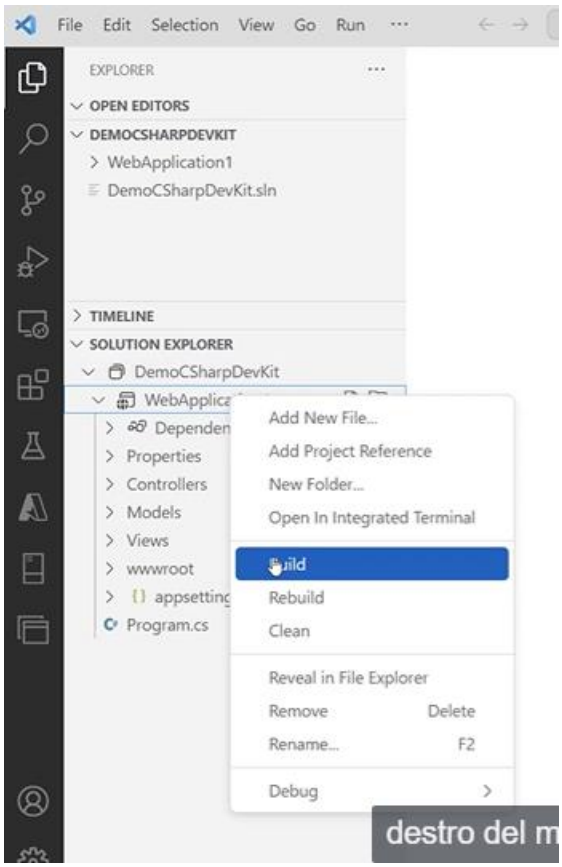



Successivamente con *dotnet run* eseguiamo l'applicazione.

SE AVESSIMO USATO VSC (quindi MacOs || Linux)



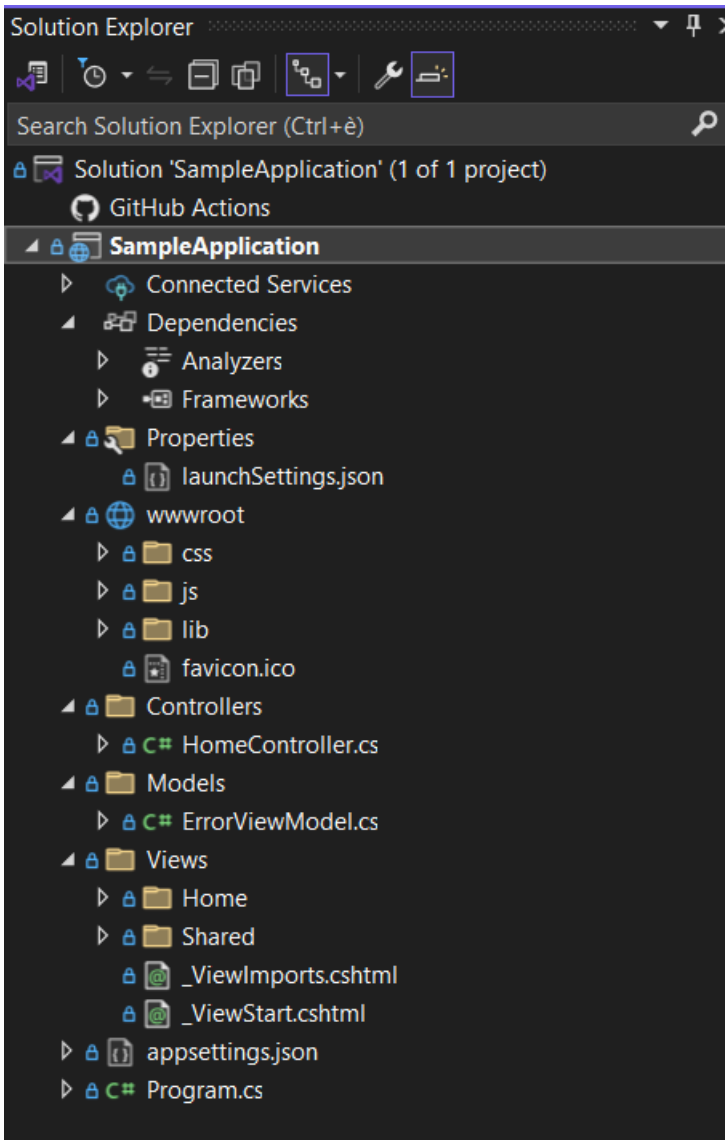






Per avviarlo basta cliccare Debug (voce presente nell'immagine precedente)

**Exploring the generated files**

Una soluzione raggruppa più progetti

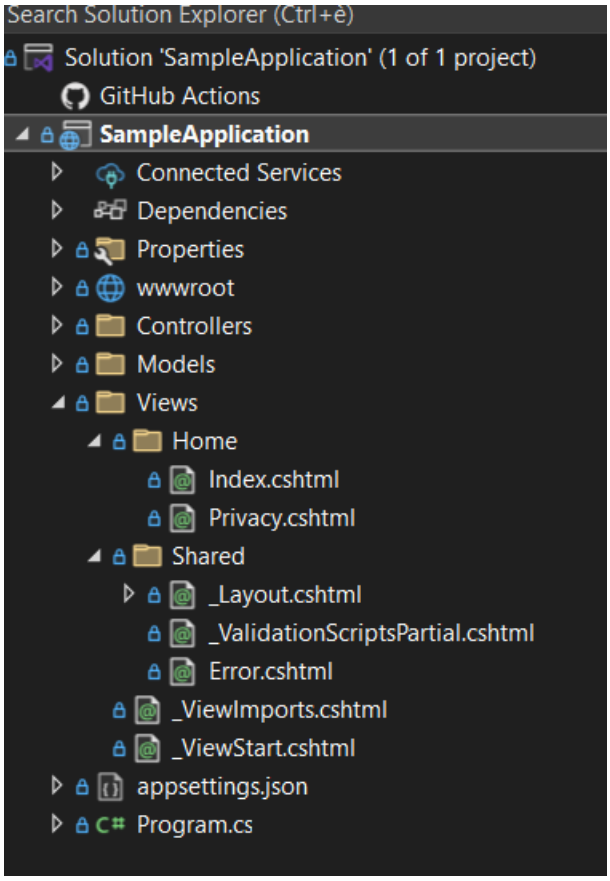



Viene creata una soluzione e un progetto al suo interno.
Servizi connessi ci consento di fare riferimento ad altri servizi

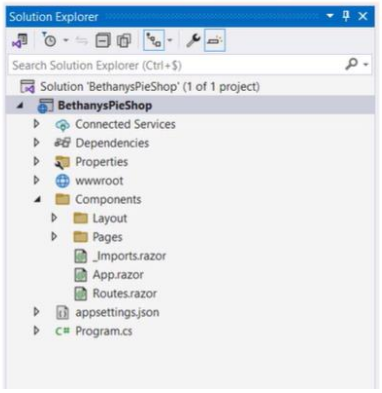
Le dipendenze elencheranno tutti i riferimenti che abbiamo nel nostro progetto (es: NuGet packages)

Proprieties contiene un file JSON chiamato launchsetting.json (lo esamineremo nella DEMO)
Program.cs contiene la classe program, ovverò il fulcro
appsetting.json, impostazione per archiviare e configurare e connettere al DB

Razor Pages:

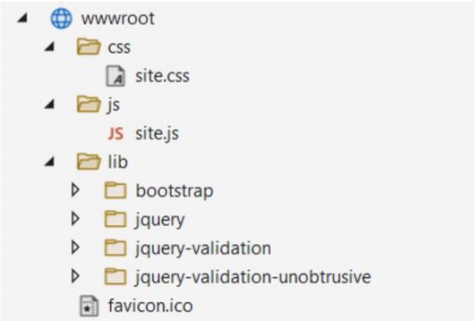



# Project Structure (Blazor Server-side Rendering)




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.

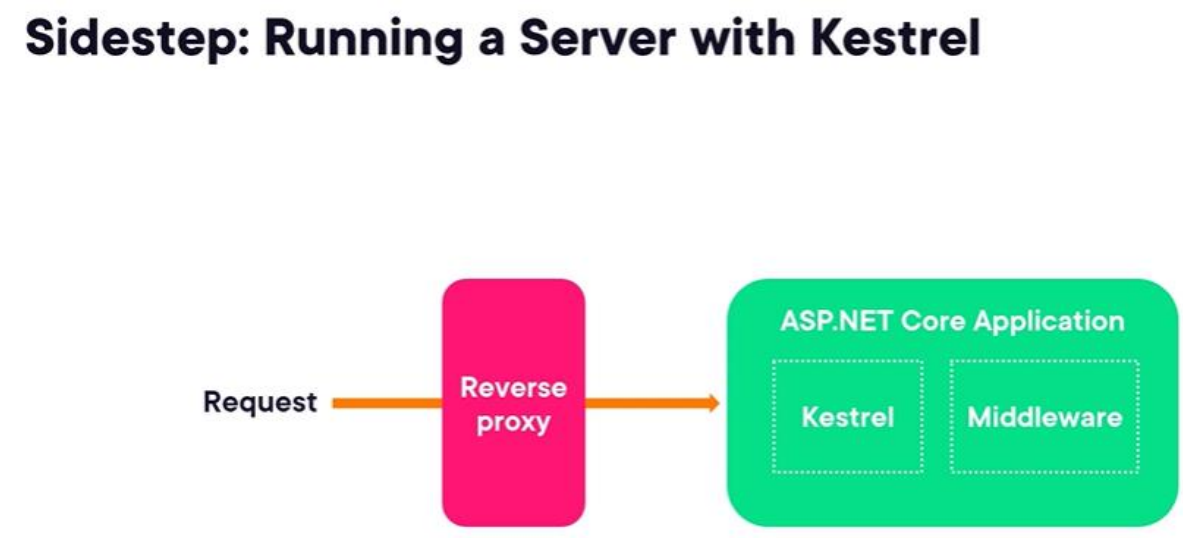
Il tutto avverà proprio all'interno di ASP.NET Core, più precisamente nel Kestrel

Program.cs*

SampleApplication

```
var builder = WebApplication.CreateBuilder(args); //Setta il Kestrel, ovvero il Server Multipiattaforma che gestisce le richieste

// 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();
```

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


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

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, dovrei avere una file web.config

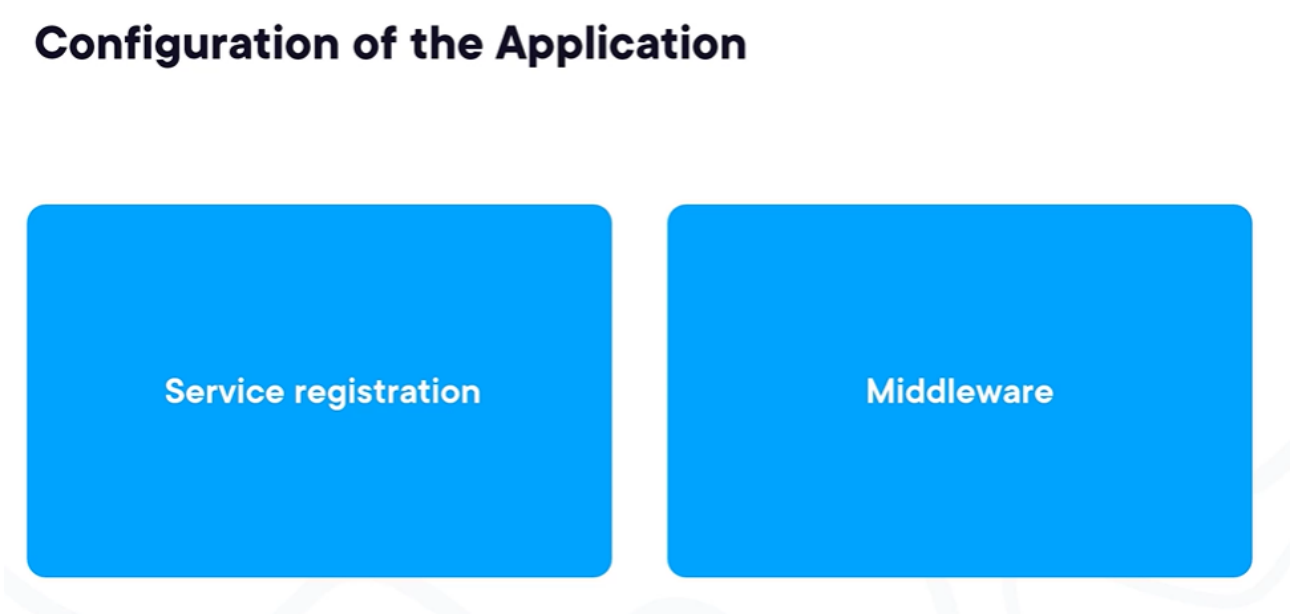
il file **web.config** non è presente perché non è ancora necessario. Verrà creato automaticamente quando pubblichi l'applicazione per **IIS**.

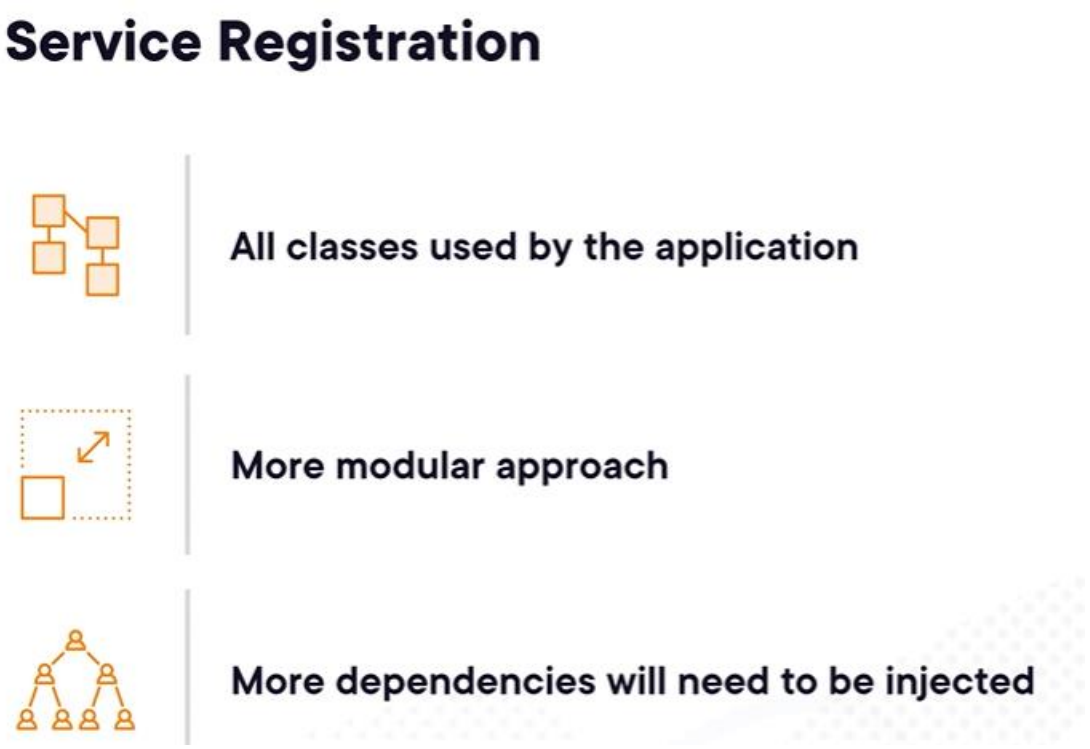
**IIS** (Internet Information Services) è un **server web** sviluppato da Microsoft per ospitare applicazioni web su **Windows**. Gestisce richieste HTTP e HTTPS, fornendo servizi come:

- **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**



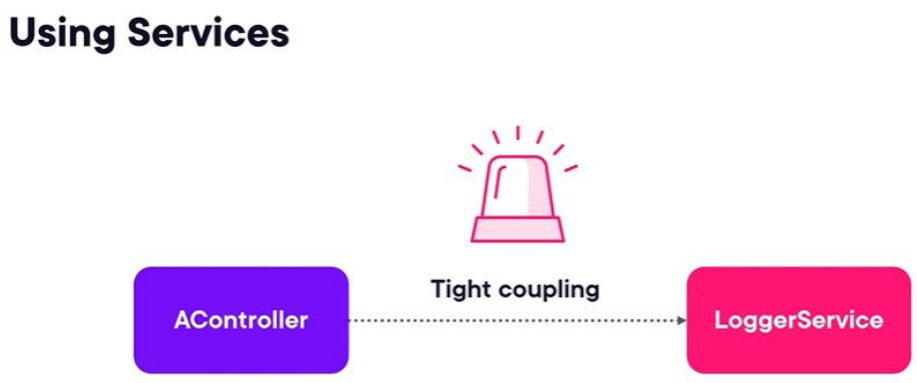



**Service registration**:
Tutte le classi che possiamo utilizzare nella nostra applicazione vengono chiamate **SERVIZI**
Codice modulare che utilizza piccole classi, queste classi dovranno essere istanziate e qui si utilizza la registrazione del servizio.

**Ipotizziamo di avere la seguente situazione**:

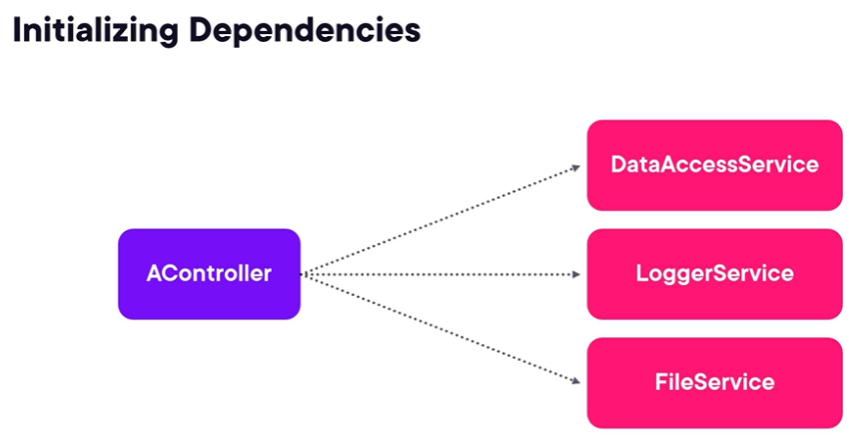
Aacontroller: Classe che dovrà eseguire alcune registrazioni, ci riferiamo alla funzionalità di registrazione di una classe con un servizio

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 registra e lasciarle eseguire la registrazione.




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

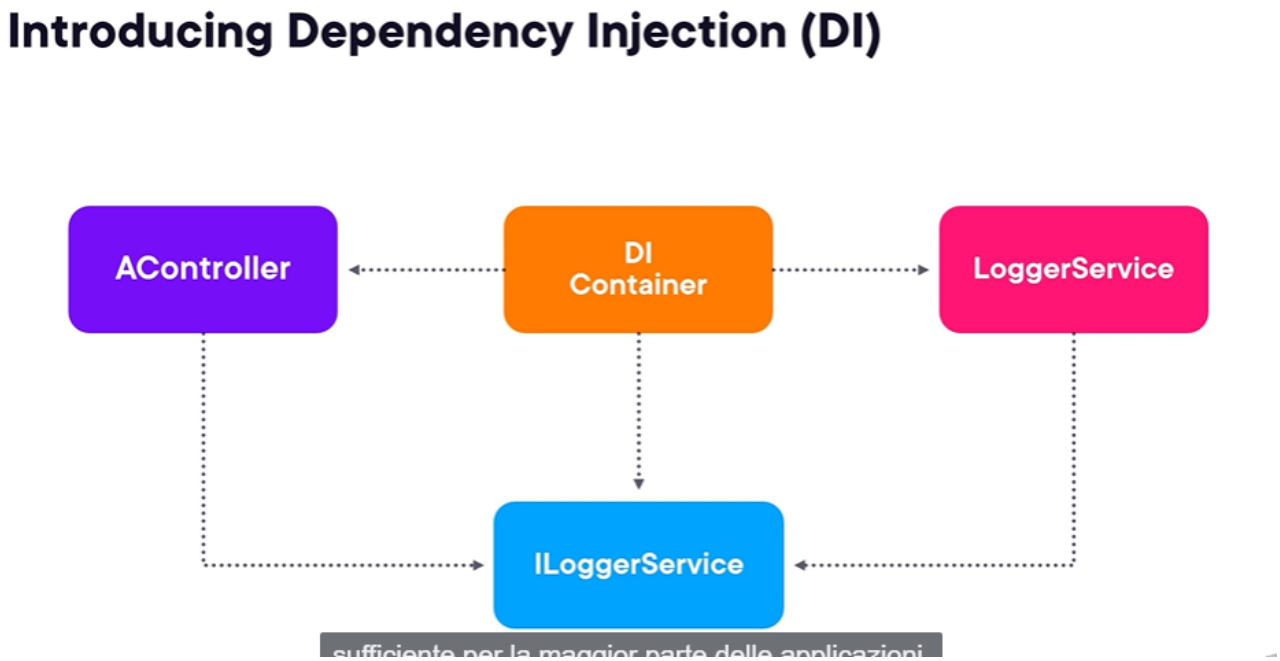
Vorrà utilizzare anche altri servizi come:



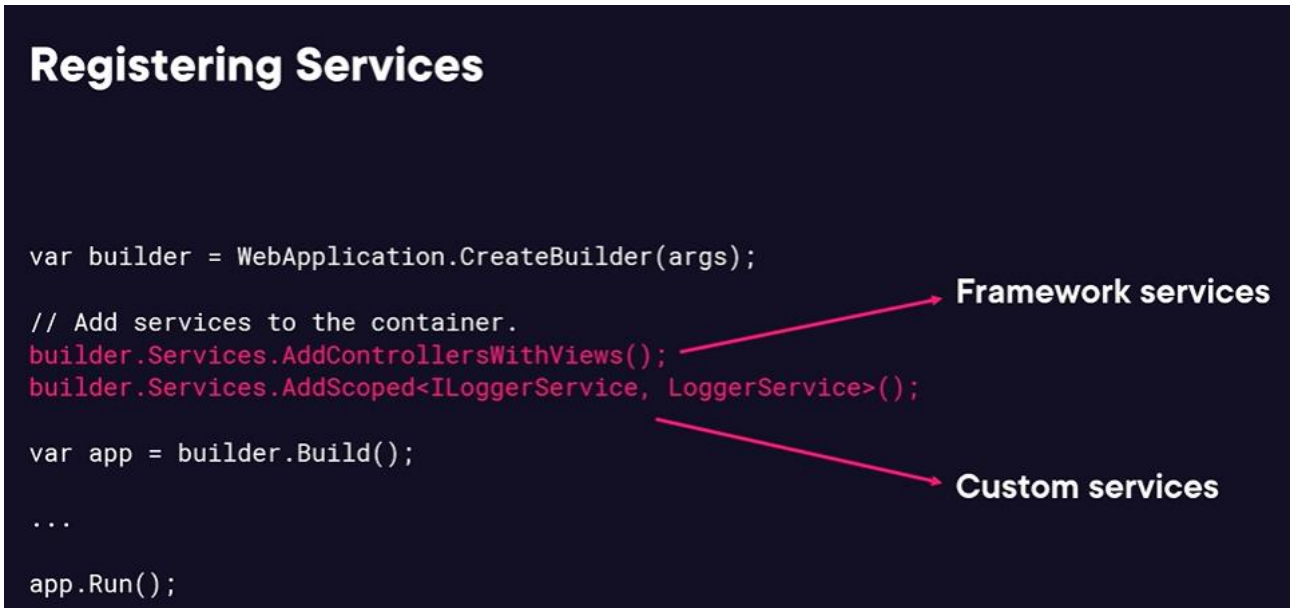

Quindi quando volgiore creare una nuova istanza di Acontroller sarò responsabile de tutte questa altre istanze (che sono pesanti)

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

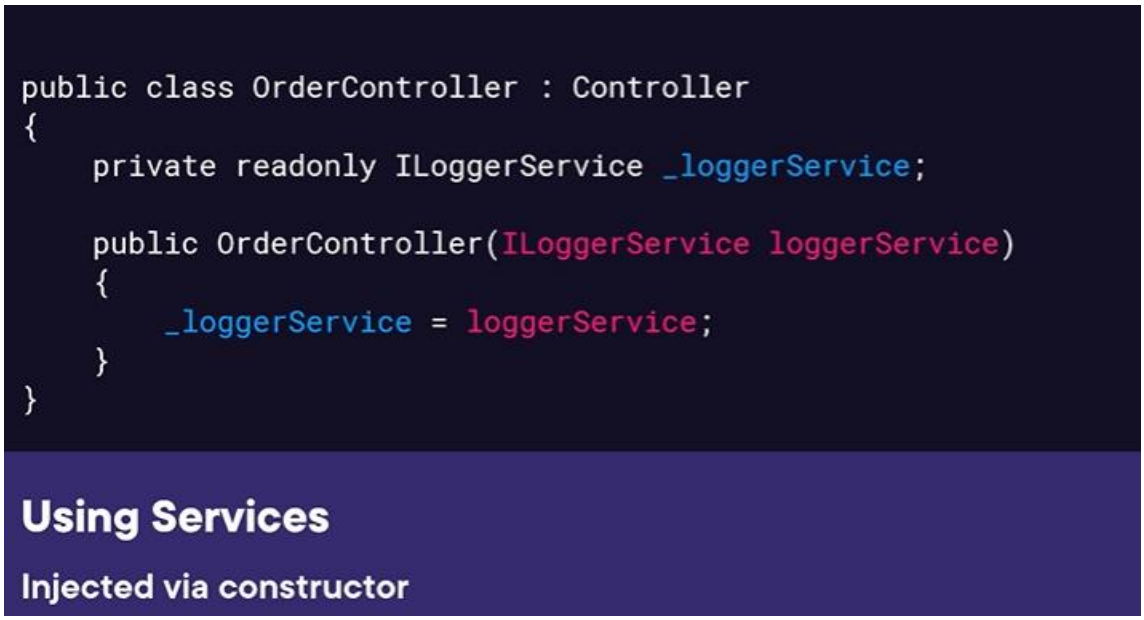
Una 3° parte ILoggerService crea le istanze:




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 raccola possiamo aggiungere i nostri servizi:**

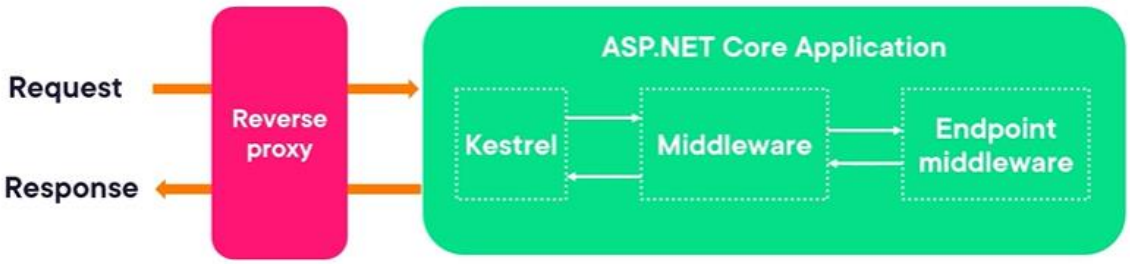



Una volta registrato, dove ora abbiamo bisogno di un'istanza, la faremo **INIETTARE OrderController usa ILoggerService, il servizio che abbiamo registrato nel nosto contenitore:**



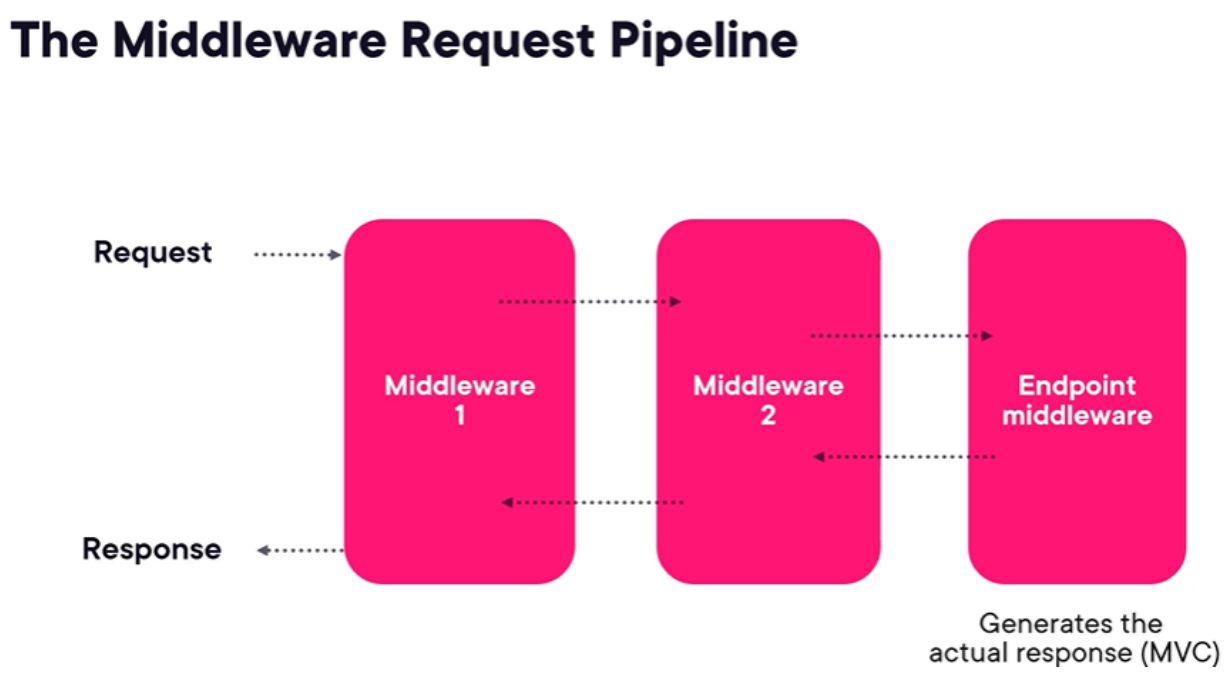

Quindi, la registrazione dei servizi era uno dei compiti del Program.cs, un altro compito o il secondo è quello della creazione del **middleware:**

# Handling Requests with Middleware

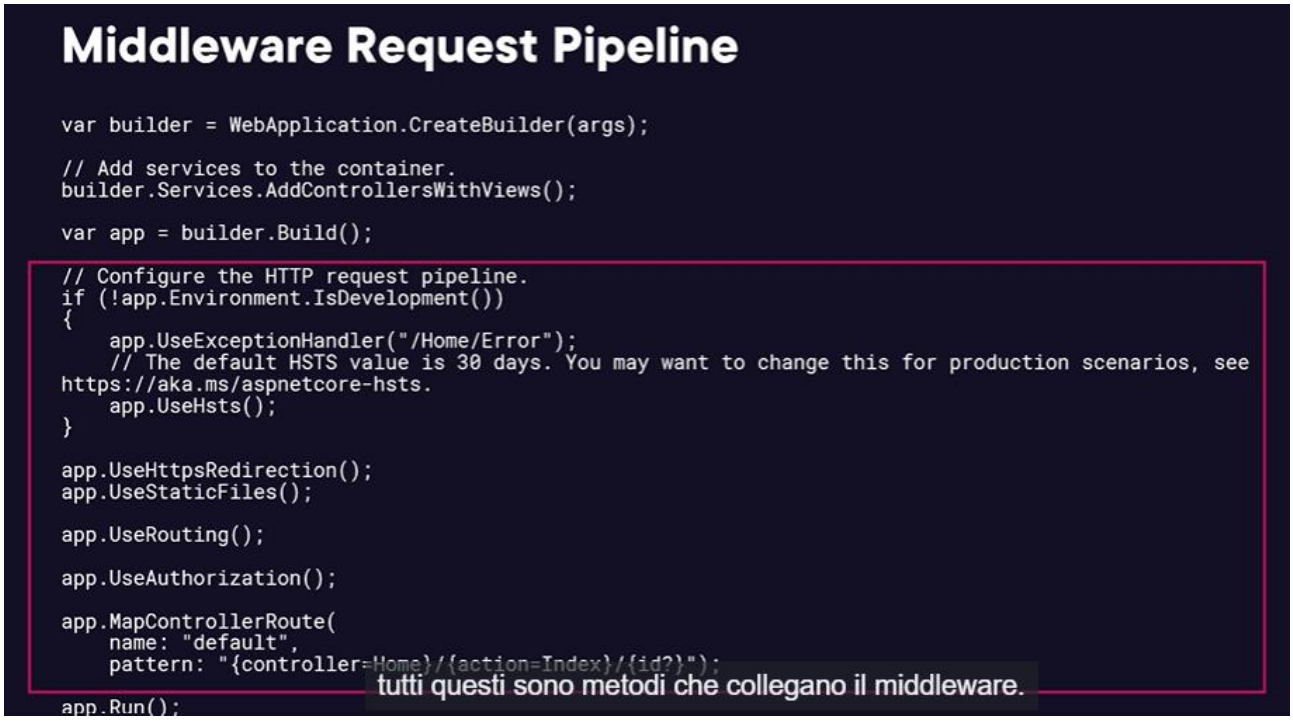



# The Middleware Request Pipeline

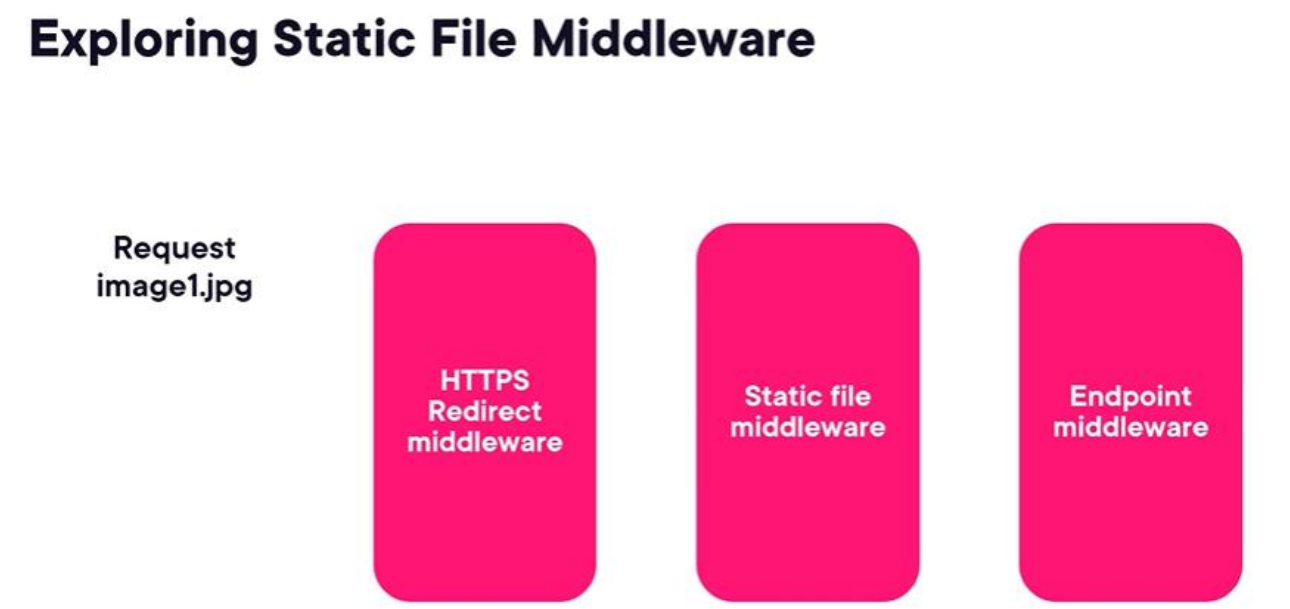
- Pipeline consists out of set of components
- Components work on request or response
- Many built-in components
- Endpoint middleware sits at the end

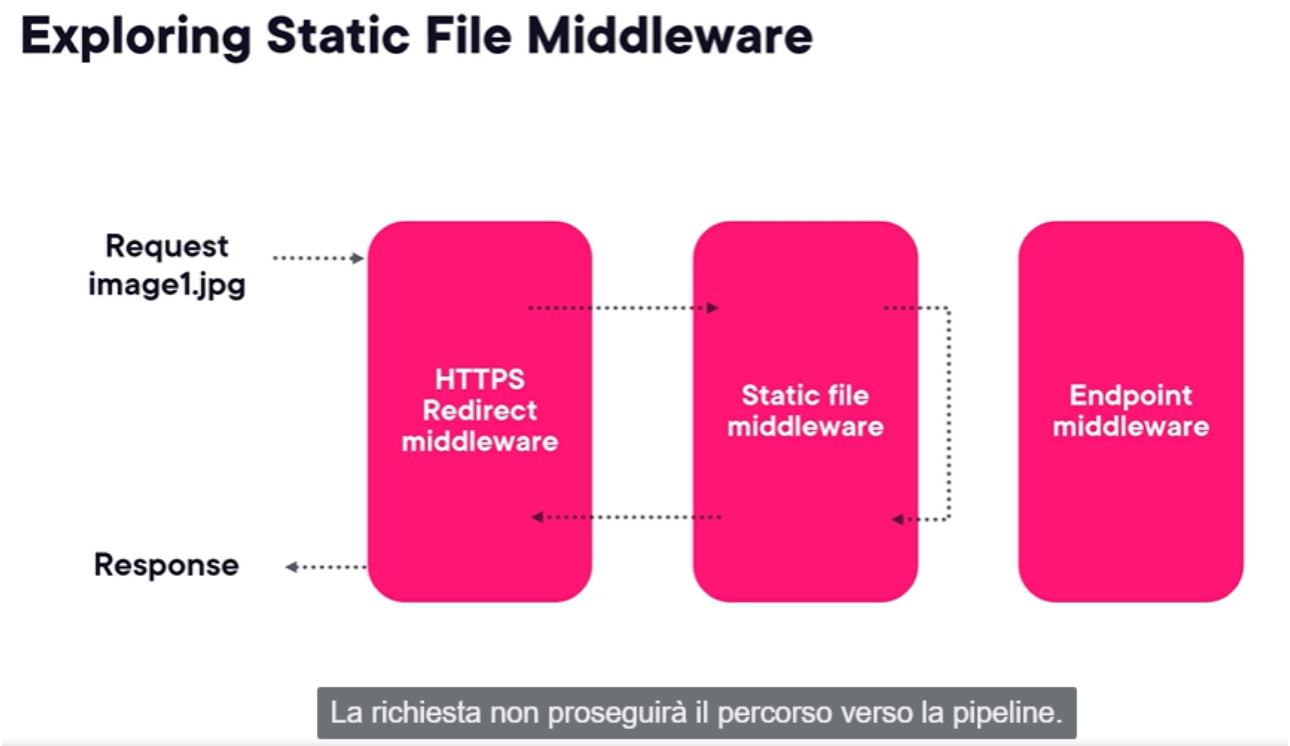



**Middleware nel Program.cs:**




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

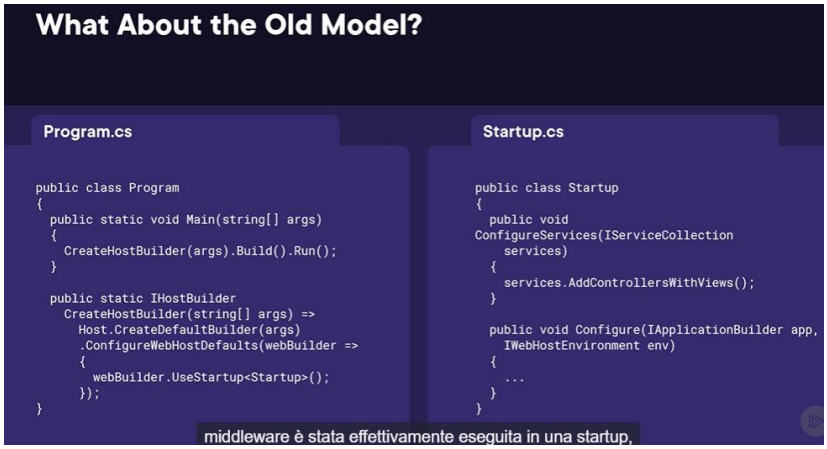



**Osservazione: La richiesta non prosegue, in quanto ha trovato già l'immagine nella sezione dei file statici, presente nella seconda componente del Middleware.**

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.

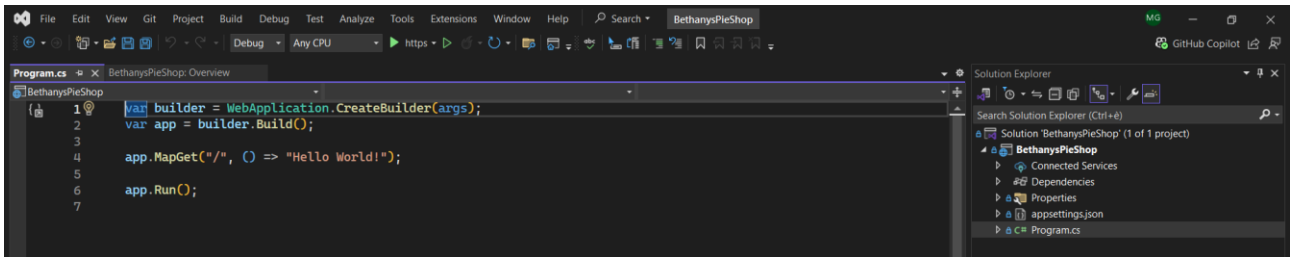
In ASP.NET.6 e quindi in modelli più vecchi, Program.cs era diviso in due parte, quindi era presenta anche Startup.cs:




**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**:



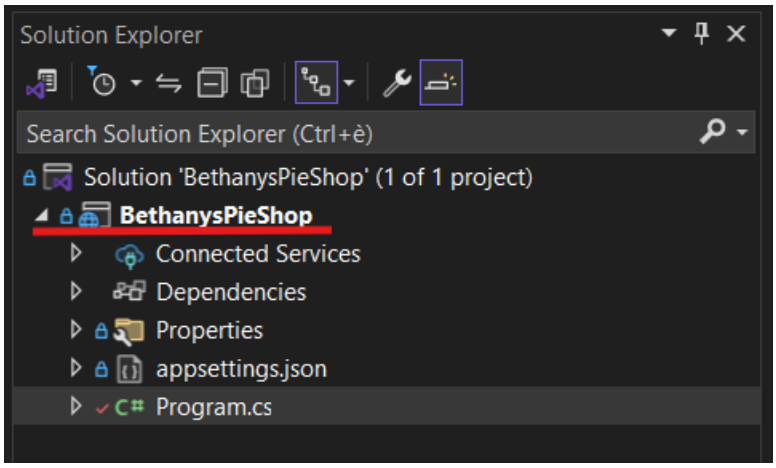

Partiamo dal file più importante, Program.cs
Attualmente non esiste una classe, non esiste un dominio statico.
Verrà quindi eseguito come un app console

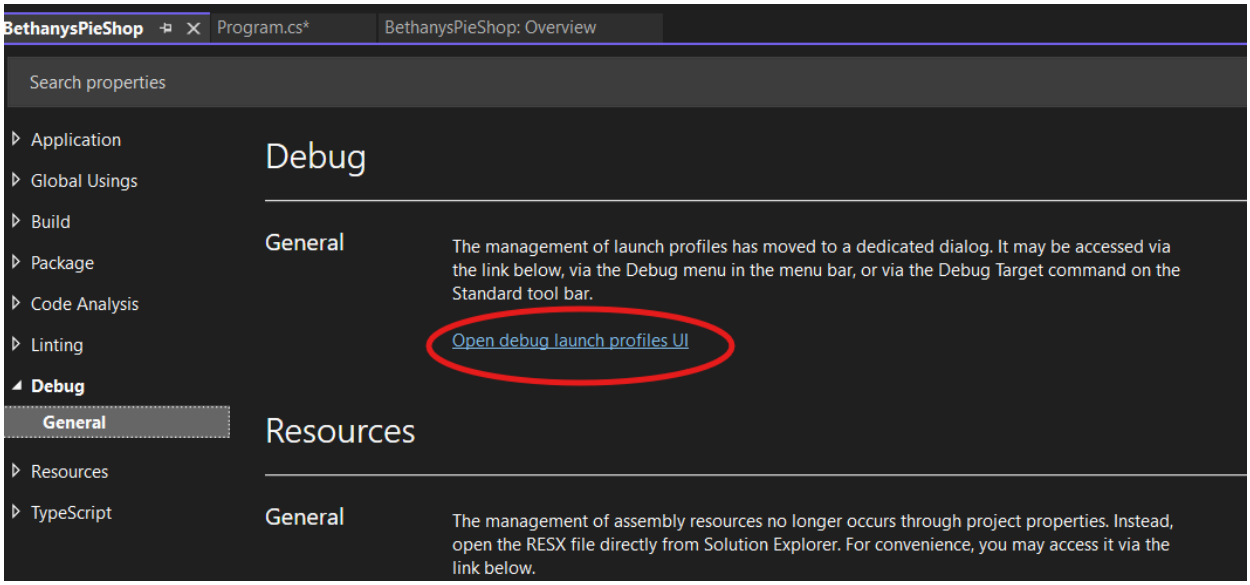
**Possiamo notare che è già presente il builder, il quale svolge operazioni di configurazioni già citate come:**
- caricare tutte le impostazioni presenti nel file appsettings.json;
- si assicura che il nostro Kestrel sia incluso e configurerà IIS;
- imposterà la cartella wwwroot che conterrà il contenuto 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:**

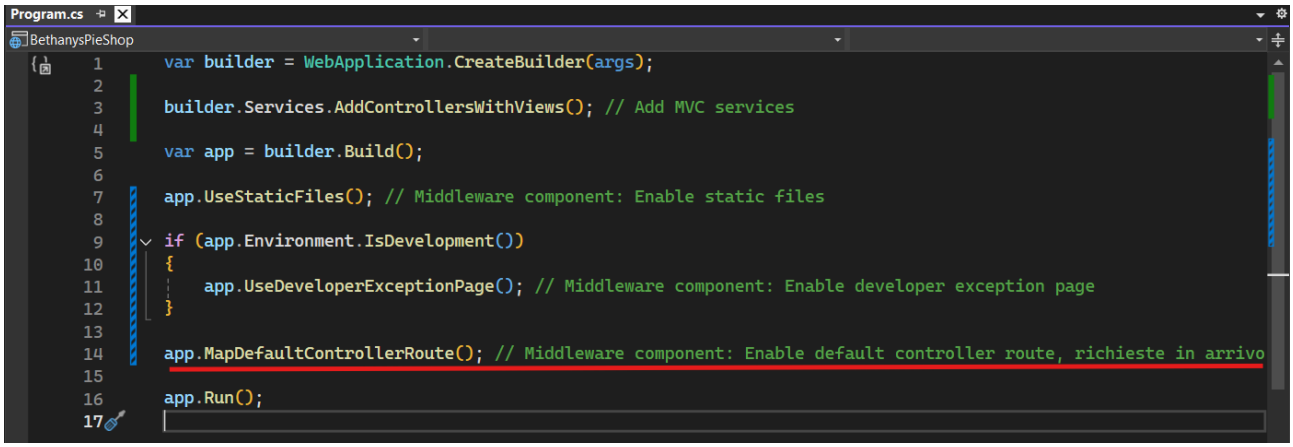
```
Program.cs*  BethanysPieShop: Overview
BethanysPieShop
1   var builder = WebApplication.CreateBuilder(args);
2
3   builder.Services.AddControllersWithViews(); // Add MVC services
4
5   var app = builder.Build();
6
7   app.UseStaticFiles(); // Middleware component: Enable static files
8
9   if (app.Environment.IsDevelopment())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

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



Aggiungiamo un altro componente del middleware:



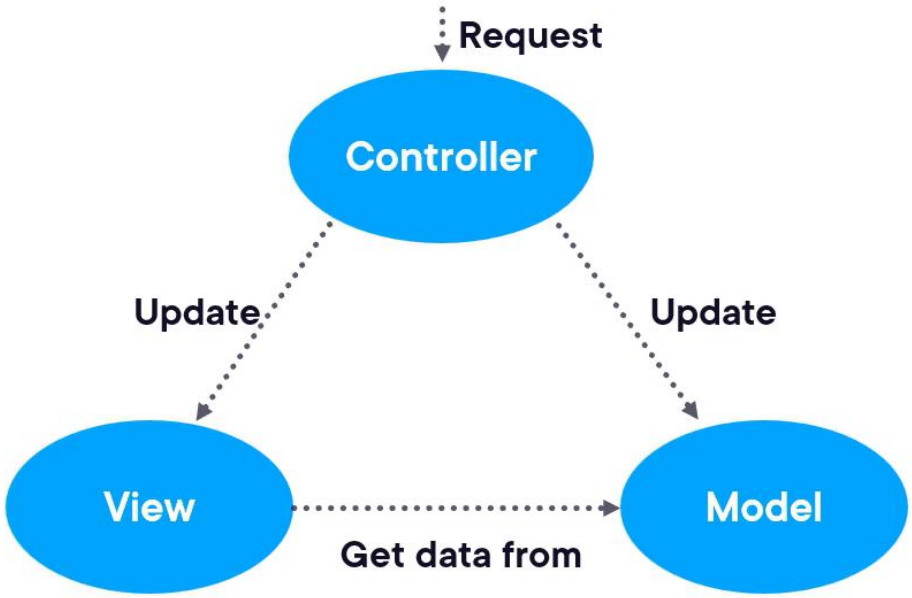

Se avviamo ora avremo un errore nel nostro client, in quanto non sono presenti ancora Views.

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

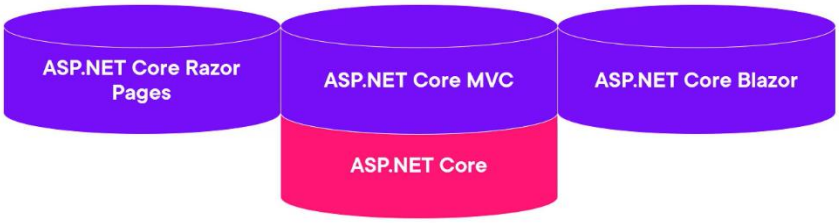
**MVC Pattern**



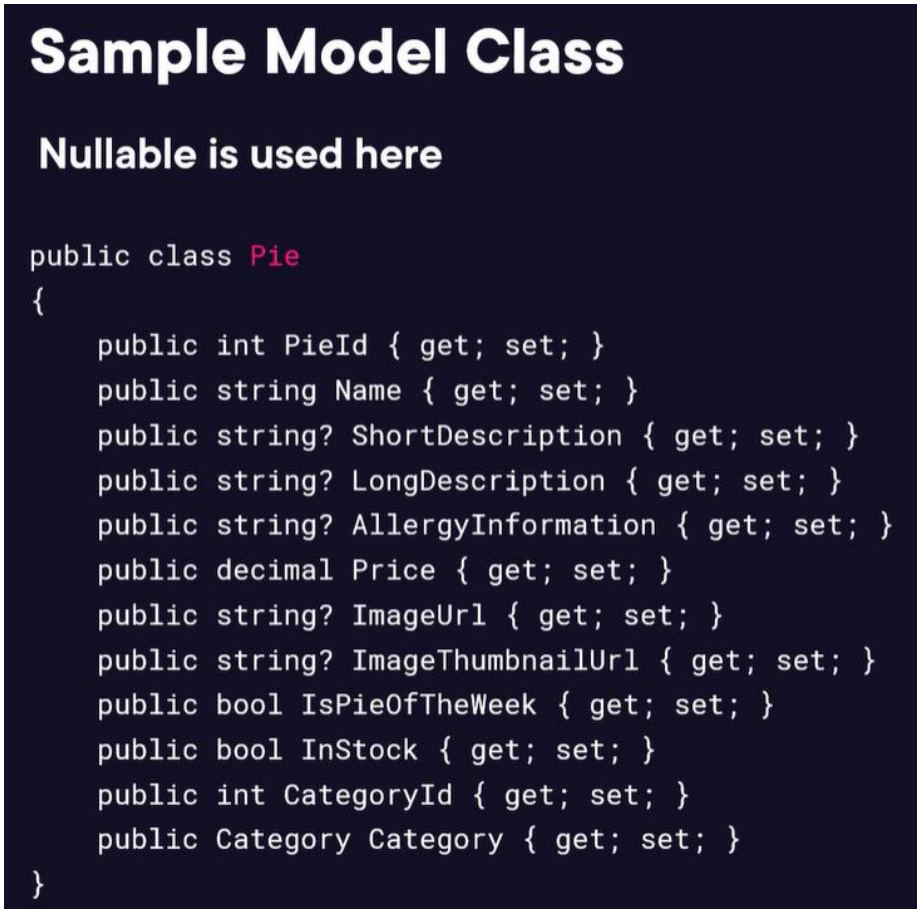
**Model**: Contengono i dati con cui lavora l'utente, ovvero classi del modello di dominio e le operazioni per lavorare con questi dati. Sono presenti anche classi che consentono l'interazione con il db, quindi anche i repository faranno parte del modello.

**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.

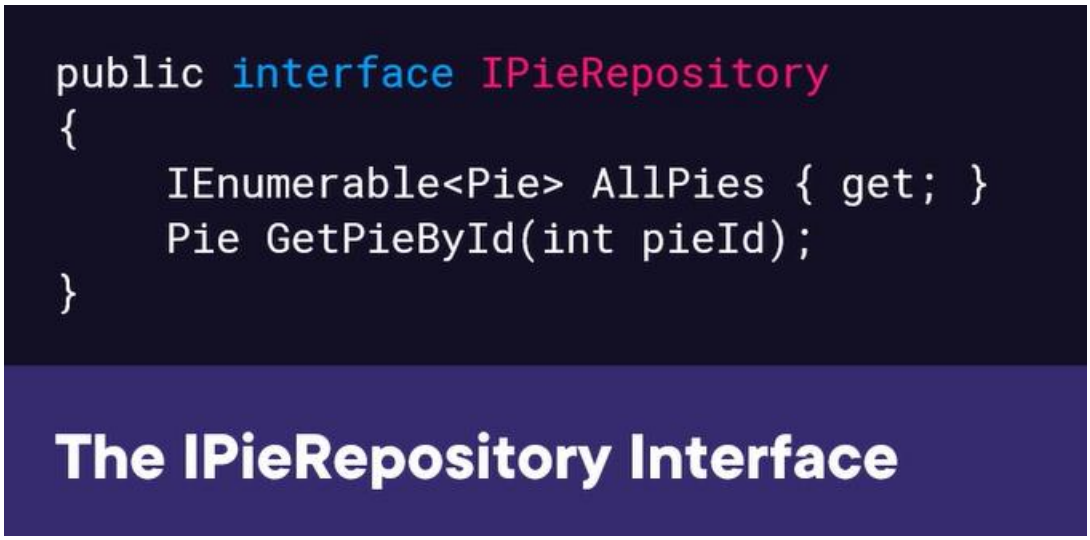
**View**: Conterrà la logica che mostra i dati all'utente in modo che possano, ancora una volta, essere elaborati da un'azione del controller

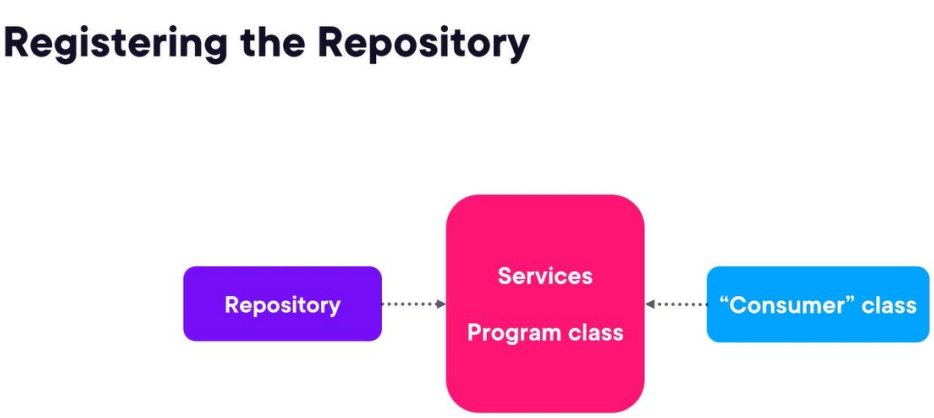



**Creating the Model and the Repository**



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



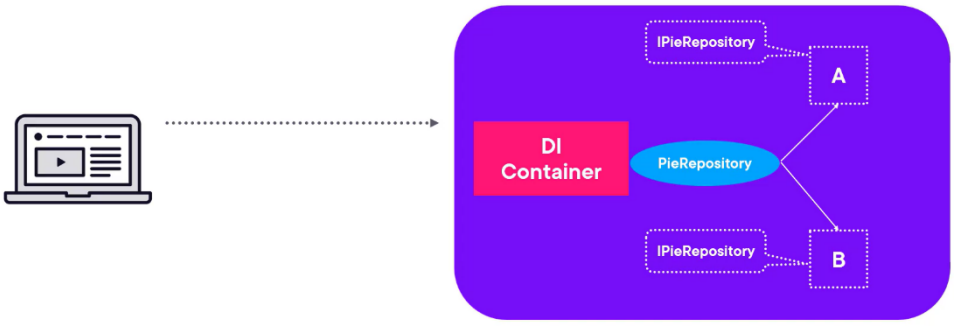
## Registering Services


- **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.)

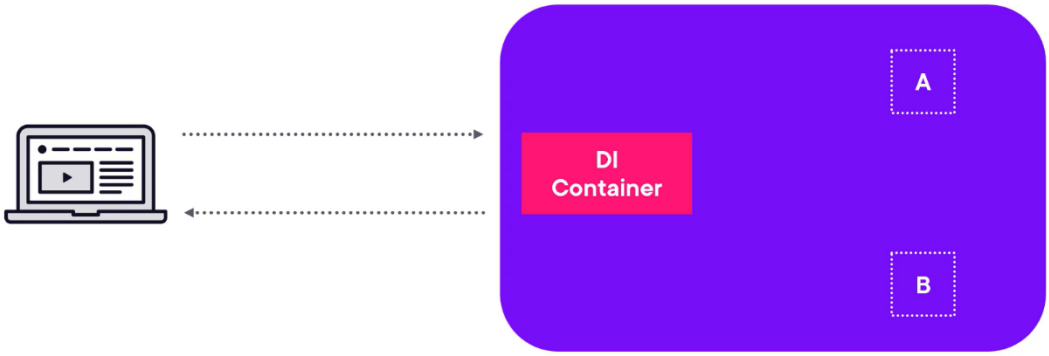
- **AddSingleton**: **Una sola istanza** per l'intera applicazione. (Quando il servizio deve **mantenere uno stato condiviso** e la creazione di più istanze sarebbe inefficiente, ad esempio per cache o impostazioni globali.)

- **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

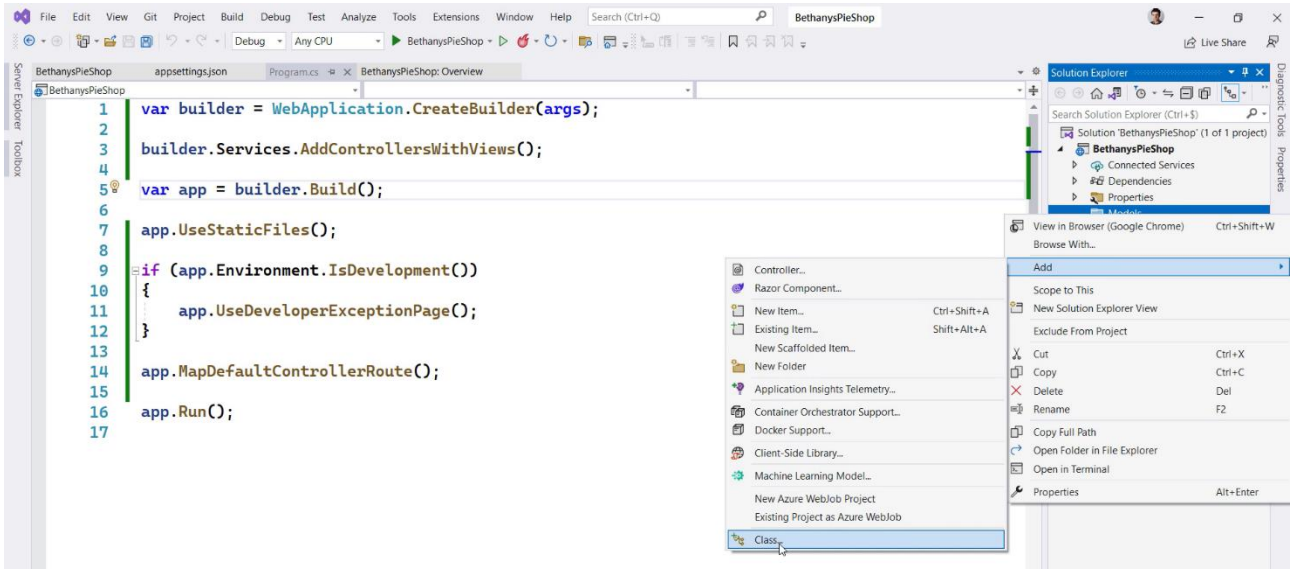


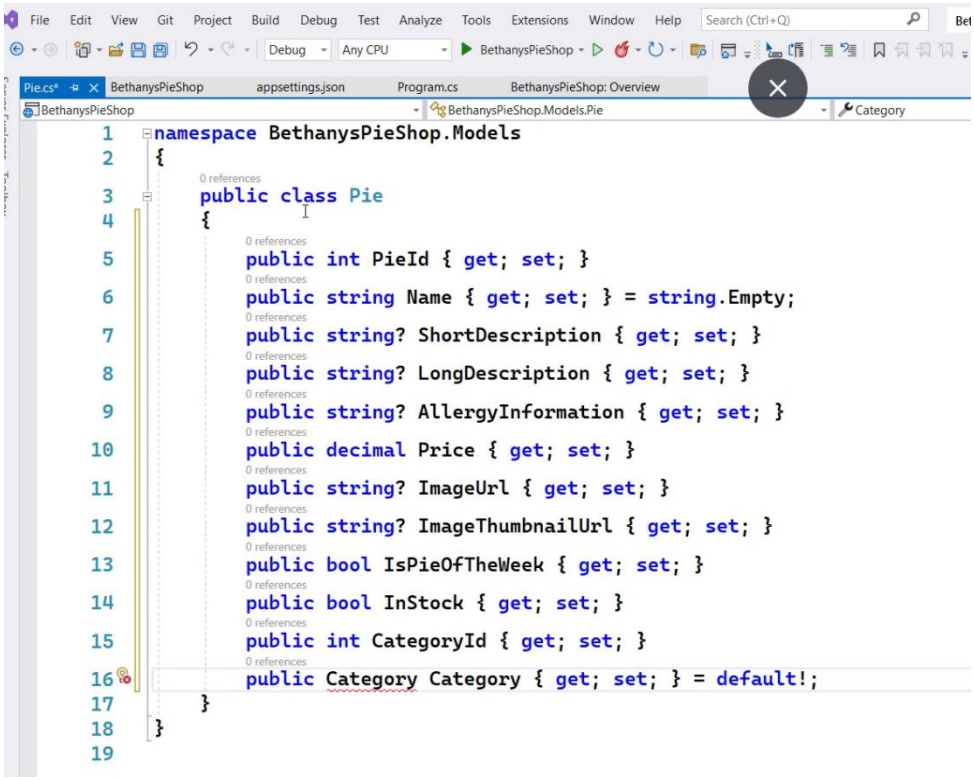

B utilizzerà la stezza istanza PieRepository, quando la richiesta viene gestita e la viene inviata al client, l'oggetto verrà scartato:



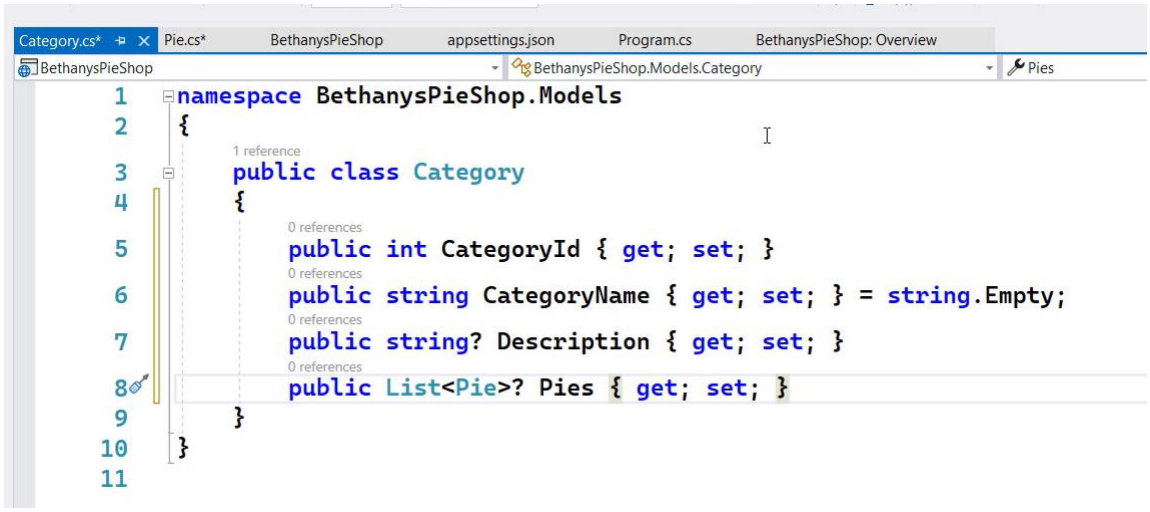

**Demo: Creating the Model and the Repository**

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



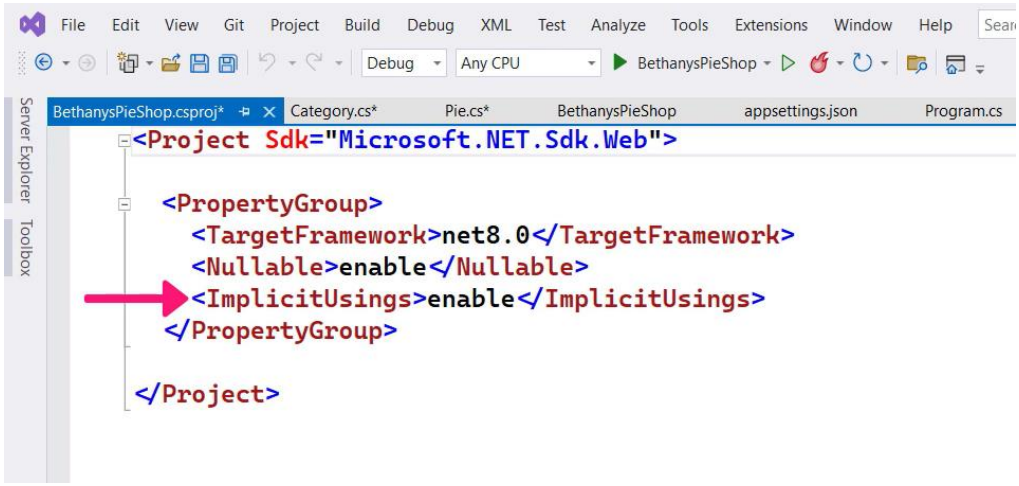



**Osservazioni:**
- E' stato creato un PieID;
- Alcune proprietà sono nullable (sulle proprietà che non richiedono un valore, in quanto nel DB potrebbero essere NULL)
- Name è settato a vuoto,
- Category ha default! "Questo operatore perdona i valori null" , per dichiarare che quella proprietà non dovrebbe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

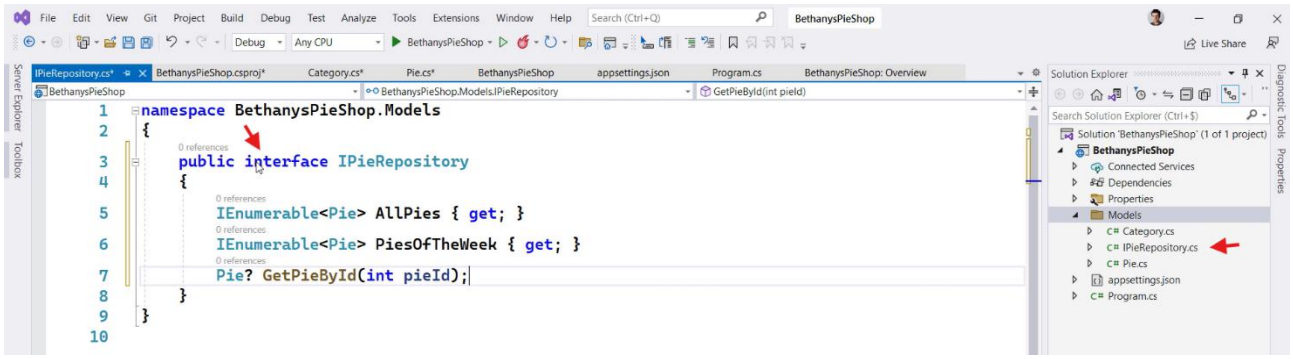



**Osservazioni:**
*public List<Pie>? Pies { get; set; }* potrebbe avere una categoria ma non un elenco di torte

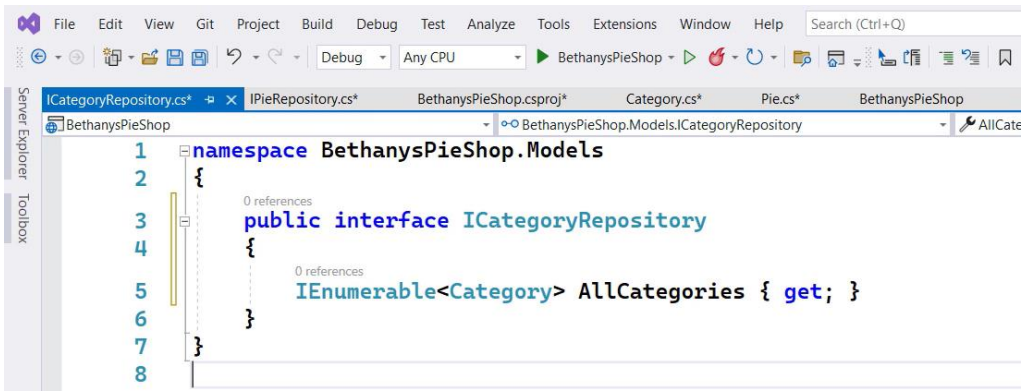
Possiamo notare che non sono presenti using all'interno delle varie classi, il motivo è questo:



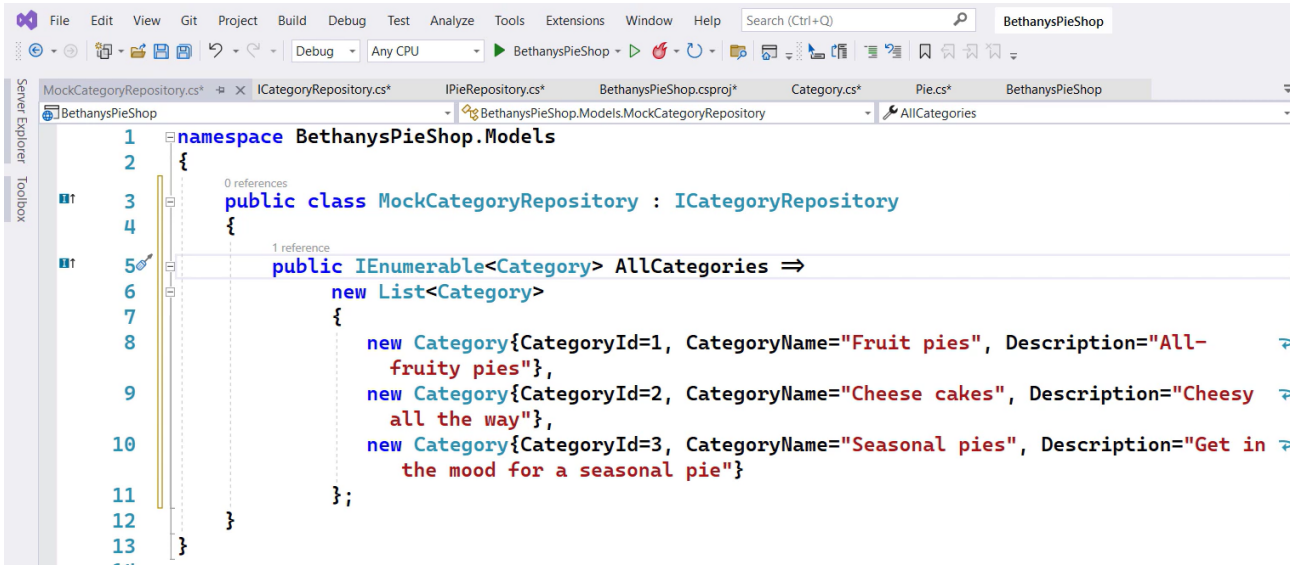

Ora creiamo IPieRepository.cs che verrà utilizzato per racchiudere tutta la logica di interazione dei dati e assicurarsi che le classi che utilizzano nella mia applicazione stiano semplicemente comunicando con questo repository:




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

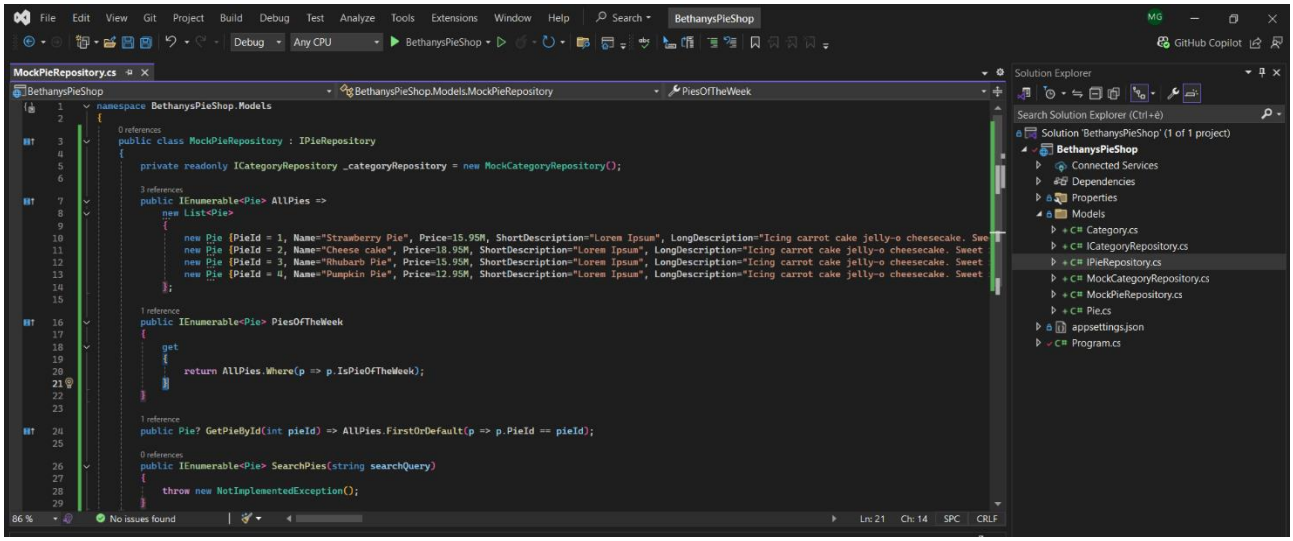


Ora utilizzeremo una mock class per poter simulare il DB, ovvero i dati che recuperiamo dal database. Quindi creiamo MockPieCategory.cs che implementerà ICategoryRepository:



```
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"}
            };
    }
}
```

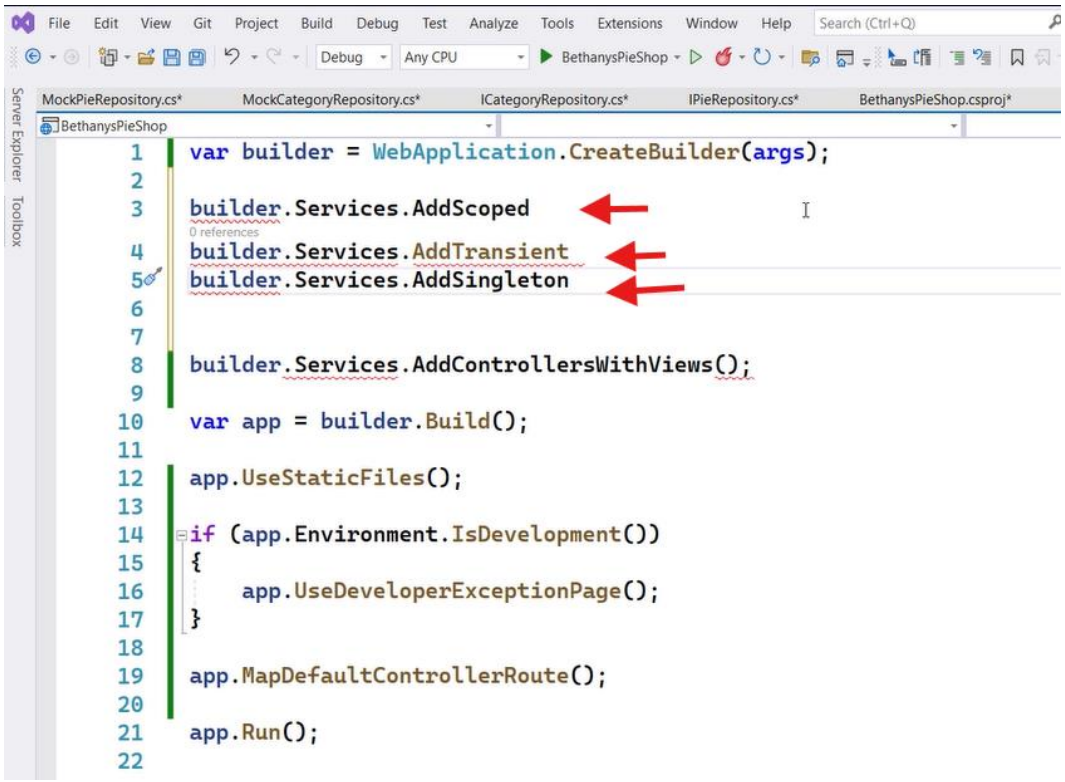
Procediamo in egual modo anche per il mock di Pie, creando quindi MockPieRepository.cs



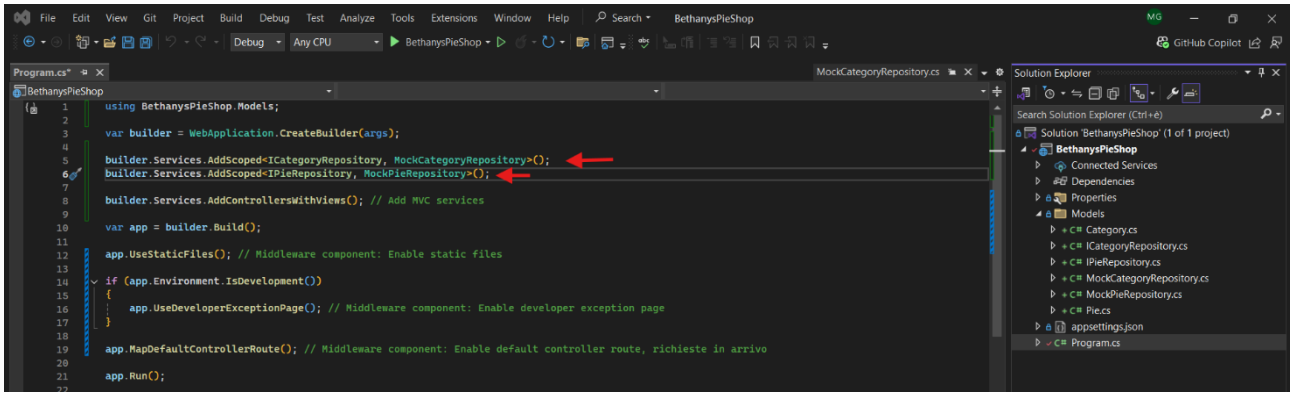
```
namespace BethanysPieShop.Models
{
    public class MockPieRepository : IPieRepository
    {
        private readonly ICategoryRepository _categoryRepository = new MockCategoryRepository();

        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. Swe
                new Pie {PieId = 2, Name="Cheese cake", Price=18.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet
                new Pie {PieId = 3, Name="Rhubarb Pie", Price=15.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet
                new Pie {PieId = 4, Name="Pumpkin Pie", Price=12.95M, ShortDescription="Lorem Ipsum", LongDescription="Icing carrot cake jelly-o cheesecake. Sweet
            };

        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:



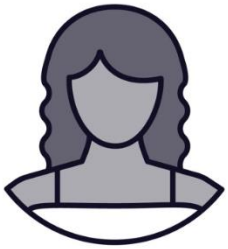

In questo caso utilizzeremo **AddScoped:**



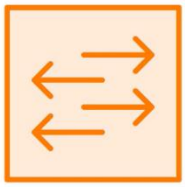

**Creating the Controller**

## The Controller

Central role



Respond to user interaction

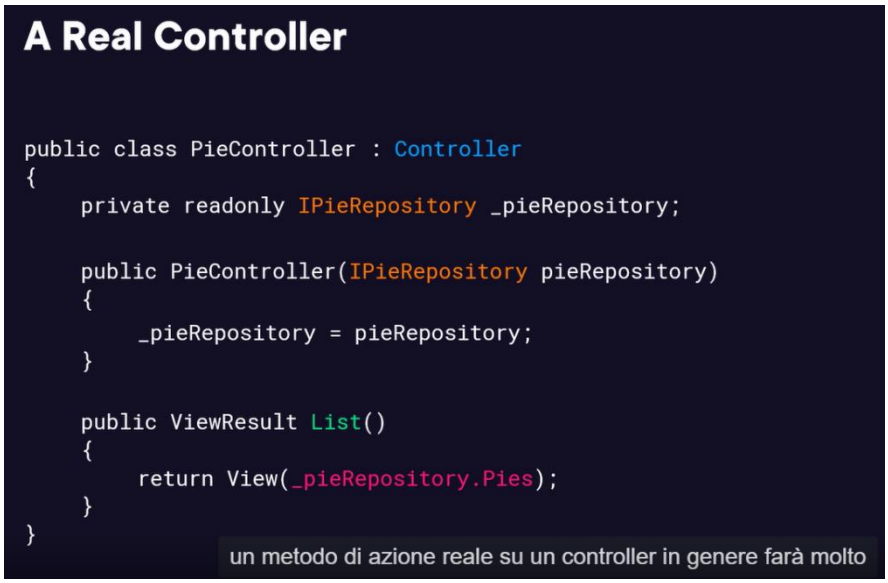


Interact with model and select view

No knowledge of data persistence

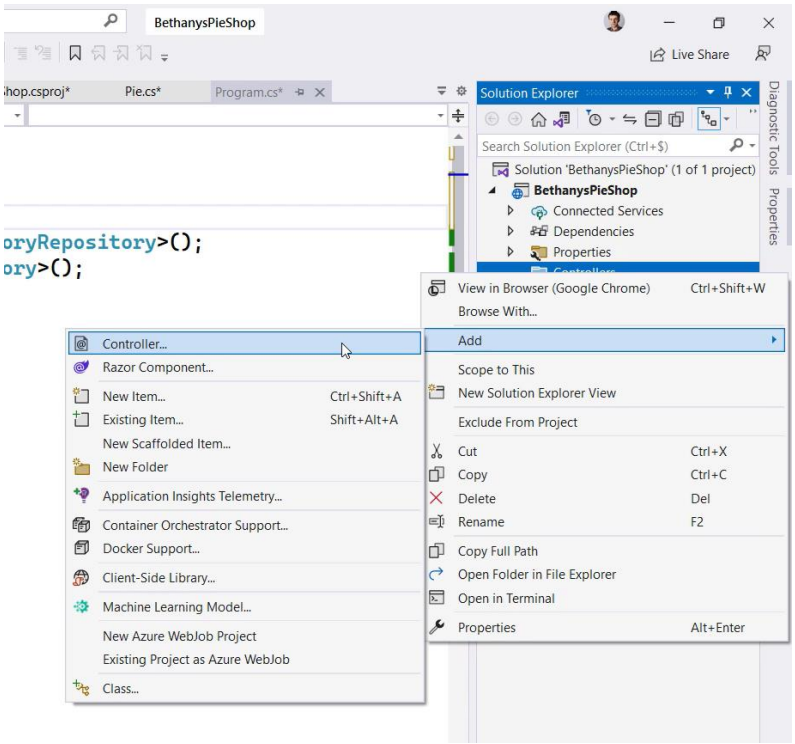
## A Basic Controller

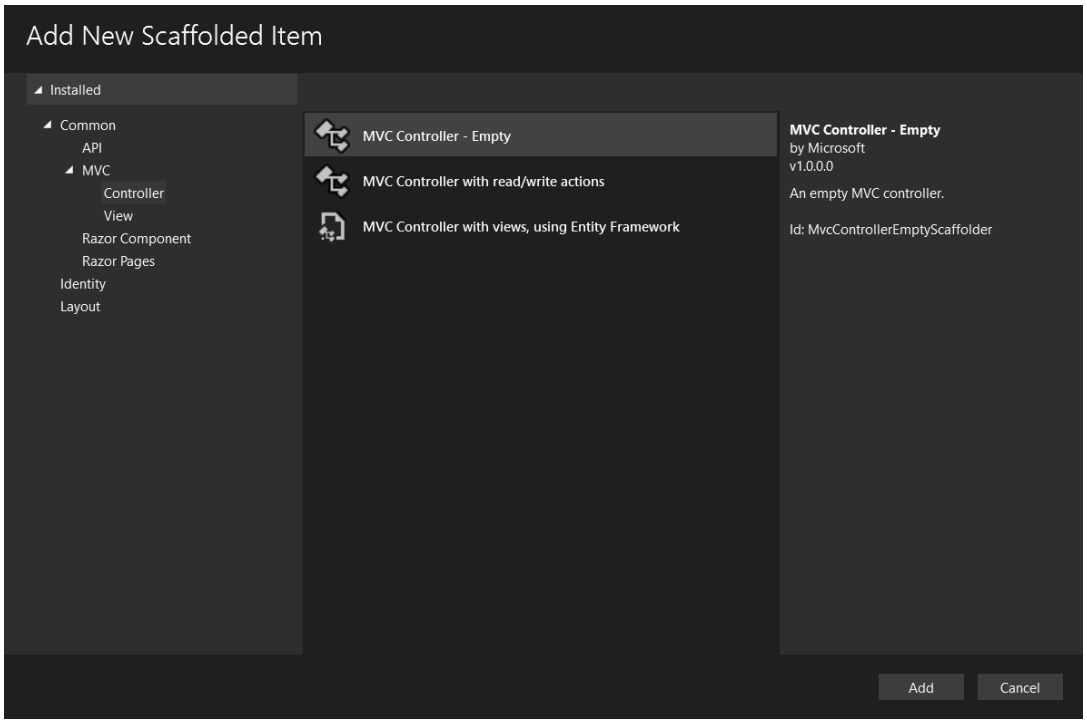
```
public class PieController : Controller
{
    public ViewResult Index()  <-- Action
    {
        return View();
    }
}
```



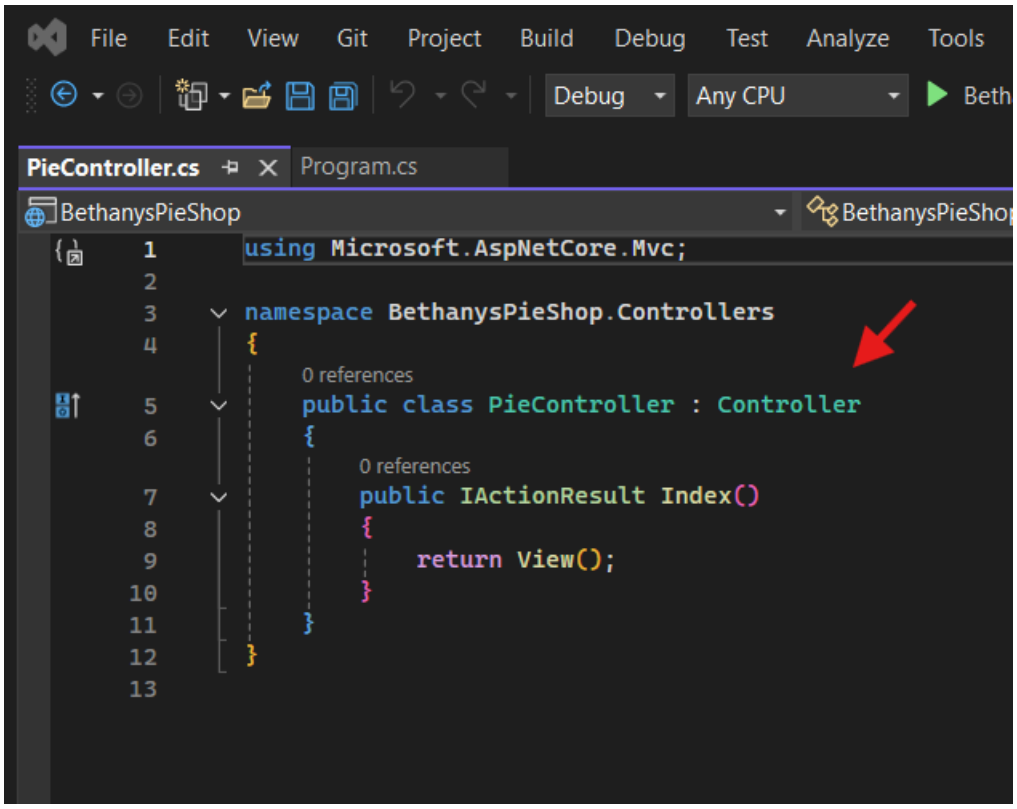

**Demo: Adding the Controller**

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





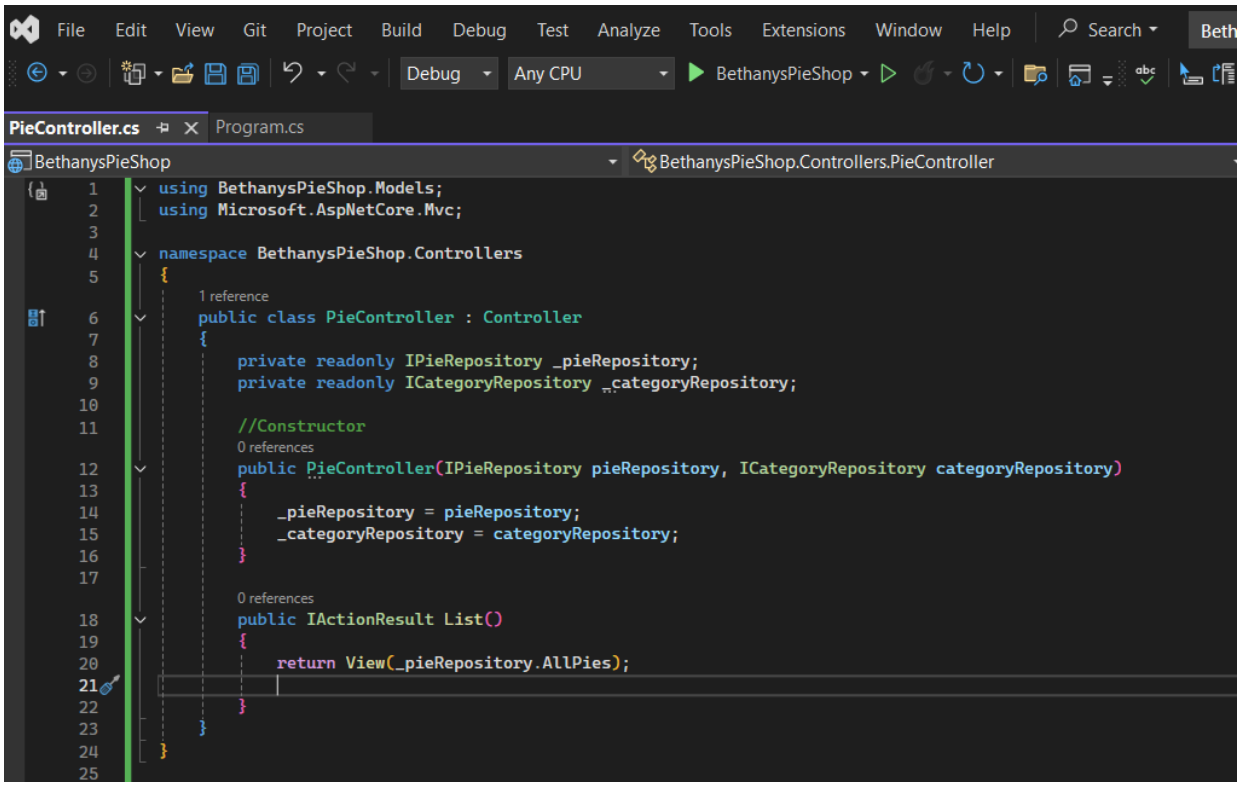

E lo chiamiamo PieController




Osservazioni: Notiamo che eredita da Controller.

