**Laboratorul 8**

**ASP.NET Core MVC , Views, Legare model**

**Views**

Am văzut ca metodele de acțiune pot returna obiectele ViewResult, care spun MVC să parseze un View și să returneze un răspuns HTML clientului. View-urile fac parte din modelul MVC si sunt utilizate pentru afișarea conținut pentru utilizator. Într-o aplicație ASP.NET Core MVC, un View este un fișier care conține elemente HTML și cod C#, care este procesat pentru a genera un răspuns. View-urile permit separarea prezentării datelor de logica care procesează cererile. Majoritatea aplicațiilor MVC folosesc motorul de vizualizare Razor, care facilitează amestecarea conținutului HTML și C#.

**Pregătirea proiectului pentru exemplificare.** Vom construe un proiect, folosind șablonul ASP.NET Core MVC Web Application pentru a crea un nou proiect gol numit Views. Pentru a activa Framework MVC și celelalte componente de middleware utile pentru dezvoltări, facem modificările prezentate în Exemplul 1 in clasa Startup.

**Exemplul *1.*** The Contents of the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
namespace Views {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

Vom crea folderul Controllers, si adăugam o clasă controller numita HomeController.cs și definim controllerul prezentat în Exemplul 2.

**Exemplul *2.*** The Contents of the HomeController.cs File in the Controllers Folder

```
using System;
```

```
using Microsoft.AspNetCore.Mvc;
namespace Views.Controllers {
    public class HomeController : Controller {
        public ViewResult Index() =>
                View(new string[] { "Apple", "Orange", "Pear" });
        public ViewResult List() => View();
    }
}
```

Pentru metoda de acțiune Index vom crea folderul Views / Home și adăugam in acesta un fișier View numit Index.cshtml cu conținutul afișat în Exemplul 3.

**Exemplul 3.** The Contents of the Index.cshtml File in the Views/Home Folder

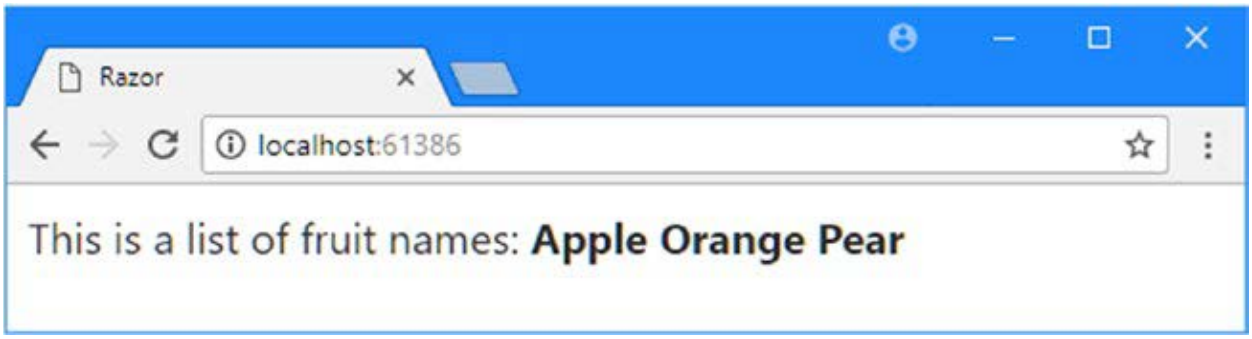
```
@model string[]
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>Razor</title>
        <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
        This is a list of fruit names:
        @foreach (string name in Model) {
                <span><b>@name</b></span>
        }
</body>
</html>
```

Cream acum si un fișier de import pentru View-uri numit _ViewImports.cshtml în folderul Views cu continulul din Exemplul 4 pentru a activa TagHelper.

**Exemplul 4.** The Contents of the _ViewImports.cshtml File in the Views Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Dacă executam proiectul, vom vedea rezultatul prezentat mai jos.



**Motorul Razor.** Vom vedea in continuare, cum analizeaza motorul Razor amestecul de elemente HTML și declarații C# și produce conținut pentru un răspuns HTTP. Razor convertește fișierele cshtml în clase C#, le compilează și apoi creează noi instanțe de fiecare dată când este necesară o vizualizare pentru a genera un rezultat. Iată clasa C# pe care Razor o creează pentru vizualizarea Index.cshtml prezentată în Exemplul 5.

**Exemplul 5.**

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Razor;
using Microsoft.AspNetCore.Mvc.Razor.Internal;
using Microsoft.AspNetCore.Mvc.Rendering;
namespace Asp {
    public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
    public IUrlHelper Url { get; private set; }
    public IViewComponentHelper Component { get; private set; }
    public IJsonHelper Json { get; private set; }
    public IHtmlHelper<string[]> Html { get; private set; }
    public override async Task ExecuteAsync() {
        Layout = null;
        WriteLiteral(@"<!DOCTYPE html><html><head>
            <meta name=""viewport"" content=""width=device-width"" />
            <title>Razor</title>
            <link asp-href-include=""lib/bootstrap/dist/css/*.min.css""
            rel=""stylesheet"" />
            </head><body class=""m-1 p-1"">This is a list of fruit names:");
        foreach (string name in Model) {
            WriteLiteral("<span><b>");
            Write(name);
            WriteLiteral("</b></span>");
        }
        WriteLiteral("</body></html>");
        }
    }
}
```

Am redat codul din clasă pentru a ușura citirea și eliminarea unor declarații C# pe care Razor le adaugă pentru instrumentare atunci când generează clasa.

Dupa cum se poate vedea numele clasei pe care o creează Razor este:

```
...
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
...
```

Razor prefixeaza numele clasei cu ASPV, urmat de numele proiectului, numele controlerului și, în final, numele fișierului View.
Multe dintre caracteristicile principale ale Razor, cum ar fi posibilitatea de a face referire la modelul de vizualizare ca @Model sunt posibile datorită clasei de bază RazorPage, din care provin clasele generate.
View-urile sunt clasele moștenite din clasa RazorPage sau clasa RazorPage<T> dacă directiva @model a fost utilizată pentru a specifica un tip de model. Clasa RazorPage oferă metode și proprietăți care pot fi utilizate în fișierele cshtml pentru a accesa caracteristicile MVC, dintre care cele mai utile sunt:
**Model**- Această proprietate returnează datele modelului furnizate de metoda de acțiune.
**ViewData** - Această proprietate returnează un obiect ViewDataDictionary care oferă acces la alte caracteristici ale datelor de vizualizare.
**ViewContext** - Această proprietate returnează un obiect ViewContext.
**Layout** - Această proprietate este utilizată pentru a specifica un layout.
**ViewBag** - Această proprietate oferă acces la obiectul ViewBag.
**TempData** - Această proprietate oferă acces la datele temporare.
**Context** - Această proprietate returnează un obiect HttpContext care descrie solicitarea curentă și răspunsul care este pregătit.
**User** - Această proprietate returnează profilul utilizatorului asociat cu această solicitare.
**RenderSection()** - Această metodă este utilizată pentru a insera o secțiune de conținut din View într-un layout.
**RenderBody()** - Această metodă introduce tot conținutul unui View care nu este inclus într-o secțiune într-un layout.
**IsSectionDefined()** - Această metodă este utilizată pentru a determina dacă un View definește o secțiune.

Pe lângă proprietățile și metodele care furnizează facilitati dezvoltatorilor, clasa RazorPage este responsabilă și de generarea conținutului de răspuns prin metoda sa ExecuteAsyc. Această metodă specifica modul în care Razor proceseaza fișierul Index.cshtml într-un set de instructiuni C#.

