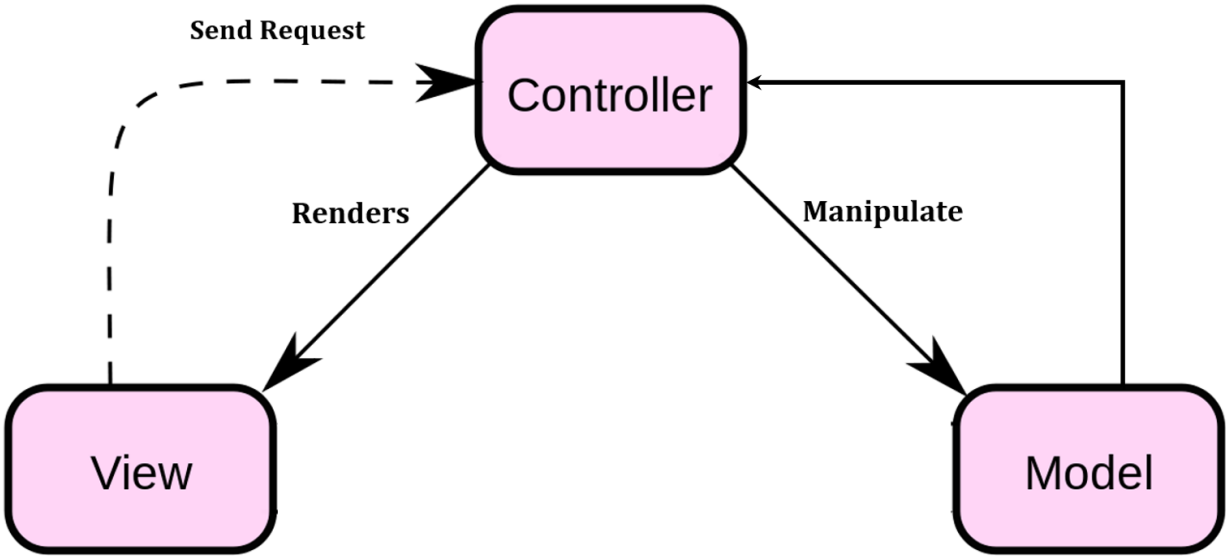
Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC
Curs 3

Cuprins:

Arhitectura MVC

Model-View-Controller (**MVC**) este un model arhitectural utilizat în dezvoltarea aplicațiilor. Succesul modelului se datorează **izolării logicii aplicației** (business logic) **față de interfața cu utilizatorul**, rezultând o aplicație unde aspectul vizual și nivelele inferioare ale regulilor de business sunt mai ușor de modificat, fără a afecta alte nivele.



Model (Stratul business – prelucrarea datelor)

Modelul este responsabil cu **gestionarea datelor** din aplicație și **manipularea acestora**. Acesta răspunde cererilor care vin din View, prin intermediul Controller-ului, Modelul comunicând direct doar cu Controller-ul. Este cel mai de jos nivel care se ocupă cu **procesarea** și **manipularea** datelor, reprezentând nucleul aplicației, fiind cel care realizează legătura cu baza de date.

Controller

Controller-ul este reprezentat de clase, fiind componenta care controlează accesul la aplicație.

În Controller:

- sunt procesate requesturile HTTP;
- se citesc datele introduse de utilizator;
- are loc procesarea prin trimiterea datelor către Model - unde se execută operațiile;
- se trimite răspunsul către View;

Așadar, Controller-ul comunică cu Modelul și cu View-ul.

Fiecare Controller conține așa numitele **Action-uri (Actions)**, **reprezentate de metode**. Aceste metode sunt **publice**, se definesc în interiorul unui Controller și sunt accesate în momentul în care are loc un request prin intermediul unei rute.

Metodele trebuie să fie **publice** pentru că MVC framework-ul să le poată accesa și invoca atunci când gestionează cererile HTTP. În arhitectura MVC, atunci când o cerere HTTP este trimisă către aplicație, ASP.NET Core folosește mecanisme de rutare pentru a potrivi URL-ul cu un **Controller** și o **metodă** corespunzătoare (**Action**). Doar metodele publice sunt accesibile pentru rutare și pot fi invocate în urma unei cereri HTTP. Dacă metodele ar fi private sau protejate, ele nu ar fi vizibile pentru motorul de rutare și, prin urmare, nu ar putea fi folosite pentru a răspunde la cereri. Metodele declarate *private* sau *protected* vor rămâne disponibile doar pentru logica internă a Controller-ului și nu vor putea fi folosite pentru gestionarea cererilor externe.

View (interfața cu utilizatorul)

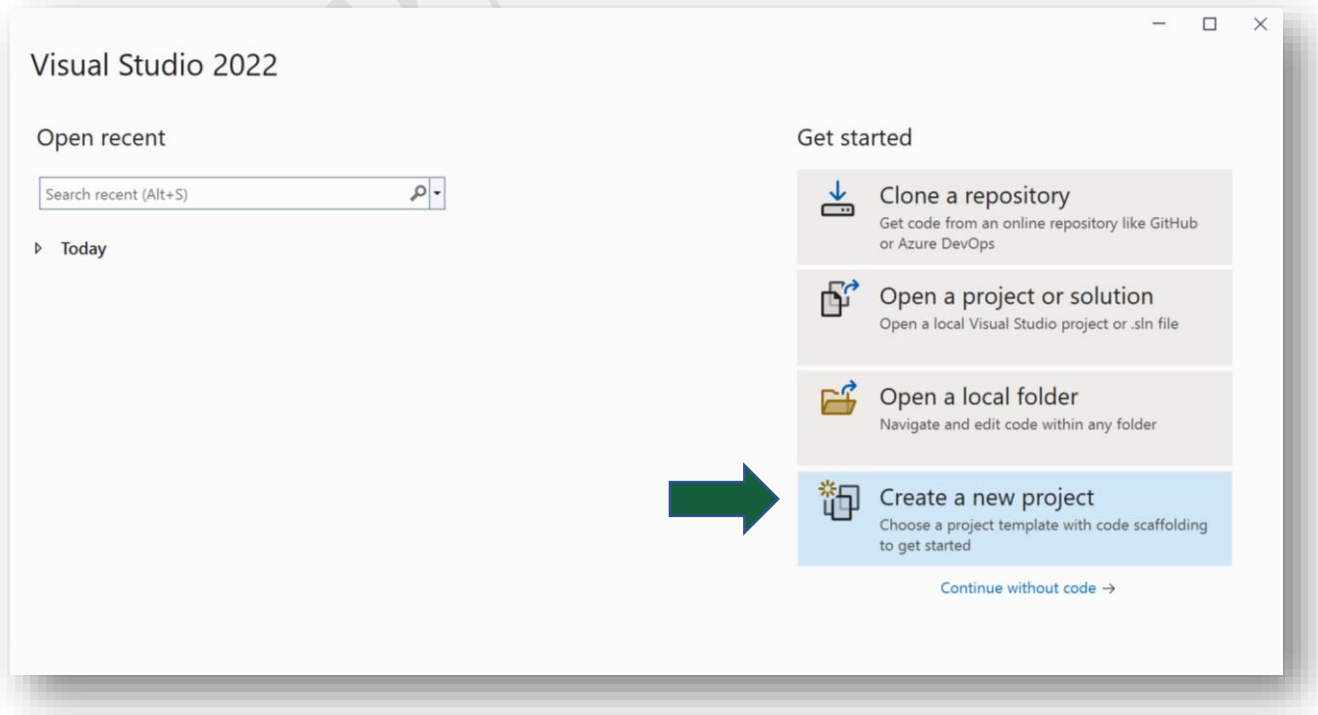
View-ul reprezintă interfața cu utilizatorul, fiind componenta arhitecturii MVC cu care utilizatorii interacționează prin intermediul browser-ului.

În View se afișeaza datele, adică înregistrările din baza de date și informațiile generate de aplicație.

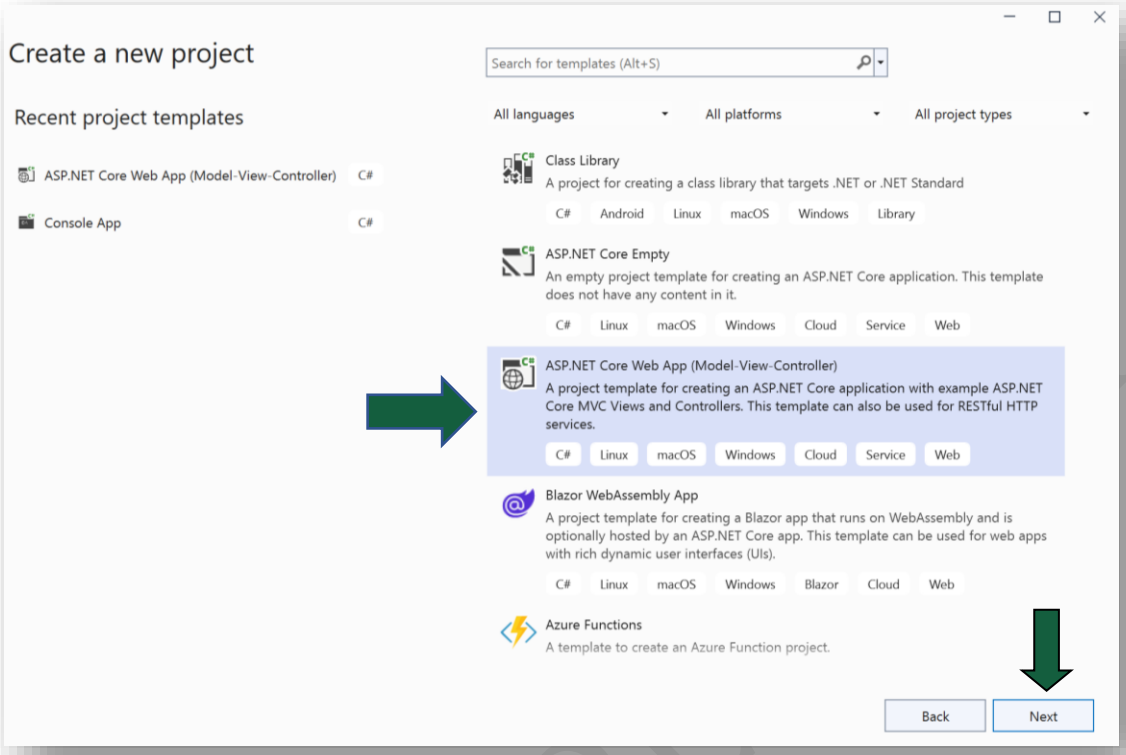
View-ul poate conține HTML static sau chiar HTML trimis din Controller (HTML dinamic). În cadrul arhitecturii MVC, View-ul comunică doar cu Controller-ul, iar cu Modelul comunică indirect tot prin intermediul Controller-ului.

Crearea unei aplicații în ASP.NET Core 9.0 (Visual Studio 2022)

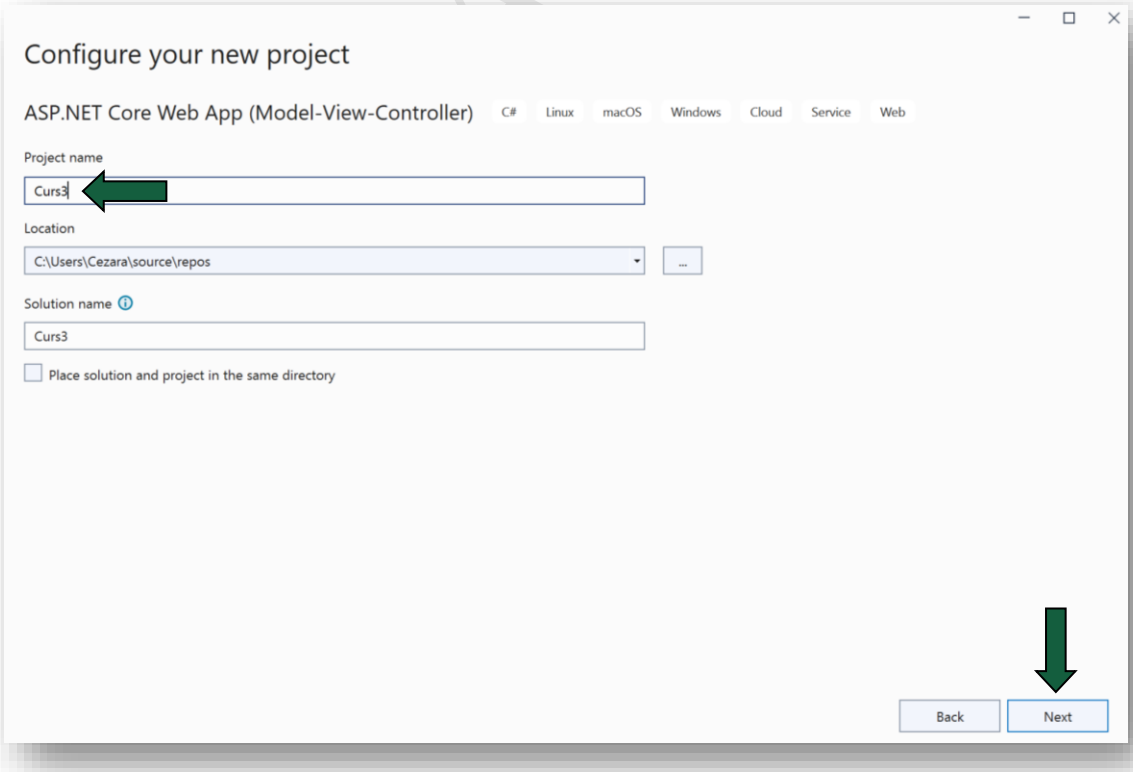
PASUL 1:



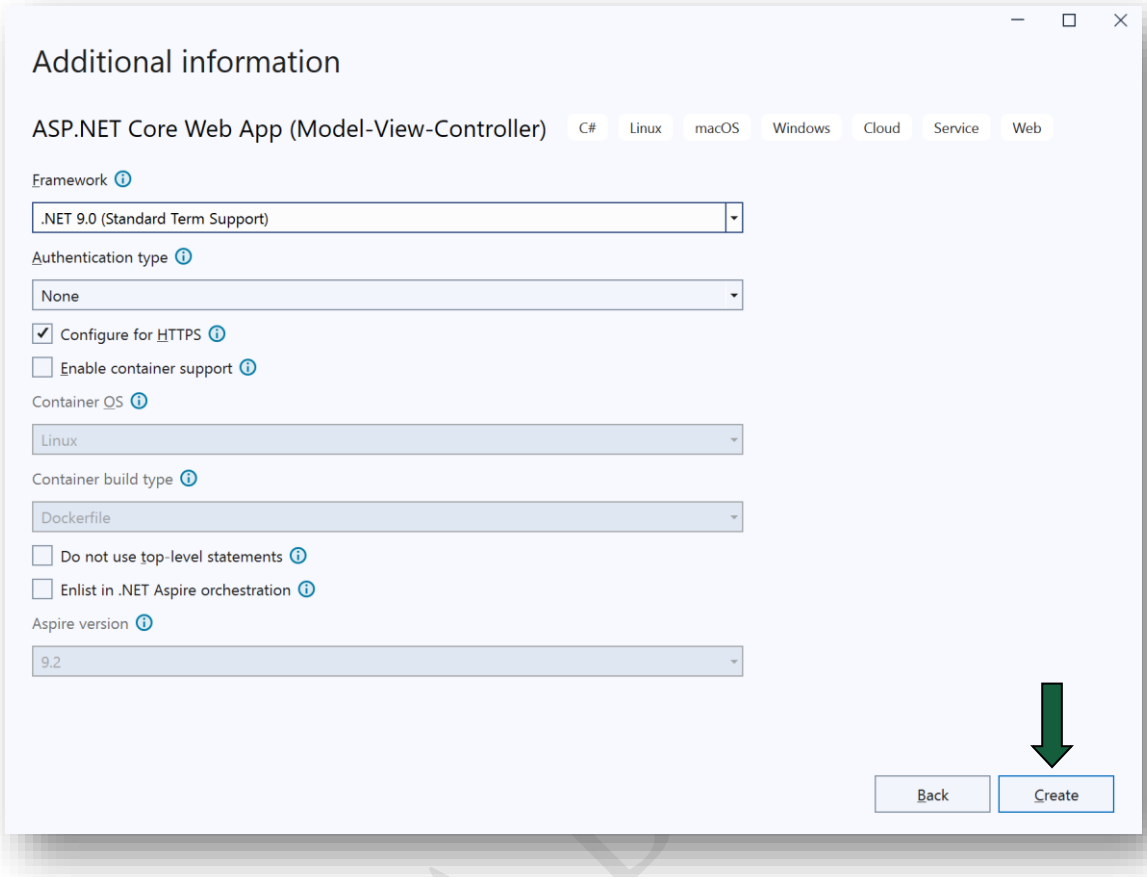
PASUL 2:



PASUL 3:

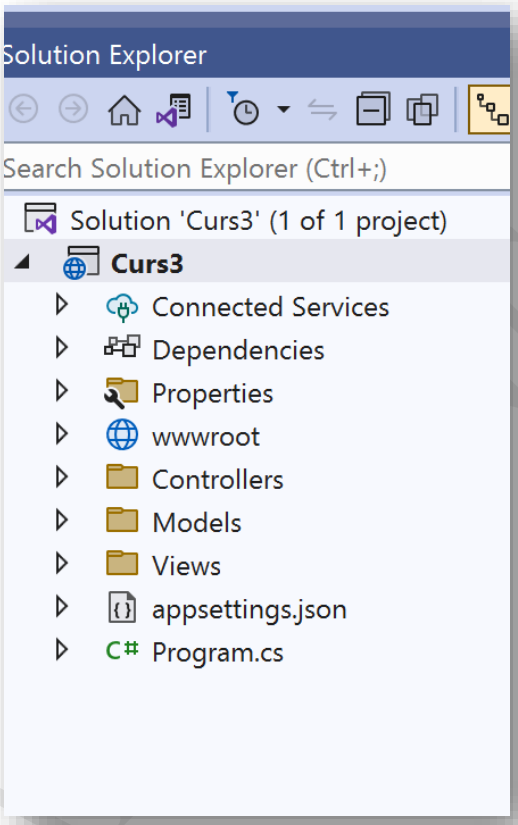


PASUL 4:



Structura unui proiect MVC – Sistemul de fișiere

În imaginea următoare este prezentată structura unui proiect MVC, realizat în ASP.NET Core 8.0. Structura proiectului este reprezentată de sistemul de fișiere și foldere pe care le conține aplicația.



- **wwwroot –** conține fișierele statice precum librării (ex: Bootstrap), imagini, scripturi (css, js);

- **Controllers –** include toate fișierele de tip Controller. Acestea sunt fișiere care conțin cod **c#** și au extensia **.cs**. MVC impune ca numele tuturor controller-elor să conțină la final cuvântul **Controller**;

- **Models –** folderul conține modelele aplicației;

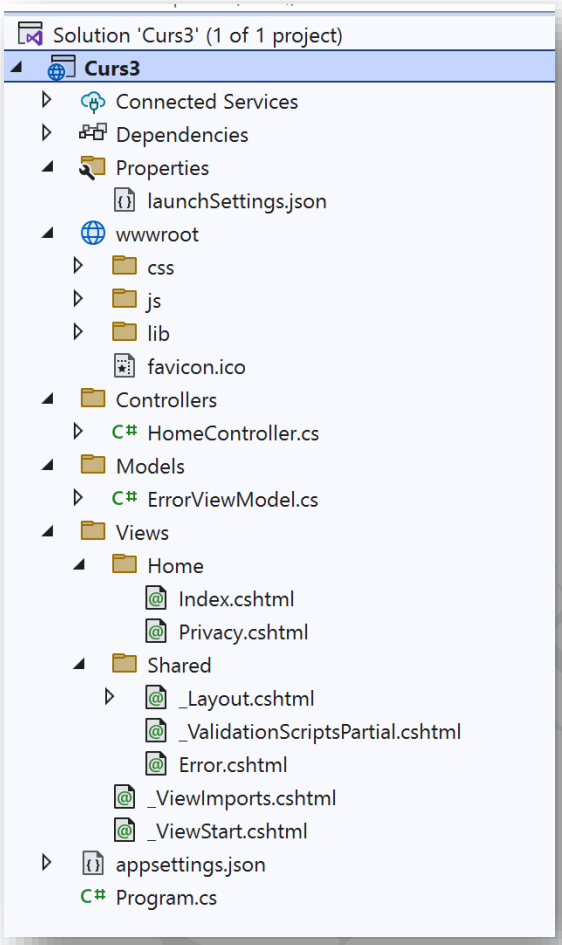
- **Views –** folderul conține fișierele de tip View (interfața cu utilizatorul) ale aplicației;

- **appsettings.json –** în acest fișier se află **setările pentru configurarea aplicației**. Acest fișier permite stocarea de setări și valori de configurare într-un format JSON, care poate fi citit și utilizat de către aplicație în timpul rulării. Este utilizat pentru a defini setări precum ***detaliile de conectare la baza de date***, ***setările de logging*** (procesul prin care aplicația înregistrează evenimente pe măsură ce funcționează - aceste evenimente sunt păstrate în fișiere de log sau trimise către servicii externe de monitorizare), ***API keys***, ***setări de acces pentru diverse servicii externe*** *(ex: email, OAuth)*, etc.

- **Program.cs –** este **punctul de pornire al aplicației** care se accesează imediat după ce aplicația este rulată. De asemenea, în acest fișier se configurează modulele aplicației: ***domeniul aplicației*** (host), ***serverul web*** (IIS - Internet Information Services, Nginx, etc), ***modulul de autentificare***, etc;

Internet Information Services (IIS) este un **server web** creat de Microsoft, care rulează pe sistemul de operare Windows și este folosit pentru a găzdui și livra aplicații web și servicii web pe Internet sau într-o rețea locală. Rolul principal al IIS este de a gestiona cererile HTTP și de a trimite răspunsurile corespunzătoare către clienți (browser-ul utilizatorului, de exemplu).

În imaginea următoare se pot observa toate folderele și fișierele existente într-un proiect nou creat. Tot sistemul de fișiere o să fie studiat pe rând în cursurile următoare.



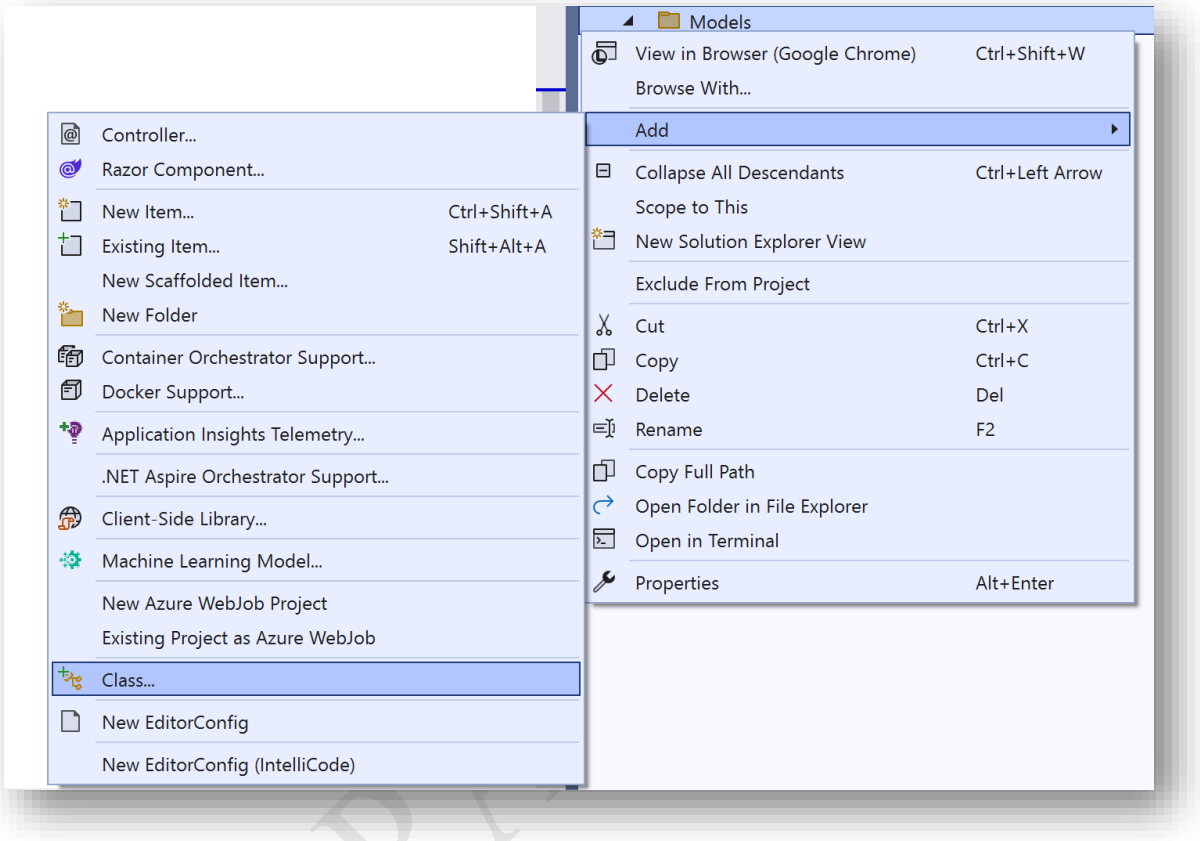
Exemplu de implementare MVC

În continuare, vom considera clasa **Product** și vom implementa fiecare componentă a arhitecturii, neimplementând încă baza de date.

Se consideră clasa ***Product***. Un produs are id – codul unic de identificare, denumire, descriere și preț. Să se adauge produse utilizând o listă (List< >), după care să se afișeze lista tuturor produselor.

PASUL 1

Se adaugă clasa Product, astfel: click dreapta pe folderul Models → Add → Class



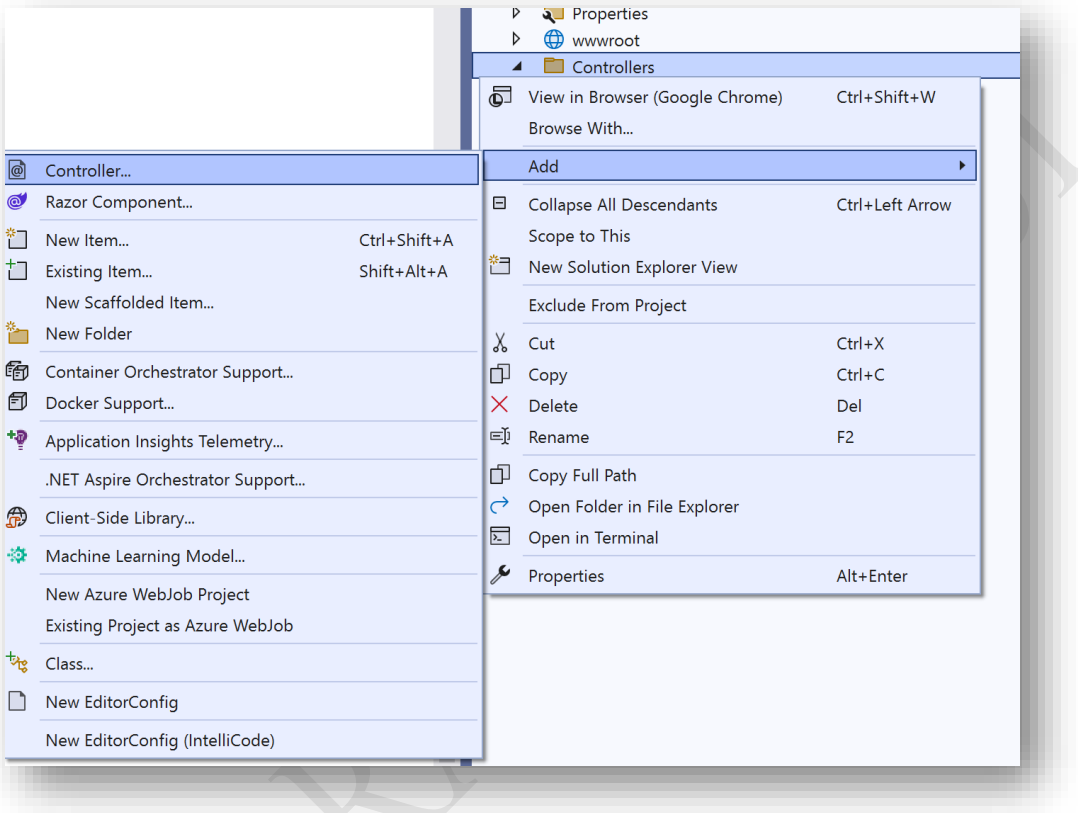
PASUL 2

Se implementează clasa Product împreună cu toate proprietățile.