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.

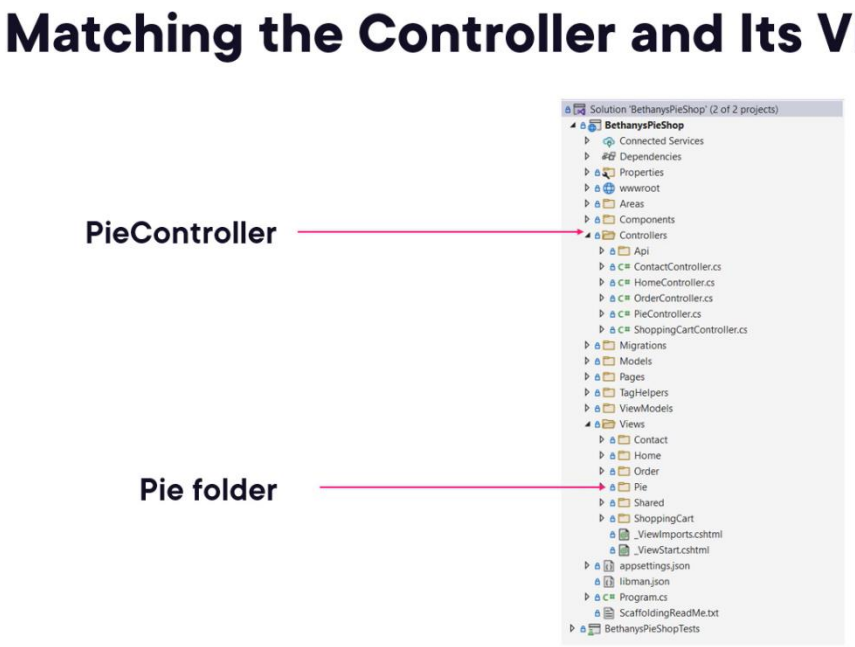
Possiamo quindi creare un'istanza di sola lettura:



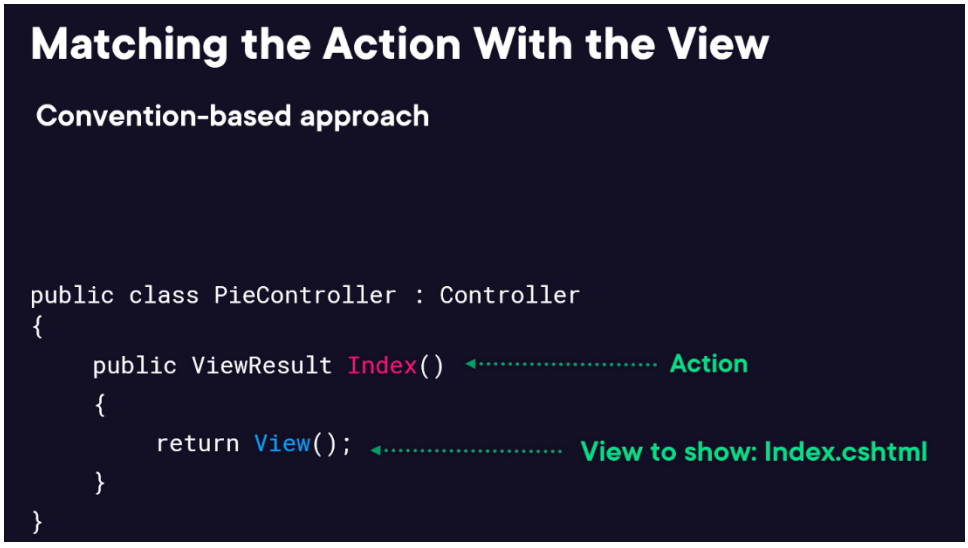

Ora ci manca solo la vista.

**Adding the View**

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




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:



**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

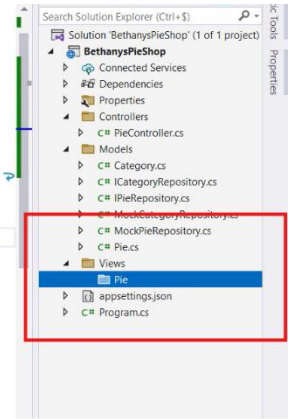
```
@model IEnumerable<Pie>
<html>
  <body>
    <div>
      @foreach (var pie in Model)
      {
        <div>
          <h2>@pie.Name</h2>
          <h3>@pie.Price.ToString("c")</h3>
          <h4>@pie.Category.CategoryName</h4>
        </div>
      }
    </div>
  </body>
</html>
```

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

**Demo: Creating first View**

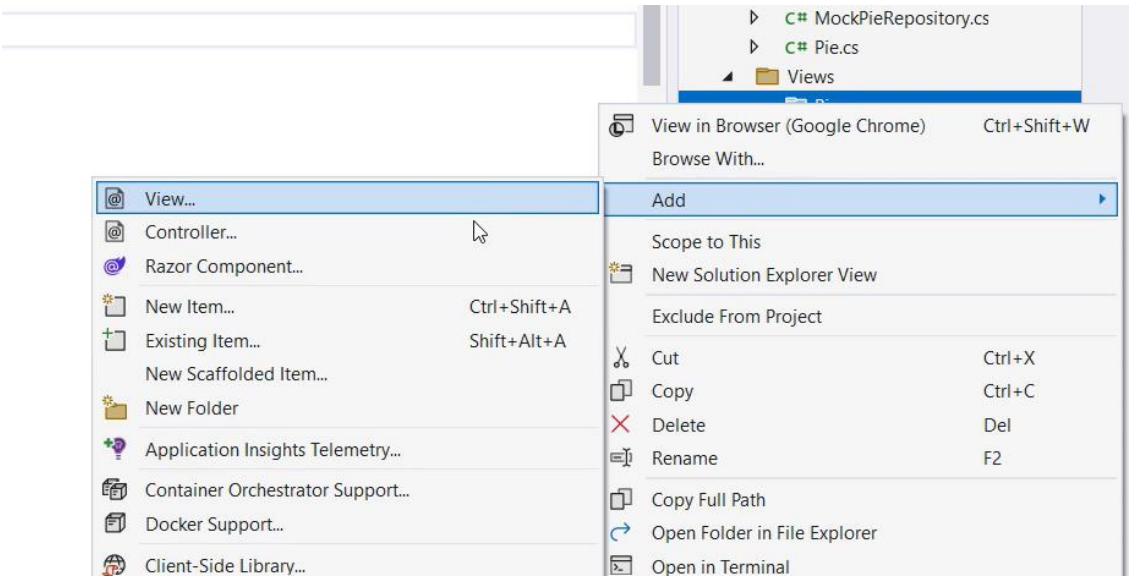
Creiamo la cartella Views nel nostro progetto, al suo interno creiamo la cartella Pie per riflettere ciò che avviene nel controller, come è stato in precedenza:

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

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

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



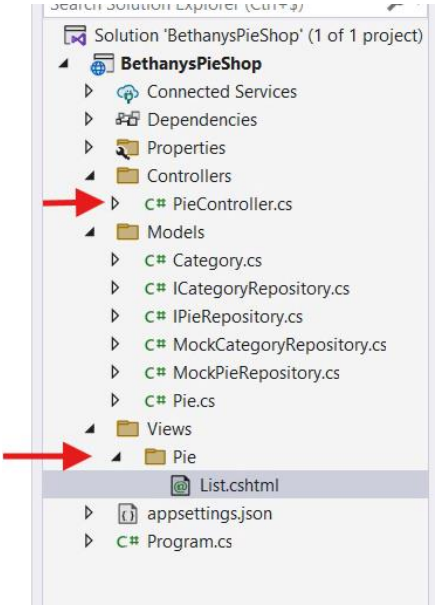

Cosi il framework ora cercherà una vista a chiamata Elenco all'interno della cartella Pie.

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:

List.cshtml*

BethanysPieShop

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <meta name="viewport" content="with=device-width" />
5       <title>Bethany's Pie Shop'</title>
6   </head>
7   <body>
8
9   </body>
10  </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:

List.cshtml* | PieController.cs* | MockPieRepository.cs | MockCategoryRepository.cs | ICategoryRepository.cs

BethanysPieShop | BethanysPieShop.Controllers.PieController

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

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:

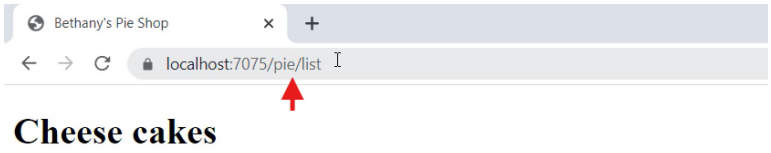
PieController.cs* List.cshtml*

BethanysPieShop

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

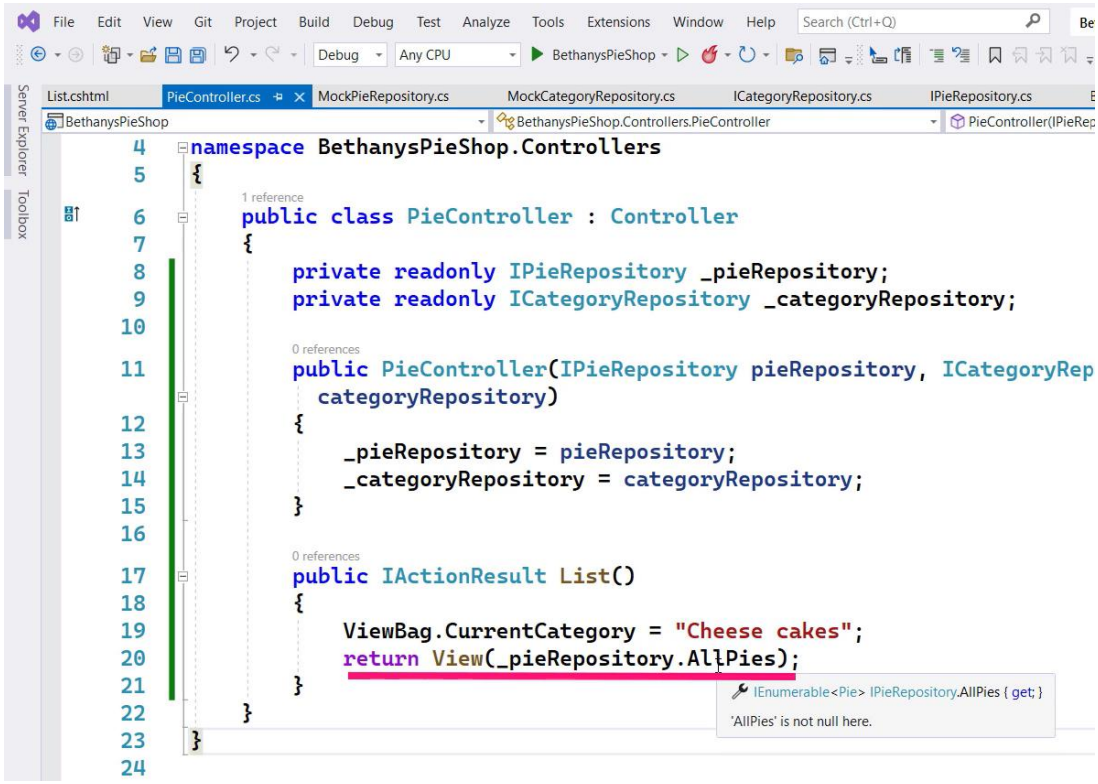
Osservazione: Il metodo *CurrentCategory* funziona in run time.

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

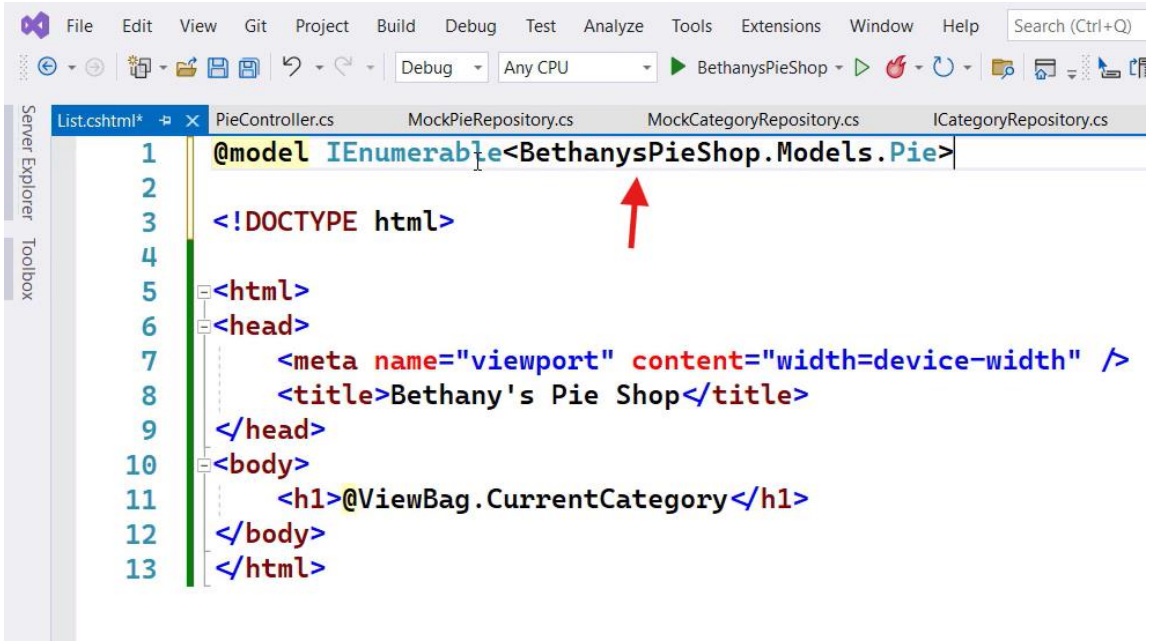



**Cheese cakes**

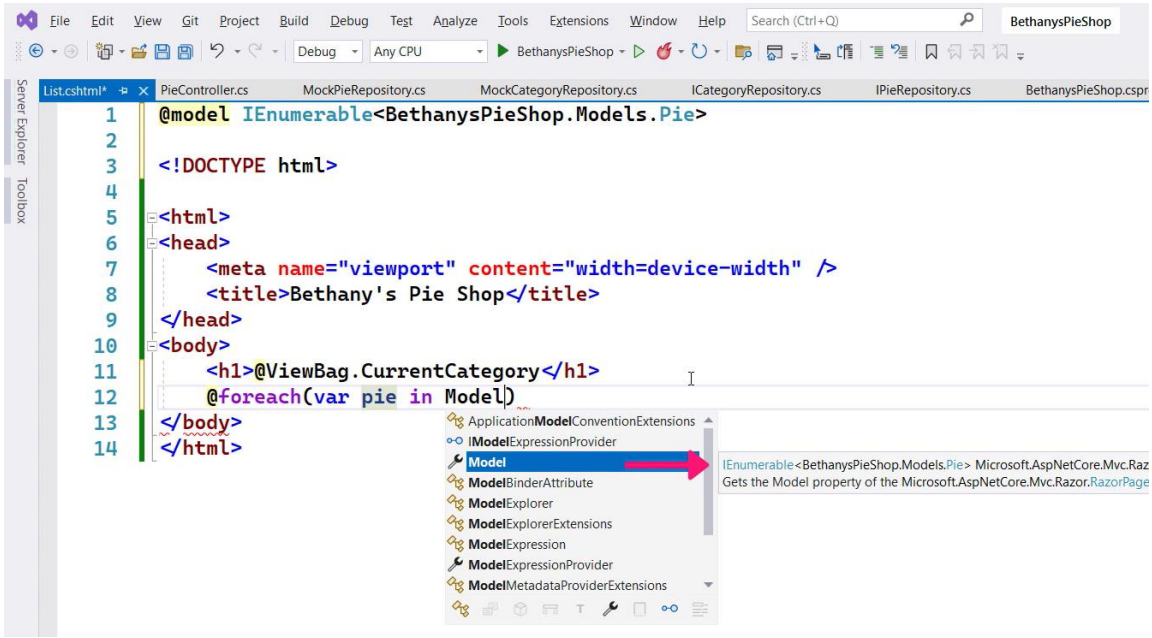
Tornando al progetto, possiamo passare alla nostra View il modello intero, la vista scaverà in queste proprietà. Consideriamo:

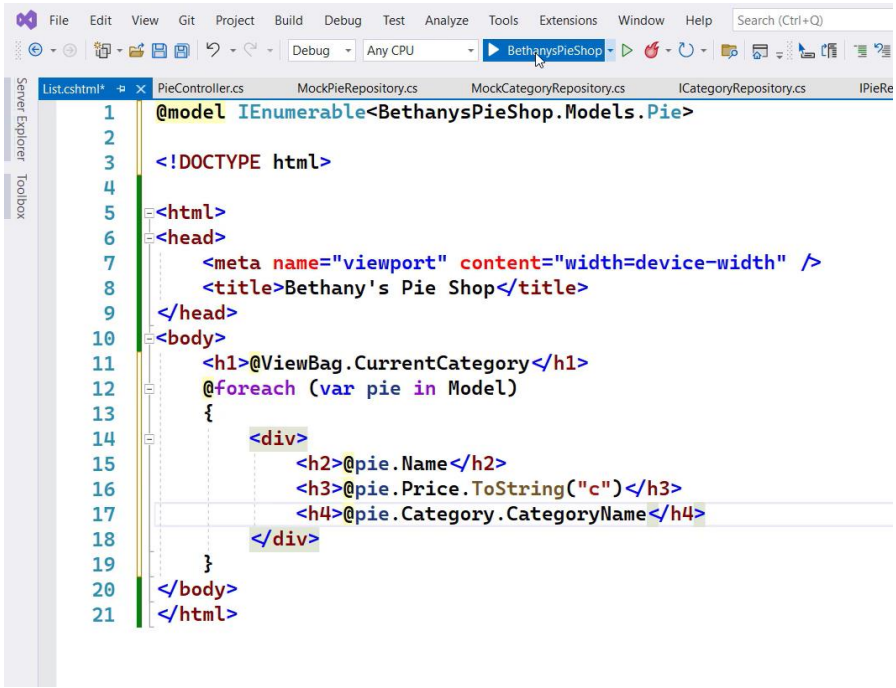



Torno su List.cshtml e aggiungo direttamente il modello IEnumerable:




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:

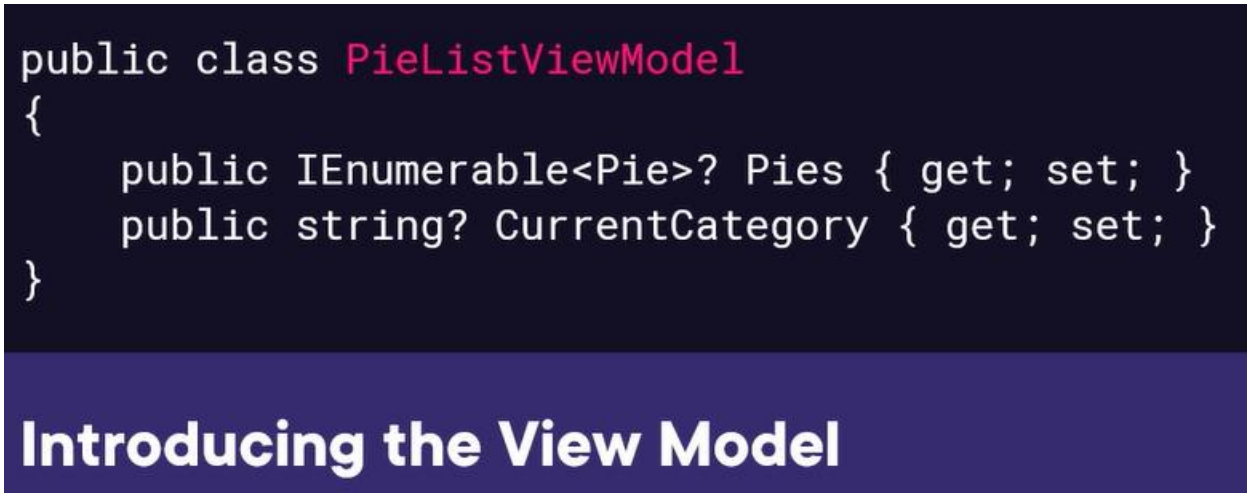



Osservazione: All'interno delle parentesi graffe non è necessario ripetere la @, a meno che non è necessario passare codice negli elementi html.

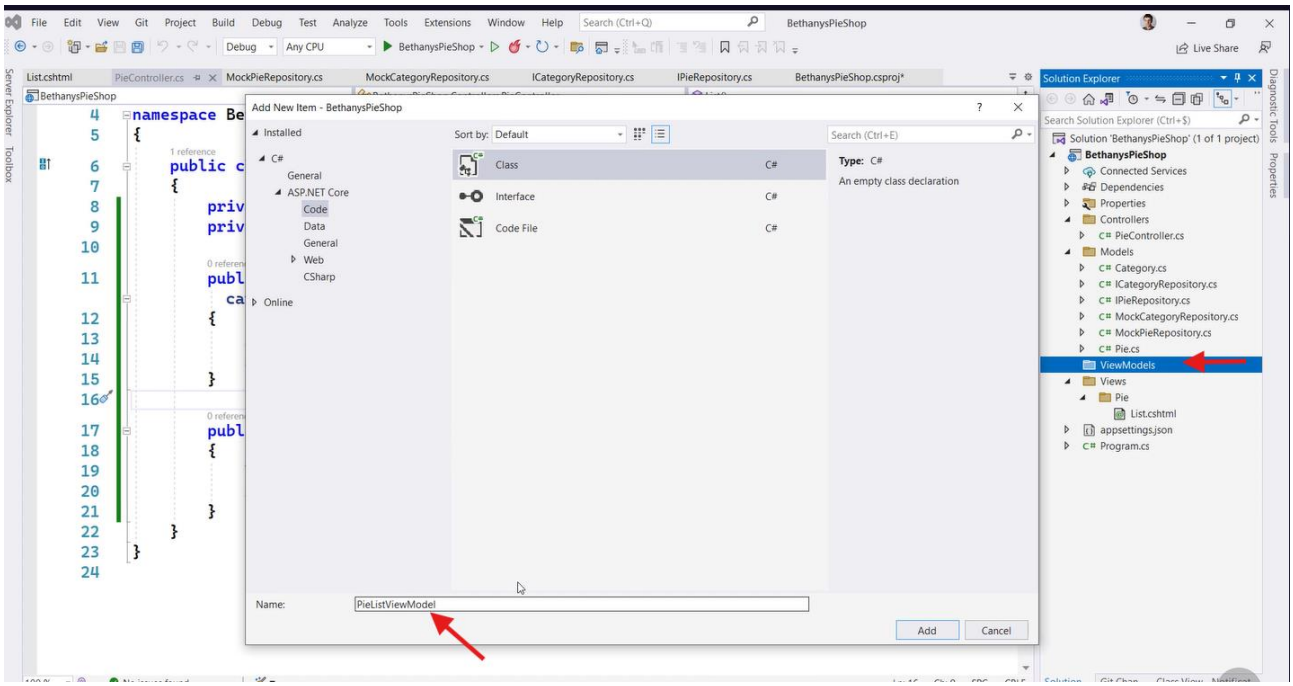
Ora dovremmo essere in grado di visualizzare (sempre inserendo l' url manualmente) la lista delle torte presenti in quella categoria, dove per ognuna è elencato nome, prezzo e nome della categoria:


**Demo: Using a View Model**

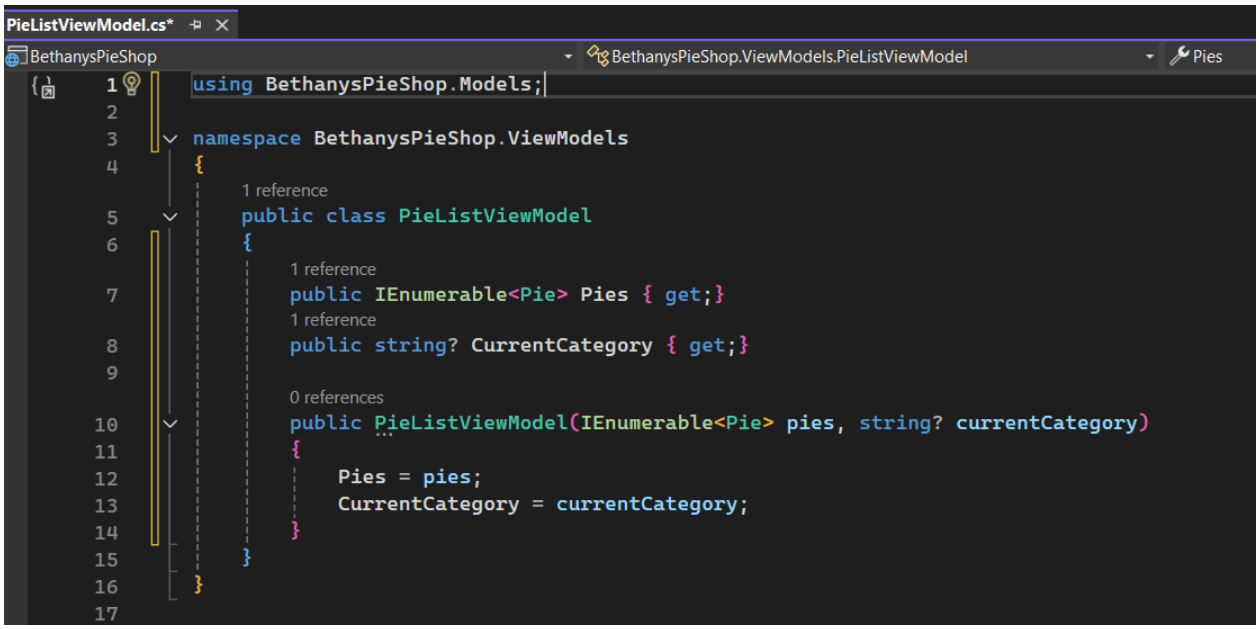



In molti casi vedremo che il modello che vogliamo passare alla vista non è MAPPATO a un modello di dominio. E' buona abitudine utilizzare un modello di visualizzazione, ovvero una classe che racchiude più proprietà in modo che la visualizzazione possa accedere a tutti i dati di cui ha bisogno.

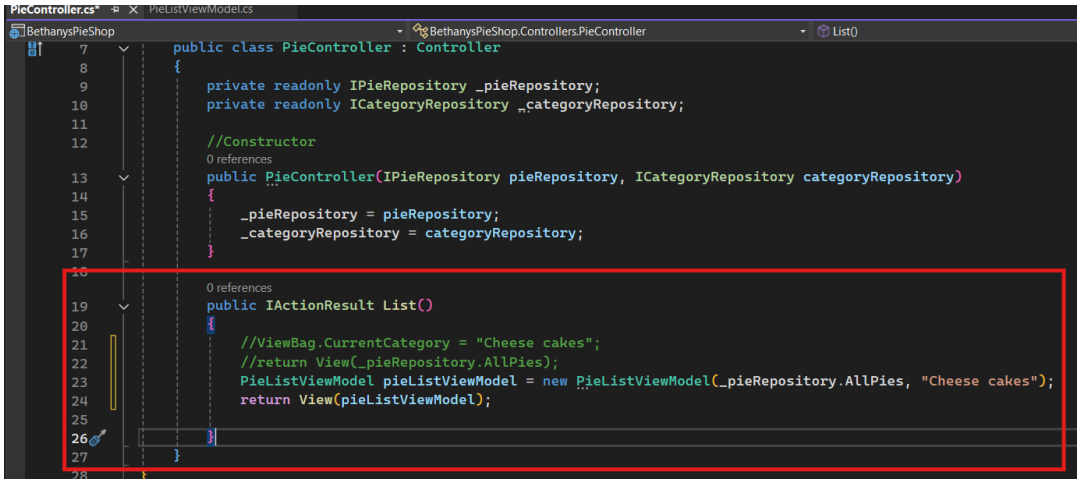
Creiamo quindi una nuova cartella nel nostro progetto **"ViewModels", conterrà delle utility class non vere e proprie entità del DB.**




Scriviamo il codice:




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



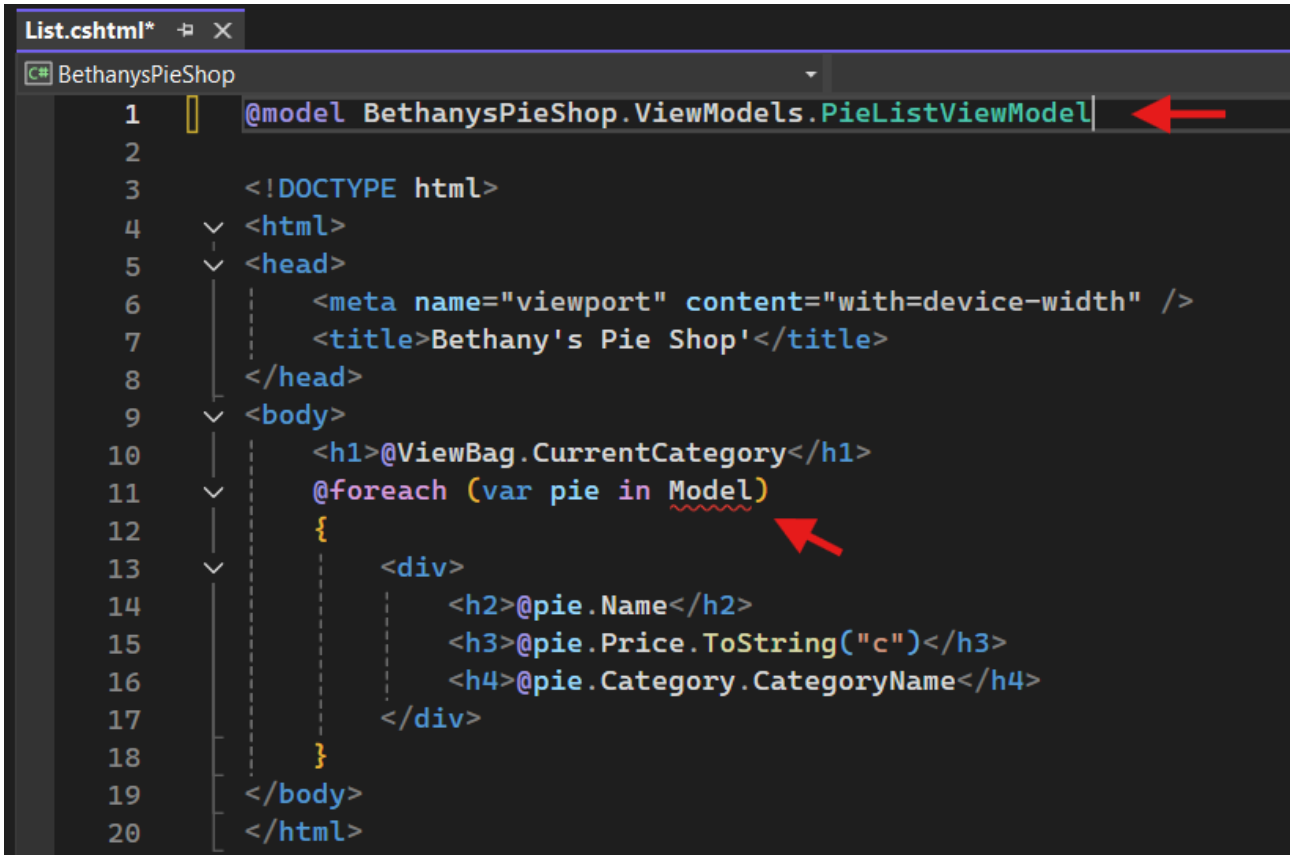
```
public class PieController : Controller
{
    private readonly IPieRepository _pieRepository;
    private readonly ICategoryRepository _categoryRepository;

    //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);

    }
}
```

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:

```
@model IEnumerable<BethanysPieShop.Models.Pie>
```

Dopo:



```
@model BethanysPieShop.ViewModels.PieListViewModel

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="with=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>
```

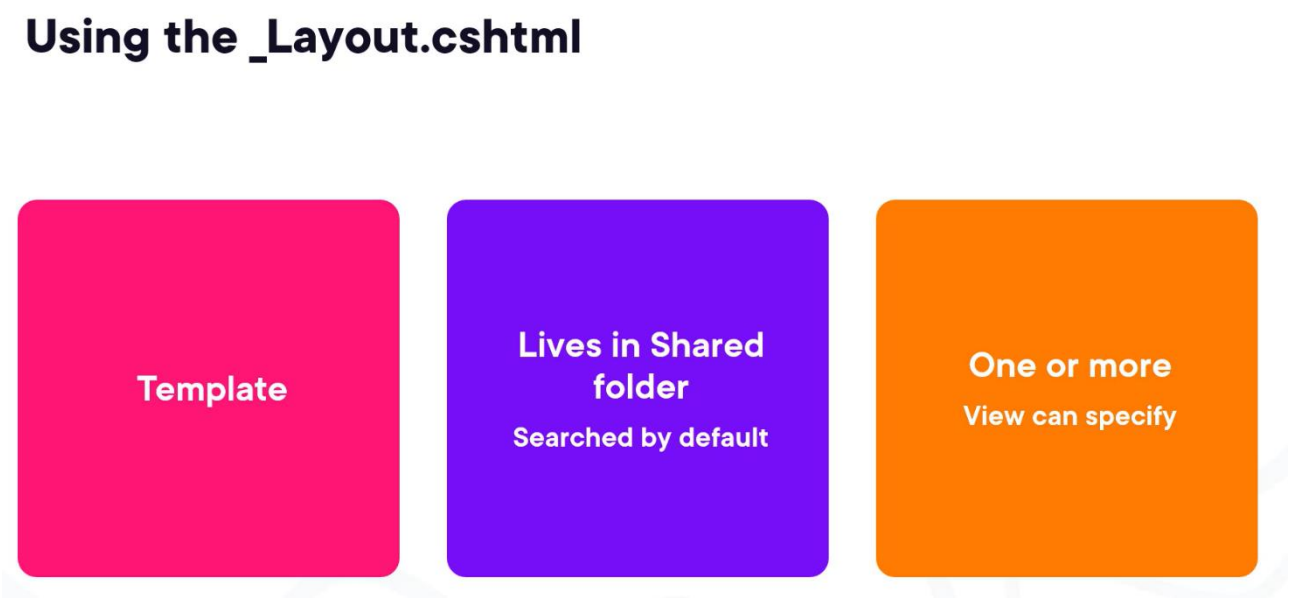
L'errore a riga 11, indica che model non è più un Ienumerables in Pies, quindi possiamo non enumerarlo, però ora abbiamo Pies, quindi scriviamo Model.Pies:

```
@model BethanysPieShop.ViewModels.PieListViewModel

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="with=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**



Esempio di modello di layout:

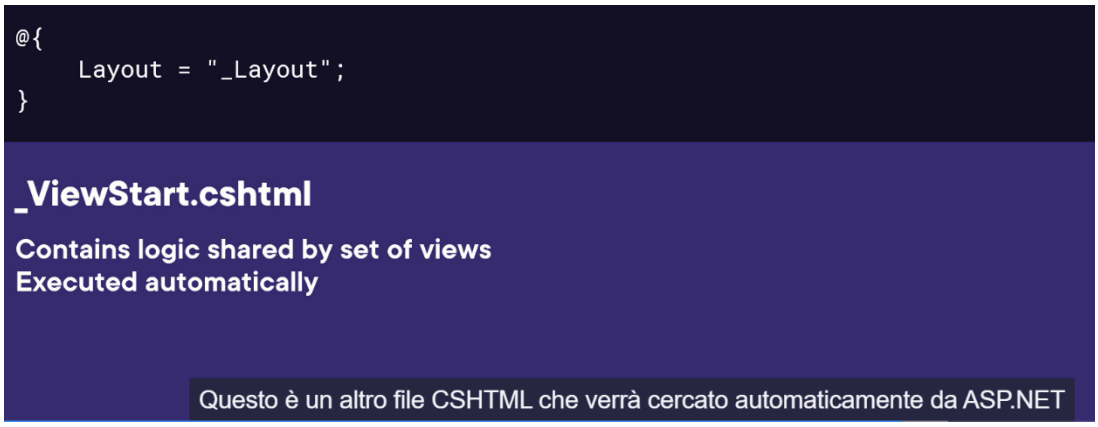
_Layout.cshtml

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

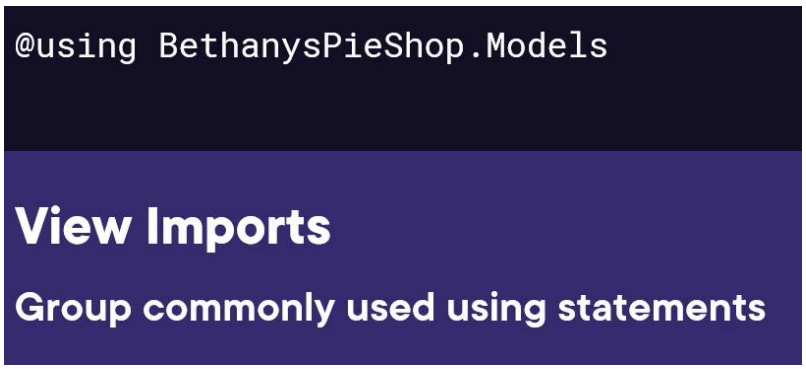
```
@{
    Layout = "_Layout";
}

@model PieListViewModel
```

View Can Specify the Layout

Osservazione: possiamo farlo individualmente in ogni visione

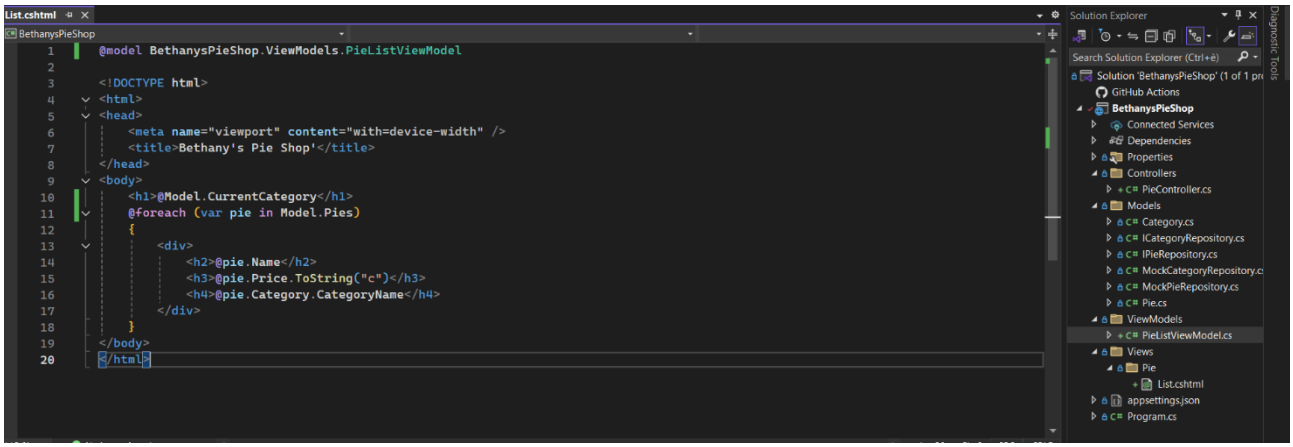



Osservazione: in _ViewStart.cshtml in genere inseriamo il codice che vogliamo eseguire quando viene chiamata una qualsiasi vista




**Demo: Adding Extra View Files**

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

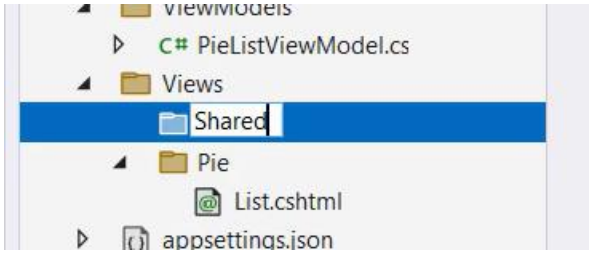



Attualmente il nostro progetto contiene una sola pagina.

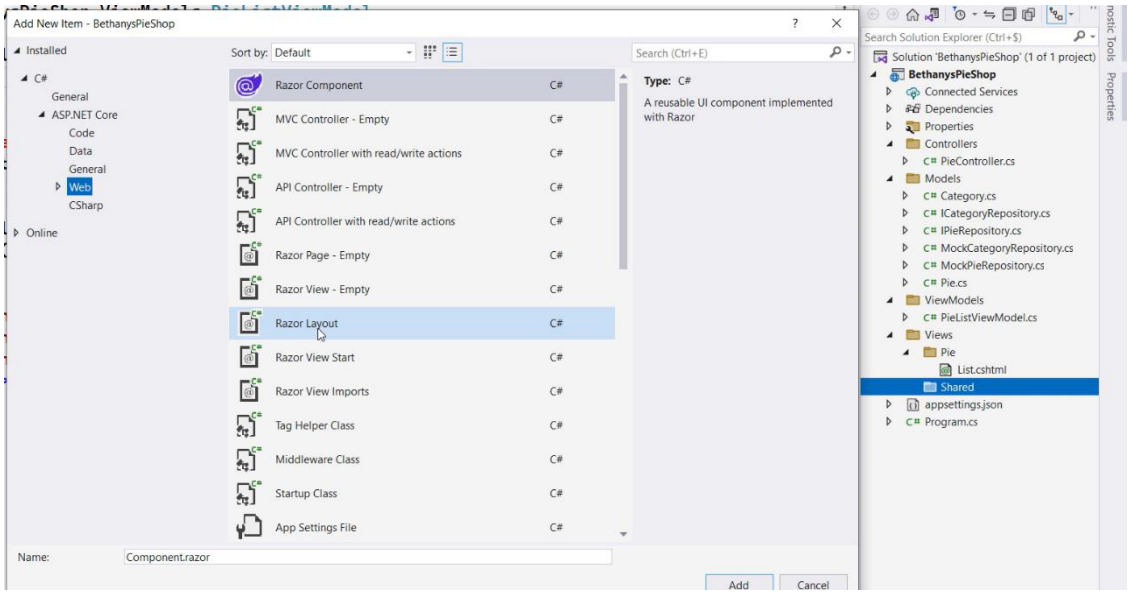
Views/Shared è la cartella condivisa tra tutte le visualizzazioni di tutti i controllori, conosciuta automaticamente

da ASP.NET Core.

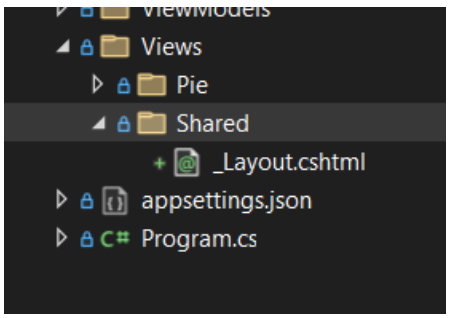
Quindi la creiamo:




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




Lasciamo il default name per ora



_Layout.cshtml* List.cshtml

BethanysPieShop

```
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>
14
```

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

PieListViewModel.cs List.cshtml PieController.cs MockPieRepository.cs MockCategoryRep

```
@model BethanysPieShop.ViewModels.PieListViewModel

<!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:

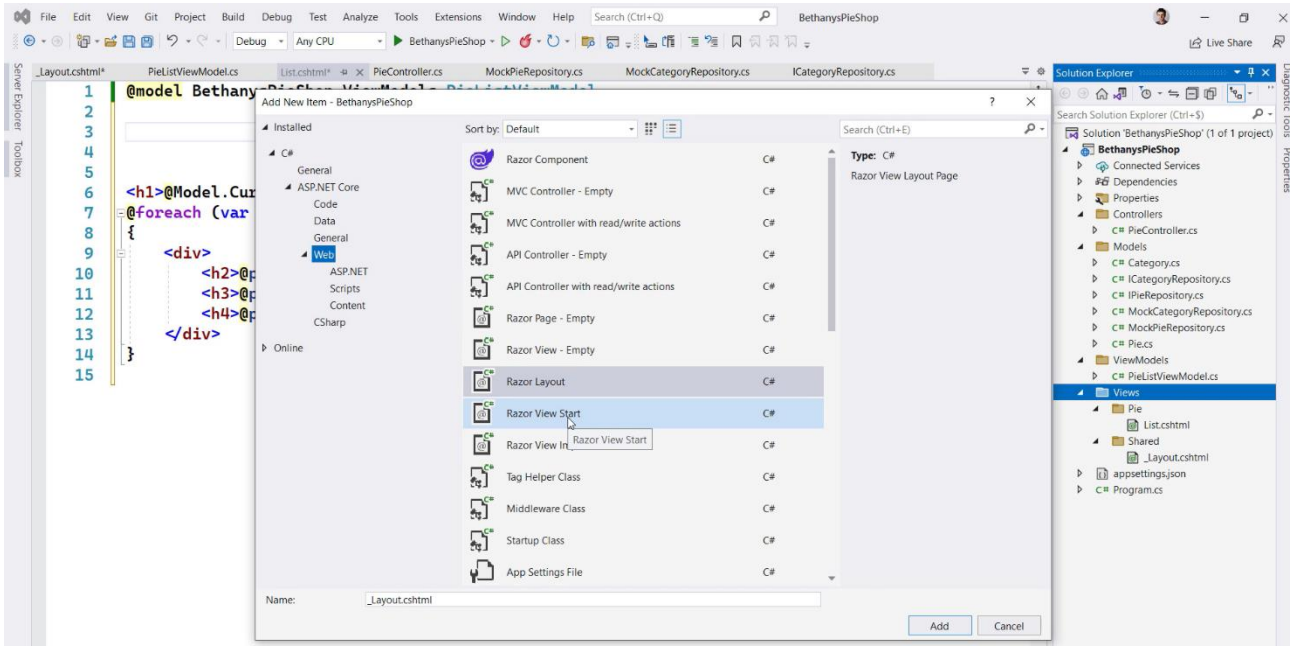
```
_Layout.cshtml*   PieListViewModel.cs   List.cshtml*   PieController.cs   MockPieRepo
1  @model BethanysPieShop.ViewModels.PieListView
2
3  @{
4      Layout = "_Layout";
5  }
6
7
8  <h1>@Model.CurrentCategory</h1>
9  @foreach (var pie in Model.Pies)
10 {
11     <div>
12         <h2>@pie.Name</h2>
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

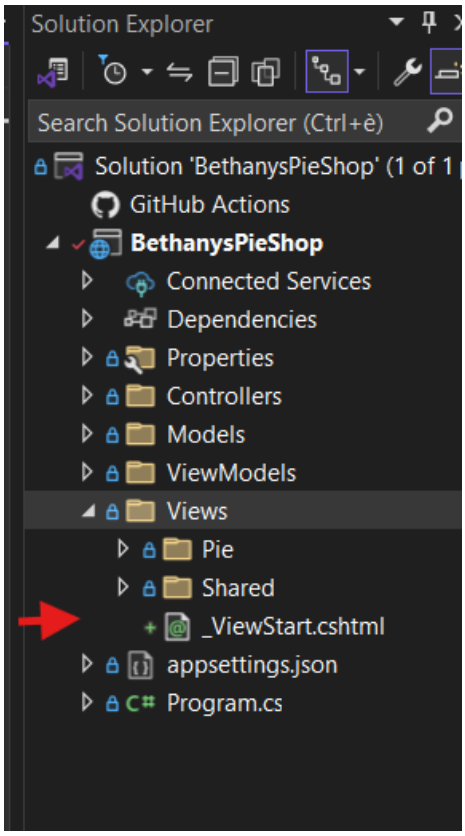
come file ViewStart, il quale verrà eseguito ogni volta che viene eseguito il rendering di una vista  e li inseriamo il codice che deve

essere eseguito ogni volta

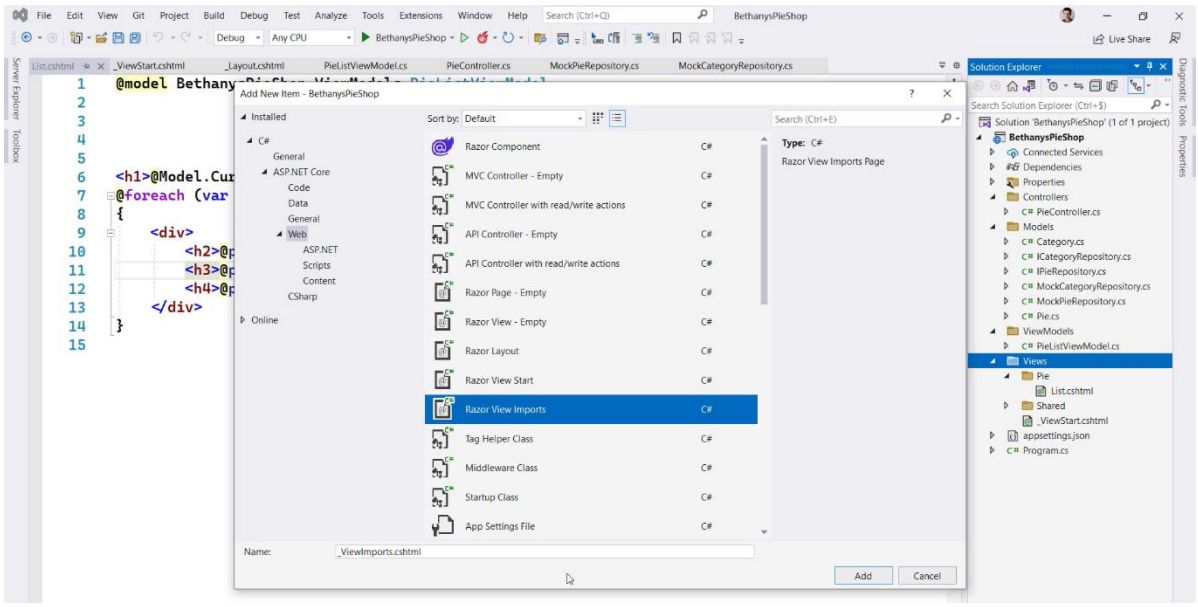
Quindi cancelliamo la porzione di codice presente nell'immagine precedente e aggiungiamo Razor View Start:




Deve essere presente nella root di View:

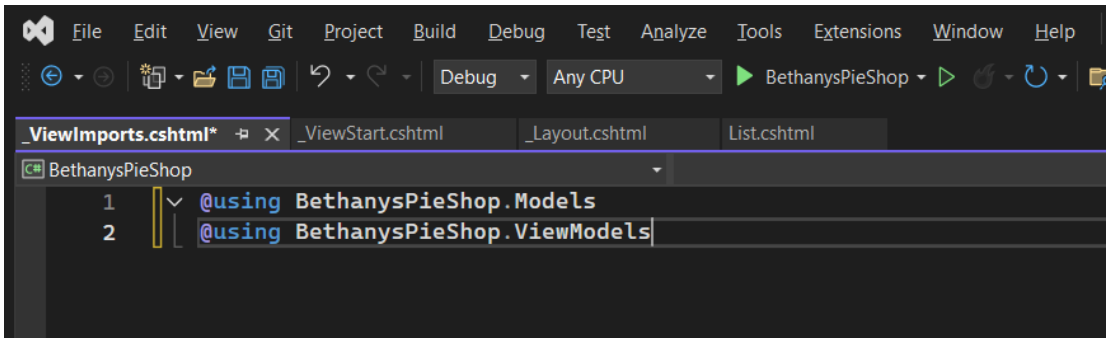



Sempre in Views aggiungiamo Razor _ViewImports.cshtml:

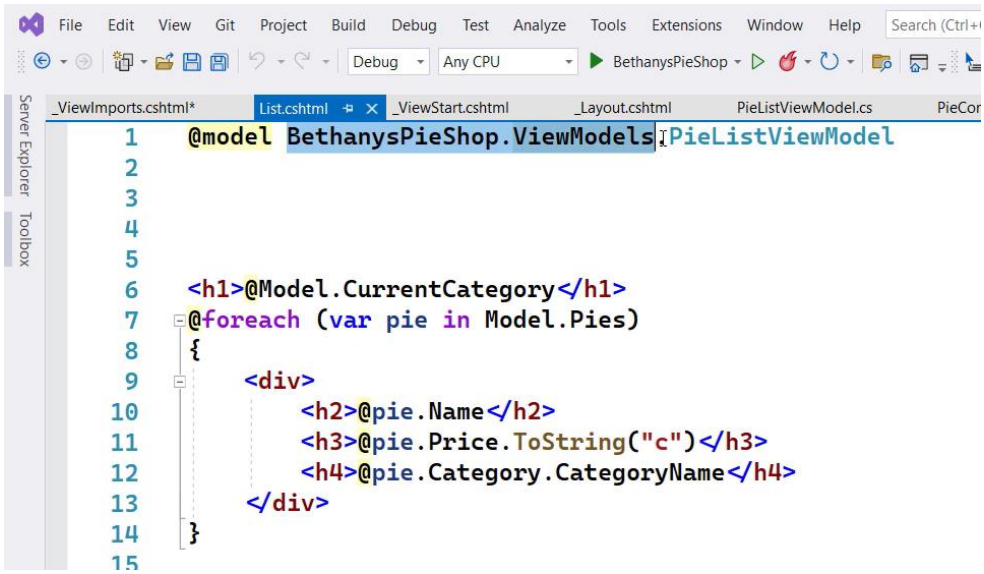



_ViewImports.cshtml sarà utilizzato per inserire altre istruzioni using globali

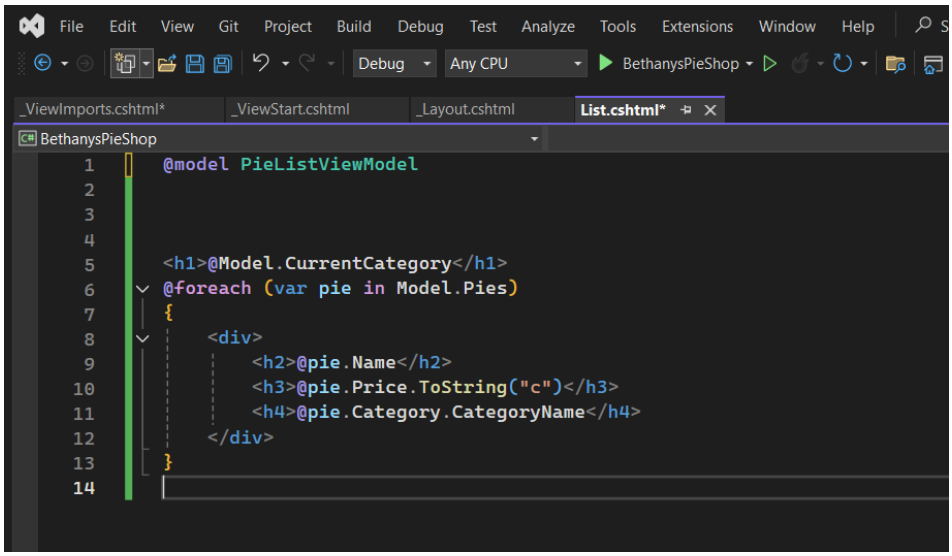
Quindi inseriamo:




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




Ottenendo:

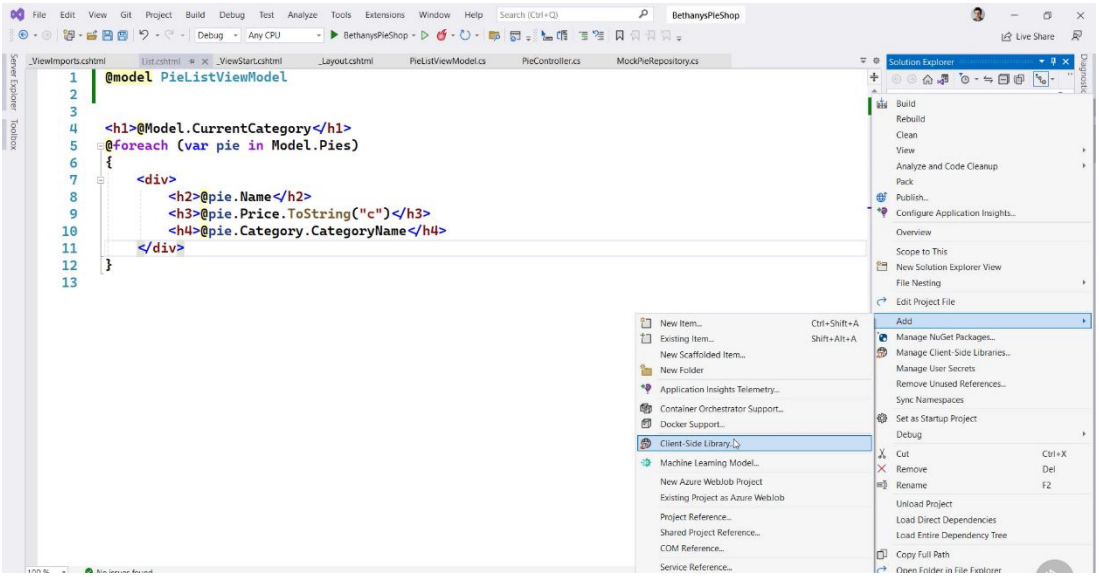


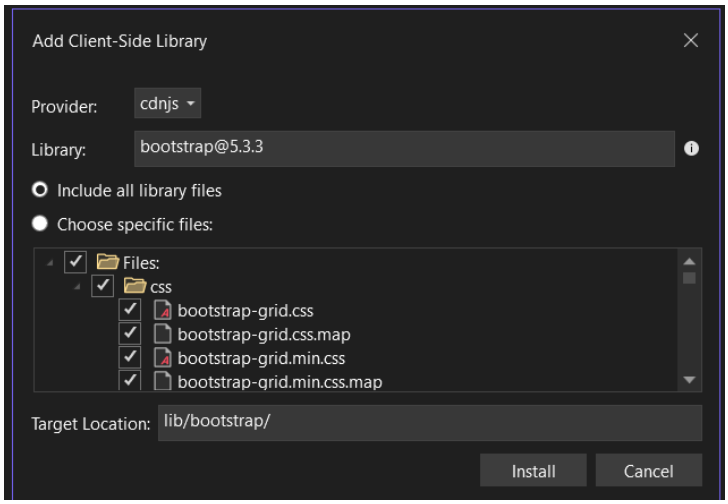
**Styling the view**

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.

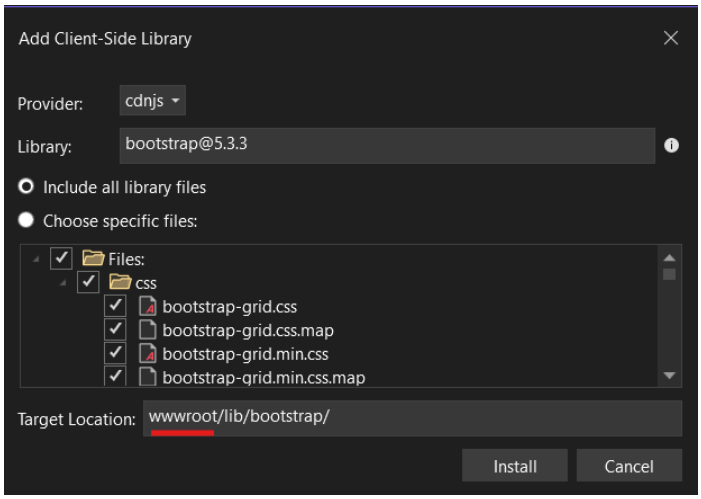

**Demo: Styling the view**

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

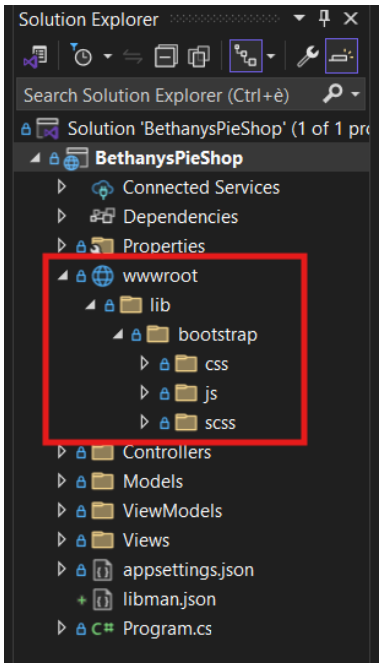




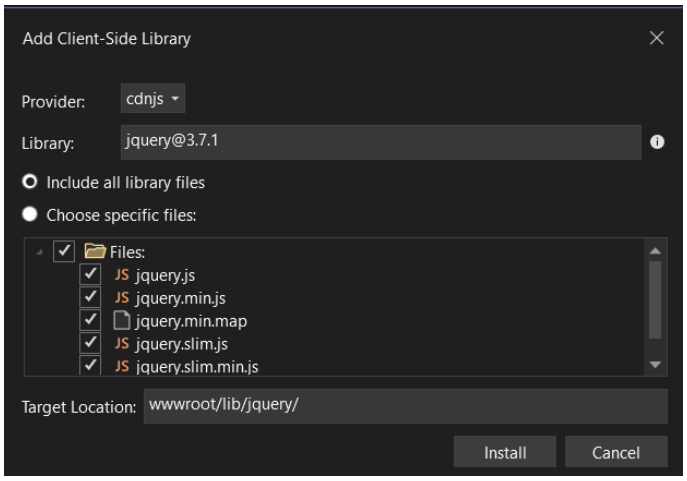

Dopo aver cercato Bootstrap l'editor ci consiglia l'ultima versione presente, ora ci tocca modificare il target location, in:



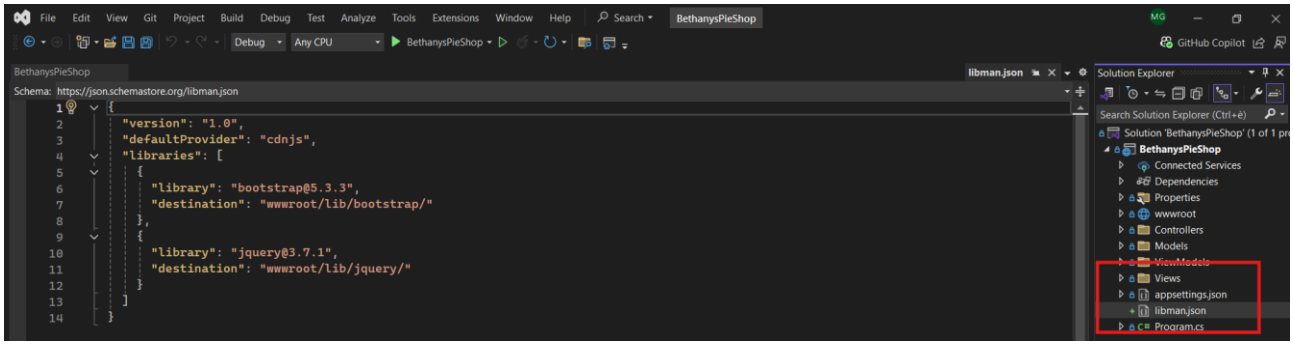

Ora abbiamo la nostra libreria:




Procediamo in maniera analoga con l'aggiunta di Jquery:



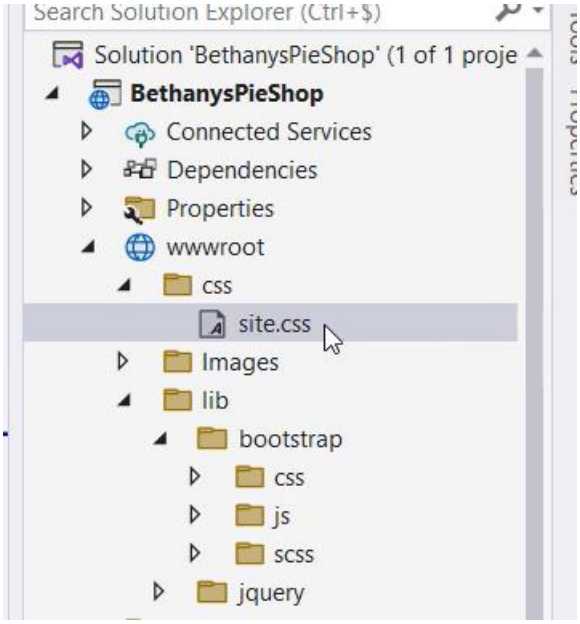

Possiamo notare le varie configurazioni nel file libman.json:



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

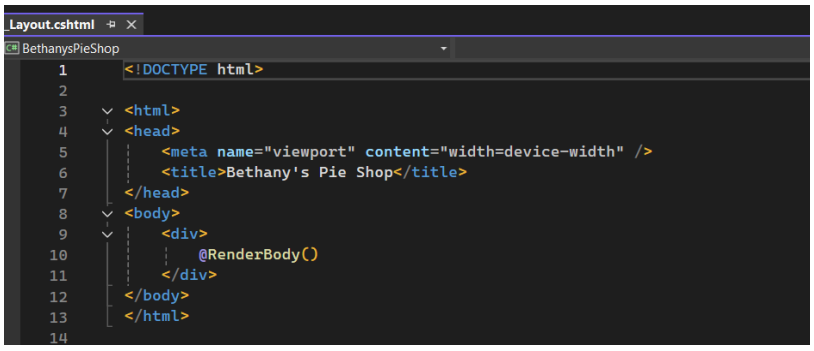
| Nome | Tipo | Dimensione compre... | Protetto da... | Dimensione | Proporzione | Ultima modifica |
|---|---|---|---|---|---|---|
| Images | Cartella di file | | | | | 08/01/2024 04:12 |
| site.css | File di origine CSS | 28 KB | No | 236 KB | 89% | 05/02/2024 04:03 |

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:




Dopo:

_Layout.cshtml

BethanysPieShop

```
1   <!DOCTYPE html>
2
3   <html>
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=Work+Sans' rel='stylesheet' type='text/css'>
8       <script src="~/lib/jquery/jquery.js"></script>
9       <script src="~/lib/bootstrap/js/bootstrap.bundle.js"></script>
10      <link href="~/css/site.css" rel="stylesheet" />
11      <base href="/" />
12  </head>
13  <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 , nel seguente modo:

_Layout.cshtml

BethanysPieShop

```
<!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>
    <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">
                        <img src="images/bethanys-pie-shop-logo_horiz-white.png" width="151" height="47" class="d-inline-block align-top mb-2 mt-1"
                             alt="Bethany's Pie Shop Logo">
                    </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>
```

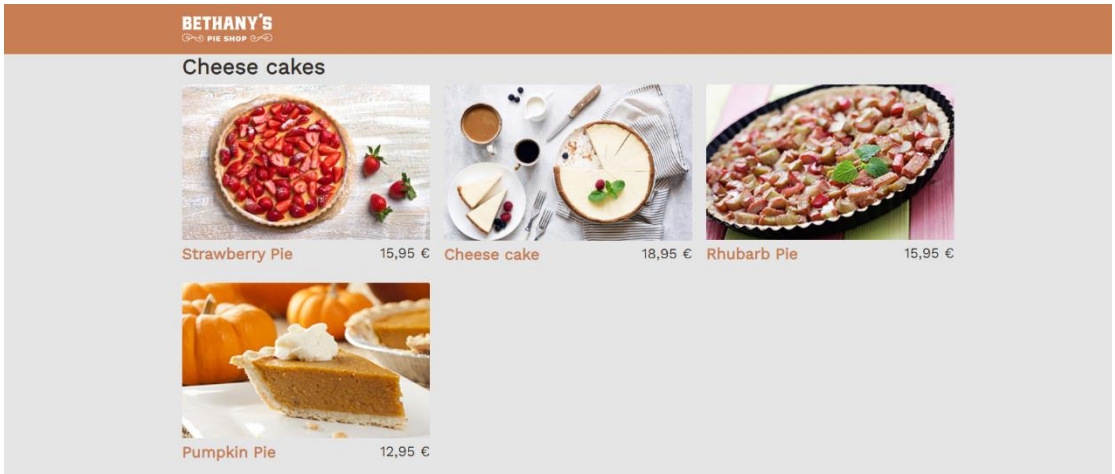
Ci rechiamo nel List.cshtml e lo modifichiamo nel seguente modo:

List.cshtml

BethanysPieShop

```
@model PieListViewModel

<h1>@Model.CurrentCategory</h1>

<div class="row row-cols-1 row-cols-md-3 g-4">

    @foreach (var pie in Model.Pies)
    {
        <div class="col">
            <div class="card pie-card">
                <img src="@pie.ImageThumbnailUrl" class="card-img-top" alt="@pie.Name">
                <div class="card-body pie-button">
                    <h4 class="d-grid">
                    </h4>

                    <div class="d-flex justify-content-between mt-2">
                        <h2 class="text-start">
                            <a class="pie-link">@pie.Name</a>
                        </h2>
                        <h5 class="text-nowrap">
                            @pie.Price.ToString("c")
                        </h5>
                    </div>
                </div>
            </div>
        </div>
    }

</div>
```

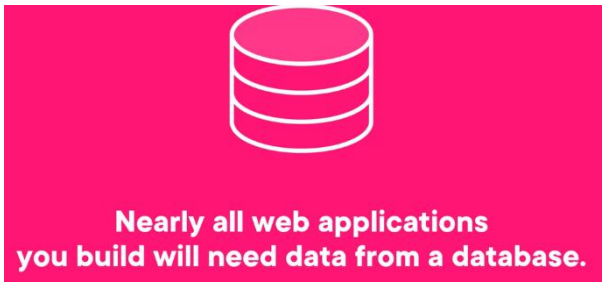
Eseguiamo per verificare che il tutto sia stato integrato correttamente (naturalmente dovremmo impostare manualmente la route nella barra degli indirizzi in quanto non è stata ancora configurata, nel nostro caso: http://localhost:5000/Pie/List)

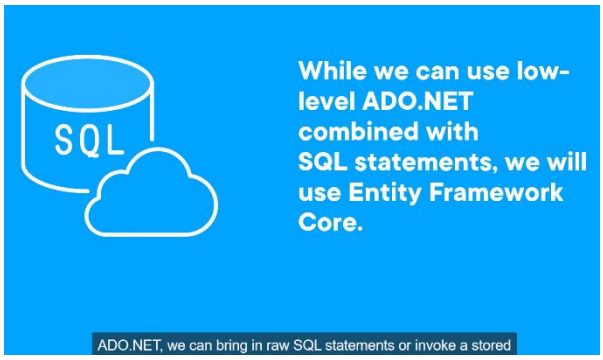


# Working with Real Data Using Entity Framework Core

- 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**



## Introducing Entity Framework Core


Osservazione: ORM "Object relational mapper"

## EF Core

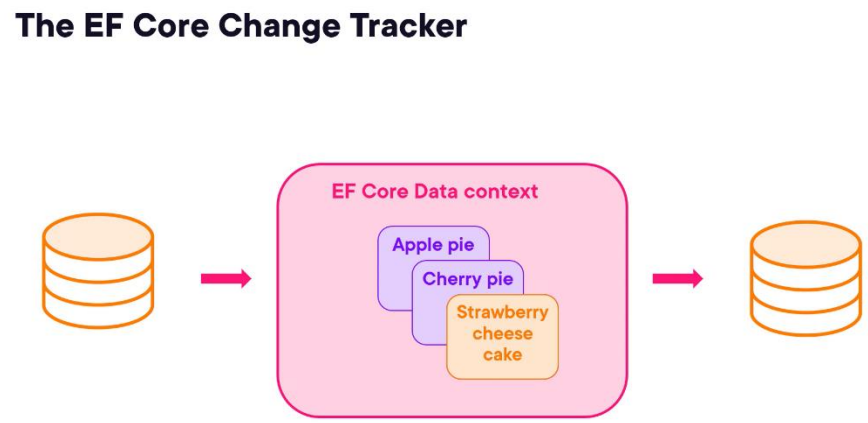

## What EF Core Does for You

**Class**

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

**Table**

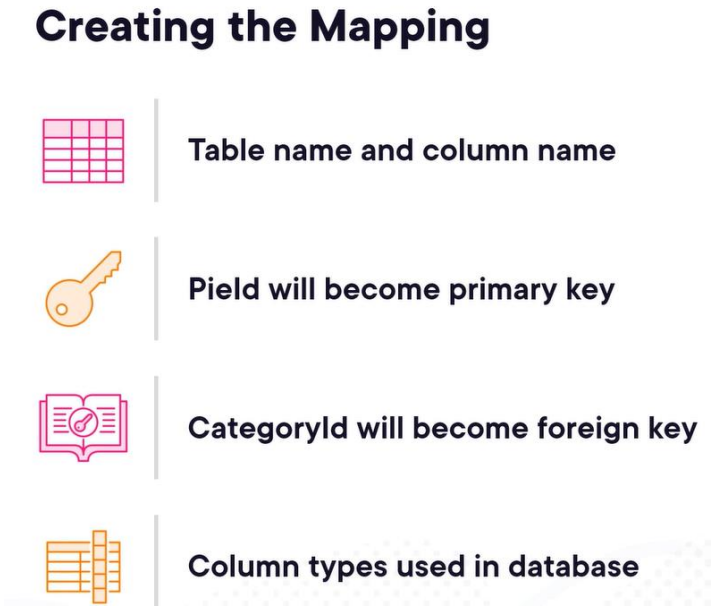
| PieId | Int (PK) |
|---|---|
| Name | String |
| Description | string |




**Adding EF Core to the Application**

```
Microsoft.EntityFrameworkCore.SqlServer      ◂ SQL Server package
Microsoft.EntityFrameworkCore.Tools          ◂ Helper package for Package Manager Console
```

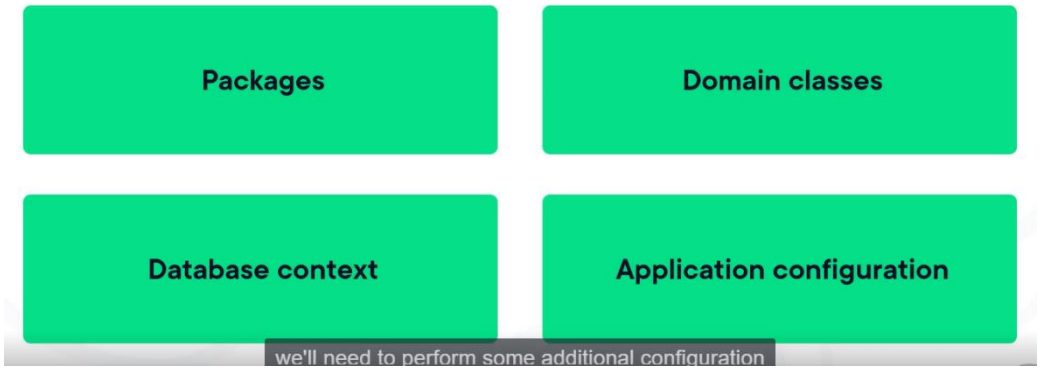
Considerando il nostro progetto Bethany's Pie Shop:



## 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

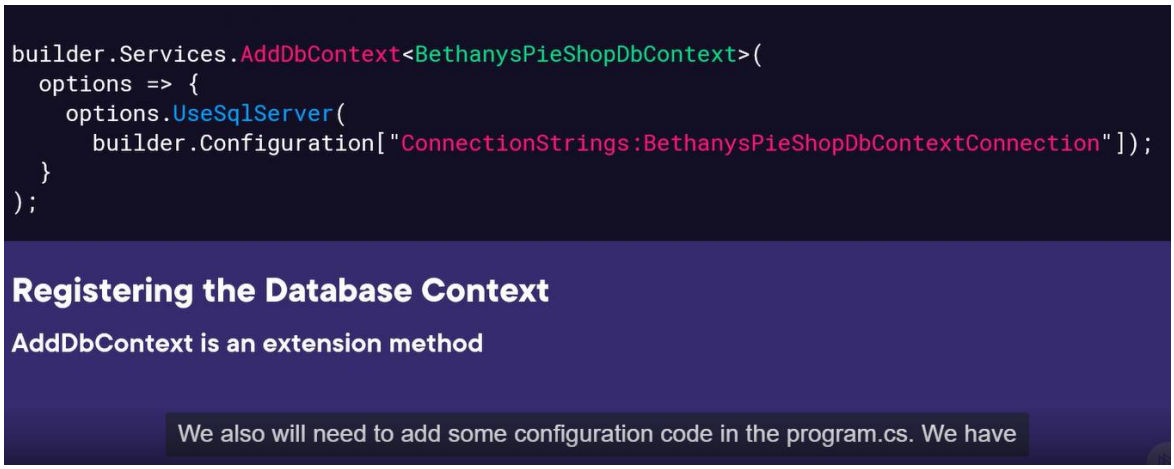



```
{
  "ConnectionStrings": {
    "BethanysPieShopDbContextConnection":
      "Server=(localdb)\\mssqllocaldb;
       Database=BethanysPieShop;
       Trusted_Connection=True;
       MultipleActiveResultSets=true"
  }
}
```

## Adding the Connection String

appSettings.json
Read automatically by default

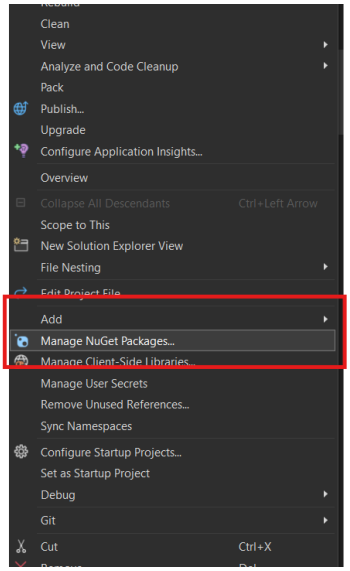
We'll need a way to let our code know how we can connect

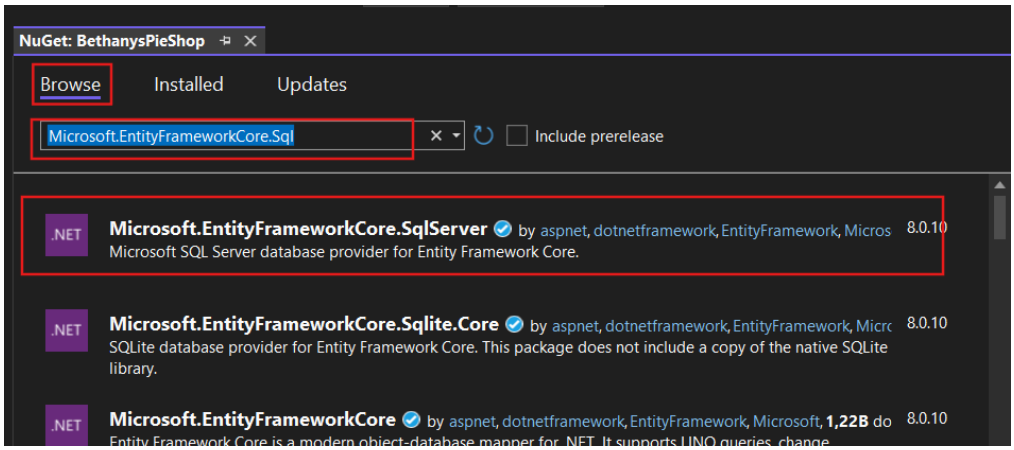



**Demo: Adding EF Core to the Application**

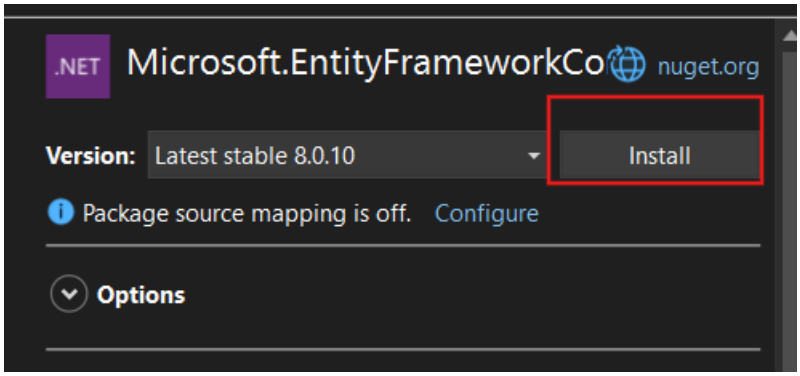
Inseriamo i NuGet packages per lavorare con EF Core.

Click destro sul progetto > manage NuGet packages...