```
...
public override async Task ExecuteAsync() {
    Layout = null;
    WriteLiteral(@"<!DOCTYPE html><html><head>
        <meta name=""viewport"" content=""width=device-width"" />
        <title>Razor</title>
        <link asp-href-include=""lib/bootstrap/dist/css/*.min.css""
            rel=""stylesheet"" />
        </head><body class=""m-1 p-1"">This is a list of fruit names:");
    foreach (string name in Model) {
        WriteLiteral("<span><b>");
        Write(name);
        WriteLiteral("</b></span>");
    }
```

```
    WriteLiteral("</body></html>");
}
…
```

**Adăugarea de conținut dinamic la un View Razor.**Scopul final al unui View este de a permite vizualizarea unei părți din modelul de domeniu. Pentru a face acest lucru, trebuie să putem adăuga conținut dinamic la View-uri. Conținutul dinamic este generat la executie și poate fi diferit pentru fiecare solicitare. Acest lucru se opune conținutului static, cum ar fi HTML, pe care îl cream atunci când scriem aplicația și este același pentru fiecare cerere. Putem adăuga conținut dinamic View-urilor în diferitele moduri.
**Inline code** – Acesta este format din fragmente de cod, precum instrucțiuni if și foreach. Acesta este instrumentul fundamental pentru crearea de conținut dinamic în View-uri, iar unele dintre celelalte abordări sunt bazate pe acesta.
**Tag helpers** - Folosit pentru a genera atribute pe elemente HTML.
**Sections** - Utilizare pentru crearea secțiunilor de conținut care vor fi introduse în layout în anumite locații.
**Partial views** - Utilizat pentru partajarea subsecțiunilor unui View între alte View-uri. View-urile parțiale pot conține cod linie, HTML, metode helper și referințe la alte View-uri parțiale.
**View components** - Se folosește pentru crearea de controale UI reutilizabile sau widget-uri care trebuie să conțină logica de afaceri.
**Utilizarea secțiunilor Layout.** Motorul View Razor acceptă conceptul de secțiune, care ne permit să furnizam regiuni de conținut într-un layout. Secțiunile Razor oferă un control mai mare asupra părților de View care sunt introduse în layout și asupra locului unde sunt plasate. Pentru a demonstra funcția de secțiuni, am editat View-ul /Views/Home/Index.cshtml, așa cum se arată în Exemplul 6.
**Exemplul 6.** Defining Sections in the Index.cshtml File in the Views/Home Folder

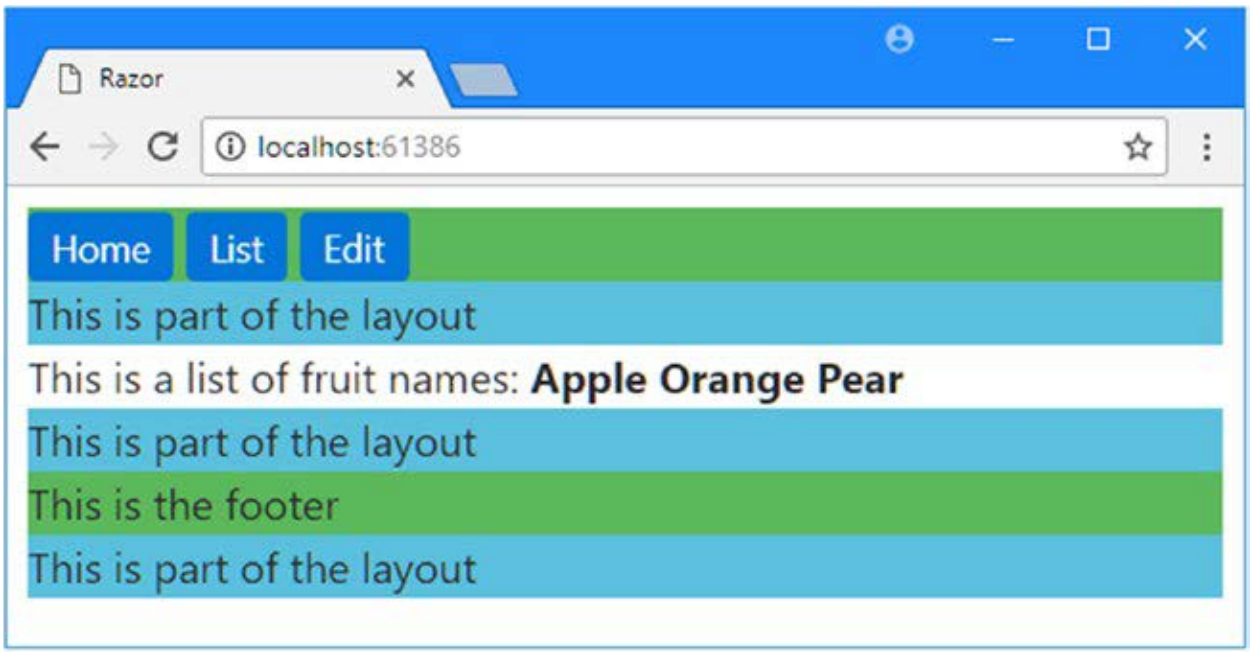
```
@model string[]
@{ Layout = "_Layout"; }
@section Header {
    <div class="bg-success">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
            <a class="btn btn-sm btn-primary" asp-action="str">@str</a>
        }
    </div>
}
This is a list of fruit names:
@foreach (string name in Model) {
<span><b>@name</b></span>
}
@section Footer {
    <div class="bg-success">
        This is the footer
    </div>
```

```
}
```
Am eliminat câteva elemente din View și am setat proprietatea Layout pentru a specifica că un fișier de layout numit _Layout.cshtml trebuie să fie utilizat pentru a reda conținutul.
Am adăugat și câteva secțiuni la View. Secțiunile sunt definite folosind expresia Razor @section urmată de un nume pentru secțiune. Am creat secțiunile numite Header și Footer. Secțiunile conțin un amestec obișnuit de HTML și expresii Razor.
Secțiunile sunt definite în View, si vor fi incluse într-un layout cu specificatia @RenderSection.
Pentru a demonstra cum funcționează acest lucru, am creat folderul Views/Shared și am adăugat un layout numit _Layout.cshtml cu conținutul afișat în Exemplul 7.

**Exemplul 7.** The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>@ViewBag.Title</title>
        <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
        @RenderSection("Header")
        <div class="bg-info">
                This is part of the layout
        </div>
        @RenderBody()
        <div class="bg-info">
                This is part of the layout
        </div>
        @RenderSection("Footer")
        <div class="bg-info">
                This is part of the layout
        </div>
</body>
</html>
```

Când Razor analizează layoutul, metoda RenderSection helper este înlocuită cu conținutul secțiunii din View cu numele specificat. Piesele din View care nu sunt conținute într-o secțiune sunt introduse în layout in locul metodei RenderBody helper.
Putem vedea efectul secțiunilor pornind aplicația, așa cum se arată în figura de mai jos.