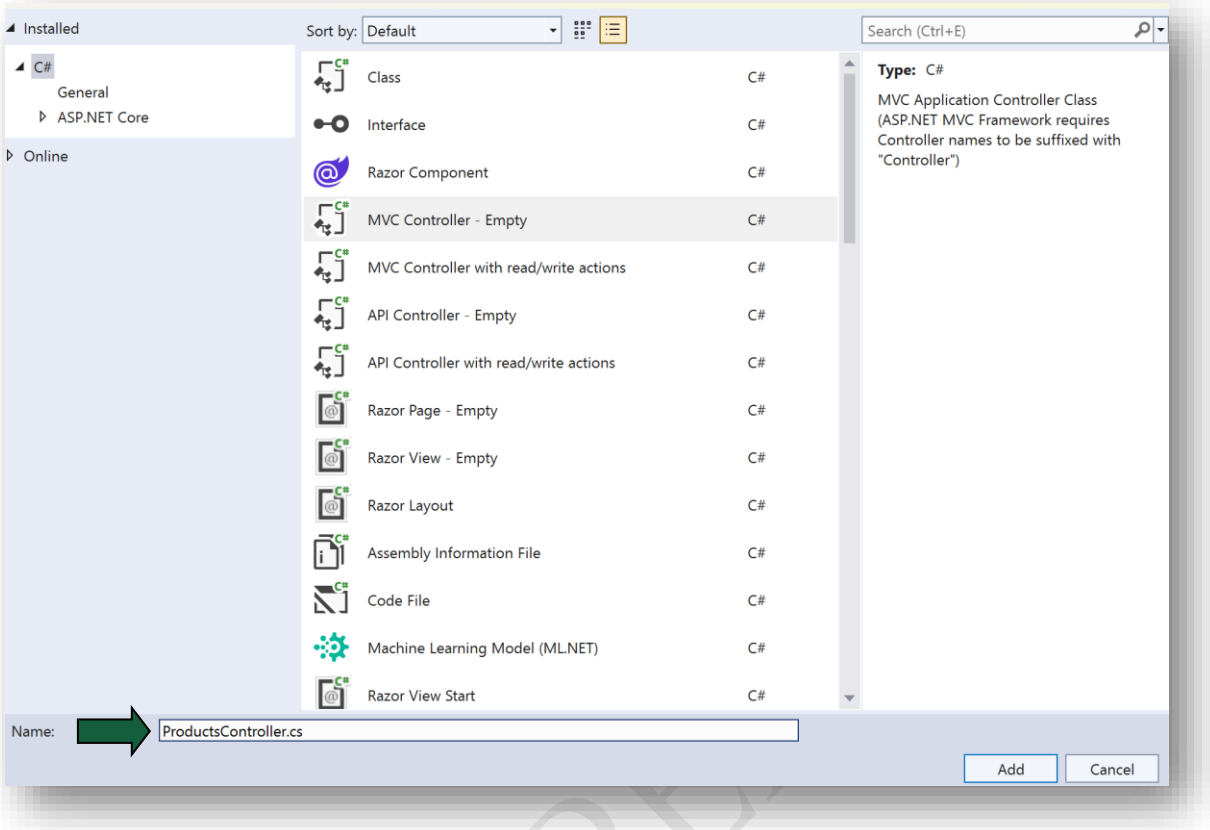
```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
}
```

PASUL 3

Se creează Controller-ul – **Products**Controller. Numele trebuie să conțină sufixul *Controller* (mai multe detalii vor fi studiate în cursul destinat Controller-ului).



După adăugarea Controller-ului, se adaugă o denumire pentru acesta.



PASUL 4

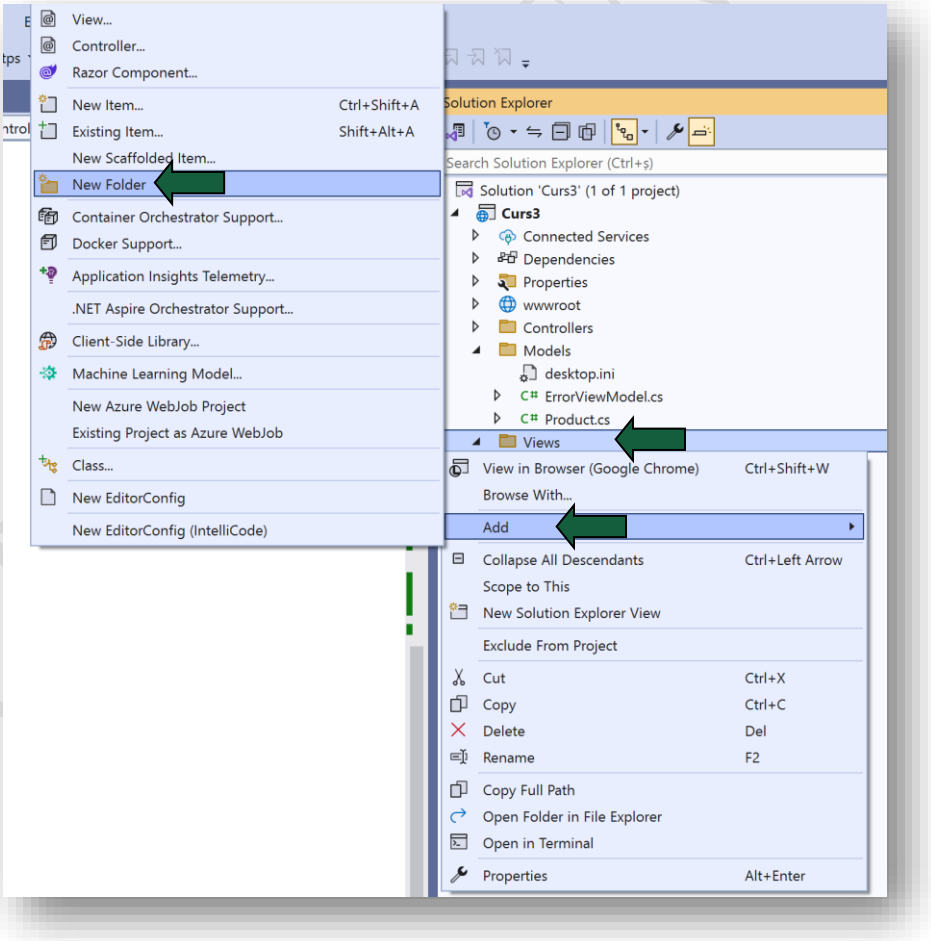
Se implementează lista de produse. Se creează 3 produse care se vor adăuga în listă.

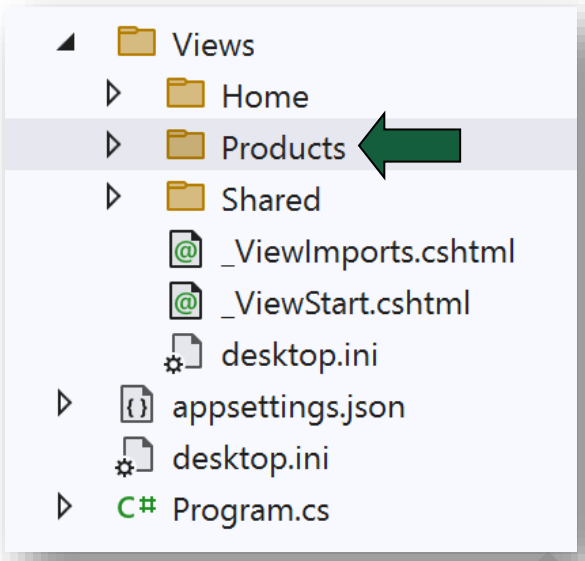
```
public class ProductsController : Controller
{

    // simulam baza de date utilizand o lista de produse

    private List<Product> products = new List<Product>
    {
        new Product { Id = 1, Name = "Produs1" , Description = "Descriere
produs 1", Price = 350},
        new Product { Id = 2, Name = "Produs2" , Description = "Descriere
produs 2", Price = 550},
        new Product { Id = 3, Name = "Produs3" , Description = "Descriere
produs 3", Price = 150}
    };
```

```
    // Metoda/Actiune cu ajutorul careia se afiseaza lista de produse

    public IActionResult Index()
    {
        // Se transmite lista de produse catre view-ul Index

        ViewBag.Products = products;
        return View();
    }
}
```

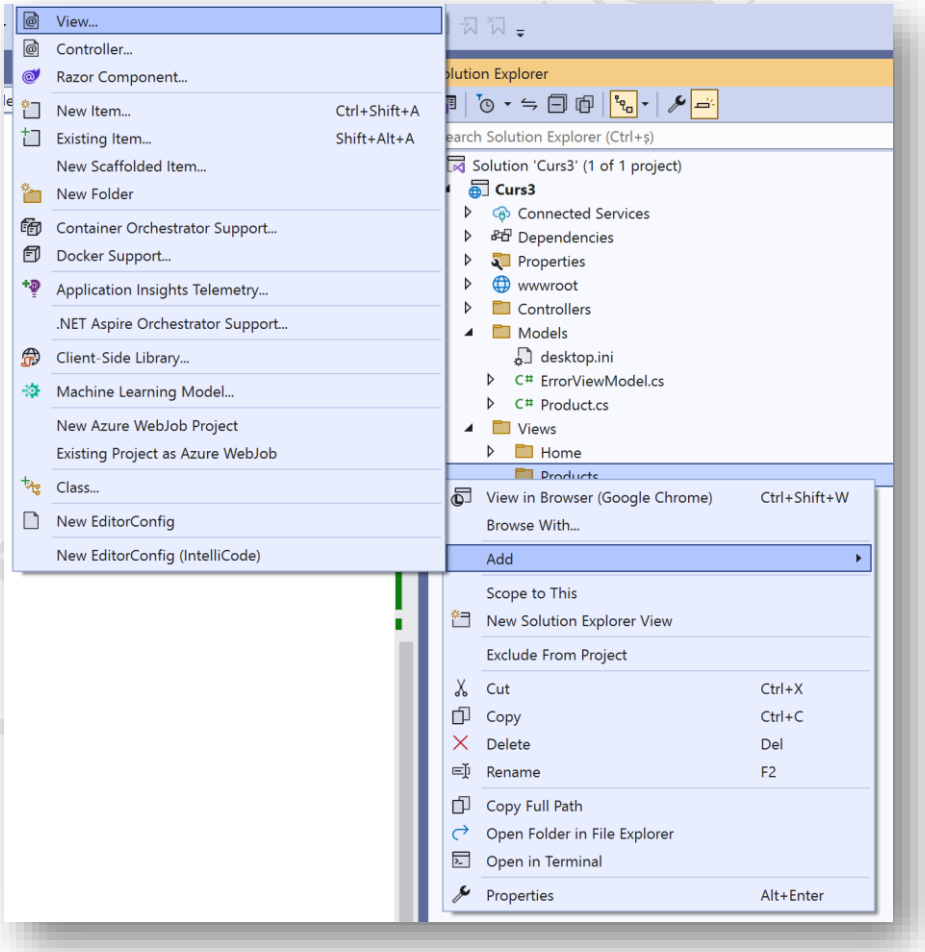
PASUL 5

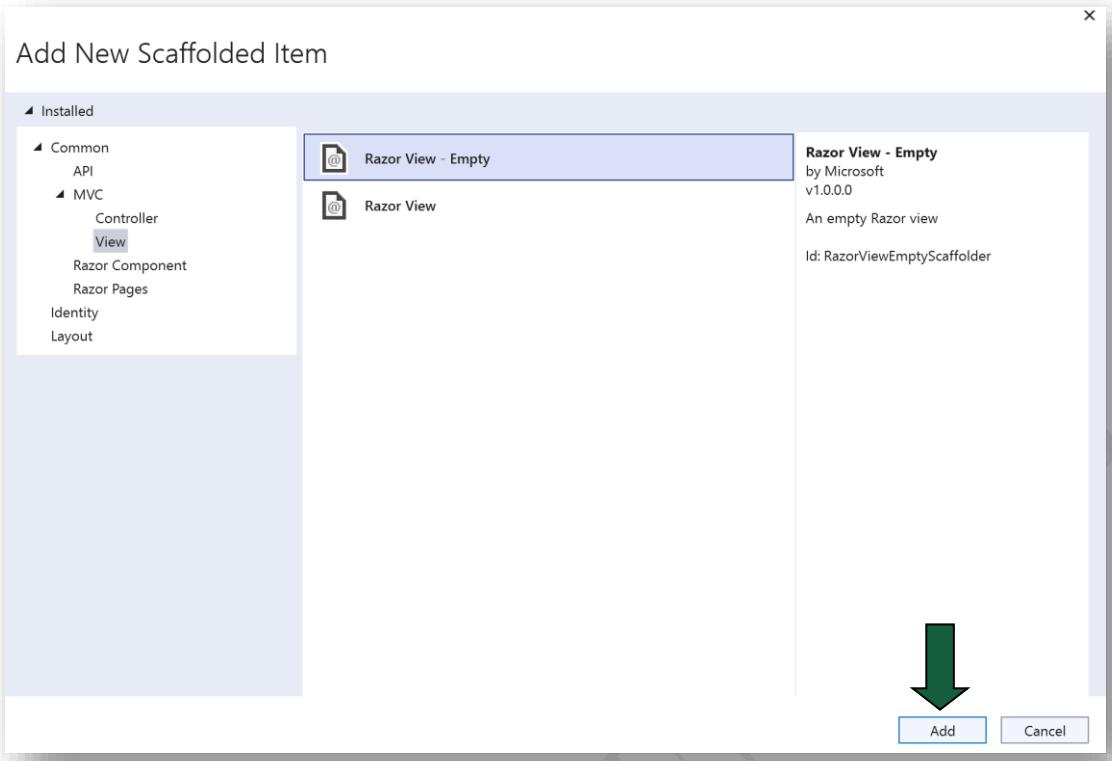
Se adaugă View-ul. Pentru adăugarea acestuia, este necesară crearea unui folder cu același nume ca și clasa. În cadrul acestui folder se va crea fișierul corespunzător, după cum urmează:

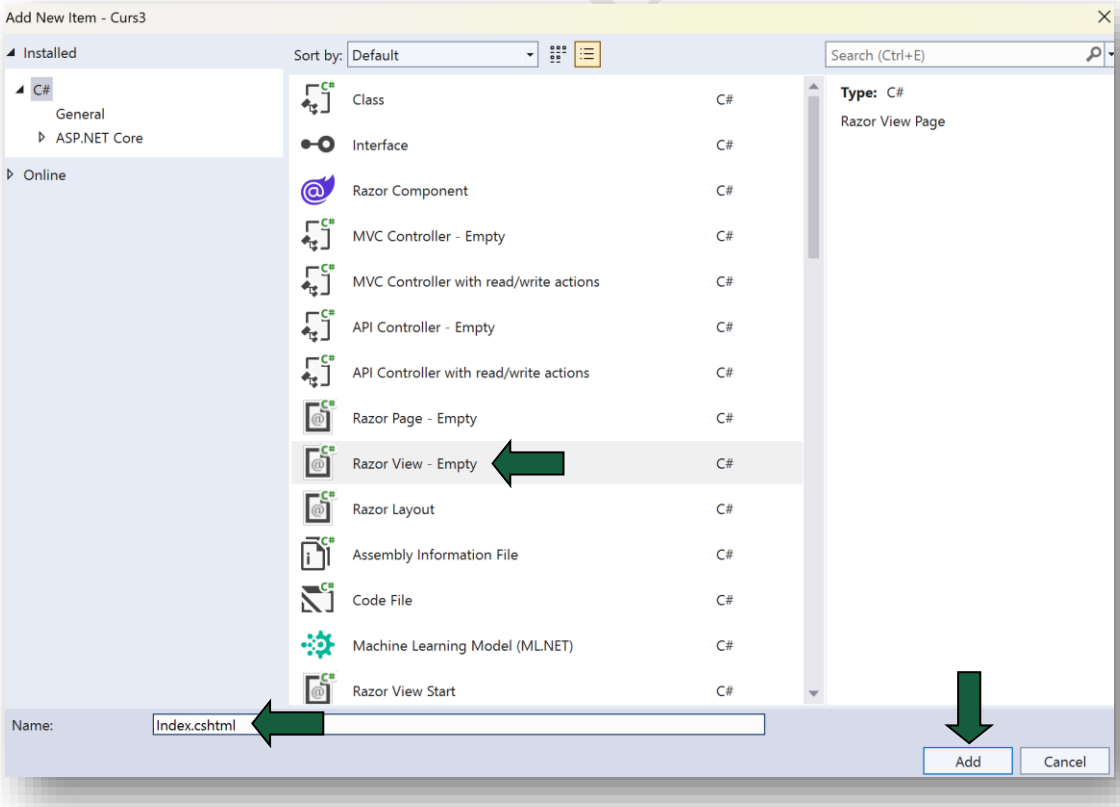




În folderul **Products** → se creează un View numit **Index.cs**




Add New Scaffolded Item
Installed
Common
API
MVC
Controller
View
Razor Component
Razor Pages
Identity
Layout
Razor View - Empty
Razor View
Razor View - Empty
by Microsoft
v1.0.0.0
An empty Razor view
Id: RazorViewEmptyScaffolder
Add
Cancel


Add New Item - Curs3
Installed
C#
General
ASP.NET Core
Online
Sort by: Default
Search (Ctrl+E)
Class C#
Interface C#
Razor Component C#
MVC Controller - Empty C#
MVC Controller with read/write actions C#
API Controller - Empty C#
API Controller with read/write actions C#
Razor Page - Empty C#
Razor View - Empty C#
Razor Layout C#
Assembly Information File C#
Code File C#
Machine Learning Model (ML.NET) C#
Razor View Start C#
Type: C#
Razor View Page
Name:
Index.cshtml
Add
Cancel

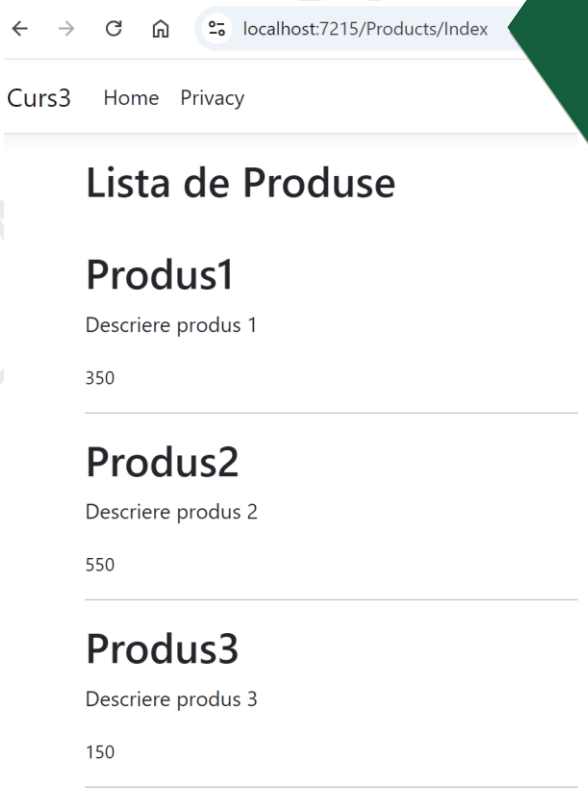
PASUL 6

Se afișează produsele în cadrul View-ului creat, utilizând cod HTML.

```
<h2>Lista de Produse</h2>
<br />

@foreach (var product in ViewBag.Products)
{
    <h2>@product.Name</h2>
    <p>@product.Description</p>
    <small>@product.Price</small>
    <hr />
}
```

PASUL 7

Se rulează aplicația în browser.



ATENȚIE la modul în care se afișează ruta! Ce se observă?

Sistemul de rutare

ASP.NET a introdus termenul de **Routing** (rutarea) pentru a elimina necesitatea mapării fiecărui URL cu un fișier fizic, asa cum era necesar în versiunea anterioară de ASP.NET Web Forms, unde fiecare URL trebuia să coincidă cu un fișier .aspx.

Rutele reprezintă diferite URL-uri din aplicație care sunt procesate de un anumit Controller și de o anumită metodă, pentru generarea unui răspuns către client. Framework-ul ASP.NET MVC invocă diverse clase de tip Controller, împreună cu diferite metode ale acestora, în funcție de URL-ul cerut de client.

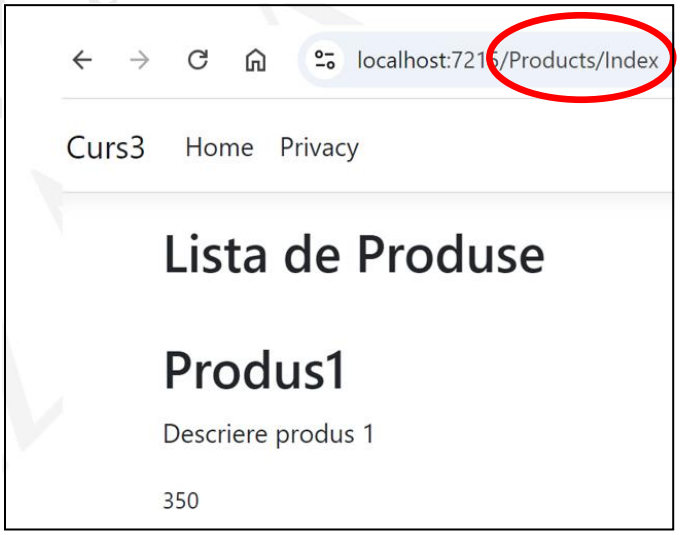
Astfel, pentru a accesa o anumită pagină, este necesar ca pentru aceasta să existe o rută definită, cât și un Controller care are o metodă (Action) care să răspundă acestei resurse.

Formatul de baza al rutelor în ASP.NET este următorul:

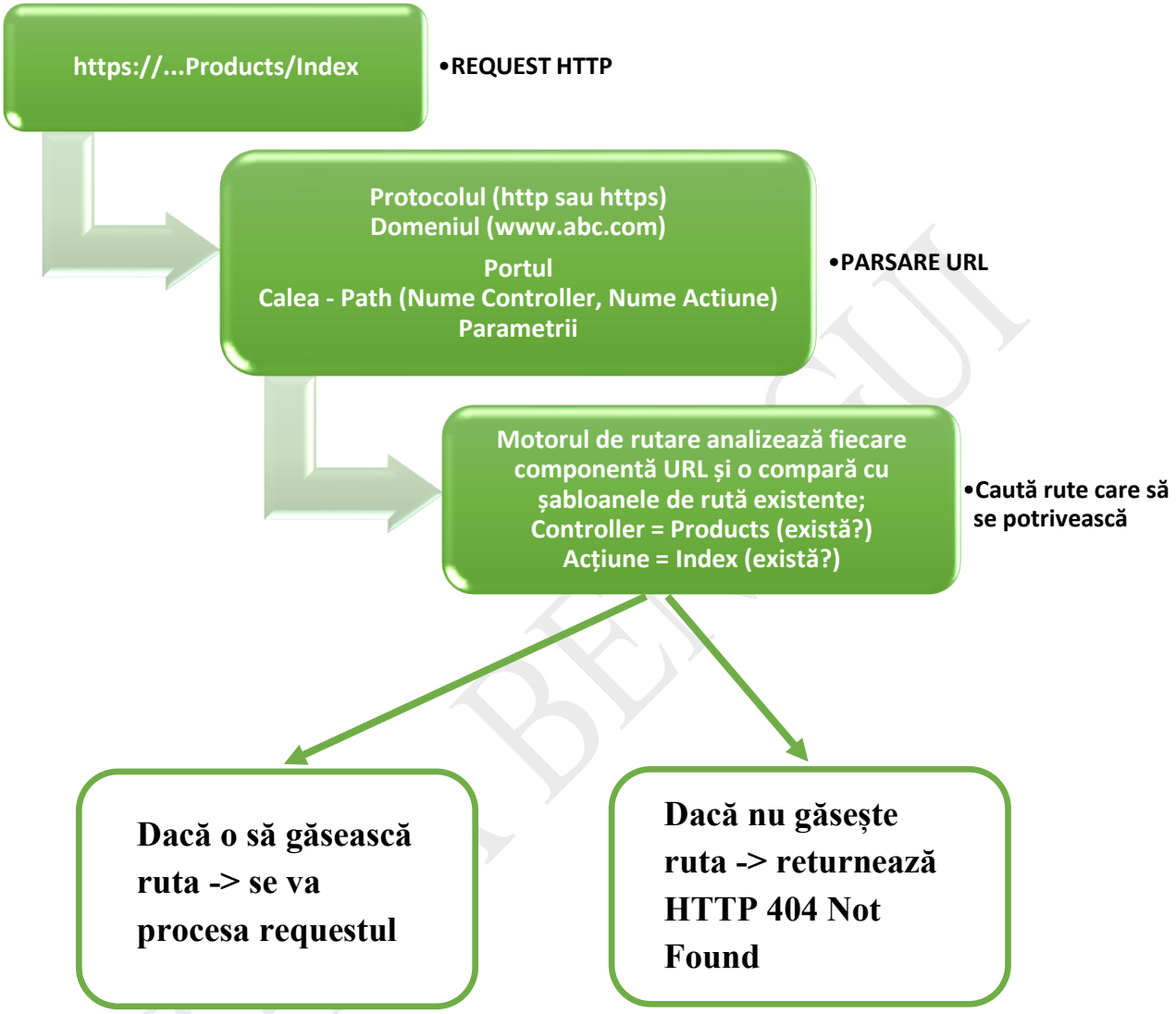
/{NumeController}/{NumeActiune}/{Parametri}

Rutele se definesc în clasa **Program.cs**

Plecând de la ruta definită anterior, se poate observa modul de funcționare al rutelor:

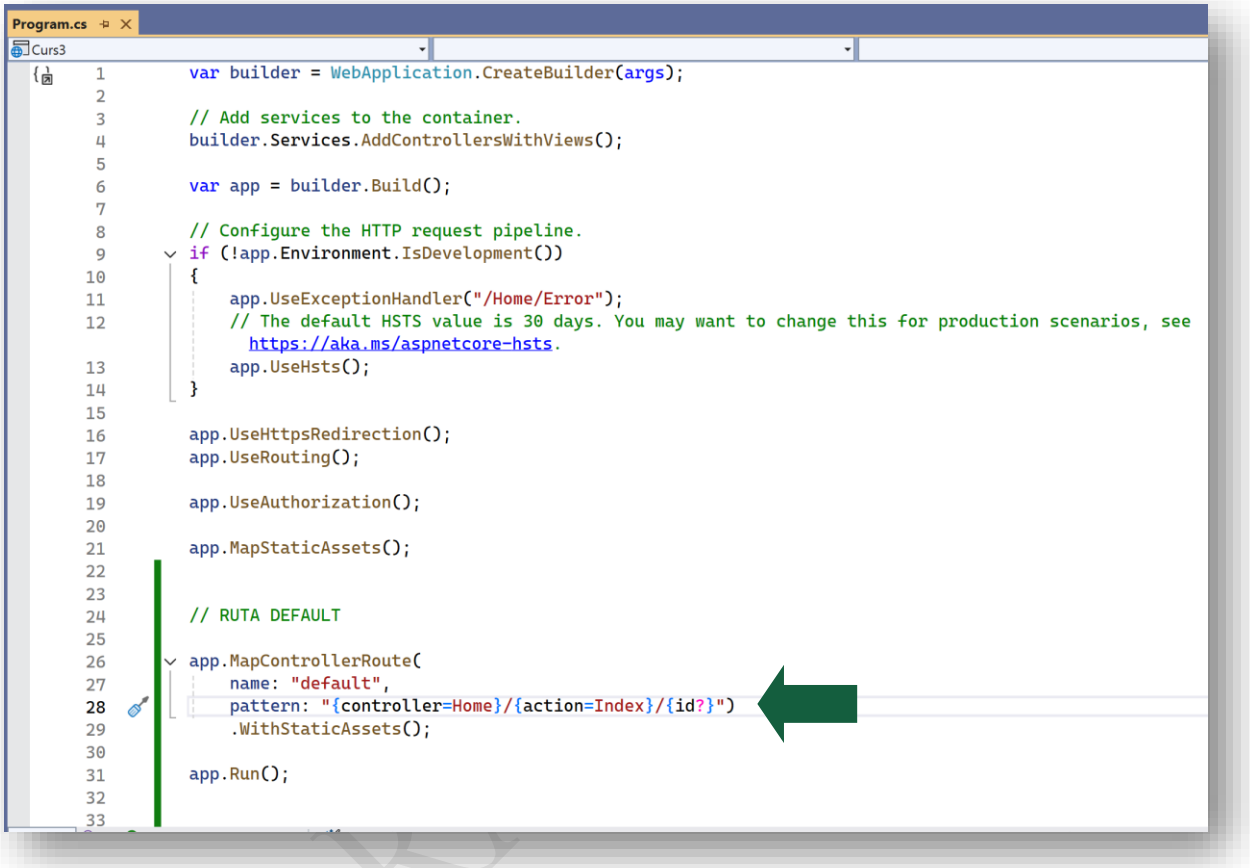


Sistemul de rutare - diagramă

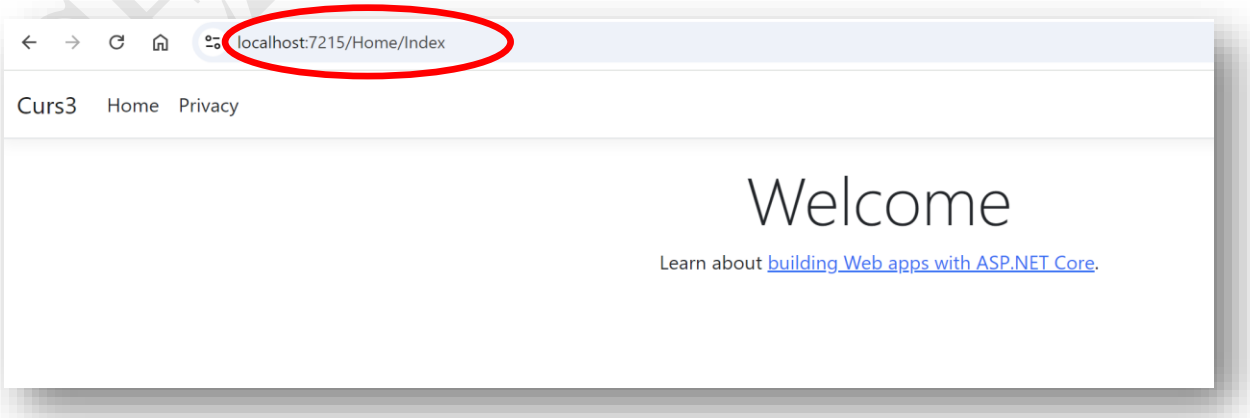


Fișierul Program.cs

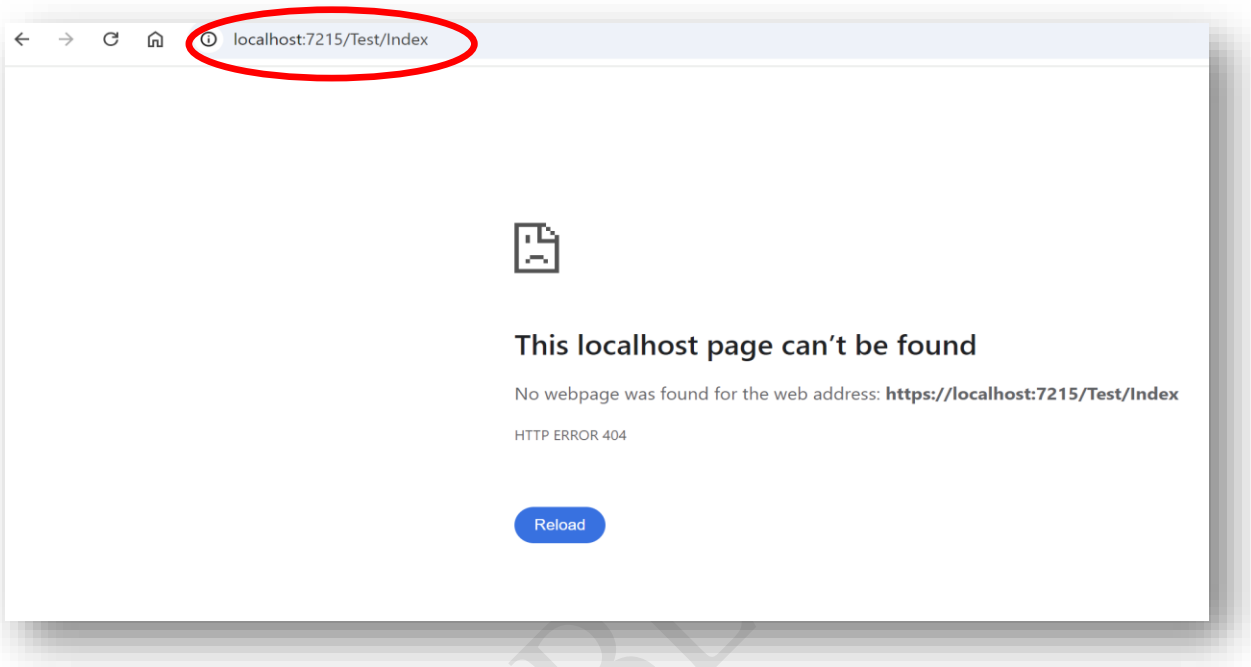
În **Program.cs**, imediat după crearea unei noi aplicații, se poate observa **ruta default**:



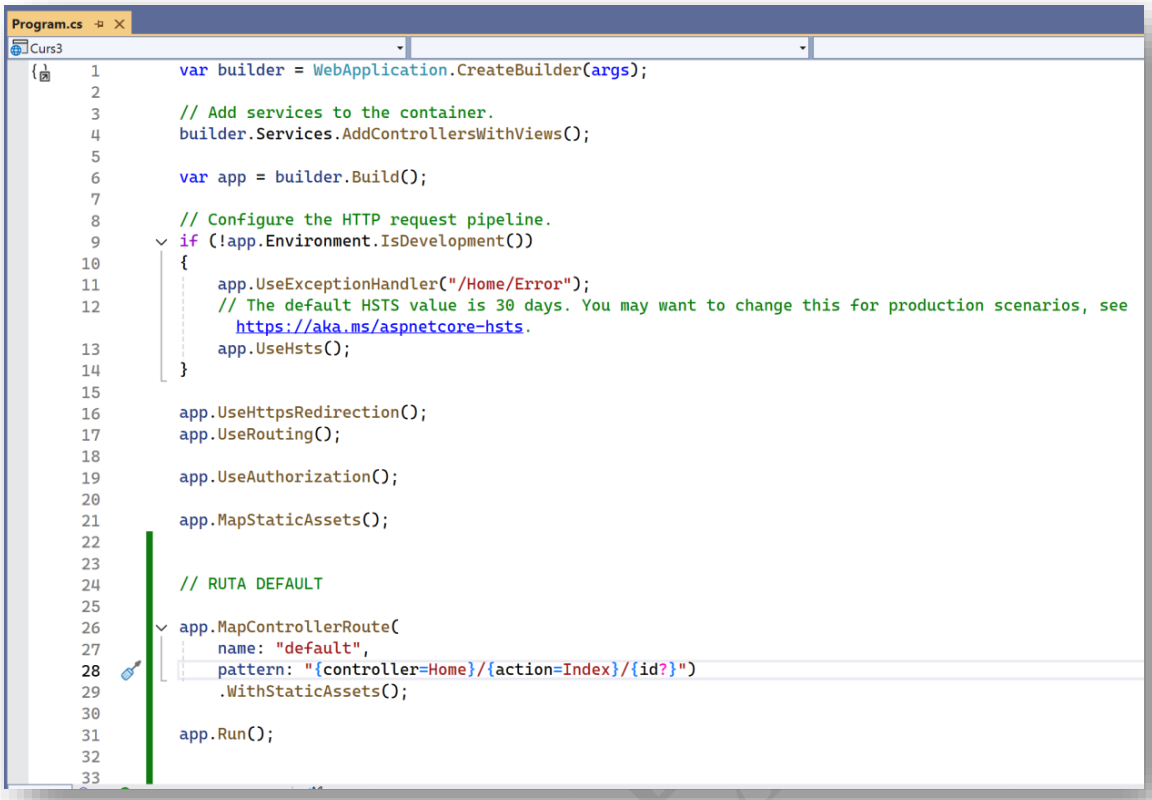
Atunci când se rulează aplicația, prima accesare a acesteia o să fie conform definiției rutei default. Se observă Controller-ul numit ***Home*** și metoda (acțiunea) numită ***Index***.



Dacă nu găsește nicio rută care să se potrivească, returnează un cod de stare HTTP 404 (Not Found) către client, indicând că resursa nu a fost găsită. În cazul de față, nu există Controller-ul numit ***Test***.



Tot în **Program.cs**, în momentul în care s-a creat un proiect utilizând template-ul Model-View-Controller, se pot observa și serviciile MVC necesare, componentele Middleware MVC, dar și componentele middleware utilizate pentru rutare.



OBSERVAȚIE:

Înainte de .NET 9 (în versiunea .NET 8)

În .NET 8 (și versiunile anterioare), fișierele statice (CSS, JS, imagini) se serveau separat de rutele MVC.

Se folosea **app.UseStaticFiles()**, care adăuga în pipeline un middleware special pentru a căuta fișiere în folderul *wwwroot*.

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

app.UseStaticFiles(); //  servește fișierele din wwwroot
app.UseRouting();
app.UseAuthorization();

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

app.Run();
```

Ce se întâmpla în versiunile anterioare:

- Cererile către /css/site.css sau /images/logo.png erau tratate de *UseStaticFiles()*;
- Cererile către /Home/Index erau tratate de MVC routing (prin *MapControllerRoute*);

Așadar, existau două sisteme paralele, complet separate. Unul pentru **fișiere statice**, altul pentru **rute MVC**.

În .NET 9 este integrat un sistem unificat de "Static Assets"

Microsoft a **unificat sistemul**. Fișierele statice pot fi servite **prin același mecanism de routing** (endpoint routing).

Acest lucru se realizează cu:

- `app.MapStaticAssets()` – care înlocuiește UseStaticFiles()
- `.WithStaticAssets()` – care leagă rutele MVC de sistemul nou

Beneficiile noului sistem

1. **Performanță mai mare** – .NET optimizează automat fișierele;
2. **Cache inteligent** – fișierele au hash în URL (site.hash.css), browserul nu le mai cere dacă nu s-au schimbat;
3. **Un singur mecanism** – totul prin endpoint routing, mai simplu de gestionat;
4. **Compatibil cu Tag Helpers** – <link asp-append-version="true" /> și alte taguri funcționează direct cu acest sistem;

Metoda **UseRouting()** din ASP.NET Core joacă un rol esențial în configurarea și activarea mecanismului de rutare în pipeline-ul middleware (**VEZI** definiția) al aplicației. Aceasta indică framework-ului că urmează să fie procesate rutele definite în aplicație.

Rolul metodei este de a **activa** mecanisnul de rutare. Aceasta transmite pipeline-ului middleware să utilizeze informațiile de rutare, adică să înceapă procesul de ***potrivire a cererilor HTTP cu rutele definite***.

ATENȚIE!

Fără această metodă, mecanismul de rutare nu va fi activat, deci nu va exista nicio potrivire între URL-urile cererilor și rutele înregistrate.

De asemenea, `UseRouting()` doar identifică ruta, adică găsește Controller-ul și Acțiunea corespunzătoare pentru cererea curentă, dar nu execută efectiv logica acelei acțiuni. Execuția efectivă a logicii cererii se realizează ulterior, prin `MapControllerRoute()` – în cadrul acestui pas are loc definirea rutei. Aceasta abordare, utilizând MapControllerRoute, a fost introdusă începând cu versiunea ASP.NET Core 6, folosind *Minimal Hosting Model*. Până la această versiune se utiliza metoda **UseEndpoints()**.

Pentru ca rutarea să funcționeze corect, **UseRouting()** trebuie să fie plasată **înainte** de alte middleware-uri care au nevoie de informațiile despre rută (cum ar fi autorizarea – **UseAuthorization()**) și **înainte** de **MapControllerRoute()**, care finalizează cererea.

Pipeline middleware

În ASP.NET Core, **pipeline-ul middleware** reprezintă lanțul de componente software prin care trece o cerere HTTP înainte de a ajunge la un răspuns. Fiecare componentă are capacitatea de a procesa cererea, de a efectua anumite acțiuni (cum ar fi autentificarea, autorizarea, logarea), și apoi de a decide dacă va trimite cererea mai departe către următoarea componentă din pipeline sau va genera un răspuns direct.

Un **middleware** este o componentă care:

- Primește o cerere HTTP
- Poate efectua anumite operații asupra cererii
- Decide dacă va trimite cererea mai departe către următoarea componentă middleware sau dacă va returna un răspuns imediat

Middleware-urile sunt apelate în ordinea în care sunt înregistrate în cod, creând un flux secvențial, cunoscut sub numele de **pipeline**.

Cererea trece prin fiecare middleware până când unul dintre ele returnează un răspuns, care apoi parcurge din nou pipeline-ul înapoi (în ordine inversă) pentru a fi trimis la client.

Exemple de middleware:

- **Rutarea** – găsește ruta potrivită pentru cererea curentă
- **Autentificarea** – verifică dacă utilizatorul este autentificat
- **Autorizarea** – verifică dacă utilizatorul are permisiuni pentru resursa cerută

Definirea unei rute

Metoda `MapControllerRoute()` este utilizată pentru a mapa rutele pentru Controllere și Acțiunile acestora.

Definirea unei rute are următorul format:

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

Se observă că variabila **app**, care este de tipul clasei **WebApplication**, oferă mai multe metode necesare definirii rutelor.

Clasa **WebApplication** reprezintă punctul central pentru configurarea și rularea unei aplicații web. Utilizarea acestei clase înlocuiește etapele mai complexe de configurare din versiunile anterioare de ASP.NET Core, oferind o modalitate mai rapidă de a **configura middleware-uri**, **rutarea** și **serviciile necesare**, toate aceste configurări realizându-se acum prin intermediul acestei clase.

Metoda **MapControllerRoute** adaugă ruta default și primește ca argumente 2 parametri:

- **name**: care reprezintă numele rutei (ex: `name: "default"`)
- **pattern**: care reprezintă modelul URL-ului sau segmentele acestuia (ex: `pattern:{controller=Home}/{action=Index}/{id?}`).

  În plus sunt definite detaliile după cum urmează:

  - **controller** – primește ca valoare numele Controller-ului care trebuie să raspundă la această rută;
  - **action** - primește ca valoare numele metodei din Controller care să răspundă la această rută;
  - pentru fiecare parametru adăugat în rută, definește **tipul parametrilor**, dacă aceștia sunt **necesari** sau **opționali**, sau se pot seta valorile implicite;

În exemplul anterior se definește ruta **/Home/Index/{id}** care este procesată de Controller-ul **HomeController** prin metoda **Index**. Parametrul **id** este opțional și poate fi omis din URL. Fiind urmat de "`?`" -> `{id?}`, înseamnă că este un parametru opțional.

În acest sistem de rutare, pot fi mai mulți parametri delimitati prin caracterul "/".

Așadar, în acest caz, metoda **app.MapControllerRoute()** mapează endpoint-uri cu pattern-ul:

```
{controller=Home}/{action=Index}/{id?}
```

Endpoint-ul din ASP.NET Core este reprezentat de Controller, fiind acea unitate responsabilă cu procesarea request-urilor.

Privind exemplul anterior, putem spune că de fiecare dată când aplicația primește un URL care se potrivește cu pattern-ul, atunci o sa se acceseze Controller-ul numit **Home** și Action-ul (metoda) **Index**. Acest lucru se întâmplă doar în cazul în care nu este specificat un alt Controller sau o altă metodă. În cazul în care sunt specificate explicit alte endpoint-uri, atunci acelea o să fie accesate.

În cazul în care aplicația este accesată fără segmentele necesare în cadrul URL-ului, adică se cere pagina principală a aplicației "/", framework-ul ASP.NET MVC va răspunde în mod implicit cu metoda **Index** din Controller-ul **Home**.

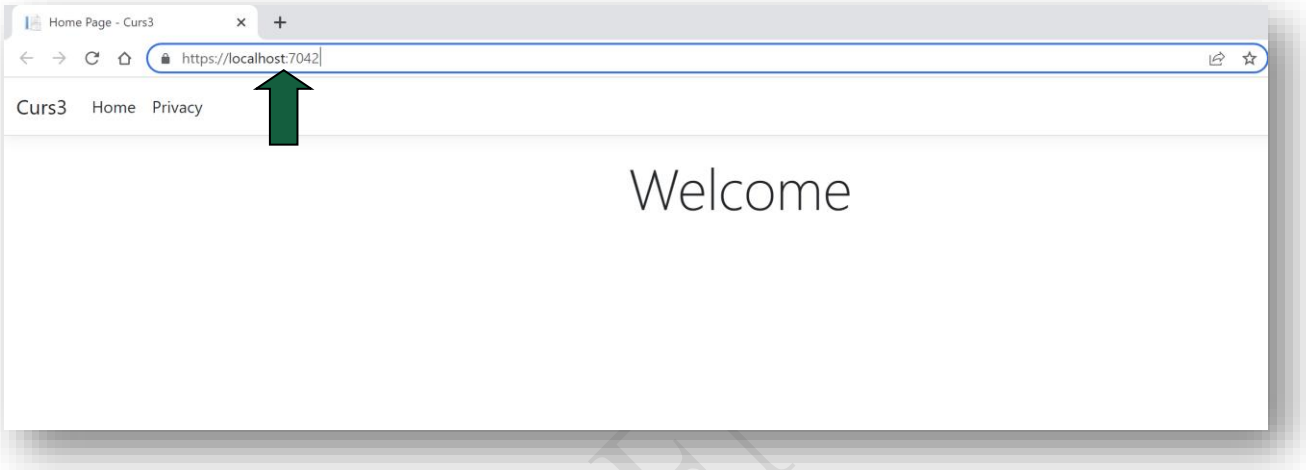
În concluzie, pattern-ul `{controller=Home}/{action=Index}/{id?}` se potrivește cu toate URL-urile de forma:

- /
- **/Home**
- **/Home/Index**
- **/Students/Index**
- **/Students/Index/5**
- **/Students/Afisare**, etc

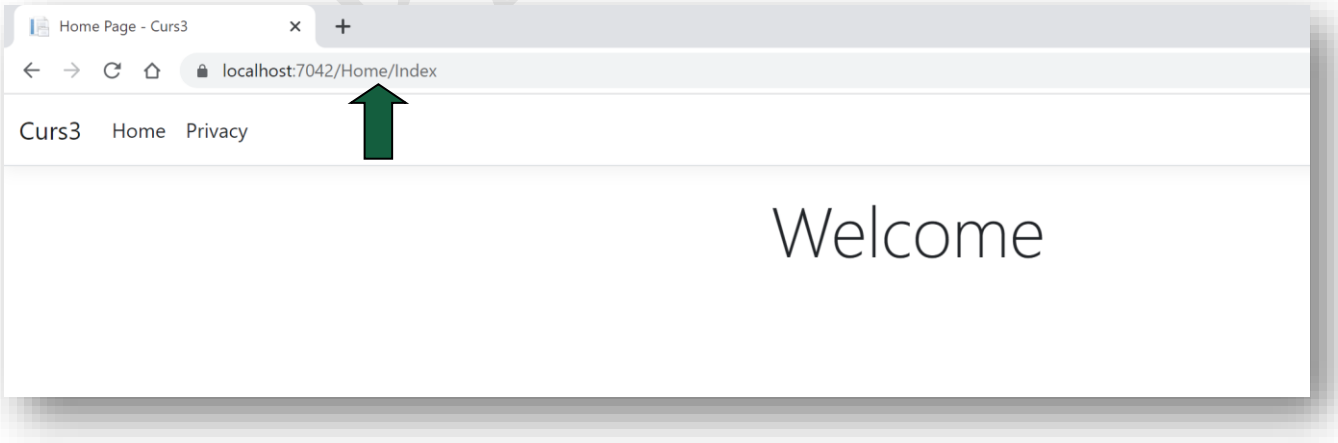
În exemplul următor se poate observa existența Controller-ului **HomeController**. În cadrul clasei HomeController există mai multe metode (Actions). Se poate observa prezența metodei **Index()**. În momentul în care există o metodă, trebuie să existe și un View asociat. View-ul trebuie să se numească identic cu metoda, deci **Index.cshtml**.

Privind configurația default a rutei, putem observa că aceasta este ruta default, adică ruta care se accesează prima și afișează către utilizatorul final interfața care se află în Index.cshtml.

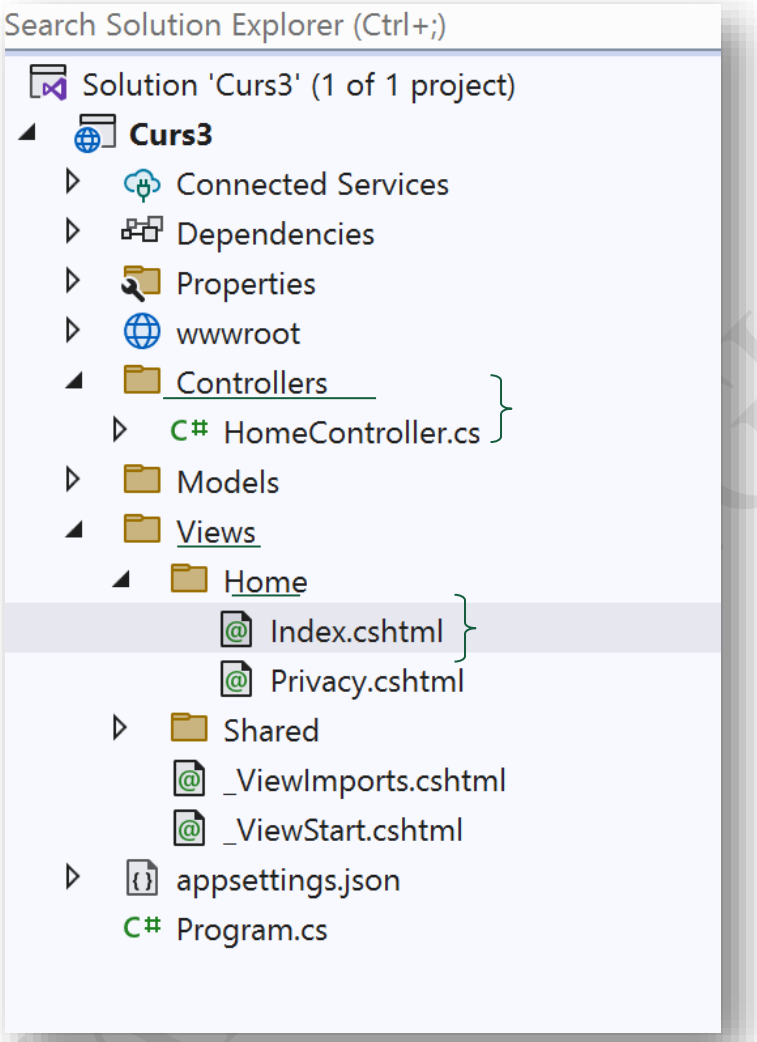
După rulare se acceseaza în browser ruta default:



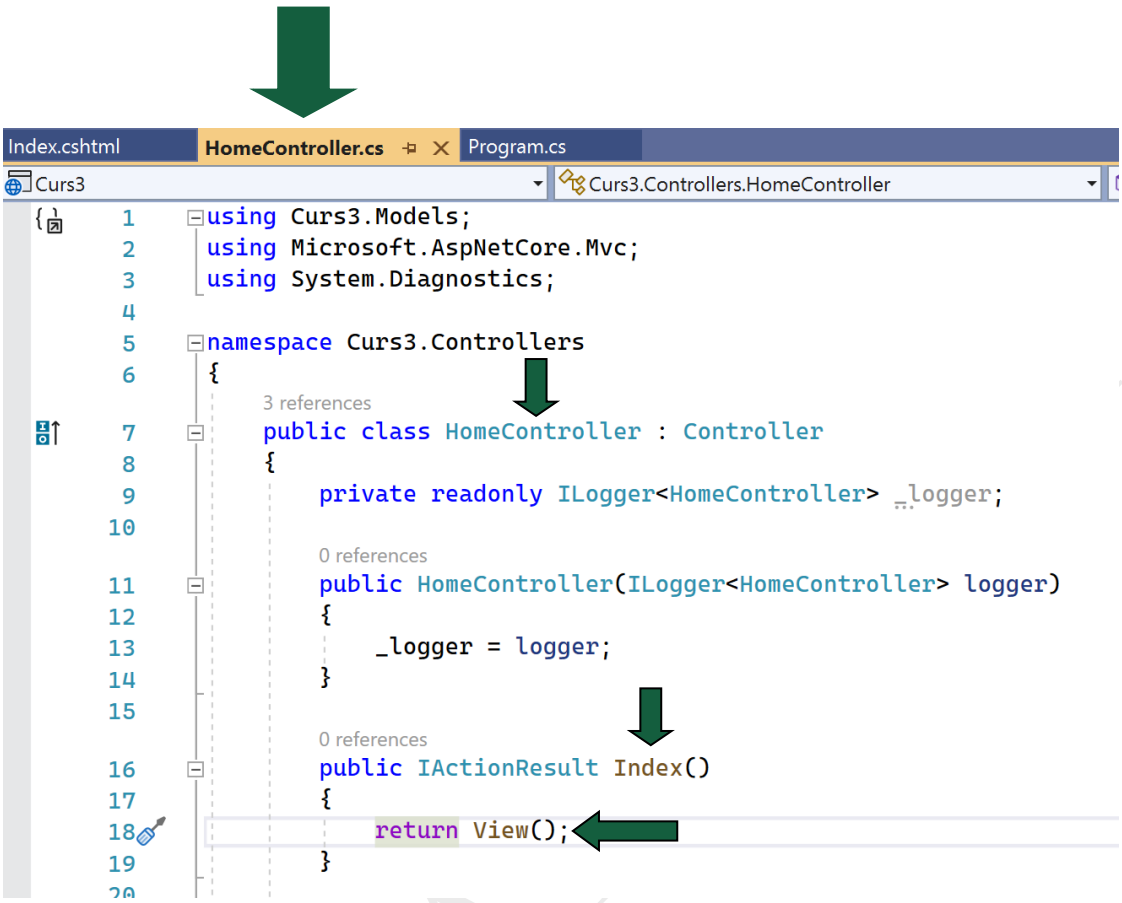
În cazul în care se accesează pagina prin scrierea întregului URL, atunci ruta o să aibă următorul format:



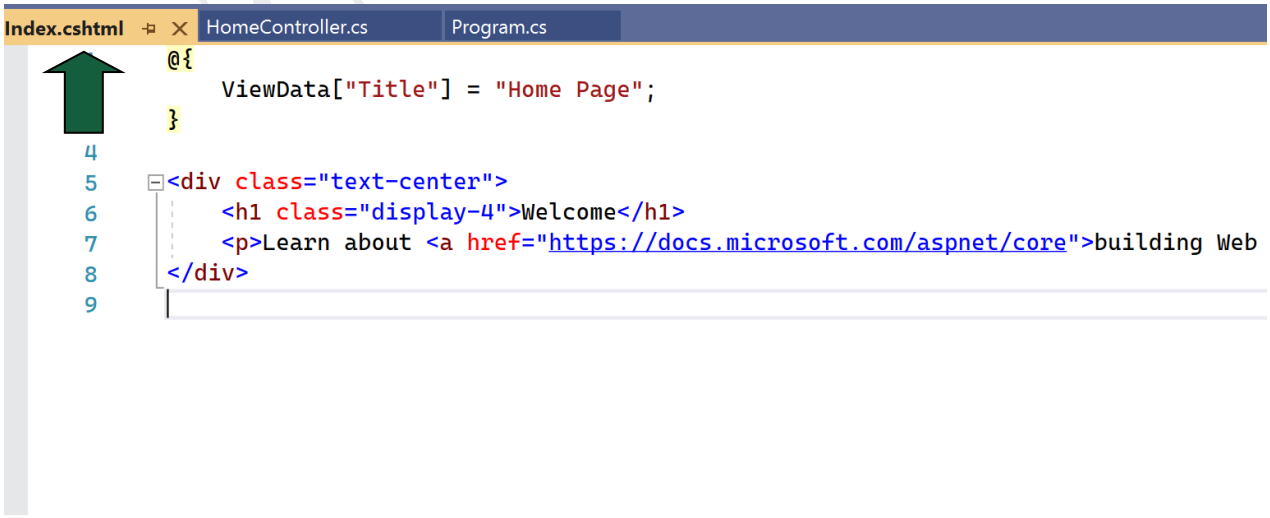
În **Solution Explorer** se observă:



În **HomeController** există metoda **Index**:



În **View** → **Folderul Home** (asociat Controller-ului HomeController) → **Index.cshtml** (pagina pentru codul html asociata metodei Index din Controller)

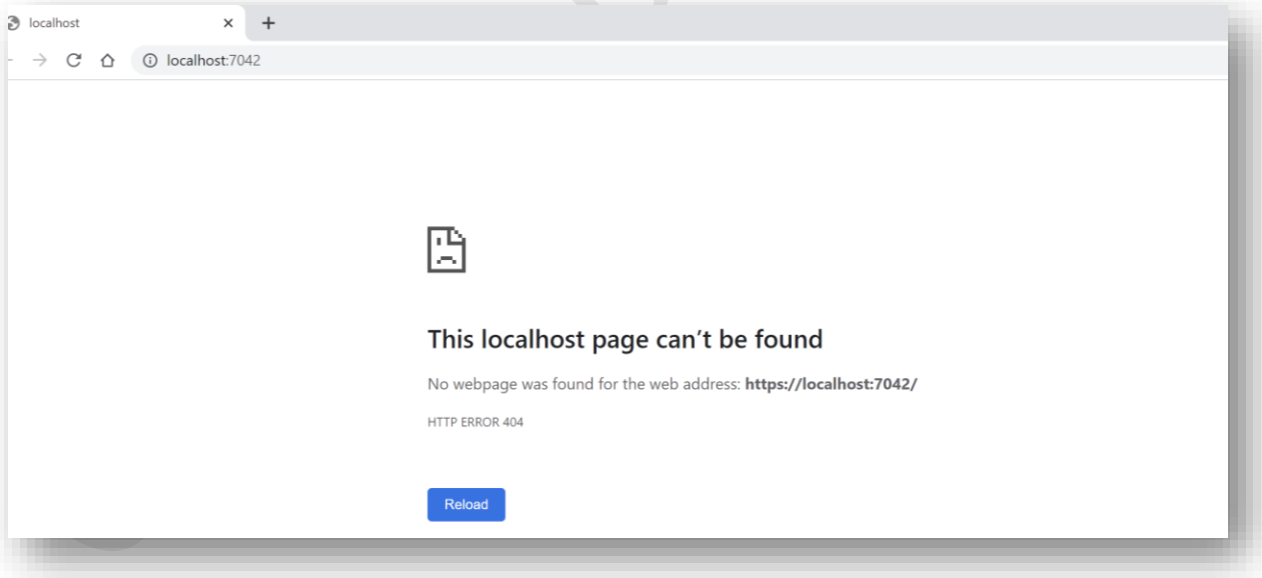


În cazul în care definiția rutei default nu conține și valorile implicite, atunci nu se pot accesa paginile aplicației.

De exemplu: se elimină ruta deja definită și se adaugă următoarea configurație:

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

*/

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

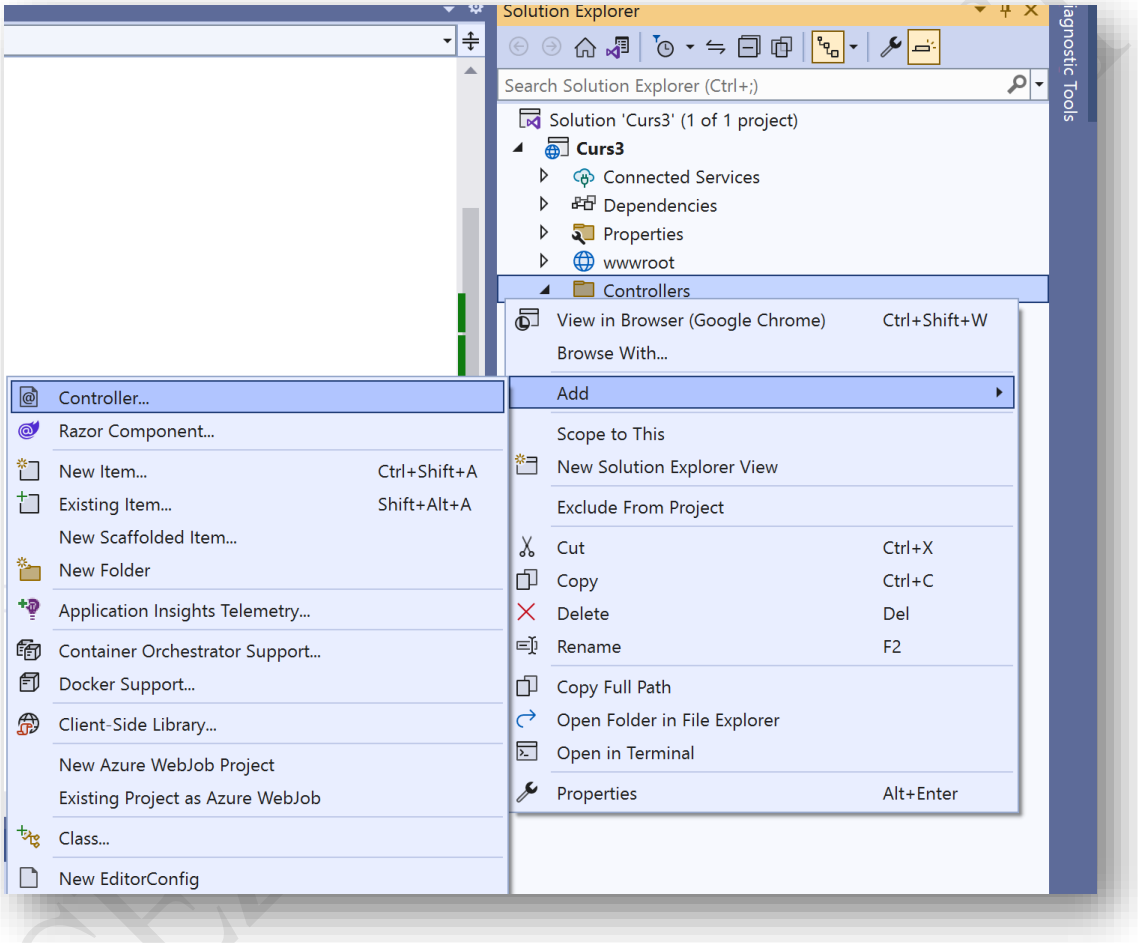
În acest caz, în momentul rulării, apare mesajul **HTTP ERROR 404** deoarece nu găsește pagina. Acest lucru se întâmplă din cauza faptului că sistemul de rutare nu poate asocia ruta cu niciun Controller.



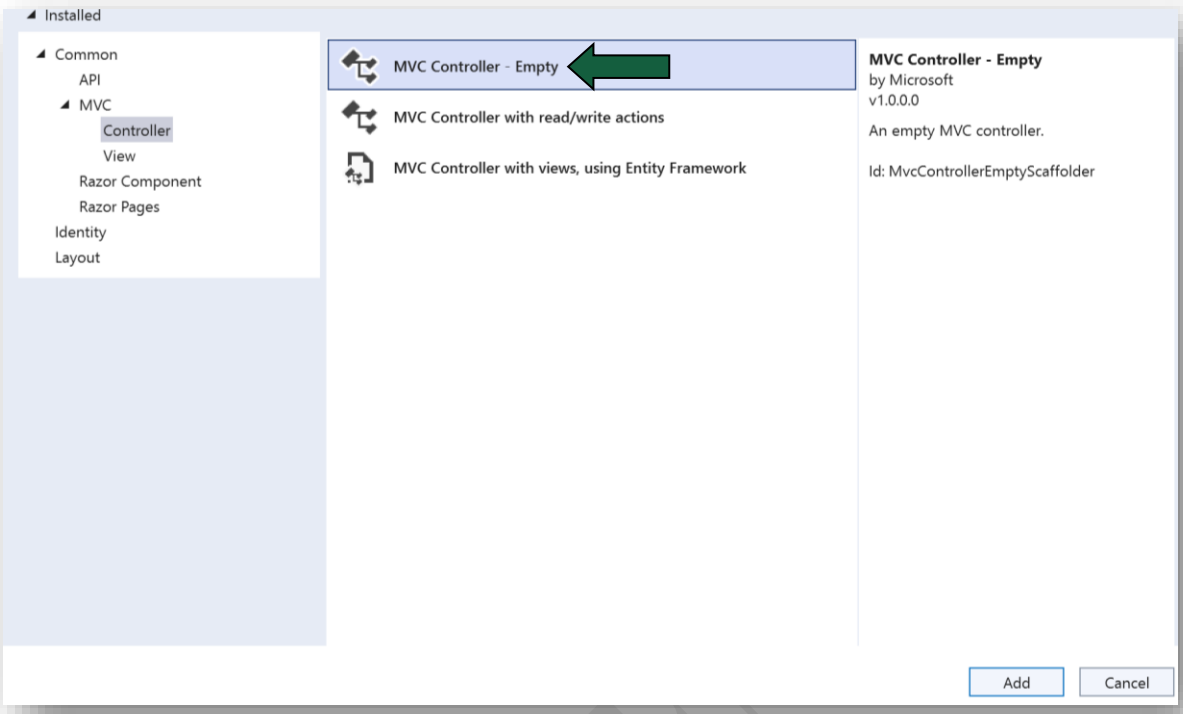
Exemple de implementare a rutelor

Configurarea rutelor

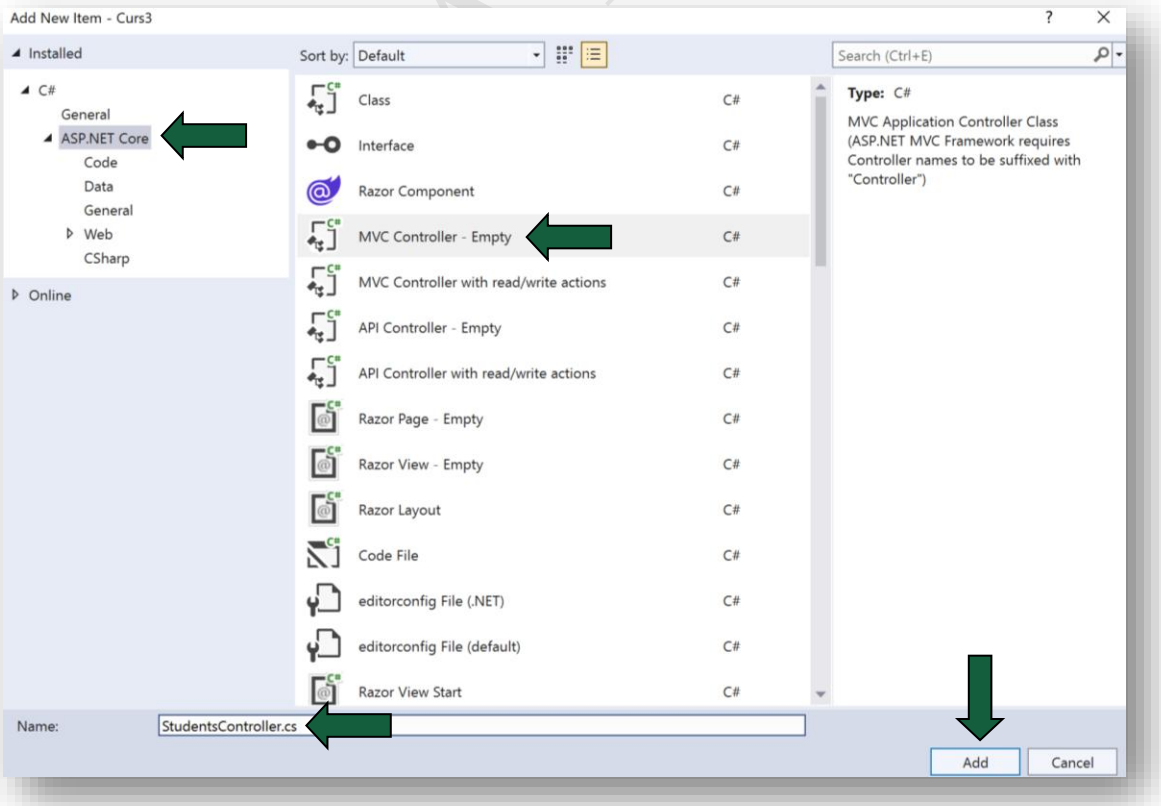
Pentru exemplele următoare se va crea un nou Controller, numit **StudentsController.** Click dreapta pe folderul Controller → Add → Controller.



Se selectează MVC Controller – Empty:



Se modifică numele noului Controller:



Exemplul 1:

Pentru afișarea unui text, o să se utilizeze metoda Index din cadrul Controller-ului **StudentsController**. În acest moment nu se va utiliza niciun View asociat.