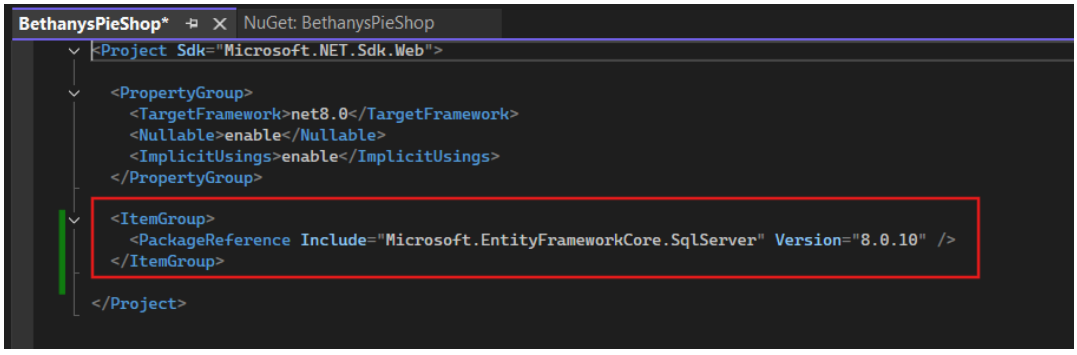





Click destro > Installa




Nel nostro csproj noteremo:




Ora creiamo nella cartella Models la classe BethanysPieShopDbContext.cs:

```cs
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    public class BethanysPieShopDbContext: DbContext
    {
        public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
        {
        }

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

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

```json
{
  "ConnectionStrings": {
    "BethanysPieShopDbContextConnection": "Server=(localdb)\\mssqllocaldb;Database=BethanysPieShop8854;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

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

Aggiungiamo il servizio in Program.cs:

```cs
using BethanysPieShop.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>();
builder.Services.AddScoped<IPieRepository, MockPieRepository>();

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

var app = builder.Build();

app.UseStaticFiles(); // Middleware component: Enable static files

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
}

app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo

app.Run();
```

**Creating the Repository**

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

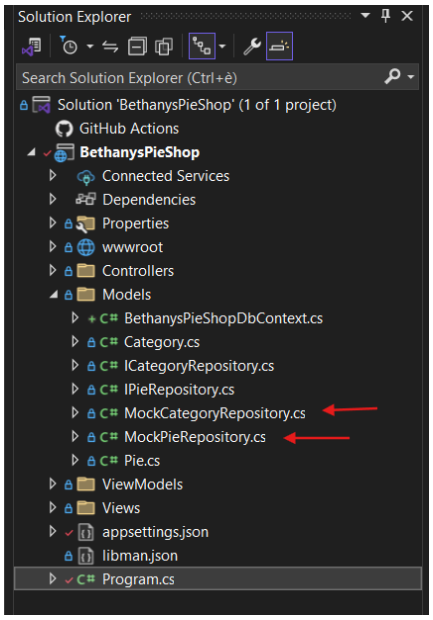
**Querying for Data Using LINQ**

## Adding New Items

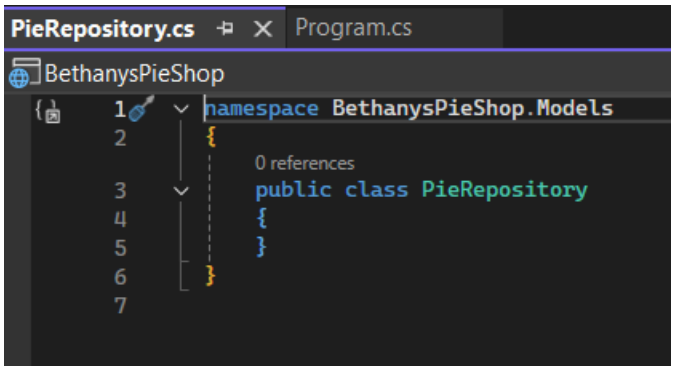
```
foreach (ShoppingCartItem? shoppingCartItem in shoppingCartItems)
{
    var orderDetail = new OrderDetail
    {
        Amount = shoppingCartItem.Amount,
        PieId = shoppingCartItem.Pie?.PieId,
        Price = shoppingCartItem.Pie?.Price
    };

    order.OrderDetails.Add(orderDetail);
}

_bethanysPieShopDbContext.Orders.Add(order);

_bethanysPieShopDbContext.SaveChanges();
```

**Demo: Creating the Repository**

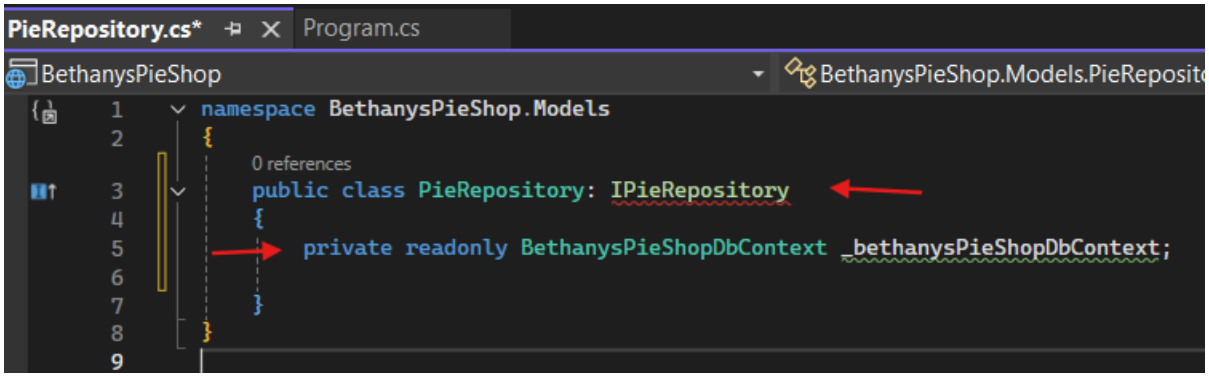
Inizialmente abbiamo usato delle classi mock per poter simulare l'utilizzo dei dati estrapolati dal database:




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




Modifichiamo:

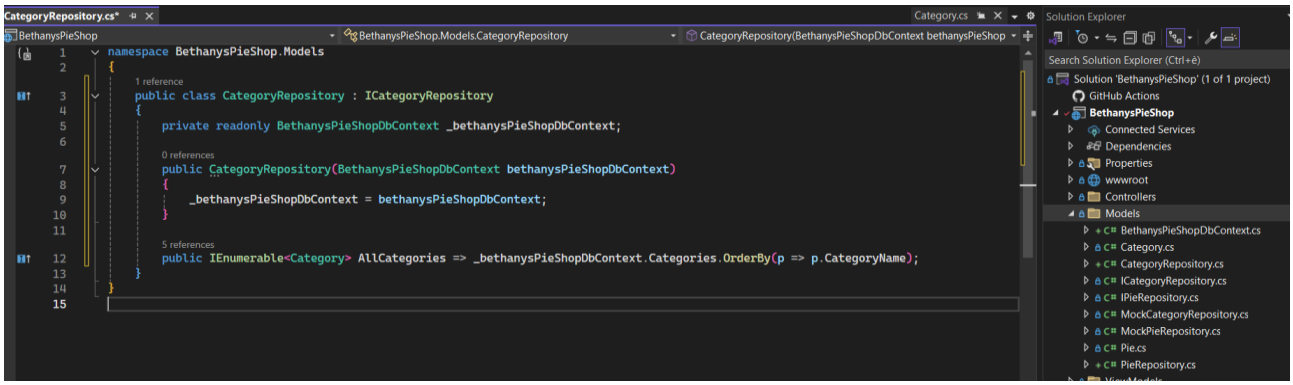



Dato che abbiamo aggiunto il servizio del DbContext possiamo effettuare usare la DI (riga 5)

Continuiamo la modifica del file nel seguente modo:

```
PieRepository.cs*  Program.cs
BethanysPieShop                                   BethanysPieShop.Models.PieReposit
 1  using Microsoft.EntityFrameworkCore;
 2
 3  namespace BethanysPieShop.Models
 4  {
        1 reference
 5      public class PieRepository: IPieRepository
 6      {
 7          private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;
 8
            0 references
 9          public PieRepository(BethanysPieShopDbContext bethanysPieShopDbContext)
10          {
11              _bethanysPieShopDbContext = bethanysPieShopDbContext;
12          }
13
            2 references
14          public IEnumerable<Pie> AllPies
15          {
16              get
17              {
18                  return _bethanysPieShopDbContext.Pies.Include(c=> c.Category);
19              }
20          }
21
            1 reference
22          public IEnumerable<Pie> PiesOfTheWeek
23          {
24              get
25              {
26                  return _bethanysPieShopDbContext.Pies.Include(c => c.Category).Where(p => p.IsPieOfTheWeek);
27              }
28          }
29
            1 reference
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 implementiamo il seguente codice:




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

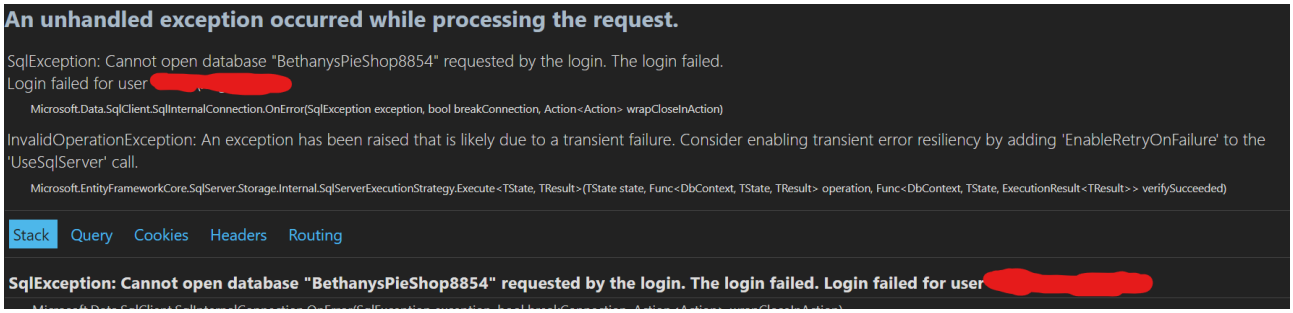
```
Program.cs
BethanysPieShop
1  using BethanysPieShop.Models;
2  using Microsoft.EntityFrameworkCore;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddScoped<ICategoryRepository, MockCategoryRepository>();
7  builder.Services.AddScoped<IPieRepository, MockPieRepository>();
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
```

Cancello semplicemente la parola Mock:

```
Program.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
9  builder.Services.AddControllersWithViews(); // Add MVC services
10 builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
11 {
12     options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"
```

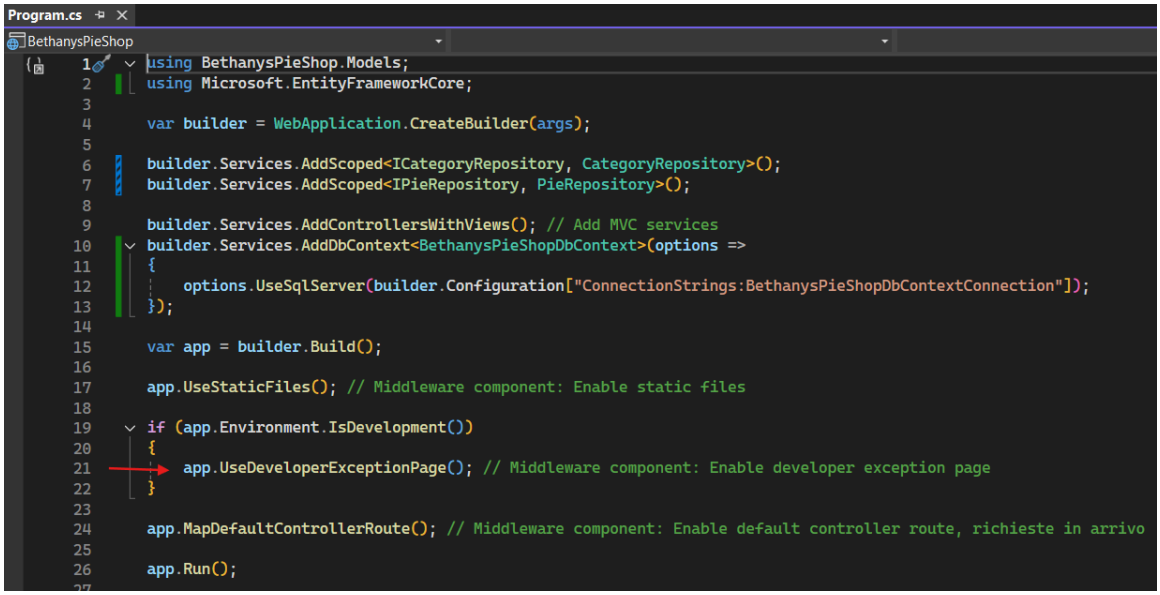
Osservazione: Ciò ci consente di evitare di modificare altri componenti come il Controller.

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



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:




**Using Migrations**

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

La migrazione necessità di due comandi del package Manager Console:

>add-migrations <MigrationName>
>update-database

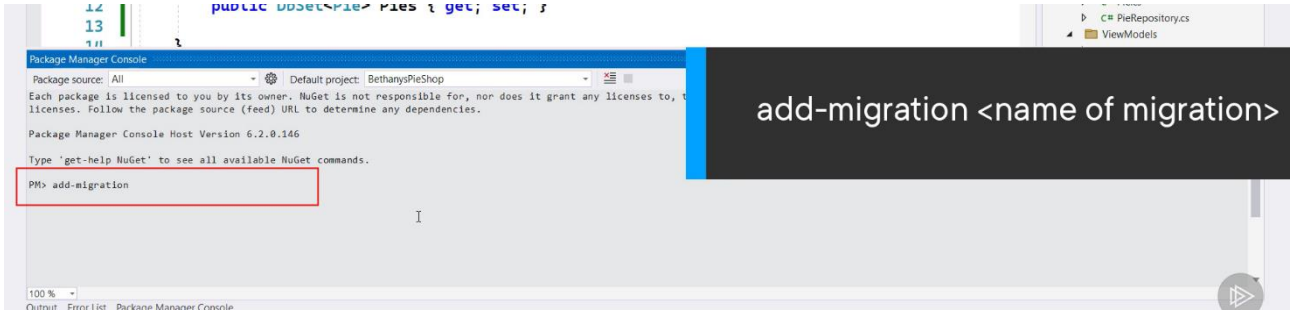
**Demo: Creating Database using migrations**

Ricordandoci di aver installato i seguenti packages nel nostro progetto:

```
<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</IncludeAss
    </PackageReference>
  </ItemGroup>

</Project>
```

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

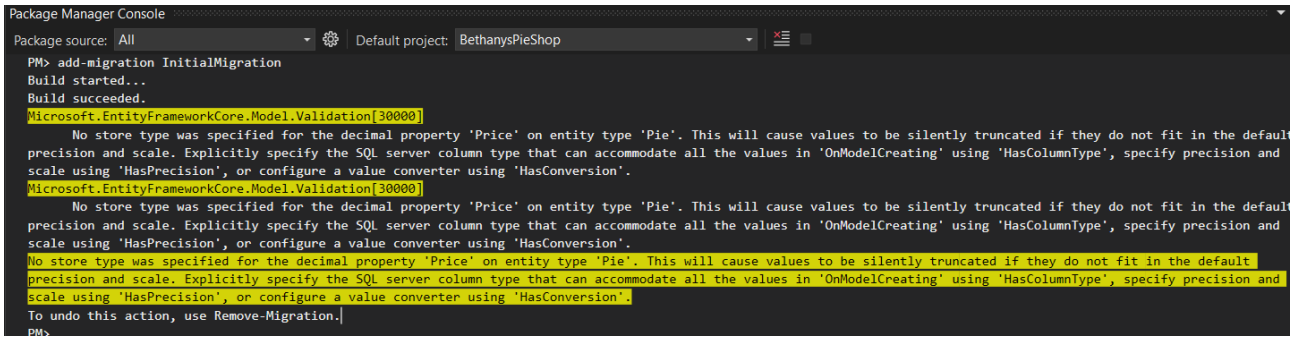
Digitiamo quindi il seguente codice:



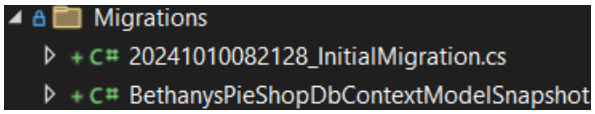

PM> add-migration InitialMigration

(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)***

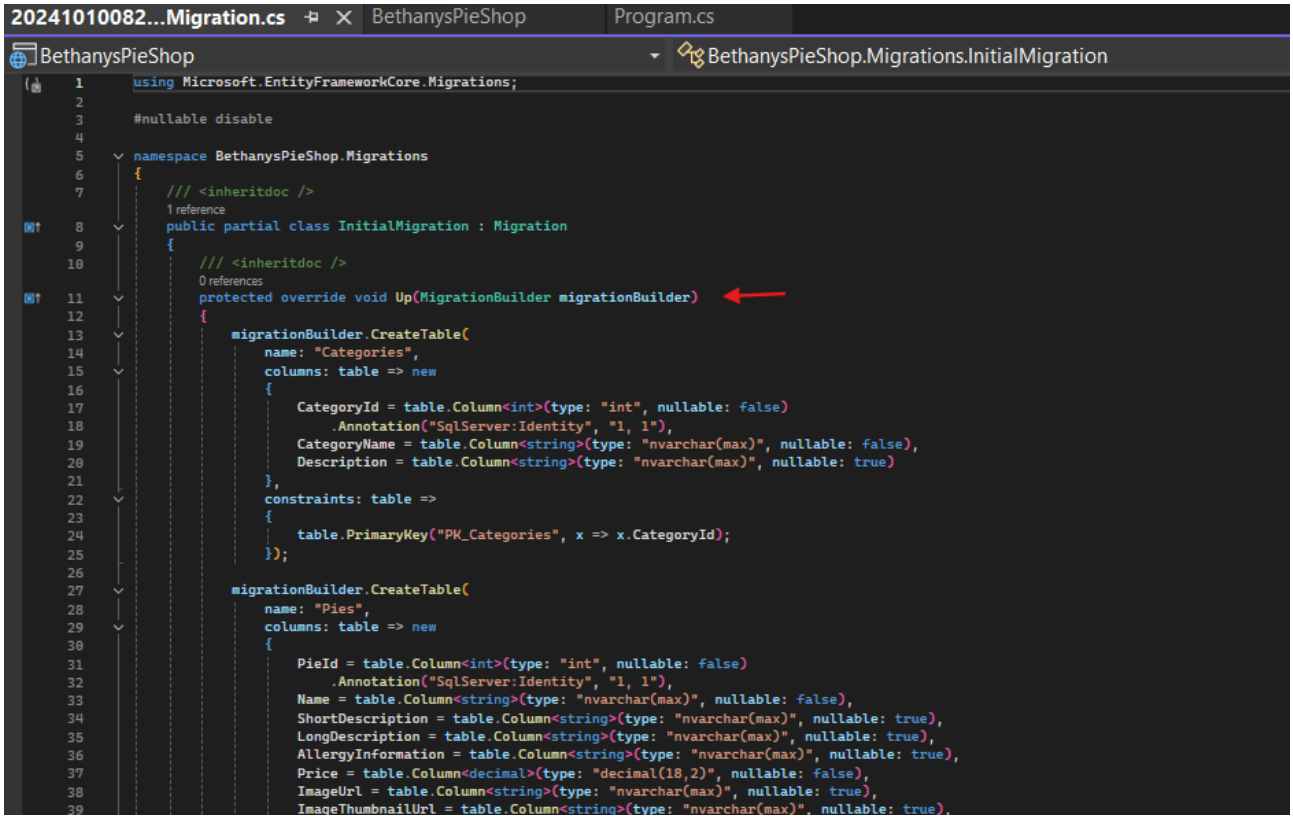
Output:



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

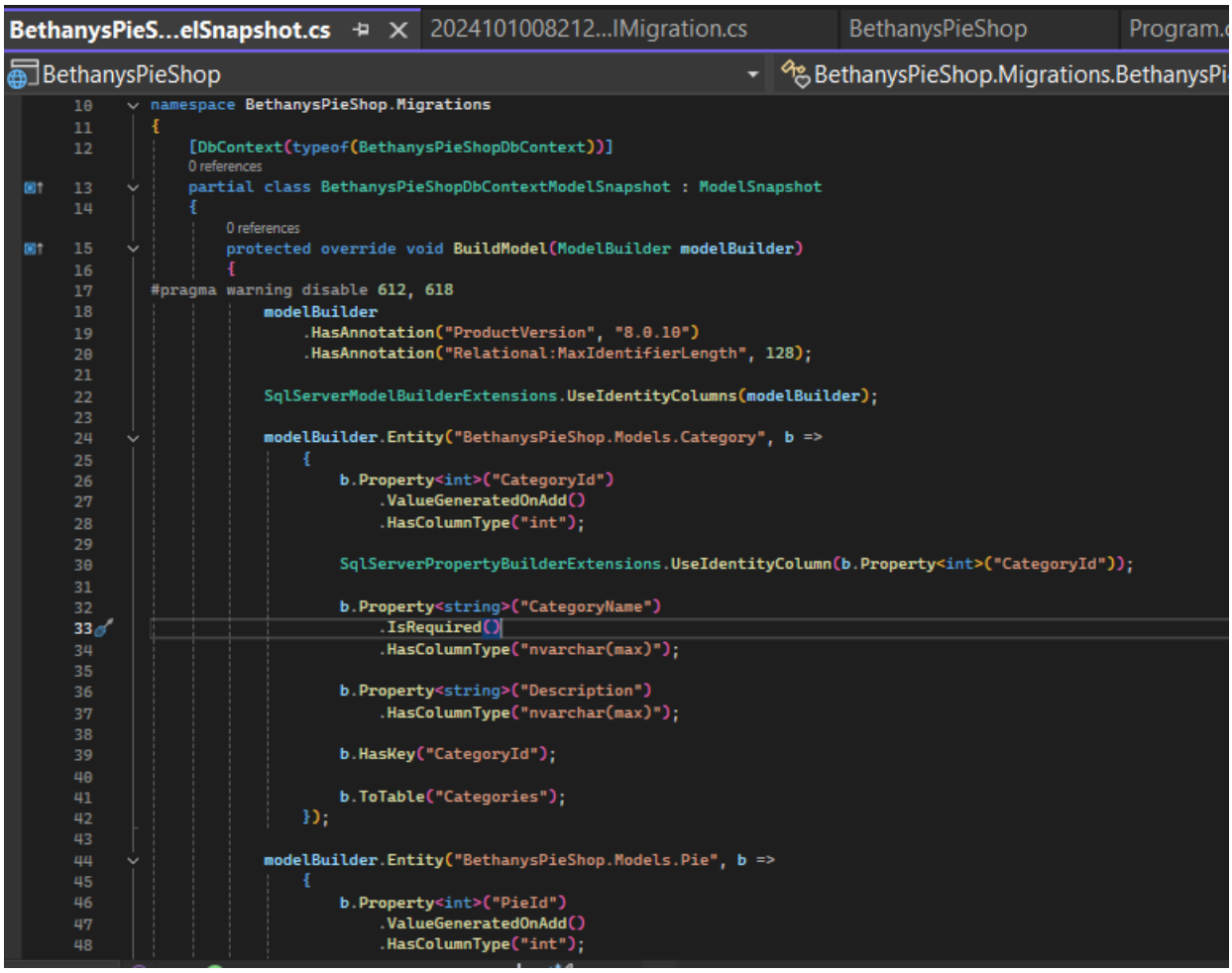



Prima parte di 2024101008218_InitialMigration.cs:




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, cosi 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:




Ora aggiorniamo il Database, quindi ritorniamo al nostro Package Manager Console e digitiamo ***update-database***

Output:

```
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 va
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 va
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 va
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 [BethanysPieShop8854];
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (51ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
```

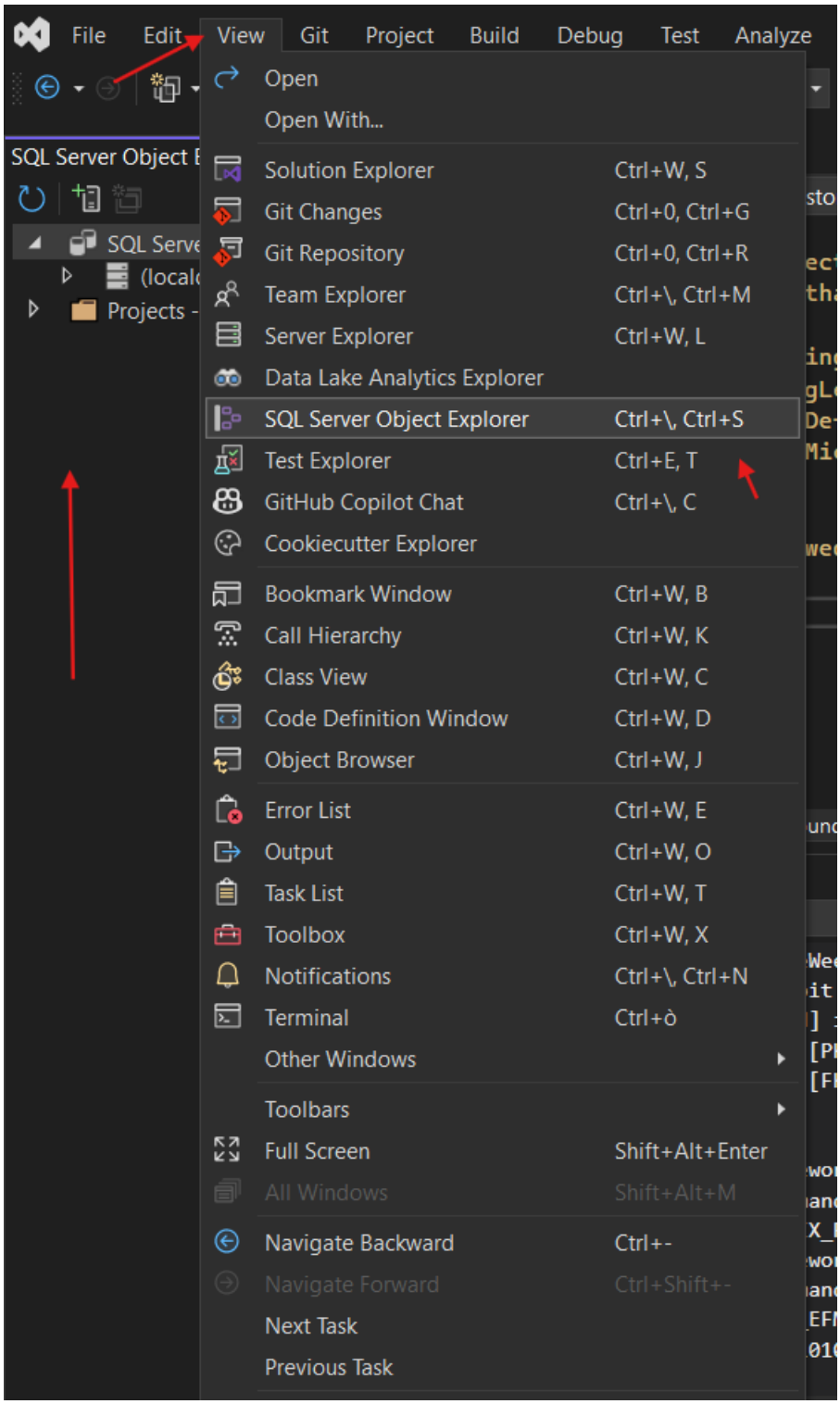
```
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [Pies] (
          [PieId] int NOT NULL IDENTITY,
          [Name] nvarchar(max) NOT NULL,
          [ShortDescription] nvarchar(max) NULL,
          [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:

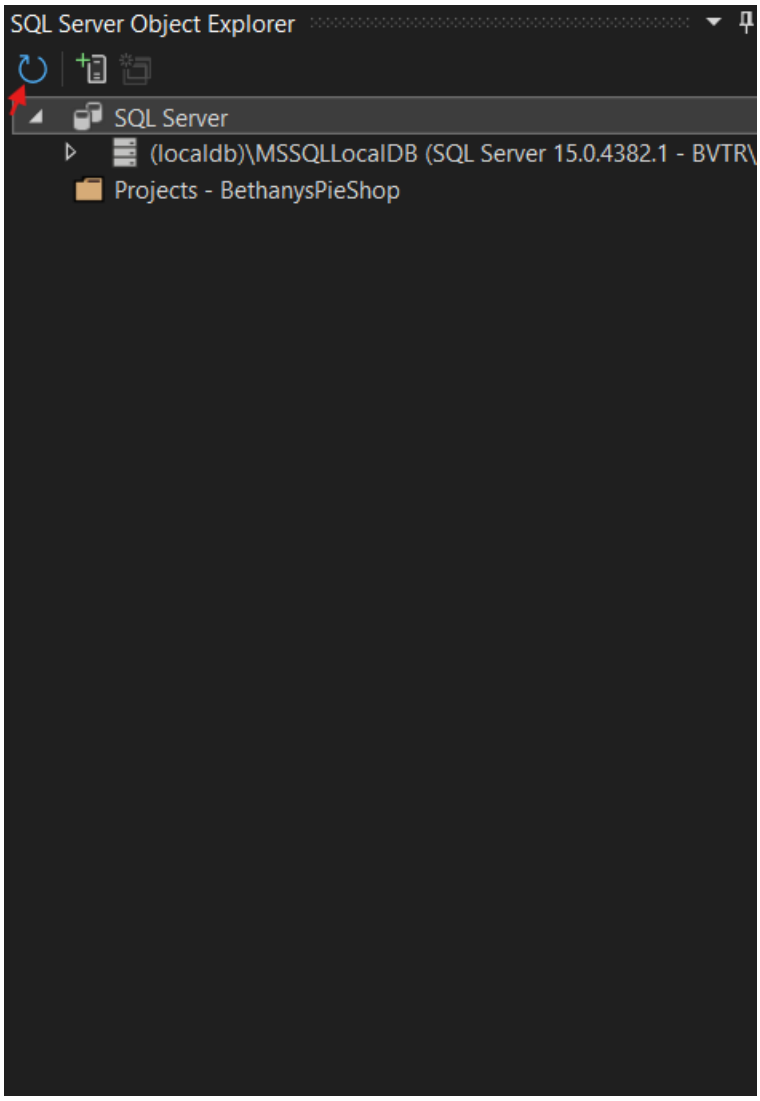
```
Schema: https://json.schemastore.org/appsettings.json
{
  "ConnectionStrings": {
    "BethanysPieShopDbContextConnection": "Server=(localdb)\\mssqllocaldb;Database=BethanysPieShop8854;Trusted_Con
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

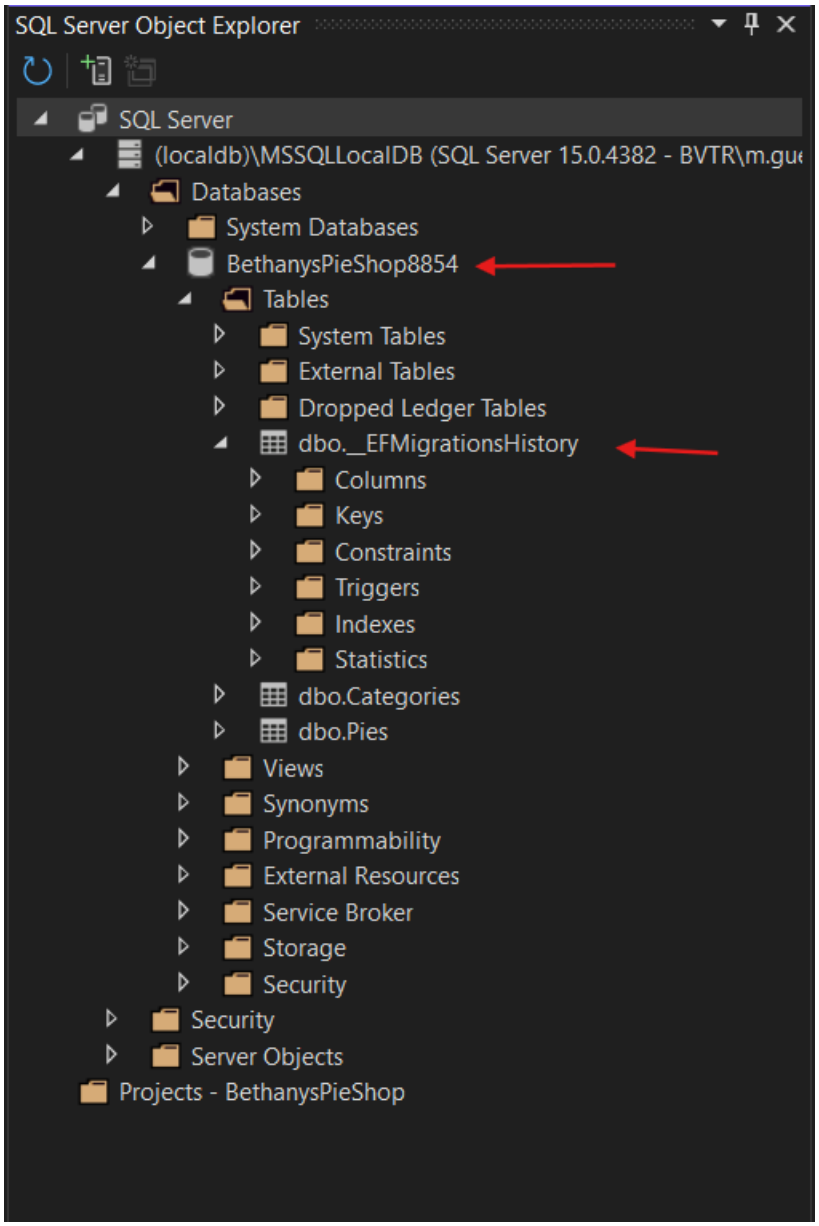
Ora apriamo il pannello per poter visualizzare SQL Server Object Explorer (View > SQL Server Object Explorer)

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

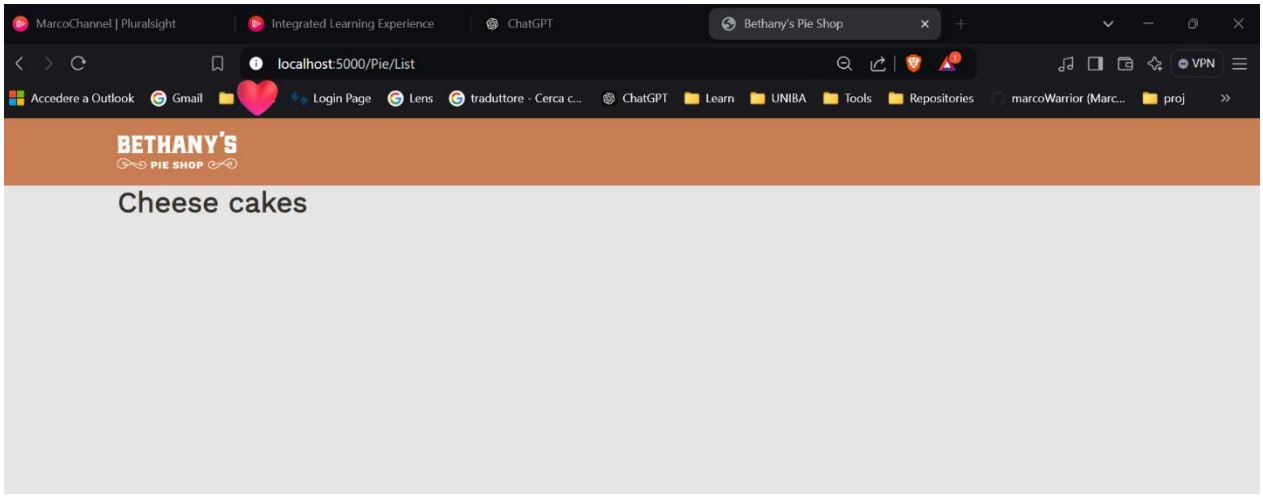



Quindi avremo questa visualizzazione dopo aver cliccato il refresh (freccia blu interno in SQ S.O.E.):


SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 15.0.4382.1 - BVTR\
Projects - BethanysPieShop




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

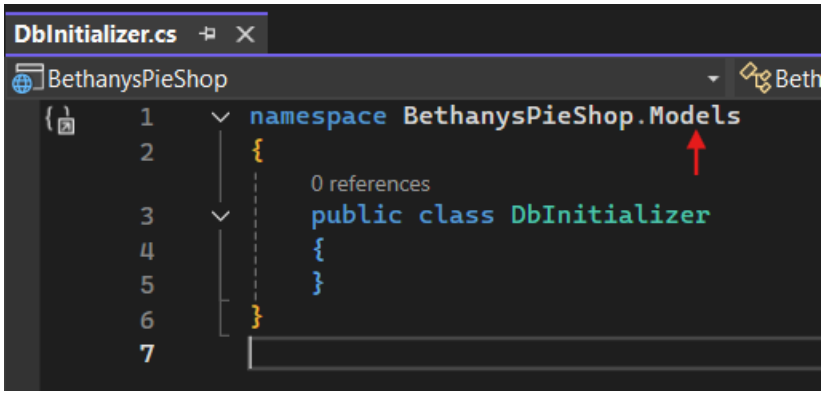


Ciò è normale, in quanto abbiamo creato il Database ma non abbiamo aggiunto ancora alcun dato.

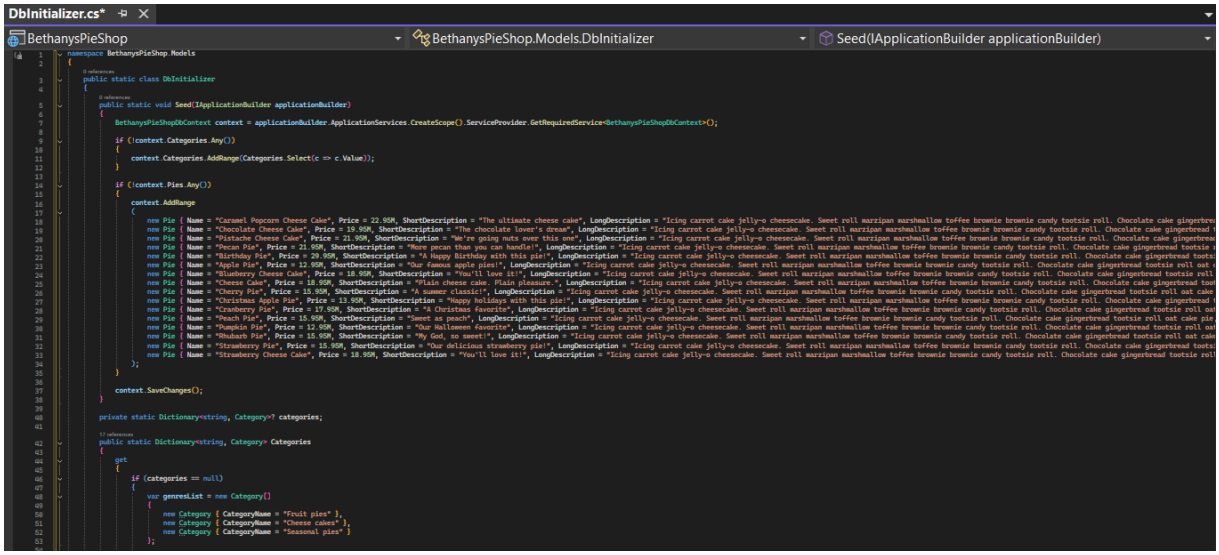
**Demo: Adding Seed Data**
Questa operazione va fatta quando avviamo un progetto nuovo, inoltre è possibile utilizzare EF Core in combinazione con un database già esistente.

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**



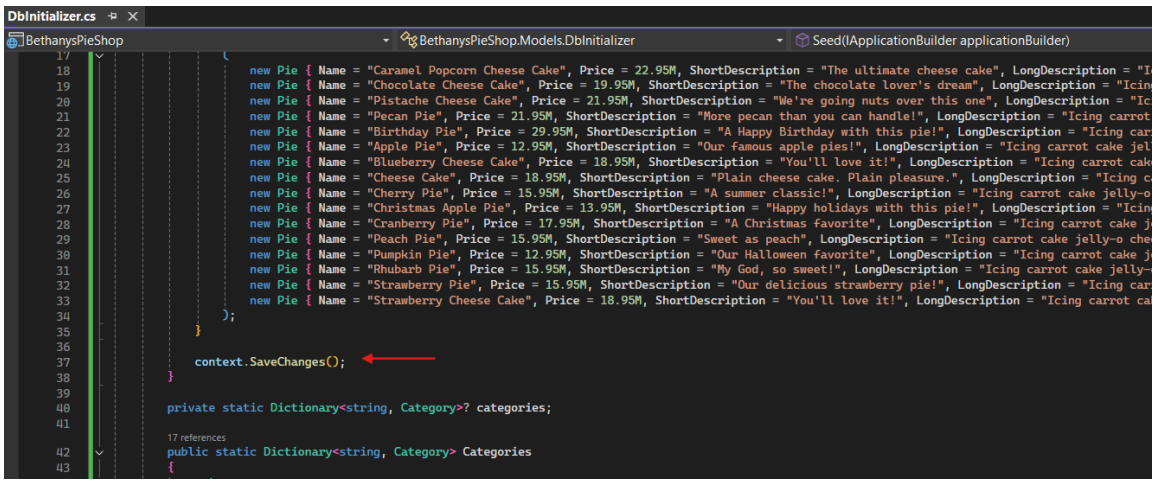

Al suo interno inseriremo una grande quantità di codice




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

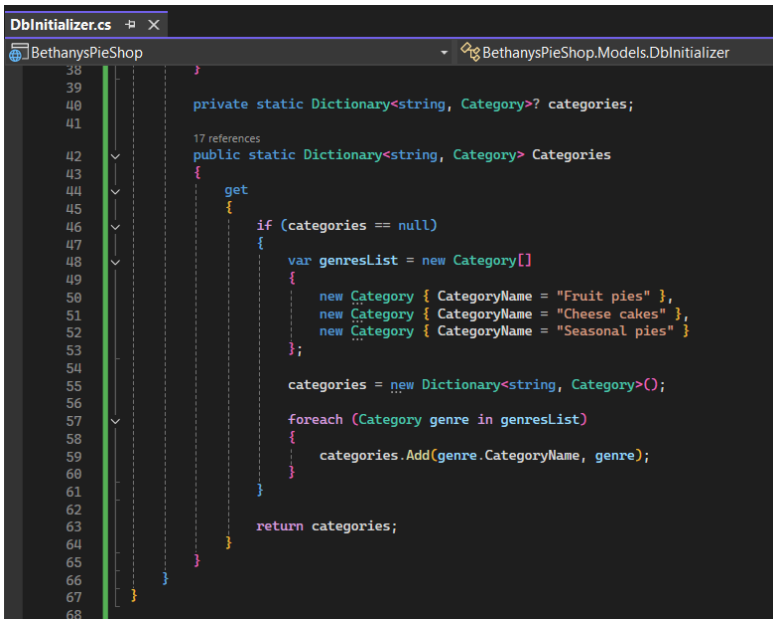
```
namespace BethanysPieShop.Models
{
    0 references
    public static class DbInitializer
    {
        0 references
        public static void Seed(IApplicationBuilder applicationBuilder)
        {
            BethanysPieShopDbContext context = applicationBuilder.ApplicationServices.CreateScope
                ().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 soprastante).
Difatti se stiamo lavorando con un db già esistenze l'if non viene preso in considerazione.
(Stesso comportamento avviene nella parte successiva dove si considerano le varie torte)



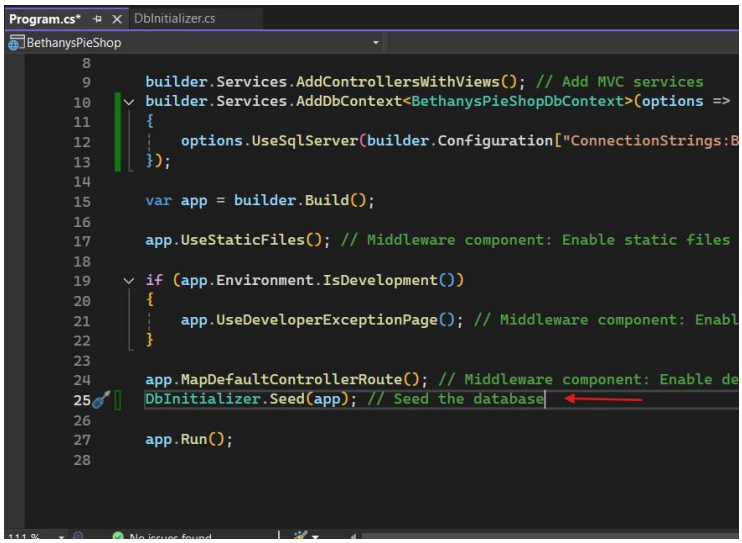

context.SaveChanges(); è anche un tracker delle modifiche, quindi aggiunge effettivamente categorie e torte ai set di dati in memoria nel contesto del database.
Quando viene chiamato esaminerà le raccolte in memoria e vedrà che a tutte queste categorie e torte è stato aggiunto lo stato.

Osservando la parte finale del codice:

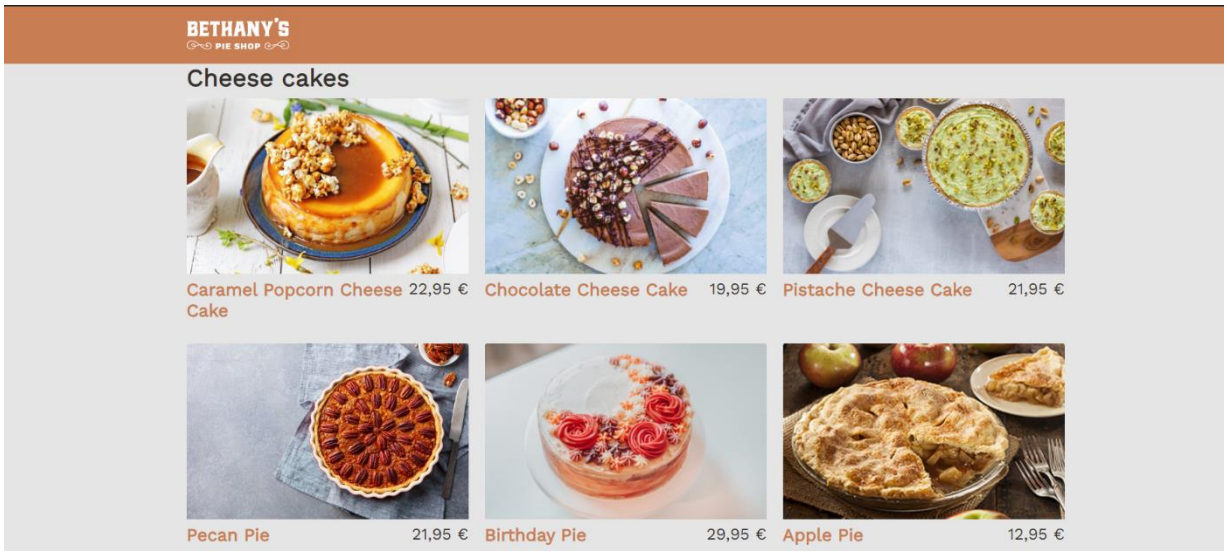



Possiamo notare un Dizionario chiamato Categories, dove ad ogni chiave CategoryName è assegnato il suo valore.

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




Salviamo tutto ed eseguiamo ora il nostro programma:

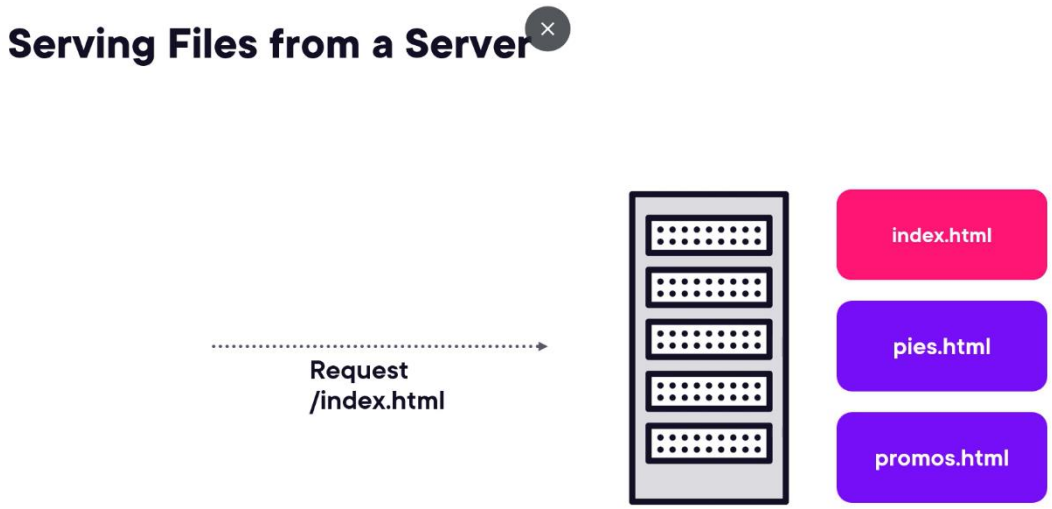



Finalmente siamo in grado di vedere i nostri dati, proveniente da un vero e proprio database.

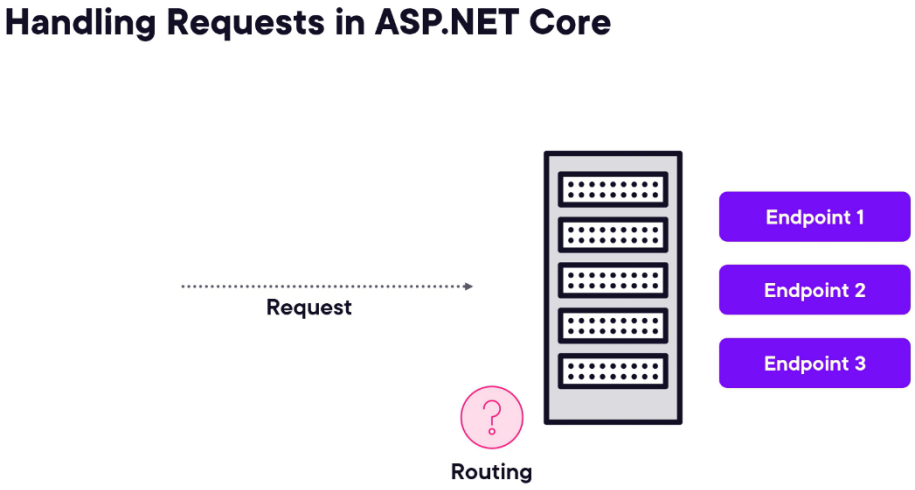
# Adding Routes and Navigation

- 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:




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.



Con MVC, le richieste vengono gestite da metodi di azione del sui 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 mapperò 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 MapGe**t, funzionerà solo se viene inviata una richiesta GET, non quando un altro verbo HTTP viene inviato a questo URL.

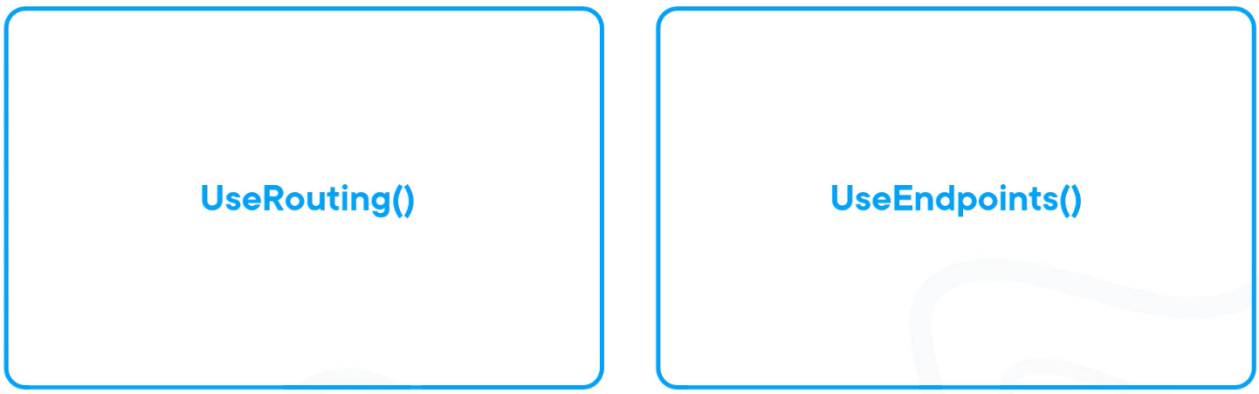
**Endpoints in ASP.NET Core**

**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: introduce funzionalità di corrispondenza del percorso nella pipeline

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

## Routing in ASP.NET Core MVC

**Types of Routing**


## Convention-based Routing

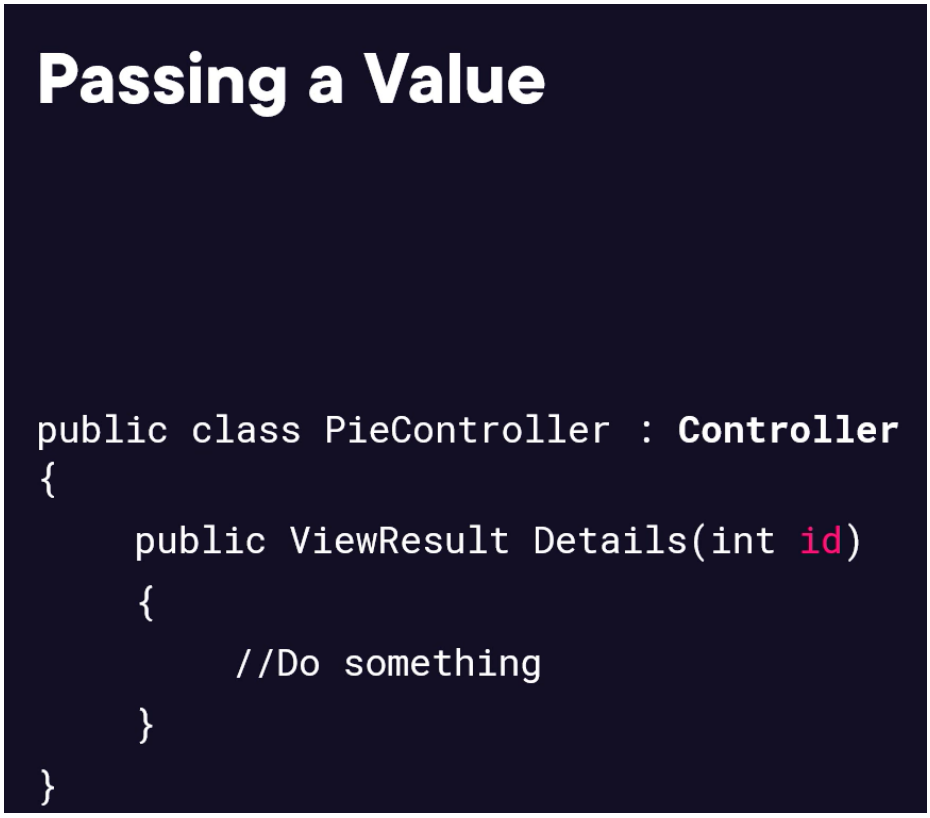

```
app.MapControllerRoute(name: "default", pattern: "{controller}/{action}");
```

## Routing to the Action Method

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

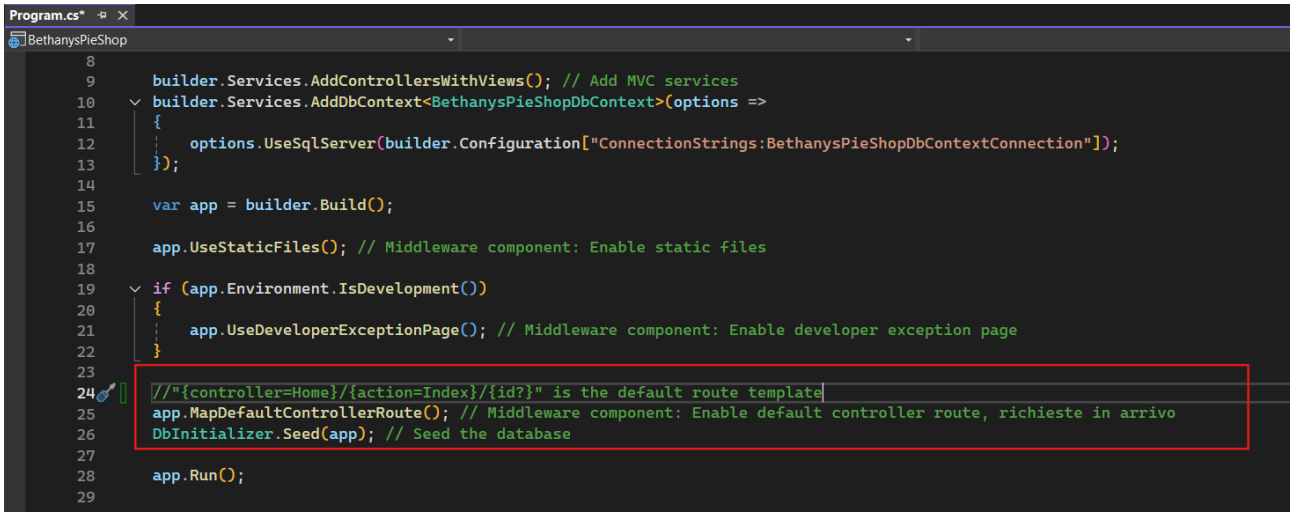
Se dovessimo aver bisogno di più segmenti, immaginiamo di voler visualizzare i dettagli di una torta con ID 1. L'URL della richiesta potrebbe essere:


In tal caso la richiesta potrebbe essere:

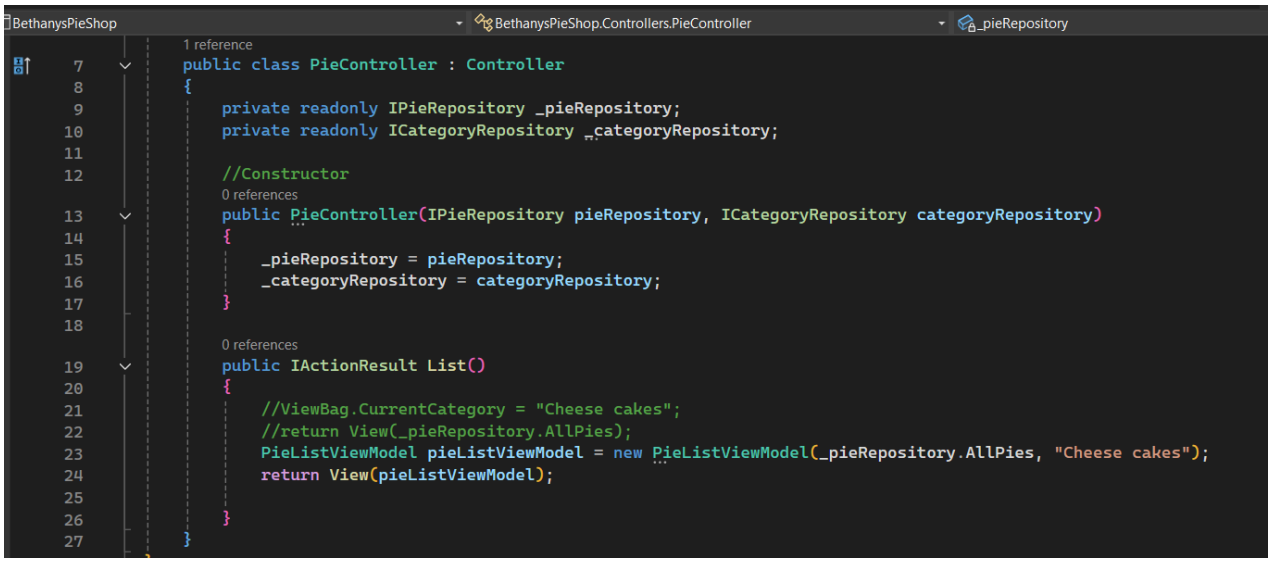



**Demo: Adding Routes to the Application**

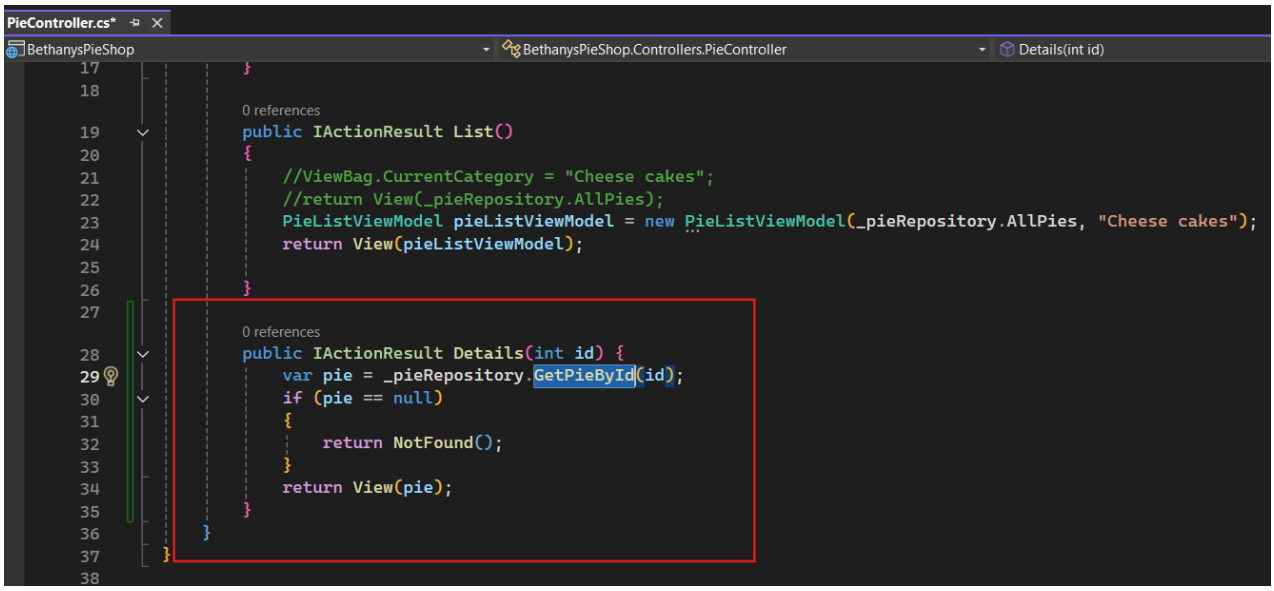
Fino ad ora siamo riusciti comunque a visualizzare le pagine inserendo manualmente l'URL, ciò + stato possibile proprio grazie a tale comando:




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

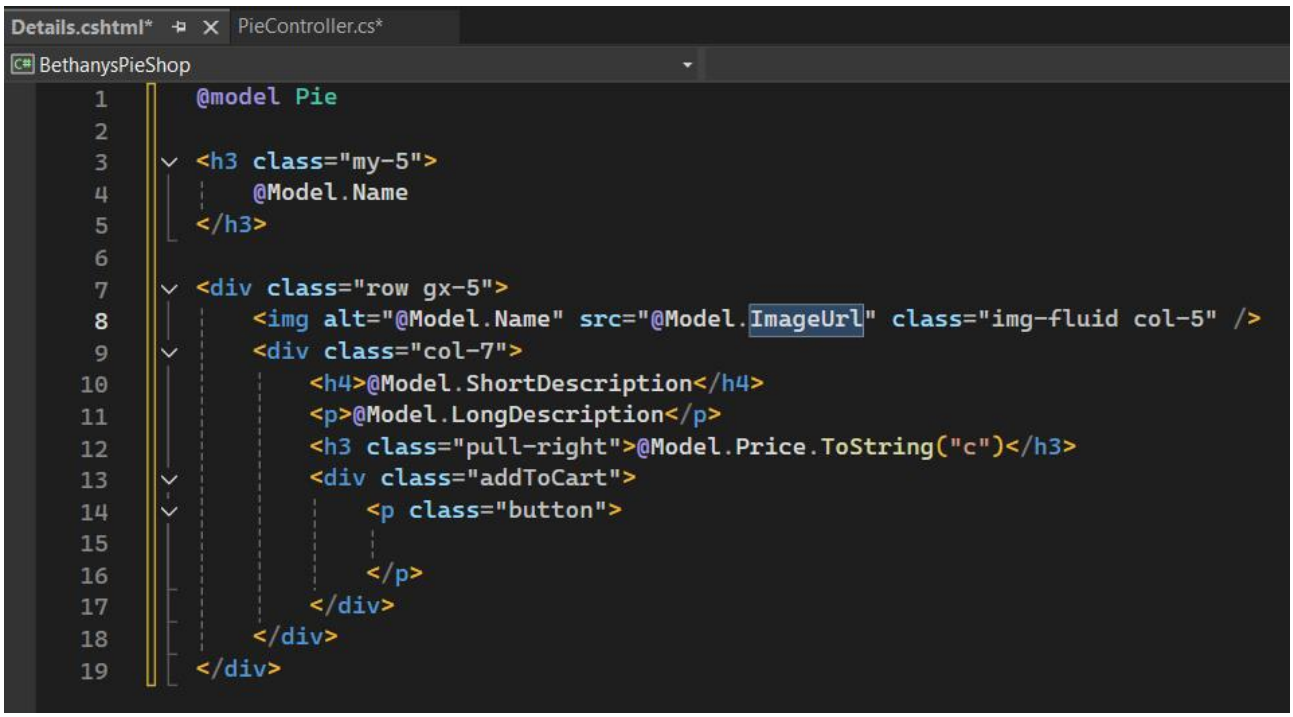


Quindi aggiungiamo ora in PieController il metodo **IActionResult Details ():**

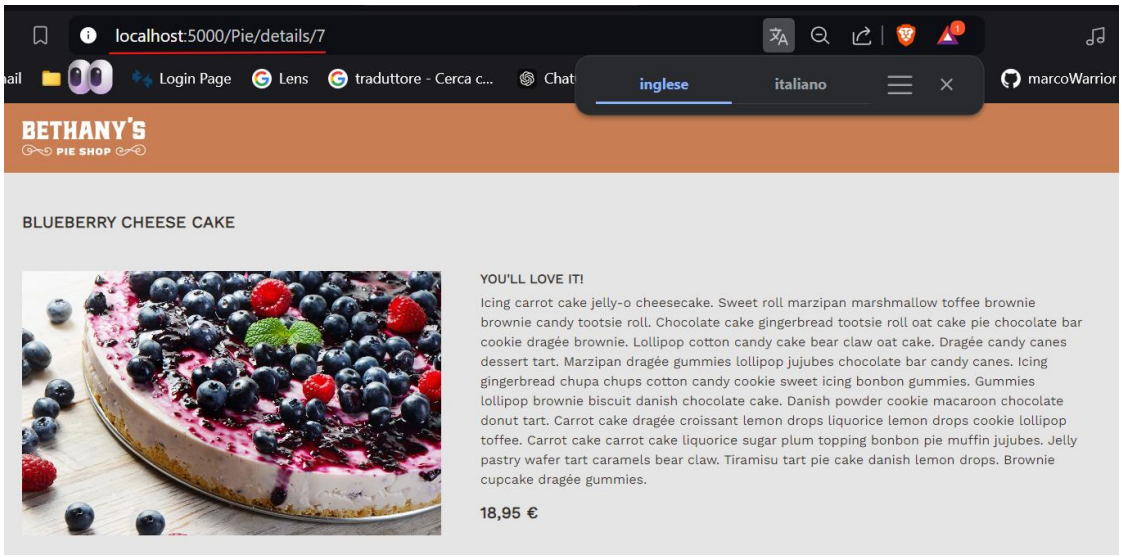



Dopodiché, creiamo una nuova vista Razor empty  chiamata **Details** in Views > Pie

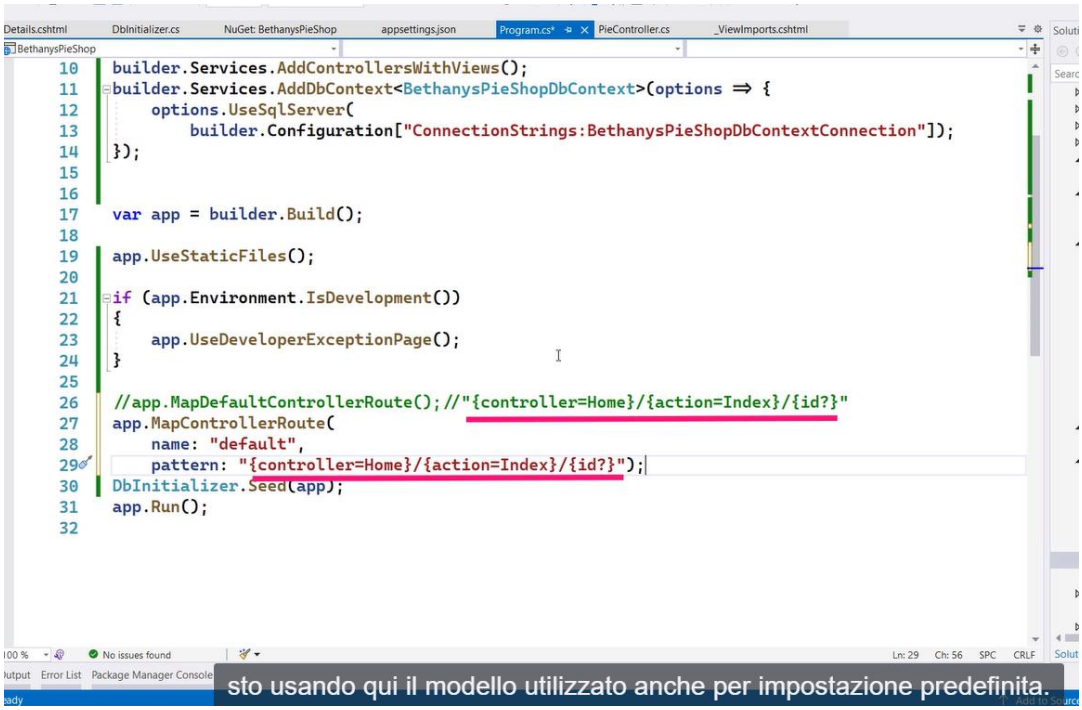
Codice completo di **Details.cshtml**




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




Volendo potremmo personalizzare il MapDEfaultControllerRoute()




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

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

app.UseStaticFiles(); // Middleware component: Enable static files

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage(); // Middleware component: Enable developer exception page
}

//"{controller=Home}/{action=Index}/{id?}" is the default route template
app.MapDefaultControllerRoute(); // Middleware component: Enable default controller route, richieste in arrivo

//Se avessimo voluto personalizzare la rotta o il pattern
//app.MapControllerRoute(
//    name: "defaulte",
//    pattern: "{controller=Home}/{action=Index}/{id?}");

DbInitializer.Seed(app); // Seed the database

app.Run();
```

**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**
- **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:

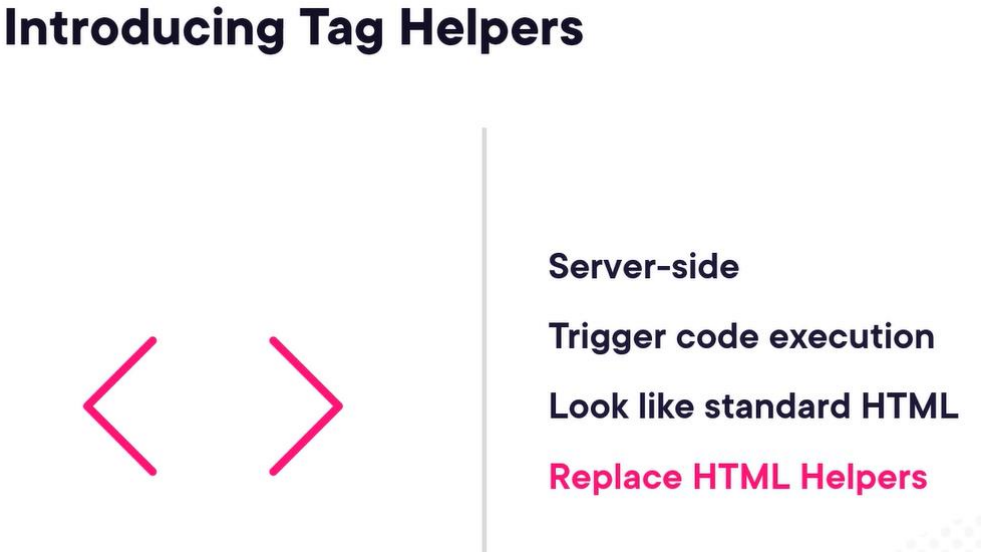
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id:int?}");
```

**Adding Constraints**

**Navigating with Tag Helpers**

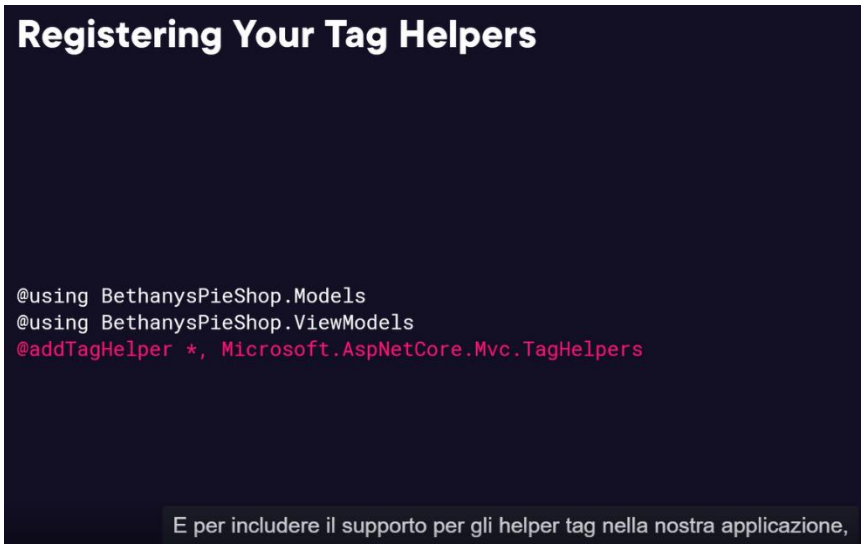
```
<a asp-controller="Pie" asp-action="List">
  View Pie List
</a>
```

**Creating a Link with Tag Helpers**



**Il codice generato sarebbe:**

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

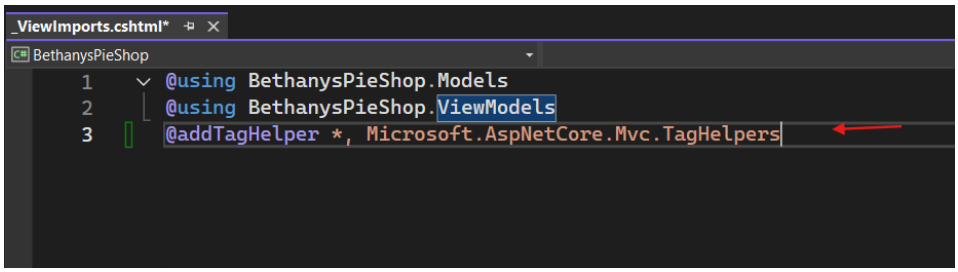




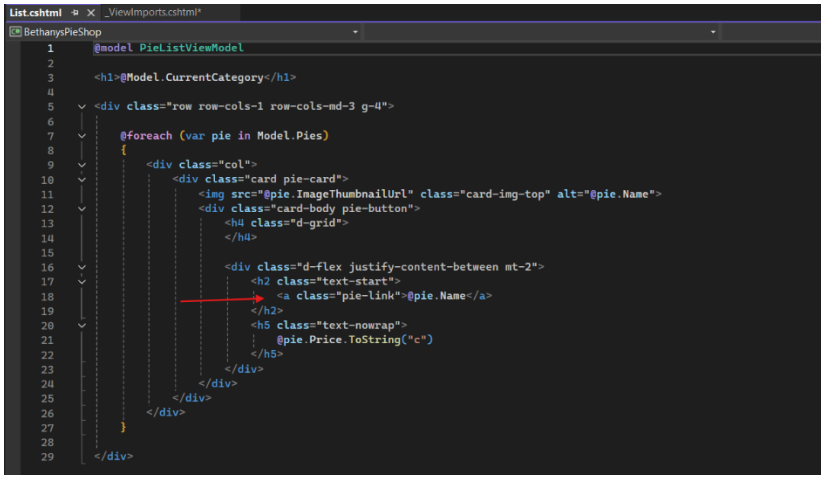
**Demo: Adding Navigation to the site**

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.

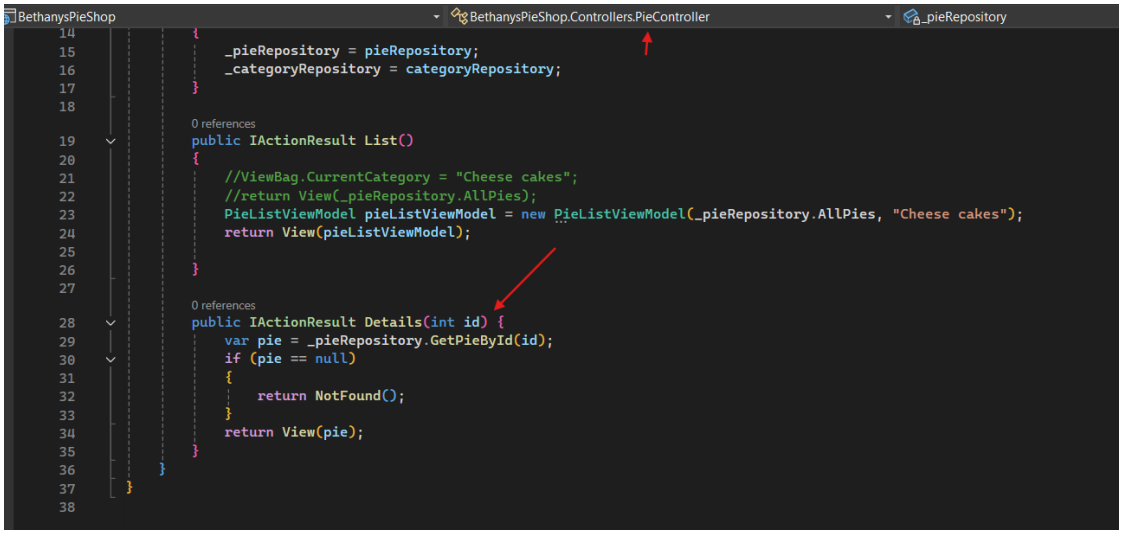
Quindi utilizziamo gli helper tag forniti con ASP.NET Core MVC, per aggiungerli globalmente ci rechiamo in ViewImports.cshtml, presente nella cartella Shared e aggiungiamo una terza istruzione:



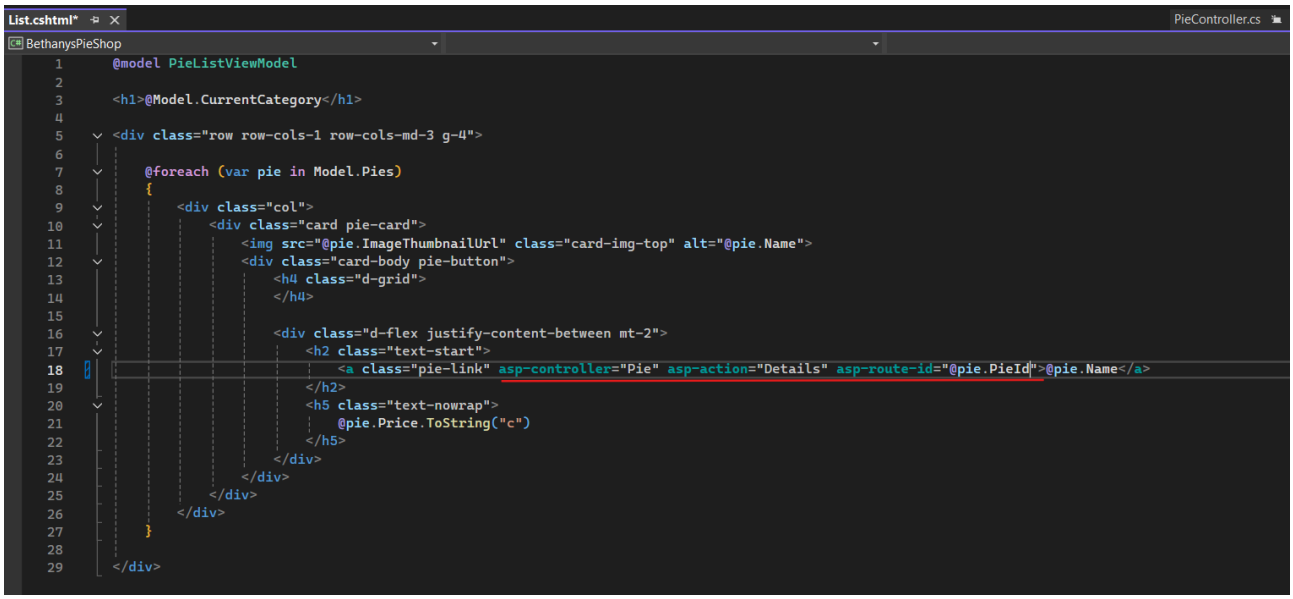

Ci rechiamo alla List.cshtml presente in View > Pie e inseriamo un collegamento:




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:

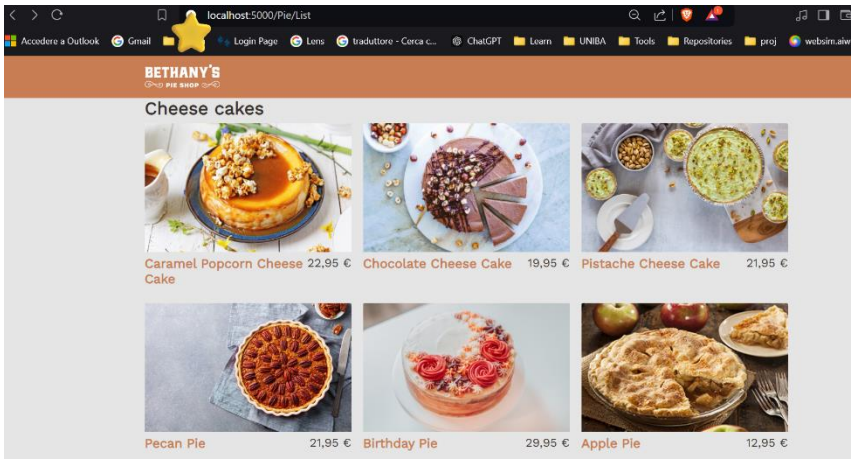


Okay, ora che sappiamo cosa dobbiamo fare, ritorniamo al file List.cshtml e possiamo applicare la modifica nel seguente modo:

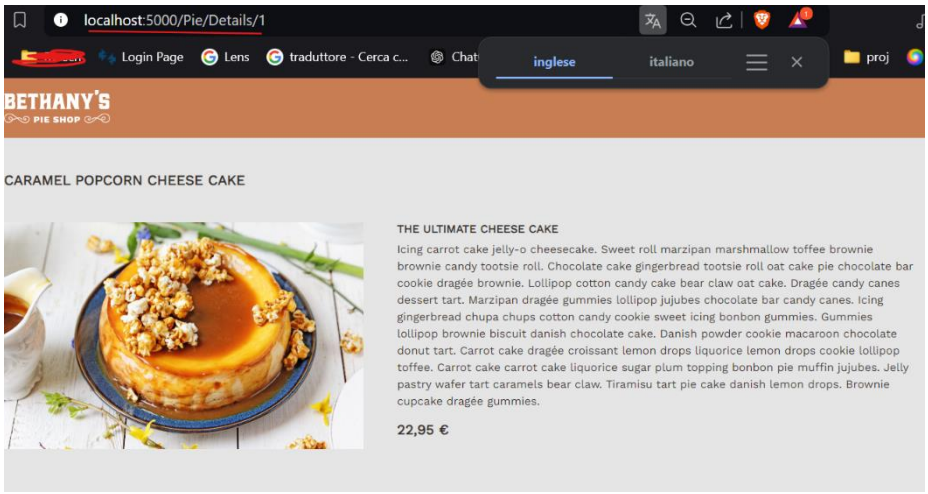



Osservazione: Dinamico, perché possiamo notare a riga 7 il @foreach di itera su ogni Pie in Model.Pies

Ora possiamo navigare nelle pagine presenti in /Pie/List

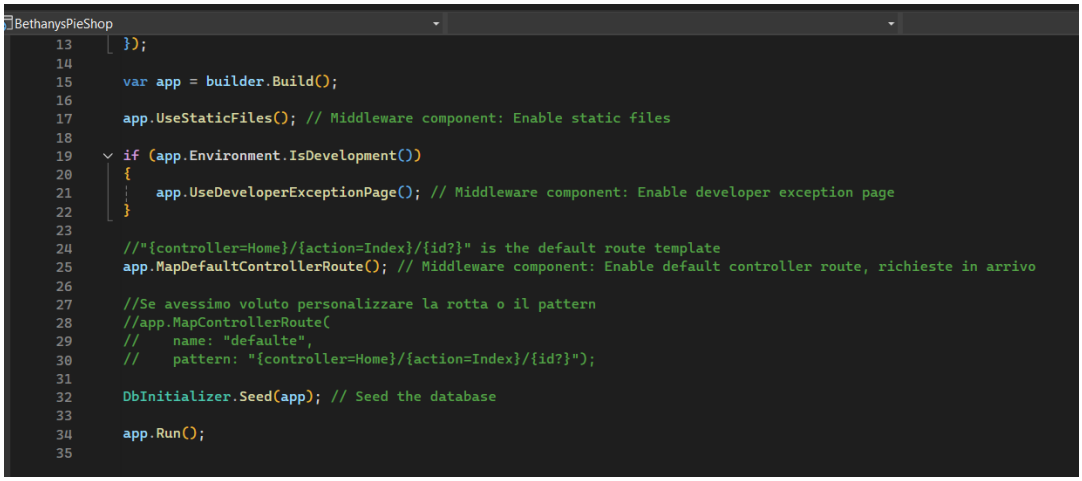



Clicco Caramel Popcorn Cheese:




Adesso è il momento di occuparci della homepage.

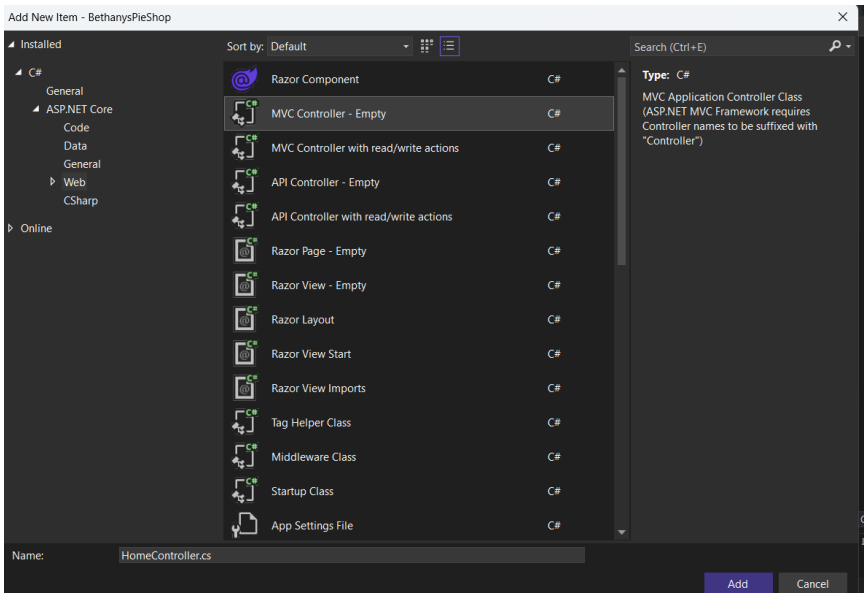
Torniamo al nostro Program.cs

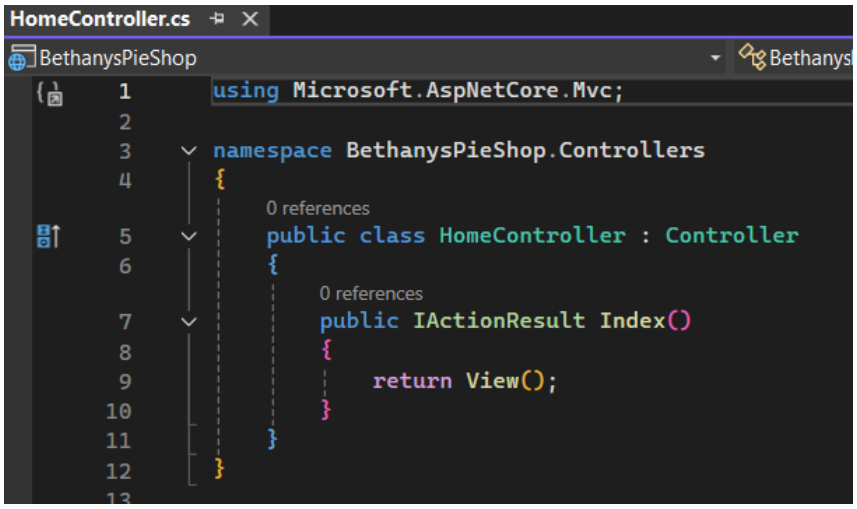



Sappiamo che se non specifichiamo un'azione verrà utilizzato Index.

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




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 IActionResutl List()*, procediamo con la creazione di un modulo specifico.

Quindi creiamo in ViewModels: add>class>HomeViewModel:

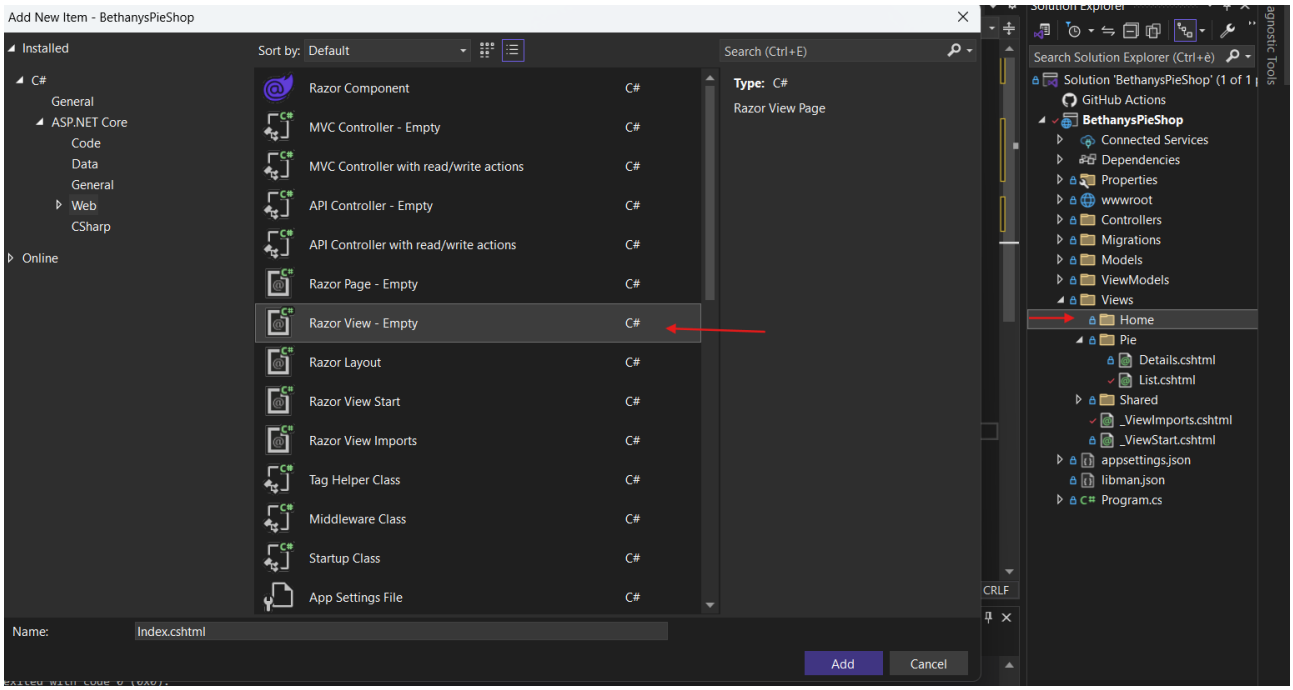
```
HomeViewModel.cs*   HomeController.cs*
BethanysPieShop                          BethanysPieShop.ViewModels.HomeViewMo
1   namespace BethanysPieShop.ViewModels
2   {
        1 reference
3       public class HomeViewModel
4       {
            1 reference
5           public IEnumerable<Pie> PiesOfTheWeek { get;}
6
            0 references
7           public HomeViewModel(IEnumerable<Pie> piesOfTheWeek)
8           {
9               PiesOfTheWeek = piesOfTheWeek;
10          }
11      }
12  }
13
```

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

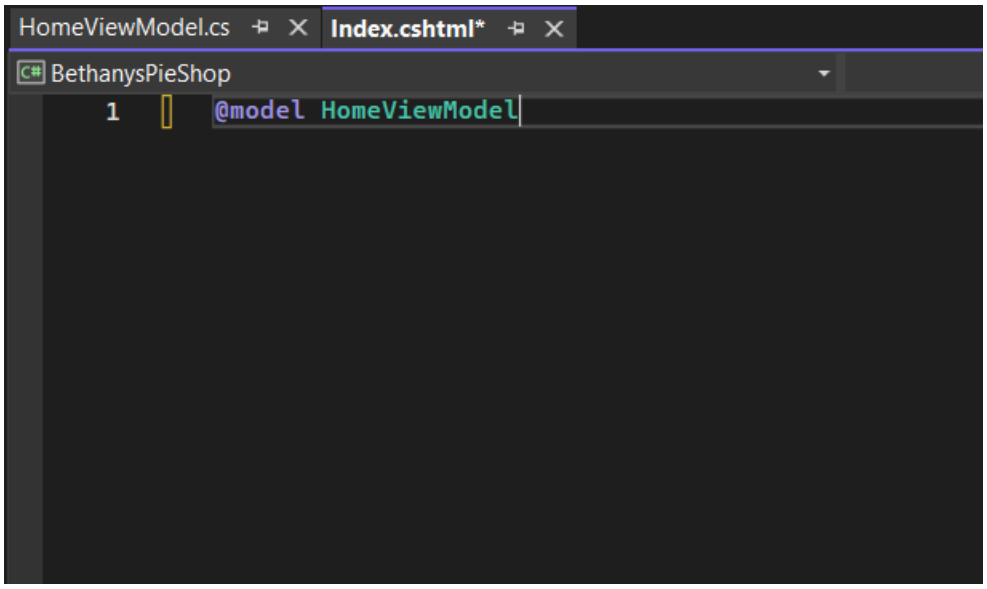
Torniamo ora in HomeController.cs e modifichiamo Index() nel seguente modo:

```
PieRepository.cs   HomeController.cs*
BethanysPieShop                          BethanysPieShop.Controllers.HomeController
1   using BethanysPieShop.Models;
2   using BethanysPieShop.ViewModels;
3   using Microsoft.AspNetCore.Mvc;
4
5   namespace BethanysPieShop.Controllers
6   {
        1 reference
7       public class HomeController : Controller
8       {
9           private readonly IPieRepository _pieRepository;
            0 references
10          public HomeController(IPieRepository pieRepository)
11          {
12              _pieRepository = pieRepository;
13          }
            0 references
14          public IActionResult Index()
15          {
16              var piesOfTheWeek = _pieRepository.PiesOfTheWeek;
17              var homeViewModel = new HomeViewModel(piesOfTheWeek);
18              return View(homeViewModel);
19          }
20      }
21  }
```

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




Lo modifichiamo nel seguente modo:




Incolliamo il nostro snippet:

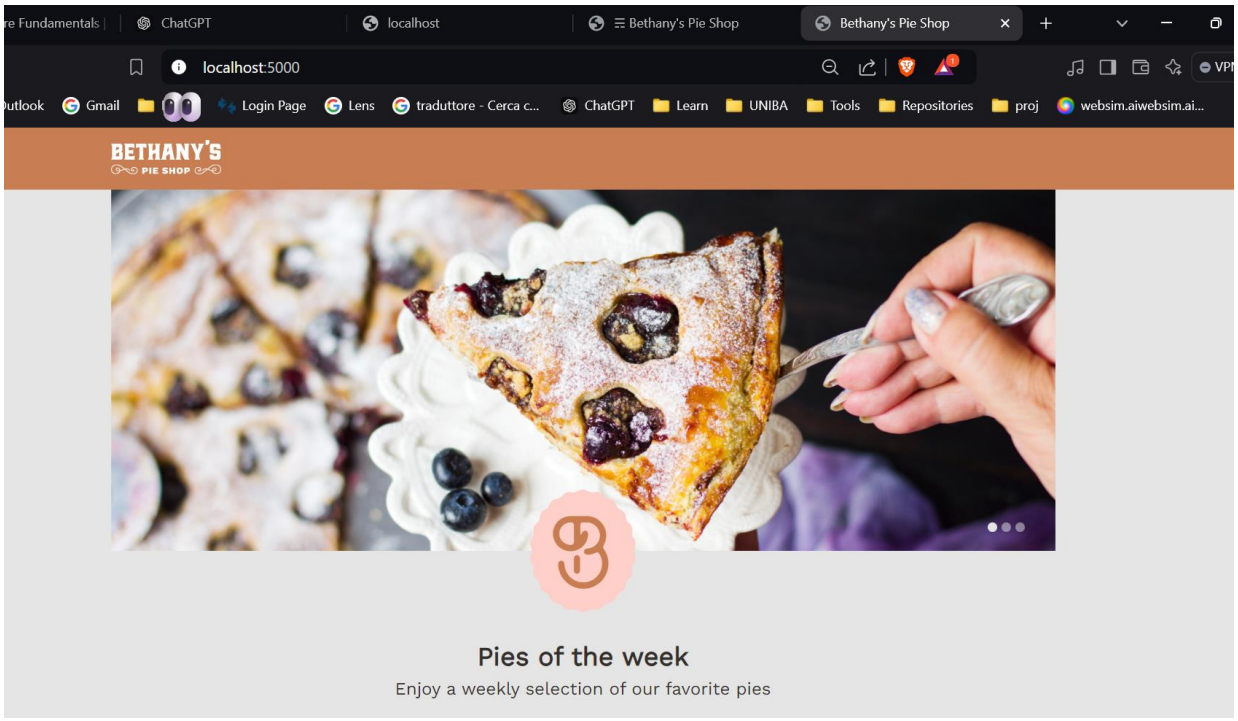
HomeViewModel.cs | Index.cshtml*

BethanysPieShop

```
@model HomeViewModel


<div id="carouselImages" class="carousel slide" data-bs-ride="true">
    <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">
            <img src="~/Images/carousel1.jpg" class="d-block w-100" alt="Carousel image 1">
        </div>
        <div class="carousel-item">
            <img src="~/Images/carousel2.jpg" class="d-block w-100" alt="Carousel image 2">
        </div>
        <div class="carousel-item">
            <img src="~/Images/carousel3.jpg" class="d-block w-100" alt="Carousel image 3">
        </div>
    </div>
</div>

<div class="text-center">
    <img src="Images/bethanys-pie-shop-logomark.png" class="logo-mark mx-auto" />
    <h1>Pies of the week</h1>
    <h5>Enjoy a weekly selection of our favorite pies</h5>

    <div class="row pies-of-the-week">
        <div class="row row-cols-1 row-cols-md-3 g-4">
            @foreach (var pie in Model.PiesOfTheWeek)
            {
                <div class="col">
                    <div class="card pie-card">
                        <img src="@pie.ImageThumbnailUrl" class="card-img-top" alt="@pie.Name">
                        <div class="card-body pie-button">
                            <h4 class="d-grid">
                            </h4>

                            <div class="d-flex justify-content-between mt-2">
                                <h2 class="text-start">
                                    <a asp-controller="Pie"
                                       asp-action="Details"
                                       asp-route-id="@pie.PieId"
                                       class="pie-link">@pie.Name</a>
                                </h2>
                                <h5 class="text-nowrap">
                                    @pie.Price.ToString("c")
                                </h5>
                            </div>
                        </div>
                    </div>
                </div>
            }
        </div>

    </div>
</div>
```

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)

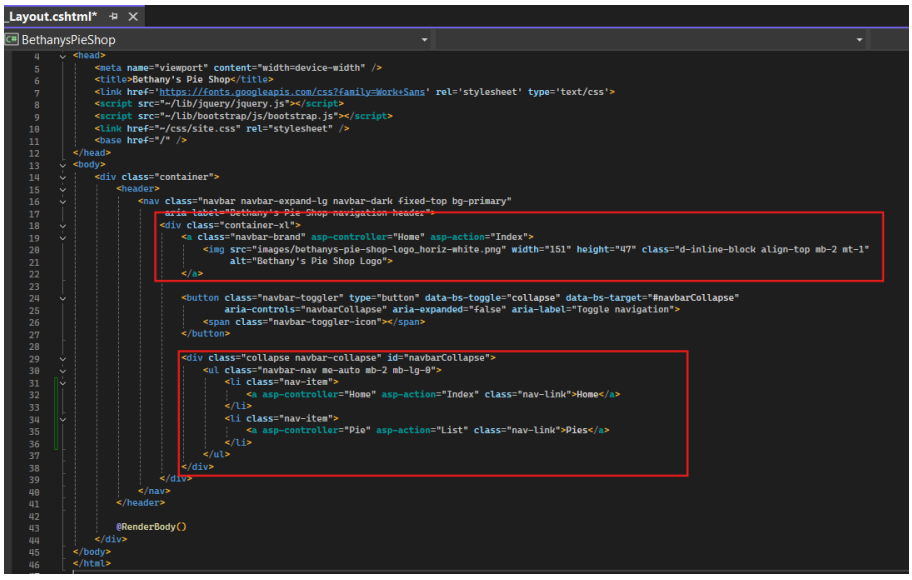
Salviamo e avviamo il programma:




Perfetto, abbiamo la nostra home page.

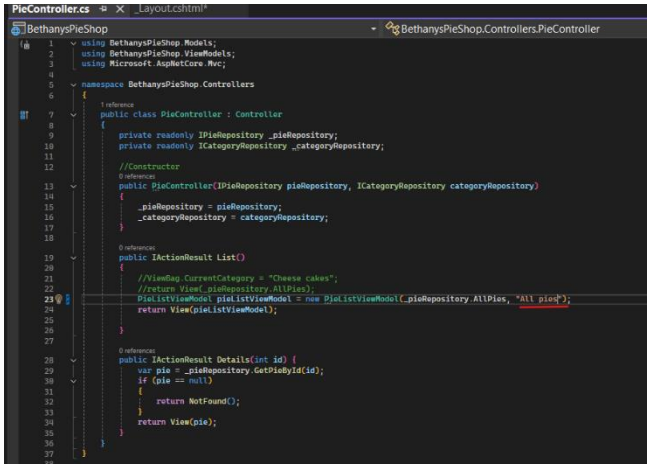
Sembra funzionare tutto perfettamente, ma non riesco ancora a raggiungere ogni pagina utilizzando un collegamento.
Cambiamo la situazione, estenderò la mia navigazione, la mia intestazione con altri collegamenti.