Un View poate defini doar secțiunile la care se face referire în layout. MVC va arunca o excepție dacă încercați să definiți secțiuni în View pentru care nu există expresie @RenderSection corespunzătoare în layout.

Amestecarea secțiunilor cu restul View-ului este neobișnuită. Convenția este de a defini secțiunile fie la începutul, fie la sfârșitul View-ului, pentru a ușura observarea regiunilor de conținut care vor fi tratate ca secțiuni și care vor fi capturate de RenderBody. O altă abordare este de a defini View-ul numai în termeni de secțiuni, inclusiv una pentru body, așa cum se arată în Exemplul 8.

**Exemplul 8.** Defining a View Using Razor Sections in the Index.cshtml File in the Views/Home Folder

```
@model string[]
@{ Layout = "_Layout"; }
@section Header {
    <div class="bg-success">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
            <a class="btn btn-sm btn-primary" asp-action="str">@str</a>
        }
    </div>
}
@section Body {
    This is a list of fruit names:
    @foreach (string name in Model) {
        <span><b>@name</b></span>
    }
}
```

```
@section Footer {
    <div class="bg-success">
        This is the footer
    </div>
}
```

Acest lucru oferă View-uri mai clare și reduce șansele ca conținutul străin să fie capturat de RenderBody. Pentru a utiliza această abordare, trebuie să înlocuim apelul RenderBody cu RenderSection așa cum se arată în Exemplul 9.

**Exemplul 9.** Rendering the Body as a Section in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>@ViewBag.Title</title>
        <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
        @RenderSection("Header")
        <div class="bg-info">
            This is part of the layout
        </div>
        @RenderSection("Body")
        <div class="bg-info">
            This is part of the layout
        </div>
        @RenderSection("Footer")
        <div class="bg-info">
            This is part of the layout
        </div>
</body>
</html>
```

**Partial View.** View-urile parțiale sunt fișiere CSHTML obișnuite dar utilizarea lor le diferențiază de View-urile obișnuite Razor. Visual Studio oferă un anumit suport pentru crearea de View-uri parțiale prepopulate, dar cea mai simplă modalitate de a crea View parțial este de a crea un View obișnuit folosind MVC View Page item template. Pentru a demonstra, am adăugat un fișier numit MyPartial.cshtml în folderul Views/Home și am adăugat conținutul afișat în Exemplul 10.

**Exemplul 10.** The Contents of the MyPartial.cshtml File in the Views/Home Folder

```
<div class="bg-info">
        <div>This is the message from the partial view.</div>
        <a asp-action="Index">This is a link to the Index action</a>
</div>
```

Putem amesteca conținut static și dinamic într-un View parțial, prin urmare putem defini mesaje și link-uri utilizand tag helper.

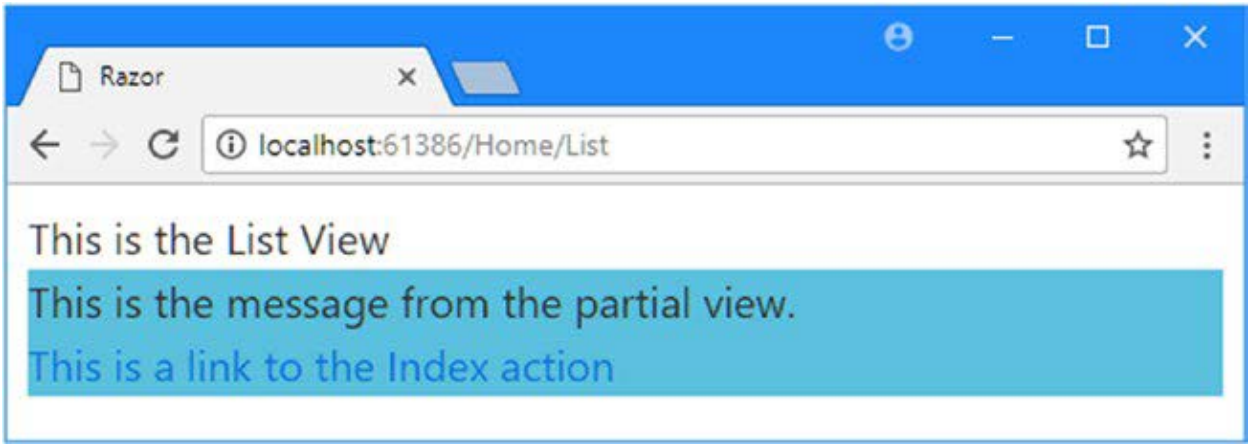
**Utilizarea unui Partial View.** Un View parțial este inclus prin apelul @Html.Partial intr-un alt View. Pentru a demonstra acest lucru, am creat un nou fișier View numit List.cshtml în folderul Views/Home și am adăugat conținutul afișat în Exemplul 11.

**Exemplul 11.** The Contents of the List.cshtml File in the Views/Home Folder

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>Razor</title>
        <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
        <body class="m-1 p-1">
        This is the List View
        @Html.Partial("MyPartial")
</body>
</html>
```

Metoda Partial este o metodă de extensie care se aplică proprietății Html adăugată clasei pe care Razor o generează din fișierul View. Acesta este un exemplu de HTML helper, care genereaza conținut dinamic în View-uri în versiunile anterioare ale MVC, dar care a fost în mare parte înlocuit de tag helpers. Argumentul transmis metodei parțiale este numele View-ului parțial, al cărui conținut este inserat în ieșirea trimisă clientului. Razor caută View-urile parțiale în același mod în care caută view-urile obișnuite (în folderele Views/ <controler> și Views/Shared).

Putem vedea efectul includerii View-ului parțial prin pornirea aplicației și navigarea către URL-ul /Home/List, așa cum se arată mai jos.




**Utilizarea View-urilor parțiale tipizate.** Putem crea View-uri parțiale tipizate și le puteți oferi obiecte de model pentru a fi utilizate atunci când este redat View-ul parțial. Pentru a

demonstra această caracteristică, cream un nou fișier View numit MyStronglyTypedPartial.cshtml în folderul Views/Home cu continutul afișat în Exemplul 12.