```
public class StudentsController : Controller
    {
        public string Index()
        {
            string response = "Hello World";

            return response;

            //return View();
        }
    }
```

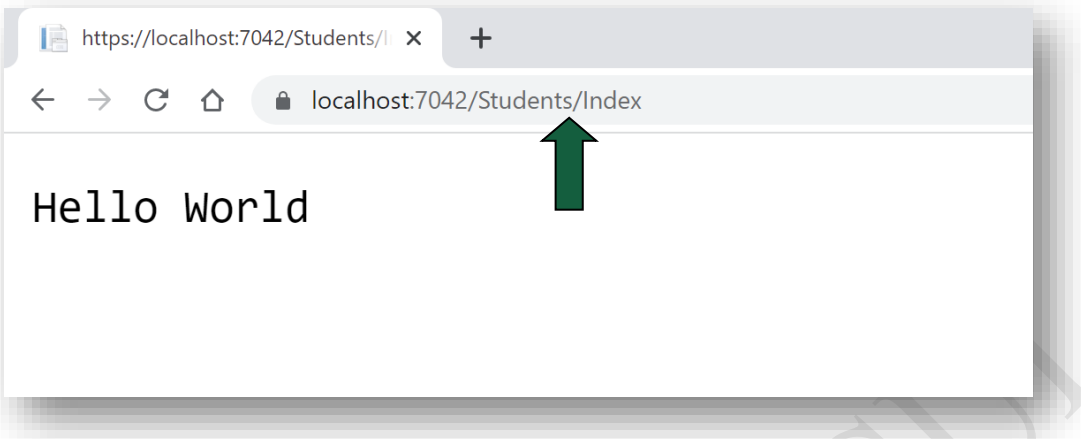
Tipul returnat de metoda ***Index*** a fost schimbat în **string** pentru a putea afișa în browser un simplu text. Astfel, valoarea de return devine valoarea variabilei **response**.

După rulare, mesajul o să fie afișat în browser, accesând ruta **/Students/Index** (**/NumeController/NumeActiune**). În momentul accesării URL-ului, request-ul se trimite aplicației, după care se încearcă maparea URL-ului cu o configurație de rută prezentă în Program.cs.

Astfel, pattern-ul o să primească noile valori:

- controller = Students
- action = Index
- id = este opțional și poate să lipsească

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



Exemplul 2:

În continuare, se definește o rută după cum urmează:

- ruta o să conțină doi parametri – **name** și **id**
- parametrul name o să aibă o valoare implicită “Everyone”;
- paramentrul id o să fie opțional

Metoda ***Index*** din Controller o să afișeze mesajul “Hello Everyone”, folosind valoarea parametrului **name**, provenita din cadrul rutei.

Definirea rutei:

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

SAU:

```
app.MapControllerRoute(
    name: "HelloEveryone",
    pattern:
"{controller=Students}/{action=Index}/{name=Everyone}/{id?}")
    .WithStaticAssets();
```

OBSERVAȚIE:

Metoda `.WithStaticAssets()` se folosește doar pentru rutele MVC sau Razor Pages care servesc pagini web ce folosesc fișiere statice (CSS, JS, imagini).

Se pune de obicei la ruta default, pentru că aceasta generează interfața principală a aplicației.

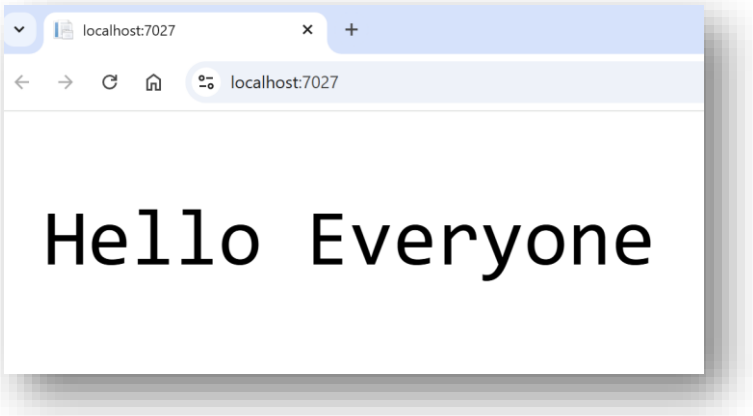
Nu este necesară la rutele de tip API sau la rute care nu afișează pagini HTML (ex: controllere de date, servicii, endpoints JSON).

Scopul ei este să lege ruta MVC de noul sistem de static assets introdus în .NET 9, care oferă caching și optimizare automată pentru fișierele din *wwwroot*.

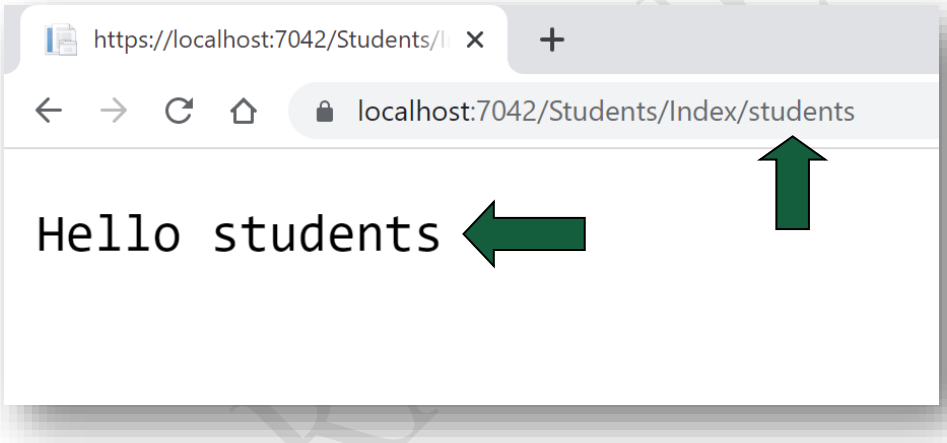
Implementarea metodei în Controller-ul **StudentsController**, metoda **Index**.

```
public string Index(string name, int? id)
{
        string response = "Hello " + name + " ";

        if (id != null)

        {
            response = response + "id = " + id;
        }

        return response;

        //return View();
}
```

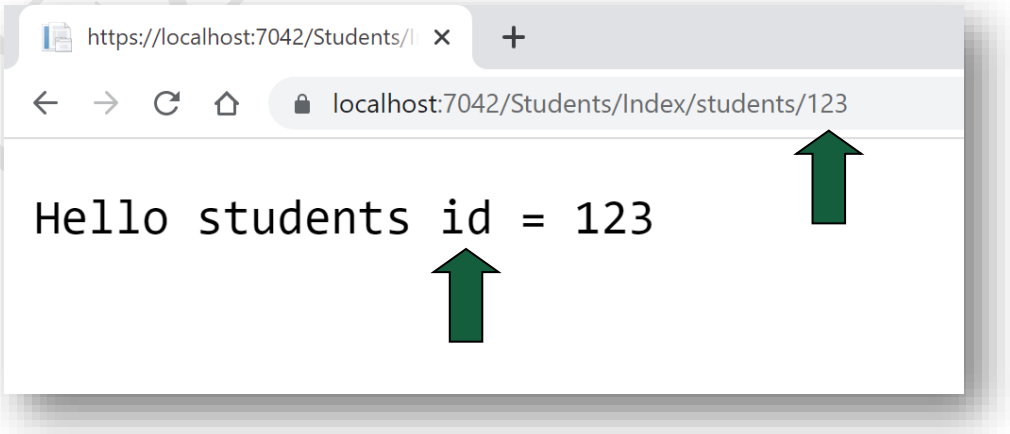
În momentul în care se rulează aplicația, fără a introduce segmentele URL-ului, se observă că pentru variabila **name** s-a transmis valoarea acesteia implicită: **Everyone**, afișându-se mesajul Hello Everyone!



Dacă variabila **name** primește o valoare în URL, se observă cum aceasta se transmite către Controller și este afișată în pagină.



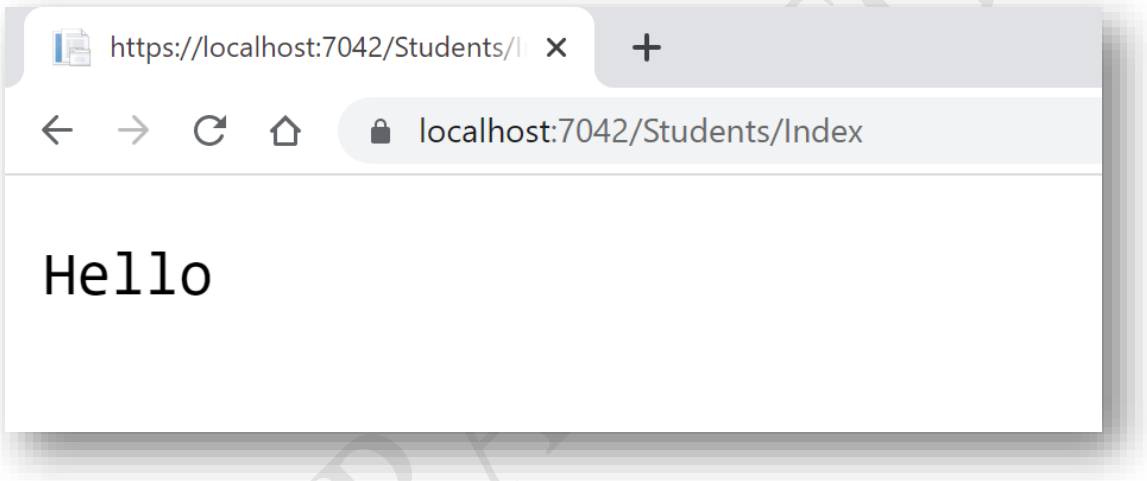
Când parametrul opțional **id** are valoare, secvența de cod specifică acestuia este executată și valoarea sa apare în răspunsul primit de la Controller:



OBSERVAȚIE:

/!\ Ruta trebuie definită inaintea rutei default, rută deja existentă în fișierul **Program.cs**, deoarece rutele sunt interpretate în mod cascadă (de sus în jos). Framework-ul utilizează prima configurație din fișier care conține același număr de parametri ca ruta accesată din browser.

De exemplu, dacă ruta default este definită inaintea rutei create în exemplul anterior, iar URL-ul de accesare este **/Students/Index**, atunci configurația rutei default o să se potrivească și vom avea un rezultat ca cel de mai jos. Nu se afișează și valoarea implicită a parametrului **name**.



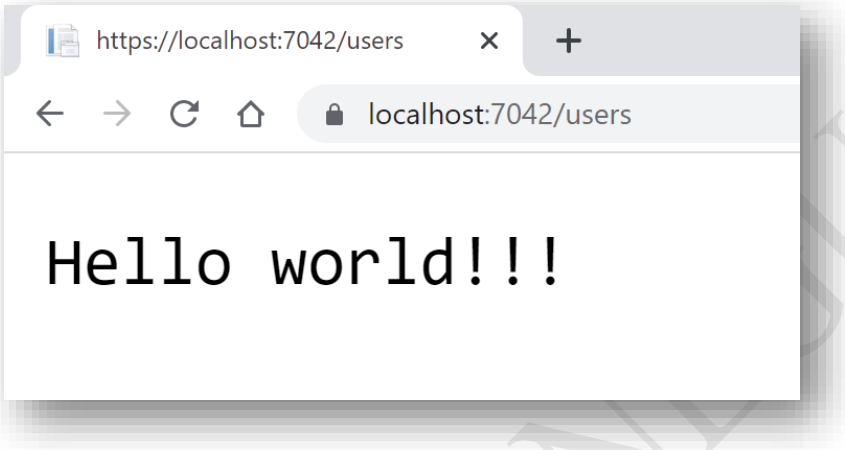
Definirea rutelor custom

Pentru fiecare **Controller** și **Acțiune** în parte se pot defini și rute custom. **De exemplu:** dacă se dorește accesarea Controller-ului *Students* și a metodei *Index* printr-un URL de forma: **/users** se poate implementa următoarea rută:

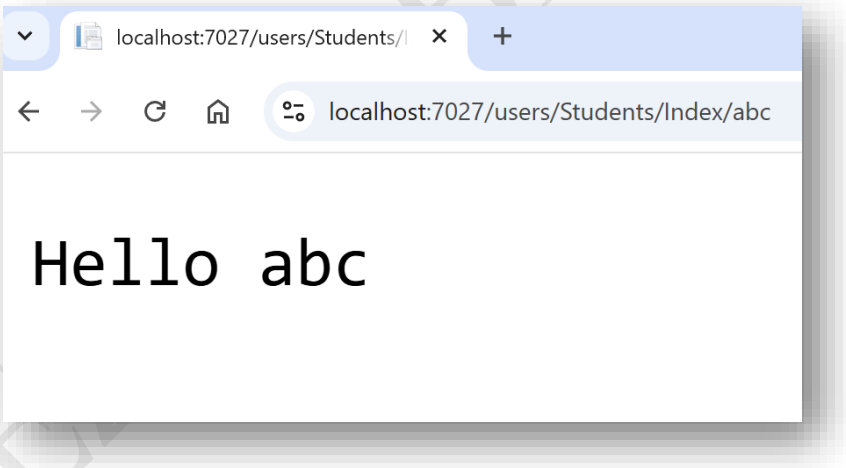
```
app.MapControllerRoute(
    name: "Users",
    pattern: "users/{controller=Students}/{action=Index}/{name=world!!!}");
```

În varianta anterioară se poate accesa **/Students/Index/name** doar prin intermediul URL-ului: **/users** deoarece restul parametrilor vor prelua valorile.

Astfel, ruta **/users** a accesat Controller-ul *StudentsController* și metoda *Index* cu paramentrul implicit *name=world!!!*



Dacă trebuie introdusă o altă valoare pentru parametrul **name**, atunci URL-ul trebuie să fie: **users/Students/Index/abc**



În același mod se poate proceda și pentru mai multe elemente în rută:

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

Definiția anterioară a rutei funcționează și pentru URL-uri de tipul:

/Home/Page/Students/Index
/Home/Page/Users/Read

Adică funcționează pentru orice alt nume de Controller și Acțiune.

Dacă se dorește limitarea unei rute la o singură acțiune, dintr-un singur Controller, atunci se folosește următoarea variantă:

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

Acest lucru se întâmplă deoarece pattern-ul nu are niciun parametru configurabil, iar ruta o să mapeze doar Controller-ul *Home* și metoda *Index*.

De asemenea, varianta anterioară se folosește și pentru cazul în care se dorește accesarea rutei doar printr-un URL custom. De exemplu: **/users**

```
app.MapControllerRoute(
    name: "Users2",
    pattern: "users/{name?}/{id?}",
    defaults: new { controller = "Students", action = "Index" });
```

OBSERVAȚIE:

/! În momentul scrierii rutelor, dezvoltatorul trebuie să se asigure că nu există ambiguitate între definițiile acestora.

Constrângerile parametrilor

Pentru a asigura un anumit tip de date sau un anumit format pentru parametrii transmiși către Controller, este necesară declararea unor constrângeri.

Există mai multe tipuri de constrângeri: de tip, length, max, min, range, regex.

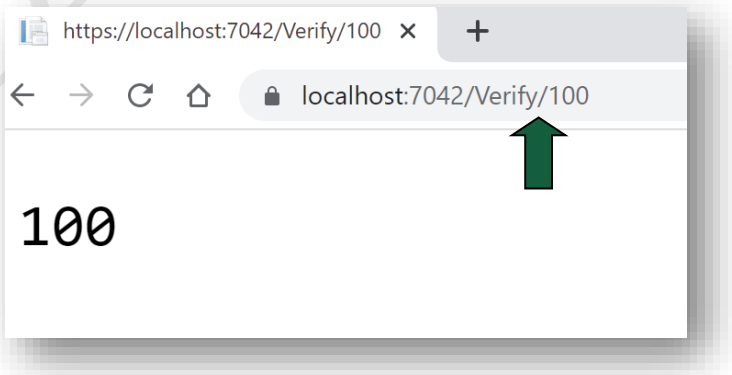
Un exemplu de constrângere este:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "Verify/{id:range(10,100)}",
    defaults: new { controller = "Home", action = "Verify" });
```

Pentru un URL de tipul **/Verify/50** se caută pattern-ul potrivit, după care se accesează parametrii specifici din **defaults** → Controller-ul **Home** și metoda **Verify**. Dacă parametrul **id** se află în intervalul închis (10, 100), atunci o să acceseze ruta, iar în caz contrar o să se afiseze 404 Not Found.

Pentru verificare se poate implementa în Controller-ul *Home*, metoda *Verify* care afișează id-ul în pagină.

```
public int Verify(int id)
   {
       return id;
   }
```



În cazul **expresiilor regulate**, se configurează ruta astfel:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "VerifyDigits/{id:regex(\\d+)}",
    defaults: new { controller = "Home", action = "VerifyDigits" });
```

Ruta se accesează folosind URL-ul: **/VerifyDigits/id**. Id-ul are o constrângere folosind o expresie regulată – se verifică dacă este număr, accesând apoi metoda **VerifyDigits** din Controller-ul **Home**.

```
public string VerifyDigits(int id)
  {
      return "VerifyDigits " + id;
  }
```