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

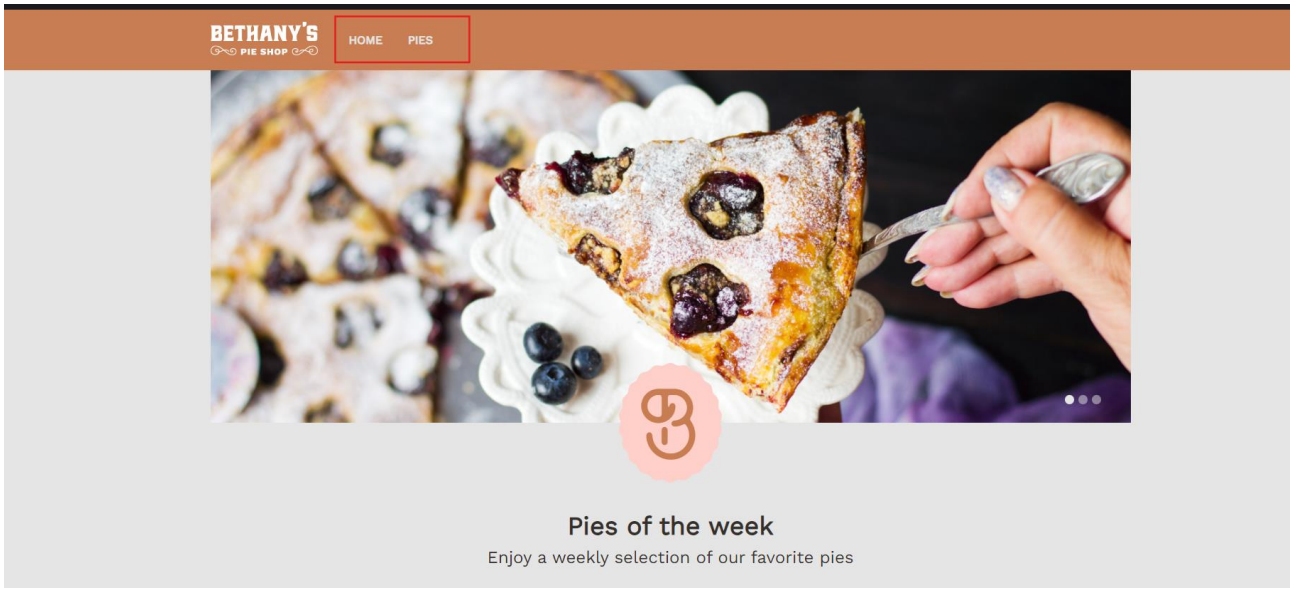


(Modifiche applicate nelle zone rosse)

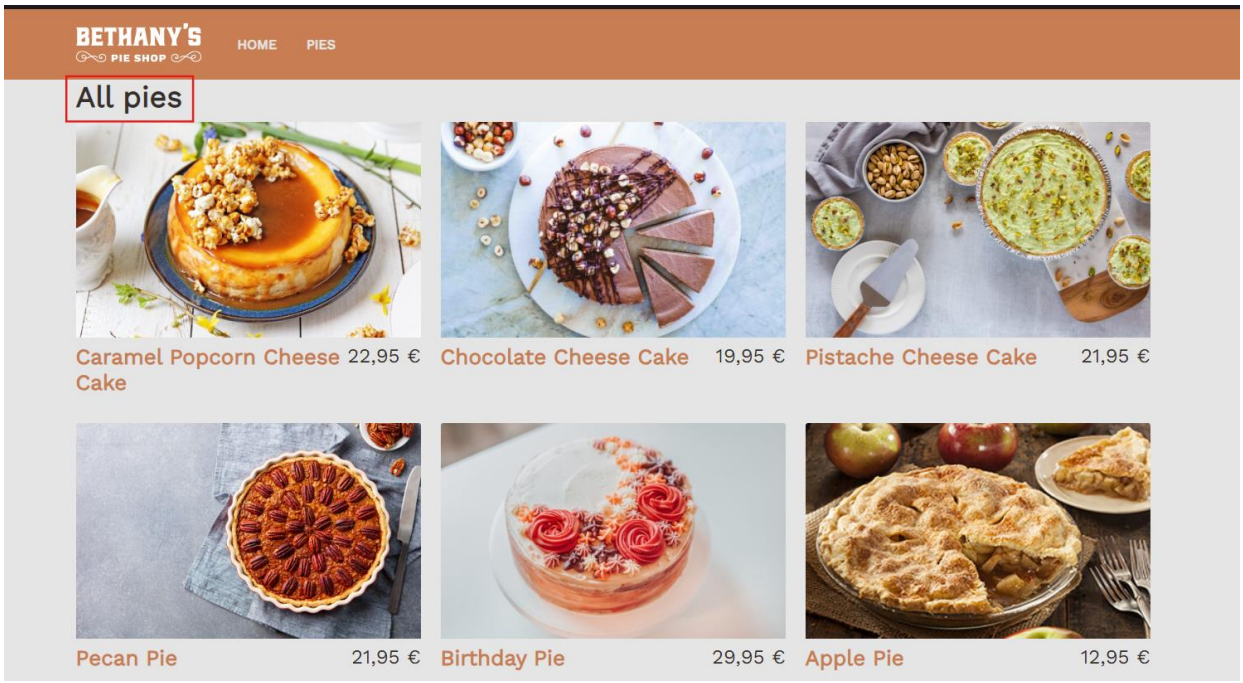
Torniamo su PieController.cs e cambiamo la stringa precendente in "All pies":




Avviamo il programma:




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

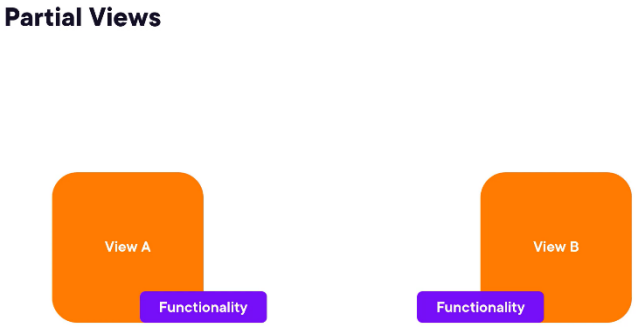



Notiamo la modifica applicata di "All pies"

# Improving the Views in the Application

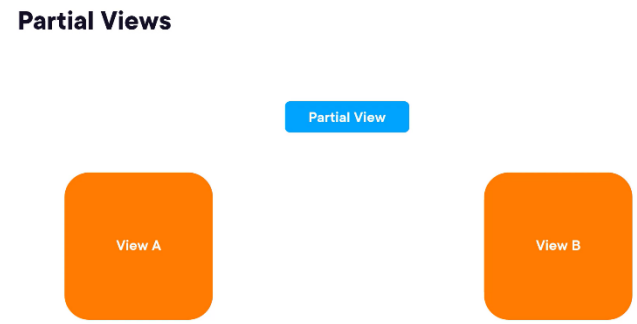
- Improving the Views in the Application
- Using Partial Views
- Demo: Adding a Partial View
- Creating the Shopping Cart
- Demo: Creating the Shopping Cart
- Working with View Components
- Demo: Creating View Components
- Creating a Custom Tag Helper
- Demo: Creating the Email Tag Helper

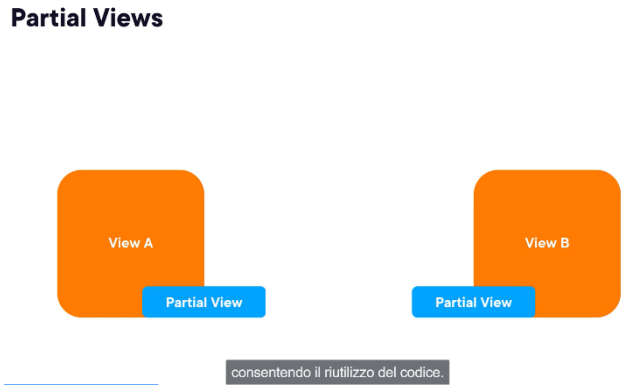
Immagina di avere due visualizzazioni:




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:



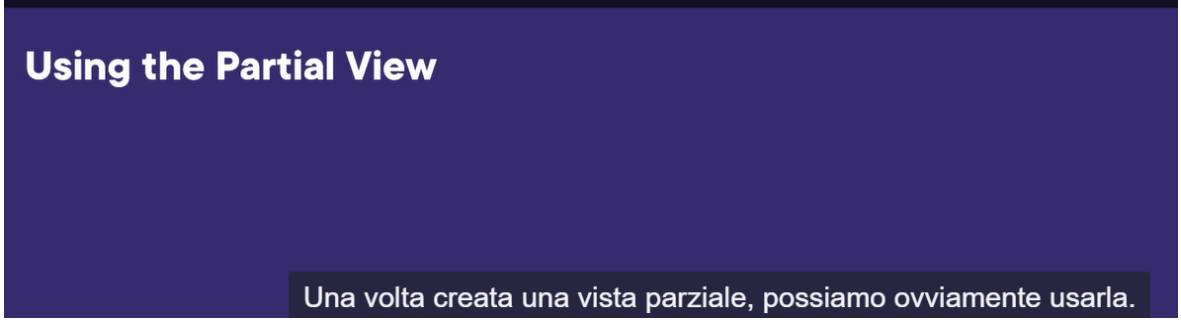



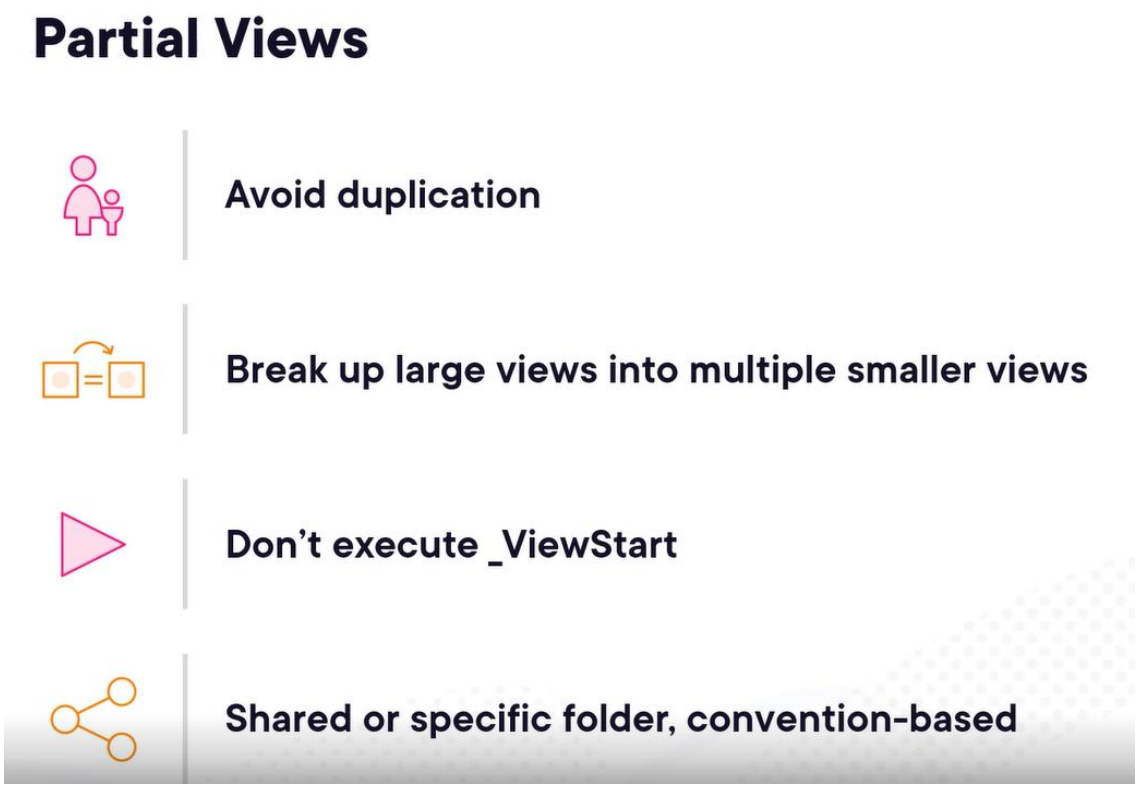


Osservazione: una partial view inizia con un underscore.

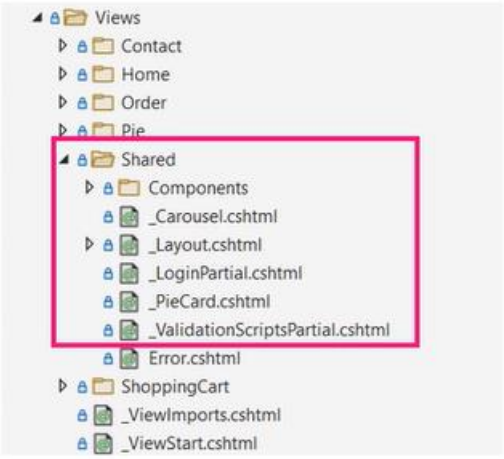
Come utilizzarla:

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




Esempio:

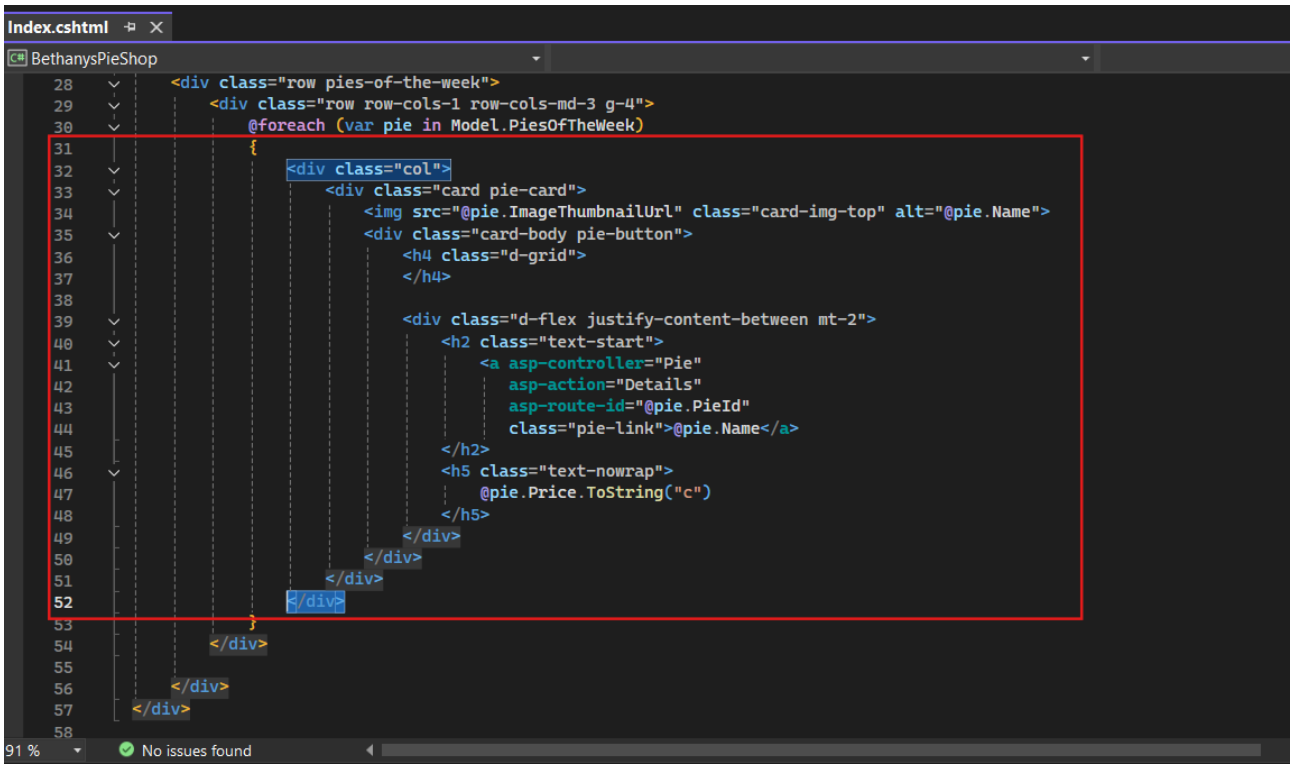



Osservazione: Posizionate in modo predefinito in tale directory

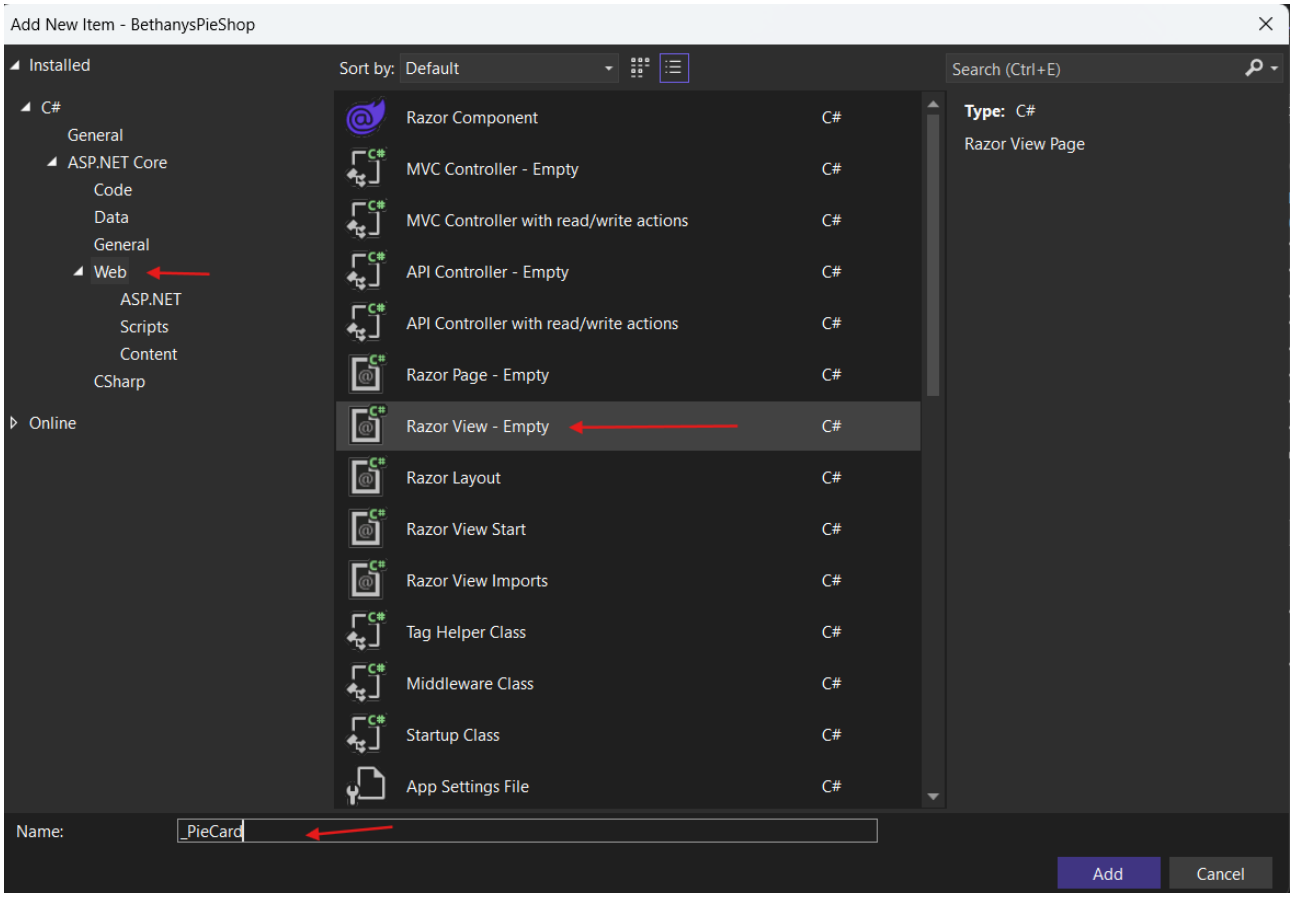
**DEMO: Adding aPartial View**

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 si che il tutto venga gestito da una partial view.

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



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

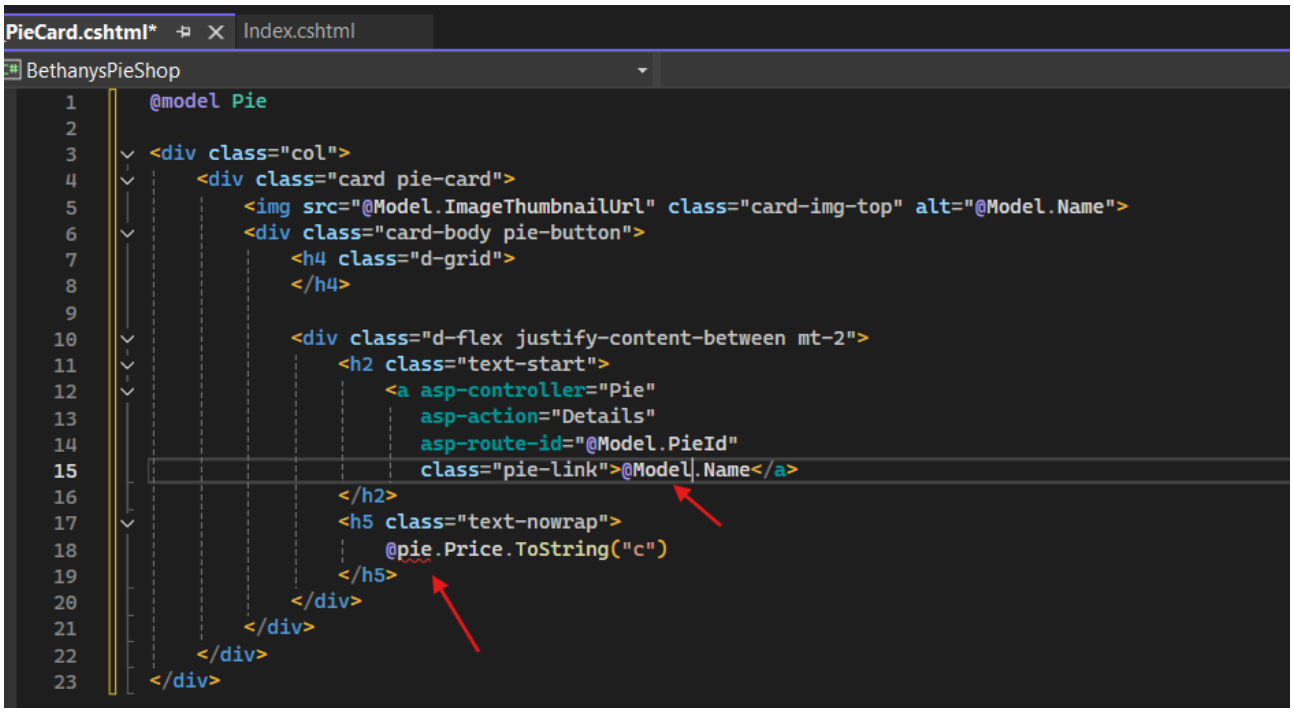



Tagliamo il codice di Index.html evidenziato precedentemente (per essere precisi da riga 32 a 52):

```
32              <div class="col">
33                  <div class="card pie-card">
34                      <img src="@pie.ImageThumbnailUrl" class="card-img-top" alt="@pie.Name">
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="text-nowrap">
47                                  @pie.Price.ToString("c")
48                              </h5>
49                          </div>
50                      </div>
51                  </div>
52              </div>
```

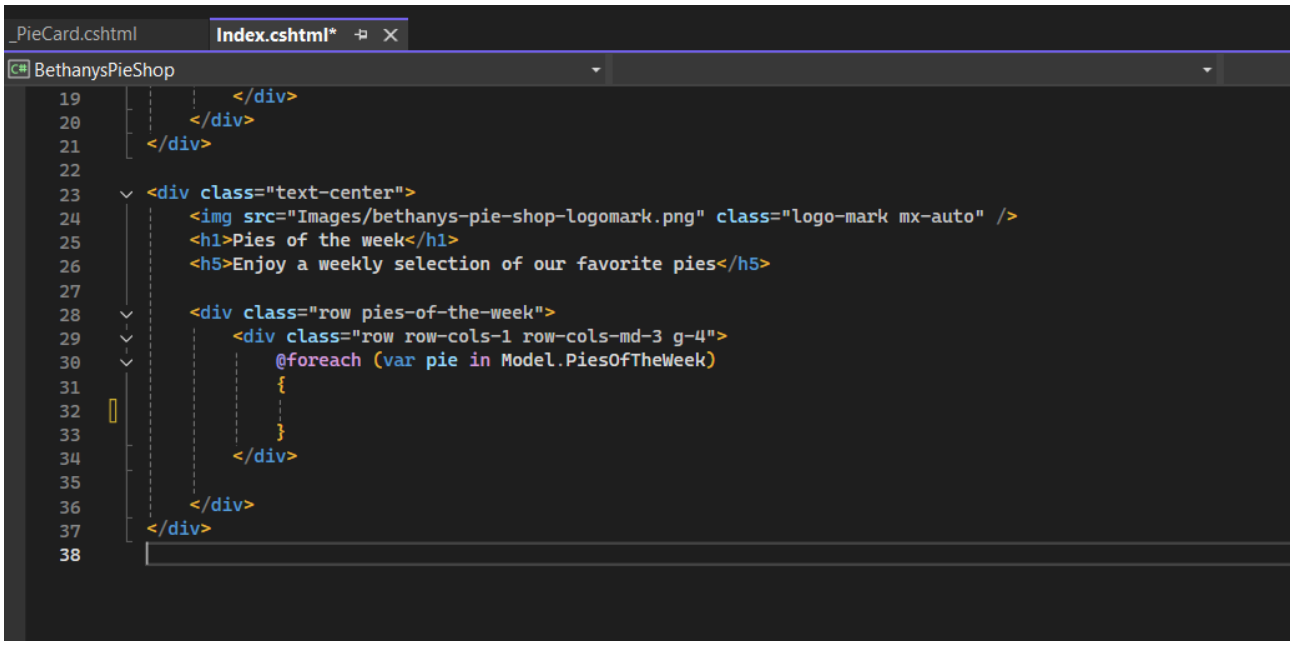
Torniamo in _PieCard.cshtml, inseriamo *@model Pie,* incolliamo il codice copiato e modifichiamo tutti i

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




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

(PRIMA)




(DOPO)

```
<div class="text-center">
    <img src="Images/bethanys-pie-shop-logomark.png" class="logo-mark mx-auto" />
    <h1>Pies of the week</h1>
    <h5>Enjoy a weekly selection of our favorite pies</h5>

    <div class="row pies-of-the-week">
        <div class="row row-cols-1 row-cols-md-3 g-4">
            @foreach (var pie in Model.PiesOfTheWeek)
            {
                <partial name="_PieCard" model="pie"/>
            }
        </div>

    </div>
</div>
```

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 Vies>Pie>List.cshtml:

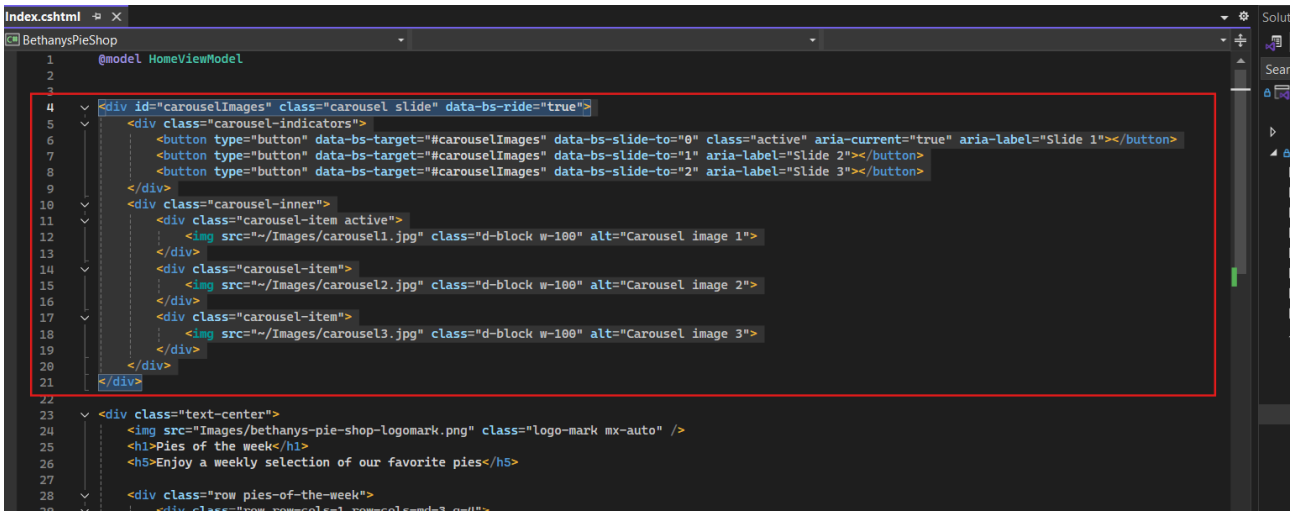
List.cshtml* | _PieCard.cshtml | Index.cshtml*

BethanysPieShop

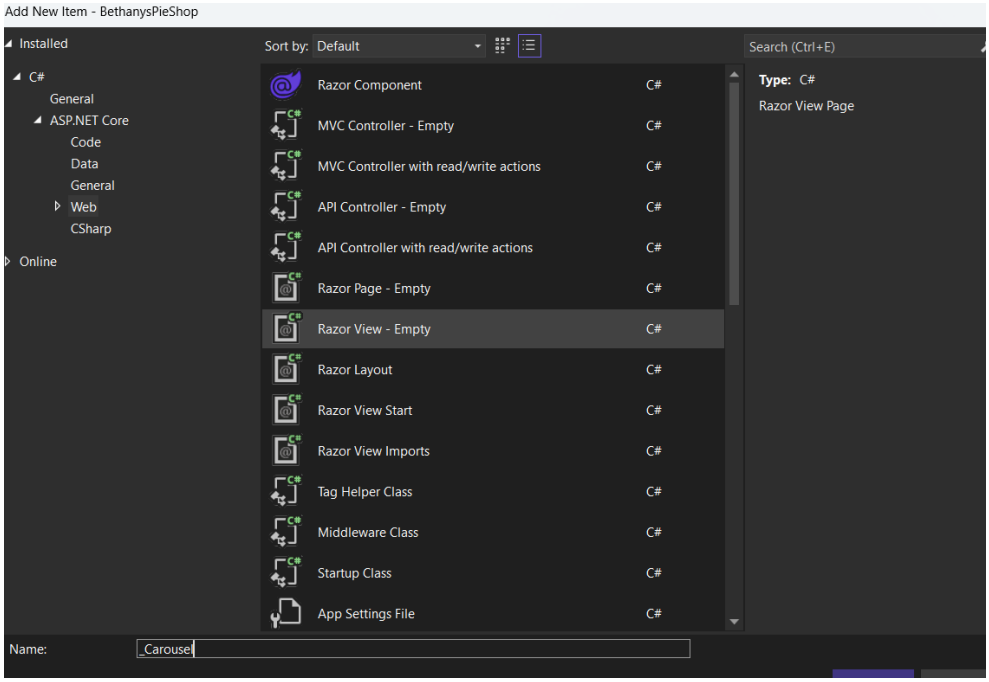
```
@model PieListViewModel

<h1>@Model.CurrentCategory</h1>

<div class="row row-cols-1 row-cols-md-3 g-4">

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

</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.chstml

Torniamo ora all'Index.cshtml:




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:

```
Carousel.cshtml  Index.cshtml*
BethanysPieShop

<div id="carouselImages" class="carousel slide" data-bs-ride="true">
    <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">
            <img src="~/Images/carousel1.jpg" class="d-block w-100" alt="Carousel image 1">
        </div>
        <div class="carousel-item">
            <img src="~/Images/carousel2.jpg" class="d-block w-100" alt="Carousel image 2">
        </div>
        <div class="carousel-item">
            <img src="~/Images/carousel3.jpg" class="d-block w-100" alt="Carousel image 3">
        </div>
    </div>
</div>
```

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

```
_Carousel.cshtml   Index.cshtml*
BethanysPieShop

@model HomeViewModel


<partial name="_Carousel" />

<div class="text-center">
    <img src="Images/bethanys-pie-shop-logomark.png" class="logo-mark mx-auto" />
    <h1>Pies of the week</h1>
    <h5>Enjoy a weekly selection of our favorite pies</h5>

    <div class="row pies-of-the-week">
        <div class="row row-cols-1 row-cols-md-3 g-4">
            @foreach (var pie in Model.PiesOfTheWeek)
            {
                <partial name="_PieCard" model="pie"/>
            }
        </div>

    </div>
</div>
```

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

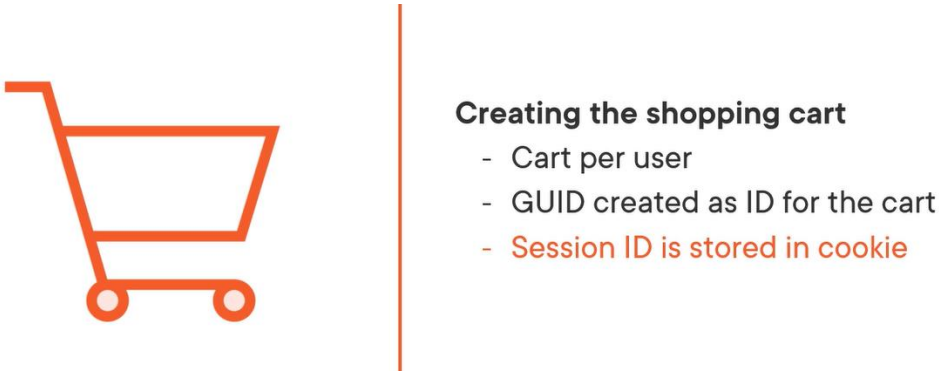
Ora il nostro Index.cshtml è più leggibile e compatto.

**Creating the Shopping Cart**

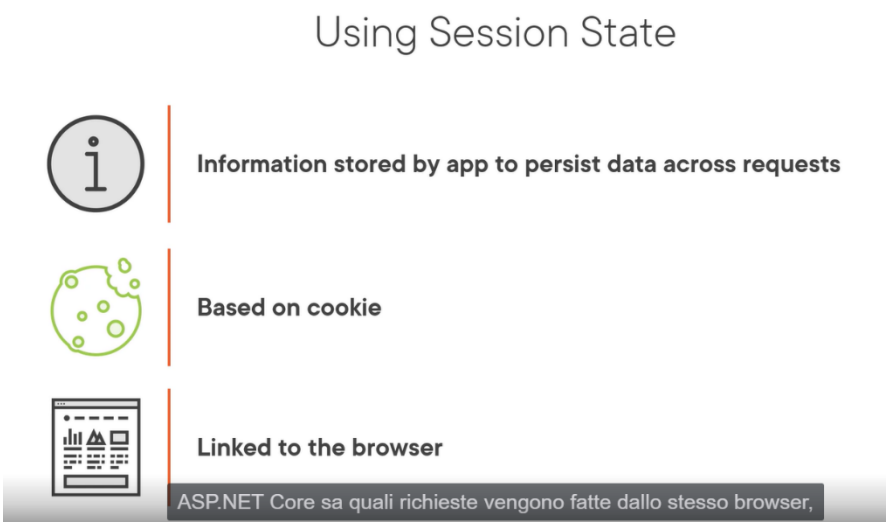
Il nostro website è **stateless**, ovvero ogni volta che una risposta è stata inviata al client, il server dimentica tutto sulla richiesta, quindi tra una richiesta e l'altra, di default, nessuna informazione è conservata sul server

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

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




*Cosa è una session state in ASP.NET?*
Permette di poter conservare sul server dati tra una richiesta e l'altra



```
builder.Services.AddSession();

...

app.UseSession();
```

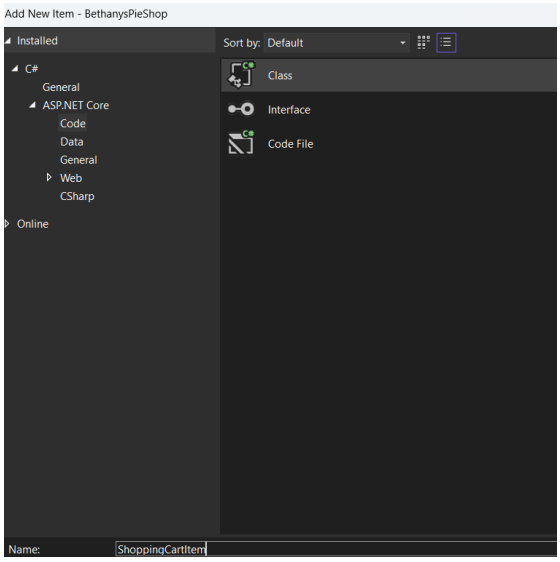
Adding Session Support in the App

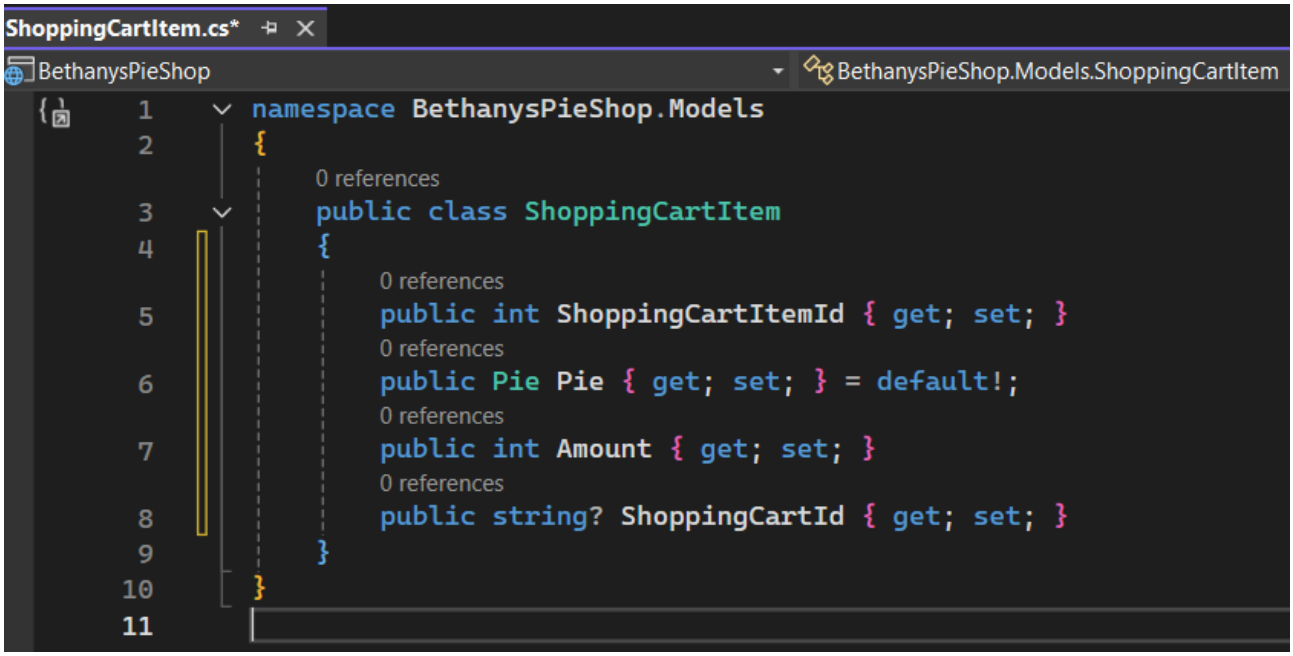
Osservazione: Bisogna quindi registrare un servizio framework nella raccolta Services

**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




**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.

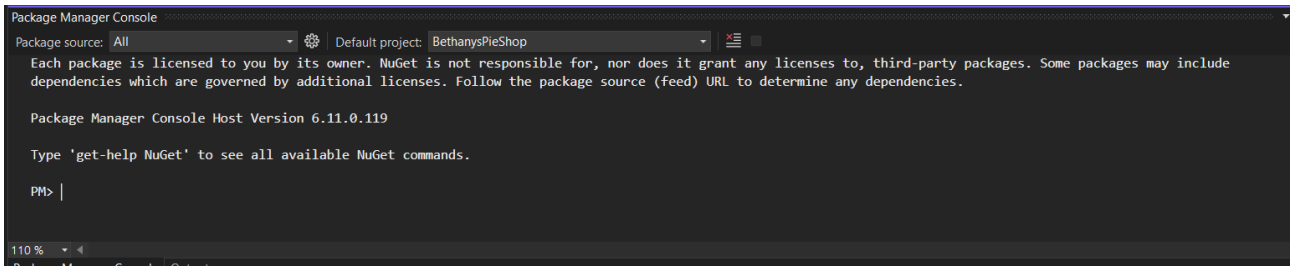
## `?` (Nullable Reference Types):

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.

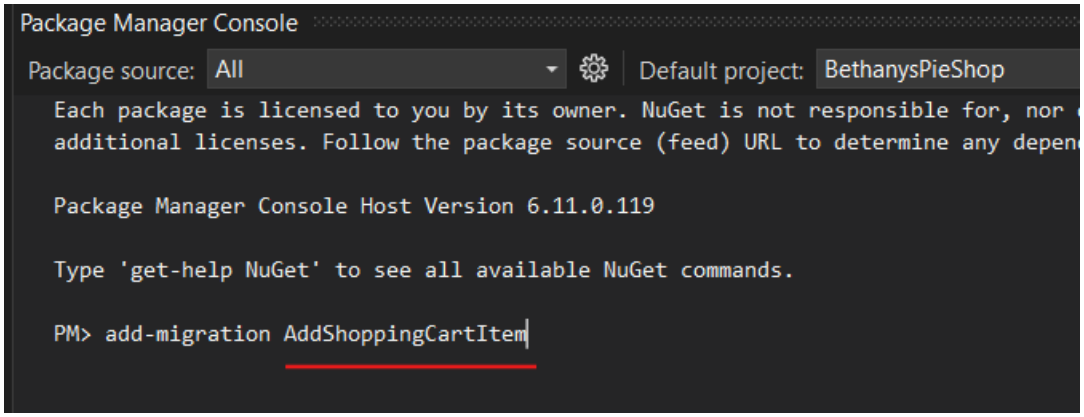
Detto ciò, giungiamo nel nostro **BethanysPieShopDbContext.cs** presente in Models e aggiungiamo la seguente proprietà:

```
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    11 references
    public class BethanysPieShopDbContext: DbContext
    {
        0 references
        public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
        {
        }

        3 references
        public DbSet<Category> Categories { get; set; }
        4 references
        public DbSet<Pie> Pies { get; set; }
        0 references
        public DbSet<ShoppingCartItem> ShoppingCartItems{ get; set; }
    }
}
```

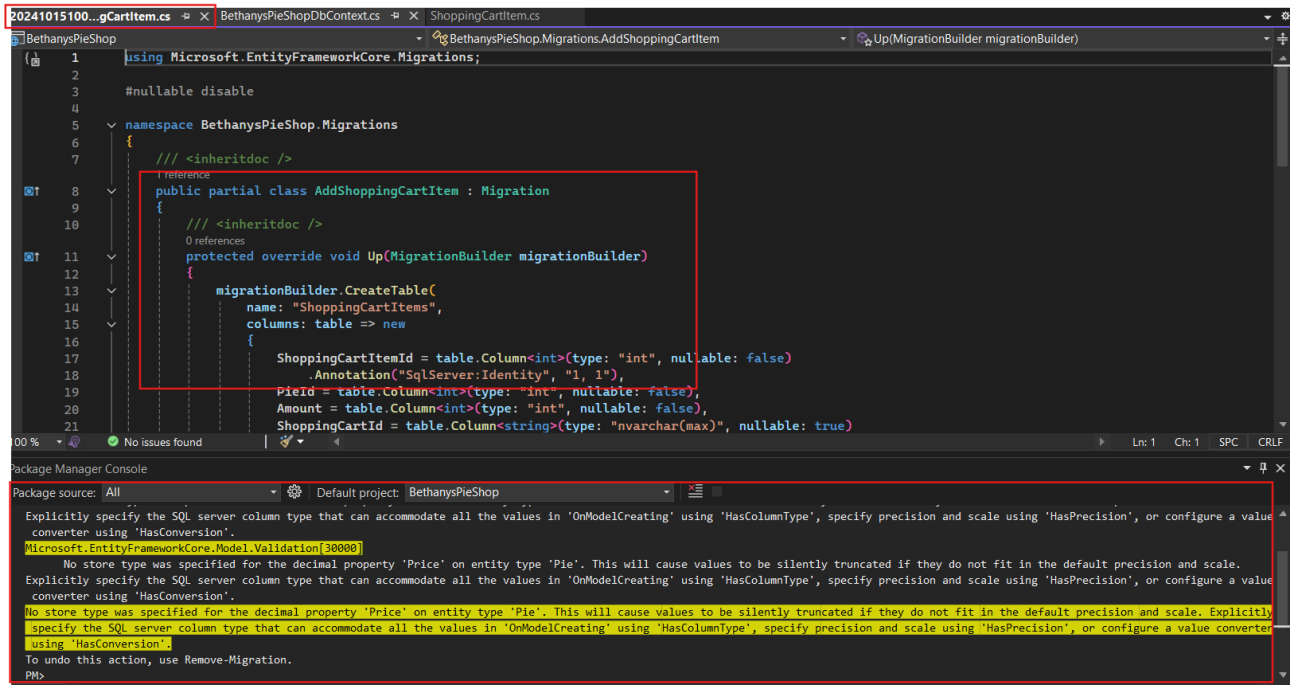
Ora dobbiamo fare la nostra migrazione per poter sincronizzare, quindi effettuiamo una **build dopo aver salvato i file modificati.**
**Apriamo il Project Manager da console:**




Digitiamo: *add-migration AddShoppingCartItem*:



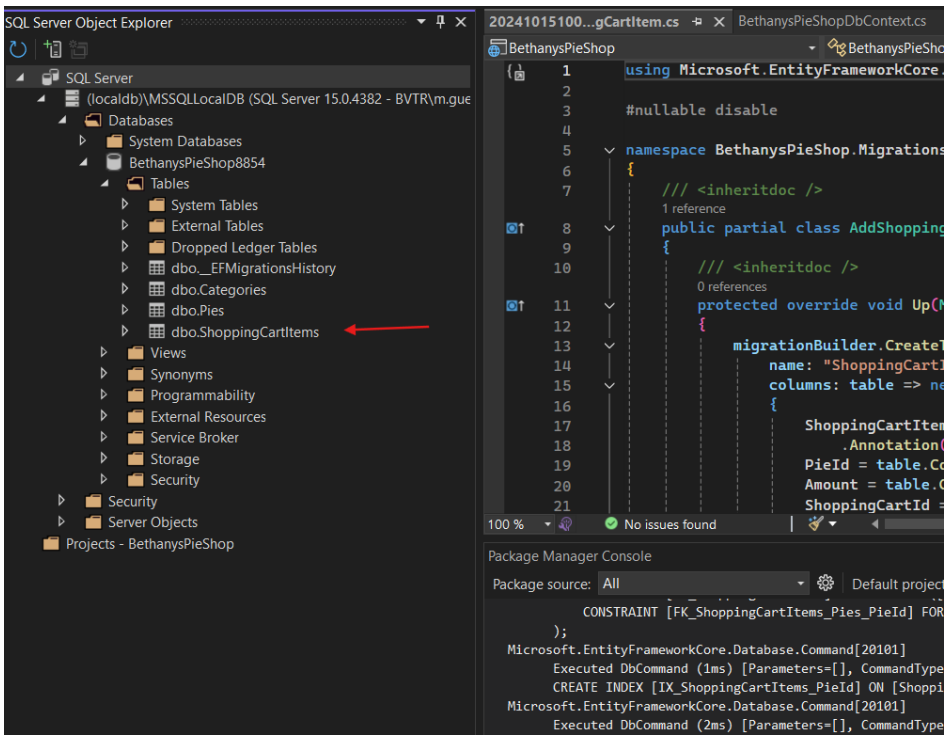

Output atteso:



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

*update-database*

Apriamo SQL Server Object Explorer:

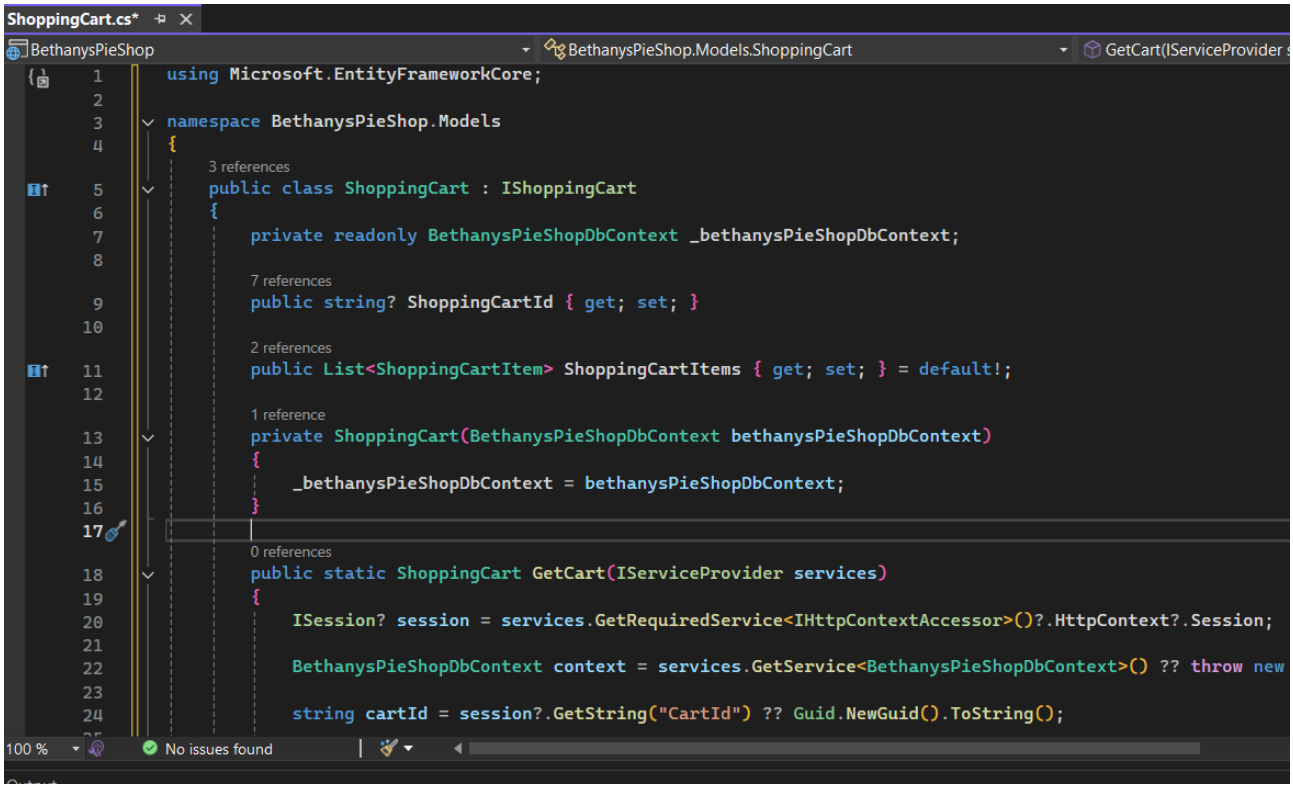



Notiamo che la tabella ShoppingCartItems è stata correttamente creata.
Osservazione: Non è stato generato seed data, in quanto verrà popolato durante l'esecuzione dell'applicazione.

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

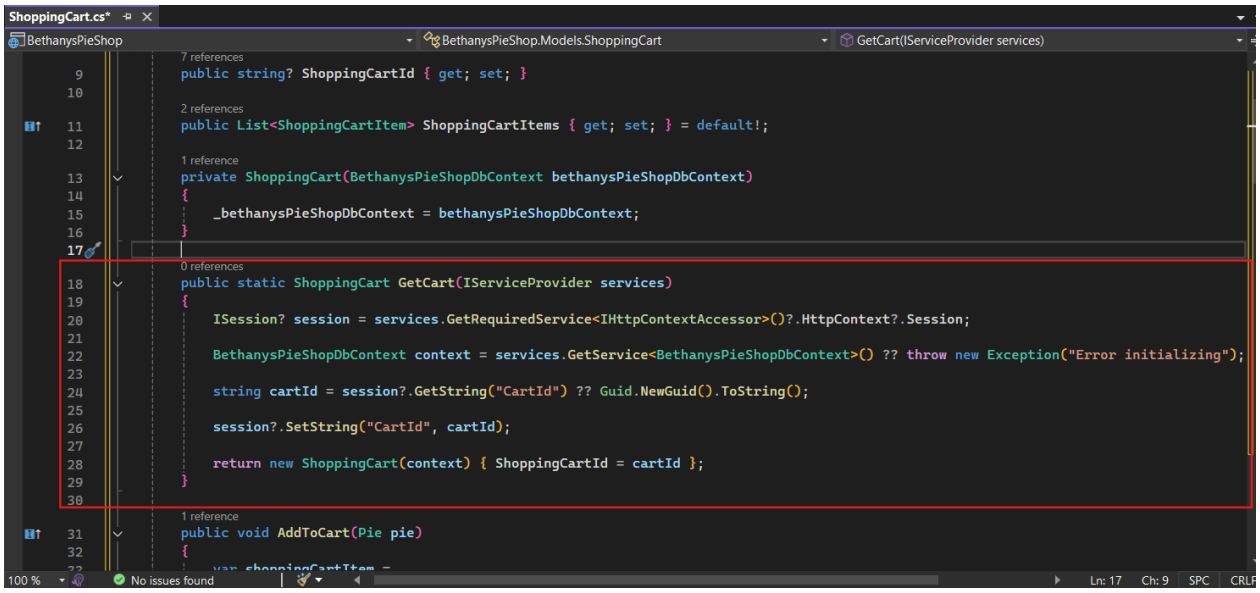
```
namespace BethanysPieShop.Models
{
    1 reference
    public interface IShoppingCart
    {
        1 reference
        void AddToCart(Pie pie);
        1 reference
        int RemoveFromCart(Pie pie);
        1 reference
        List<ShoppingCartItem> GetShoppingCartItems();
        1 reference
        void ClearCart();
        1 reference
        decimal GetShoppingCartTotal();
        2 references
        List<ShoppingCartItem> ShoppingCartItems { get; set; }
    }
}
```

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




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

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




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 DbContex e successivamente verifichiamo in base alla sessione se esiste già il valore chiamato CartId per l'utente in arrivo.
ASP.NET Core utilizza un coockie per associare diverse richieste dello stesso utente, della stessa macchina.

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

```
Program.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
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 ShoopingCart per la richiesta, quindi tutti i luoghi all'interno della richiesta che hanno accesso a ShoppinCart 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:

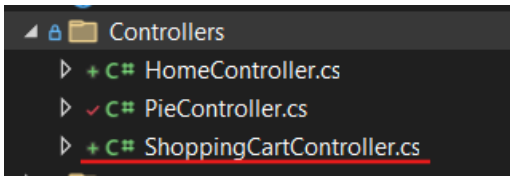
```
1 reference
public static ShoppingCart GetCart(IServiceProvider services)
{
    ISession? session = services.GetRequiredService<IHttpContextAccessor>()?.HttpContext?.Session;
```

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

```
Program.cs*
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: "defaulte",
35      //    pattern: "{controller=Home}/{action=Index}/{id?}");
36
```

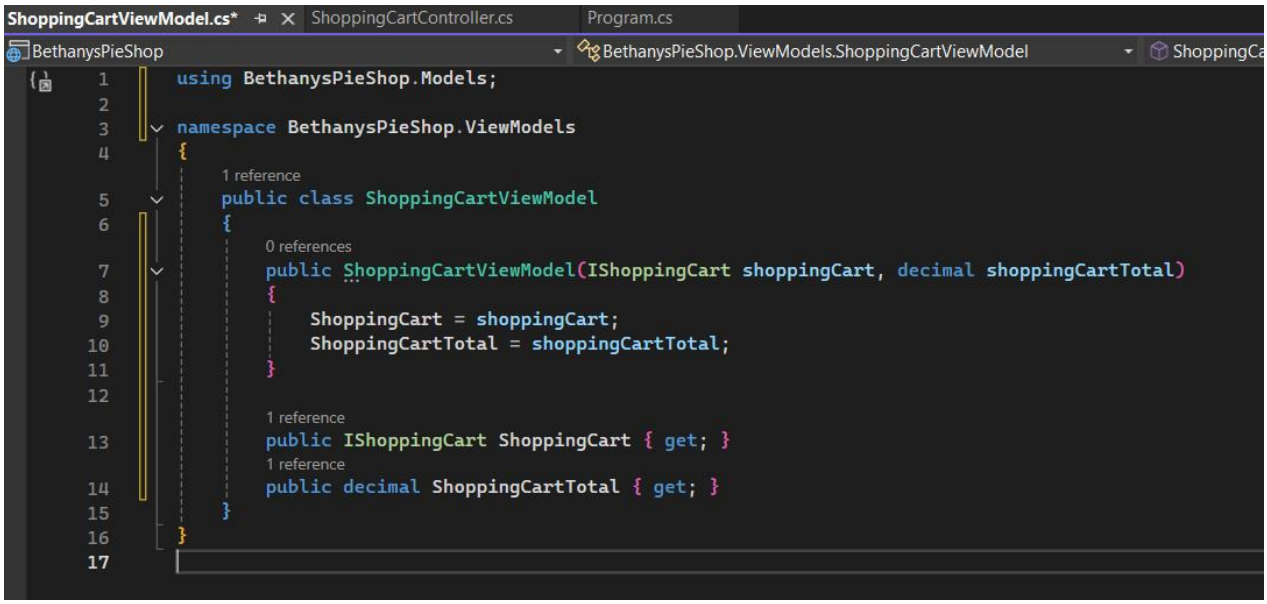
Salviamo e buildiamo e procediamo come sempre nlla creazione di un controller e le viste necessarie.

Creiamo quindi ShoppingCartController.cs

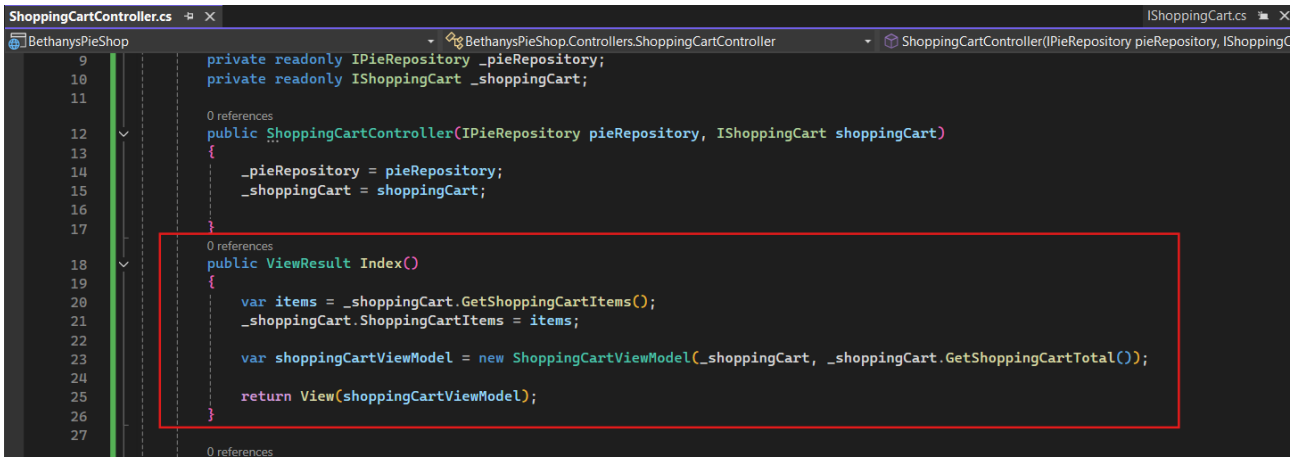



```
ShoppingCartController.cs   Program.cs
BethanysPieShop   BethanysPieShop.Controllers.ShoppingCartController   ShoppingCartCont
1   using BethanysPieShop.Models;
2   using Microsoft.AspNetCore.Mvc;
3
4   namespace BethanysPieShop.Controllers
5   {
        1 reference
6       public class ShoppingCartController: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

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




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

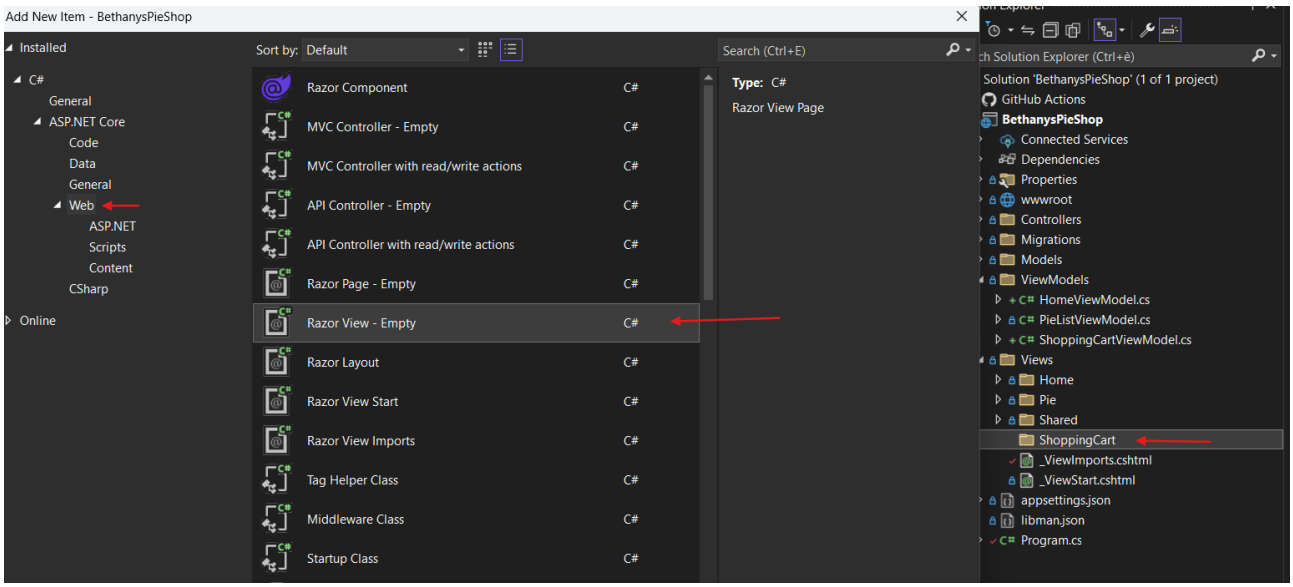



Osservazione: Chiederò al cartello tutti gli items, dopo creo un nuovo ShoppingCartViewModel, passandogli gli articoli oltre al totale ed infine ritorno la vista.

```
27
                0 references
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
                0 references
39              public RedirectToActionResult RemoveFromShoppingCart(int pieId)
40              {
41                  var selectedPie = _pieRepository.AllPies.FirstOrDefault(p => p.PieId == pieId);
42
43                  if (selectedPie != null)
44                  {
45                      _shoppingCart.RemoveFromCart(selectedPie);
46                  }
47                  return RedirectToAction("Index");
48              }
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:

Index.cshtml* ShoppingCartController.cs

BethanysPieShop

```
@model ShoppingCartViewModel

<h3 class="my-5">
    Shopping cart
</h3>


<div class="row gx-3">
    <div class="col-8">
        @foreach (var line in Model.ShoppingCart.ShoppingCartItems)
        {
            <div class="card shopping-cart-card mb-2">
                <div class="row">
                    <div class="col-md-4">
                        <img src="@line.Pie.ImageThumbnailUrl" class="img-fluid rounded-start p-2" alt="@line.Pie.Name">
                    </div>
                    <div class="col-md-8">
                        <div class="card-body">
                            <h5 class="card-text">@line.Amount x @line.Pie.Name</h5>
                            <div class="d-flex justify-content-between">
                                <h6>@line.Pie.ShortDescription</h6>
                                <h2>@line.Pie.Price.ToString("c")</h2>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        }
    </div>
    <div class="col-4">
        <div class="card shopping-cart-card p-3">
            <div class="row">
                <h4 class="col">Total:</h4>
                <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
            </div>
            <hr />
            <div class="text-center d-grid">
            </div>
        </div>
    </div>
</div>
```

Osservazione:

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

```
        }
    </div>
    <div class="col-4">
        <div class="card shopping-cart-card p-3">
            <div class="row">
                <h4 class="col">Total:</h4>
                <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
            </div>
            <hr />
            <div class="text-center d-grid">
            </div>
        </div>
    </div>
</div>
```

No issues found

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

_PieCard.cshtml*

BethanysPieShop

```
@model Pie

<div class="col">
    <div class="card pie-card">
        <img src="@Model.ImageThumbnailUrl" class="card-img-top" alt="@Model.Name">
        <div class="card-body pie-button">
            <h4 class="d-grid">
                <a class="btn btn-secondary"
                   asp-controller="ShoppingCart"
                   asp-action="AddToShoppingCart"
                   asp-route-pieId="@Model.PieId"> + Add to Cart</a>
            </h4>

            <div class="d-flex justify-content-between mt-2">
                <h2 class="text-start">
                    <a asp-controller="Pie"
                       asp-action="Details"
                       asp-route-id="@Model.PieId"
                       class="pie-link">@Model.Name</a>
                </h2>
                <h5 class="text-nowrap">
                    @Model.Price.ToString("c")
                </h5>
            </div>
        </div>
```

110 % No issues found

Adoperiamo in egual modo in Details.cshtml:

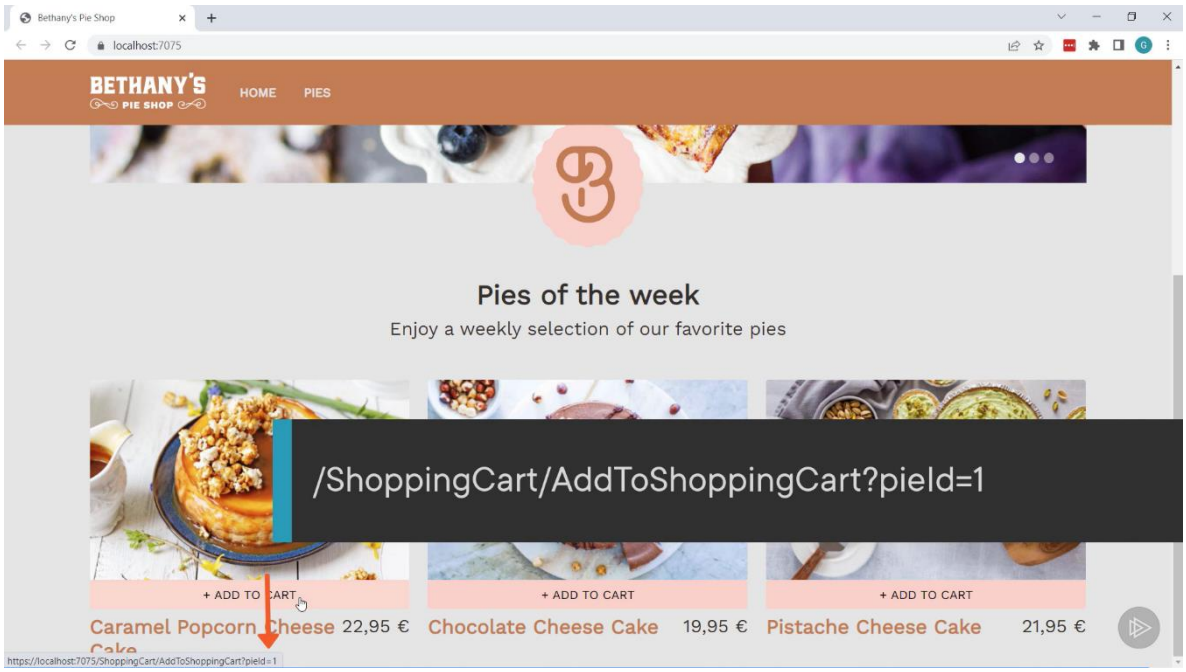
Details.cshtml*

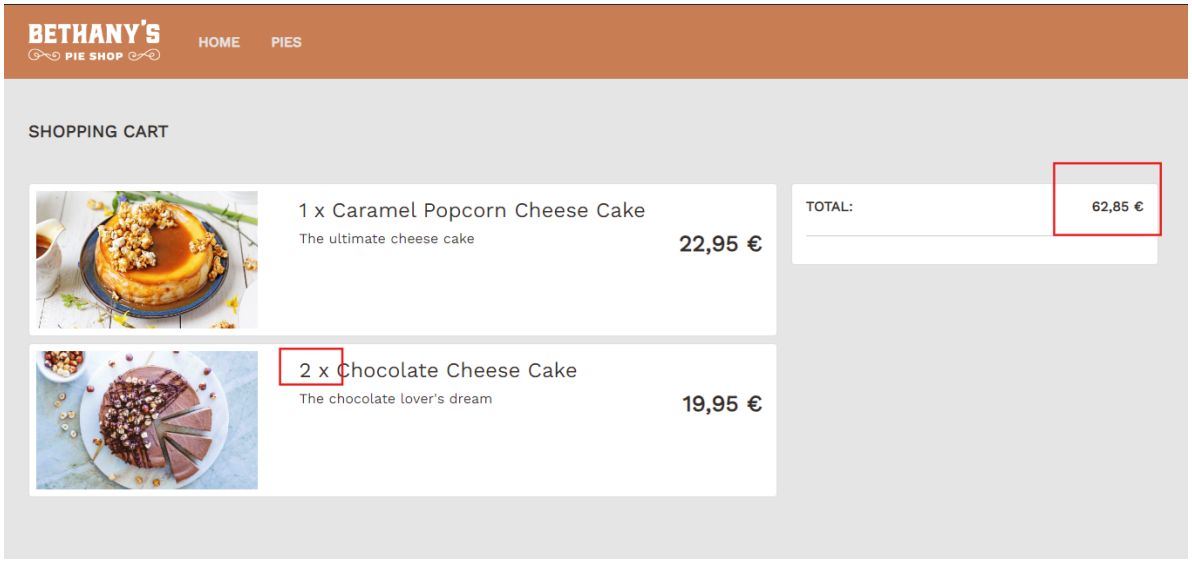
BethanysPieShop

```
@model Pie

<h3 class="my-5">
    @Model.Name
</h3>

<div class="row gx-5">
    <img alt="@Model.Name" src="@Model.ImageUrl" class="img-fluid col-5" />
    <div class="col-7">
        <h4>@Model.ShortDescription</h4>
        <p>@Model.LongDescription</p>
        <h3 class="pull-right">@Model.Price.ToString("c")</h3>
        <div class="addToCart">
            <p class="button">
                <a class="btn btn-secondary"
                   asp-controller="ShoppingCart"
                   asp-action="AddToShoppingCart"
                   asp-route-pieId="@Model.PieId">Add to Cart</a>"

            </p>
        </div>
    </div>
</div>
```

110 % No issues found

Salviamo ed eseguiamo l'applicazione.




**Working with View Components**

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

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

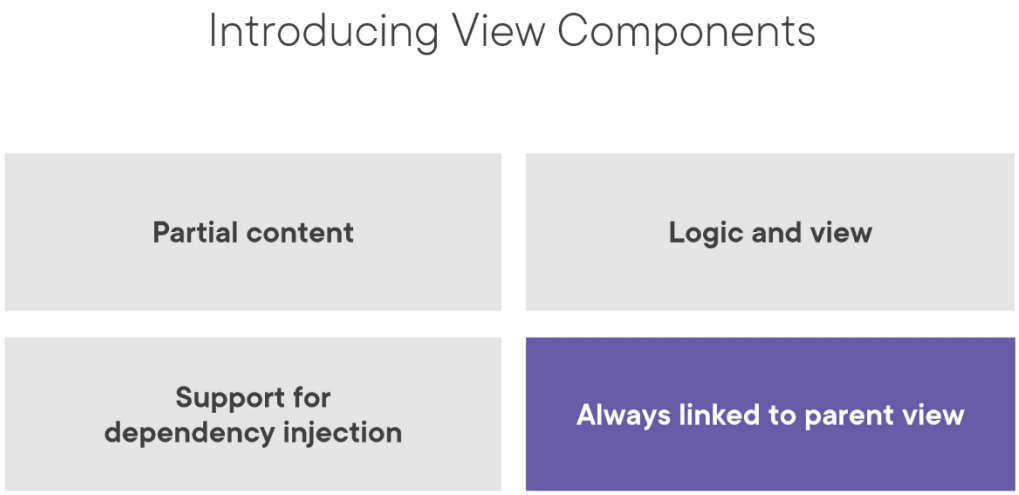
Partial View Limitations

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

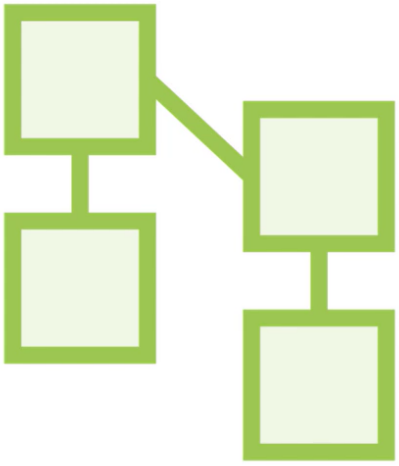
Osservazione: Le partial view funzionano con i dati che gli vengono passati dalla vista chiamante, nel nostro caso era l'elenco della panoramica della torta.

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

Il **View Component:**




**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'apporccio della classe base ViewComponent, quindi la prima opzione.

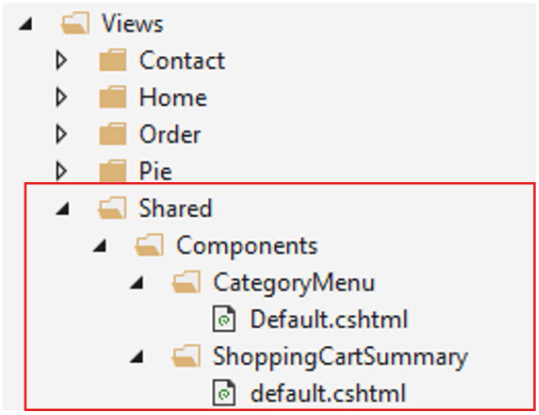
```
public class ShoppingCartSummary : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        return View(model);
    }
}
```

## 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 alterntiva un tag helper (ciò che faremo nel progetto):

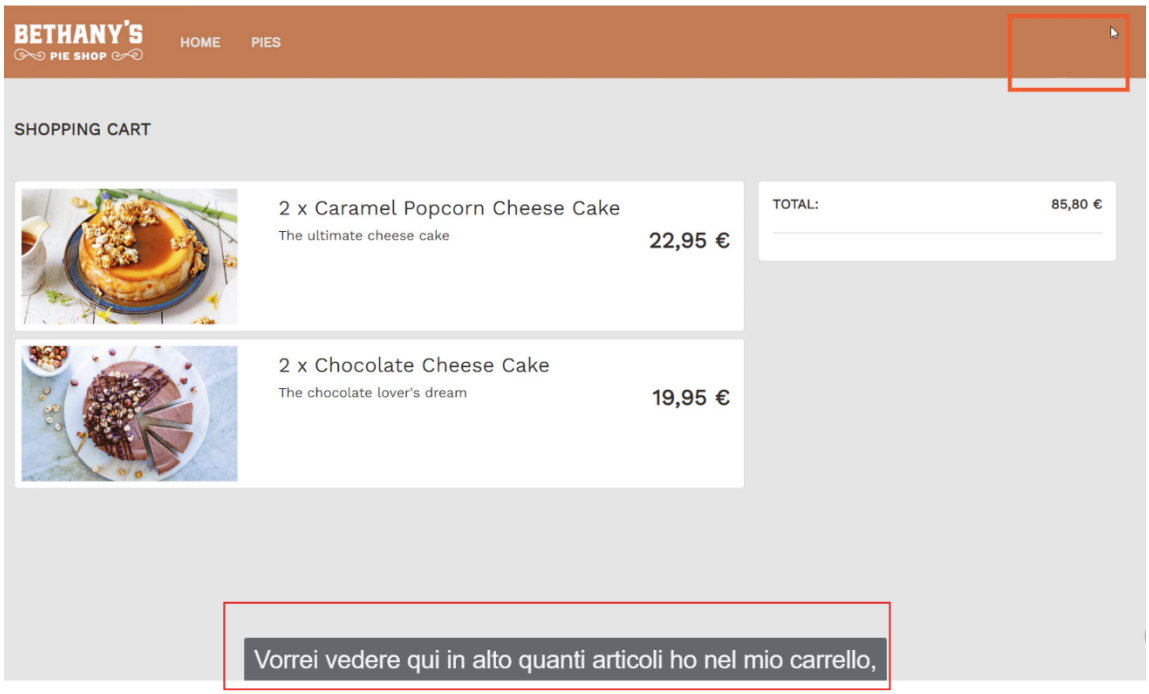
```
<vc:shopping-cart-summary></vc:shopping-cart-summary>
```

## Invoking a View Component with a Tag Helper

dobbiamo quindi specificare il nome del componente di visualizzazione.

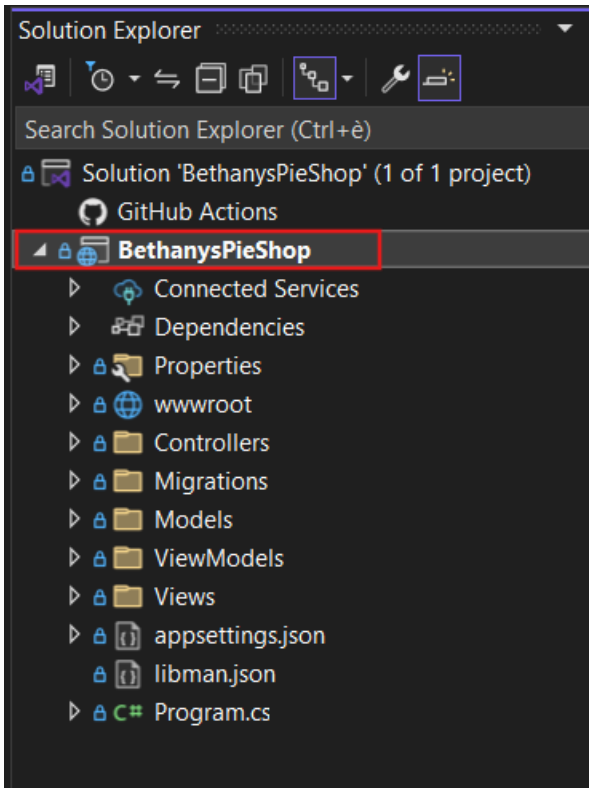
**Demo: Creating View Components**

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

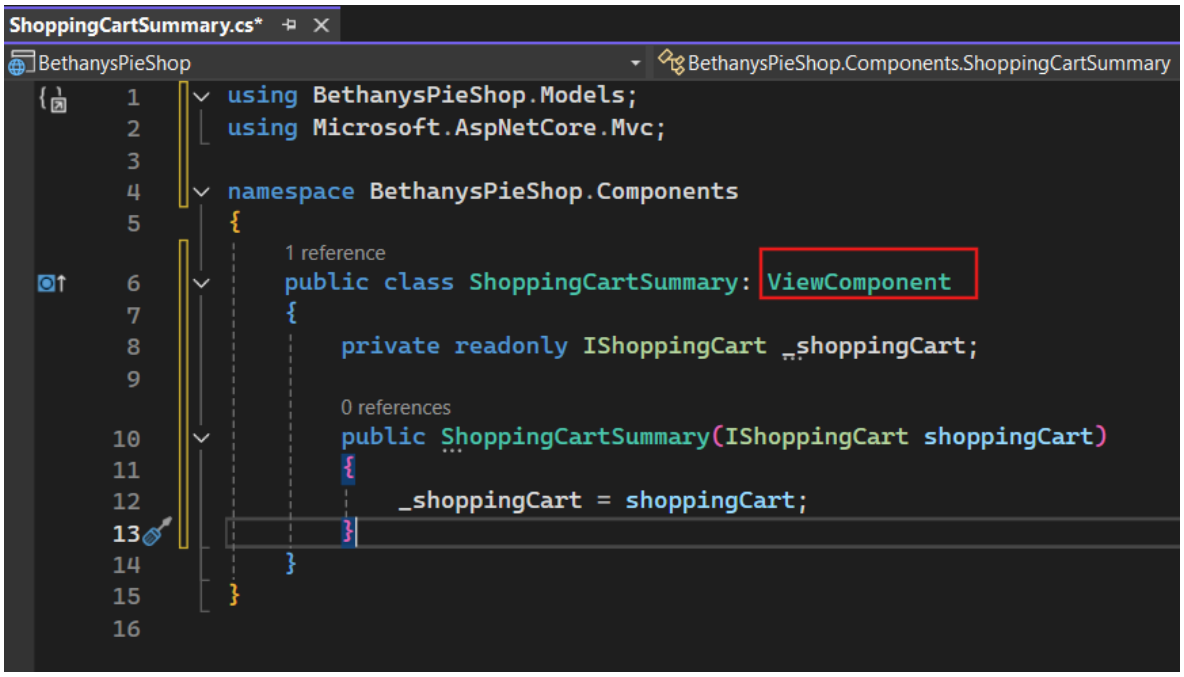



Come già detto una PartialView è più limitata rispetto a un ComponentView, in quanto dipende dalla vista padre (esempio _PieCard.cshtml)

**Creiamo la cartella Components all'interno del progetto BethanysPieShop**

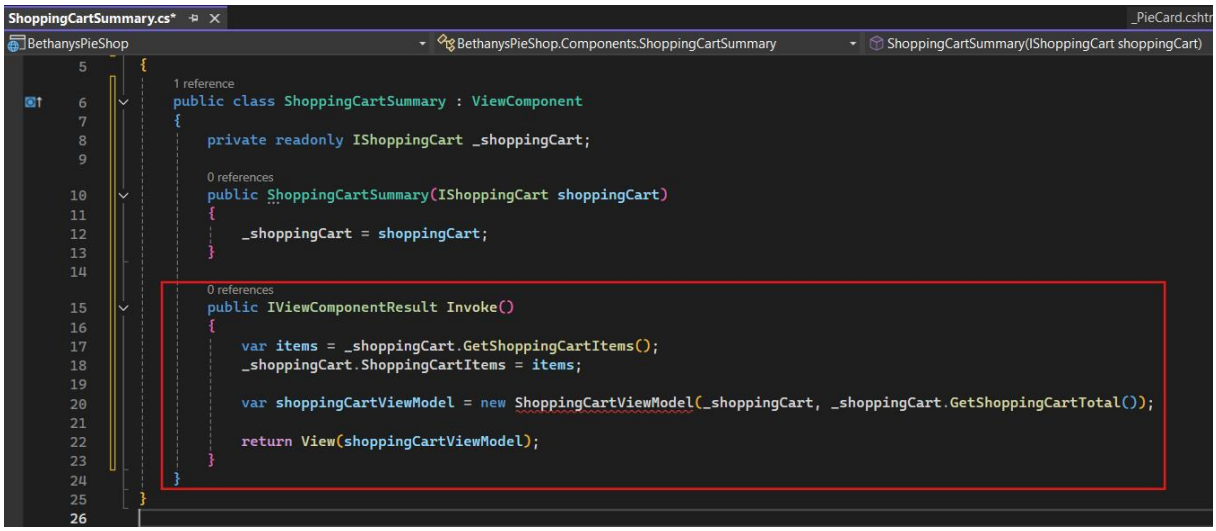


Al suo interno creiamo la classe ShoppingCartSummary.cs:



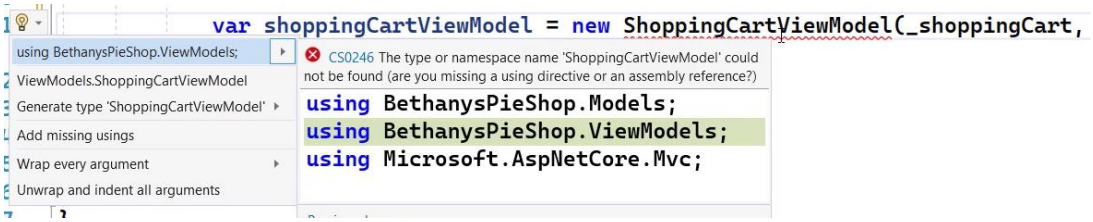

Ci serve IShoppingCart se no come potrei contare quanti articoli ci sono nel carrello?
Questa è una istanza-scope quindi non accederemo sempre al database

Aggiungiamo il metodo Invoke():

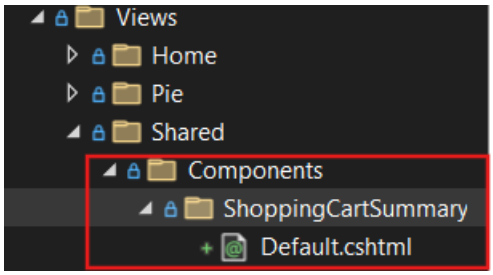



Osservazione: Simile a ciò che abbiamo fatto nel controller, però non specifichiamo un metodo di azione specifico

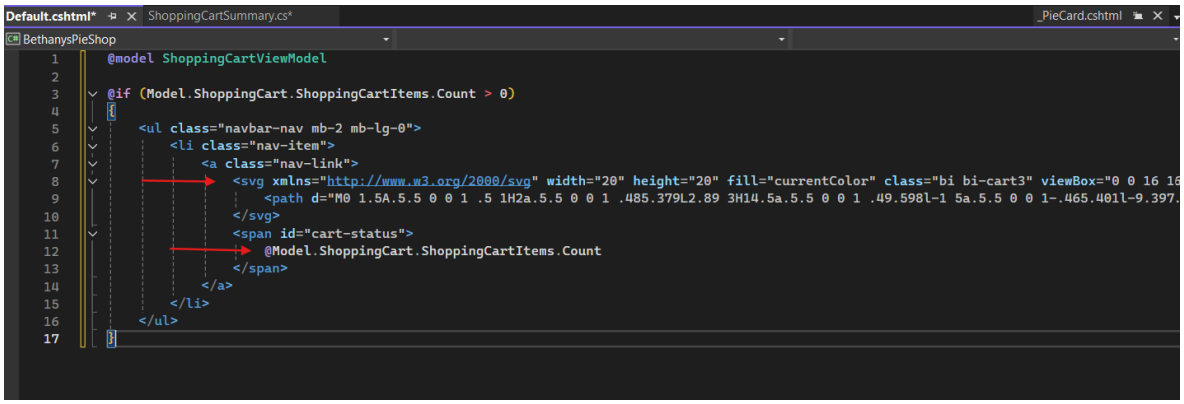
(L'errore lo risolviamo sempre con la lampadina:)




Fatto ciò, creiamo le seguenti cartelle e file:




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




**Osservazione:**

Riga 8: SVG per disegnare il carrello

Riga 12: mostriamo il conteggio degli articoli

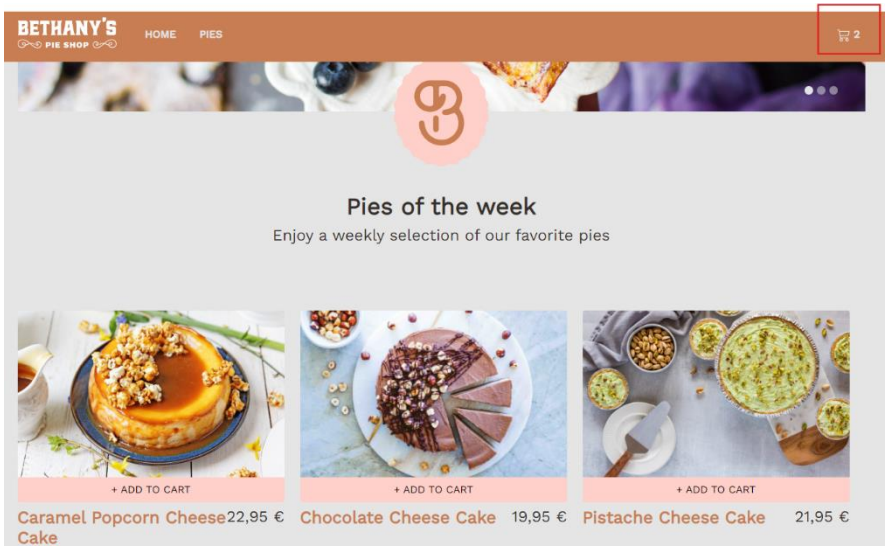
E' necessario ora aggiungere in **_ViewImports.cshtml**:

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

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

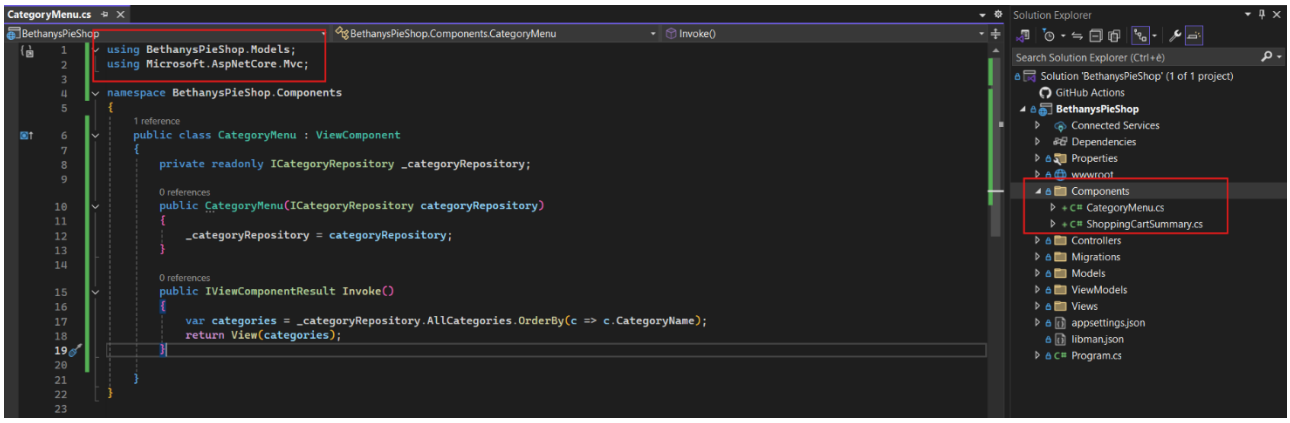
```
_Layout.cshtml*
BethanysPieShop
22                  </a>
23
24                  <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="
25                          aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigat
26                      <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                      <vc:shopping-cart-summary></vc:shopping-cart-summary>
39                  </div>
40              </div>
41          </nav>
42      </header>
43
44      @RenderBody()
45  </div>
46  </body>
47  </html>
48
```

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

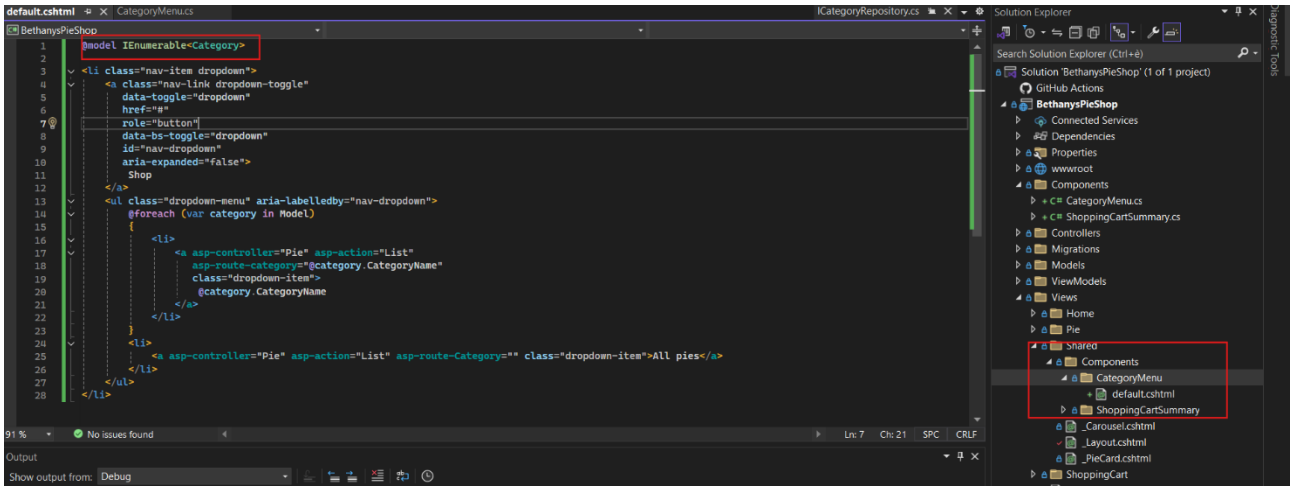


Ora aggiungiamo un altro component "SHOP" al posto di "HOME" e "PIES" per poter filtrare le varie categorie.