**Exemplul 12.** The Contents of the MyStronglyTypedPartial.cshtml File in the Views/Home Folder

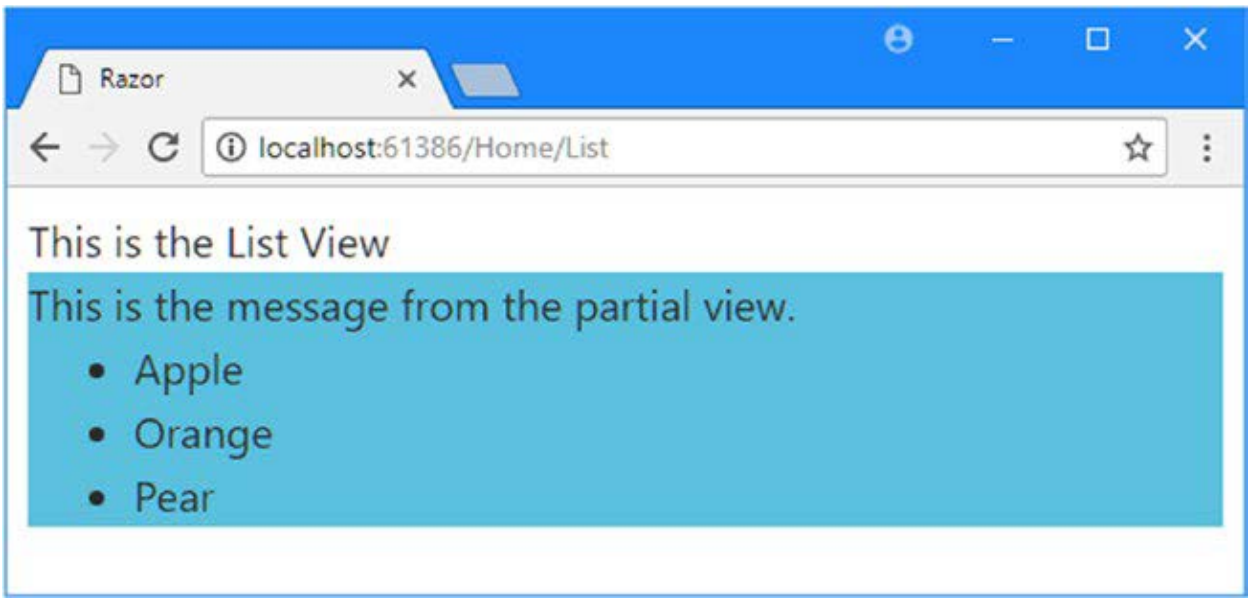
```
@model IEnumerable<string>
<div class="bg-info">
    This is the message from the partial view.
    <ul>
        @foreach (string str in Model) {
            <li>@str</li>
        }
    </ul>
</div>
```

Tipul modelului de vizualizare este definit folosind expresia standard @model și am folosit o buclă @foreach pentru a afișa conținutul obiectului modelului de vizualizare ca elemente dintr-o listă HTML. Pentru a demonstra utilizarea acestei vizualizări parțiale, am actualizat fișierul /Views/Home/List.cshtml, așa cum se arată în Exemplul 13.

**Exemplul 13.** Using a Strongly Typed Partial View in the List.cshtml File in the Views/Home Folder

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>Razor</title>
        <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
        This is the List View
        @Html.Partial("MyStronglyTypedPartial",
                new string[] { "Apple", "Orange", "Pear" })
</body>
</html>
```

Diferența față de exemplul anterior este că transmitem un argument suplimentar metodei de Partial care furnizează modelul de vizualizare. Putem vedea vizualizarea parțială tipizată pornind aplicației și navigand către URL-ul /Home/ List, așa cum se arată mai jos.




**Model Binding**
Legarea modelului este procesul de creare a obiectelor .NET care utilizează datele de la solicitarea HTTP pentru a furniza metodele de acțiune cu argumentele de care au nevoie.
Vom vedea modul în care funcționează sistemul de legare a modelului,cum se pot lega tipurile simple, tipurile complexe și colecțiile.

**Pregătirea proiectului pentru exemplificare.** Vom folosi șablonul aplicației ASP.NET Core Web Application (.NET Core) pentru a crea un nou proiect gol numit MvcModels.
**Crearea modelului și a depozitului.** Vom crea folderul Models și vom adăuga un fișier de clasă numit Person.cs, pe care ill vom folosi pentru a defini clasele și enumeratiile prezentate în Exemplul 14.
**Exemplul 14.** The Contents of the Person.cs File in the Models Folder

```
using System;
namespace MvcModels.Models {
    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }
    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
```

```
        public string Country { get; set; }
    }
    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

În continuare, vom adăuga un fișier de clasă numit Repository.cs în folderul Models și vom defini interfața și clasa de implementare afișată în Exemplul 15.

**Exemplul 15.** The Contents of the Repository.cs File in the Models Folder

```
using System.Collections.Generic;
namespace MvcModels.Models {
    public interface IRepository {
        IEnumerable<Person> People { get; }
        Person this[int id] { get; set; }
    }
    public class MemoryRepository : IRepository {
        private Dictionary<int, Person> people
                        = new Dictionary<int, Person> {
                [1] = new Person {PersonId = 1, FirstName = "Bob",
                    LastName = "Smith", Role = Role.Admin},
                [2] = new Person {PersonId = 2, FirstName = "Anne",
                    LastName = "Douglas", Role = Role.User},
                [3] = new Person {PersonId = 3, FirstName = "Joe",
                    LastName = "Able", Role = Role.User},
                [4] = new Person {PersonId = 4, FirstName = "Mary",
                    LastName = "Peters", Role = Role.Guest}
        };
        public IEnumerable<Person> People => people.Values;
        public Person this[int id] {
            get {
                return people.ContainsKey(id) ? people[id] : null;
            }
            set {
                people[id] = value;
            }
        }
    }
}
```

Interfața IRepository definește o proprietate People pentru a prelua toate obiectele din model și un indexator care permite să fie recuperate sau stocate obiecte individuale. Clasa MemoryRepository implementează interfața folosind un dicționar cu un conținut implicit.

Implementarea depozitului nu este persistentă, astfel încât starea aplicației va reveni la conținutul implicit atunci când este oprită sau repornită.

**Creare controller și View.** Vom crea folderul Controllers, si adăugam in acesta o clasă numit HomeController.cs care defineste controllerul prezentat în Exemplul 16. Controlerul se bazează pe injecția de dependență pentru a primi un depozit, pe care îl folosește în metoda Index pentru a selecta un singur obiect de Person din depozit folosind valoarea proprietății sale PersonId.

**Exemplul 16.** The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public ViewResult Index(int id) => View(repository[id]);
    }
}
```

Cream folderul Views/Home și adăugam in acesta un fișier View Razor numit Index.cshtml definit ca în Exemplul 17, care prezintă unele proprietăți din obiectul model intr-un tabel.

**Exemplul 17.** The Contents of the Index.cshtml File in the Views/Home Folder

```
@model Person
@{ Layout = "_Layout"; }
<div class="bg-primary m-1 p-1 text-white"><h2>Person</h2></div>
<table class="table table-sm table-bordered table-striped">
        <tr><th>PersonId:</th><td>@Model.PersonId</td></tr>
        <tr><th>First Name:</th><td>@Model.FirstName</td></tr>
        <tr><th>Last Name:</th><td>@Model.LastName</td></tr>
        <tr><th>Role:</th><td>@Model.Role</td></tr>
</table>
```

View-ul Index.cshtml se bazează pe un layout partajat. Am creat folderul Views/Shared și am adăugat un layout numit _Layout.cshtml, al cărui conținut poate fi văzut în Exemplul 18.

**Exemplul 18.** The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="width=device-width" />
        <title>@ViewBag.Title</title>
        <link asp-href-include="/lib/bootstrap/dist/**/*.min.css" rel="stylesheet" />
        @RenderSection("scripts", false)
</head>
<body class="m-1 p-1">
```

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

Layout-ul include un element link pentru foaia de stil Bootstrap și redă conținutul View-ului. Există și o secțiune scripts opționala. Pentru a simplifica View-urile utilizate, am adăugat spațiul de nume care conține clasele de model la fișierul _ViewImports.cshtml din folderul Views, așa cum se arată în Exemplul 19.

**Exemplul 19.** Importing Namespaces in the _ViewImports.cshtml File in the Views Folder

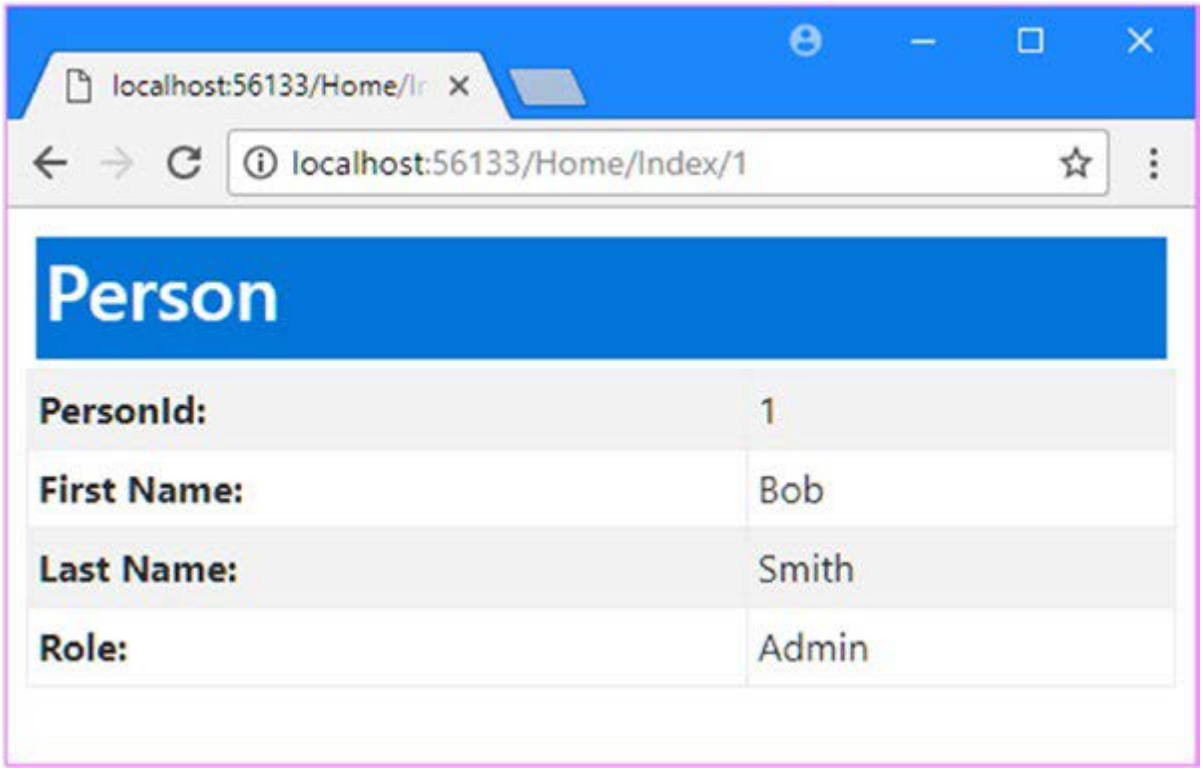
```
@using MvcModels.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

**Configurarea aplicației.** Pentru a finaliza configurația inițială a aplicației din exemplu, vom activa framework-ul MVC și un middleware util pentru dezvoltare în clasa Startup, așa cum se arată în Exemplul 20. De asemenea, am creat un serviciu pentru depozit, astfel încât controlorul să poată avea acces la modelul de date.

**Exemplul 20.** The Contents of the Startup.cs File in the MvcModels Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using MvcModels.Models;
namespace MvcModels {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

Daca pornim aplicația și solicitam URL-ul /Home/Index/1 vom vedea rezultatul prezentat mai jos.




Legarea modelului este o punte între solicitarea HTTP și metodele de acțiune C#. Majoritatea aplicațiilor MVC se bazează pe legarea modelului într-o anumită măsură.
Adresa URL pe care am solicitat-o conținea valoarea proprietății PersonId a obiectului Persoana pe care doream să o vizualizam, astfel:
/Home/Index/1
MVC a a folosit id-ul din URL ca argument atunci când a apelat la metoda Index din clasa controllerului Home pentru a furniza cererea.