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 <li> 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:**

```
//public IActionResult List()
//{
//    //ViewBag.CurrentCategory = "Cheese cakes";

//    //return View(_pieRepository.AllPies);

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

0 references
public ViewResult List(string category)
{
    IEnumerable<Pie> pies;
    string? currentCategory;

    if (string.IsNullOrEmpty(category))
    {
        pies = _pieRepository.AllPies.OrderBy(p => p.PieId);
        currentCategory = "All pies";
    }
    else
    {
        pies = _pieRepository.AllPies.Where(p => p.Category.CategoryName == category)
            .OrderBy(p => p.PieId);
        currentCategory = _categoryRepository.AllCategories.FirstOrDefault(c => c.CategoryName == category)?.CategoryName;
    }

    return View(new PieListViewModel(pies, currentCategory));
}
```

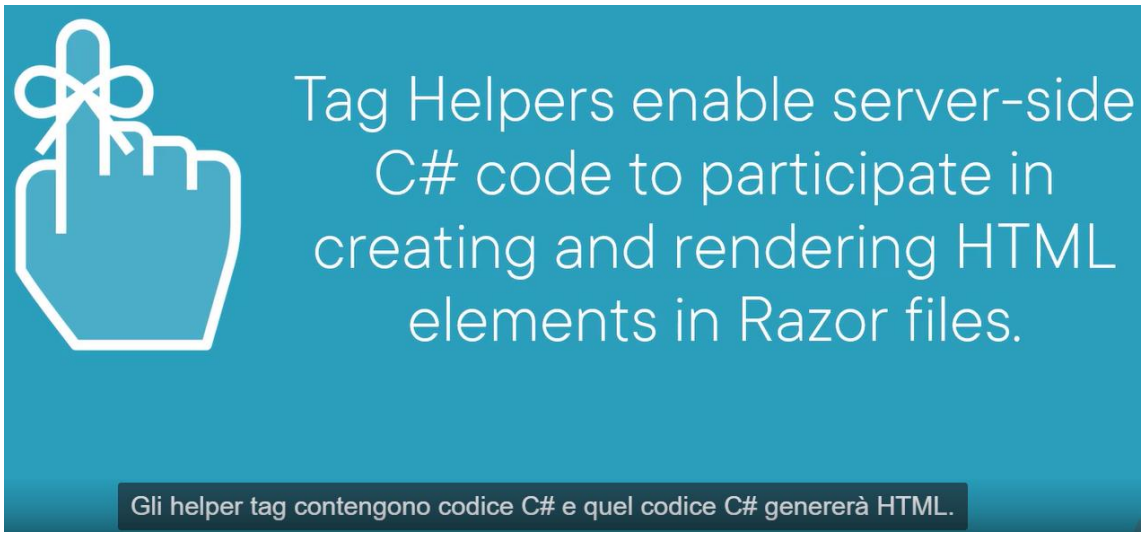
Abbiamo modificato anche _Layout.cshtml nel seguente modo:

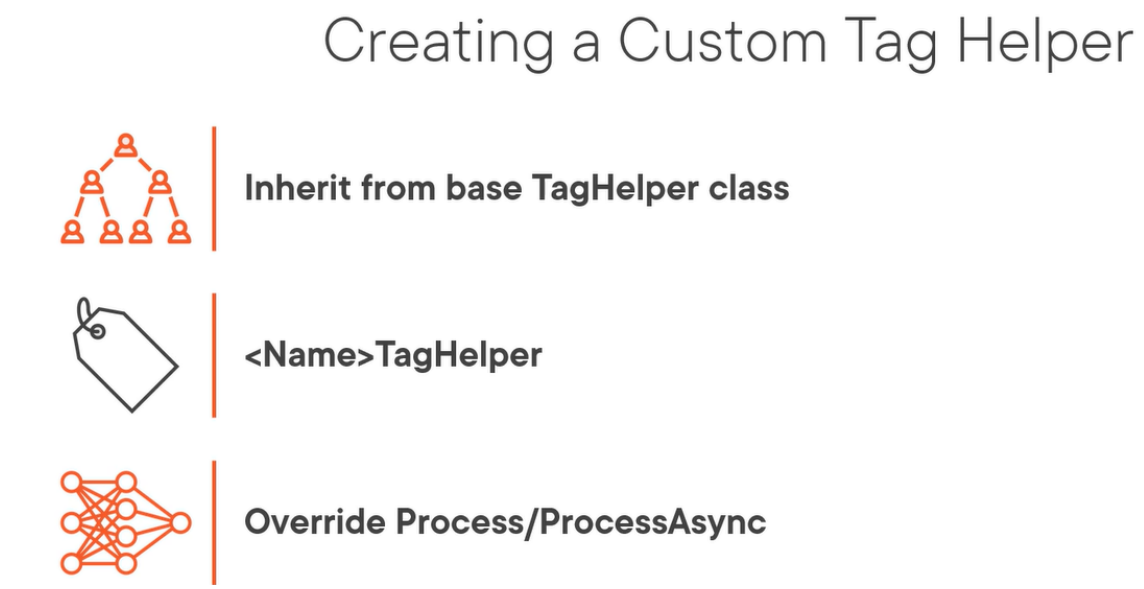
_Layout.cshtml*

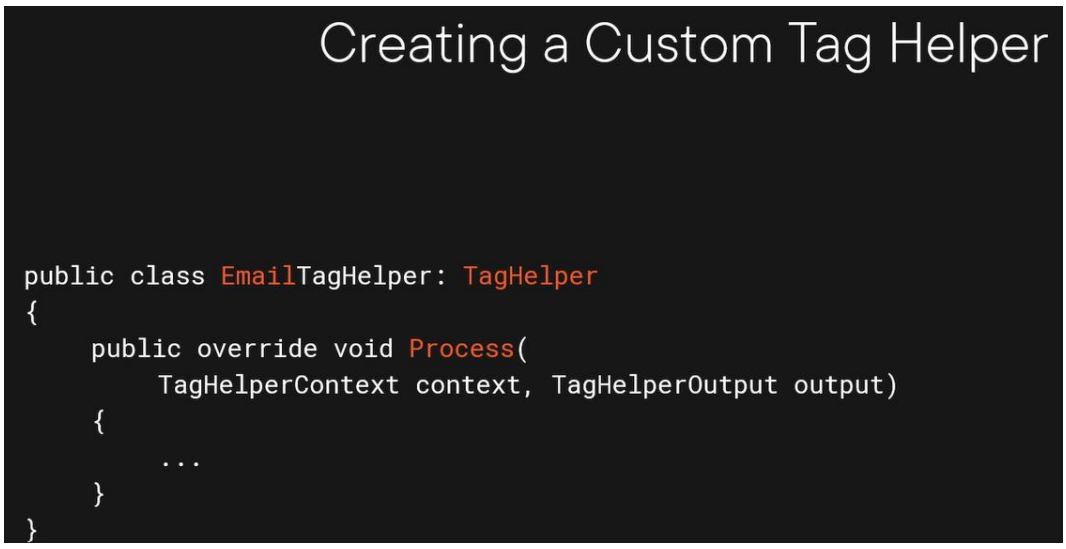
BethanysPieShop

```
<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">
                    <img src="images/bethanys-pie-shop-logo_horiz-white.png" width="151" height="47" class="d-inline-block align-top mb-2 mt-1"
                         alt="Bethany's Pie Shop Logo">
                </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">
                        <vc:category-menu></vc:category-menu>
                        <li class="nav-item">
                            <a asp-controller="Contact" asp-action="Index" class="nav-link">Contact</a>
                        </li>
                    </ul>
                    <vc:shopping-cart-summary></vc:shopping-cart-summary>
                </div>
            </div>
        </nav>
    </header>

    @RenderBody()
</div>
</body>
</html>
```

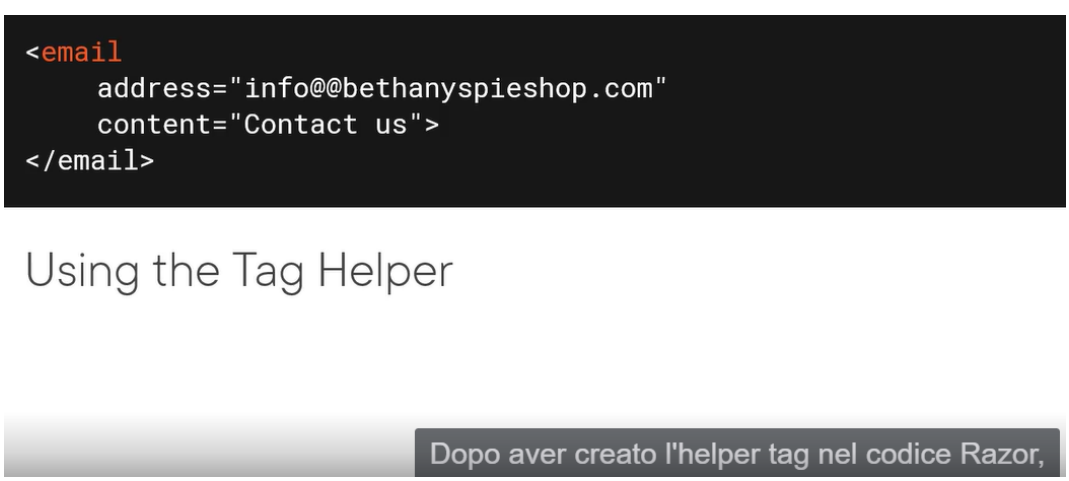
**Creating a Custom Tag Helper**

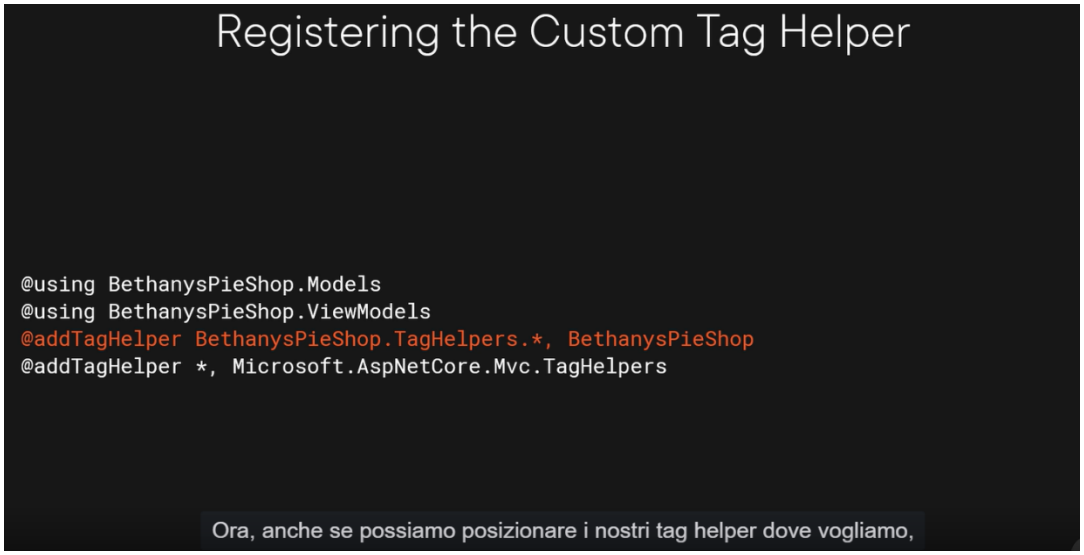

Tag Helpers enable server-side C# code to participate in creating and rendering HTML elements in Razor files.
Gli helper tag contengono codice C# e quel codice C# genererà HTML.


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>
```



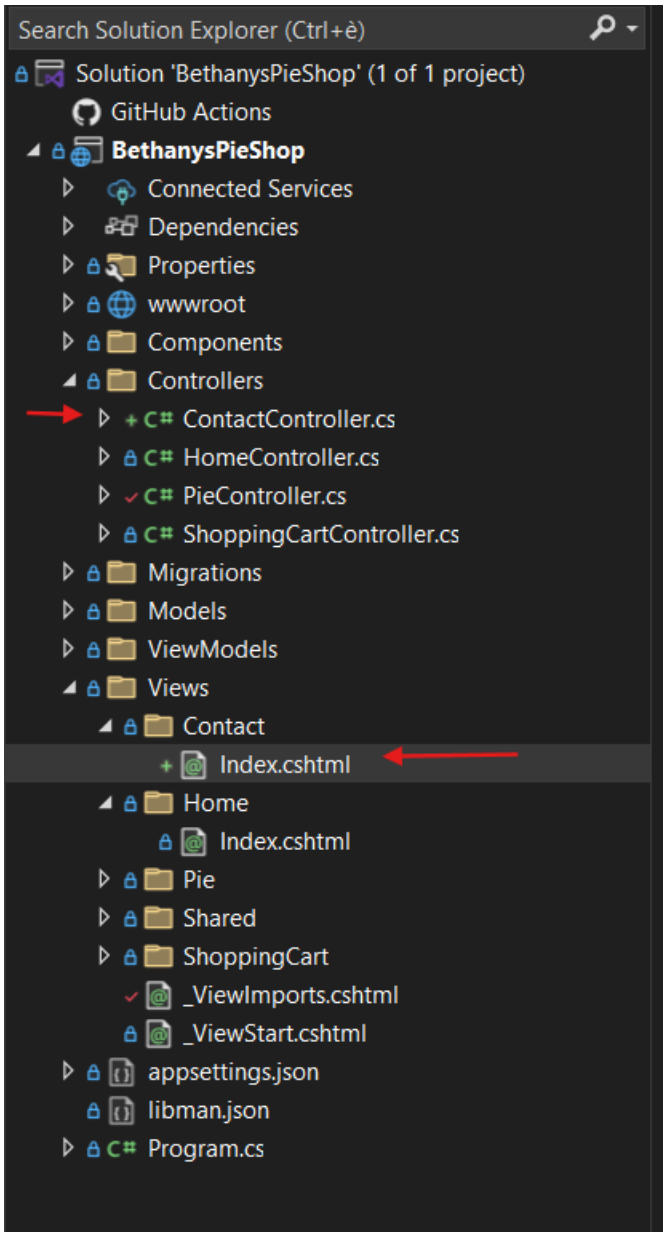



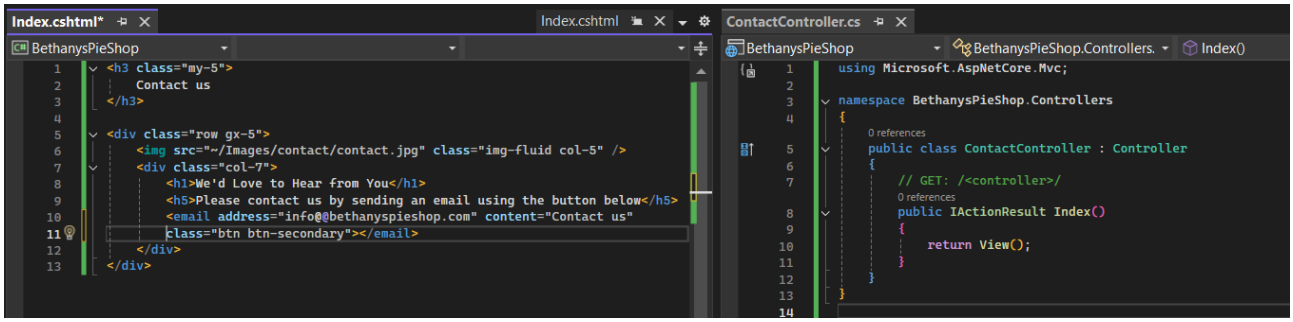

**Demo: Creating the Email Tag Helper**

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

Il tag helper che creeremo ci consentirà ci 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:**




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

```
EmailTagHelper.cs
BethanysPieShop | BethanysPieShop.TagHelpers.EmailTagHelper | Content
1  using Microsoft.AspNetCore.Razor.TagHelpers;
2
3  namespace BethanysPieShop.TagHelpers
4  {
       0 references
5      public class EmailTagHelper: TagHelper
6      {
           1 reference
7          public string? Address { get; set; }
           1 reference
8          public string? Content { get; set; }
9
           0 references
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     }
```

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

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

Salviamo e buildiamo.

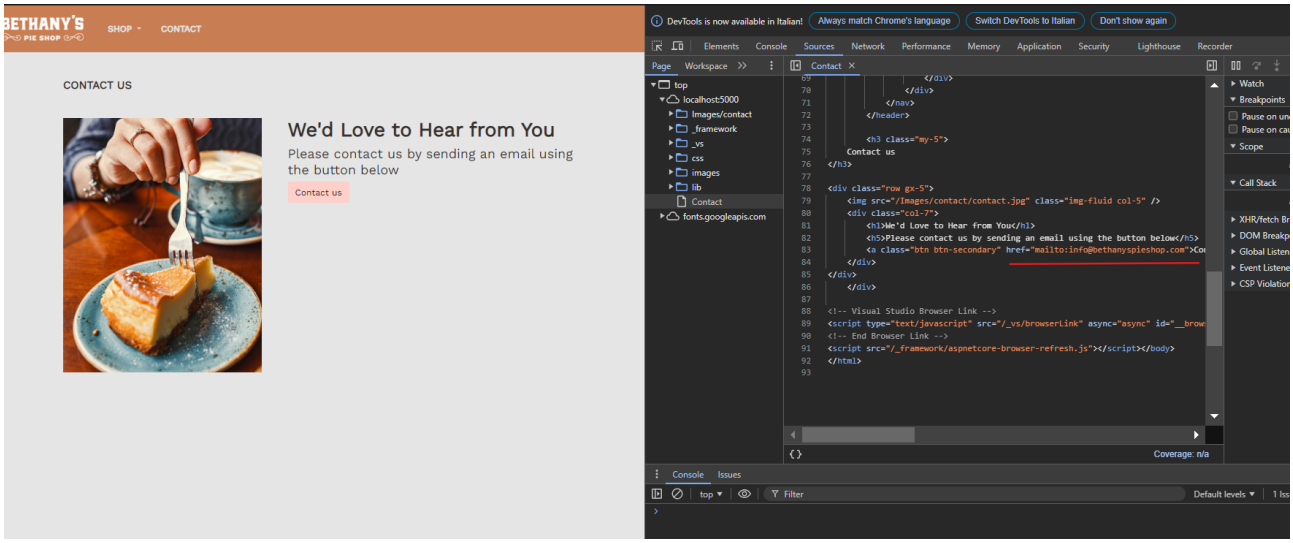
Torniamo in Index.cshtml di Views > Contact e ora possiamo utilizzare il nostro tag helper.
**Noi lo abbiamo già fatto implementato scrivendo il seguente elemento**:
*<email address="info@@bethanyspieshop.com" content="Contact us" class="btn btn-secondary"></email>*

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

Index.cshtml

BethanysPieShop

```
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                          <img src="images/bethanys-pie-shop-logo_horiz-white.png" width="151" height="47" class="d-inline-block align-t
21                               alt="Bethany's Pie Shop Logo">
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>
```

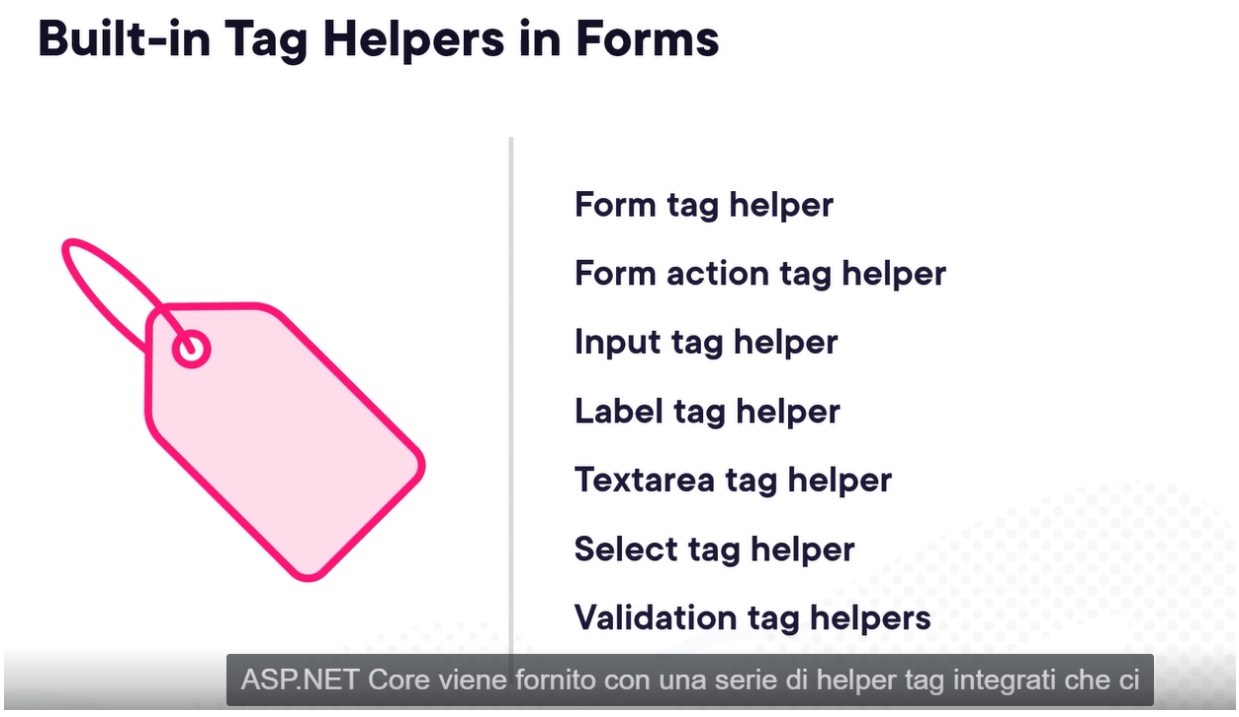



Osservazione: Una volta salvato e avviato l'applicazione, giungendo nella vista Contact possiamo notare con il DevTools che è la risorsa mailto:infor@bethanyspieshop.com è presente ed è un link che una volta cliccato aprirà sul nostro client il servizio di posta elettronica configurato sulla nostra macchina, 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**




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>
```

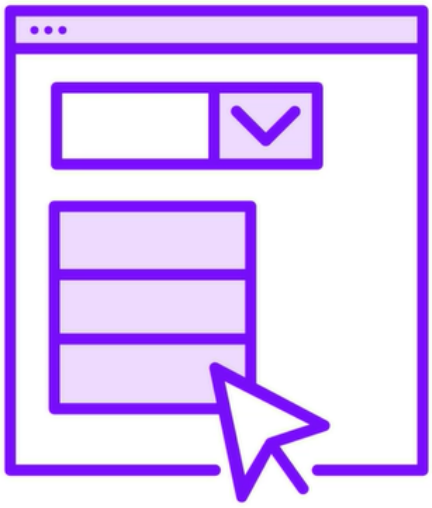
◂ Label Tag Helper

◂ Resulting HTML

◂ Attributes on Model

◂ **Other HTML attributes**

# Form Tag Helpers



**asp-controller**

**asp-action**

**asp-route-***

**asp-route**

**asp-antiforgery**


**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**:

Order.cs

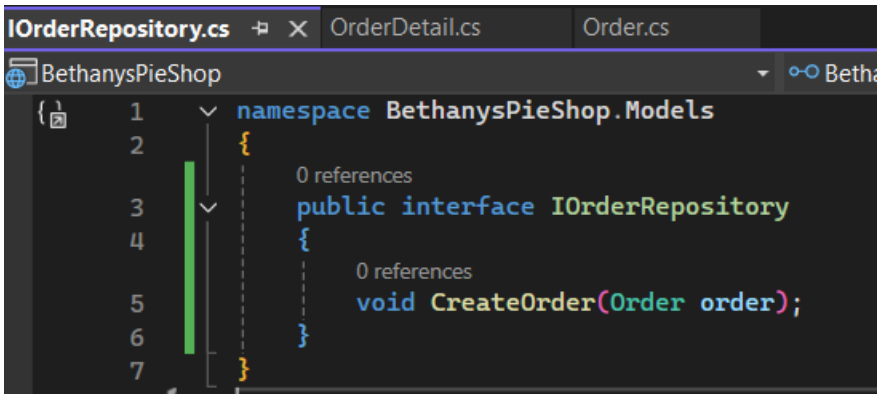
BethanysPieShop — BethanysPieShop.Models.Order

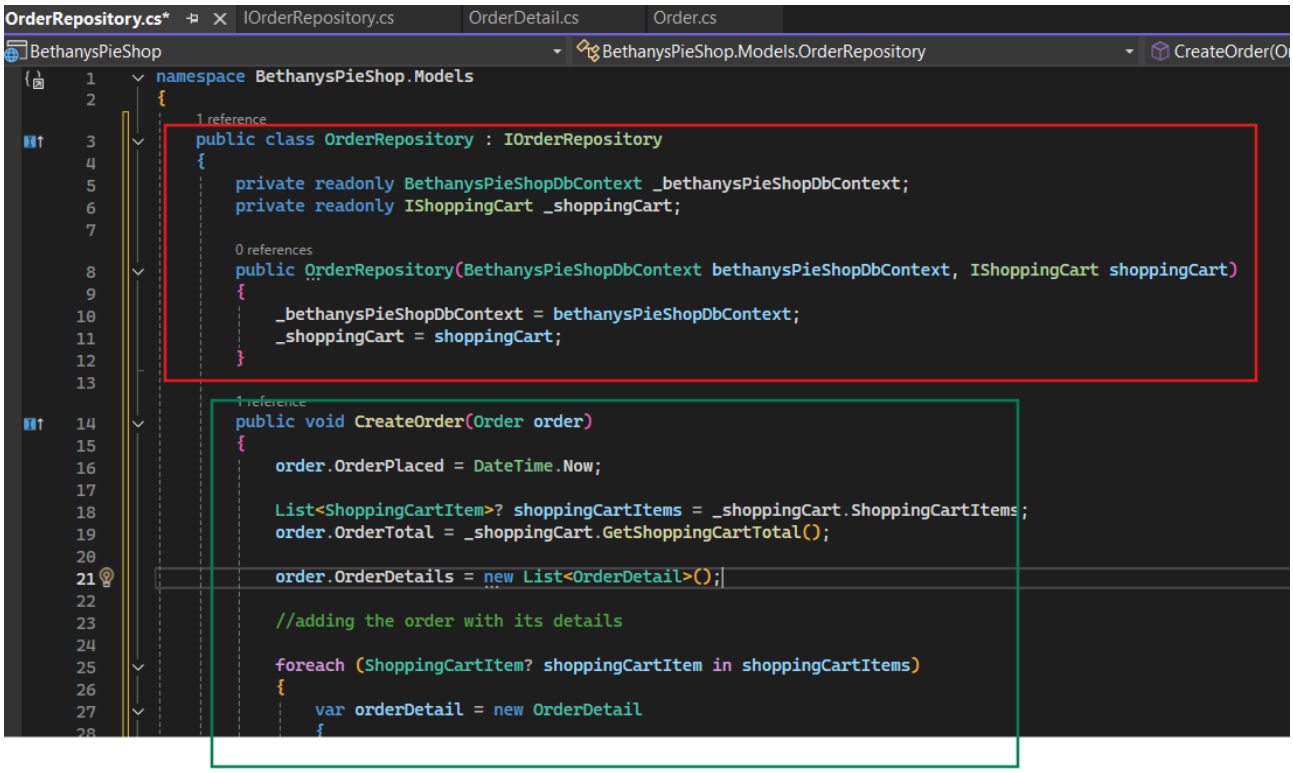
```
1   namespace BethanysPieShop.Models
2   {
        0 references
3       public class Order
4       {
            0 references
5           public int OrderId { get; set; }
6
            0 references
7           public List<OrderDetail>? OrderDetails { get; set; }
8
            0 references
9           public string FirstName { get; set; } = string.Empty;
10
            0 references
11          public string LastName { get; set; } = string.Empty;
12
            0 references
13          public string AddressLine1 { get; set; } = string.Empty;
14
            0 references
15          public string? AddressLine2 { get; set; }
16
            0 references
17          public string ZipCode { get; set; } = string.Empty;
```

OrderDetail.cs | Order.cs

BethanysPieShop — BethanysPieShop.Models.Or

```
1   namespace BethanysPieShop.Models
2   {
        1 reference
3       public class OrderDetail
4       {
            0 references
5           public int OrderDetailId { get; set; }
            0 references
6           public int OrderId { get; set; }
            0 references
7           public int PieId { get; set; }
            0 references
8           public int Amount { get; set; }
            0 references
9           public decimal Price { get; set; }
            0 references
10          public Pie Pie { get; set; } = default!;
            0 references
11          public Order Order { get; set; } = default!;
12      }
13  }
```




**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 aggiunga, salvataggio e tracking del db:

```
1 reference
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
        {
            Amount = shoppingCartItem.Amount,
            PieId = shoppingCartItem.Pie.PieId,
            Price = shoppingCartItem.Pie.Price
        };

        order.OrderDetails.Add(orderDetail);
    }

    _bethanysPieShopDbContext.Orders.Add(order);

    _bethanysPieShopDbContext.SaveChanges();
}
}
}
```

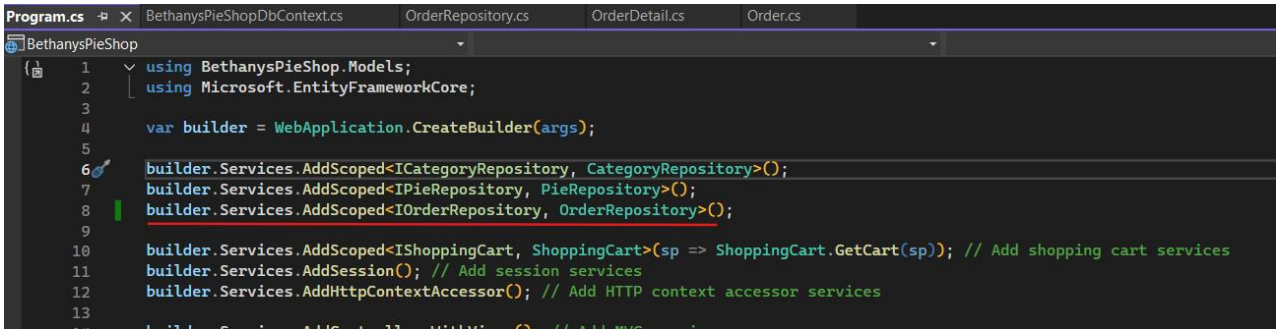
Notiamo l'errore a riga 37 perché nel DbContext di Bethany's Pie Shop devo ancora aggiungre un DbSet per Orders e OrderDetails.

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

```
BethanysPie...DbContext.cs*  OrderRepository.cs*  IOrderRepository.cs  OrderDetail.cs  Order.cs
BethanysPieShop                BethanysPieShop.Models.BethanysPieShopDbContext
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    18 references
    public class BethanysPieShopDbContext : DbContext
    {
        0 references
        public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> opt
        {

        }

        3 references
        public DbSet<Category> Categories { get; set; }
        4 references
        public DbSet<Pie> Pies { get; set; }
        8 references
        public DbSet<ShoppingCartItem> ShoppingCartItems { get; set; }
        1 reference
        public DbSet<Order> Orders { get; set; }
        0 references
        public DbSet<OrderDetail> OrderDetails { get; set; }
    }
}
```

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

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

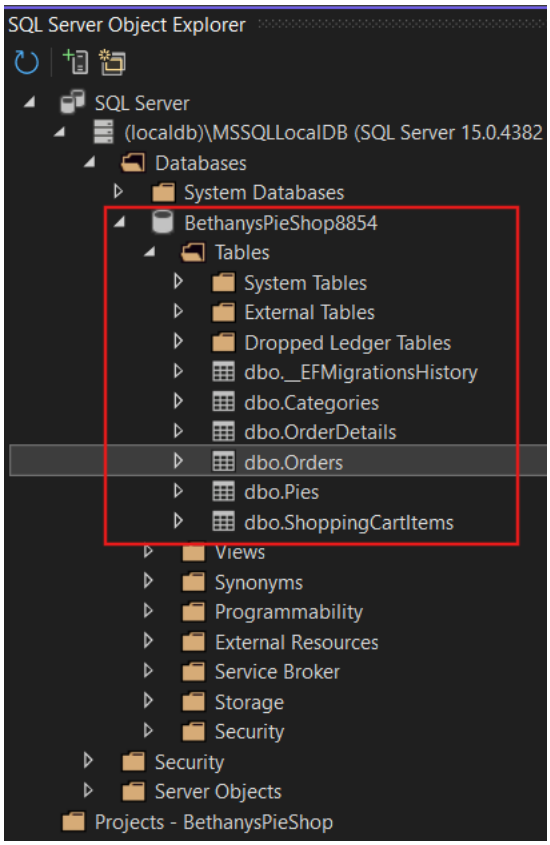



Eseguiamo una build.

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

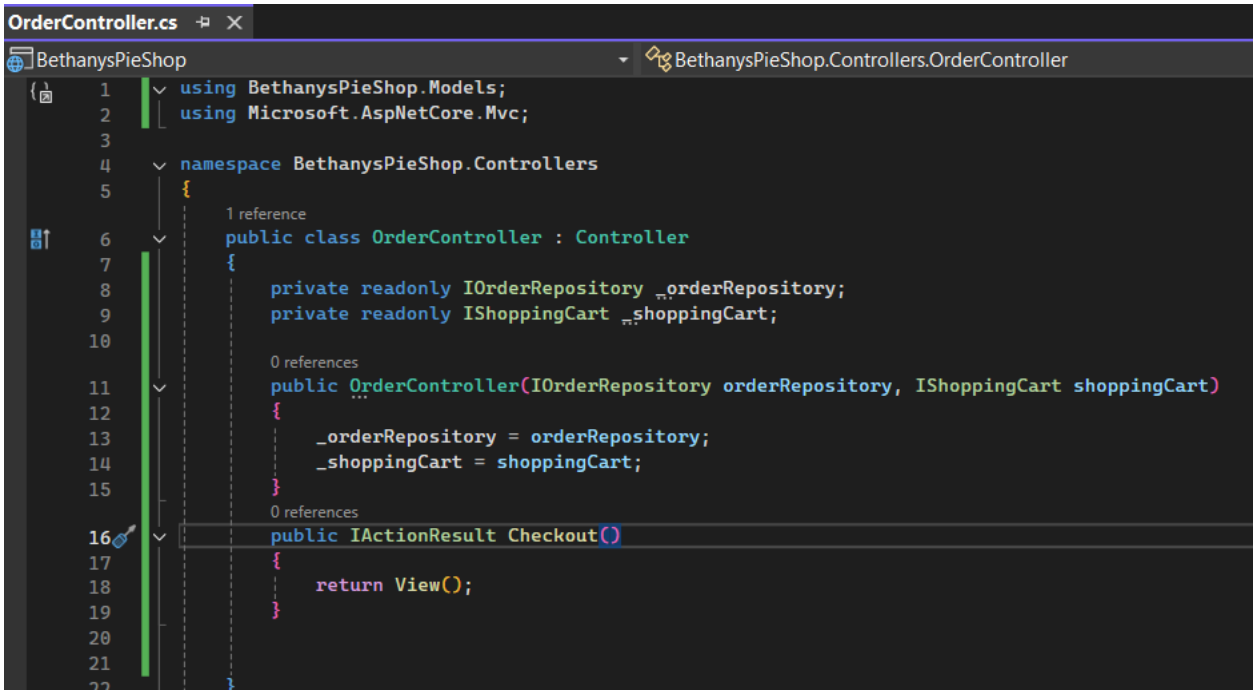
Apriamo il **PM Console e digitiamo:**
*add-migration OrderAdded*

Dopo l'aggiunta digitiamo:
*update-database*

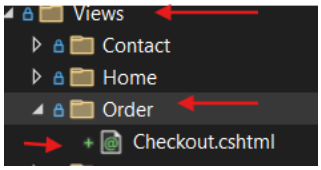


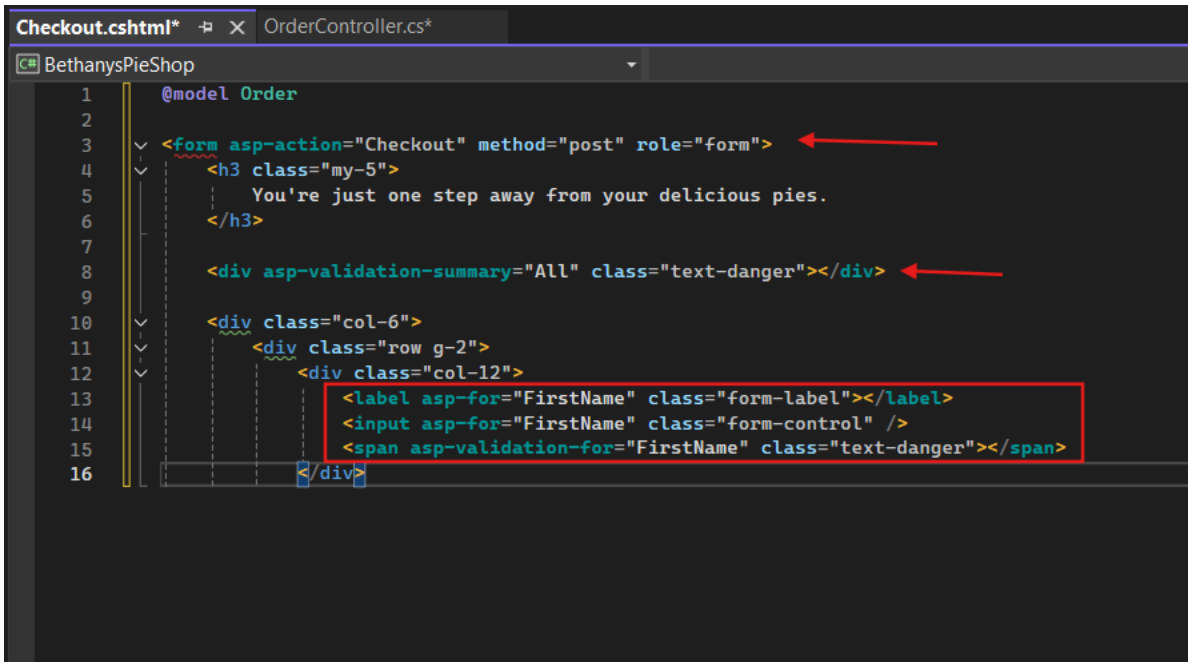

**Ora possiamo procedere con la creazione del Modulo Ordine**

Creiamo un OrderController:



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




**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 proèprietà.

Quindi creo un 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:

Checkout.cshtml*

BethanysPieShop

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

            <div class="col-12">
                <label asp-for="Email" class="form-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>

            <div class="col-12">
                <label asp-for="PhoneNumber" class="form-label"></label>
                <input asp-for="PhoneNumber" class="form-control" />
                <span asp-validation-for="PhoneNumber" class="text-danger"></span>
            </div>
        </div>

        <div class="mt-2">
            <div class="col-md-offset-2 col-md-5">
                <input type="submit" class="btn btn-secondary" value="Complete order" />
            </div>
        </div>
    </div>
</form>
```

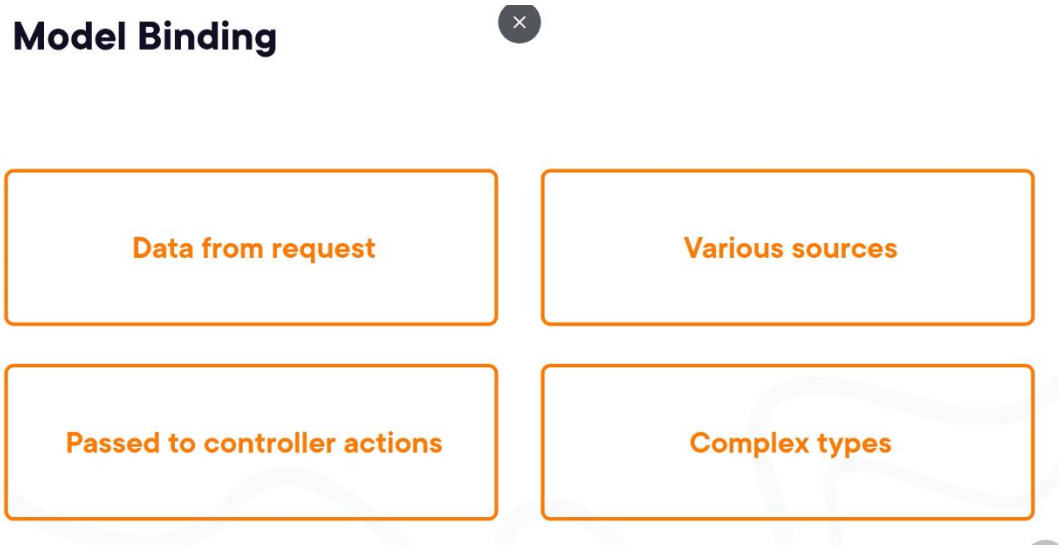
Adesso ci assicuriamo che sia possibile navigare sullo schermo dal carrello, quindi ci rechiamo in Index.cshtml presente nella cartella ShoppingCart e aggiungiamo il collegamento:

```
    </div>
    <div class="col-4">
        <div class="card shopping-cart-card p-3">
            <div class="row">
                <h4 class="col">Total:</h4>
                <h4 class="col text-end">@Model.ShoppingCartTotal.ToString("c")</h4>
            </div>
            <hr />
            <div class="text-center d-grid">
                <a class="btn btn-secondary" asp-controller="Order" asp-action="Checkout"><h4>Check out now!</h4></a>
            </div>
        </div>
    </div>
</div>
```

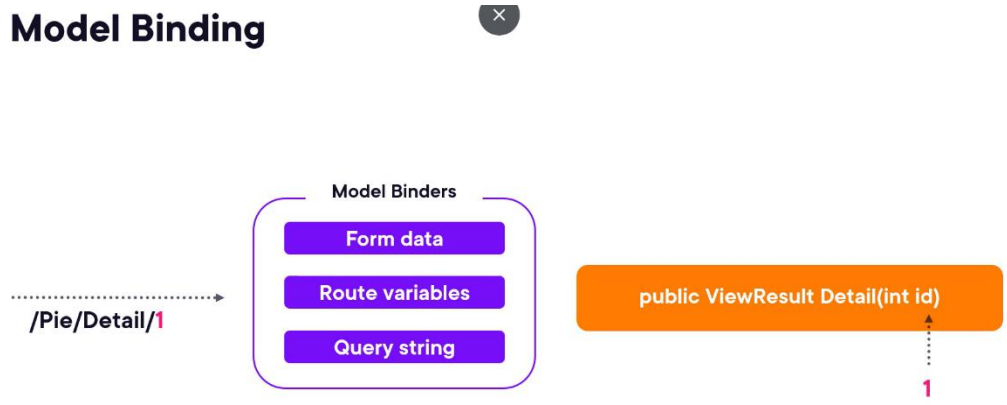
Ora siamo in grado di poter visualizzare il Checkout del nostro Carrello e poter visionare il form per l'acquisto, naturalmente il submit non inolterà ancora nessun dato.

**Understanding Model Binding**

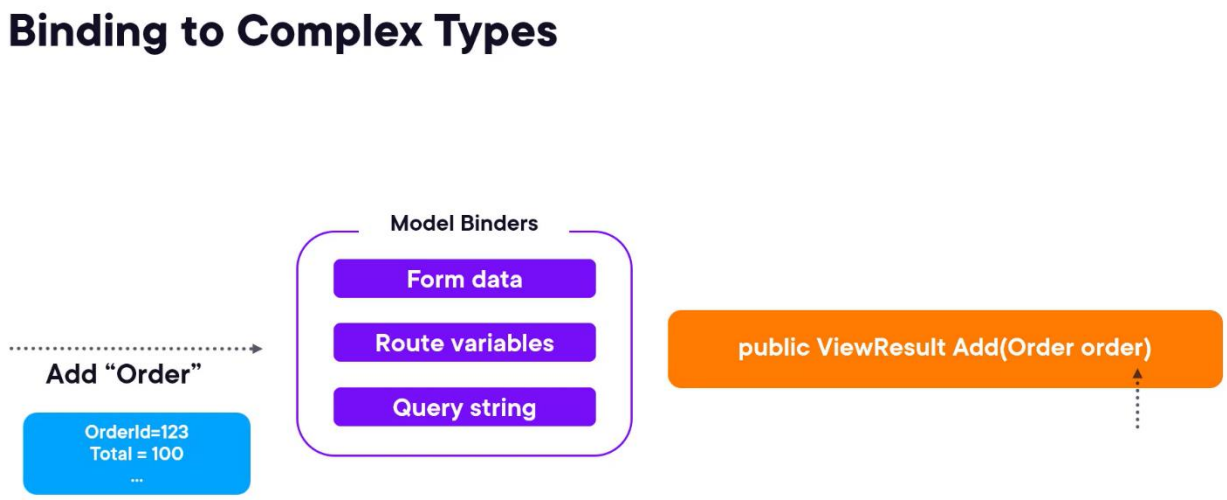
I dati inviati con il form (modulo) sono presenti nel corpo, la richiesta sarà di tipo post.
Per evitare di riscrivere codice ridondante e ottenere maggiore scalabilità, ASP.NET ci viene incontro con il model binding, il quale ci permette di ottenere l'accesso ai dati in arrivo con la richiesta.



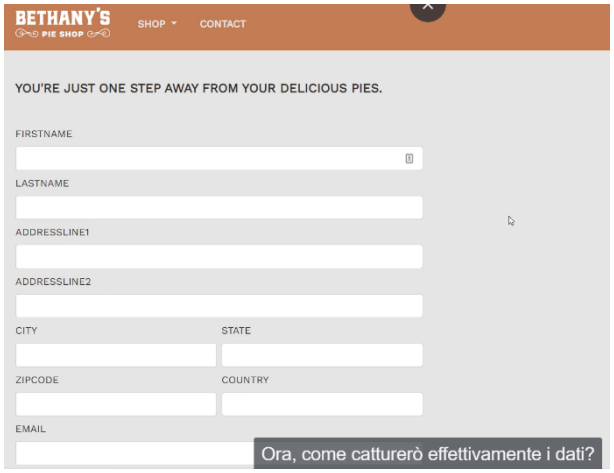

Immagina di avere un metodo di azione *public ViewResult Detail(int id)* , che richiede un id di tipo int come parametro, ASP.NET core necessita che sia presente un valore per poter chiamare questo metod, l'associazione del modello tenterà di trovare il valore per questo parametro id, quando eseguiamo 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 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




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



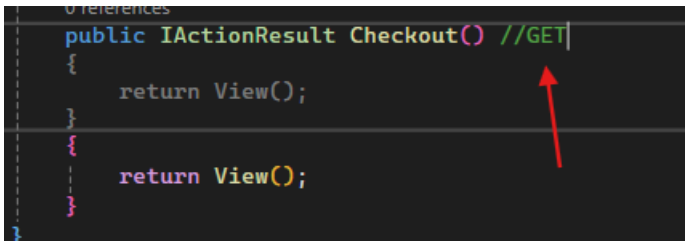

**Demo: Accessing Posted Data Using Model Binding**




Torniamo al nostro Controller:

```
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:



Checkout.cshtml | OrderController.cs*

BethanysPieShop

```
@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"></sp
            </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"></spa
            </div>
```

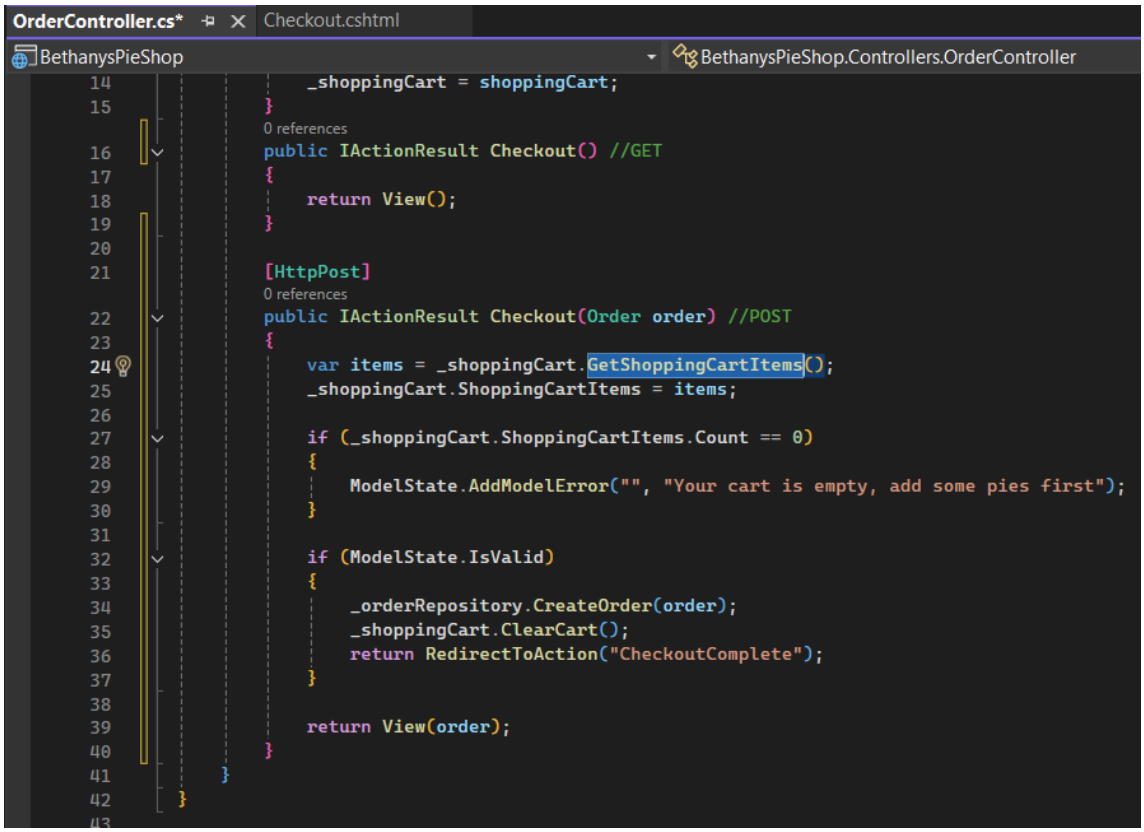
In Checkout.cshtml abbiamo un post, devo creare un altro metodo di azione che verrà richiamato quando viene ricevuto un post

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:




Osservazioni:
Verifichiamo prima che l'utente abbia effettivamente articoli nel carrello.
Se cosi 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.

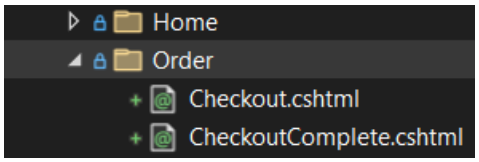
Il ModelState ha anche una proprietà di Validità (riga 32), è valida se l'associazione del modello è andata a buon fine e non stati aggiunti ulteriori errori.

In caso positivo, viene creato l'ordine con _orderRepository e quindi nel db, viene cancellata la cart, ed effettuiamo un reinderizzamento 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:

```
OrderController.cs*
BethanysPieShop | BethanysPieShop.Controllers.OrderController | CheckoutComplete()

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
            0 references
42      public IActionResult CheckoutComplete()
43      {
44          ViewBag.CheckoutCompleteMessage = "Thanks for your order. You'll soon enjoy our delicious pies!";
45          return View();
46      }
47  }
48  }
49
```

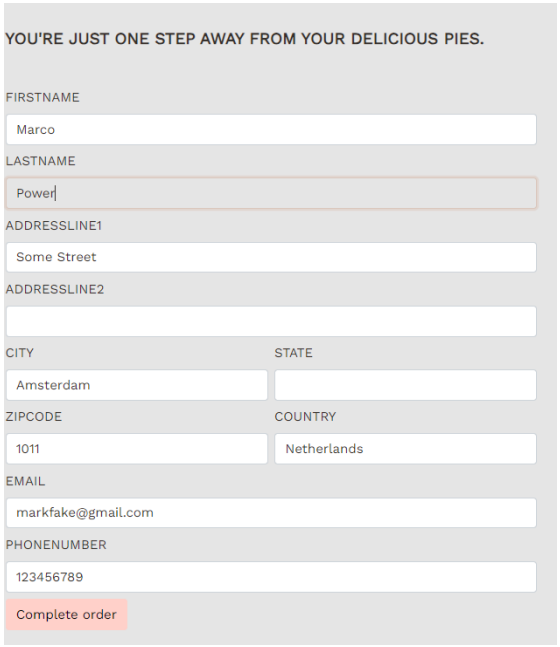
Creiamo la vista:



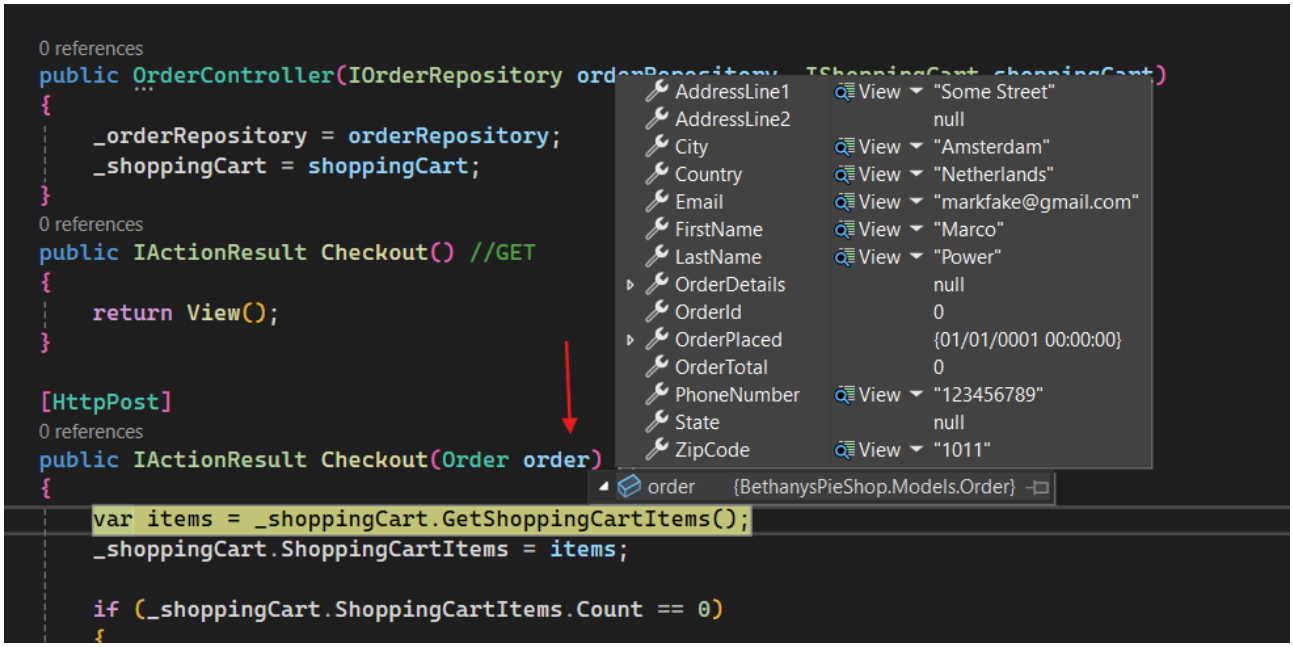

Inseriamo un breakpoint a riga 24 di OrderController.cs e avviamo:

```
            0 references
22          public IActionResult Checkout(Order order) //POST
23          {
24              var items = _shoppingCart.GetShoppingCartItems();
25              _shoppingCart.ShoppingCartItems = items;
26
27              if (_shoppingCart.ShoppingCartItems.Count == 0)
```

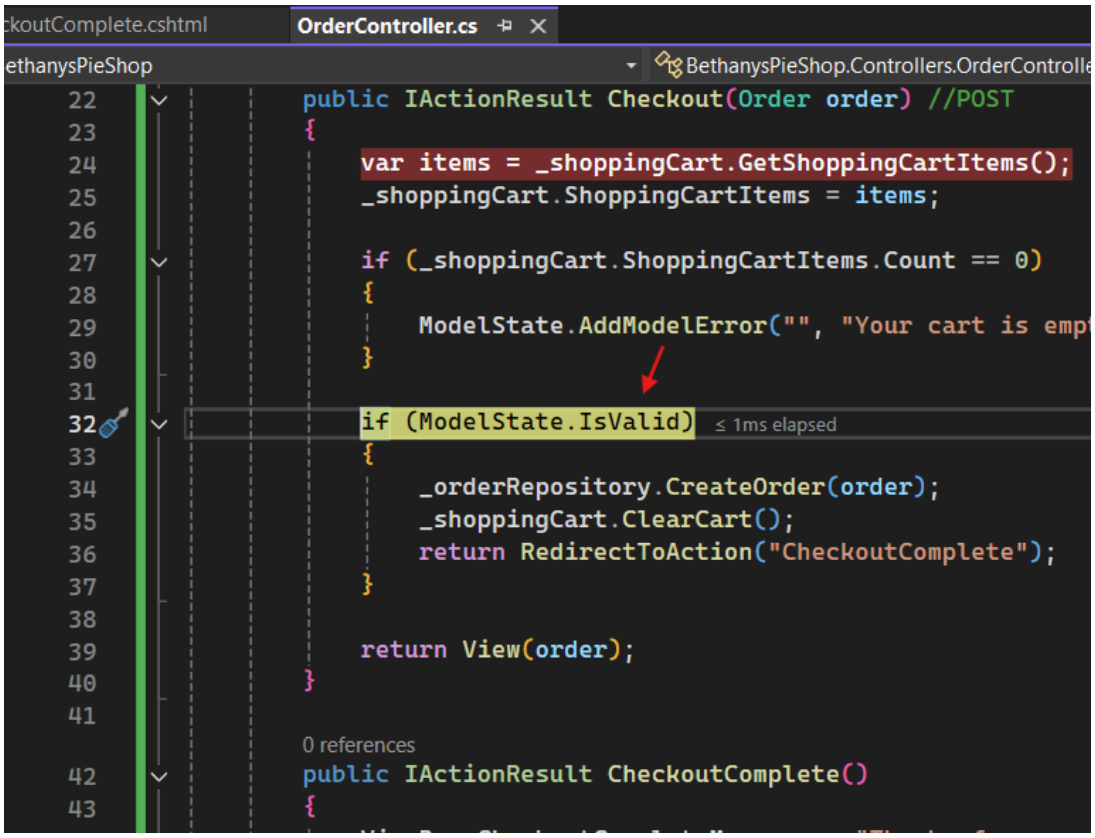
Di default avremo la vista del checkout senza dati, inseriamo dei dati manualmente:

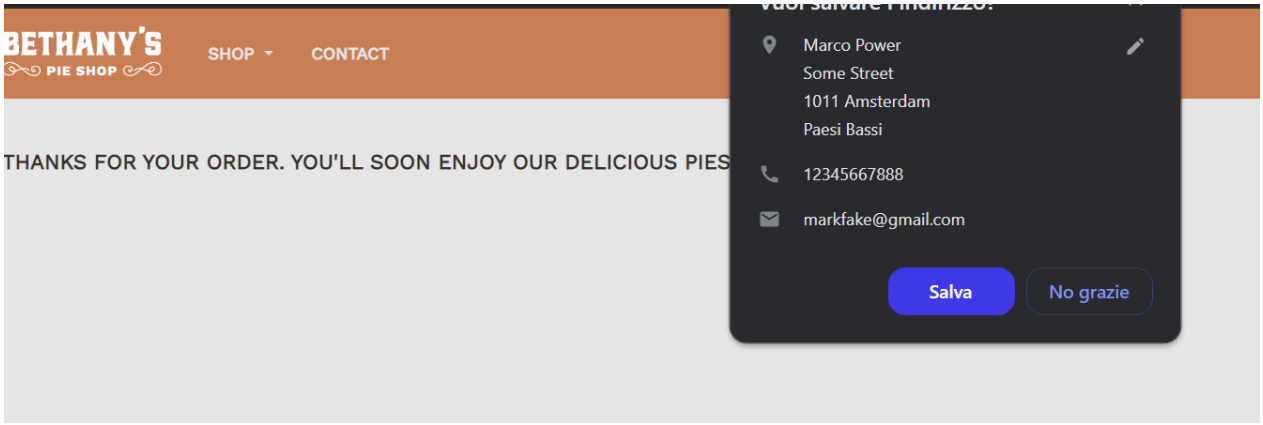



Click su *Complete order* questo attiverà il trigger dell'invio del modulo, quindi tornando nel nostro IDE con il debug attivo, noteremo che l'oggetto ordern ha effettuato con successo il model binding:



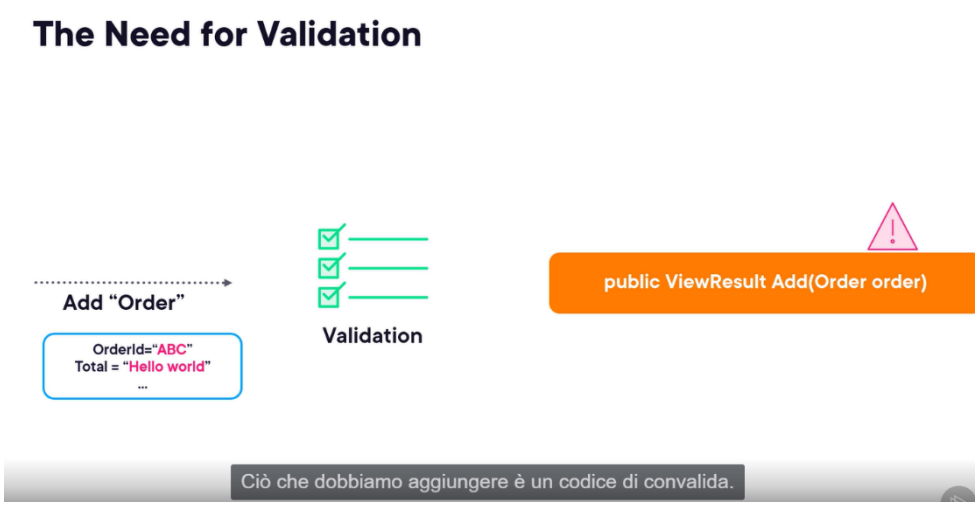

Proseguendo alle istruzioni successive, noteremo che IsValid è true se tutti i campi obbligatori sono stati inseriti:




**Adding Validation**

Cosa accade se l'utente inserisce dati non corretti?





Solo quando il modello sarà valido salverà l'ordine

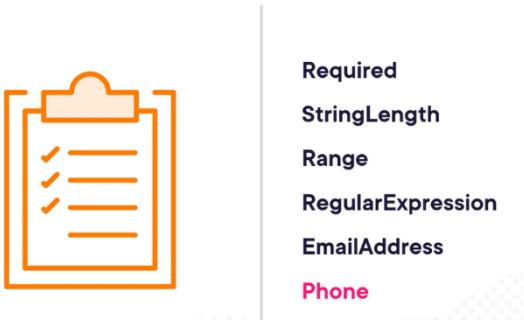
E' possibile eseguire una convalida del modello sull'intera istanza di un modello in una sola volta utilizzando ModelState.isValid, GetValidationState accetta una singola proprietà ed infine AddModelError

## ModelState Properties


## Validation


(Non verrà fatto in questo corso)

## Validation Attributes



## 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 mosterà io 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:

Order.cs*

BethanysPieShop | BethanysPieShop.Models.Order

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

Regular expression applicato sulla mail:

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

Andando in Checkout.cshtml

Checkout.cshtml

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"

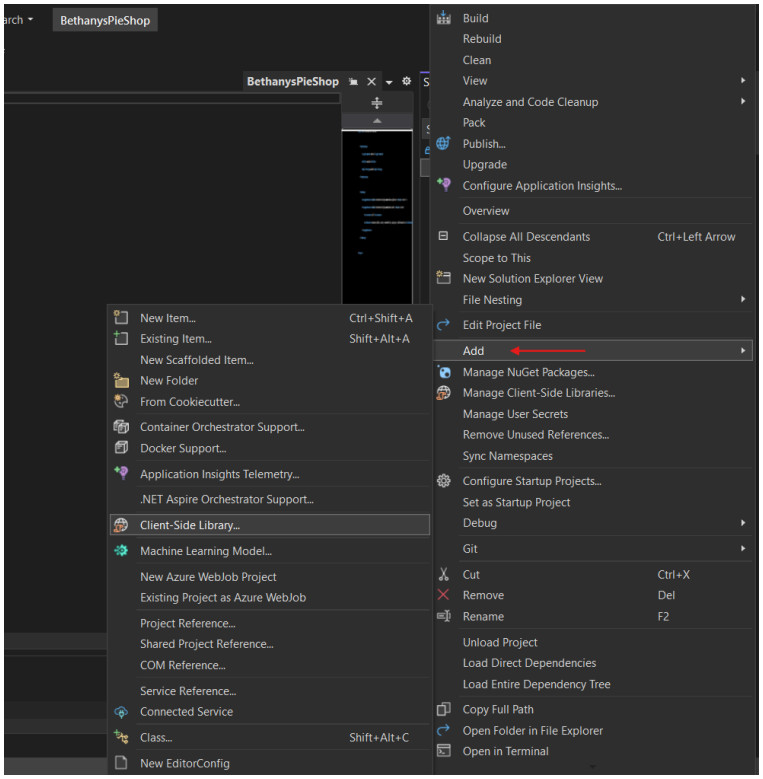
Difatti, se proviamo a cliccare su Complete Order del form ci compaiono errori di convalida che verranno visualizzati 2 volte.

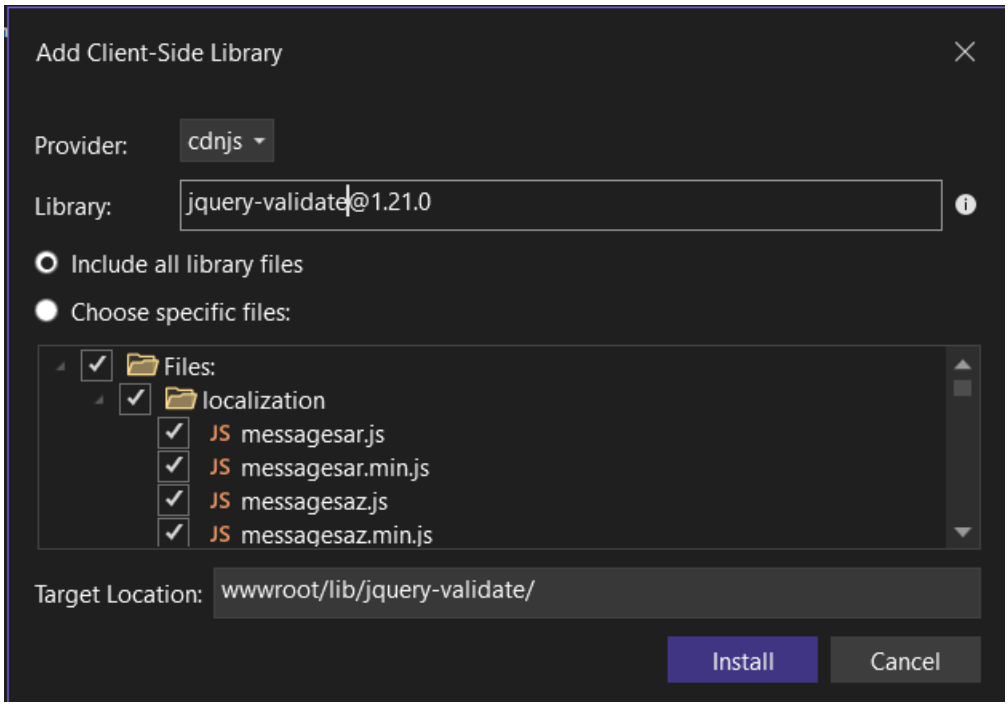
Questi errori sono lato server, però possiamo farlo anche lato client che può offrire all'utente un'esperienza migliore, vediamo come farlo.

**Demo: Adding Client-side Validation**

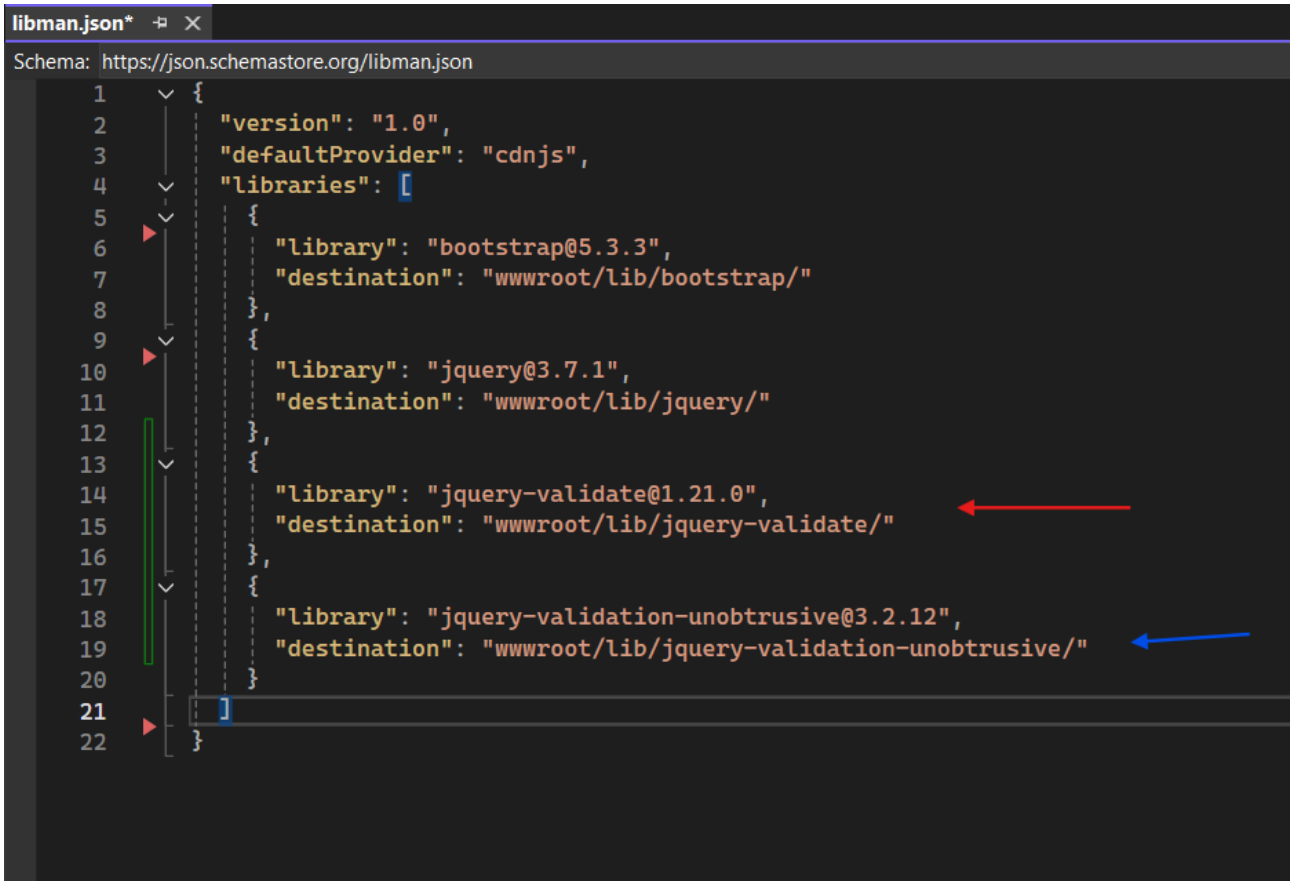
Introduciamo ora la convalida Jquery.

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

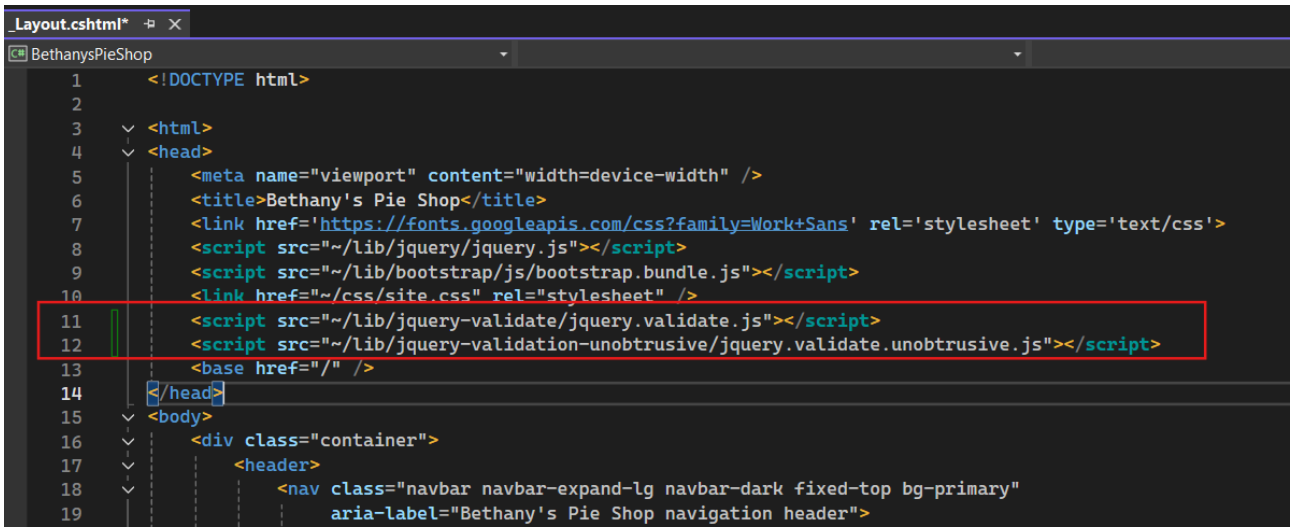




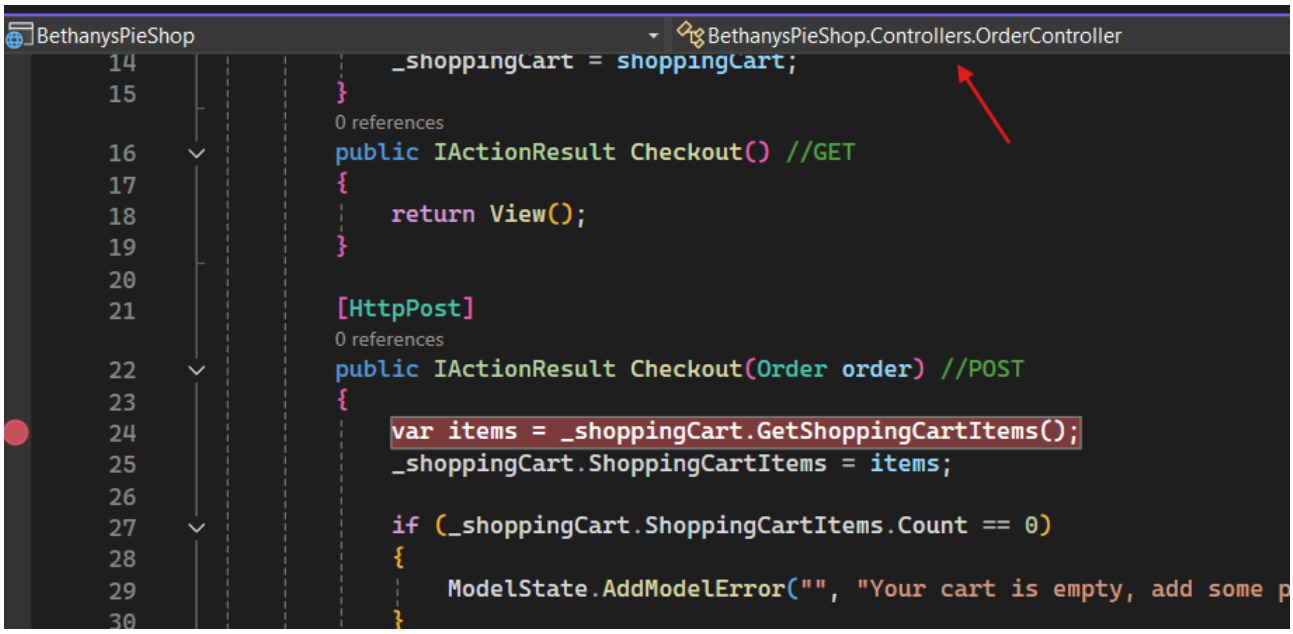

Ora in libman.json ho anche Jquery e aggiungiamo sotto una librerie di estensione/supporto a jquery-validate



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




Salviamo ed effettuiamo un piccolo debug inserendo un breakpoint qui:



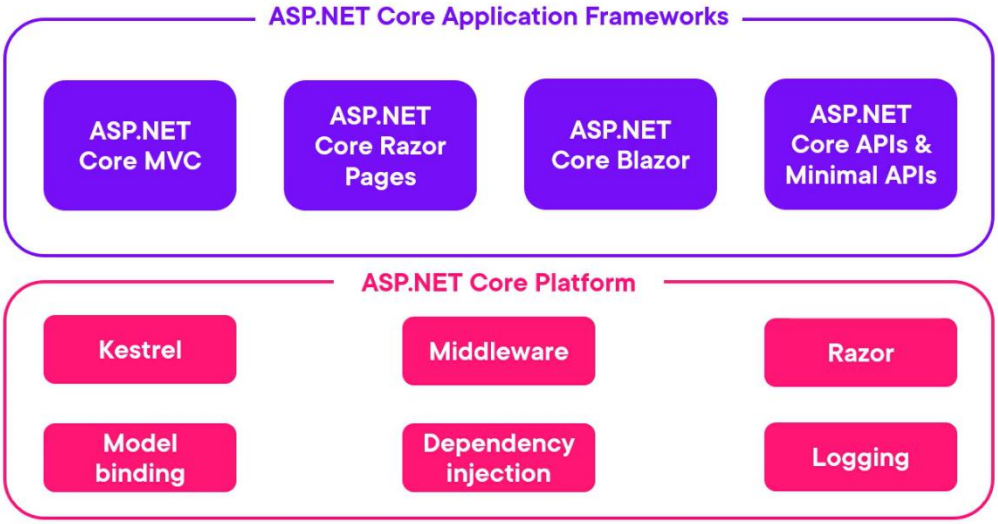

Ora i messaggi di errore saranno mostrati lato Client.

**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

```
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>
```

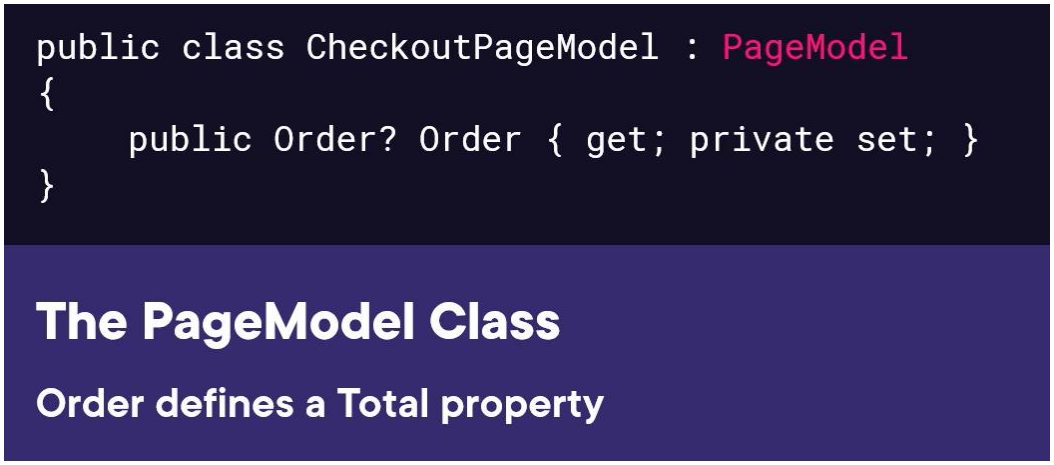
**A First Razor Page**

**@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:


Osservazione: Model scritto con la 'M' maiuscola.



# Page Handler Methods

Allows for separating of page and its data

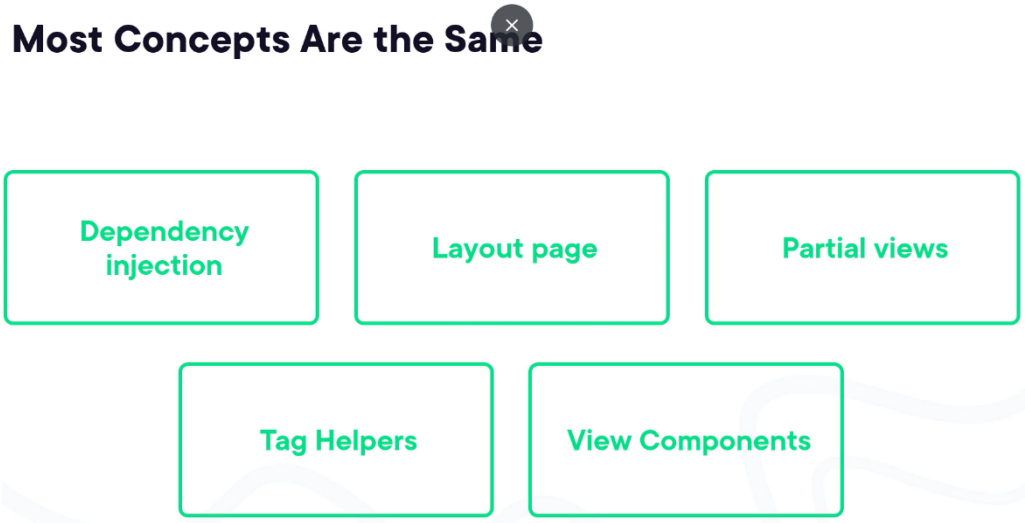
OnGet()

OnPost()

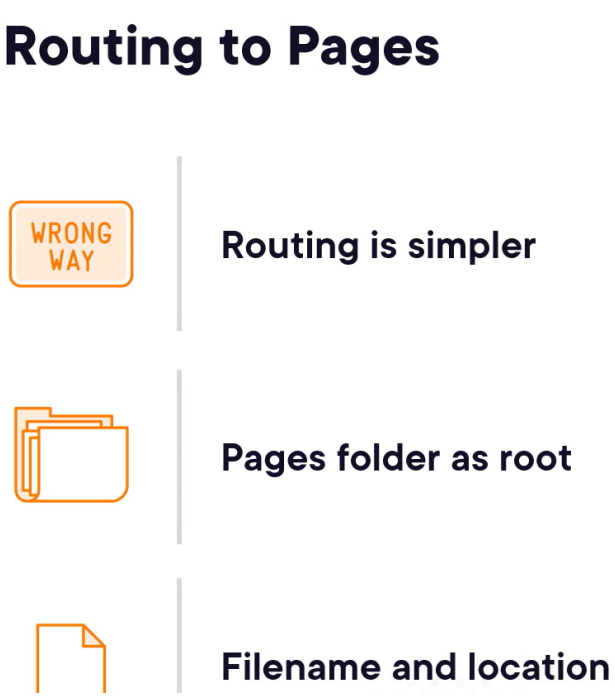
Async versions exist too

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


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

## Using OnGet()




Cosa è leggermente diverso:



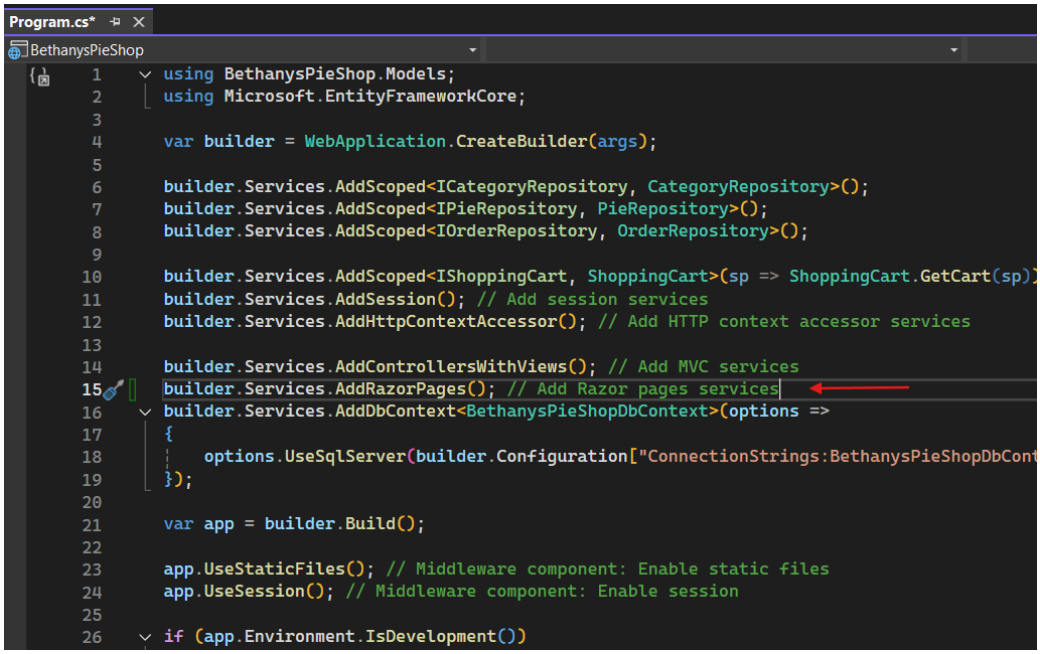

Esempio:

# 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<br>@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:

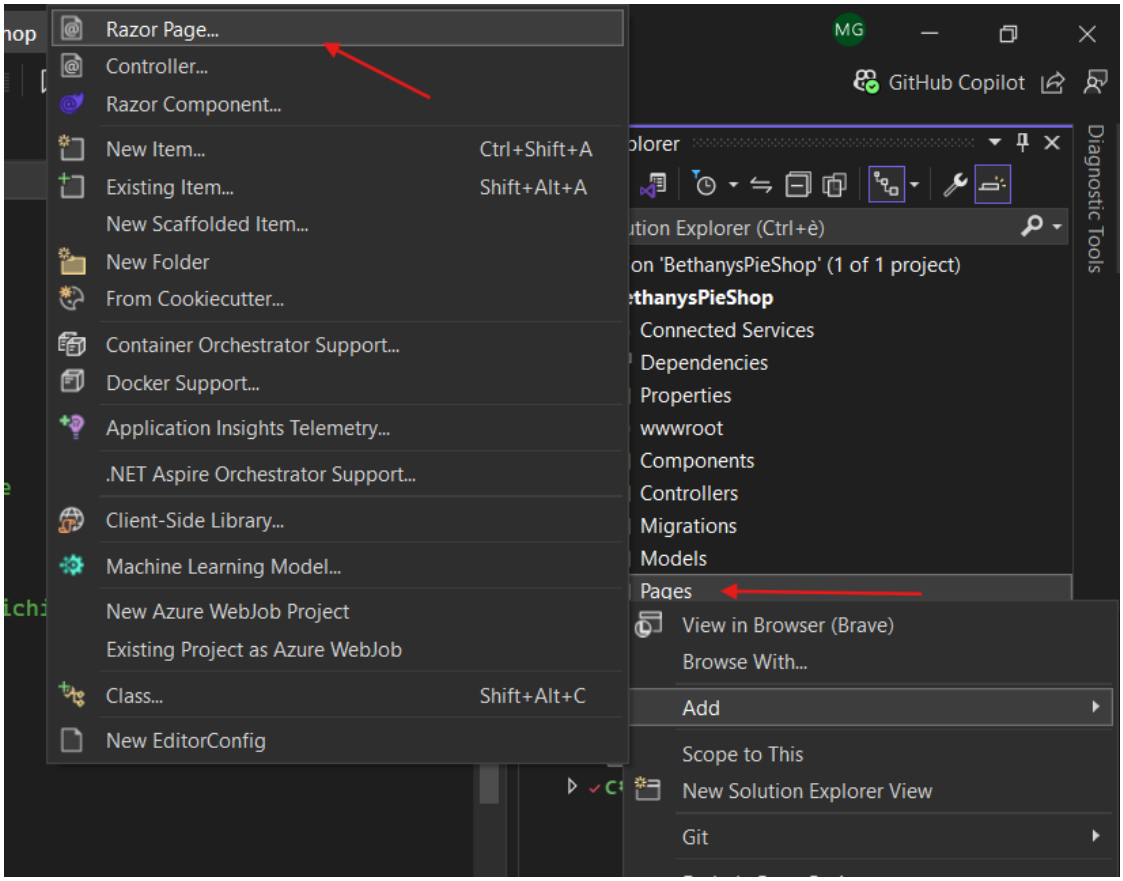



Osservazione: l'ordine in cui aggiungiamo i servizi non ha importanza, nel Middleware si.

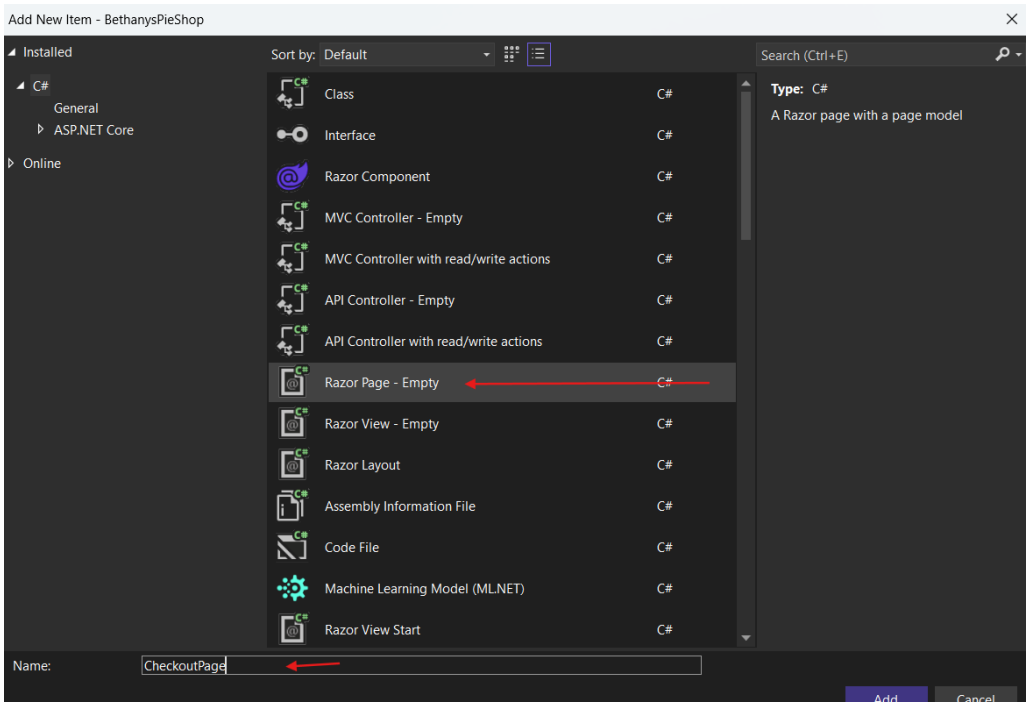
Aggiungiamo ora il middleware che permette di abilitare il modello Razor Page.:

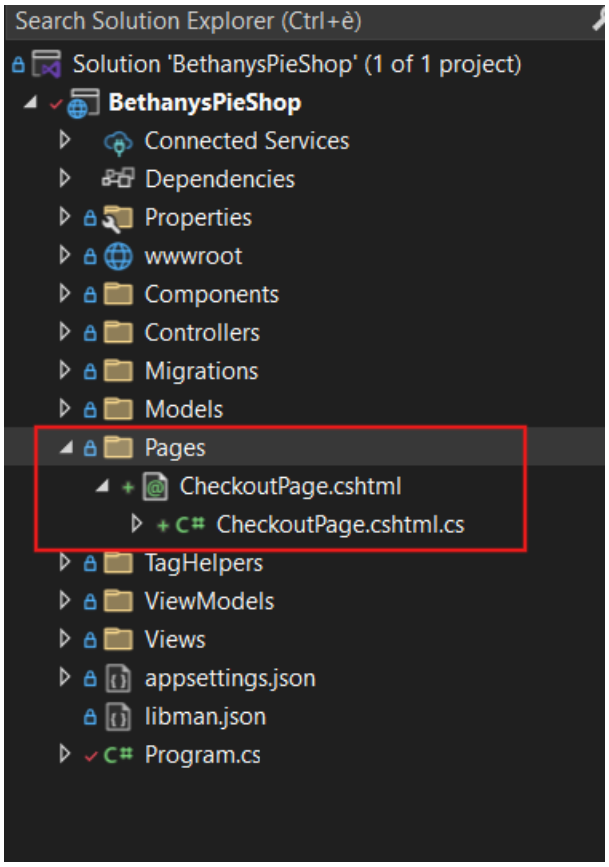
```
Program.cs*
BethanysPieShop
22
23      app.UseStaticFiles(); // Middleware component: Enable static files
24      app.UseSession(); // Middleware component: Enable session
25
26      if (app.Environment.IsDevelopment())
27      {
28          app.UseDeveloperExceptionPage(); // Middleware component: Enable devel
29      }
30
31      //"{controller=Home}/{action=Index}/{id?}" is the default route template
32      app.MapDefaultControllerRoute(); // Middleware component: Enable default c
33
34      //Se avessimo voluto personalizzare la rotta o il pattern
35      //app.MapControllerRoute(
36      //    name: "defaulte",
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:

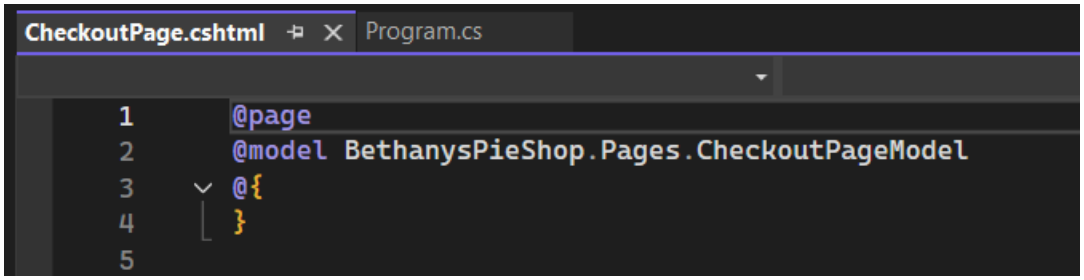


Ci assicuriamo di scegliere una Razor Page - Empty e la chiamiamo CheckoutPage:



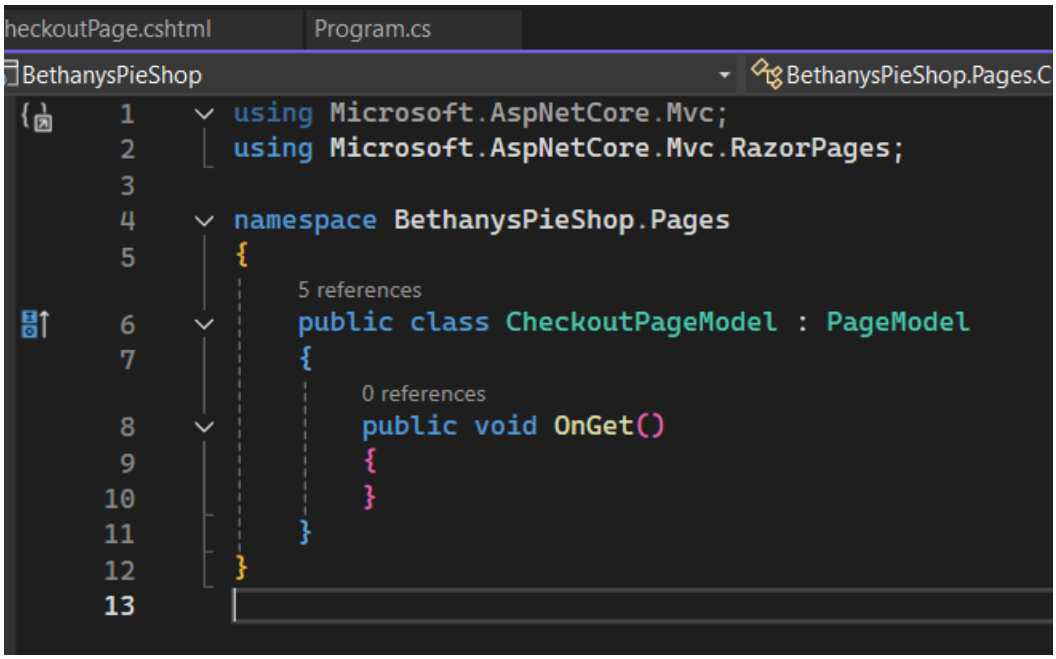



Osservazione: una Razor Page contiene un file .cshtml.cs "Code Behind"




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

Apriamolo:

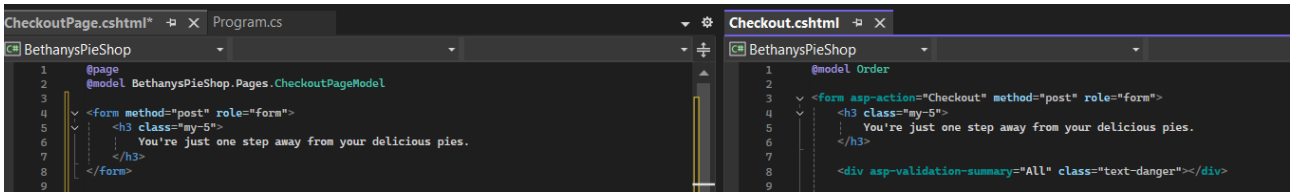



La convenzione vuole che ottiene lo stesso nome ma con suffisso .cs
Eredita da PageModel.

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:**

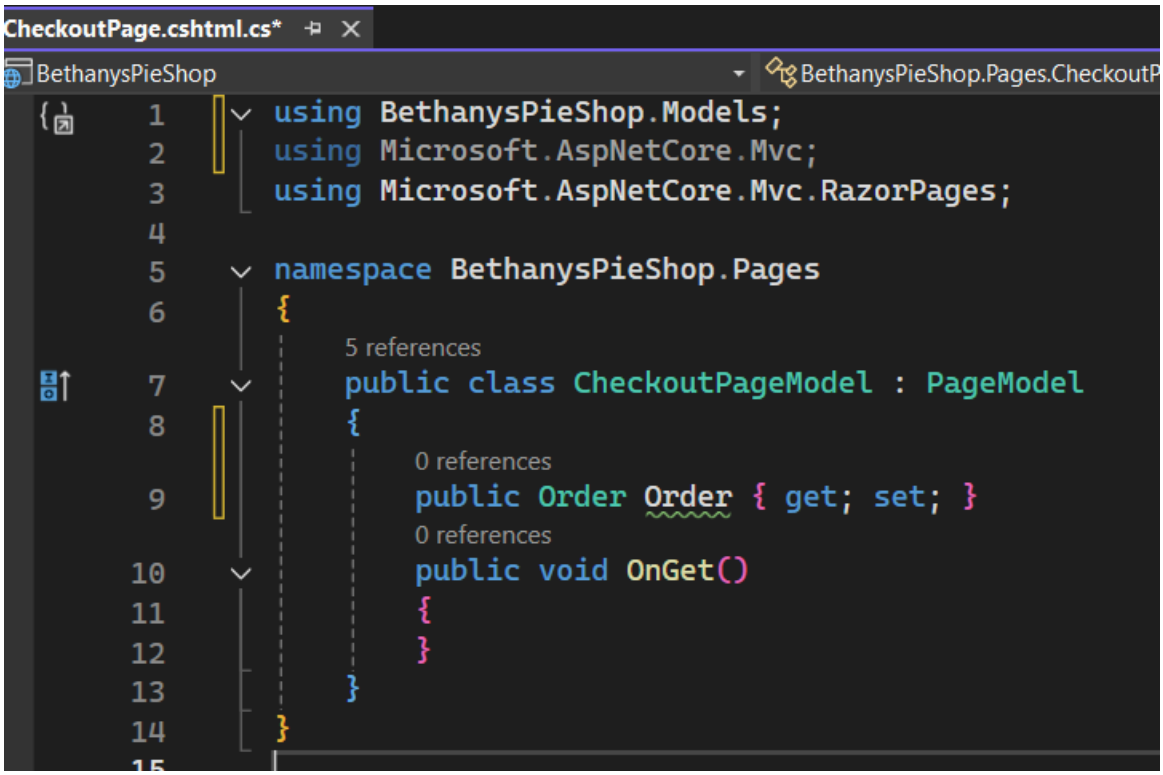
Aggiungiamo una piccola parte, come sarebbe stato fatto nel Checkout di Views, immagine che mostra la differenza:




Possiamo notare che per impostazione predefinita. Io 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 creaimo una proprietà di tipo Order:




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

CheckoutPage.cshtml*

BethanysPieShop

```
@page
@model BethanysPieShop.Pages.CheckoutPageModel

<form 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="Order.FirstName" class="form-label"></label>
                <input asp-for="Order.FirstName" class="form-control" />
                <span asp-validation-for="FirstName" class="text-danger"></span>
            </div>

            <div class="col-12">
                <label asp-for="Order.LastName" class="form-label"></label>
                <input asp-for="Order.LastName" class="form-control" />
                <span asp-validation-for="Order.LastName" class="text-danger"></span>
            </div>

            <div class="col-12">
                <label asp-for="Order.AddressLine1" class="form-label"></label>
                <input asp-for="Order.AddressLine1" class="form-control" />
                <span asp-validation-for="Order.AddressLine1" class="text-danger"></span>
            </div>

            <div class="col-12">
                <label asp-for="Order.AddressLine2" class="form-label"></label>
                <input asp-for="Order.AddressLine2" class="form-control" />
                <span asp-validation-for="Order.AddressLine2" class="text-danger"></span>
```

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

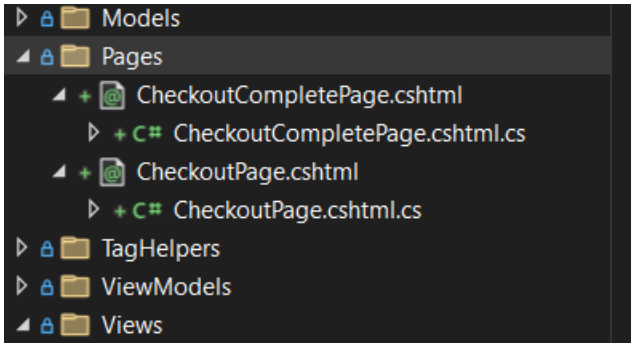
Capiamo cosi quanto siano utili le help tager e quanto il codice sia rimasto praticamente lo stesso.

Torniamo a CheckoutPageModel e lo completiamo:

CheckoutPage.cshtml.cs*

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

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


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

    0 references
    public void OnGet()
    {
    }


    0 references
    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:



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

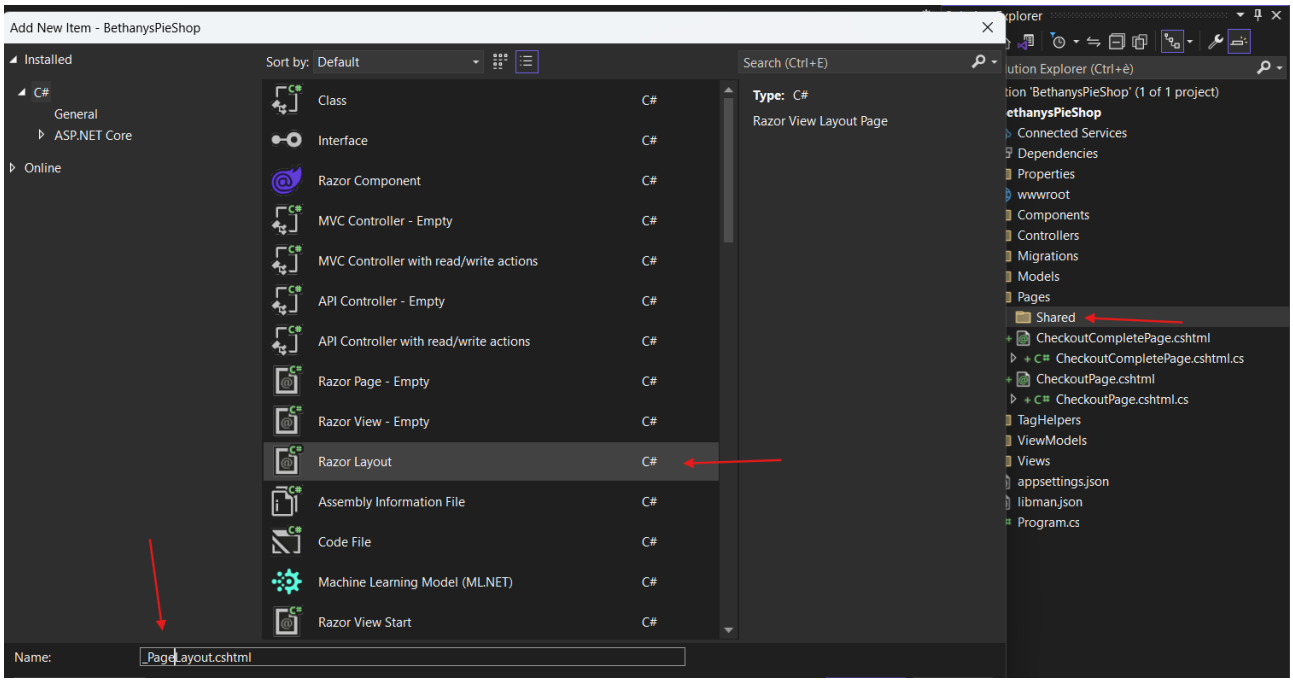
Osservazione: al suo interno usiamo un ViewData Dizionario.

Il suo .cs sarà cosi:

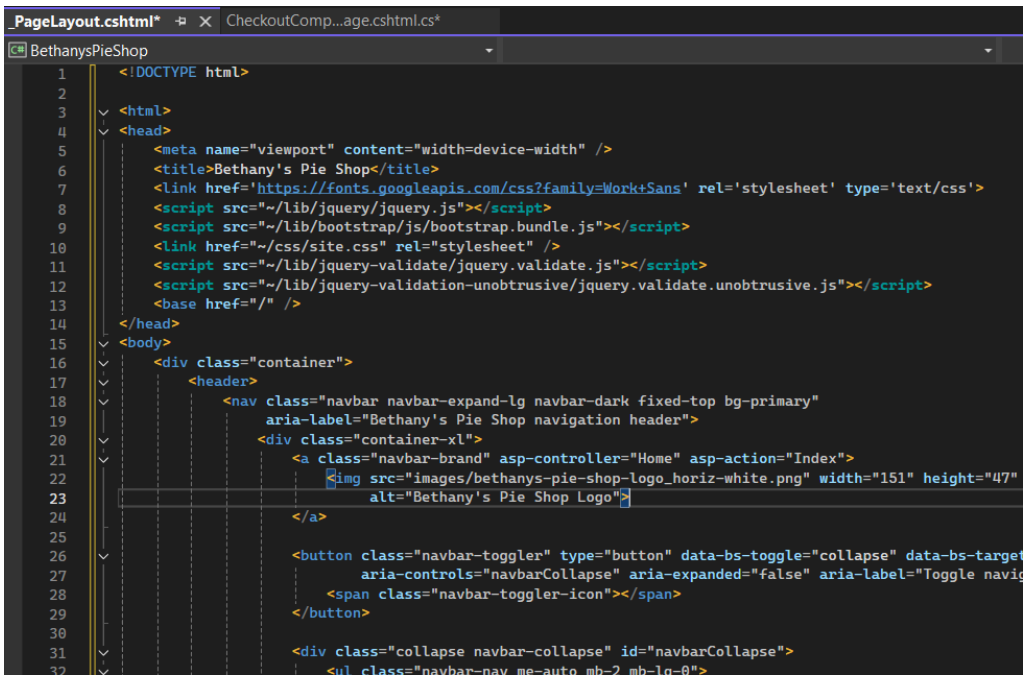
```
CheckoutCom...ge.cshtml.cs*
BethanysPieShop | BethanysPieShop.Pages.CheckoutCompletePageModel | OnGet()
1   using Microsoft.AspNetCore.Mvc;
2   using Microsoft.AspNetCore.Mvc.RazorPages;
3
4   namespace BethanysPieShop.Pages
5   {
        5 references
6       public class CheckoutCompletePageModel : PageModel
7       {
            0 references
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 in _ViewImports.cshtml.**
**Quindi creiamo un Layout separato e quindi una nuova cartella condivisa**

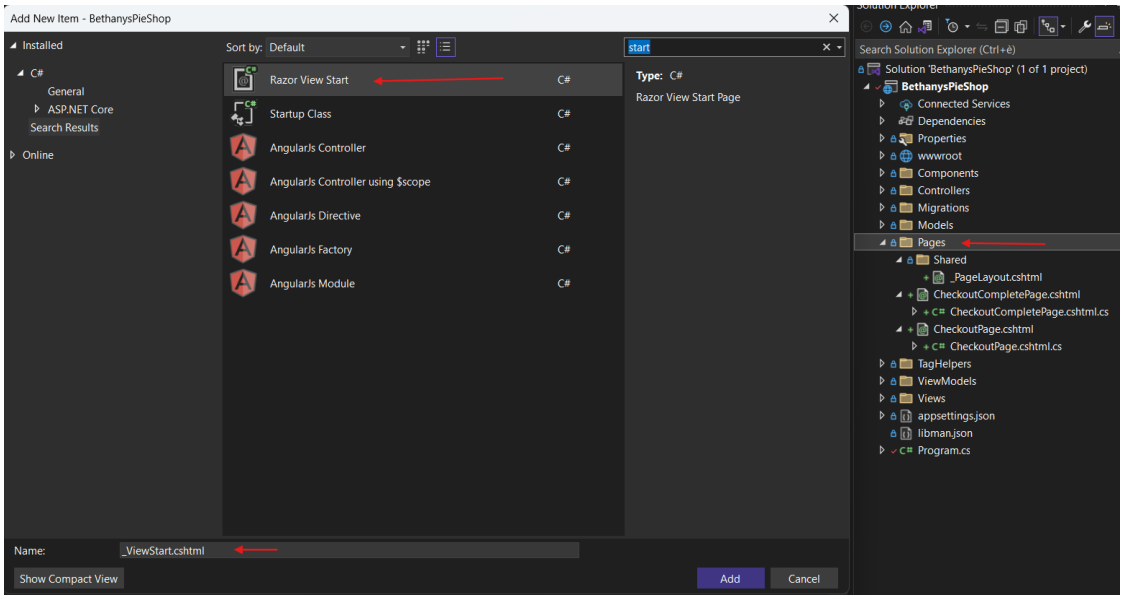


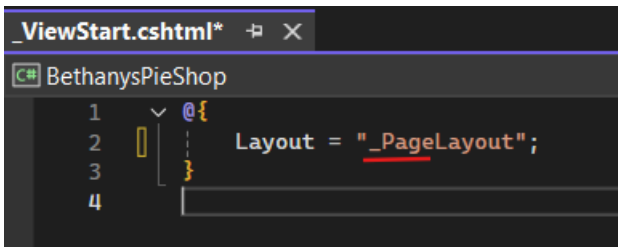

Inseriamo il _Layout di Views > Shared:



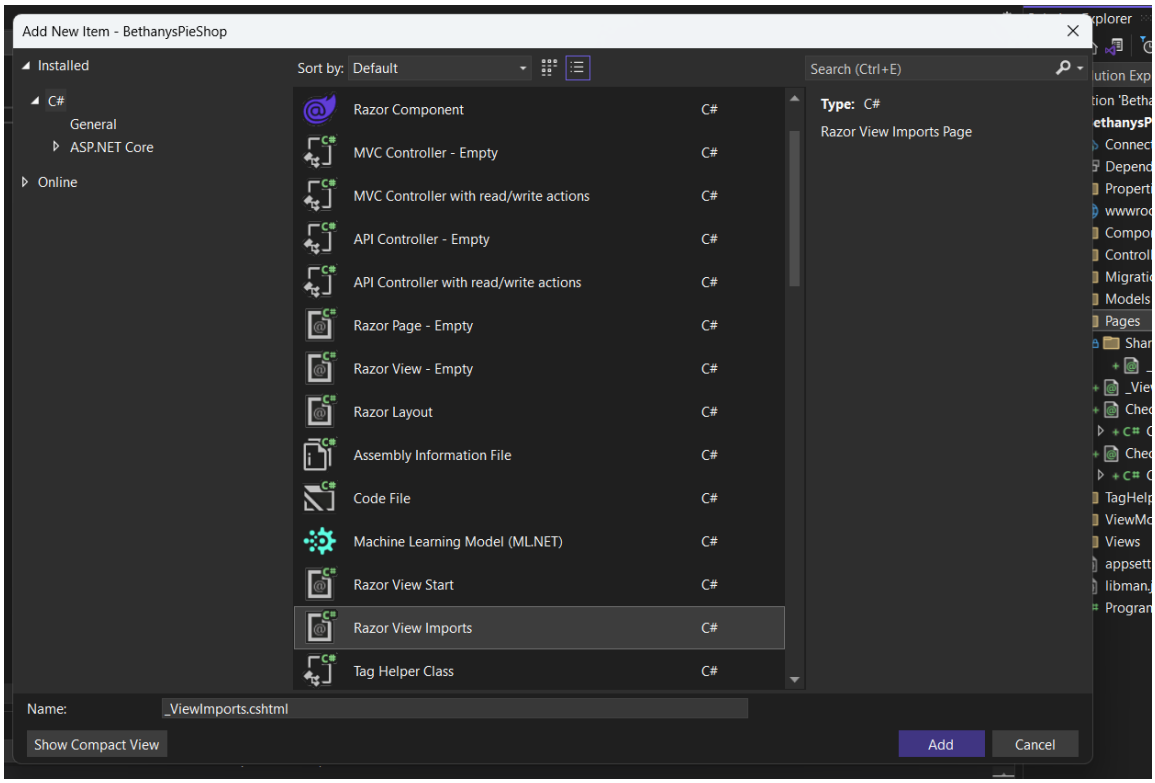

Le pagine devono sapere di questo _PageLayout.cshtml quindi creiamo nella Root che nel nostro caso è Pages altri files:

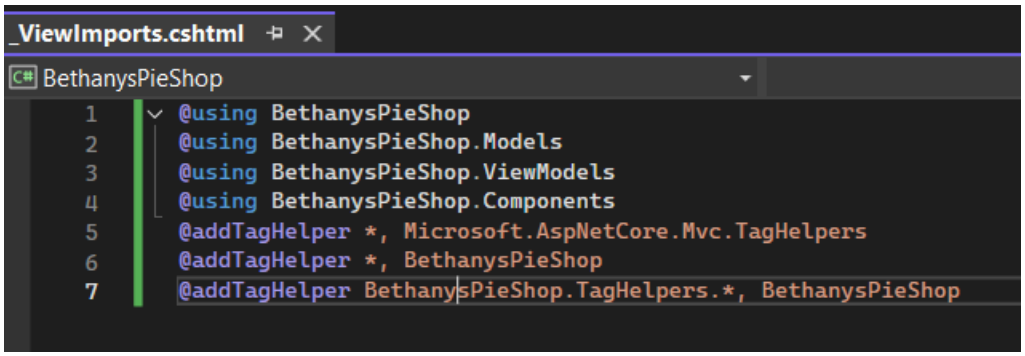
_ViewStart:




Ora inseriamo un _ViewImports:



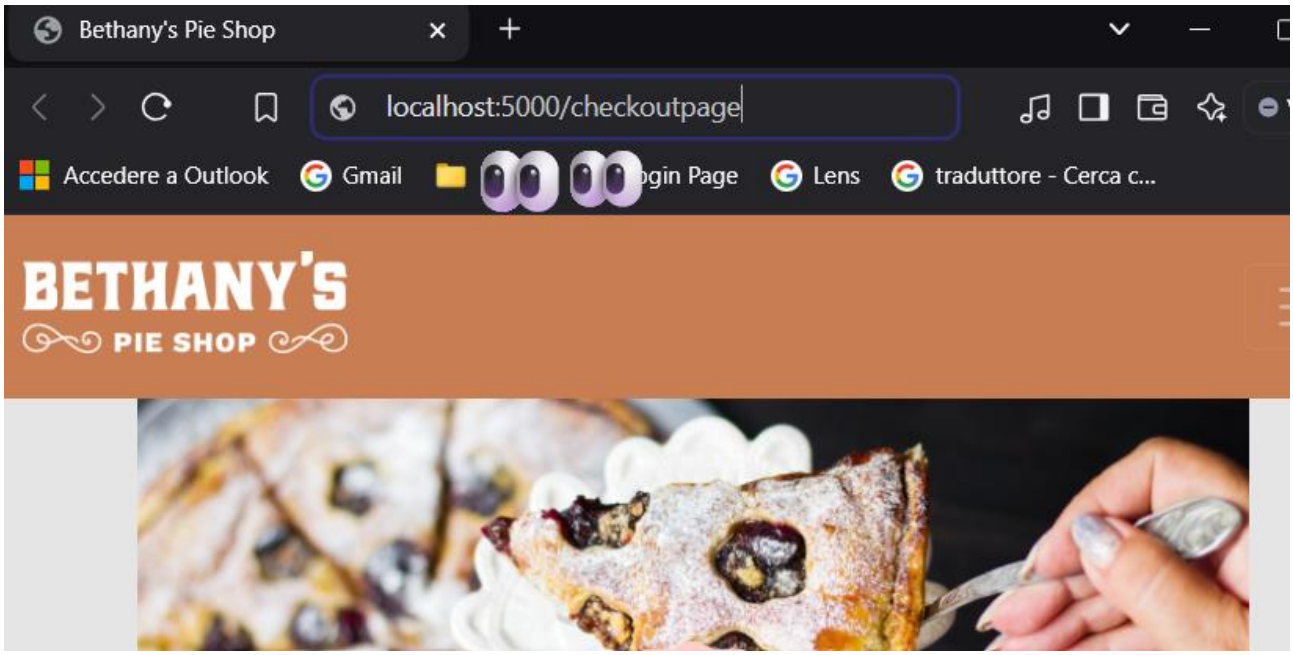




Anche qui abbiamo copiato dall'altro import che avevamo.

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

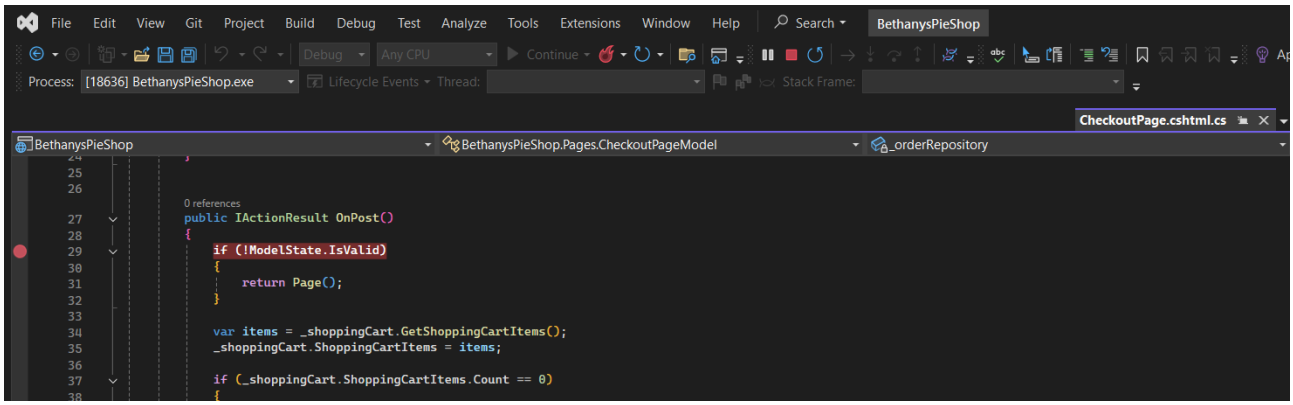
```razor
@page
@model BethanysPieShop.Pages.CheckoutPageModel

<form 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="Order.FirstName" class="form-label"></label>
                <input asp-for="Order.FirstName" class="form-control" />
                <span asp-validation-for="Order.FirstName" class="text-danger"></span>
            </div>

            <div class="col-12">
                <label asp-for="Order.LastName" class="form-label"></label>
                <input asp-for="Order.LastName" class="form-control" />
                <span asp-validation-for="Order.LastName" class="text-danger"></span>
            </div>

            <div class="col-12">
```

Ora eseguiamo:

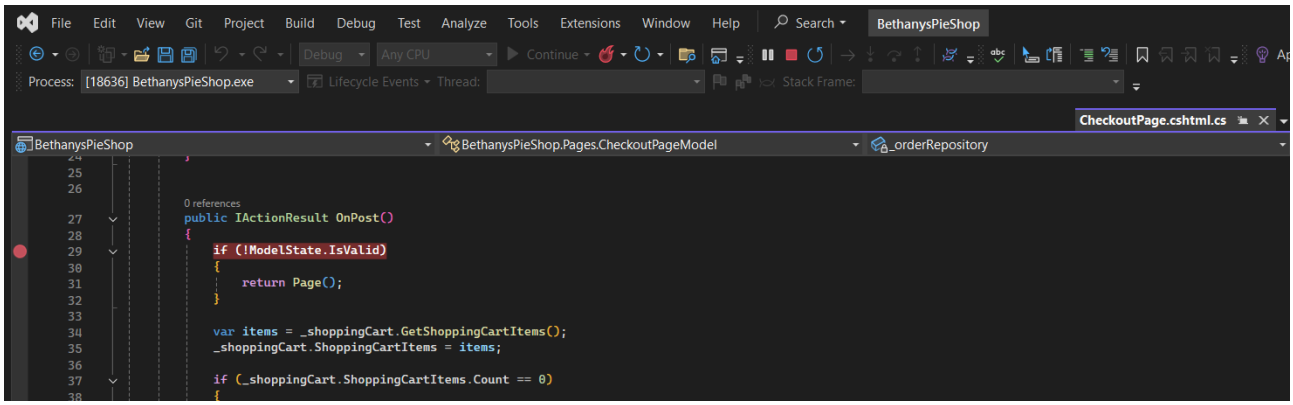



Osservazione: Ci basta semplicemente inserire checkoutpage per giungere alla pagina di checkout senza specificare la cartella Pages, questa è l'impostazione predefinita, questa è la radice.

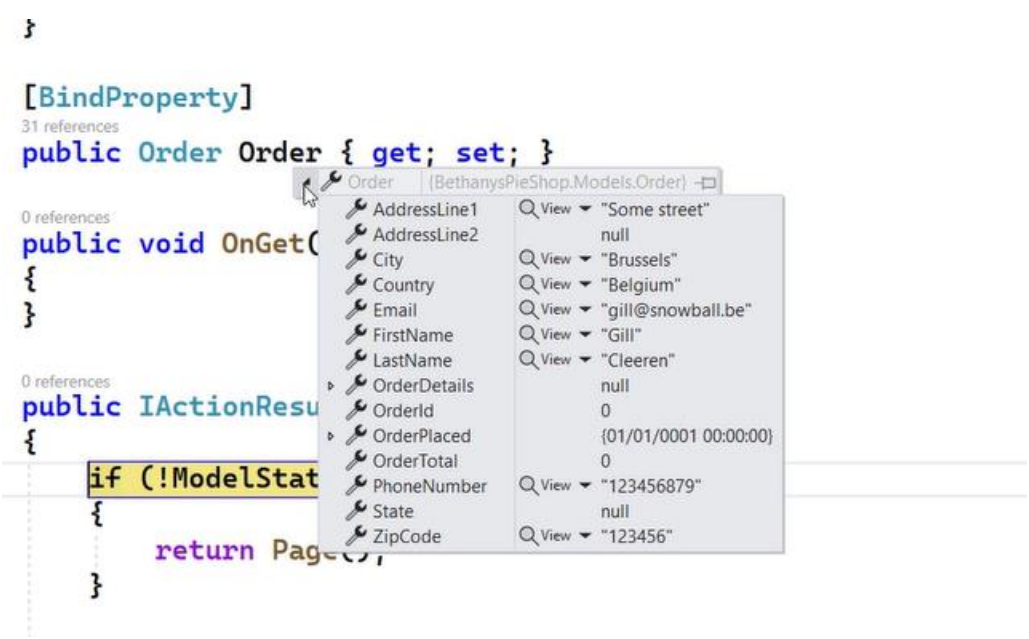
Se inseriamo un breakpoint qui:




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:




Una volta fatto possiamo vedere le associazioni:

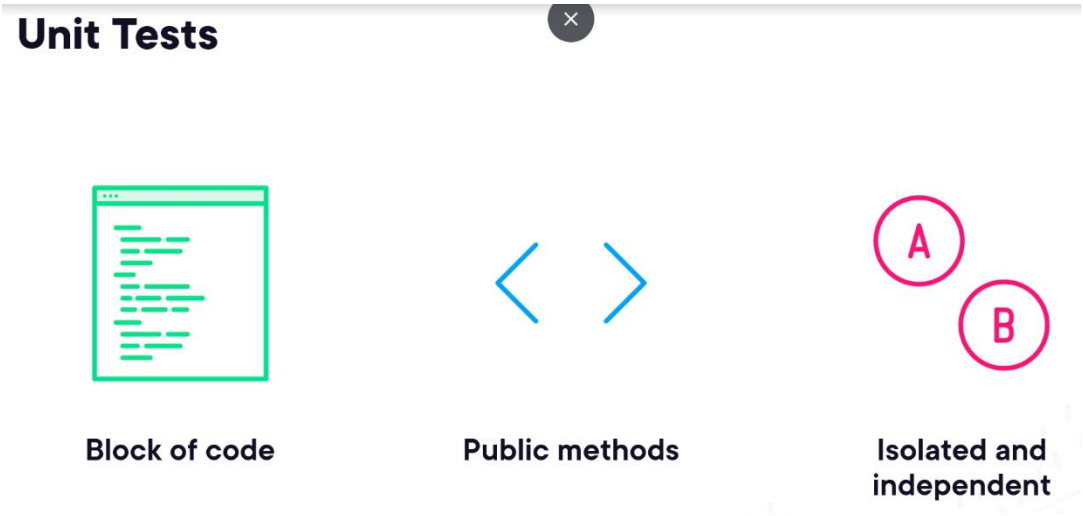


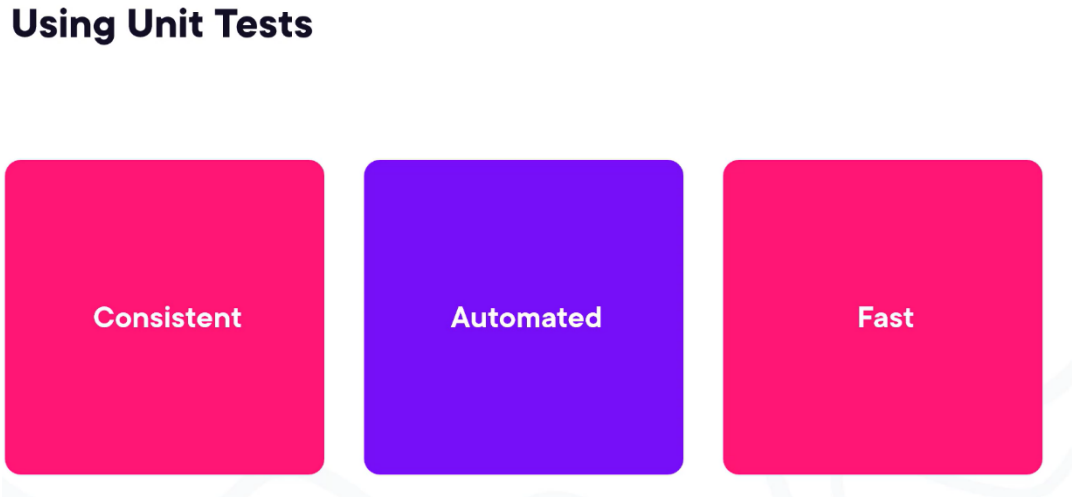

Questo è tutto sull'approccio con Razor Pages.

# 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*




**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 Frekwork

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

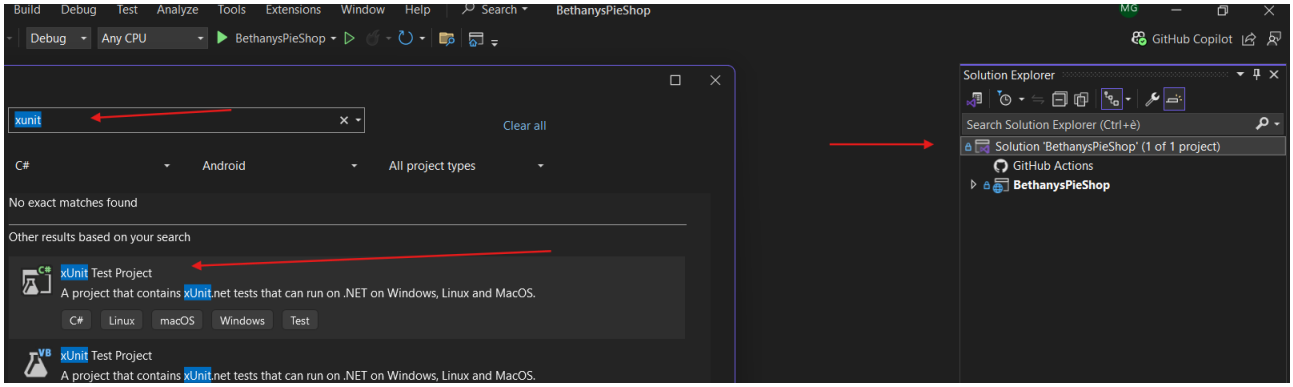
Esempio:

## A Simple Unit Test

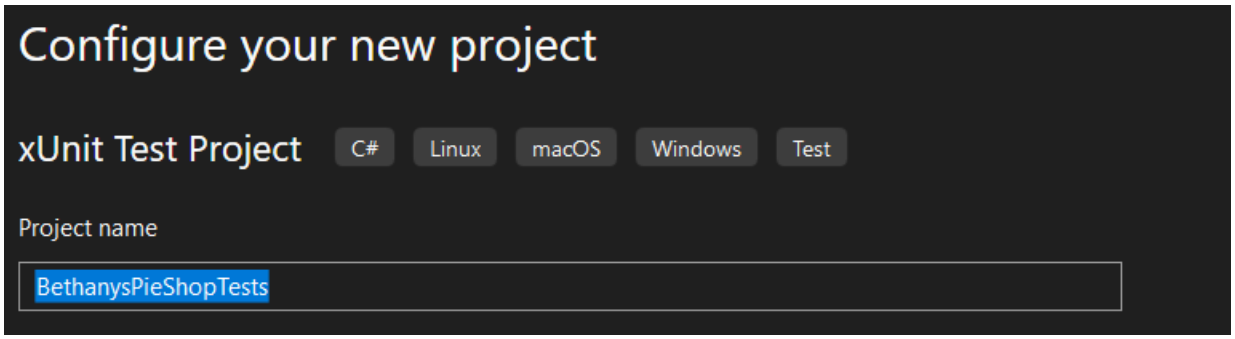
```
public void CanUpdatePiePrice()
{
    // Arrange
    var pie =
        new Pie { Name = "Sample pie", Price = 12.95M };

    // Act
    pie.Price = 20M;

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

**Demo: Testing the Controllers**

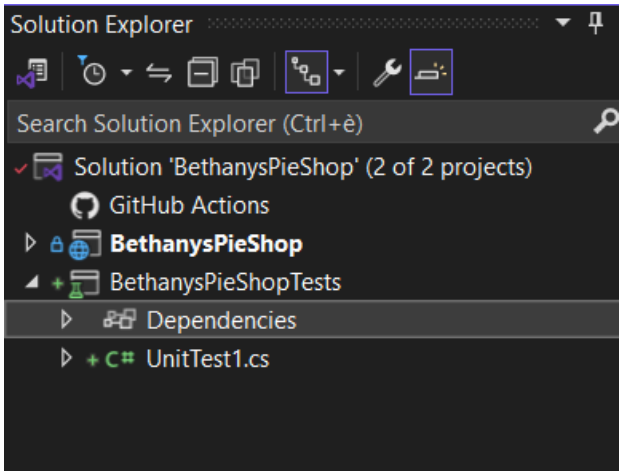
Aggiungiamo un progetto di test unitario alla nostra soluzione.
Click destro sulla soluzione > New project > digitiamo xUnit Test Project

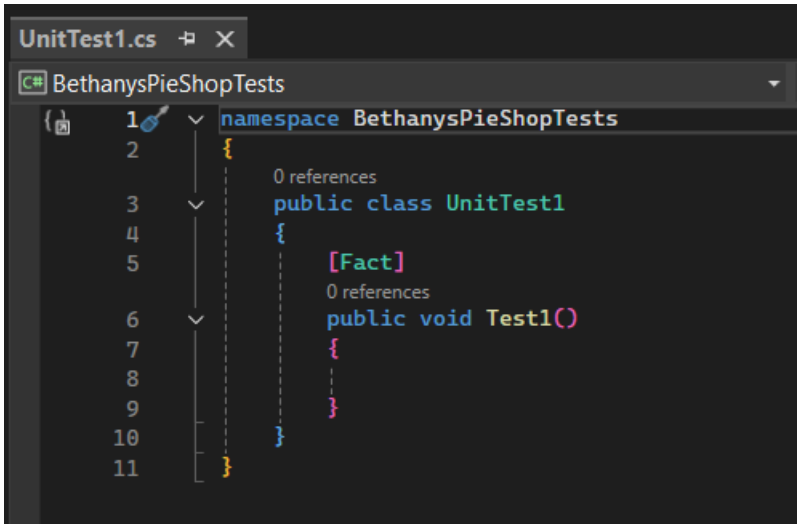



Lo chiamiamo nel seguente modo:

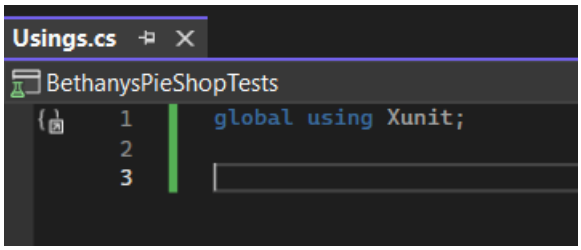



Dopo aver scelto Dot Net 8, di default avremo ciò:

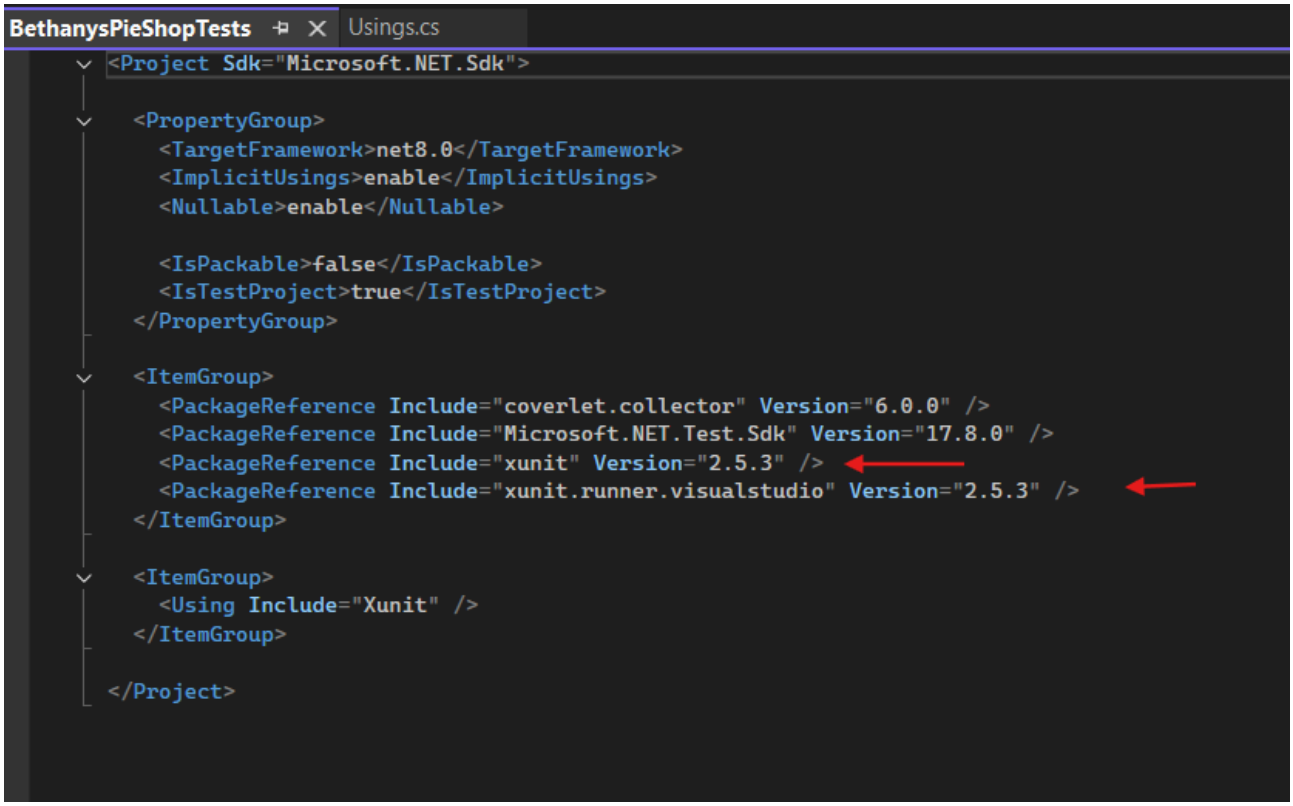



Cancelliamo questa classe e creiamo Usings.cs:




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




Lo modifichaimo nel seguente modo:

BethanysPieShopTests.csproj | Usings.cs

```
<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="..\BethanysPieShop\BethanysPieShop.csproj" />
    </ItemGroup>

</Project>
```

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

BethanysPieShopTests.csproj | Usings.cs

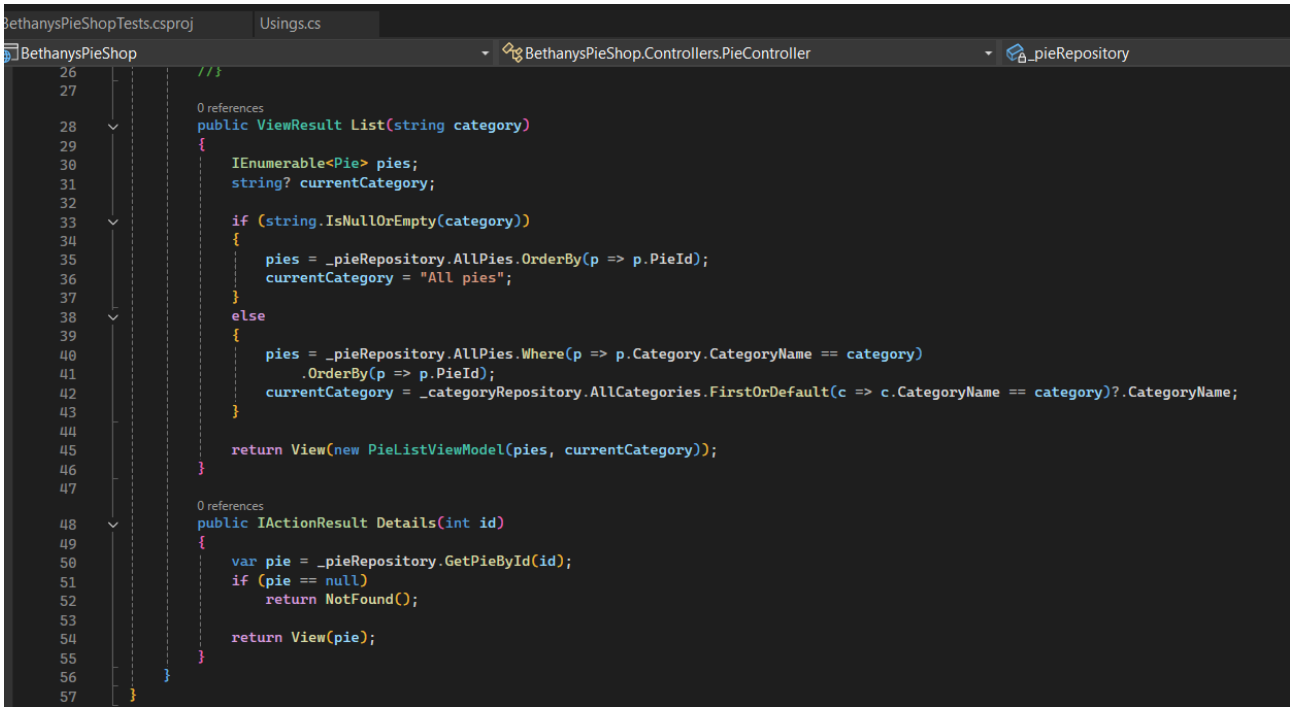
BethanysPieShop | BethanysPieShop.Controllers.PieController

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

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

        0 references
        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
        //    return View(piesListViewModel);
        //}

        0 references
        public ViewResult List(string category)
        {
            IEnumerable<Pie> pies;
            string? currentCategory;

            if (string.IsNullOrEmpty(category))
            {
```

87 % | No issues found

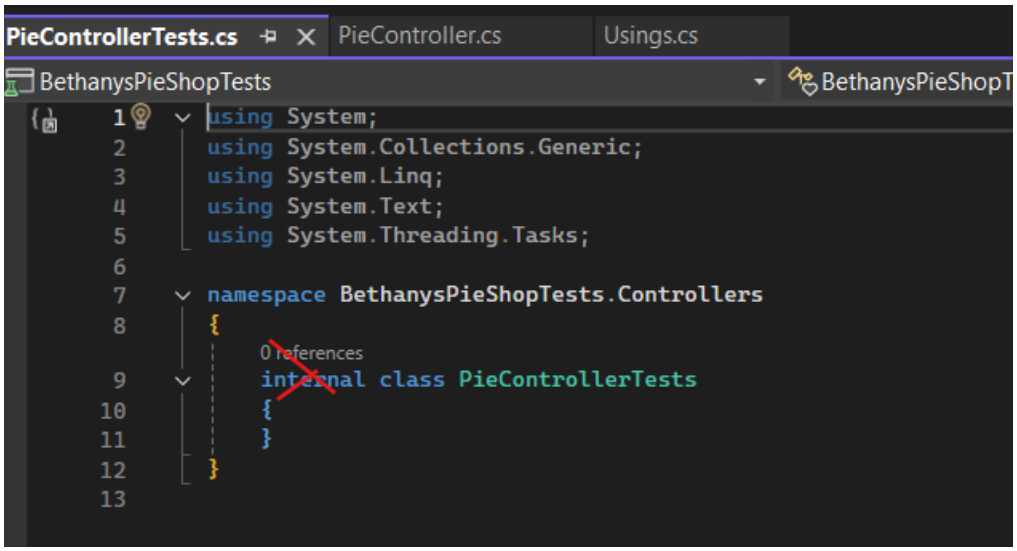
Ora creeremo una versione simulata di questi e li inietteremo nel PieController quando vogliamo testarlo.

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



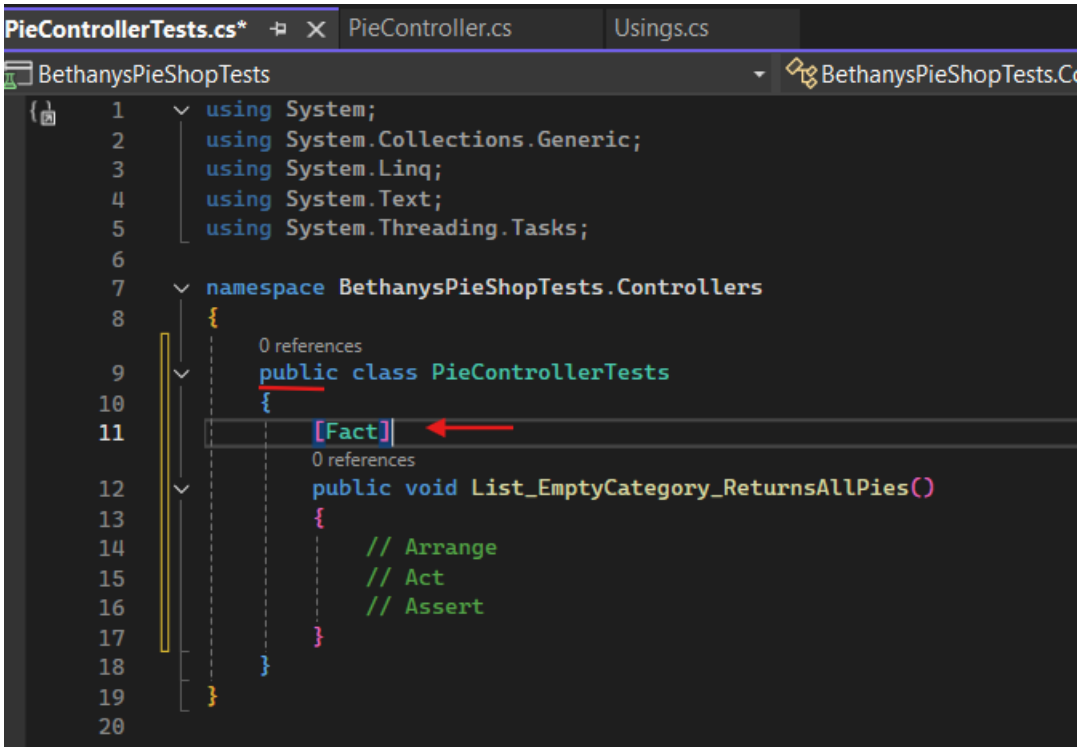

Dopo averlo esaminato andiamo nel nostro progetto di test "BethanysPieShop**Tests"** e creiamo la cartella Controllers

Al suo interno creiamo PieControllerTests.cs




Eliminiamo internal e scriviamo public

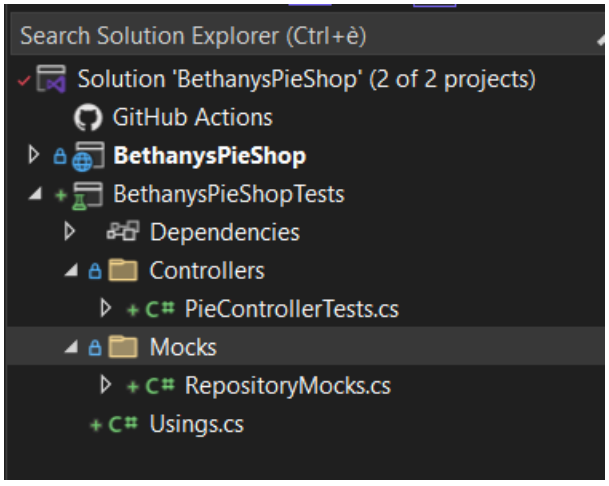
Per prima cosa testiamo il metodo List che simula il caso in cui non passiamo nessuna Categoria come parametro, il metodo del test deve essere parlante.



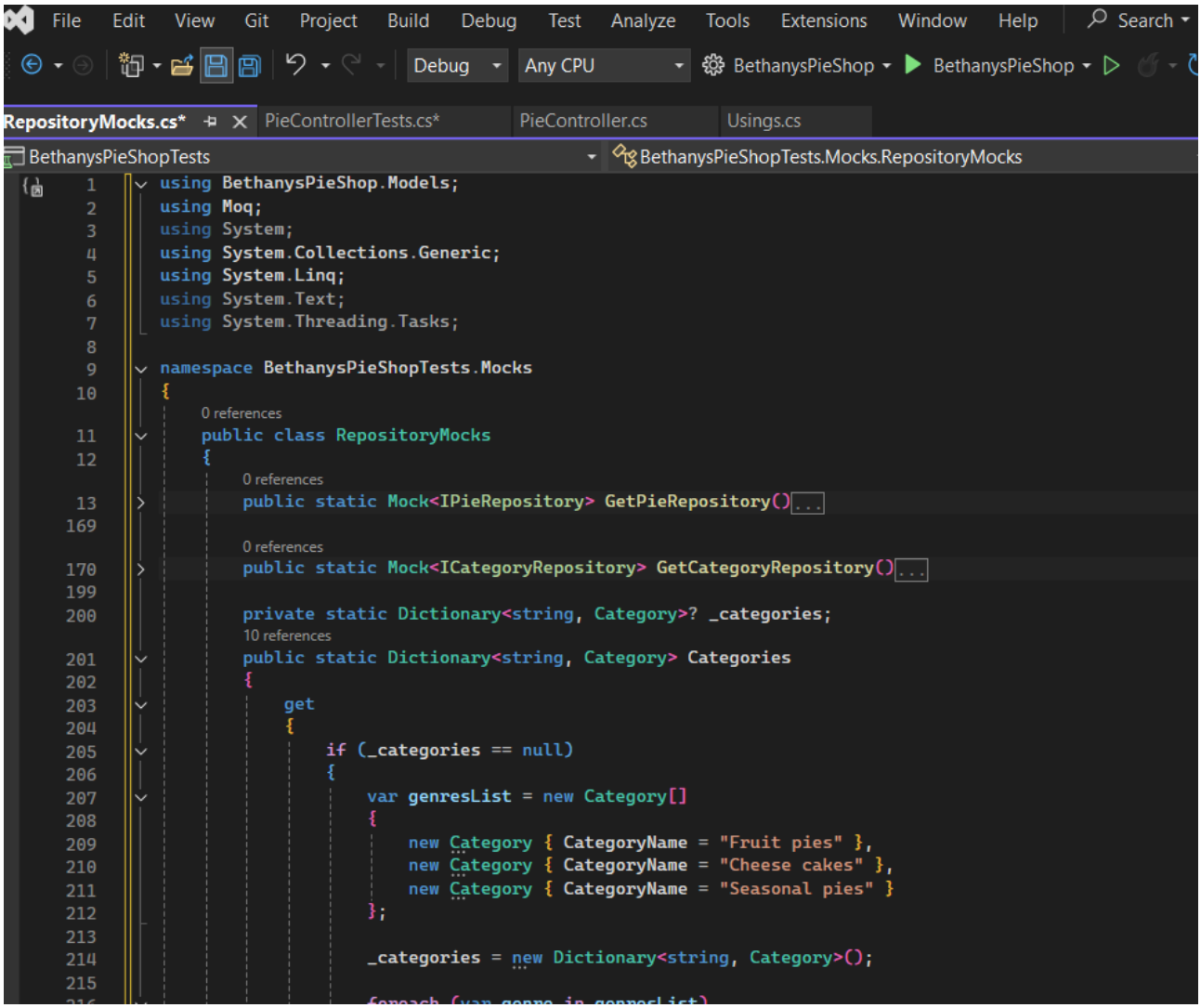

Osservazione: Aggiungiamo l'attributo Fact per far si che questo metodo è un test unitario.

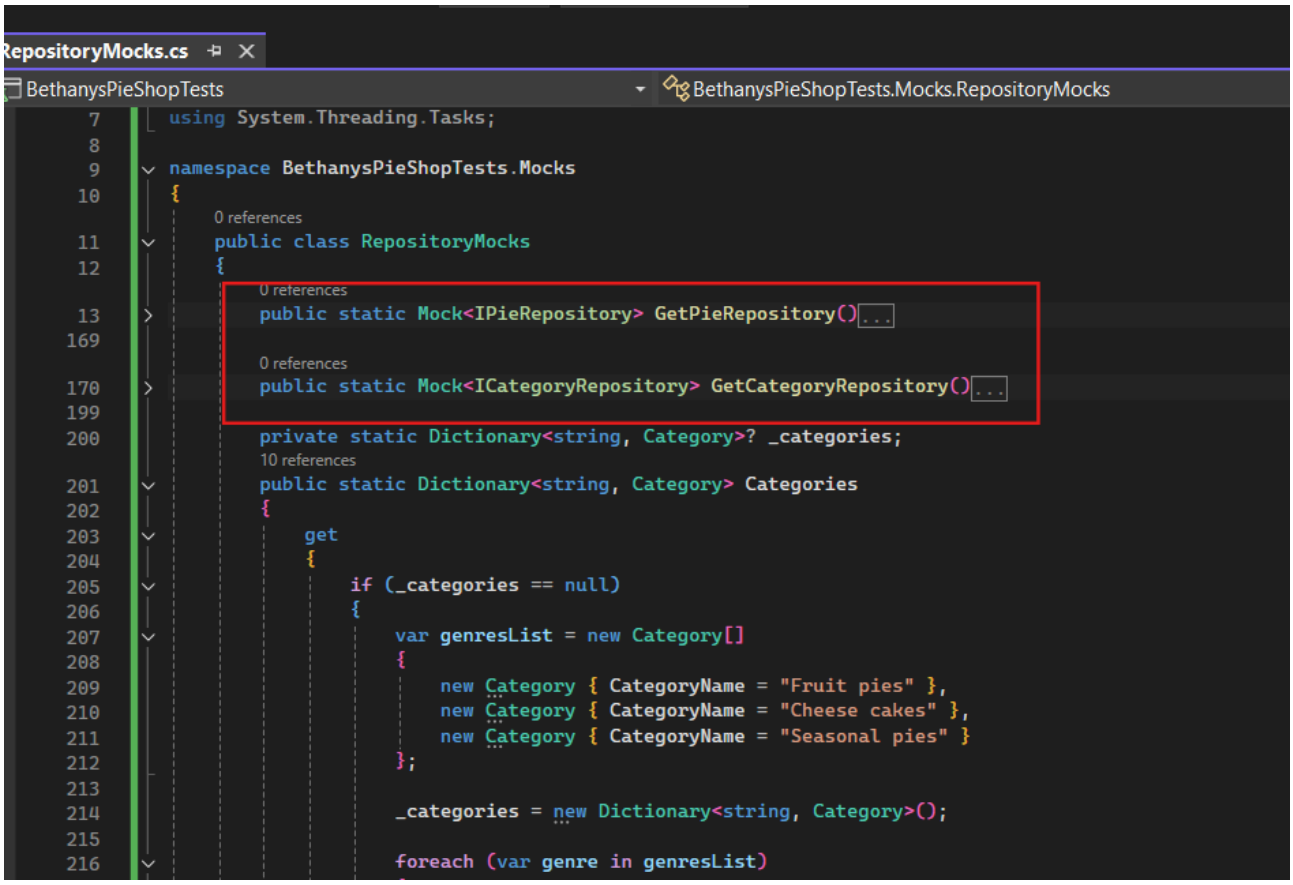
Ci ricordiamo che ci servono un PieRepository e un CatergoryRepository.
Non usiamo quelli veri ma una versione fittizia un mock.

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



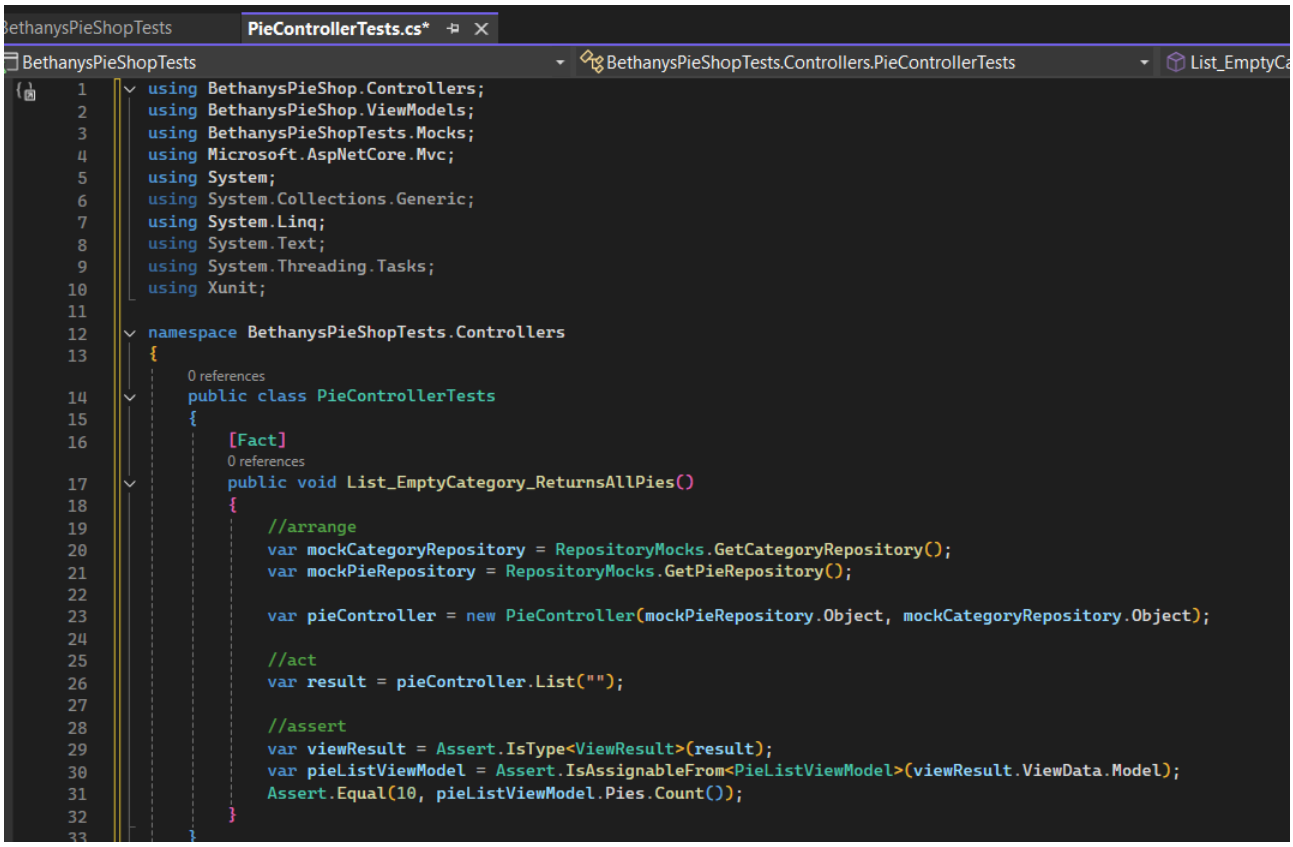
Incolliamo i nostri snippets:






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

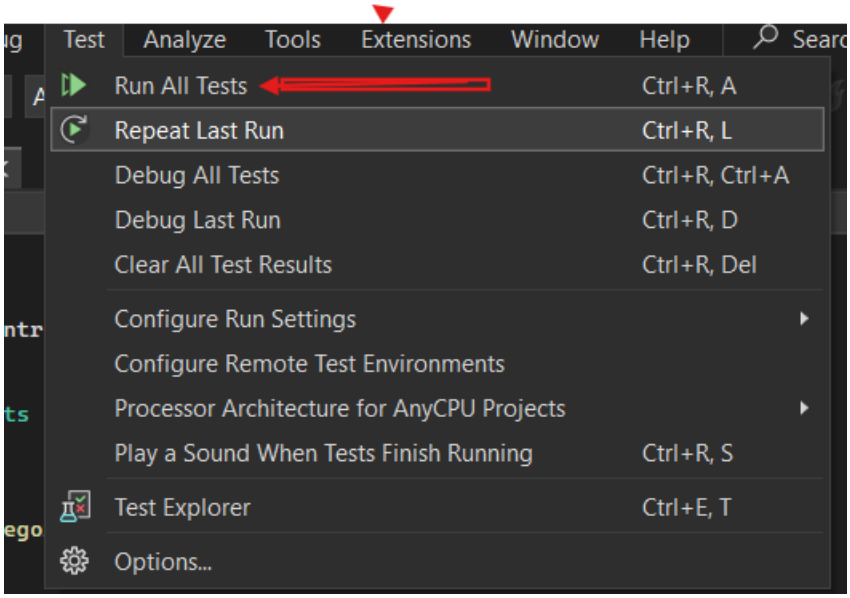
Torniamo al nostro PieControllerTest.cs e lo completiamo nel seguente modo:



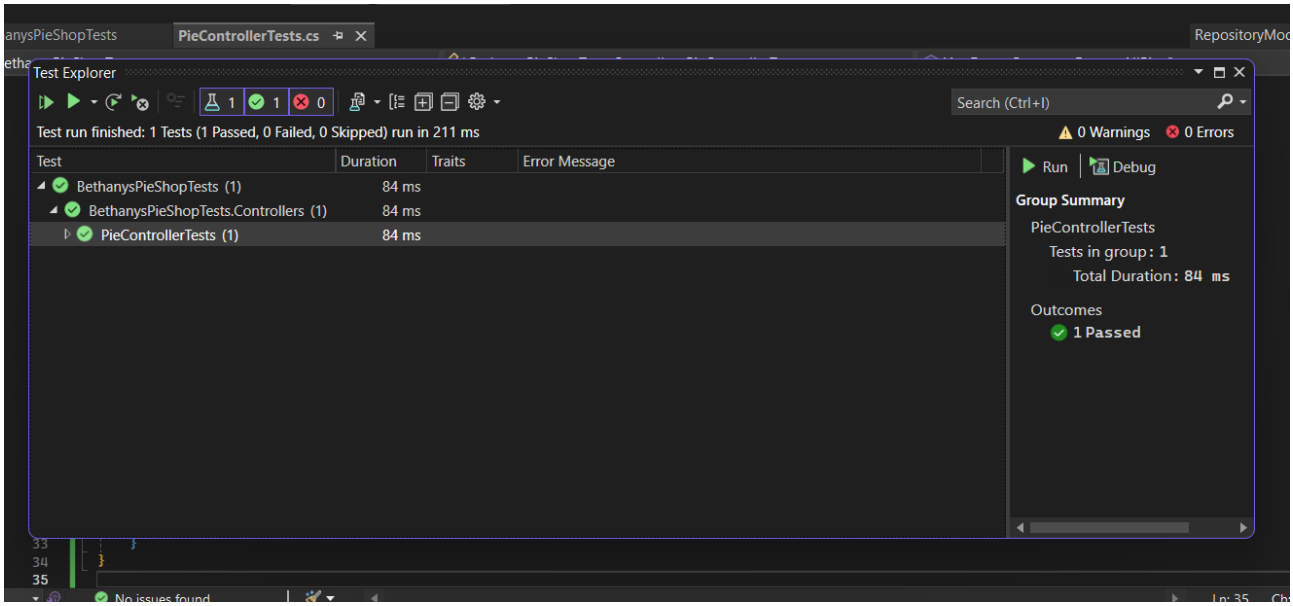

Osservazione sugli assert:
riga 29: Verifichiamo che quanto restituito sia di tipo viewResult (in quanto in PieContoller.cs ritorniamo una vista)
riga 30: Verifichiamo che ViewRsult.ViewData.Model quindi il modello che abbiamo passato (ovvero il parametro passato al return View del metodo List di PieController.cs)

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

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




Output:

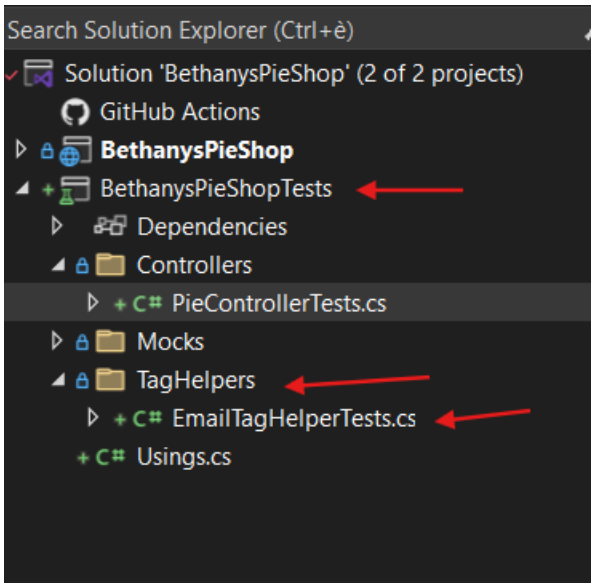



**Demo: Testing Tag Helper**

**Ricordandoci EmailTagHelper presente nella cartella TagHelpers**

```
BethanysPieShop                                   BethanysPieShop.TagHelpers.EmailTagHelper
 1  using Microsoft.AspNetCore.Razor.TagHelpers;
 2
 3  namespace BethanysPieShop.TagHelpers
 4  {
        2 references
 5      public class EmailTagHelper: TagHelper
 6      {
            2 references
 7          public string? Address { get; set; }
            2 references
 8          public string? Content { get; set; }
 9
            0 references
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:

EmailTagHelperTests.cs

BethanysPieShopTests

BethanysPieShopTests.TagHelpers.EmailTagHelp

```
using BethanysPieShop.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Xunit;

namespace BethanysPieShopTests.TagHelpers
{
    0 references
    public class EmailTagHelperTests
    {
        [Fact]
        0 references
        public void Generates_Email_Link()
        {
            EmailTagHelper emailTagHelper = new EmailTagHelper() { Address = "test@bethanyspieshop.com", Content = "Email" }; ;

            var tagHelperContext = new TagHelperContext(
                new TagHelperAttributeList(),
                new Dictionary<object, object>(), string.Empty);

            var content = new Mock<TagHelperContent>();

            var tagHelperOutput = new TagHelperOutput("a",
                new TagHelperAttributeList(),
                (cache, encoder) => Task.FromResult(content.Object));

            // Act
            emailTagHelper.Process(tagHelperContext, tagHelperOutput);

            Assert.Equal("Email", tagHelperOutput.Content.GetContent());
            Assert.Equal("a", tagHelperOutput.TagName);
            Assert.Equal("mailto:test@bethanyspieshop.com", tagHelperOutput.Attributes[0].Value);
        }

    }
}
```

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).

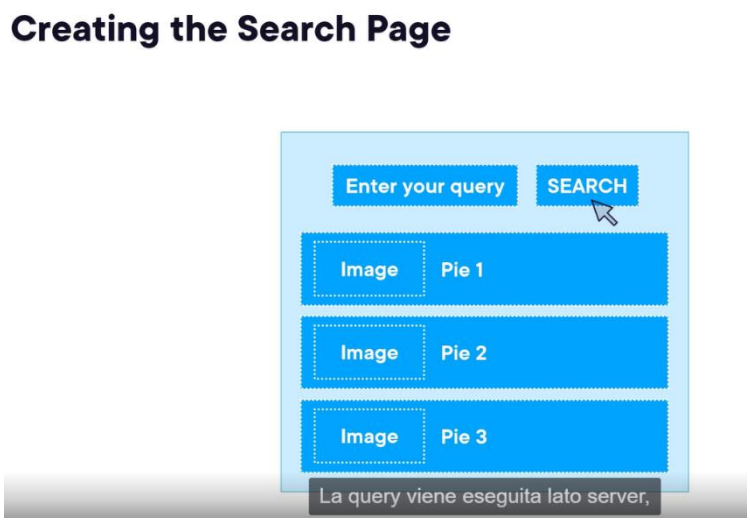
In Assert verifichiamo quindi il contenuto, 'a', e il "mailto"

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

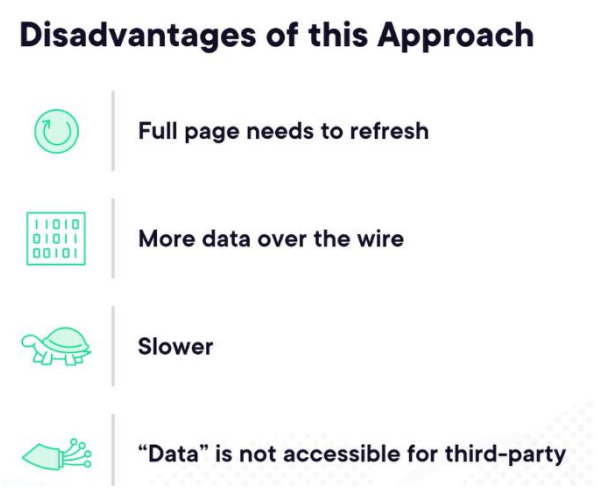
# Making the Site Interactive

- 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

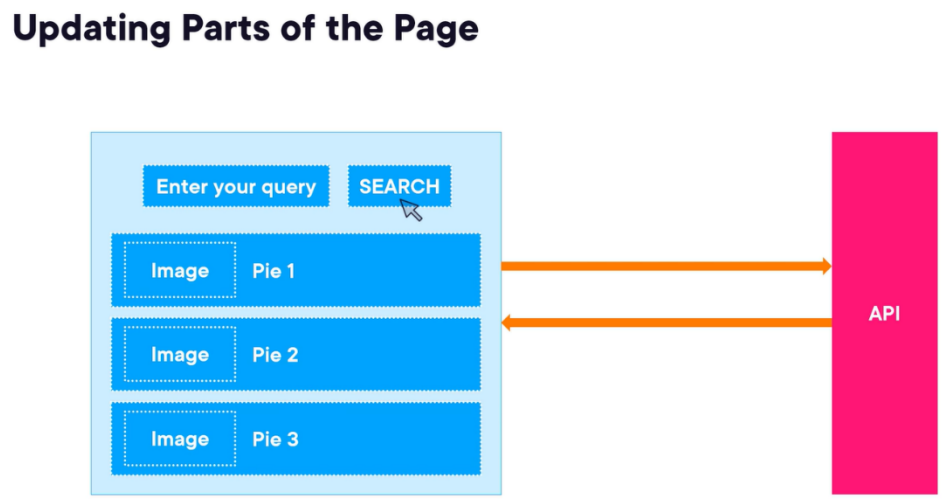
**Searching Using JavaScript and an ASP.NET Core API**




Però questo approccio presenta alcuni svantaggi:




Quindi creeremo un'API, fondamentalmente un servizio, utilizzando ASP.NET Core e che non restituirà HTML, ma solo i dati:



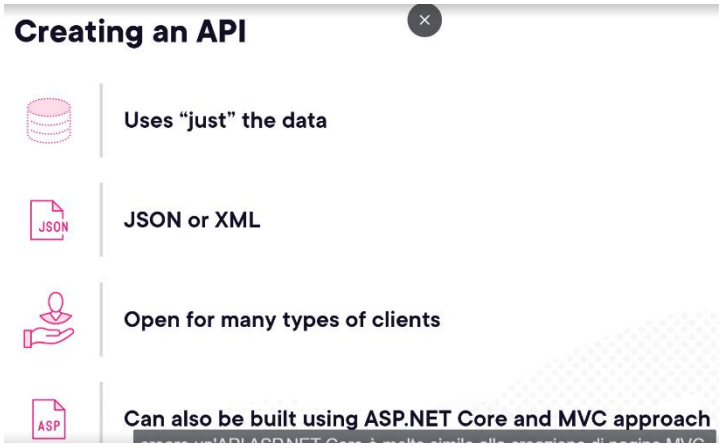

Utilizzando JavaScript quindi codice lato client, questo comunicherà con l'API per attivare l'esecuzione della query di ricerca sul server.

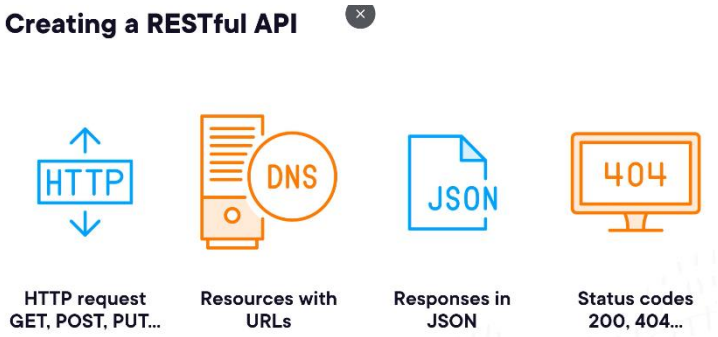
L'Api restituirà quindi semplicemente i dati.

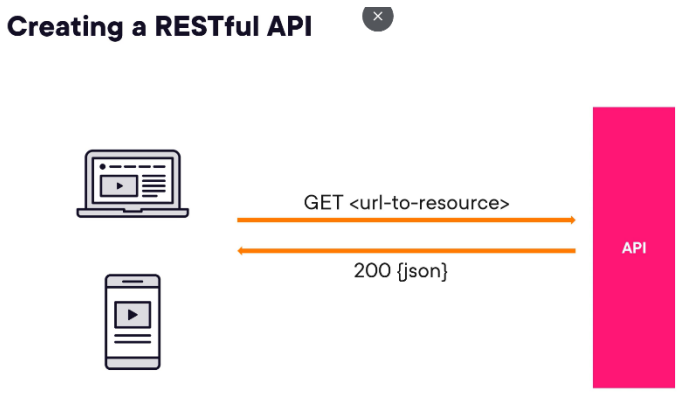
Effettueremo un aggiornamento parziale della pagina invece di eseguire nuovamente il rendering dell'intera pagina (grazie a JQuery)

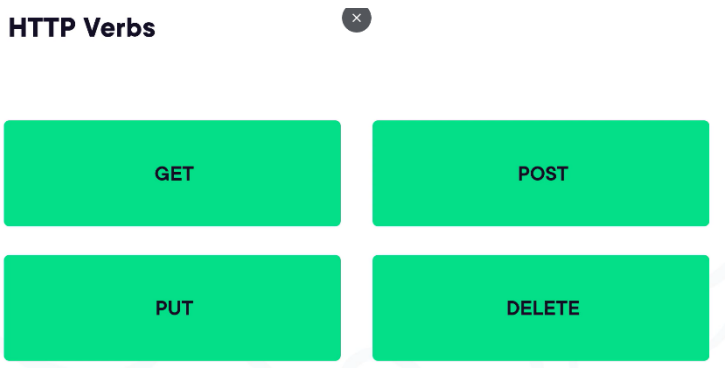
Quindi quello che faremo è:
1. Creare un API che restituisca solo i dati
2. Includere JQuery per visualizzare i risultati

**Creating an ASP.NET Core RESTful API**


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
HTTP
DNS
JSON
404
HTTP request
GET, POST, PUT...
Resources with
URLs
Responses in
JSON
Status codes
200, 404...


Creating a RESTful API
GET <url-to-resource>
200 {json}
API


HTTP Verbs
GET
POST
PUT
DELETE

## JSON Response


### 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

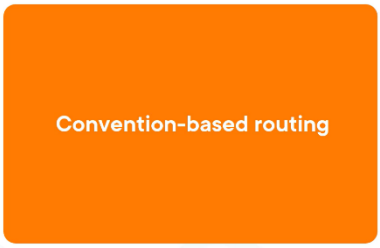
Not needed separately if already done for regular MVC

```
public class PieController : ControllerBase
{

}
```

**Creating a Controller**

ControllerBase adds support for access to HttpContext, Request...

**Routing Options in ASP.NET Core**





```
[Route("api/[controller]")]
public class PieController : ControllerBase
{

}
```

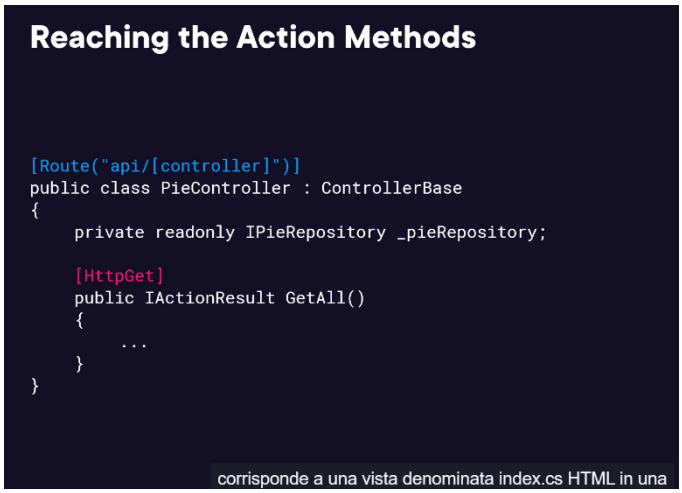
**Using the Route Attribute**

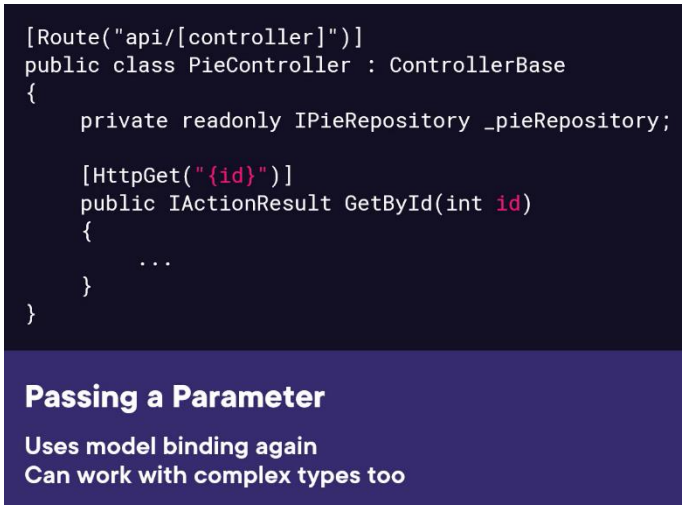
Accessible via /api/search

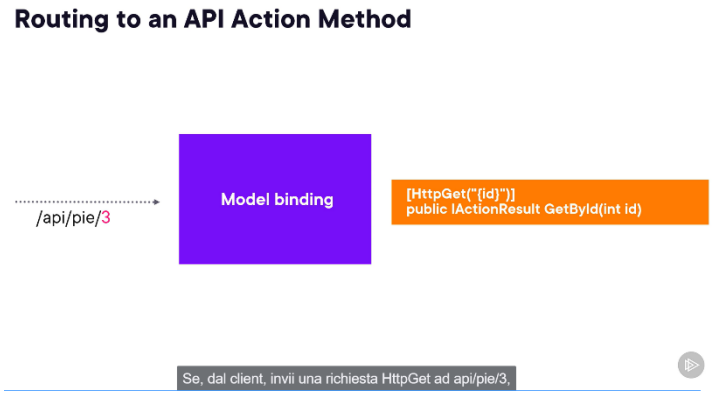

```
[Route("api/[controller]")]
public class PieController : ControllerBase
{

}
```

**Using the Route Attribute**

Accessible via /api/search



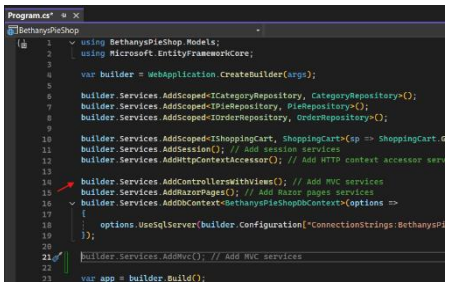



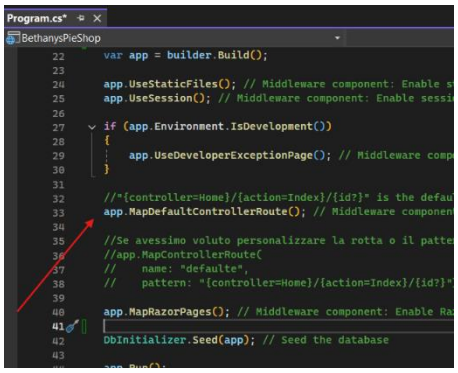




**Demo: Setting up the API**

Ora includiamo una API.

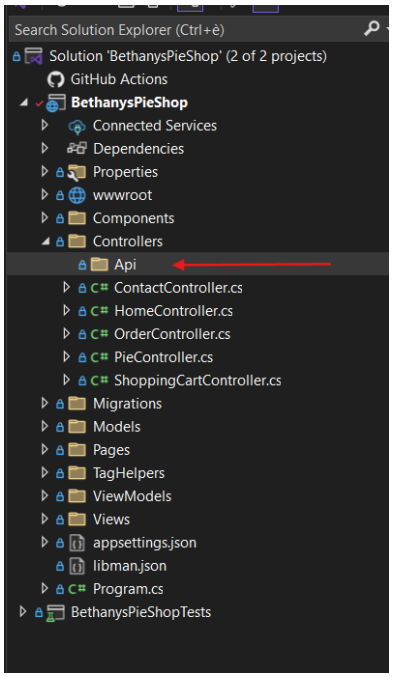
Come dimostrato nella teoria, dovremmo aggiungere il servizio Controller, nel nostro caso non è necessario in quanto ne abbiamo giù uno che include le viste, inoltre abbiamo già anche un supporto per il Routing:



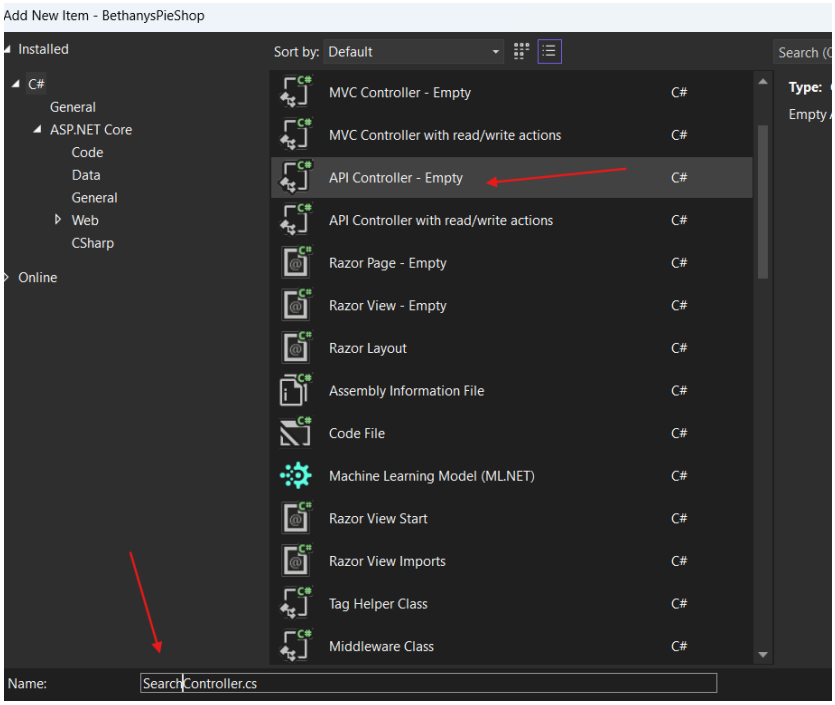



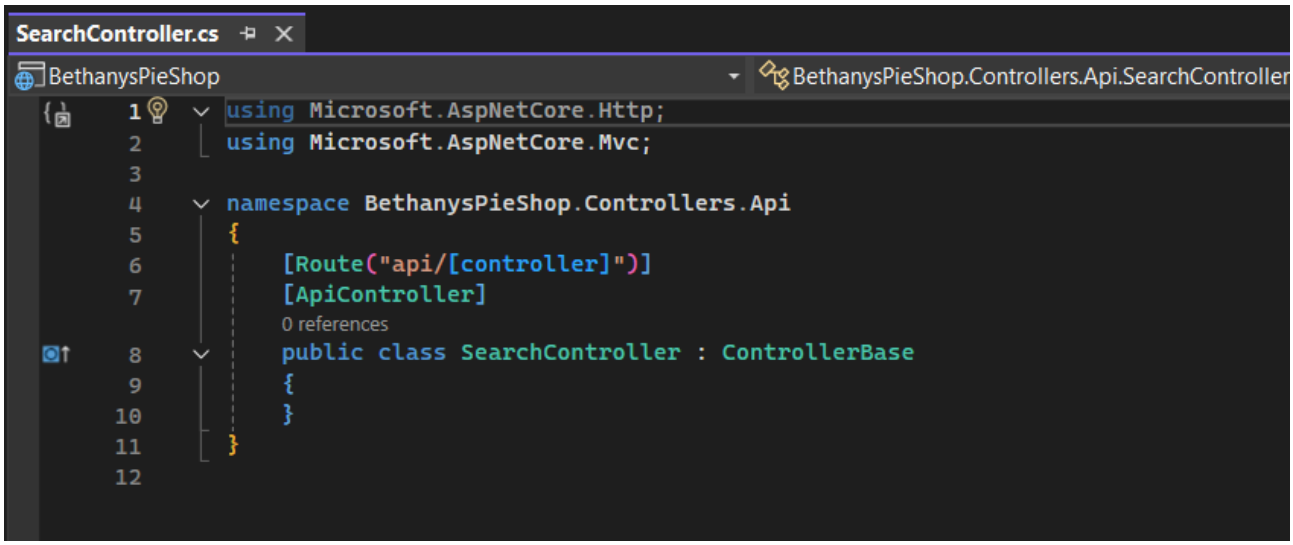

Dobbiamo quindi aggiungere un Controller che sia differente da quello MVC, quindi un controller che si occupi solo della logica API.

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



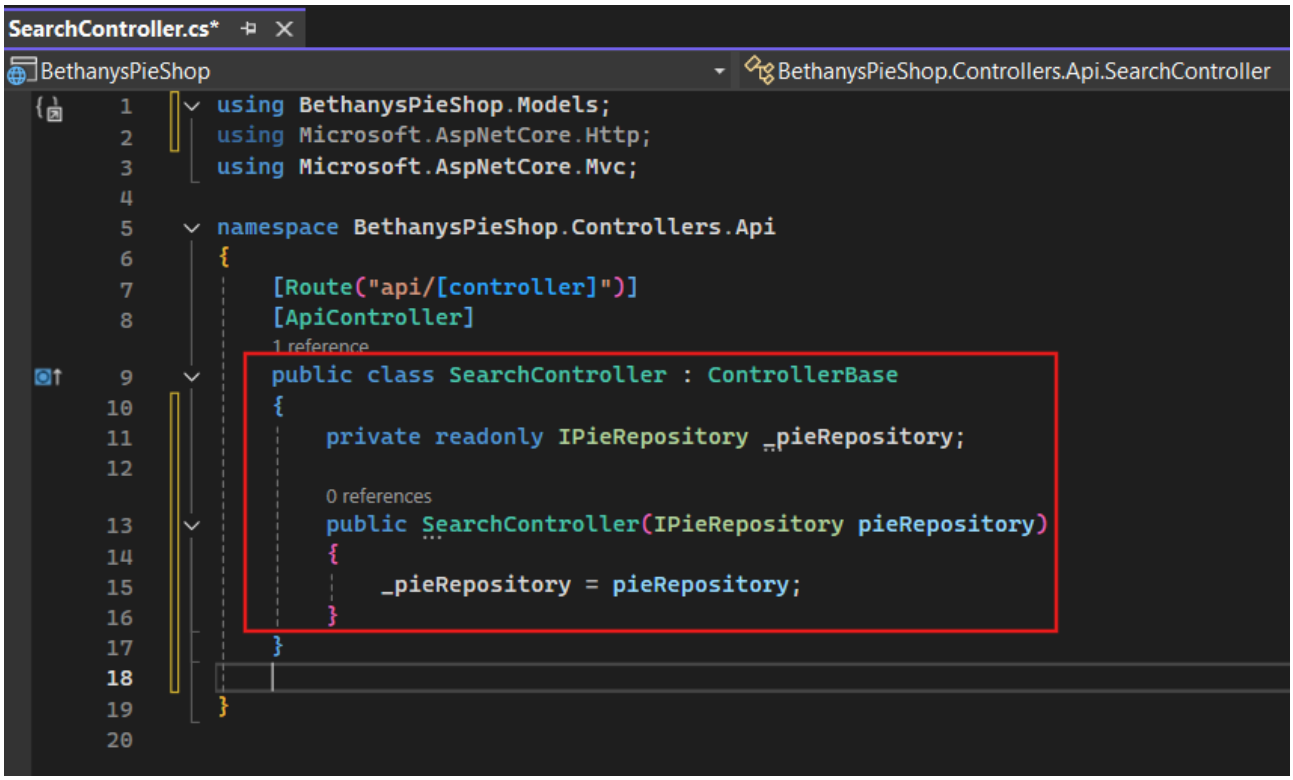

Riga 6: indica che stiamo utilizzando un percorso specifico per instradare le richieste a questo controller.

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.

Detto ciò, inseriamo le D.I.:




Ed ora l'action method:

SearchController.cs

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]
        1 reference
9       public class SearchController : ControllerBase
10      {
11          private readonly IPieRepository _pieRepository;
12
            0 references
13          public SearchController(IPieRepository pieRepository)
14          {
15              _pieRepository = pieRepository;
16          }
17      }
18
19      [HttpGet]
        0 references
20      public IActionResult GetAll()
21      {
22
23      }
24
25      [HttpGet("{id}")]
        0 references
26      public IActionResult GetById(int id)
27      {
28
29      }
30  }
31
```

(Lasciamo questo script cosi momentaneamente)

**Creating the API Responses**

SearchController.cs

BethanysPieShop — BethanysPieShop.Controllers.Api.SearchController

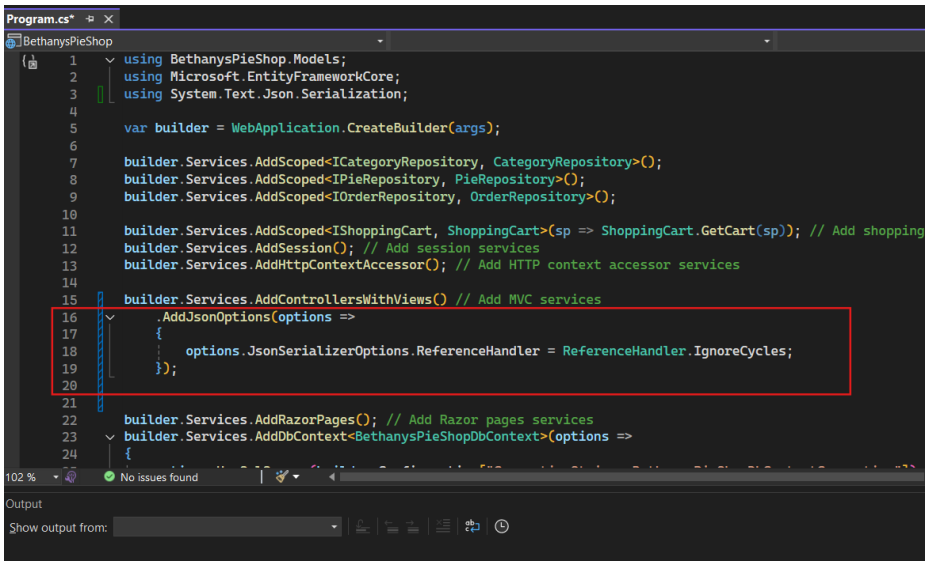
```
7       [Route("api/[controller]")]
8       [ApiController]
        1 reference
9       public class SearchController : ControllerBase
10      {
11          private readonly IPieRepository _pieRepository;
12
            0 references
13          public SearchController(IPieRepository pieRepository)
14          {
15              _pieRepository = pieRepository;
16          }
17
18          [HttpGet]
            0 references
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}")]
            0 references
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  }
```

77 % — No issues found

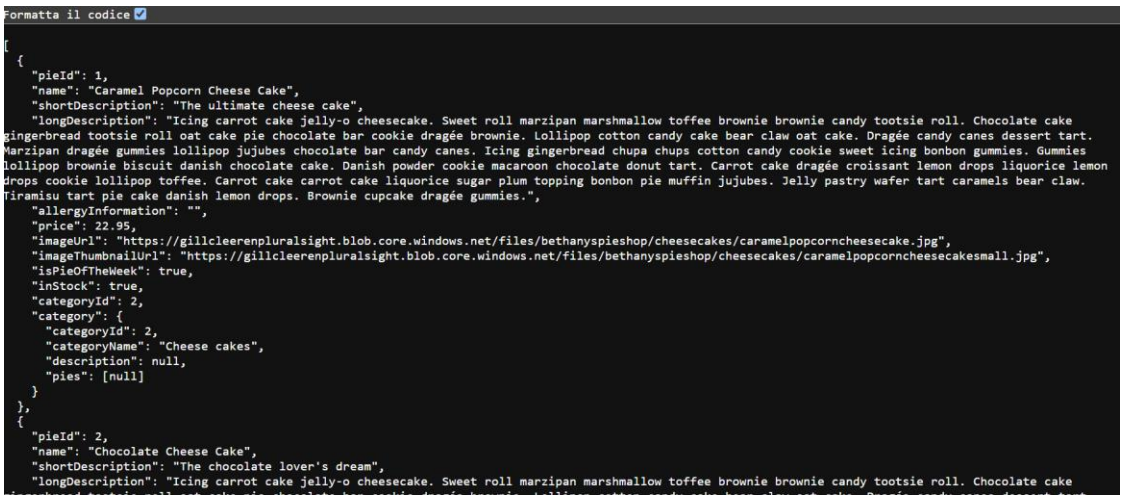
Output

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 rifierimento alle torte e cosi 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:



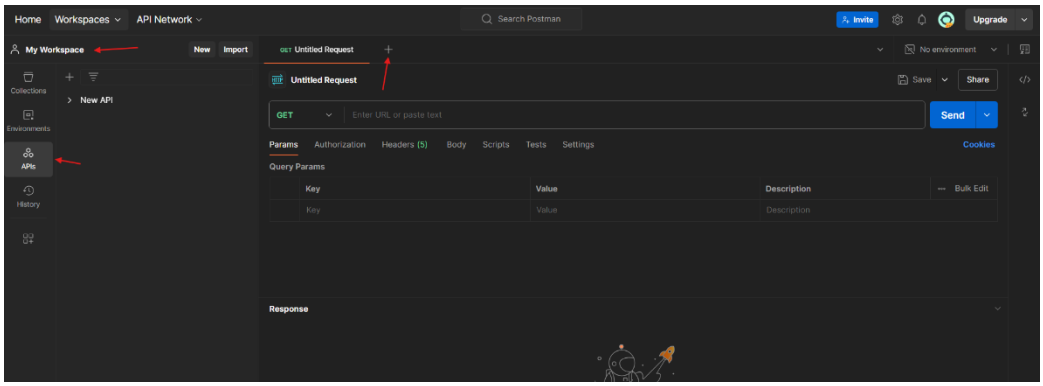

Okay, salviamo, avviamo e ora giungiamo all'url http://localhost:5000/api/search
Verrà mostrato il Json:

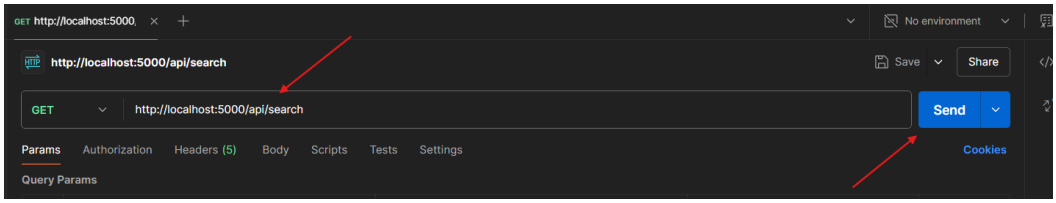


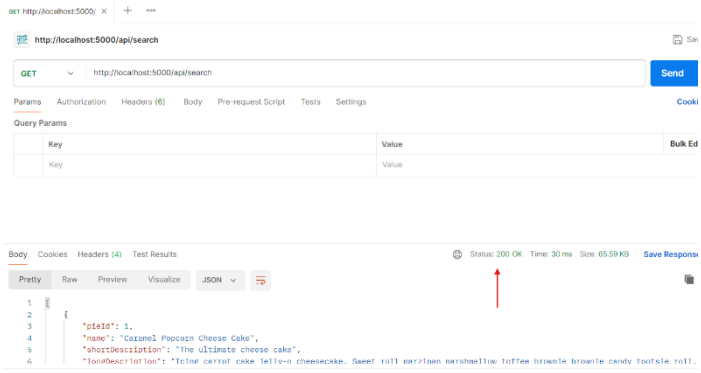

Volendo potrei scrivere http://localhost:5000/api/search/1 e mi restituirà solo quella torta.