```
...
public ViewResult Index(int id) => View(repository[id]);
...
```

Pentru a putea invoca metoda Index, MVC are nevoie de o valoare pentru argumentul id, iar furnizarea acestei valori este responsabilitatea sistemului de legare a modelului, care este responsabil pentru furnizarea de valori care pot fi utilizate pentru a invoca metode de acțiune.
Sistemul de legare a modelului se bazează pe model binders, care sunt componente responsabile pentru furnizarea valorilor de date dintr-o parte a cererii sau aplicației. Binderii impliciti ai modelului caută valorile datelor în trei locuri:

- Form data values
- Routing variables
- Query strings

Fiecare sursă de date este inspectată în ordine până la găsirea unei valori pentru argument.

Valoarea implicită pentru tipurile nullable este nulă, ceea ce îmi permite să diferențiez între cererile în care solicitarea nu conține o valoare care poate fi analizată într-un int și cereri care admit valoarea 0.

**Legare tipuri complexe.** Când parametrul metodei de acțiune este un tip complex (cu alte cuvinte, orice tip care nu poate fi analizat dintr-o singură valoare de șir), atunci procesul de legare a modelului folosește reflecția pentru a obține un set de proprietăți publice ale tipului țintă și realizează procesul de legare la fiecare pe rând. Pentru a demonstra cum funcționează acest lucru, am adăugat două metode de acțiune la controlerul Home, așa cum se arată în Exemplul 21.

**Exemplul 21.** Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
        public ViewResult Create() => View(new Person());
        [HttpPost]
        public ViewResult Create(Person model) => View("Index", model);
    }
}
```

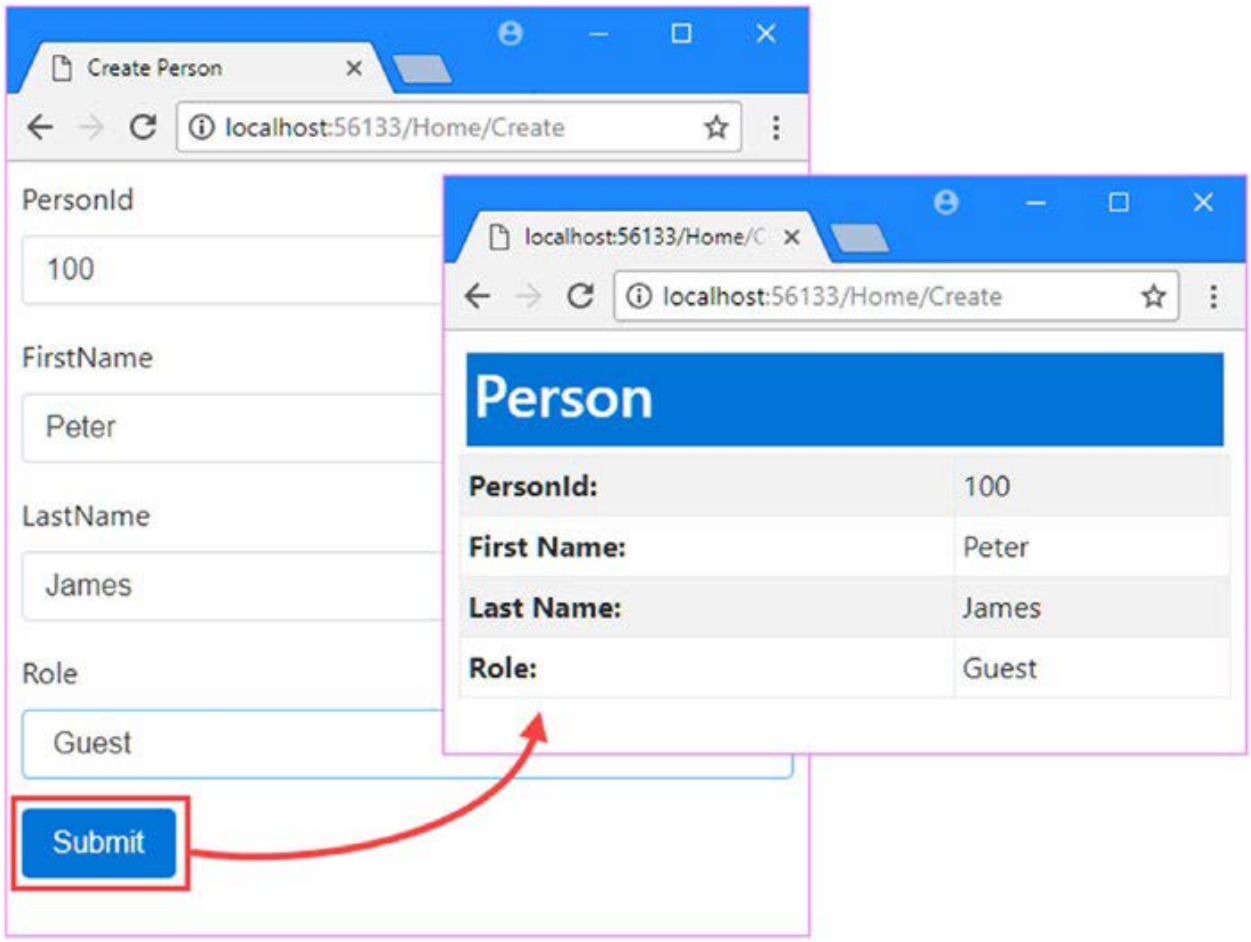
Versiunea metodei Create fără parametri creează un obiect Person și îl transmite metodei View, care are ca efect selectarea View-ului implicit asociat cu acțiunea. Adăugam un fișier View numit Create.cshtml în folderul Views/Home definit ca în Exemplul 22.

**Exemplul 22.** The Contents of the Create.cshtml File in the Views/Home Folder

```
@model Person
@{
        ViewBag.Title = "Create Person";
        Layout = "_Layout";
}
<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="PersonId"></label>
        <input asp-for="PersonId" class="form-control" />
```

```
    </div>
    <div class="form-group">
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="LastName"></label>
        <input asp-for="LastName" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Role"></label>
        <select asp-for="Role" class="form-control"
        asp-items="@new SelectList(Enum.GetNames(typeof(Role)))"></select>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Acest View conține un form care permite furnizarea de valori pentru unele dintre proprietățile unui obiect Person și conține un element de formular care postează datele înapoi la versiunea metodei Create din controlerul Home care a fost decorata cu atributul HttpPost.
Metoda de acțiune care primește datele formularului utilizează View-ul /View/Home/Index.cshtml pentru a o afișa.
Puteți vedea cum funcționează acest lucru pornind aplicația, si accesând /Home / Create, completând formularul și făcând clic pe butonul Submit, așa cum se arată mai jos.




Când datele formularului sunt trimise către server, procesul de legare a modelului descoperă că metoda de acțiune necesită un tip complex: un obiect Person. Clasa Person este examinată pentru a descoperi proprietățile sale publice. Pentru fiecare proprietate publică care returnează un tip de proprietate simplă, binder-ul model încearcă să localizeze o valoare a cererii, la fel cum s-a întâmplat în exemplul precedent.
Deci, de exemplu, binder-ul model găsește proprietatea PersonId și caută o valoare PersonId în aceleași locații in care a căutat o valoare id în secțiunea anterioară. Deoarece datele formularului conțin o valoare adecvată, care este configurată folosind tag helper-ul asp-for ca un element de intrare, aceasta este valoarea care va fi utilizată.
Dacă o proprietate necesită un alt tip complex, procesul se repetă pentru noul tip. Setul de proprietăți publice este obținut, iar binder-ul încearcă să găsească valori pentru toate proprietățile. Diferența este că numele proprietăților pot fi incuibărate. De exemplu, proprietatea HomeAddress a clasei Person este de tipul Address, așa cum este evidențiat aici:

```
using System;
namespace MvcModels.Models {
    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
```

```
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }
    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }
    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

Când căuta o valoare pentru proprietatea Line1, binder-ul de model caută o valoare pentru HomeAddress.
Utilizarea prefixelor înseamnă că View-urile trebuie să includă informațiile pe care le leagă modelul de legătură. Acest lucru se realizează cu ușurință folosind tag helpers, care adaugă automat prefixele necesare elementelor pe care le transformă. În Exemplul 22, am extins formularul astfel încât să afiseze date despre adresă.
**Exemplul 22.** Updating the Form in the Create.cshtml File in the Views/Home Folder

```
@model Person
@{
    ViewBag.Title = "Create Person";
    Layout = "_Layout";
}
<form asp-action="Create" method="post">
    <div class="form-group">
        <label asp-for="PersonId"></label>
        <input asp-for="PersonId" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="LastName"></label>
        <input asp-for="LastName" class="form-control" />
    </div>
```

```
    <div class="form-group">
        <label asp-for="Role"></label>
        <select asp-for="Role" class="form-control"
        asp-items="@new SelectList(Enum.GetNames(typeof(Role)))"></select>
    </div>
    <div class="form-group">
        <label asp-for="HomeAddress.City"></label>
        <input asp-for="HomeAddress.City" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="HomeAddress.Country"></label>
        <input asp-for="HomeAddress.Country" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Când utilizam un tag helper, numele proprietății cuibărit este specificat folosind convențiile C#, HomeAddress.Country.

**Legarea la Array-uri și Colecții.** Procesul de legare a modelului are câteva caracteristici pentru legarea datelor de solicitare la tablouri și colecții, pe care le vom vedea în continuare.

**Legare la tablouri.** O caracteristică elegantă a binder-ului model implicit este modul în care acceptă parametrii metodei de acțiune care sunt tablouri.

Pentru a demonstra acest lucru, am adăugat o nouă metodă la controlerul Home numita Name, pe care o puteți vedea în Exemplul 23.

**Exemplul 23.** Adding an Action Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index(int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
        public ViewResult Create() => View(new Person());
        [HttpPost]
        public ViewResult Create(Person model) => View("Index", model);
        public ViewResult Names(string[] names) => View(names ?? new string[0]);
```

```
    }
}
```
Metoda de acțiune Names are un parametru de tip array numit names. Binder-ul de model va căuta orice element de date care se numește names și va crea un tablou care conține acele valori. Pentru a furniza vizualizarea metodei de acțiune, vom crea un View Razor numit Nume.cshtml în folderul Views/Home și vom adăugat continutul prezentat în Exemplul 24.

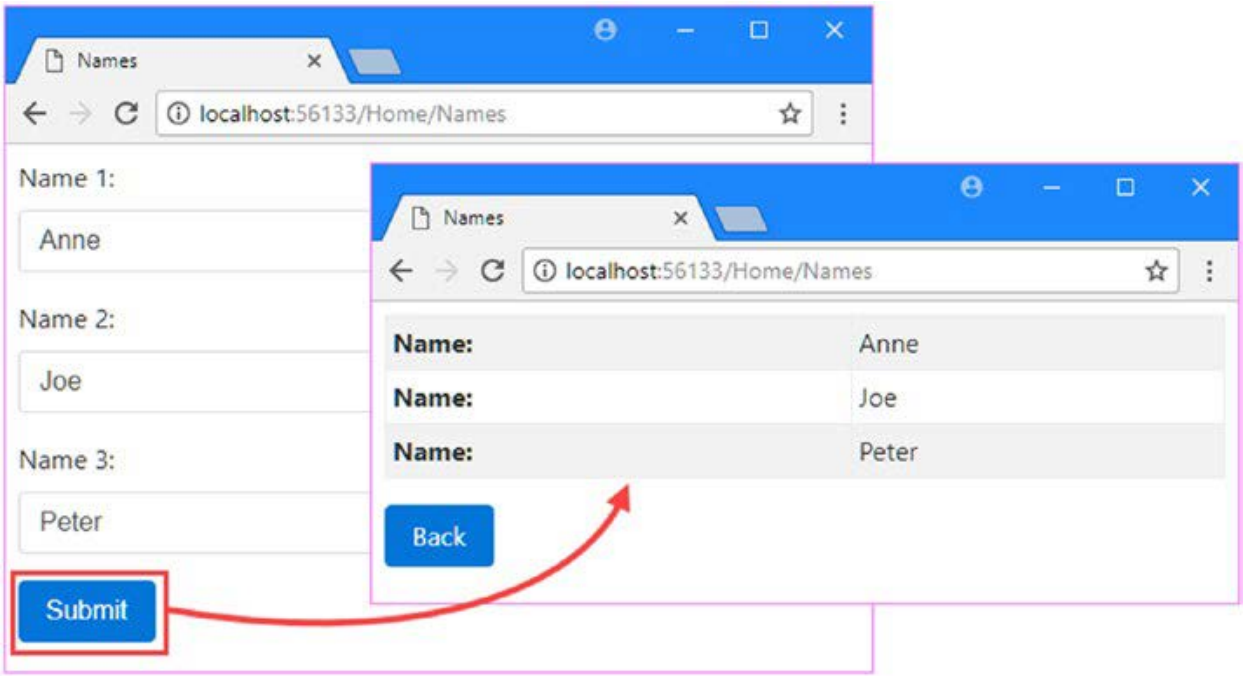
**Exemplul 24.** The Contents of the Names.cshtml File in the Views/Home Folder

```
@model string[]
@{
    ViewBag.Title = "Names";
    Layout = "_Layout";
}
@if (Model.Length == 0) {
    <form asp-action="Names" method="post">
        @for (int i = 0; i < 3; i++) {
            <div class="form-group">
                <label>Name @(i + 1):</label>
                <input id="names" name="names" class="form-control" />
            </div>
        }
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
} else {
    <table class="table table-sm table-bordered table-striped">
        @foreach (string name in Model) {
            <tr><th>Name:</th><td>@name</td></tr>
        }
    </table>
    <a asp-action="Names" class="btn btn-primary">Back</a>
}
```
Acest View afișează conținut diferit în funcție de numărul de articole disponibile în modelul de vizualizare. Dacă nu există articole, atunci View-ul afișează un formular care conține trei elemente de intrare identice, astfel:

```
...
<form method="post" action="/Home/Names">
    <div class="form-group">
        <label>Name 1:</label>
        <input id="names" name="names" class="form-control" />
    </div>
    <div class="form-group">
        <label>Name 2:</label>
        <input id="names" name="names" class="form-control" />
    </div>
    <div class="form-group">
```

```
        <label>Name 3:</label>
        <input id="names" name="names" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
...
```

Pentru a vedea efectul, pornim aplicația, navigam la URL-ul /Home/Name si completam formularul. Când trimitem formularul completat, vom vedea că toate valorile introduse sunt afișate, așa cum se arată mai jos.




**Legarea la colecții.** Procesul de legare este acceptat nu numai pentru tablouri ci si pentru colectii. În Exemplul 25 am schimbat tipul parametrului metodei de acțiune Name cu o listă tipizată.

**Exemplul 25.** Using a Strongly Typed Collection in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Collections.Generic;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        // ...other action methods omitted for brevity...
        public ViewResult Names(IList<string> names) =>
```

```
            View(names ?? new List<string>());
    }
}
```

Am folosit interfața IList <T>. Nu este necesar să specificam o clasă concretă implementata, desi este posibil si acest luctu. În Exemplul 26, am modificat fișierul View names.cshtml pentru a utiliza noul tip de model.

**Exemplul 26.** Using a Collection As the Model Type in the Names.cshtml File in the Views/Home Folder

```
@model IList<string>
@{
    ViewBag.Title = "Names";
    Layout = "_Layout";
}
@if (Model.Count == 0) {
    <form asp-action="Names" method="post">
        @for (int i = 0; i < 3; i++) {
            <div class="form-group">
                <label>Name @(i + 1):</label>
                <input id="names" name="names" class="form-control" />
            </div>
        }
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
} else {
    <table class="table table-sm table-bordered table-striped">
        @foreach (string name in Model) {
            <tr><th>Name:</th><td>@name</td></tr>
        }
    </table>
    <a asp-action="Names" class="btn btn-primary">Back</a>
}
```

Funcționalitatea acțiunii Nume este neschimbată, dar acum sunt capabil să lucrez cu o clasă colecție și nu numai cu un tablou.

**Specificarea unei surse de legare la model.**

După cum am explicat la începutul capitolului, procesul implicit de legare a modelului caută date în trei locuri: valorile de date ale formularului, datele de rutare și șirul de interogare a cererii.

Secvența implicită de căutare nu este întotdeauna utilă, fie pentru că dorim întotdeauna ca datele să provină dintr-o anumită parte a cererii, fie pentru că dorim să utilizam o sursă de date care nu este căutată în mod implicit. Funcția de legare a modelului include un set de atribute care sunt utilizate pentru a trece peste comportamentul implicit de căutare, așa cum este descris în continuare.

**FromForm -** Acest atribut este utilizat pentru a selecta datele din formular ca sursă de date obligatorii. Numele parametrului este utilizat pentru a localiza o valoare a formularului în mod implicit, dar aceasta poate fi modificată folosind proprietatea Name, care permite specificarea unui alt nume.
**FromRoute -** Acest atribut este utilizat pentru a selecta sistemul de rutare ca sursă de date obligatorii. Numele parametrului este utilizat pentru a localiza o valoare a datelor de rută în mod implicit, dar aceasta poate fi modificată folosind proprietatea Name, care permite specificarea unui alt nume.
**FromQuery -** Acest atribut este utilizat pentru a selecta șirul de interogare ca sursă de date obligatorii. Numele parametrului este utilizat pentru a localiza o valoare a șirului de interogare în mod implicit, dar aceasta poate fi modificată folosind proprietatea Name, care permite să fie specificată o cheie de interogare diferită.
**FromHeader -** Acest atribut este utilizat pentru a selecta un header ca sursă de date obligatorii. Numele parametrului este folosit ca nume de header în mod implicit, dar acesta poate fi schimbat folosind proprietatea Name, care permite specificarea unui alt nume de header.
**FromBody -** Acest atribut este utilizat pentru a specifica faptul că body ar trebui utilizat ca sursă de date de legare.
Atributele FromForm, FromRoute și FromQuery vă permit să specificați că datele de legare a modelului vor fi obținute de la una dintre locațiile standard, dar fără secvența normală de căutare. De exemplu, daca folosim această adresă URL:
/Home/Index/3?id=1
Această adresă URL conține două valori posibile care pot fi utilizate pentru parametrul id al metodei de acțiune Index de pe controlerul principal. Sistemul de rutare va atribui segmentul final al adresei URL unei variabile numite id, care este definit în modelul URL din clasa Startup, iar șirul de interogare conține, de asemenea, o valoare id. Modelul de căutare implicit înseamnă că datele de legare a modelului vor fi preluate din datele de rută și șirul de interogare va fi ignorat. Pentru a schimba acest comportament, în Exemplul 27, am aplicat atributul FromQuery metodei de acțiune.
**Exemplul 27.** Selecting the Query String in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index([FromQuery] int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
```

```
        }
    }
}
```

Am aplicat atributul FromQuery parametrului id, ceea ce înseamnă că numai șirul de interogare va fi utilizat atunci când procesul de legare a modelului caută o valoare de date id.
Atributul FromHeader permite folosirea anteturilor de solicitare HTTP ca sursă pentru date obligatorii. În Exemplul 28, am adăugat o metodă simplă de acțiune controlerului Home care primește un parametru legat folosind date dintr-un antet de solicitare HTTP standard.
**Exemplul 28.** Model Binding from a Header in the HomeController.cs Filein the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index([FromQuery] int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
        public string Header([FromHeader]string accept) => $"Header: {accept}";
    }
}
```

Metoda de acțiune Header definește un parametru accept, valoarea care va fi preluată din headerul Accept în cererea curentă și returnată ca rezultat al metodei. Dacă rulați aplicația și solicitați adresa URL /Home/Header, veți vedea un astfel de rezultat (deși rezultatul exact poate diferi în funcție de browserul pe care îl utilizați):

```
Header: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

Nu toate datele trimise de clienți sunt trimise ca date de formular, cum ar fi atunci când un client JavaScript trimite date JSON către un controler API. Atributul FromBody specifică faptul că organismul de solicitare trebuie decodat și utilizat ca sursă de date de legare a modelului. În Exemplul 29, am adăugat o metoda de acțiune noua Body care demonstrează modul în care funcționează acest lucru.
**Exemplul 29.** Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
namespace MvcModels.Controllers {
    public class HomeController : Controller {
```

```
        private IRepository repository;
        public HomeController(IRepository repo) {
            repository = repo;
        }
        public IActionResult Index([FromQuery] int? id) {
            Person person;
            if (id.HasValue && (person = repository[id.Value]) != null) {
                return View(person);
            } else {
                return NotFound();
            }
        }
        public ViewResult Header(HeaderModel model) => View(model);
        public ViewResult Body() => View();
        [HttpPost]
        public Person Body([FromBody]Person model) => model;
    }
}
```

Am decorat parametrul pentru metoda Body care acceptă cererile POST cu atributul FromBody, ceea ce înseamnă că conținutul din body va fi decodat și utilizat pentru legarea modelului.

**Exercitii.**

1. **Studiati si testati exemplele din textul de mai sus.**