Per testare un API non utilizziamo un browser, ma **Postman**

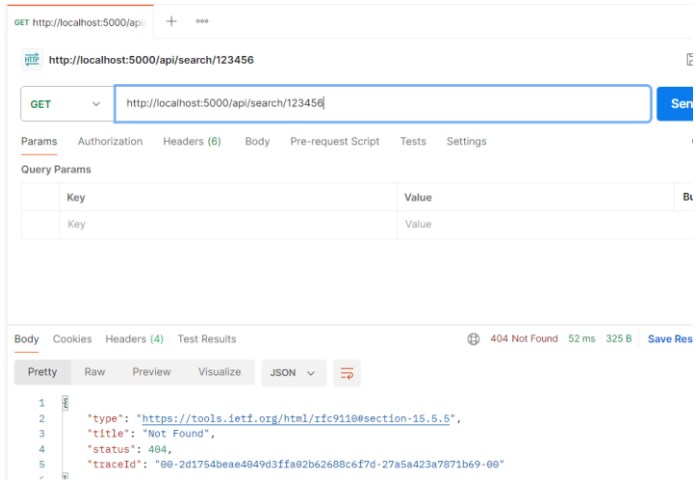
MyWorkspace > APIs > +



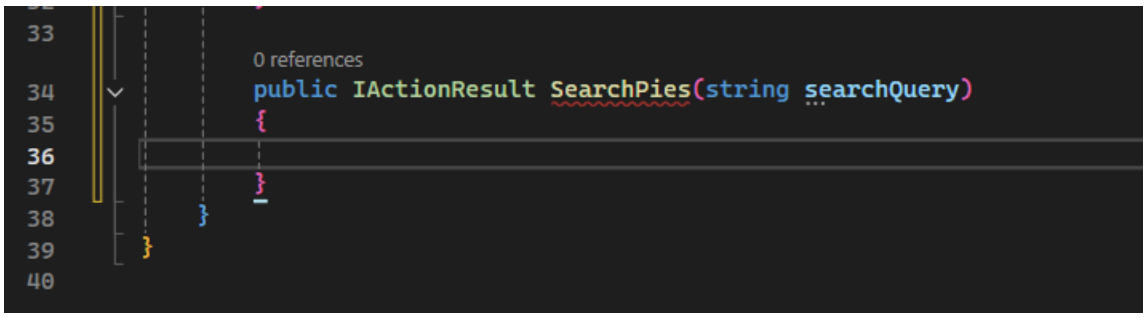




Mentre se inseriamo una torta che non esiste:

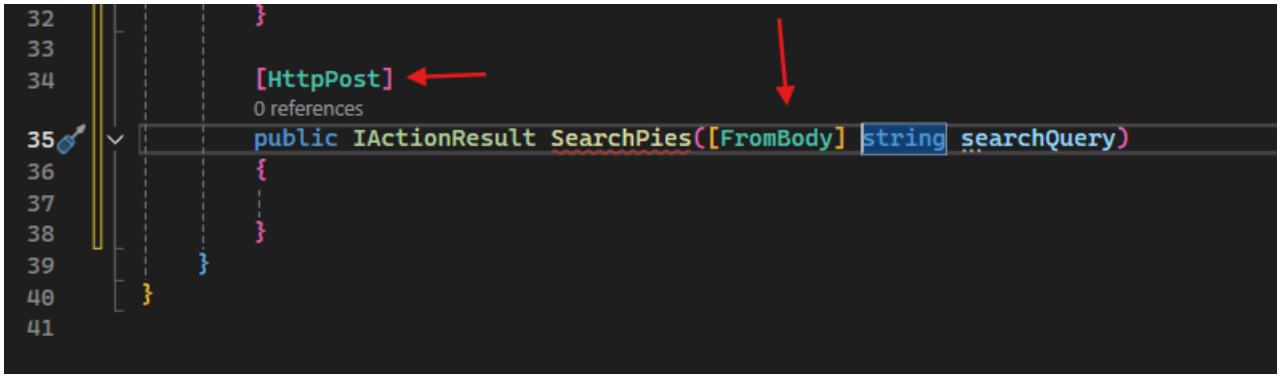


Ora dobbiamo ancora implementare la funzionalità di ricerca

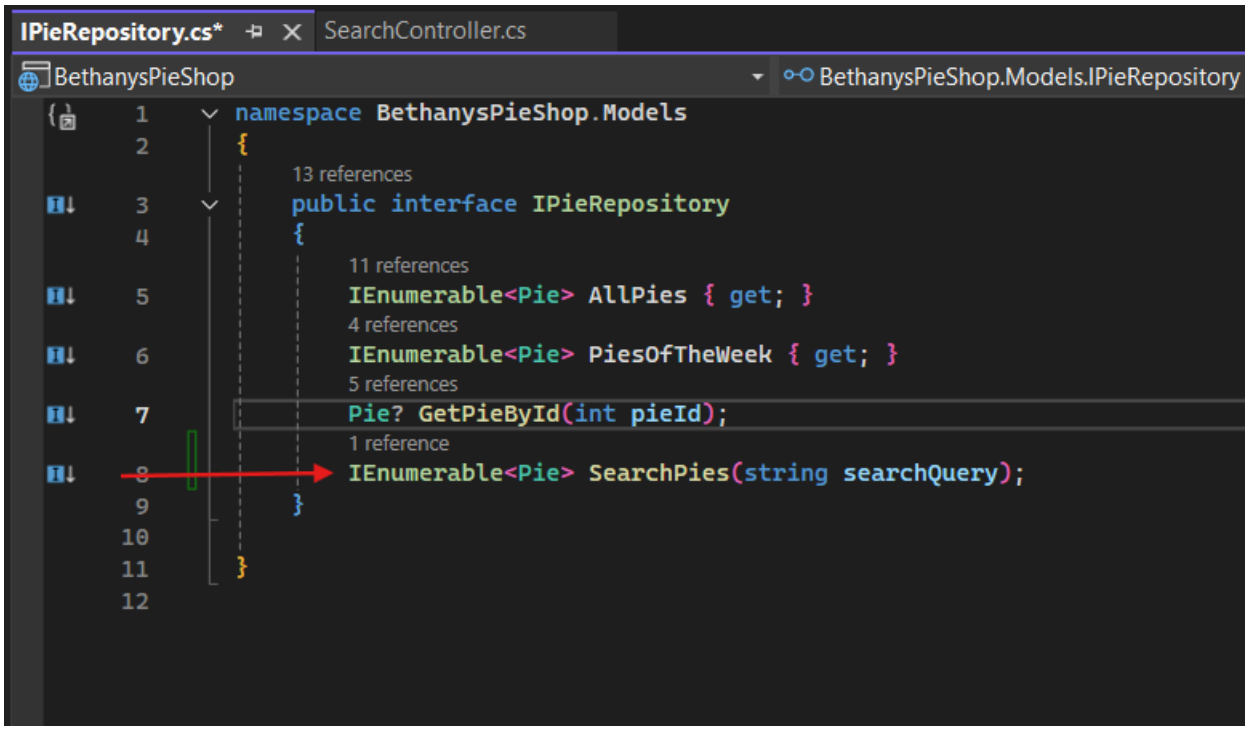
Quindi ritorniamo su SearchController.cs:



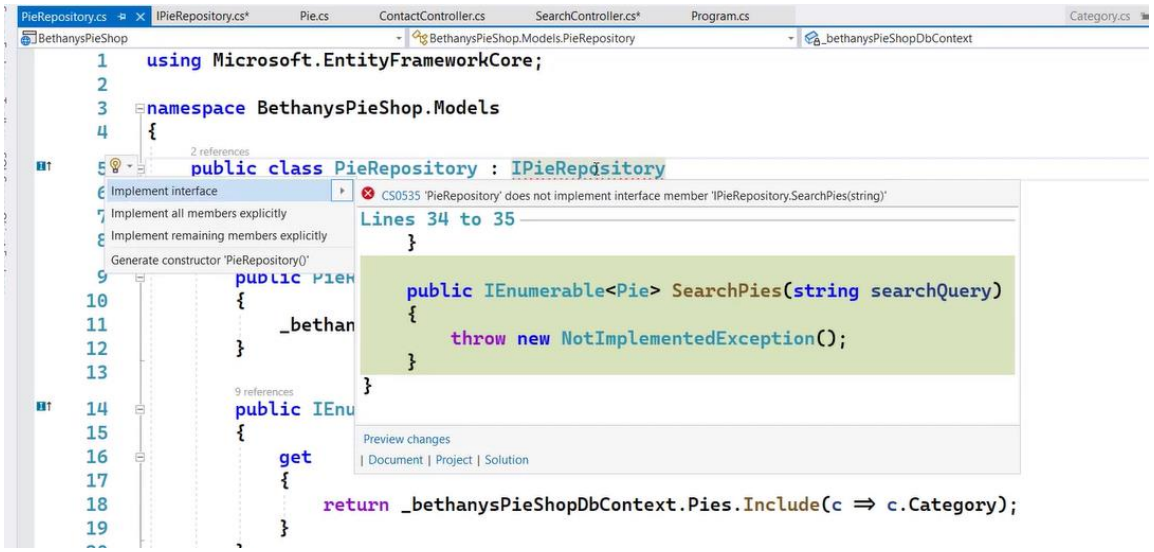

Ora per poter raggiungere questo metodo devo anche assegnarli un attributo, ciò che accadtrà è che il client, o l'applicazione web, invierà quella searchQuery alla mia API.
L'invio dei dati a un browser avverrà con una richiesta POST, proprio come abbiamo fatto con il modulo creato in un modulo precedente.



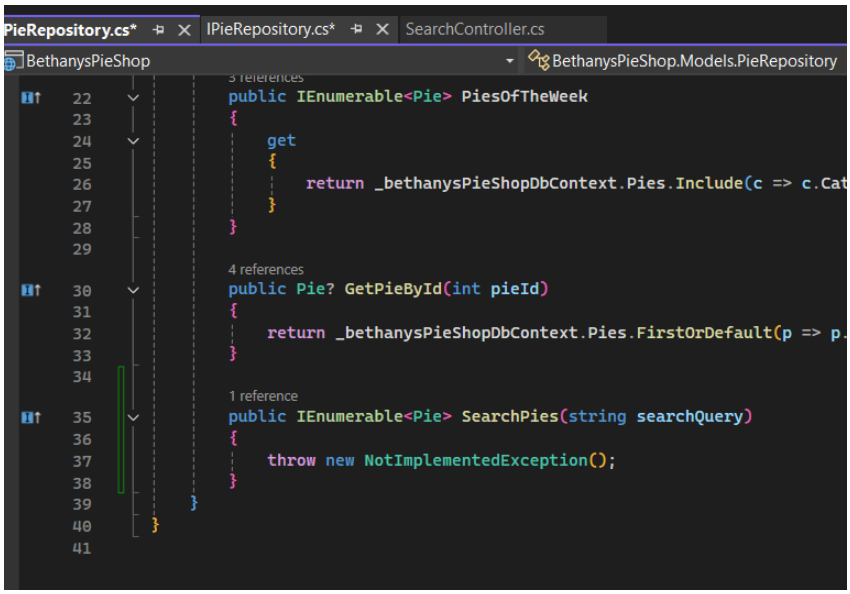

Quindi dobbiamo aggiungere in IPieRepository.cs una nuova funzionalità:



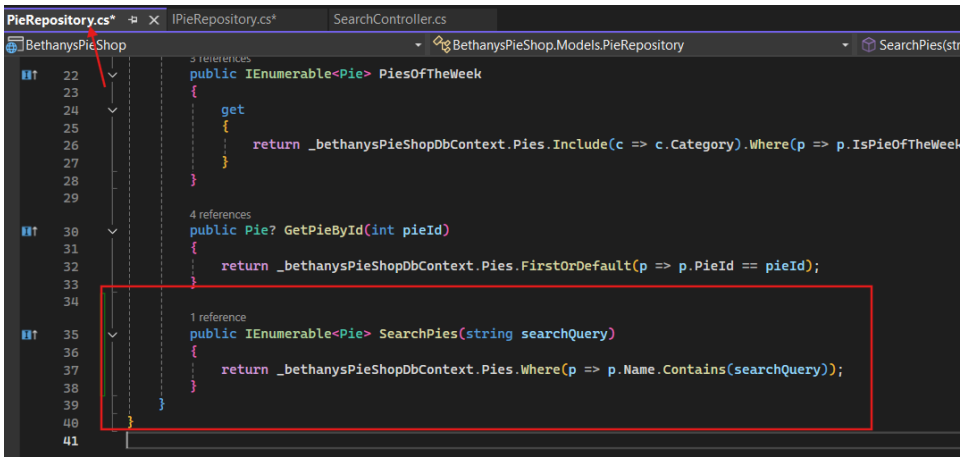

Va quindi effettivamente implementato in PieRepository.cs:



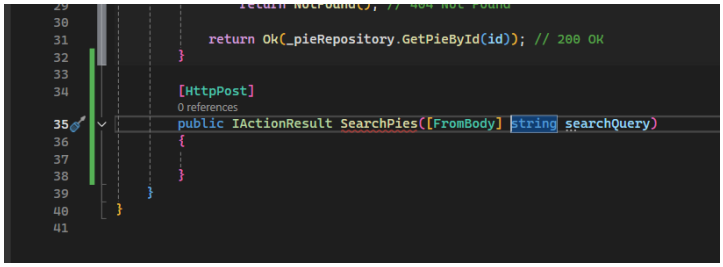

Dopo aver cliccato "Implement interface" lo vedremo aggiunto alla fine del codice:




Però dobbiamo modificare riga 37 di PieRepository.cs:

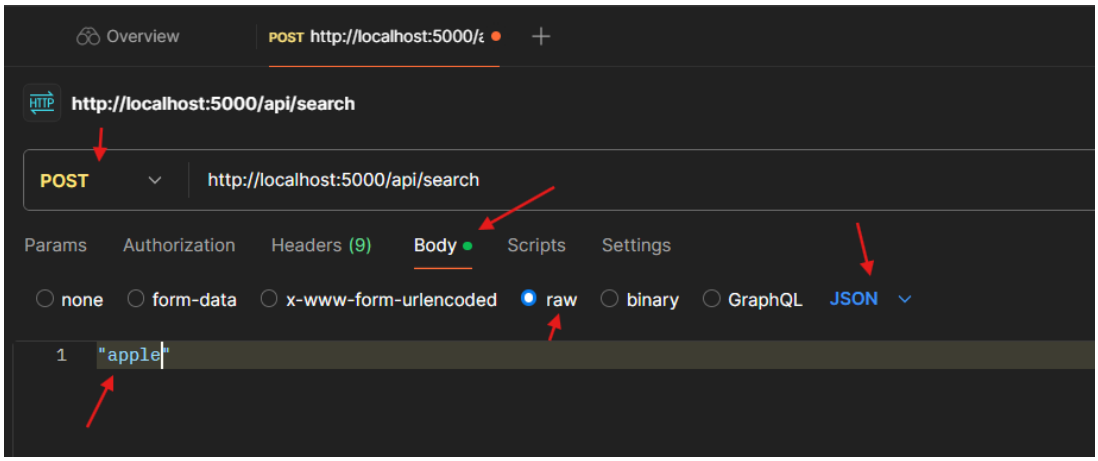


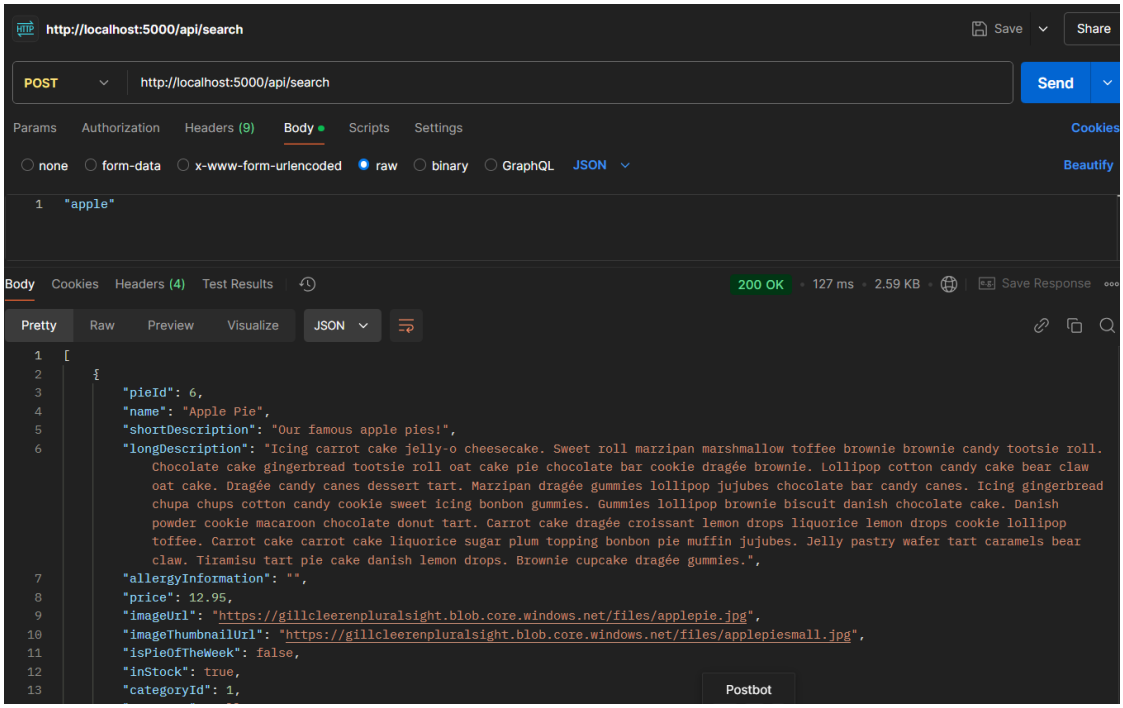

Okay, ora possiamo tornare nel nostro SearchController.cs è aggiungere l'implementazione al metodo:



```
33
34          [HttpPost]
            0 references
35          public IActionResult SearchPies([FromBody] string searchQuery)
36          {
37              IEnumerable<Pie> pies = new List<Pie>();
38              if (!string.IsNullOrEmpty(searchQuery))
39              {
40                  pies = _pieRepository.SearchPies(searchQuery);
41              }
42              return new JsonResult(pies);
43          }
44      }
45  }
46
```

Testiamo su PostMan e cambiamo il verbo in POST, impostiamo Body con raw e JSON, all'interno della prima riga scriviamo la stringa "apple"



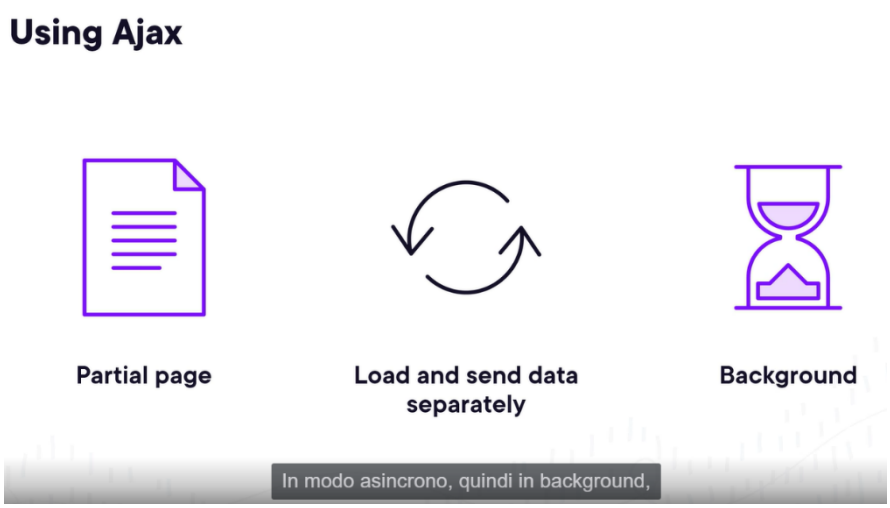



Verranno quindi mostrate le Torte che contengono "apple", difatti abbiamo Apple Pie e Christmas Apple Pie, restituiti come json.

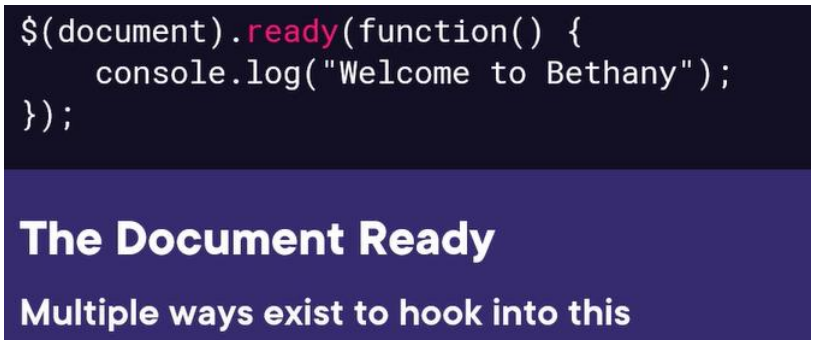
Considerazioni: Postman è stato molto utile per testare il POST, inoltre possiamo notare lo Status: 200 Ok

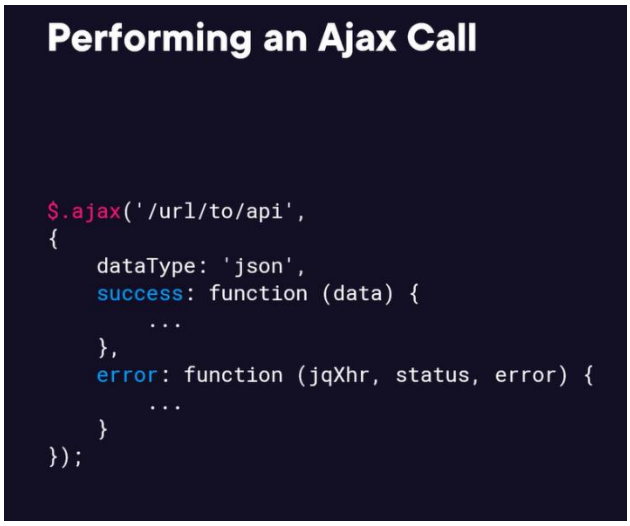
Possiamo dire che l'API è pronta.

**Adding jQuery adn Ajax**







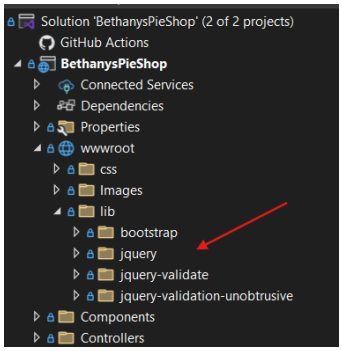



**Demo: Creating the Search Page with Ajax and the API**

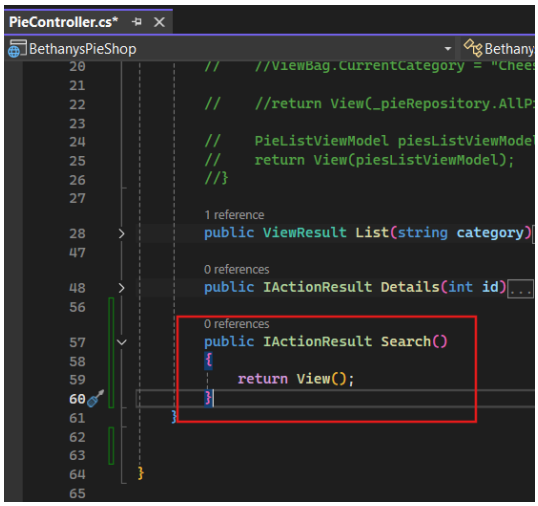
Utilizzeremo jQuery per effettuare una chiamata asincrona alla nostra API, una chiamata ajax e invierà il valore di ricerca immesso dall'utente.

Si spera che l'api risponderà con una risposta JSOM 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 tati JSON, verrà acquisito da jQuery e utilizzato per aggiornare la visualizzazione.

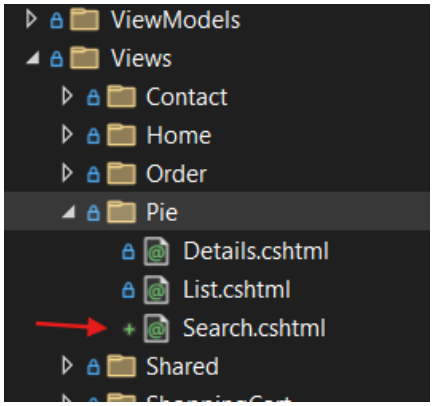
Abbiamo già jQuery:

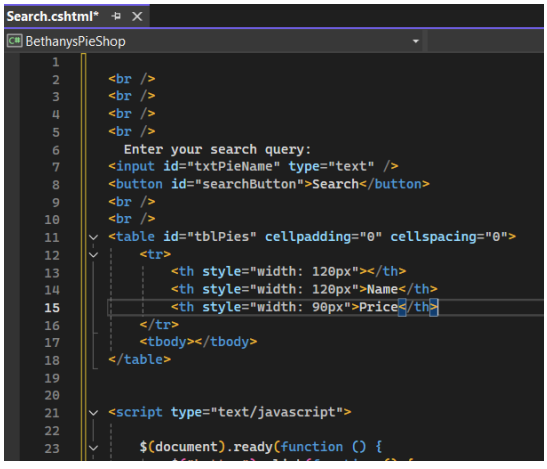
Prima di procedere aggiungiamo in PieController.cs l'action Result per il Search():




Creiamo una vista per Search.cshtml:




All'interno di Search.cshtml aggiungeremo il codice per creare la chiamata asincrona:



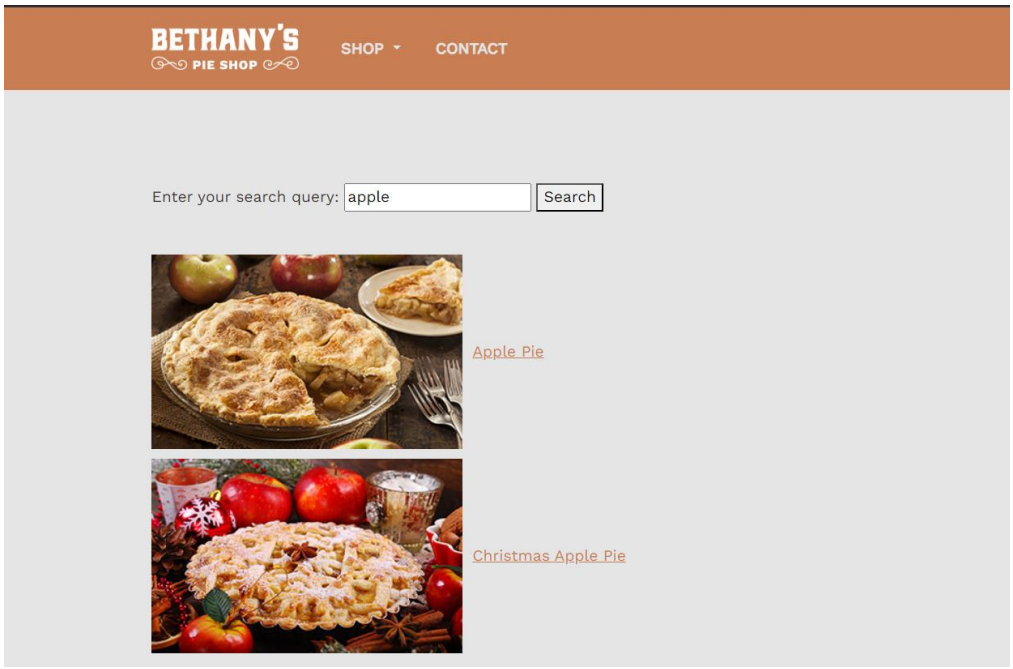
```
<script type="text/javascript">

    $(document).ready(function () {
        $("button").click(function () {
            var searchQuery = $.trim($("#txtPieName").val());
            $("table tbody").html("");
            $.ajax({
                type: "POST",
                url: "/api/Search",
                data: "\"" + searchQuery + "\"",
                contentType: "application/json; charset=utf-8",
                dataType: "json",
                success: function (pies) {
                    var table = $("#tblPies");
                    table.find("tr:not(:first)").remove();
                    $.each(pies, function (i, pie) {
                        $("tbody").append($("<tr>"));
                        appendElement = $("tbody tr").last();
                        appendElement.append($("<td>").html('<img src=' + pie.imageThumbnailUrl + '>'));
                        appendElement.append($("<td>").html('<a href="/pie/details/' + pie.pieId + '">' + pie.name + '</a>'));
                        appendElement.append($("<td>").html(pie.price));
                    });
                },
                error: function (xhr, status, error) {
                    console.log(xhr)
                }
            });
        });
    });

</script>
```

Fatto cil ci rechiamo in Shared > _Layout.cshtml e aggiungiamo:

```
_Layout.cshtml*   Search.cshtml   Search.cshtml
BethanysPieShop
                    </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">
                            <vc:category-menu></vc:category-menu>
                            <li class="nav-item">
                                <a asp-controller="Contact" asp-action="Index" class="nav-link">Contact</a>
                            </li>
                        </ul>
                        <ul class="navbar-nav mb-2 mb-lg-0">
                            <li class="nav-item">
                                <a class="nav-link" asp-controller="Pie" asp-action="Search">
                                    <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" fill="currentColor" class="bi bi-search"
                                        <path d="M11.742 10.344a6.5 6.5 0 1 0-1.397 1.398h-.001c.03.04.062.078.098.115l3.85 3.85a1 1 0 0 0
                                    </svg>
                                </a>
                            </li>
                        </ul>
                        <vc:shopping-cart-summary></vc:shopping-cart-summary>
                    </div>
                </div>
            </nav>
        </header>

        @RenderBody()
    </div>
```

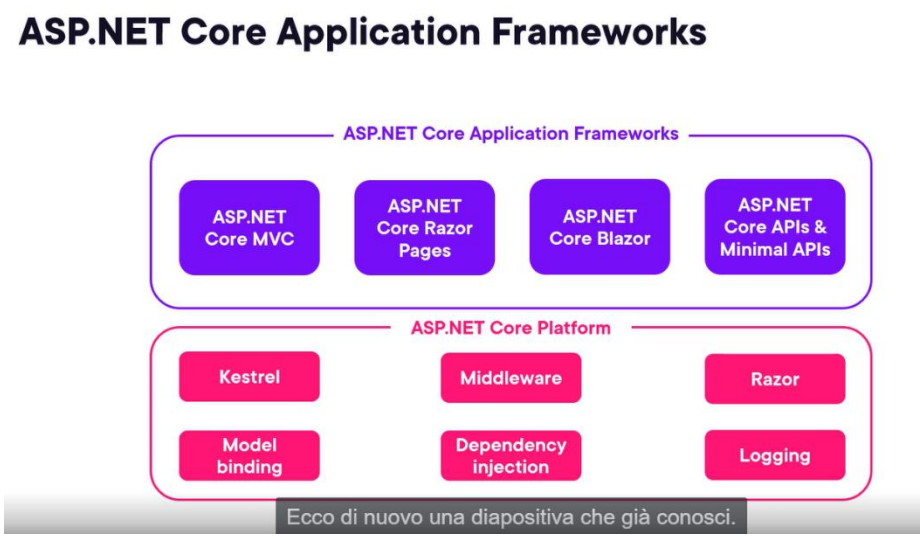
Osservazione:
Un'icona di ricerca nell'intestazione, un collegamento che porterà all'azione di ricerca di pieController, un svg che rappresenta la lente di ingrandimento.

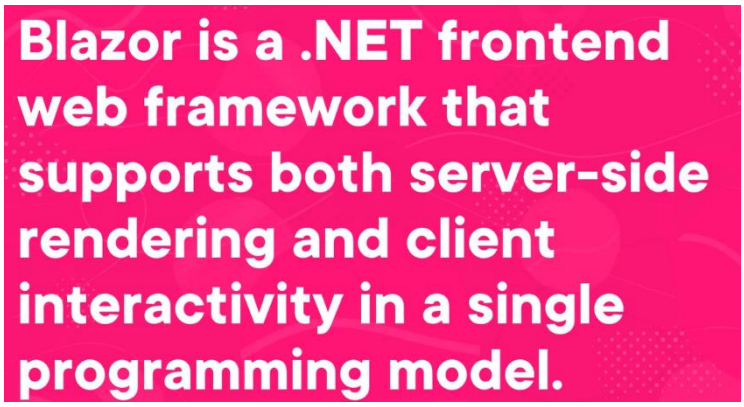
Possiamo ora avviare l'applicazione, cliccare la l'icona della lente di ingrandimento e digitare "apple"

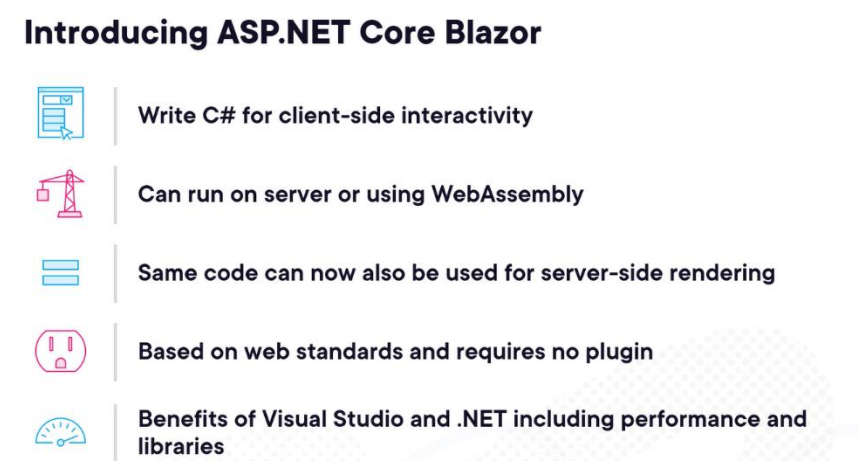


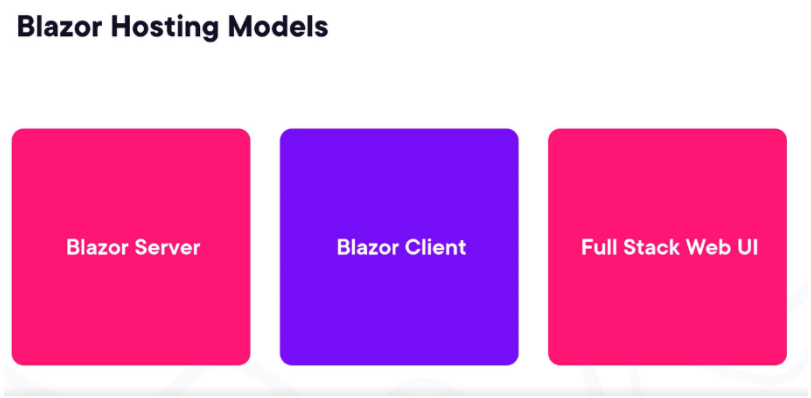

**Introducing ASP.NET Core Blazor**

**Stessa cosa che abbiamo fatto precedentemente con l'ausilio di jQuery e Ajax ma con Blazor, quindi solo C#,**



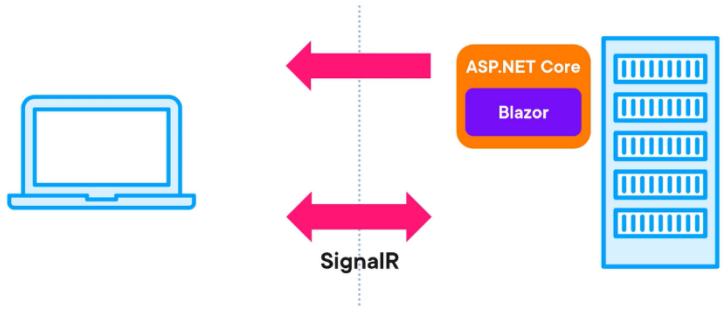



Inizieremo con **Blazor Server**:

**Blazor Server**




Nel modello di hosting lato serve il codice Blazor eseuirà 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.

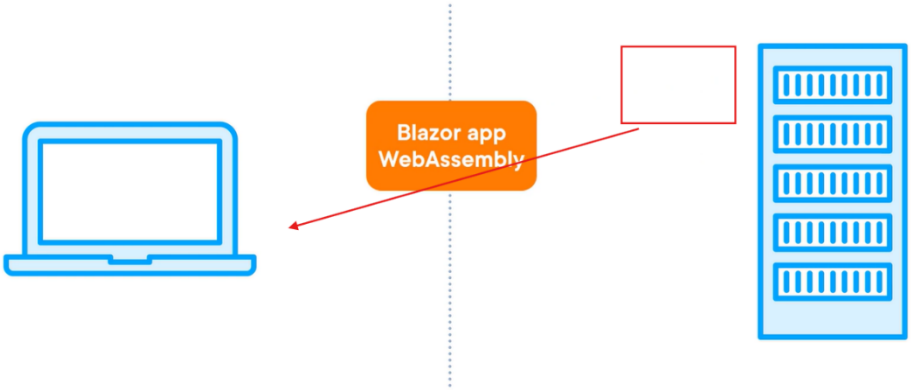
Per tutto ciò viene creata una istanza per salvare il tutto in memoria.

**Blazor Client (WASM)**:

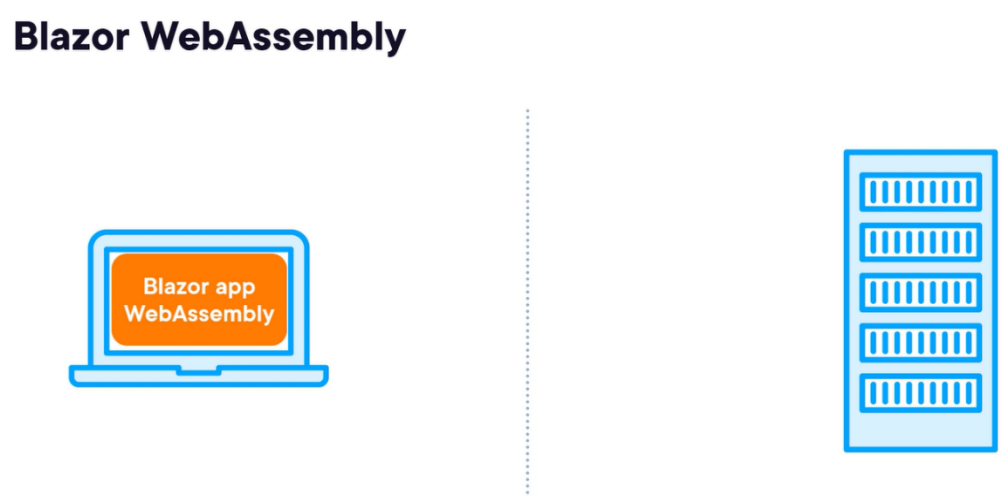
Permette di aggiungere interattività lato client.

Questo modelli si basa sil conetto che il nostro codice confezionato, verrà scaricato sul computer client e verrà eseguito

**Blazor WebAssembly**




Ed eseguito da li:




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.

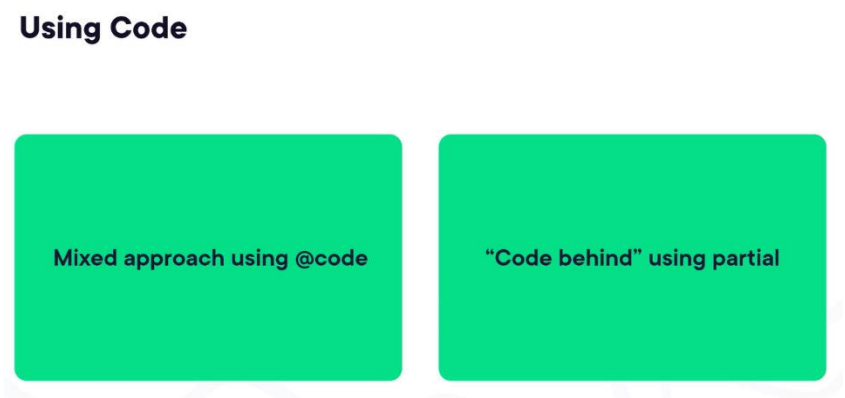
**Full Stack Web UI (Server-side rendering Interactivity can be added):**

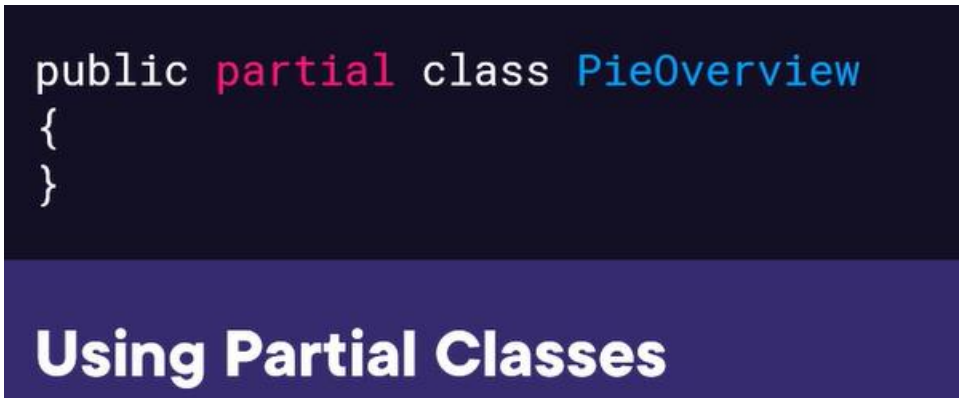

Osservazione: Possiamo notare che anziché codice JavaScript è presente codice C#

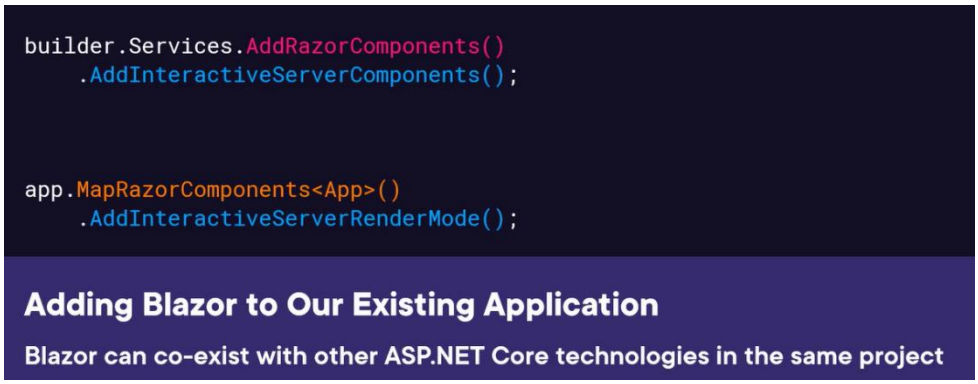
Avremo un componente principale con diversi livelli di nidificazione.

Un component può essere anche una pagina o un pulsante.

In genere conterrà codice markup, codice Razor e C#.



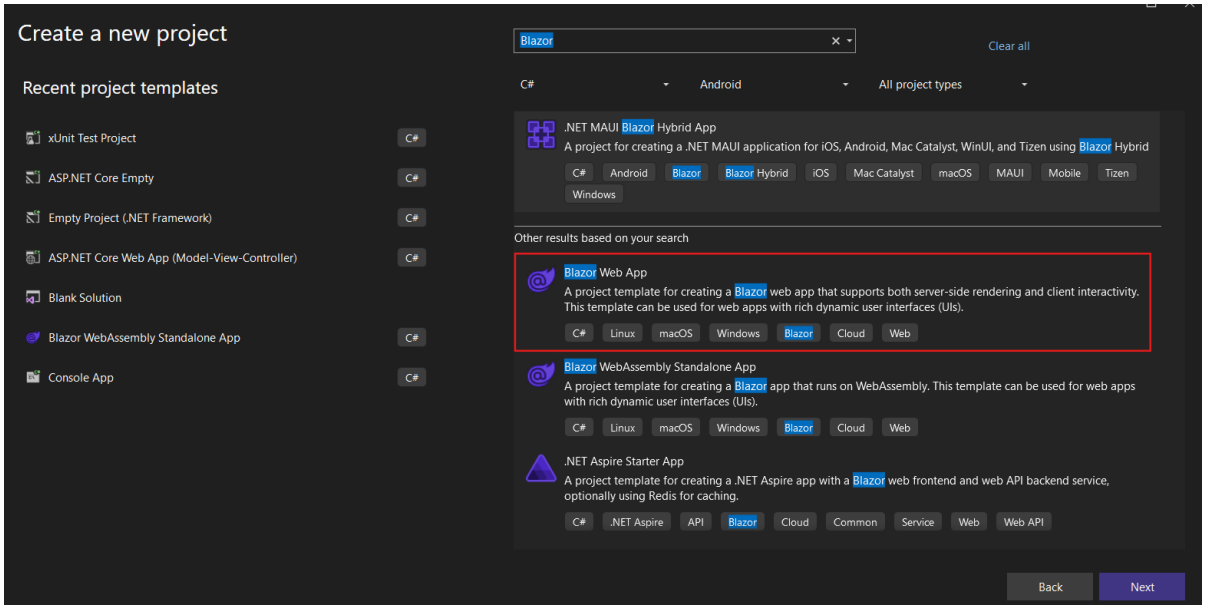



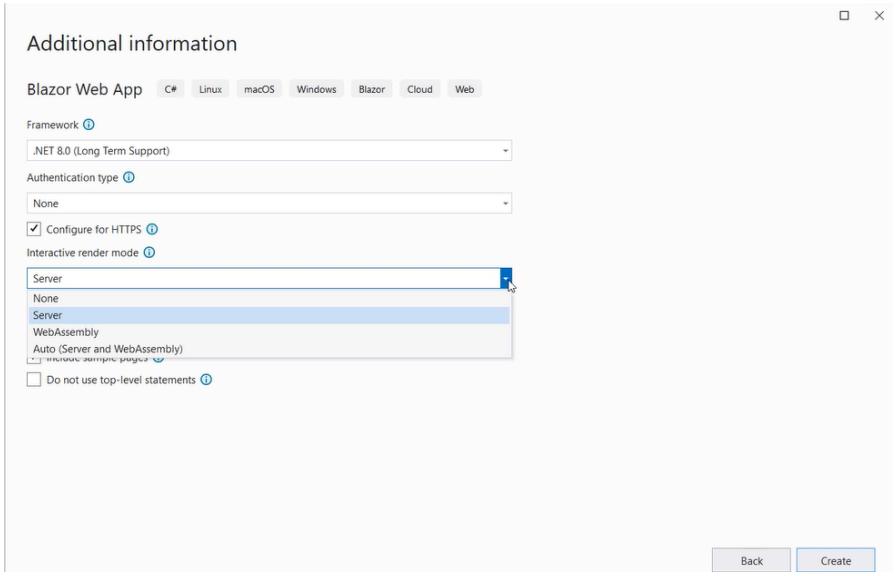



Tecnicamente, questo introduce il supporto per il rendering lato server e offriamo i supporto per l'interattività lato client con Blazor.

**Demo: Exploring a New Blazor Project**

Creiamo quindi un nuovo progetto Blazor.

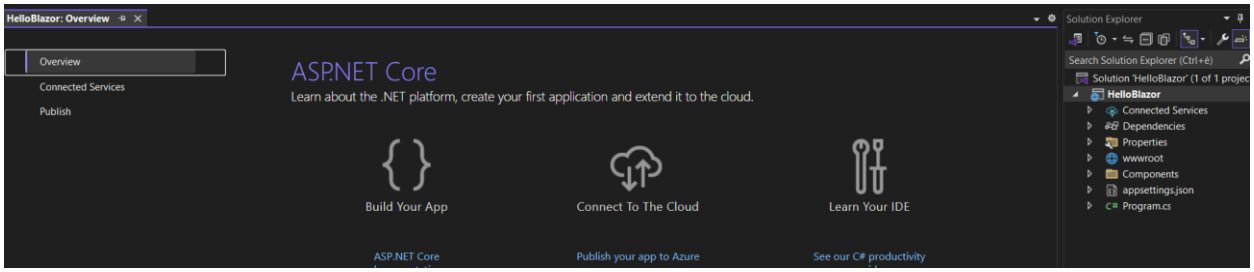




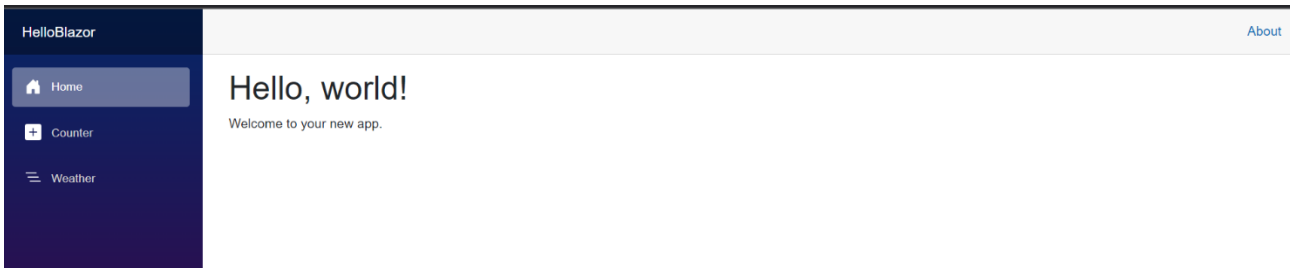

Osservazione: questa è la parte più importante, questo modello sarà Server, il che significa che il codice Blazor verrà eseguito sul lato server e i componenti restituiranno codice HTML statico inviato al client.

Quini un modello simile a quello MVC e Razor Pages.

Scegliamo quindi Server e procediamo alla creazione:

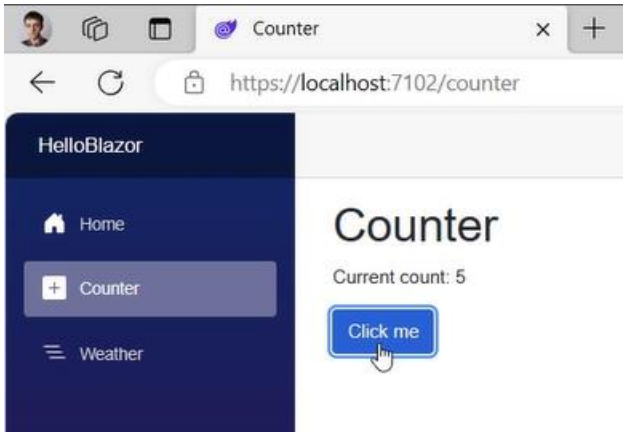



Prima di guarda il codice eseguiamo la nostra applicazione



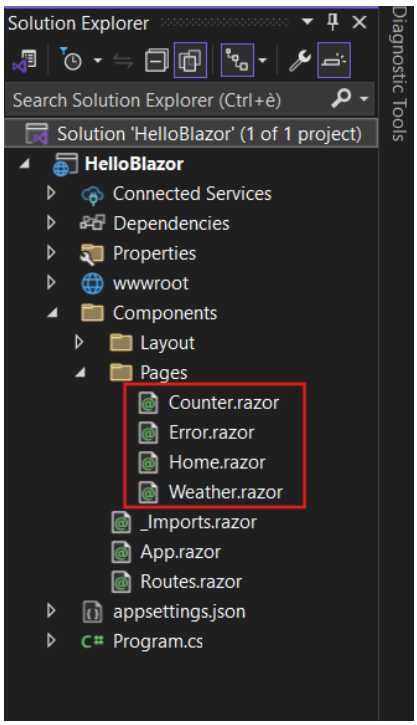

Osservazione: La home page è molto semplice, mostra solo i risultati statici.

La pagina Meteo mostra alcuni dati, ma mostra prima il caricamento mentre i dati arrivano ,




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:



```razor
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Osservazioni:

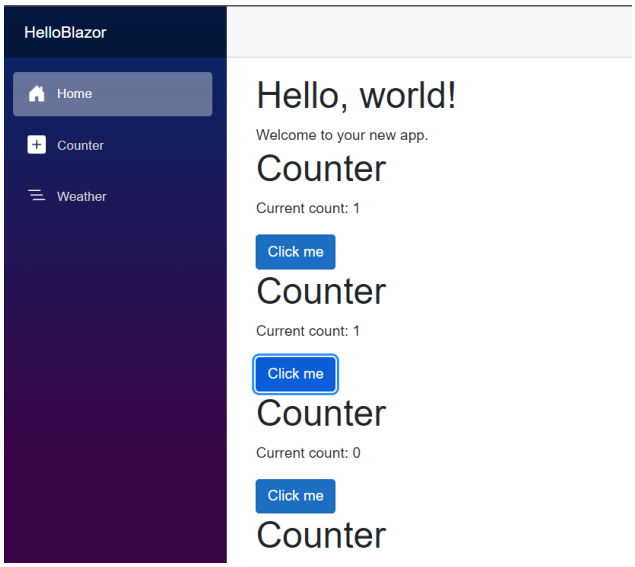
1: Indirizzo, quindi /counter è URL per accedere a questo componente.

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:

```razor
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<Counter></Counter>
<Counter></Counter>
<Counter></Counter>
<Counter></Counter>
<Counter></Counter>
```




Questo è un grande vantaggio, in quanto TUTTO in Blazor è un componente, facendo cosi ho creato 4 istanze del contatore nella mia HomePage e sono tutte singole isole con funzionalità che vengono caricate quando navighiamo verso il componete Home

Essendo una Blazor application è molto simile a quella che abbiamo creato precedentemente, difatti, anche qui abbiamo un Program.cs:

Home.razor

HelloBlazor

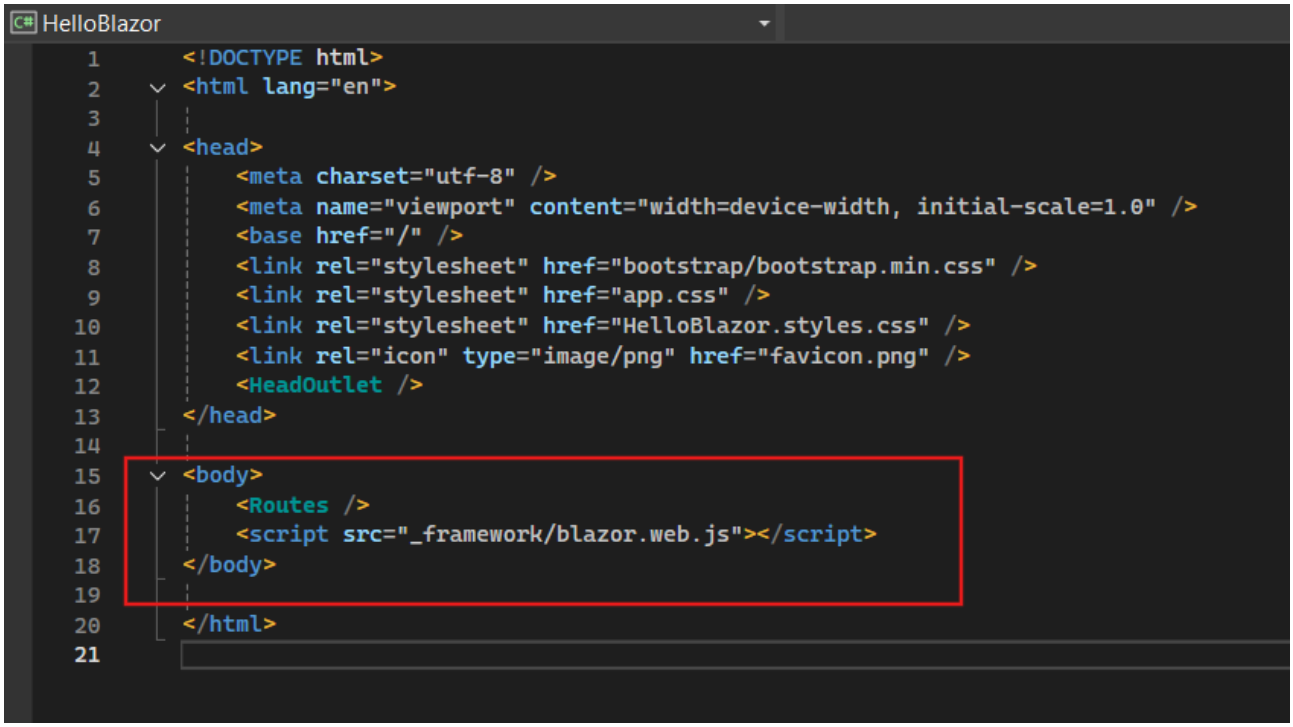
```
1   using HelloBlazor.Components;
2
3   var builder = WebApplication.CreateBuilder(args);
4
5   // Add services to the container.
6   builder.Services.AddRazorComponents()
7       .AddInteractiveServerComponents();
8
9   var app = builder.Build();
10
11  // Configure the HTTP request pipeline.
12  if (!app.Environment.IsDevelopment())
13  {
14      app.UseExceptionHandler("/Error", createScopeForErrors: true);
15      // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
16      app.UseHsts();
17  }
18
19  app.UseHttpsRedirection();
20
21  app.UseStaticFiles();
22  app.UseAntiforgery();
23
24  app.MapRazorComponents<App>()
25      .AddInteractiveServerRenderMode();
26
27  app.Run();
```

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.

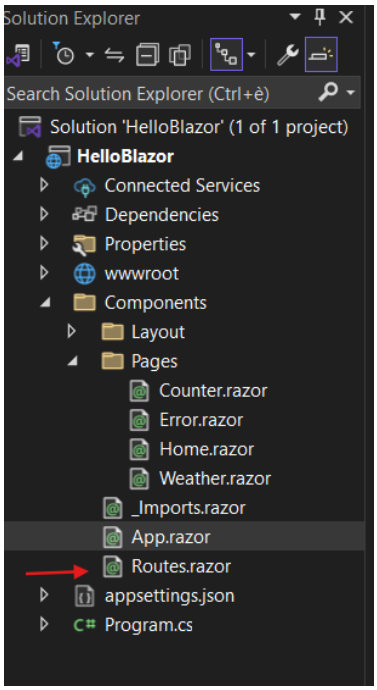
Successivamente abbiamo il middleware a riga 24.

App.razor è il root dell'applicazione, osserviamolo:

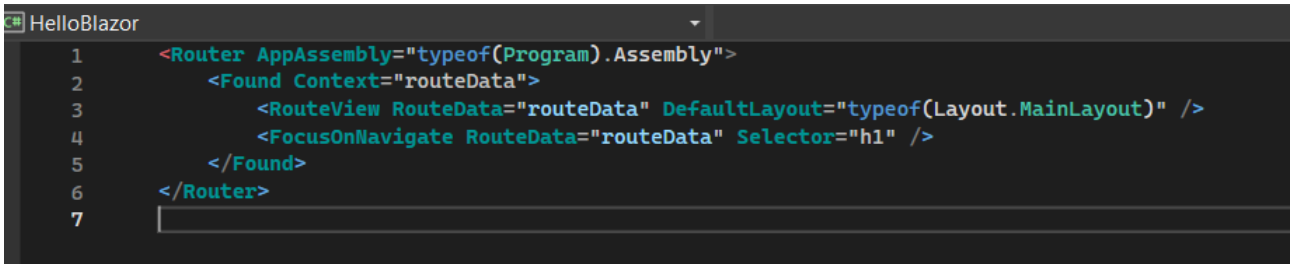



Osservazione: Assomiglia a un normale HTML, ma abbiamo nel body il componente Routes, che fa si che i componenti siano renderizzati e l'uso dello script blazor.web.js.

Il riferimento a Routes.razor:




Esaminiamo Routes.razor:

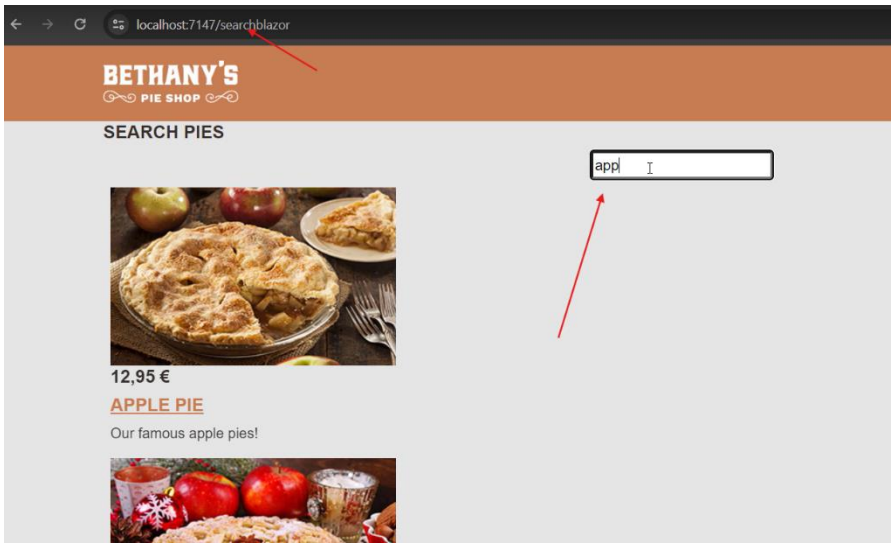



Osservazione: E' il componente responsabile della mappatura effettiva di una richiesta in entrata su un componente da mostrare, specifica inoltre un layout da utilizzare.

Senza analizzare anche il MainLayour.razor, procediamo direttamente all'aggiunta di componenti Blazor nel nostro progetto

**Demo: Creating the Search Page Using Blazor**

Risultato che vogliamo ottenere:




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:

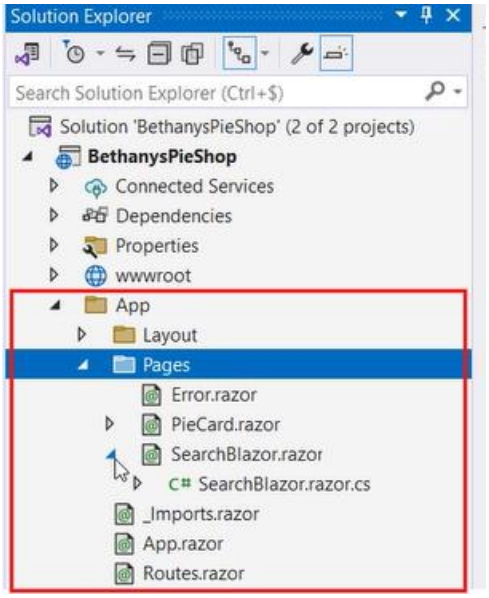
```
Program.cs*
BethanysPieShop
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<IPieRepository, PieRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();

builder.Services.AddScoped<IShoppingCart, ShoppingCart>(sp => ShoppingCart.GetCart(sp)); // Add shopping cart
builder.Services.AddSession(); // Add session services
builder.Services.AddHttpContextAccessor(); // Add HTTP context accessor services

builder.Services.AddControllersWithViews() // Add MVC services
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
    });


builder.Services.AddRazorPages(); // Add Razor pages services
builder.Services.AddRazorComponents().AddInteractiveServerComponents(); // Add Razor components services
builder.Services.AddDbContext<BethanysPieShopDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration["ConnectionStrings:BethanysPieShopDbContextConnection"]);
});

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

In questo caso aggiungiamo interattività utilizzando BlazorServer.

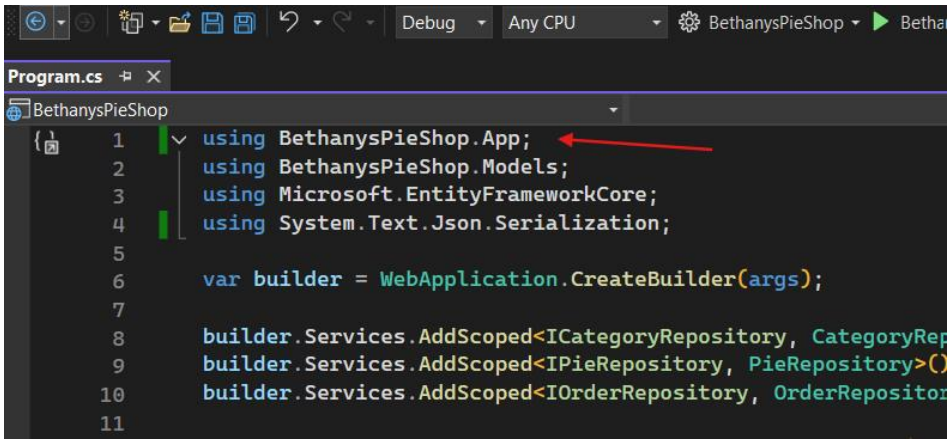
Ora giungiamo verso la fine del Program.cs e mappiamo Razor Components e il ServerRenderMode:

```
Program.cs
BethanysPieShop
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: "defaulte",
46      //    pattern: "{controller=Home}/{action=Index}/{id?}");
47
48      app.UseAntiforgery(); // Middleware component: Enable antiforgery endopoint 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
```

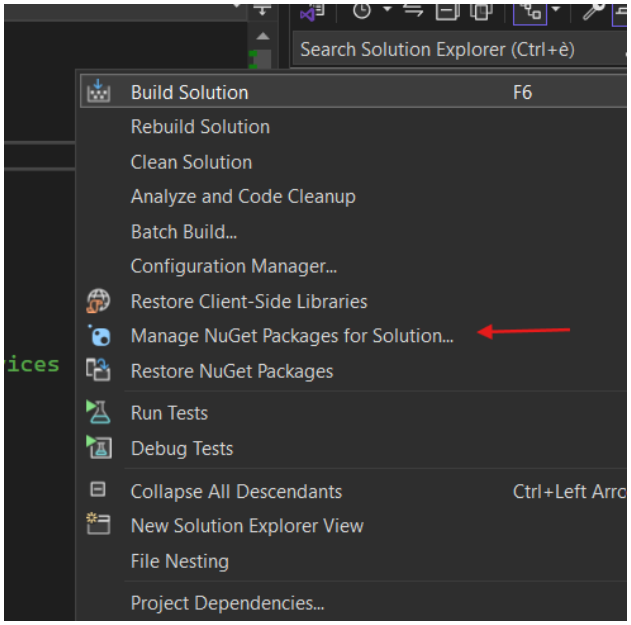
IMPORTANTE:
In questa fase si sono manifestati degli errori, in quanto stiamo importanto Blazor all'interno di un progetto MVC e Razor Pages senza aver creato una soluzione/progetto Blazor, per ovviare tale errore ho dovuto copiare l'intera cartella App:




Ho aggiunto il seguente using in Program.cs:




Ed ho installato i vari package nuGet alla soluzione facendo click destro sulla soluzione:



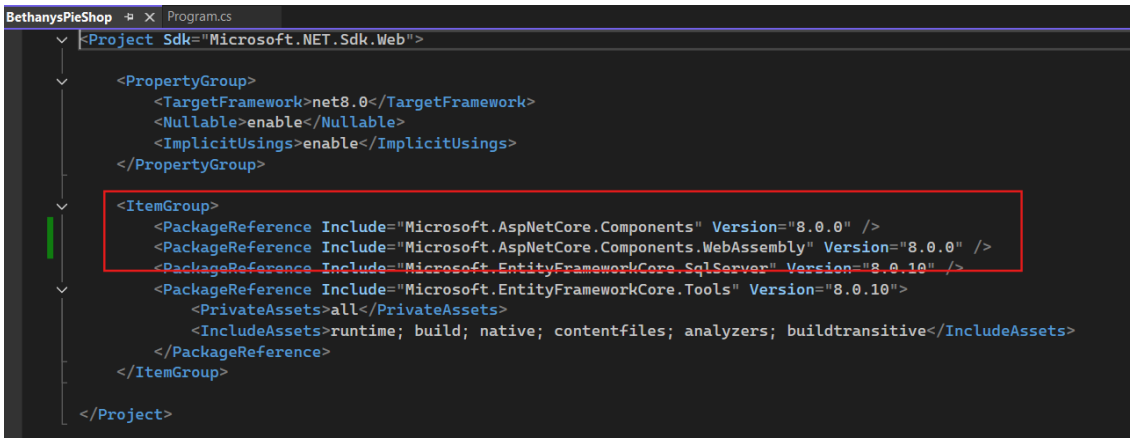

I packages ricercati sono:

- **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.

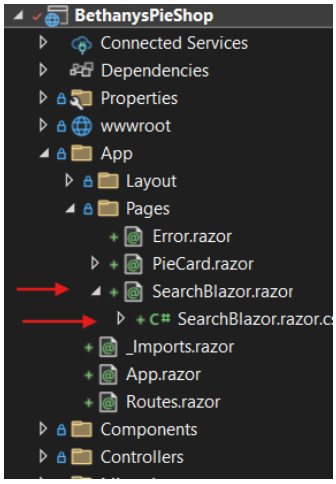
  Dopodiché eseguire "dotnet restore" da terminale.

Per verificare la presenza dei pacchetti installati possiamo esaminare il nostro .csprogj:

- 

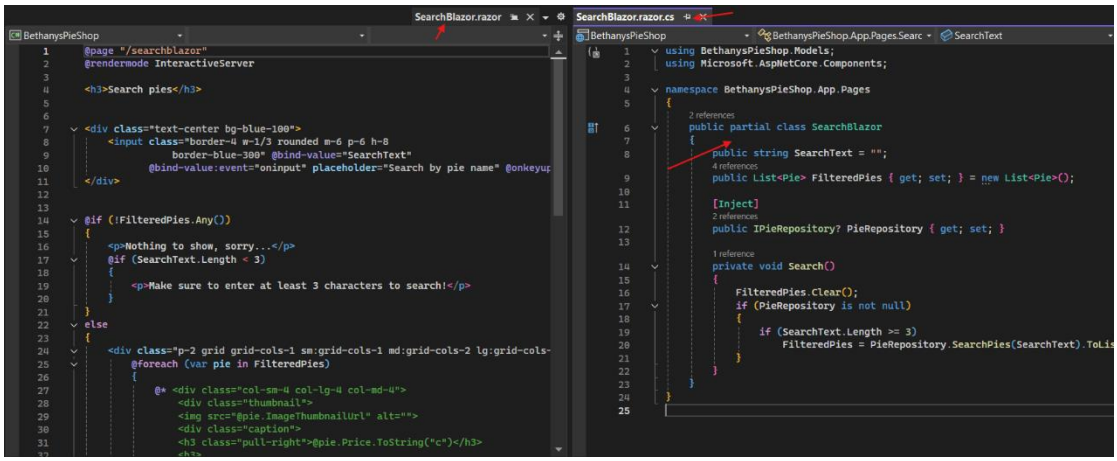

Questa procedura dovrebbe risolvere l'errore di del tipo generico <App>.

Analizzando la cartella SearchBlazor.razor presente in App >Pages:



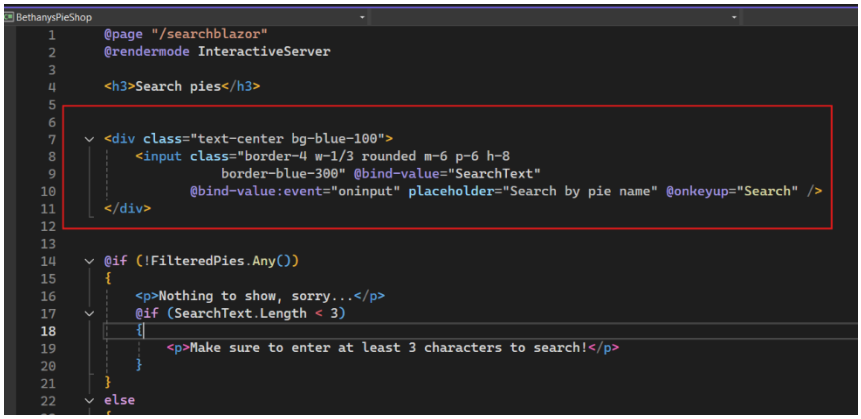

Notiamo una componente Blazor e un code behind.

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




Osservazione: La classe code behind deve essere partial per poter essere associata al componente Blazor stesso.

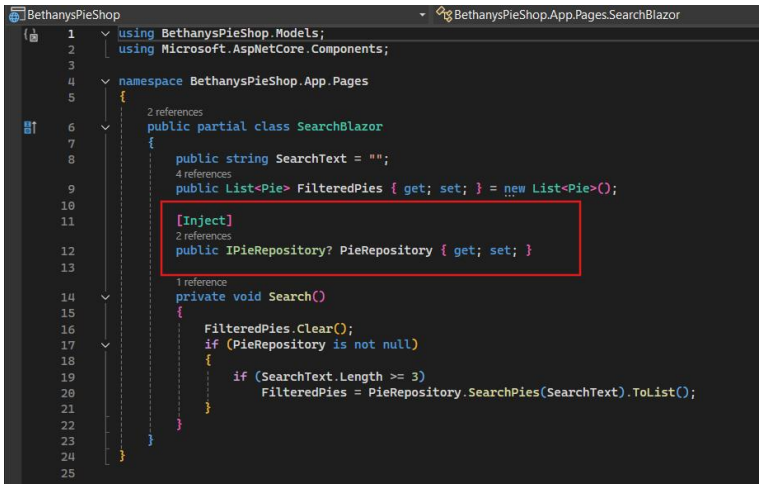
Molto importante è una porzione di codice in SearchBlazor.razor:




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:




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.

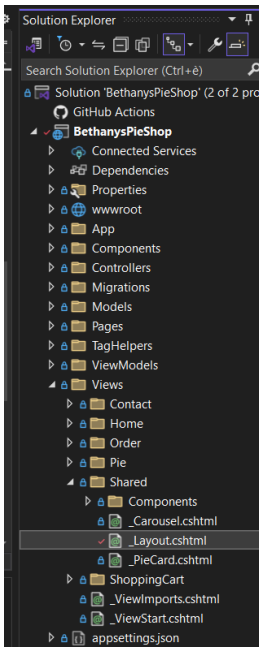
Tornando al metodo Search() il metodo triggerà solo se sono presenti almeno 3 caratteri, in tal caso ritorna una lista di torte filtratrate  consideranzo la stringa ricercata.

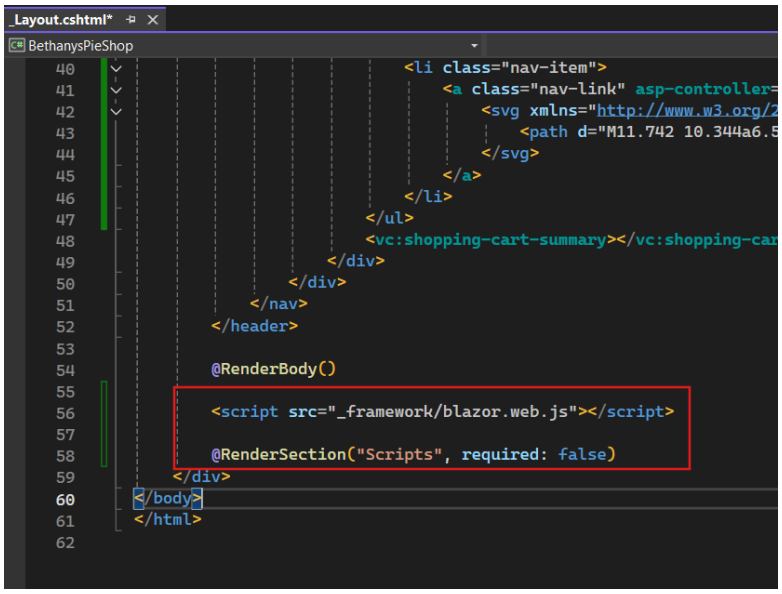
Inoltre mostriamo in SearchBlazor.razor se non c'è niente da mostrare, inoltre comunichiamo all'utente di inserire almeno 3 caratteri:

```
@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">
                <img src="@pie.ImageThumbnailUrl" alt="">
                <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>
                </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].

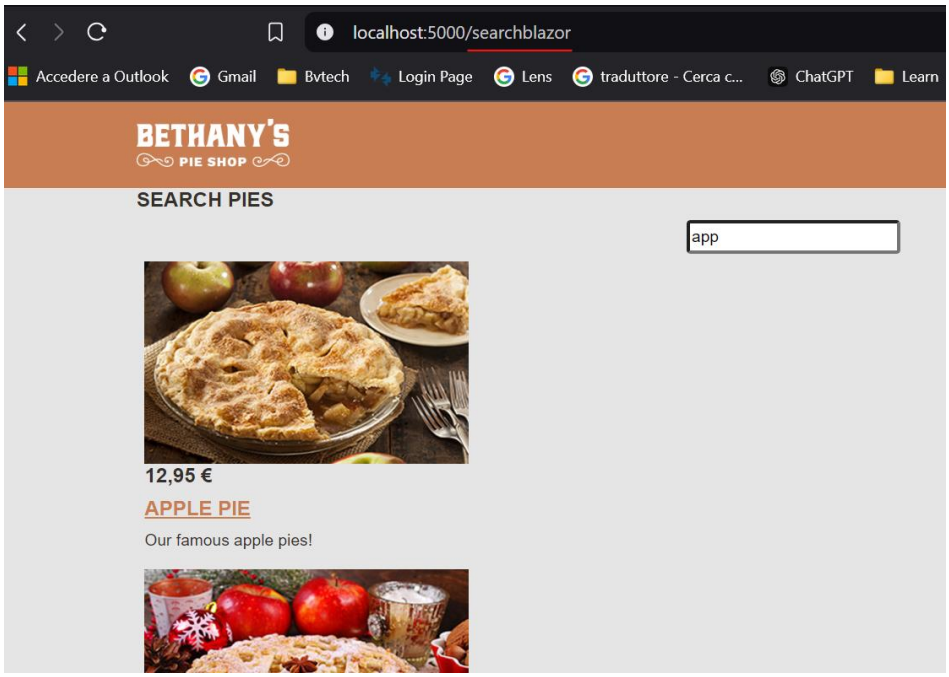
Un'implementazione fondamentale è quella di dover inserire un riferimento a blazor.we. .js in _Views > Shared > _Layout.cs:




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

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

Scaffholding; è possibile generare per noi molto codice dell'interfaccia utente, come schermate di accesso, schermate di password dimenticata e cosi 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.

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>();
```

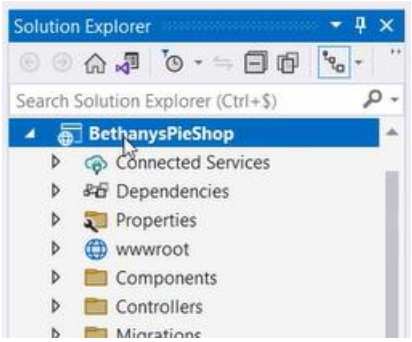
**Configuration Options**

**Password length, user options...**

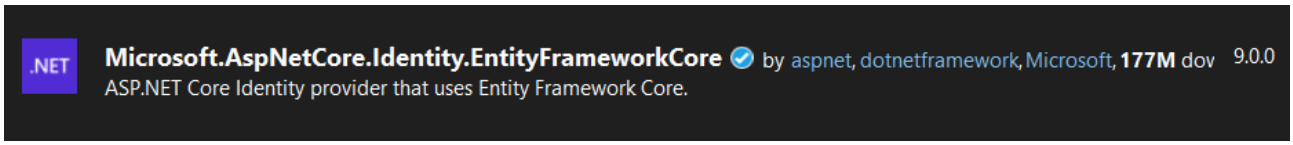
Se ad esempio desideri obbligare gli utenti a utilizzare una password complessa, cosa che dovremmo fare, possiamo configurare Identity in modo che non consenta password semplici.

**Demo: Adding ASP.NET Core Identity to the Application**

Iniziamo aggiungendo i pacchetti corretti nel nostro progetto:

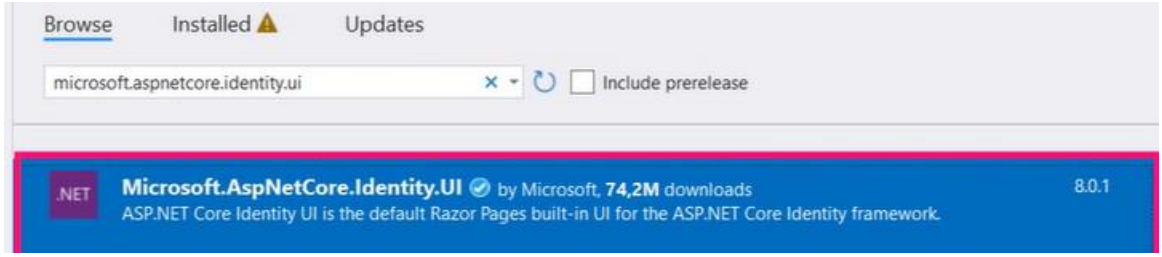



Right click > Manage Nuget Packages..
Ricerchiamo "Microsoft.aspnetcore.identity.EntityFrameworkCore"




Lo installiamo

Dopoché stessa procedura con il seguente package:




Ora dobbiamo aggiornare il nostro DbContext:

Prima:

```
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    19 references
    public class BethanysPieShopDbContext: DbContext
    {
        0 references
        public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
        {
        }

        3 references
        public DbSet<Category> Categories { get; set; }
        5 references
        public DbSet<Pie> Pies { get; set; }
        8 references
        public DbSet<ShoppingCartItem> ShoppingCartItems{ get; set; }
        1 reference
        public DbSet<Order> Orders { get; set; }
        0 references
        public DbSet<OrderDetail> OrderDetails { get; set; }
    }
}
```

Dopo:

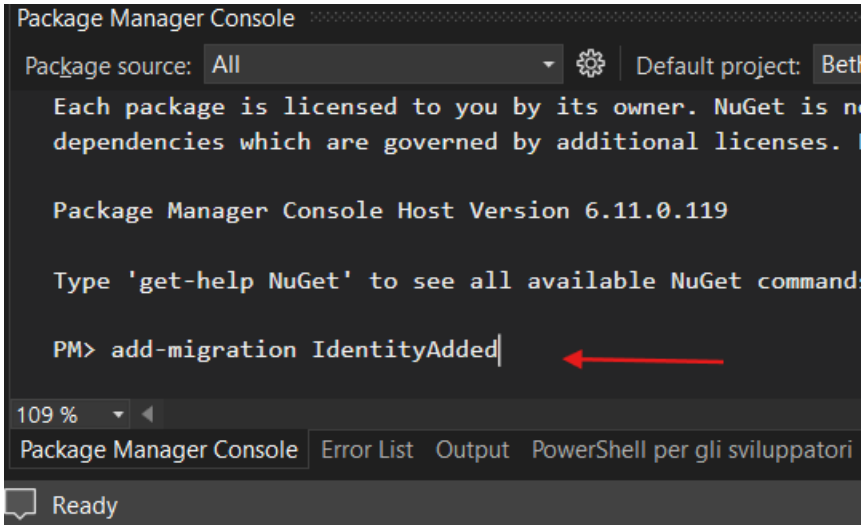
```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace BethanysPieShop.Models
{
    19 references
    public class BethanysPieShopDbContext: IdentityDbContext
    {
        0 references
        public BethanysPieShopDbContext(DbContextOptions<BethanysPieShopDbContext> options) : base(options)
        {
        }

        3 references
        public DbSet<Category> Categories { get; set; }
```

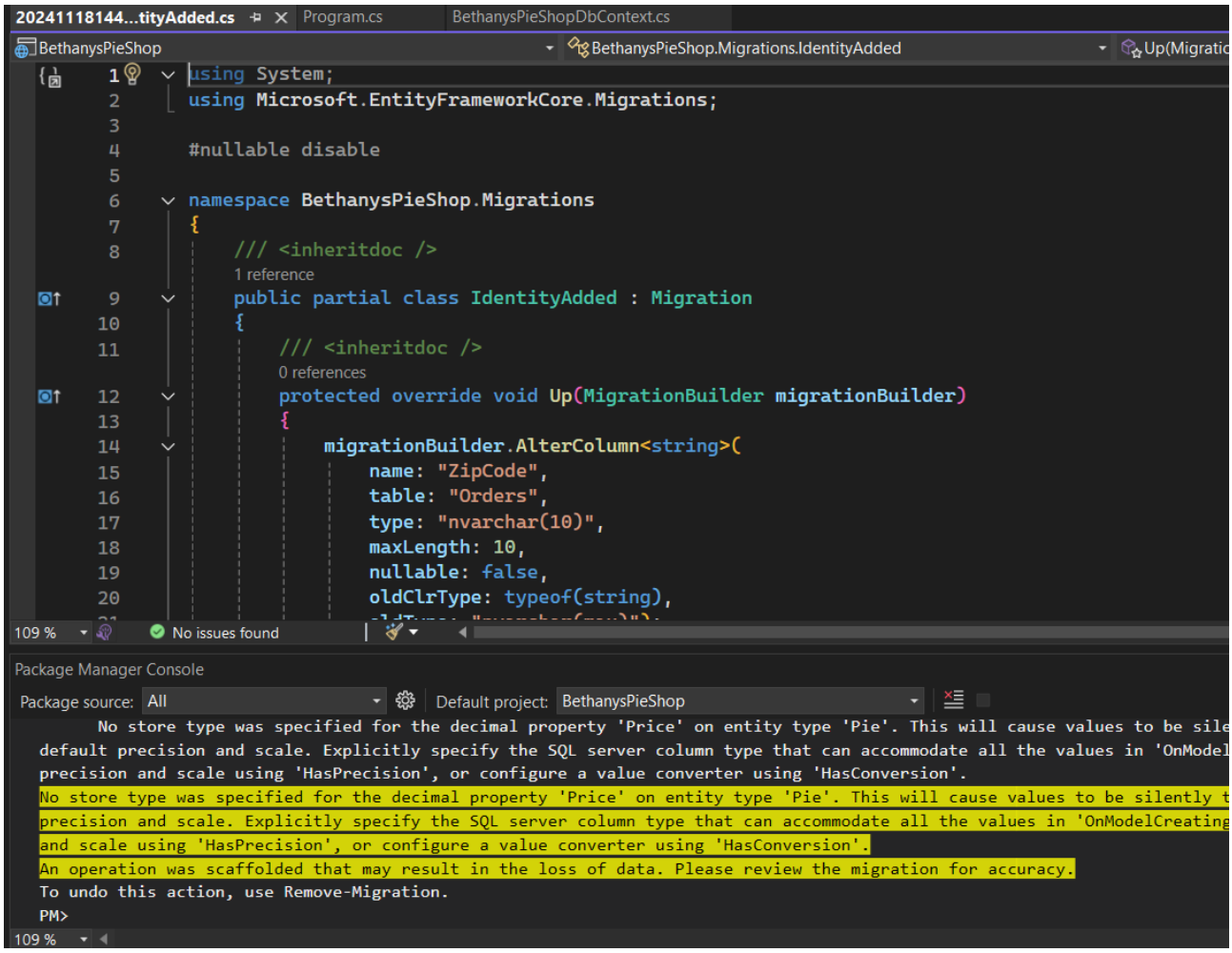
Ora il nostro DbContext è in grado di riconoscere anche le identità e i ruoli.

Ci spostiamo nel Program.cs per aggiungere i middleware necessari:

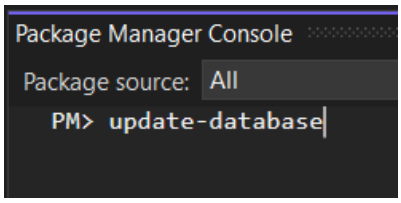

Fatto ciò, salviamo ed effettuiamo una BUILD, in quanto dobbiamo eseguire una migrazione.

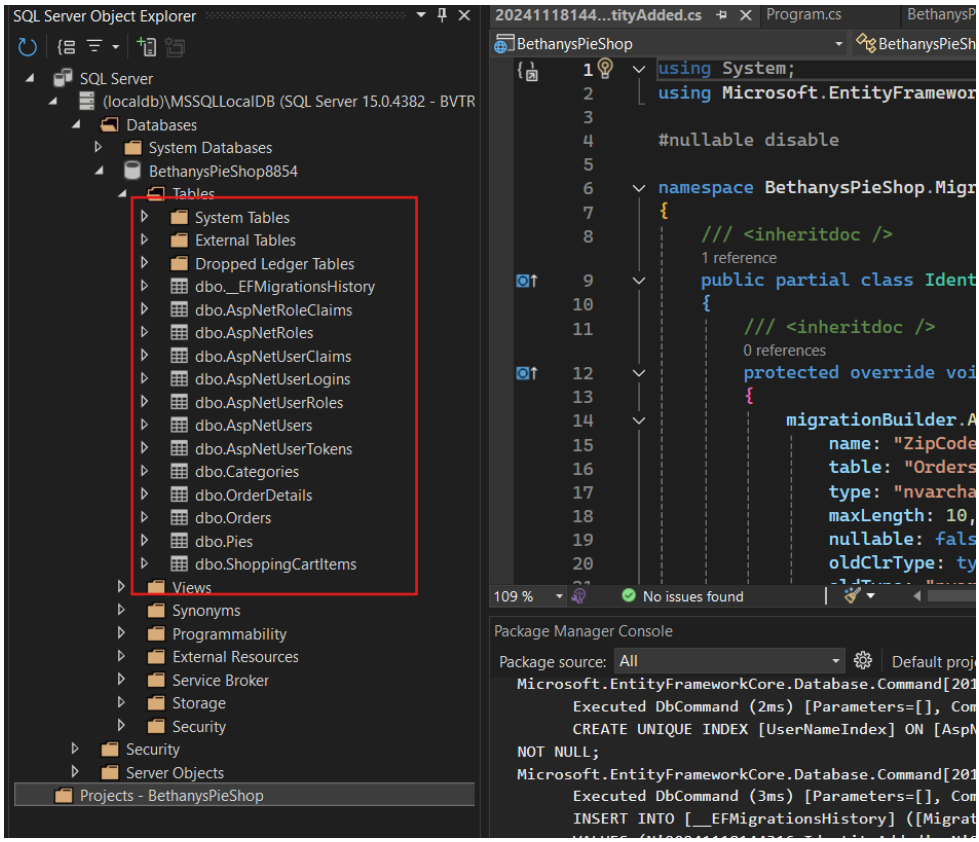
**Dopodiché apriamo il P.M.:**






Ora aggiorniamo il nostro database chiamando

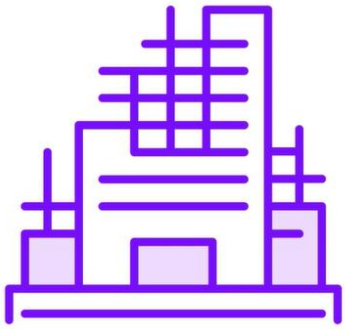





La nostra applicazione è pronta per, adesso dobbiamo aggiungere l'autenticazione al sito.

**Adding Authentication to the Site**


Possiamo utilizzare lo **Scaffholding** per evitare una grande mole di lavoro individuale:



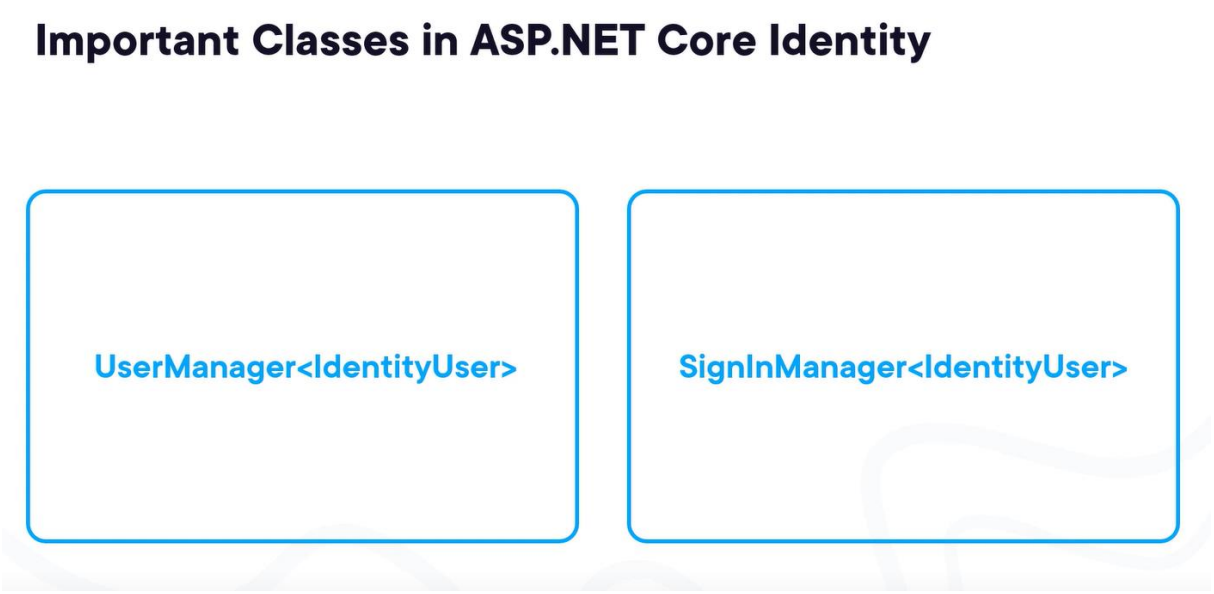
**Scaffolding**

- Visual Studio tooling or through scaffolder 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




**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.

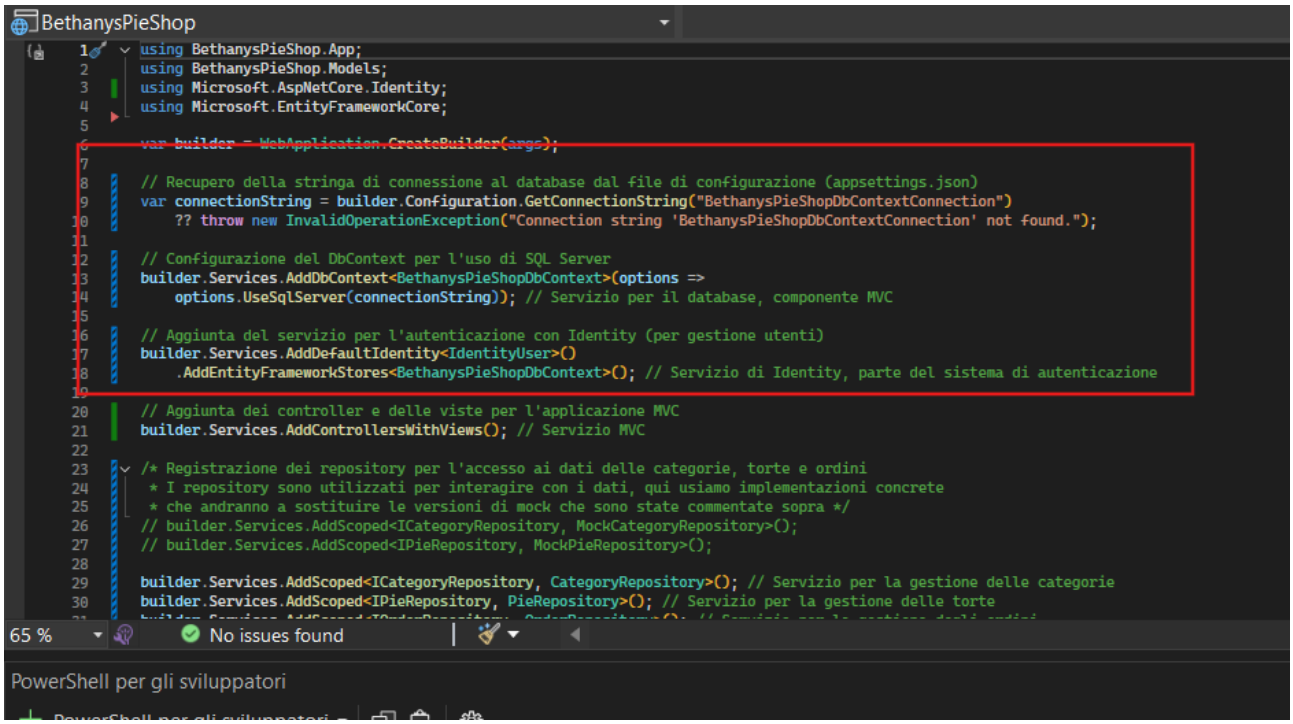
Click destro sul nostro progetto > Add New Scaffoldem Item > Identity > Add

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

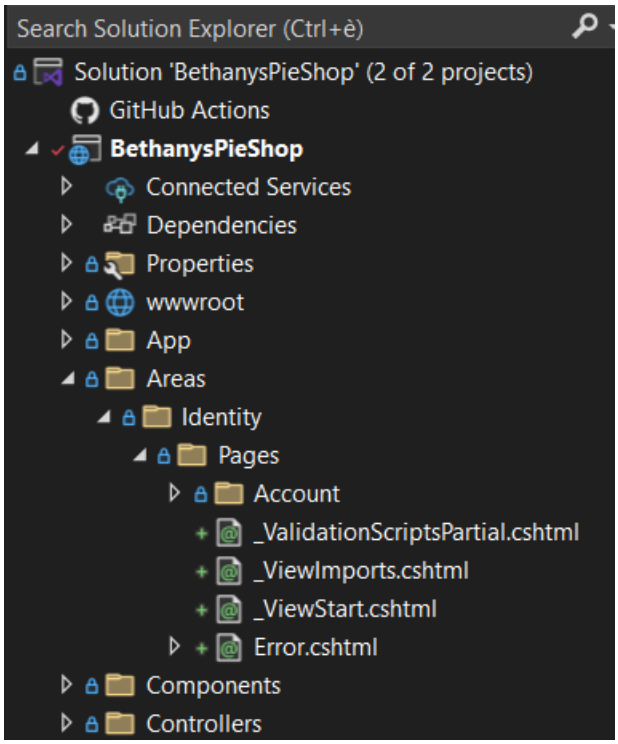
**Noteremo delle modifiche nel program.cs**




Osservazione:
Prima riga 16 era cosi:

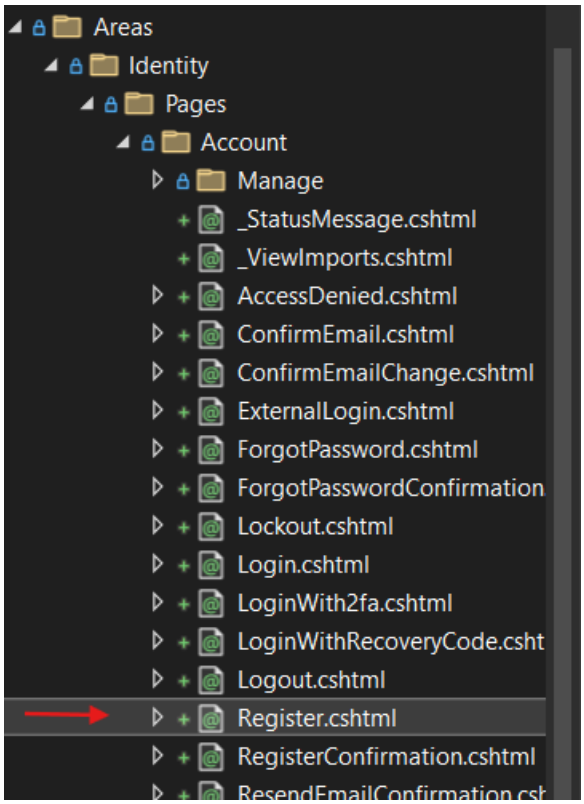

L'ho modificata per rendere i test più semplici.

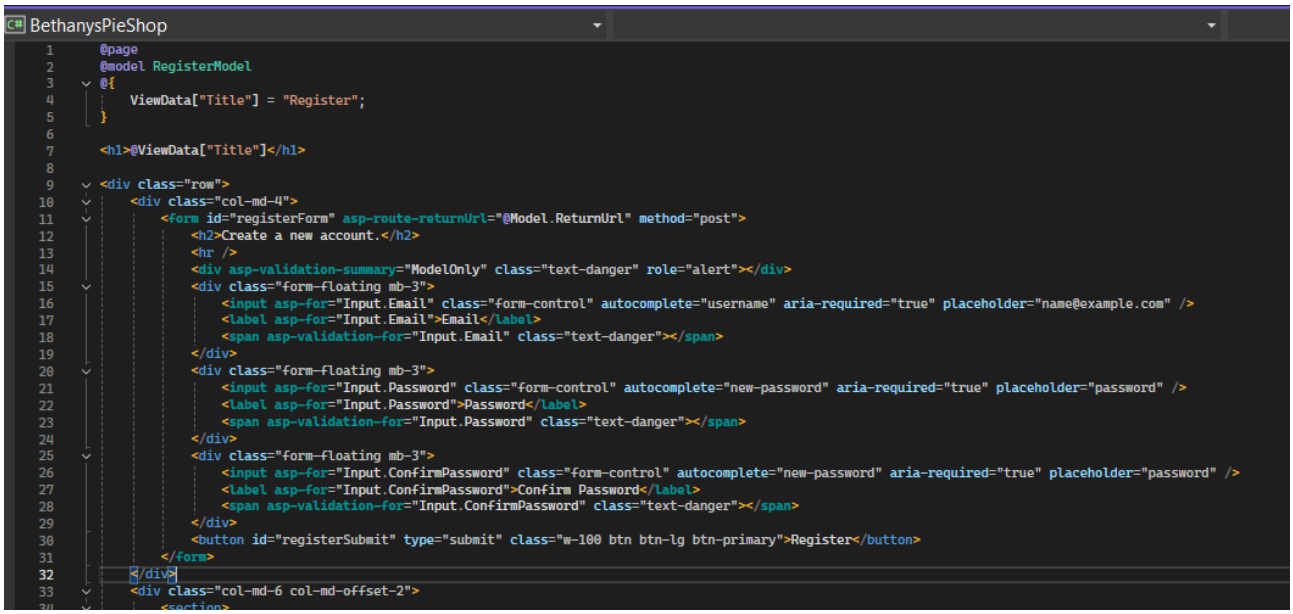
Abbiamo una nuova cartella generata dallo Scafflholding, la cartella **Areas:**




Utilizzata per organizzare le funzionalità correlate alle identità.

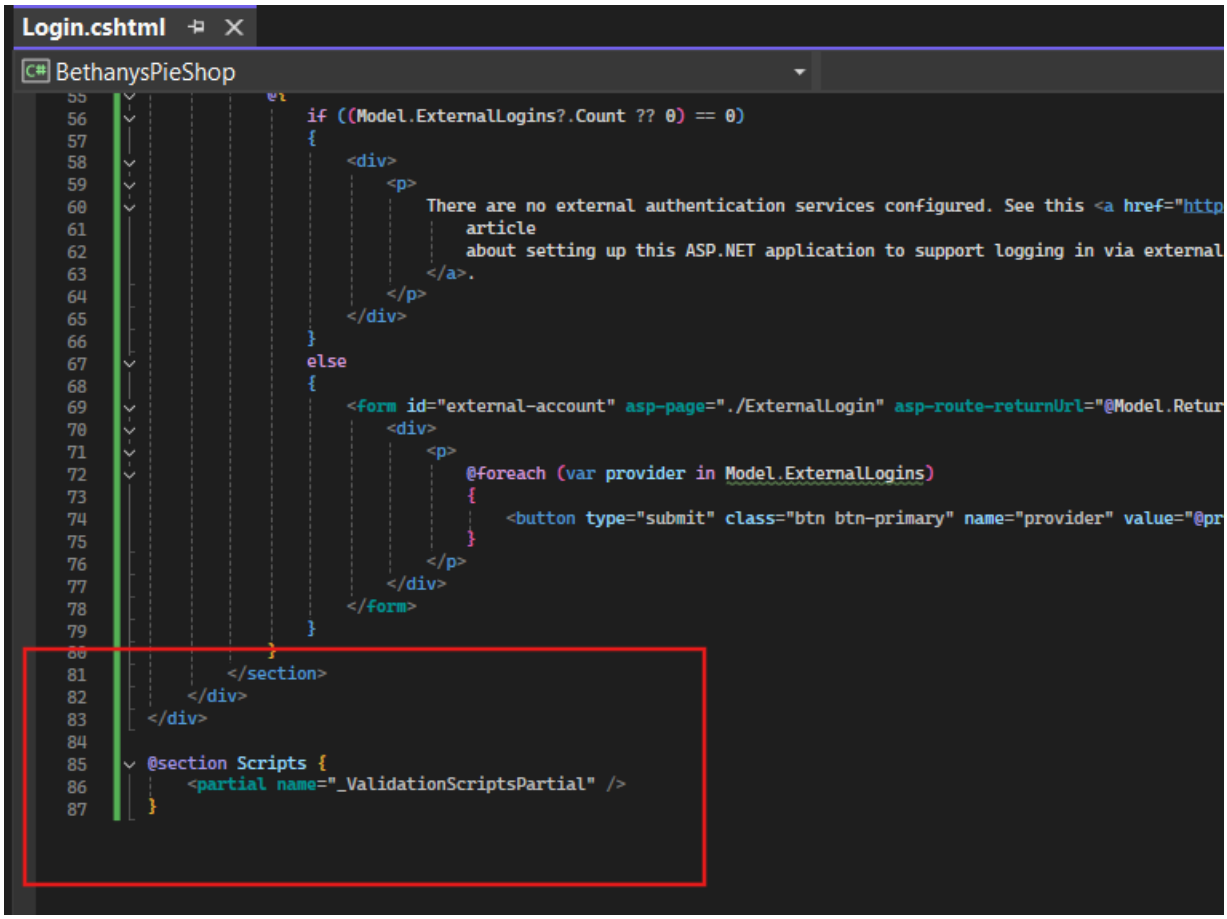
Giungiamo in Register.cshtml:





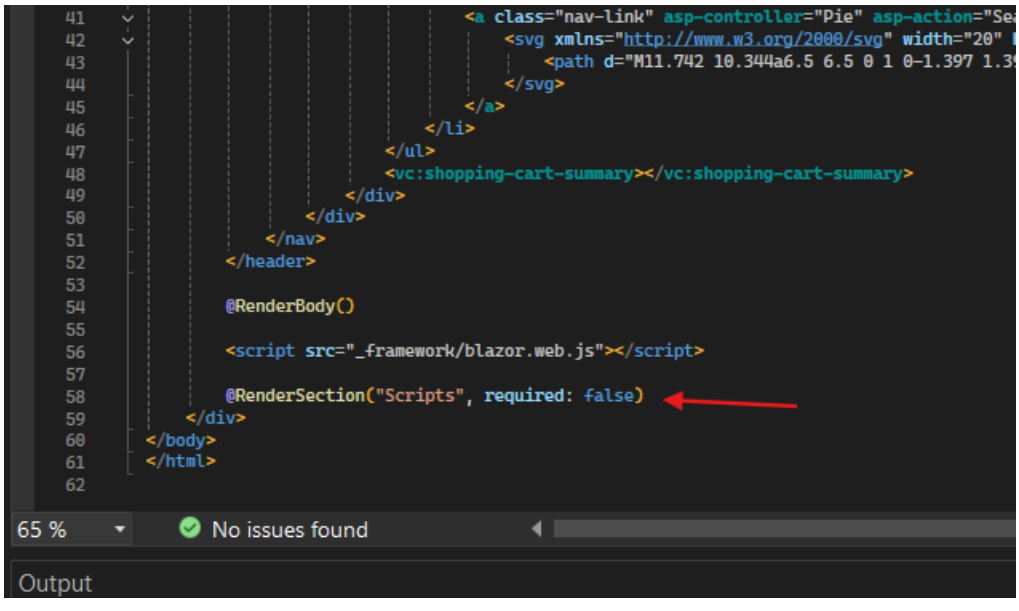

In breve Microsoft ci fornisce molto codice utile per Identity.
Quello che faremo è sovrascrivere il codice dell'interfaccia utente con uno snippet (il nostro codice avrà 40 rige)

Facciamo lo stesso con login.cshtml, in modo che corrispondano al resto della nostra applicazione.




Osservazione: Riga 85 si occupa del rendering, in modo che i vari script possono essere richiamtati dal nostro layout.

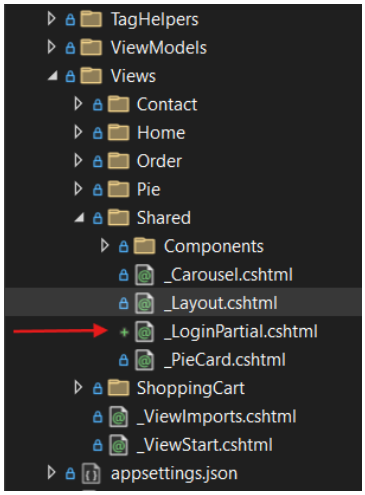
Quindi andiamo in _Layout.cshtml, presente in Views > Shared e aggiungiamo ciò:

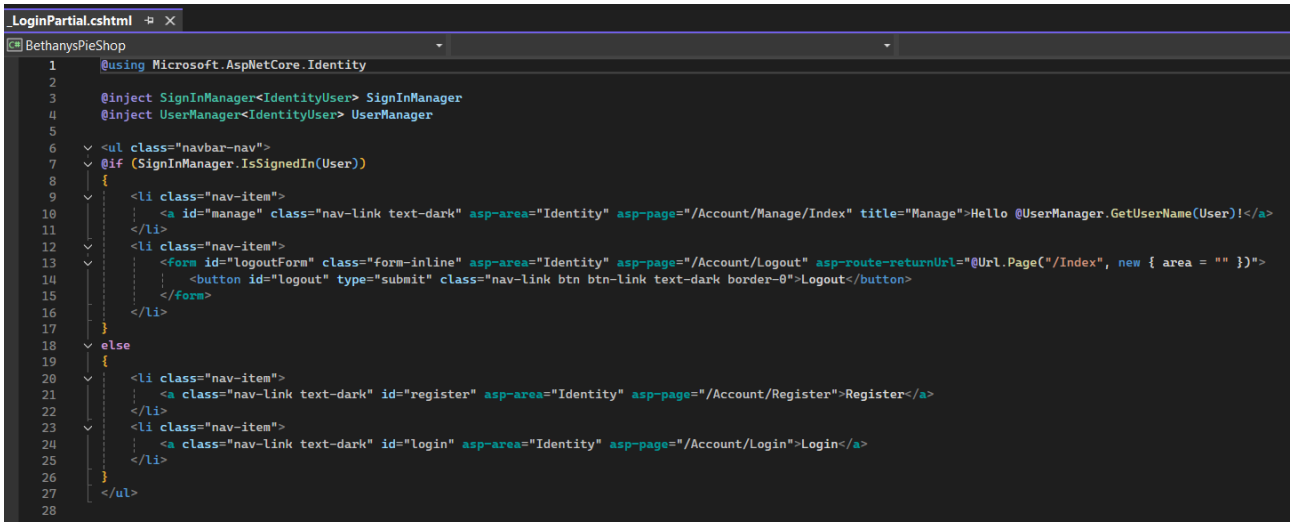



Farà in modo che pagine come login e registro che definiscono una sezione aggiuntiva chiamata Scripts possano

includerla in questo posto nel layout

Andiamo in uun altrofile che è stato generato, sempre in shared, chiamato _LoginPartial.cshtml:




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, difatti 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 ale pagine di registrazione e accesso.

Nel nostro caso rimuoviamo text-dar a riga 21 e 24 per essere coerenti con il Layout di BethanPieShop:

LoginPartial.cshtml

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        <li class="nav-item">
            <a id="manage" class="nav-link" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @UserManager.GetUserName(User)!</a>
        </li>
        <li class="nav-item">
            <form id="logoutForm" class="form-inline" asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })">
                <button id="logout" type="submit" class="nav-link btn btn-link">Logout</button>
            </form>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link" id="register" asp-area="Identity" asp-page="/Account/Register">Register</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" id="login" asp-area="Identity" asp-page="/Account/Login">Login</a>
        </li>
    }
</ul>
```

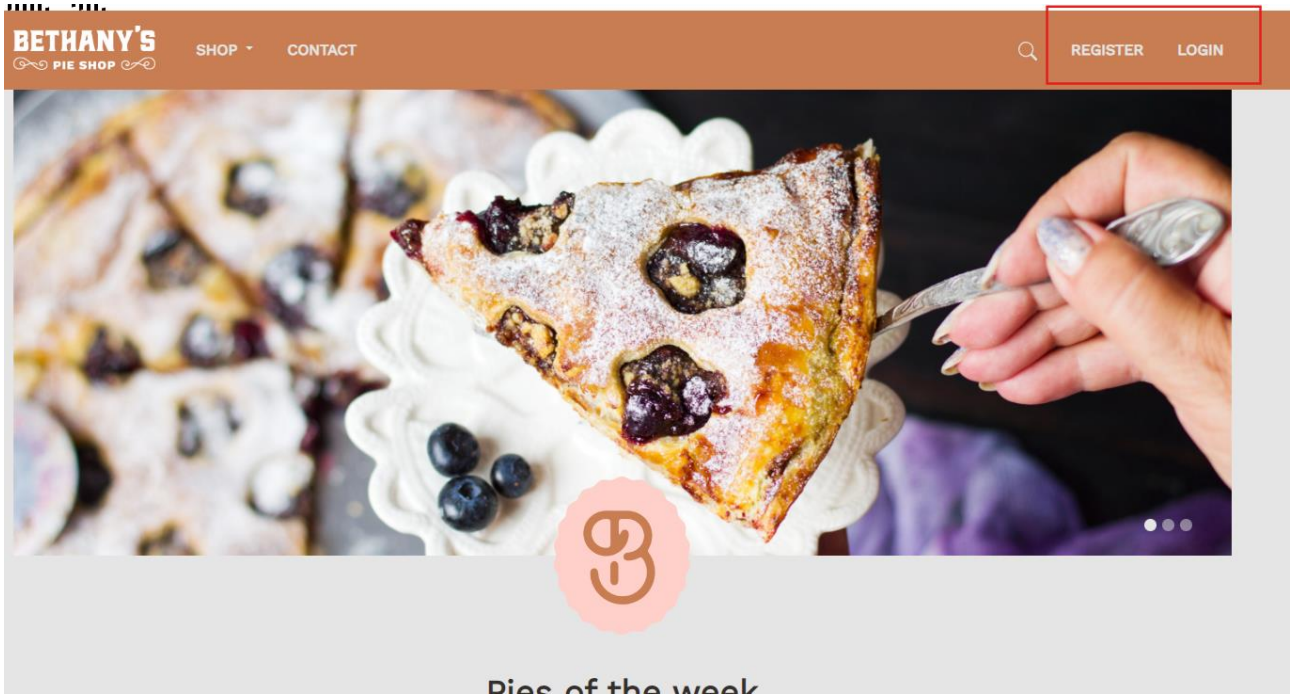
Torniamo al nostro _Layout.cshtml e aggiungiamo LoginPartial:

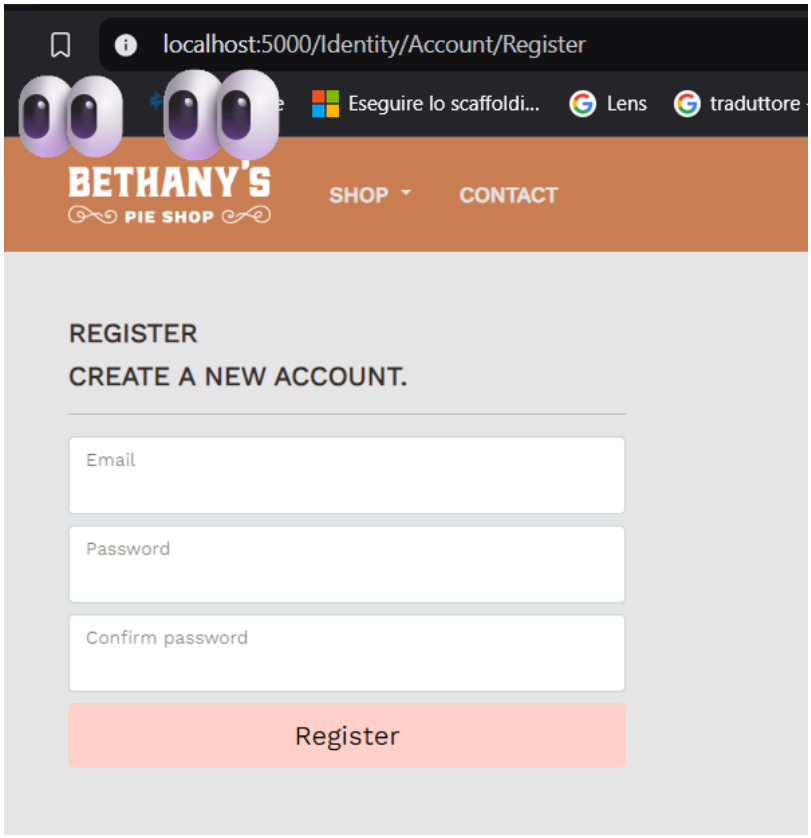
_

```
                    <a class="nav-link" asp-controller="Pie" asp-action="Search">
                        <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20
                            <path d="M11.742 10.344a6.5 6.5 0 1 0-1.397 1.398h-.001c.
                        </svg>
                    </a>
                </li>
            </ul>
            <vc:shopping-cart-summary></vc:shopping-cart-summary>
            <partial name="_LoginPartial" />
        </div>
    </div>
</nav>
</header>

@RenderBody()

<script src="_framework/blazor.web.js"></script>

@RenderSection("Scripts", required: false)
```

Osservazione: Riga 57 si può cancellare se decidiamo di non utilizzare la componente Blazor nel nostro progetto.

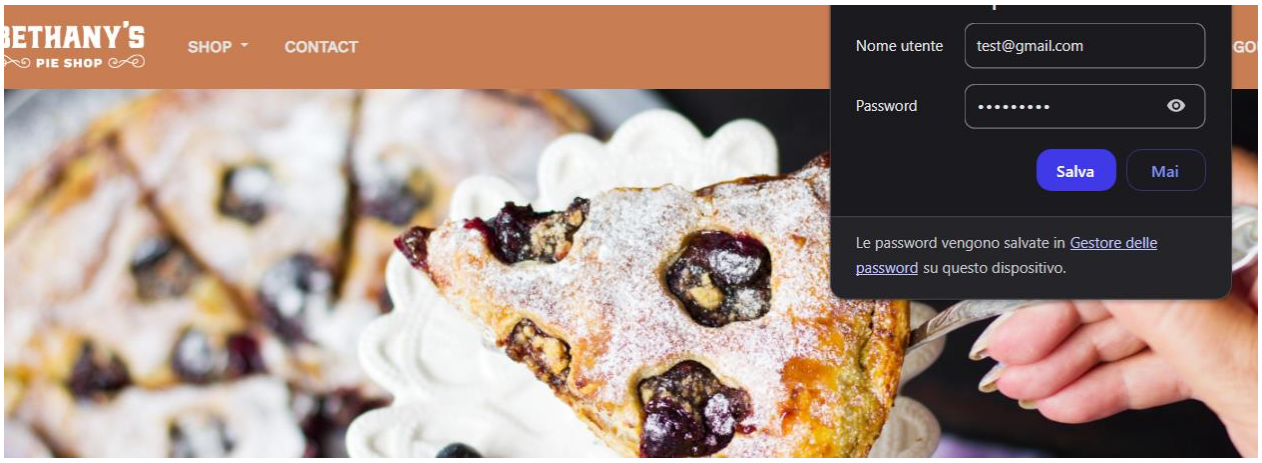
Salviamo, buildiamo ed eseguiamo:



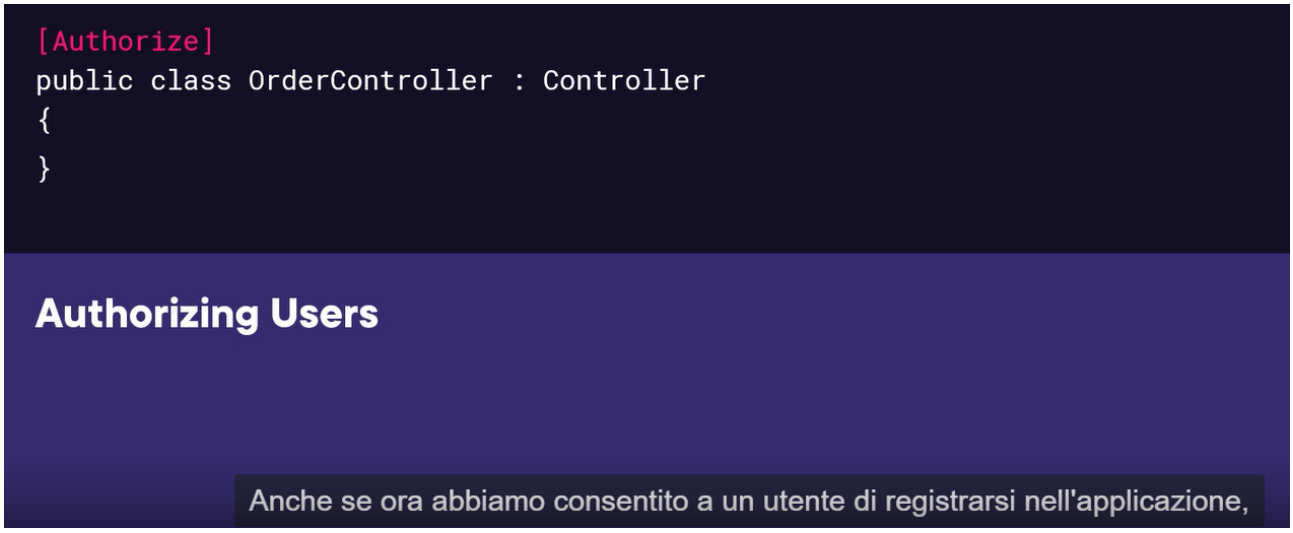

Possiamo notare l'url della pagina Register che inizia con Identity/Account.

Proviamo a registrarci:





**Using Authorization**




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()
    { }
}
```

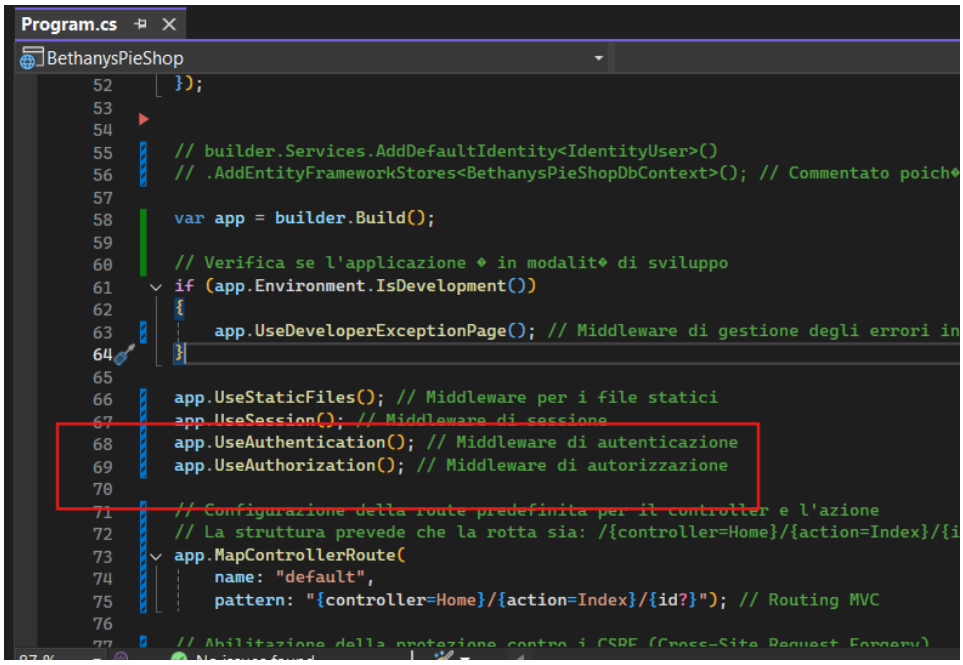
**Action Level**

```
[Authorize(Roles = "Administrator")]
public class OrderController : Controller
{
}
```

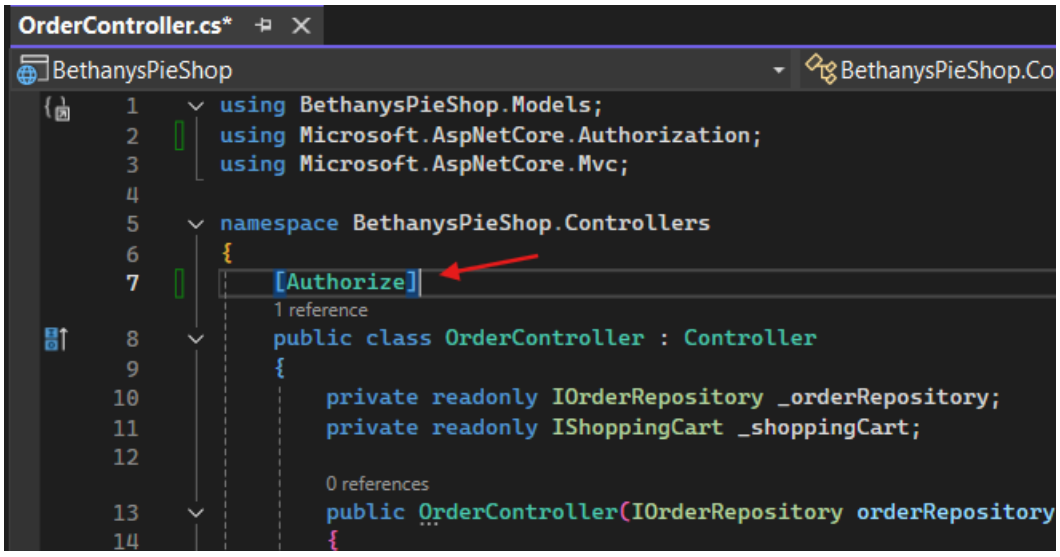
**Combining with Roles**

**Demo: Using Authorization**

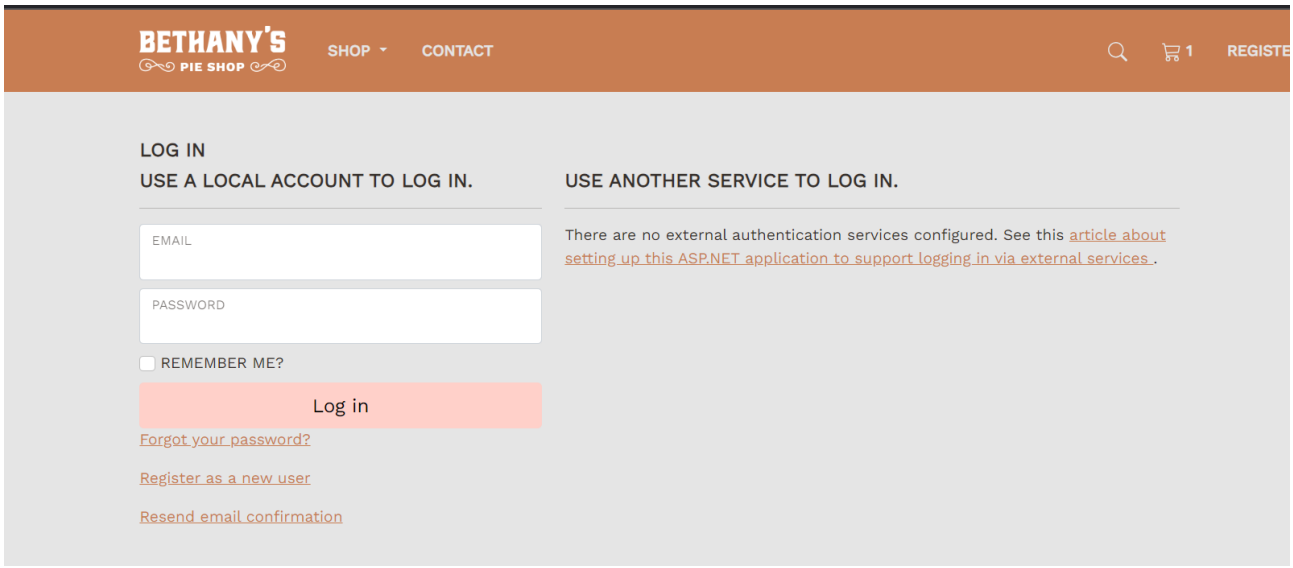
Andiamo in Program.cs e aggiungiamo i seguenti Middlware:




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:




Fatto ciò, noteremo che avviene un redirect alla pagina di registrazione:




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.

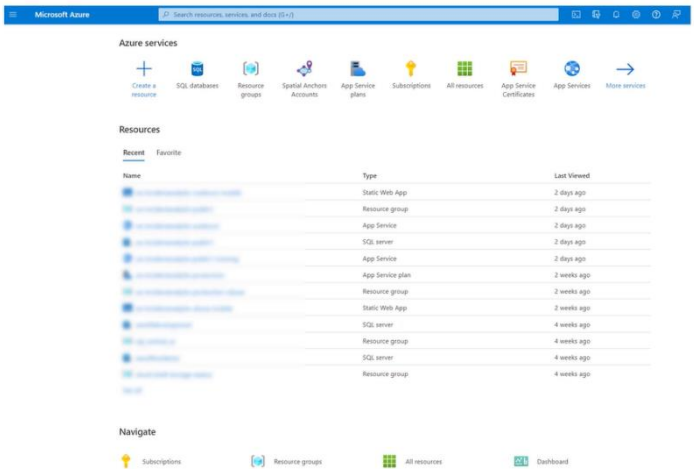
Effettueremo il Deploying usando Azure App Service

**Understanding the Azure App Service**

**Microsoft Azure is Microsoft's cloud computing service.**

**It offers SAAS, PAAS and IAAS and supports many languages and application types.**

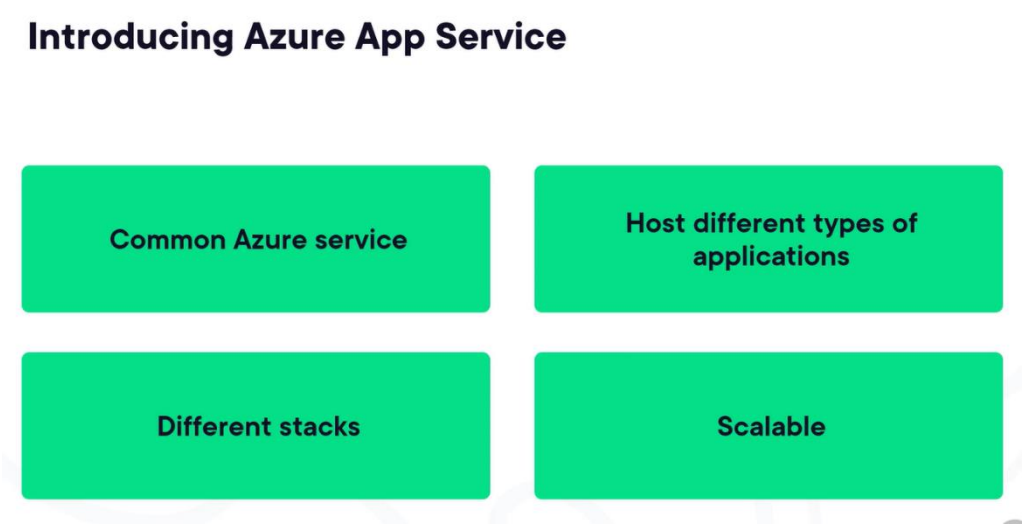
## Azure Portal




## Azure Services

- Compute services
- Messaging
- Identity
- Data services
- Storage
- AI

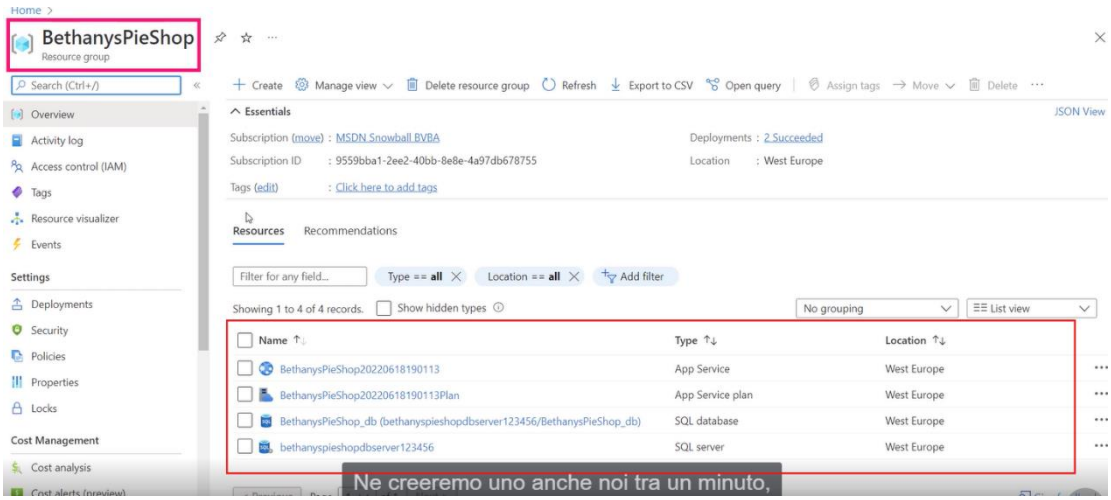
Noi utilizzeremo un app service:



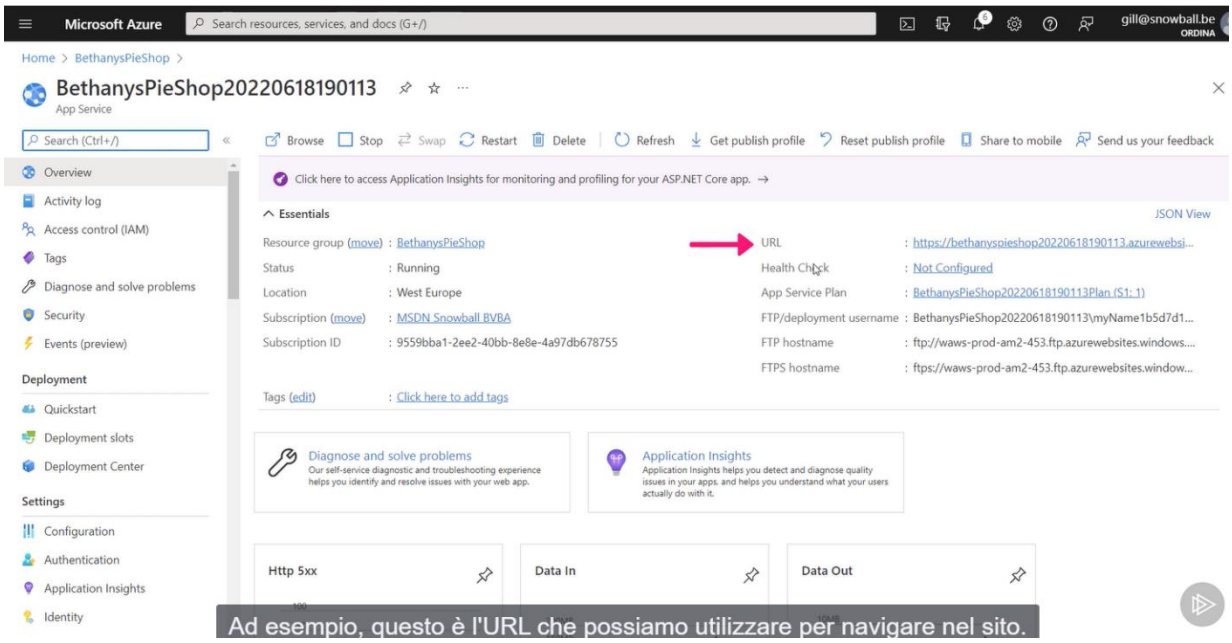

Dobbiamo deployare anche il nostro D.B.:

**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)



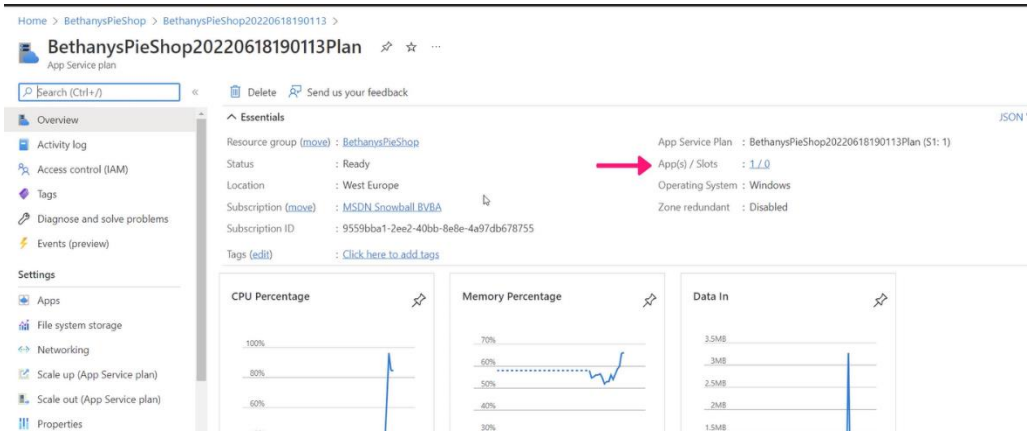

App Service è la più comune:



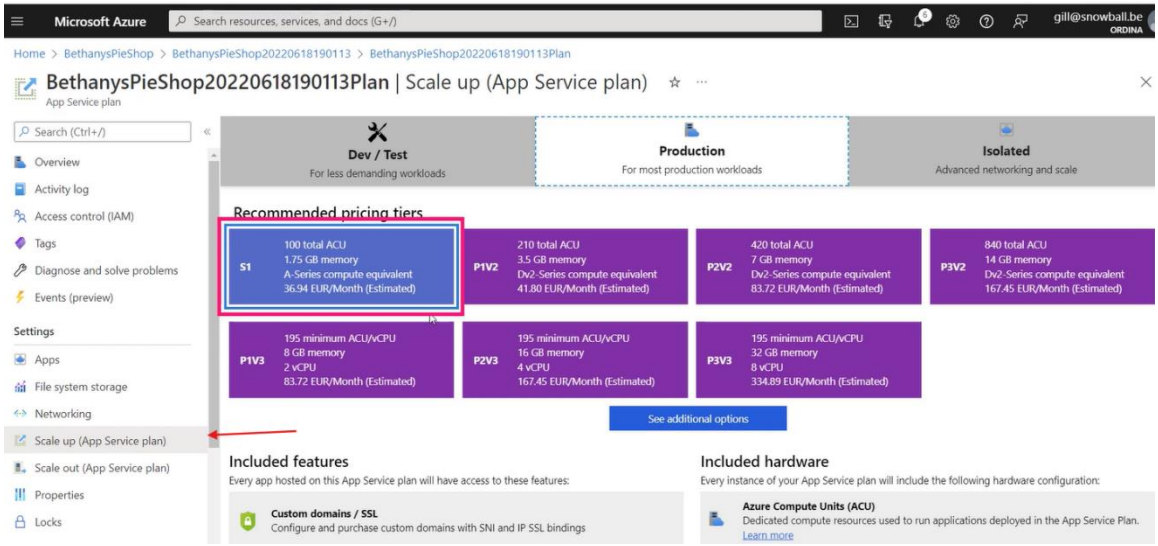

Osservazione: App Service Plan è il server che ospiterà il nostro servizio di app.

Attualmente il piano è un S1, cliccandolo vediamo il piano di servizio, dove viene mostrato che un'app è in esecuzione

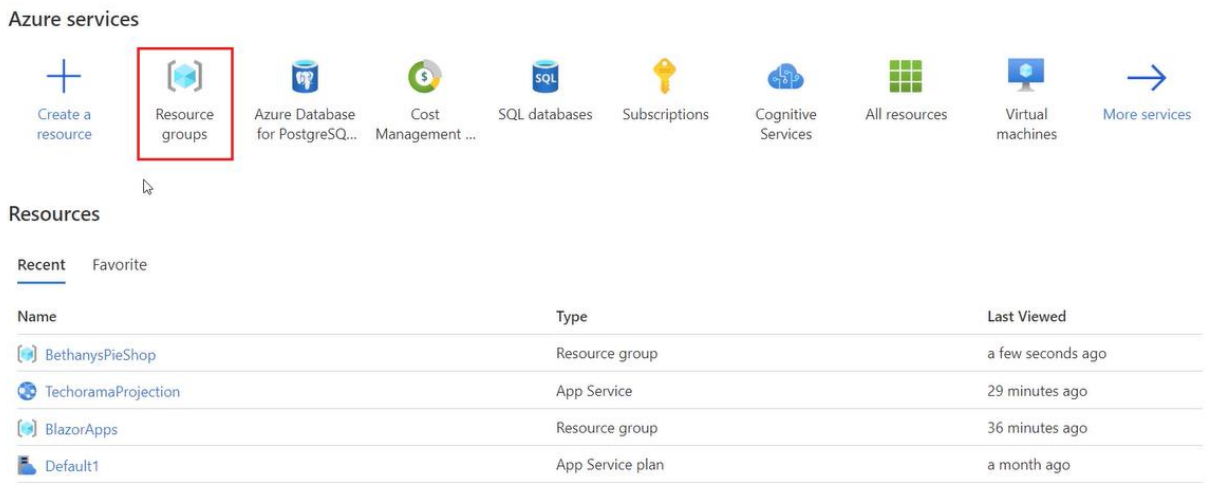
---

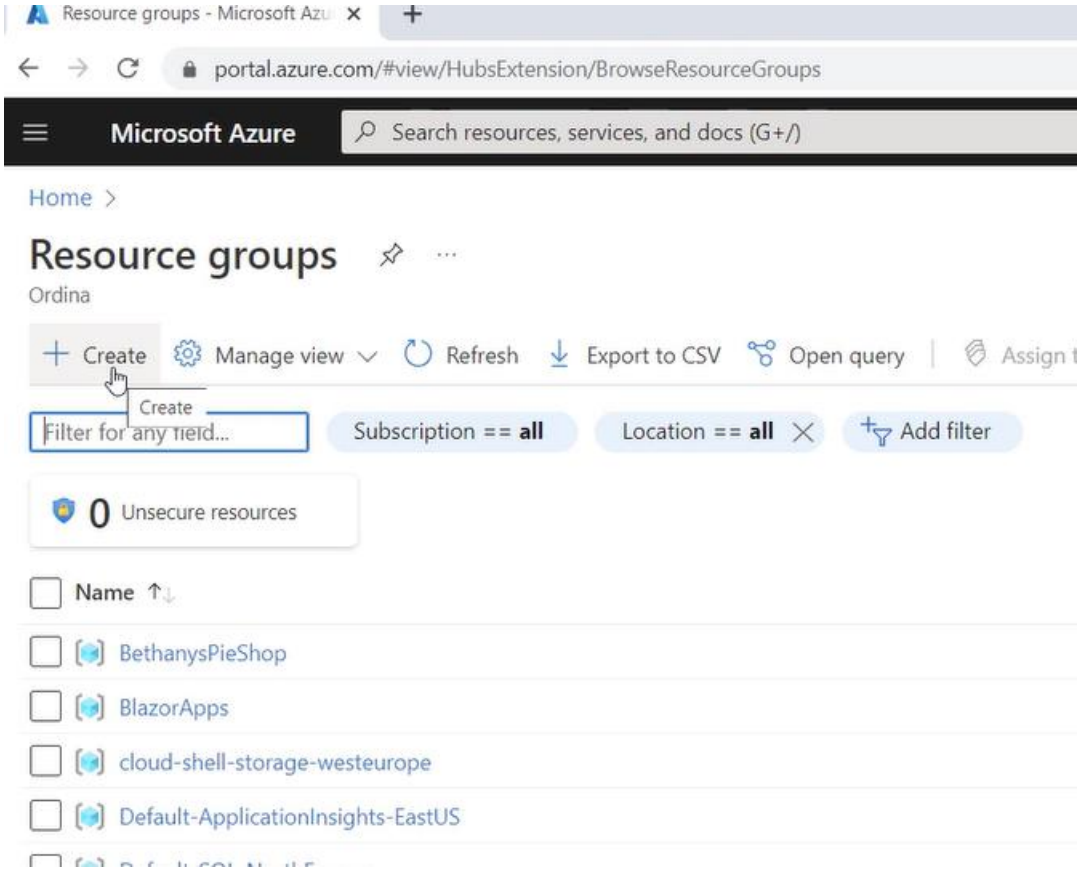


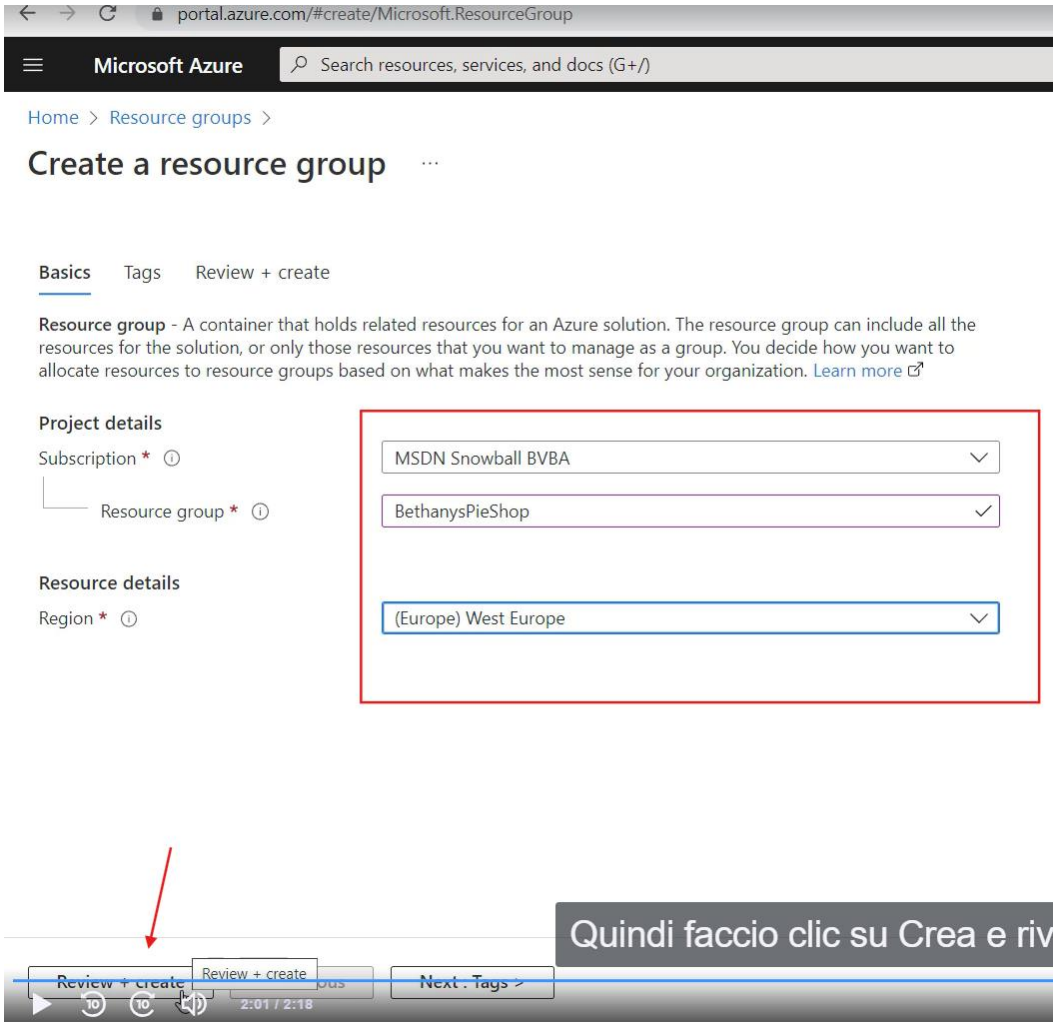

Cliccando Scale Up, nel pannello di scorrimento a sinistra, possiamo constatare il traffico:




Iniziamo quindi con la creazione di un nuovo gruppo di risorse in cui inseriremo quindi tutte le risorse correlate:



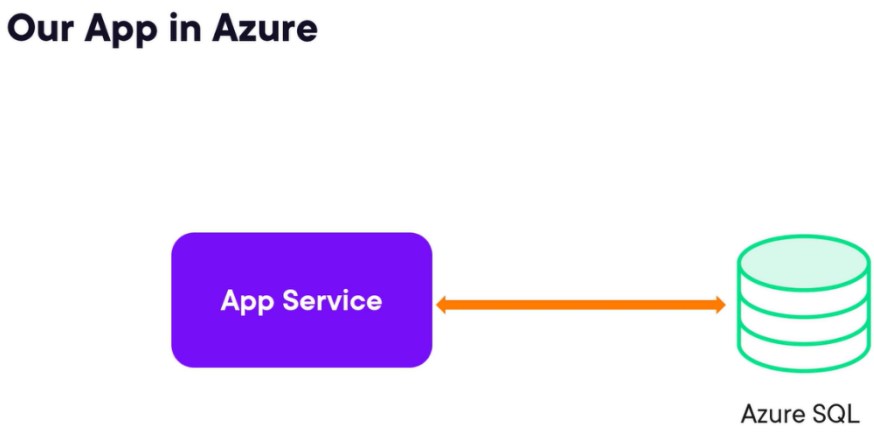

Resource groups - Microsoft Azu
portal.azure.com/#view/HubsExtension/BrowseResourceGroups
Microsoft Azure
Search resources, services, and docs (G+/)
Home
Resource groups
Ordina
Create
Manage view
Refresh
Export to CSV
Open query
Create
Filter for any field...
Subscription == all
Location == all
Add filter
0 Unsecure resources
Name
BethanysPieShop
BlazorApps
cloud-shell-storage-westeurope
Default-ApplicationInsights-EastUS




Verrà mostrato un messaggio di conferma.

**Deploying the Application to an Azure App Service**

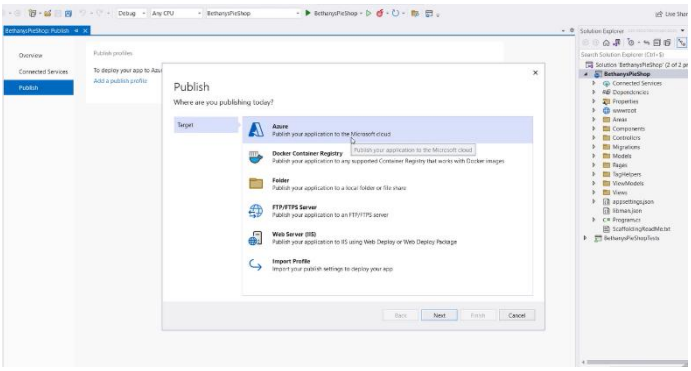
Ora siamo pronti per deployare

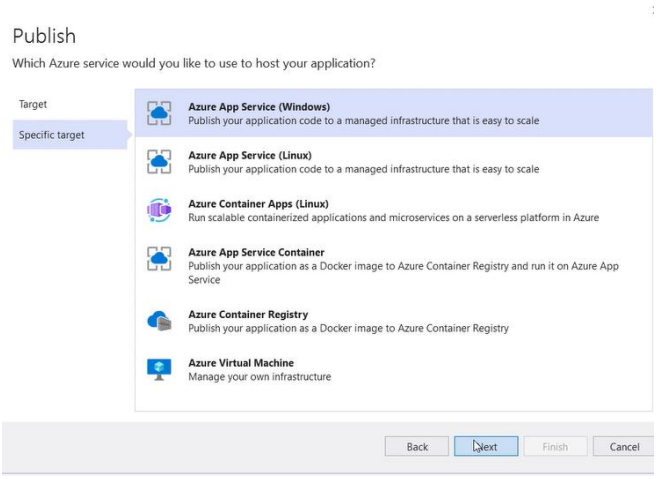


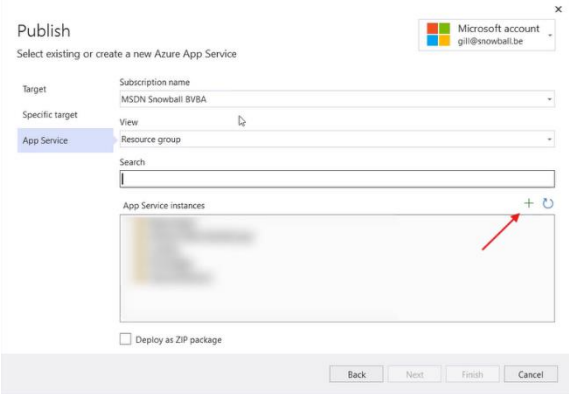
La soluzione/opzione più semplice è Visual Studio publish, click destro del mouse sul progetto in Solution Explorer.

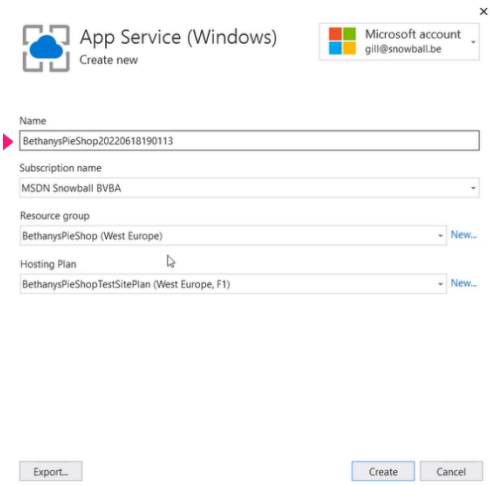
Oppure utilizzando i CI/CD "Continous Integration" "Continuous Deployement".


**Demo: Deploying the Site**

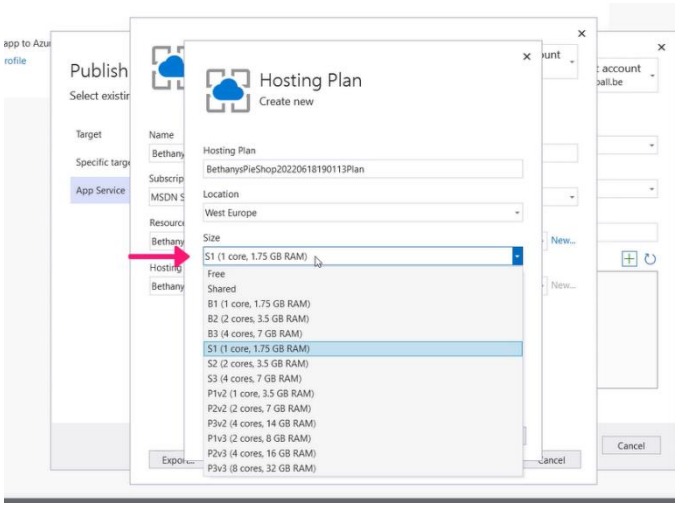




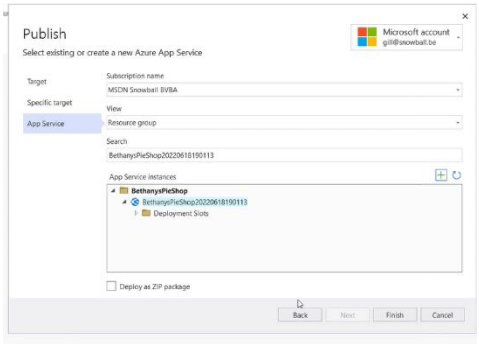




Creiamo un nuovo Hostin Plan cliccando su "New..."

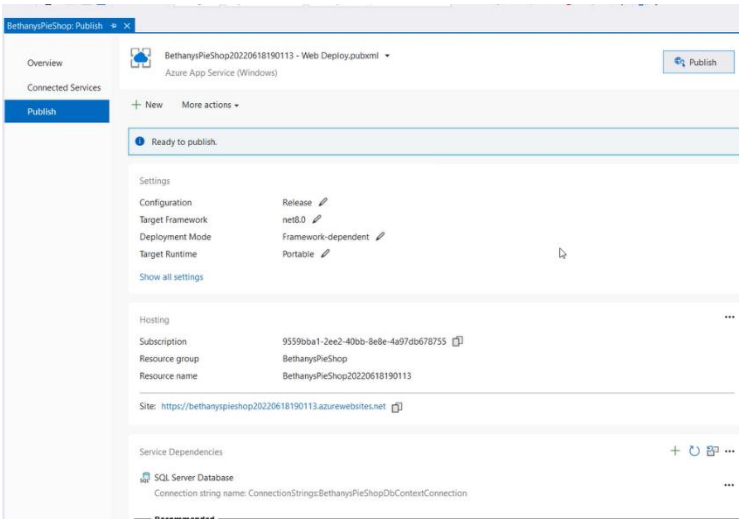



Già size S1 ha un costo mensile, assicurati di eliminare le risorse al termine di questa demo

Okay > Create

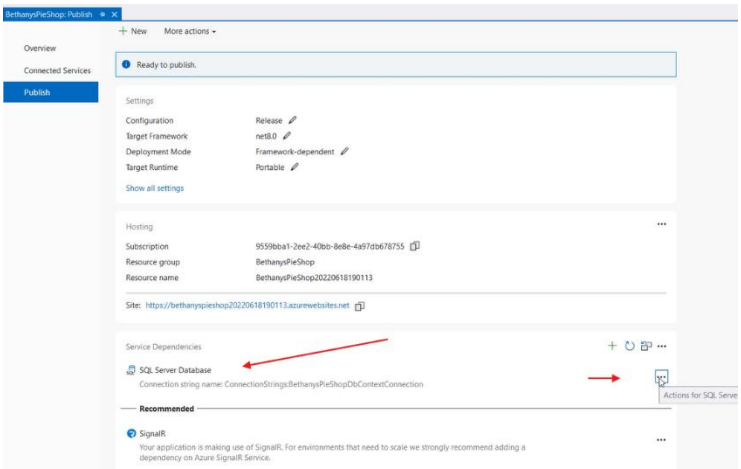



Finish > Close

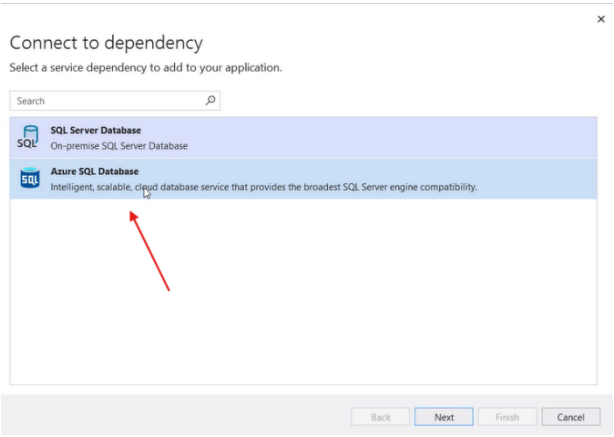



Sarà necessario cliccare **Publish** in alto a destra per ultimare del tutto la pubblicazione

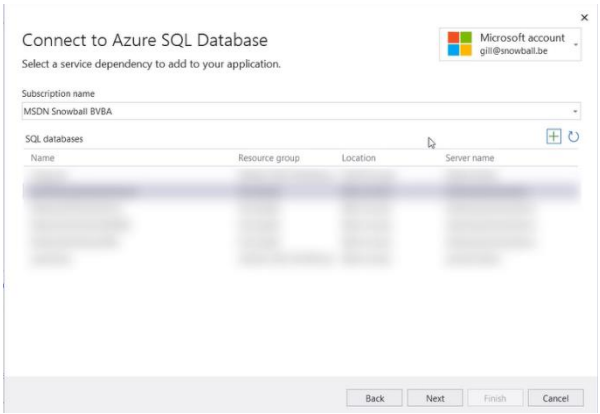
Inoltre possiamo notare che Visual Studio ha rilevato una Service Dependencies SQL Server Database.



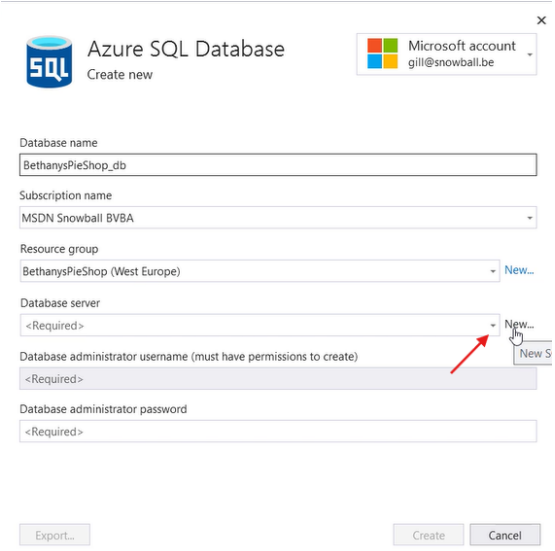
Difatti ci serve un DB, quindi clicchiamo e successivamente selezioniamo "Connect".

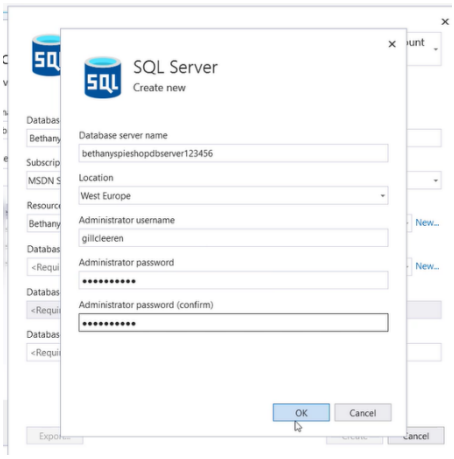



Verranno mostrati i DB esistenti:

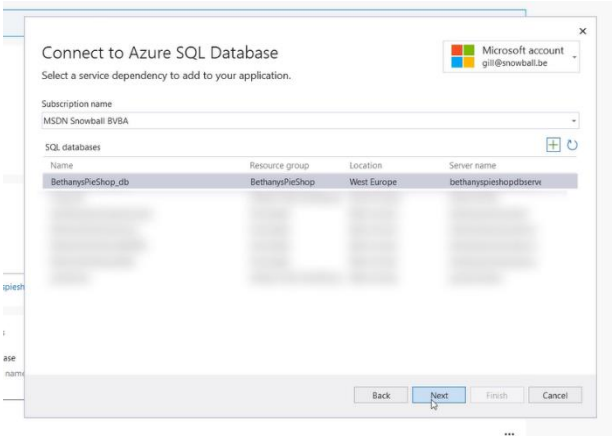



Ma noi creeremo uno nuovo cliccando sul simbolo del '+'

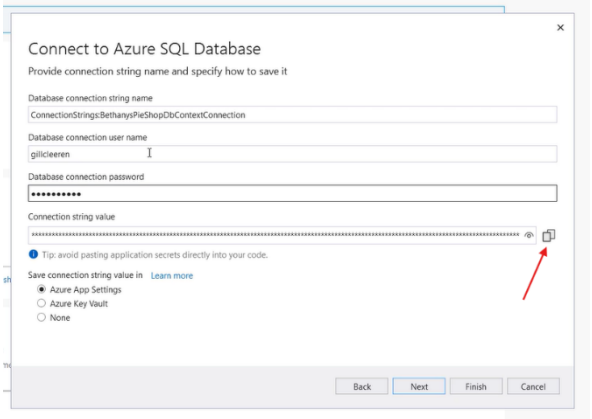





Okay > Create e vedremo il DB creato:

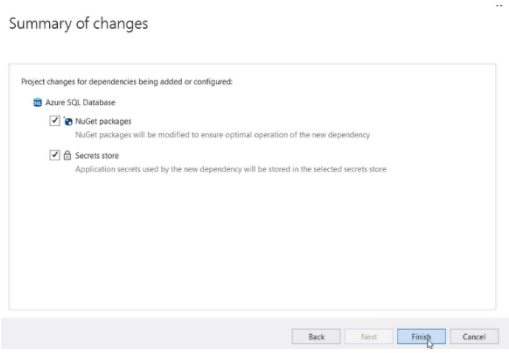



Clicchiamo su "Next" e dovremmo reinserire i dati e password per la connessione al DB:



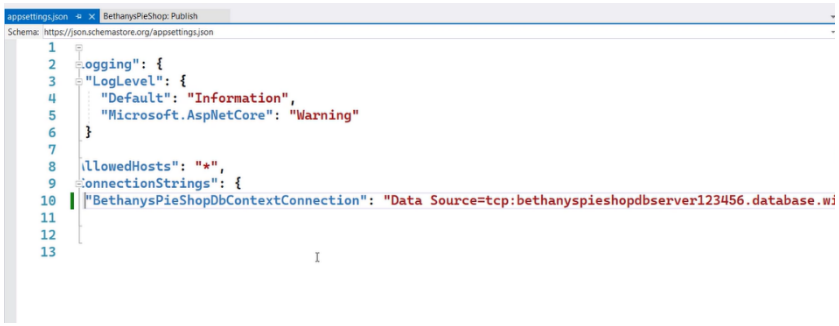

Dopo aver inserito i dati delle nostre credenziali di accesso, copiamo il connection string value.

Click su "Next"



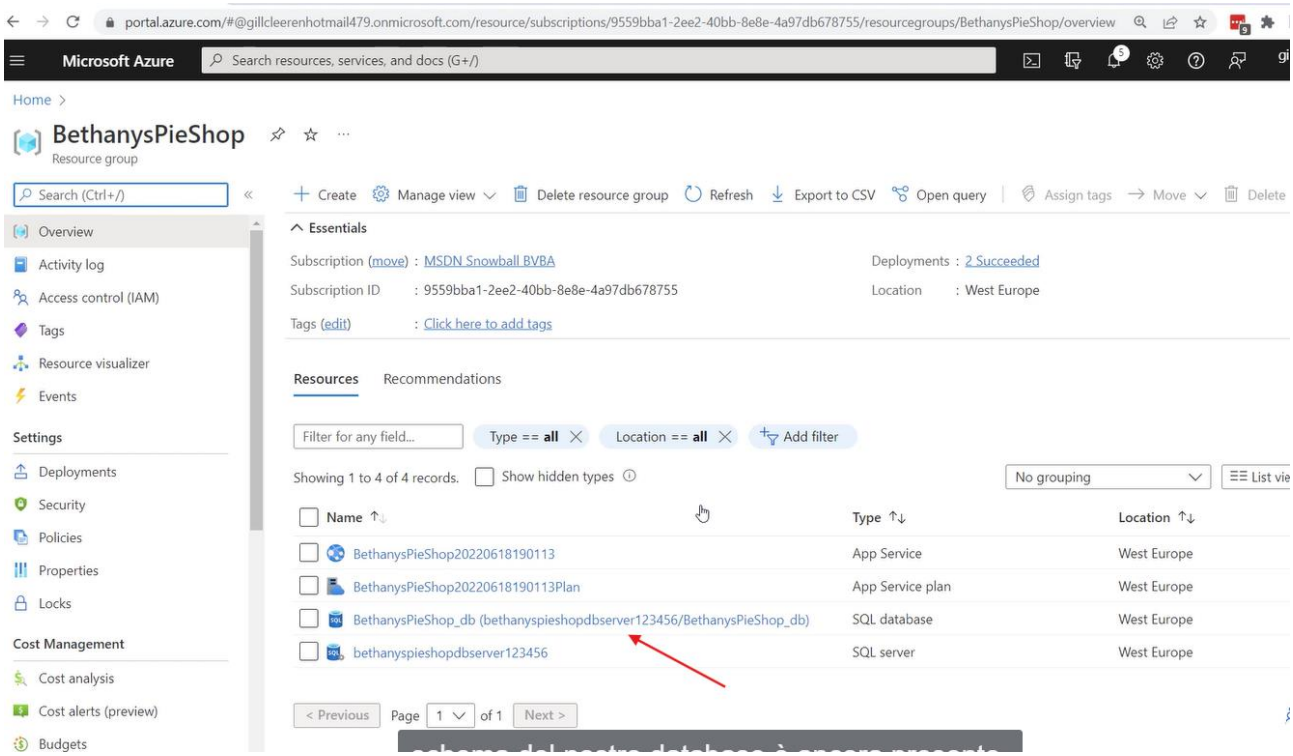

Ora dobbiamo assicurarci che il nostro db schema vengo implementato in quel database.

Quindi andiamo in appsettings.json:
e sostituiamo l'attuale DbContextConnetcion string con quello copiato nel clipboard:

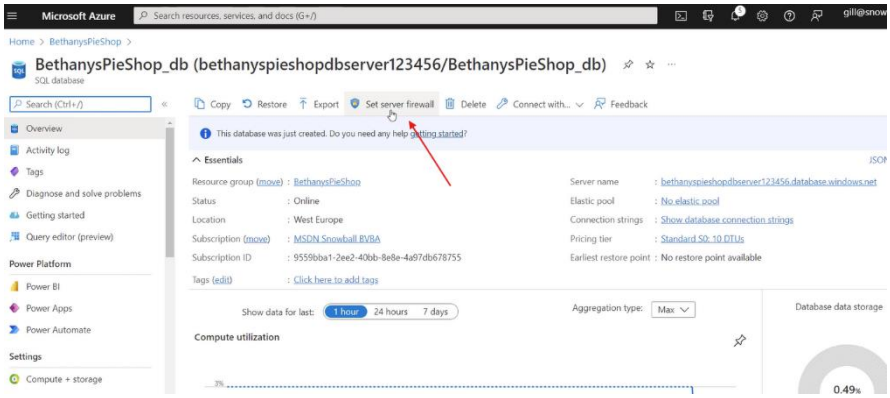



Ora dobbiamo aggiornare il db per far si che avveng ala migrazione sul database in Azure.

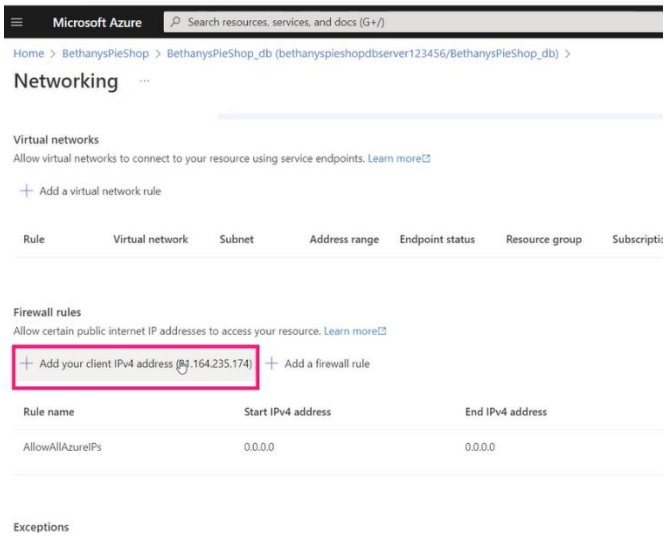
Dobbiamo però prima tornare al portare e aggiornare il firewall, perché per imimpostazione di default non saremo in grado di accederci da VS.



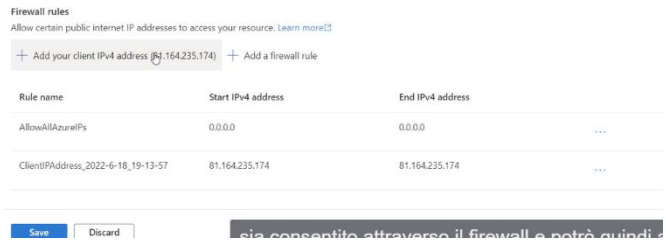

Osservazione: Manca il nostro DB Schema, selezioniamo la risorsa mostrata nell'immagine.




Scrolliamo in basso fino a Firewall rules e aggiungiamo l'IP del nostro client corrente.

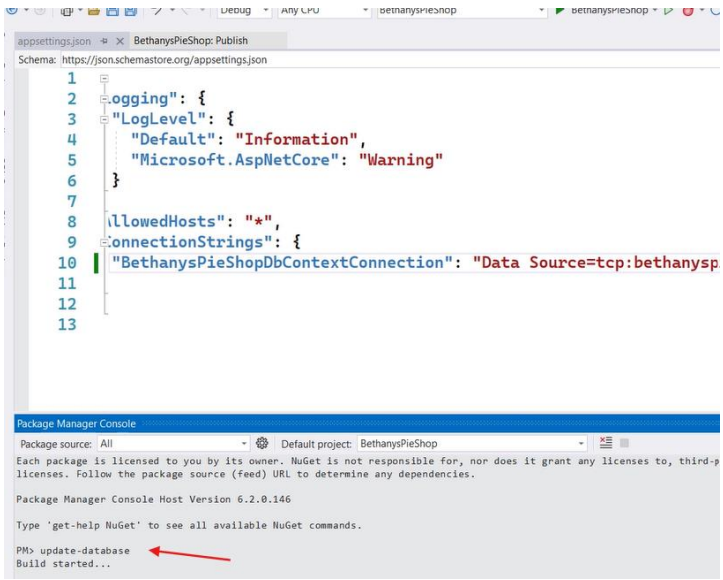


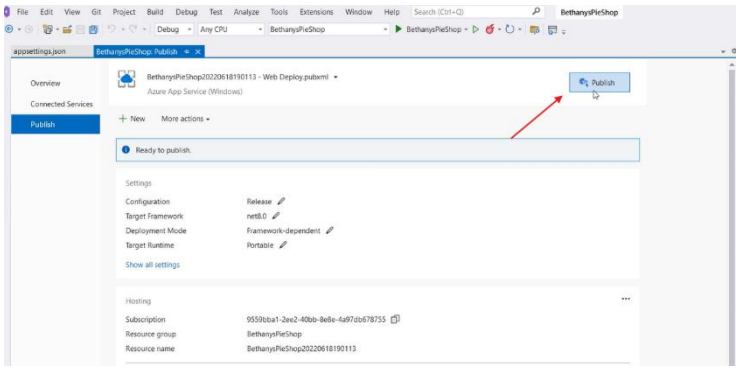

Verrà mostrato:




Clicchiamo su "Save".

Ora torniamo in VS e apriamo il Package Manager Console:






Finish! Il nostro sito è stato pubblicato con successo su Azure: