

Laboratorul 7

ASP.NET Core MVC , Rutare, Controllere si Actiuni

Routarea

ASP.NET presupune că a existat o relație directă între adresele URL solicitate și fișierele de pe server. Sarcina serverului era să primească solicitarea din browser și să livreze ieșirea din fișierul corespunzător. Această abordare a funcționează foarte bine pentru formularele web, unde fiecare pagină ASPX este un fișier care ofera un răspuns autonom la o solicitare. Aceasta abordare nu mai are sens pentru o aplicație MVC, în care solicitările sunt procesate prin metode de acțiune ale clasei controller și nu există o corelație unu la unu cu fișierele de pe disc. Pentru a gestiona adresele URL MVC, platforma ASP.NET folosește sistemul de rutare ASP.NET Core MVC. Sistemul de rutare are două funcții si anume :

- Sa examineze o adresă URL primită și sa selecteze controllerul și acțiunea pentru a gestiona solicitarea.
- Sa genereze adrese URL de ieșire. Acestea sunt adresele URL care apar în HTML-ul redat din View-uri, astfel încât atunci când utilizatorul face clic pe link sa fie invocate o acțiune dorită (moment în care devine din nou o adresă URL).

Sistemul de rutare permite gestionarea flexibilă a cererilor fără ca adresele URL să fie legate de structura claselor din proiectul Visual Studio. Maparea între adresele URL și controlerele și metodele de acțiune este definită în fișierul **Startup.cs** sau prin aplicarea atributului Route la controlere.

Pentru exemplele din aceasta secțiune vom crea un proiect folosind șablonul ASP.NET Core Web Application (.NET Core) gol, pe care îl numim UrlsAndRoutes. Pentru a adăuga suport pentru MVC Framework, paginile de eroare pentru dezvoltatori și fișierele statice, adăugăm configurațiile prezentate în Exemplul 1 la clasa Startup.

Exemplul 1. Configuring the Application in the Startup.cs File in the UrlsAndRoutes 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;

namespace UrlsAndRoutes {

public class Startup {

public void ConfigureServices(IServiceCollection services) {

```

        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvc();
    }
}
}

```

Crearea clasei de Model. Scopul nostru este de a vedea cum se potrivesc adreselor URL ale solicitării la acțiunile controllerului. De aceea singura clasă de model de care avem nevoie specifică informații despre controller și metoda de acțiune care a fost selectată pentru a procesa o solicitare. Vom crea folderul Modele și adăugăm o clasă numită Result.cs, definită ca în Exemplul 2.

Exemplul 2. The Contents of the Result.cs File in the Models Folder

```

using System.Collections.Generic;
namespace UrlsAndRoutes.Models {
    public class Result {
        public string Controller { get; set; }
        public string Action { get; set; }
        public IDictionary<string, object> Data { get; }
            = new Dictionary<string, object>();
    }
}

```

Proprietățile **Controller** și **Action** vor fi utilizate pentru a indica modul în care a fost procesată o solicitare, iar dicționarul Data va fi utilizat pentru a stoca alte detalii despre cererea produsă de sistemul de rutare.

Crearea clasei controllerelor. Avem nevoie de niște controlere simple pentru a demonstra cum funcționează rutarea. Vom crea folderul Controllers și vom adăuga o clasă numită HomeController.cs, al cărui conținut este afișat în Exemplul 3.

Exemplul 3. The Contents of the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;
namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            }
        );
    }
}

```

```

    });
}
}

```

Metoda de acțiune Index definită de controlerul Home apelează la metoda View pentru a reda un View numit Result, pe care va trebui să îl construim, și oferă un obiect Result ca obiect model. Proprietățile obiectului model sunt setate folosind funcția nameof și vor fi utilizate pentru a indica ce controler și metodă de acțiune au fost utilizate pentru a deservei o solicitare.

Dupa același tipar adăugam o clasa CustomerController.cs în folderul Controllers și definim controlerul afișat în Exemplul 4.

Exemplul 4. The Contents of the CustomerController.cs File in the Controllers Folder
using Microsoft.AspNetCore.Mvc;

using UrlsAndRoutes.Models;

```

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {
        public IActionResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });
        public IActionResult List() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(List)
            });
    }
}

```

Al treilea și ultimul controler este definit într-un fișier numit AdminController.cs, pe care l-am adăugat în folderul Controllers, așa cum se arată în Exemplul 5.

Exemplul 5. The Contents of the AdminController.cs File in the Controllers Folder

using Microsoft.AspNetCore.Mvc;

using UrlsAndRoutes.Models;

```

namespace UrlsAndRoutes.Controllers {
    public class AdminController : Controller {
        public IActionResult Index() => View("Result",
            new Result {
                Controller = nameof(AdminController),
                Action = nameof(Index)
            });
    }
}

```

Crearea View-ului. Am specificat View-ul Result în toate acțiunile definite, ceea ce îmi permite să creez un singur View care va fi partajată de toți controlorii. Vom crea folderul Views / Shared și vom adăuga un View numit Result.cshtml, al cărui conținut este afișat în Exemplul 6.

Exemplul 6. The Contents of the Result.cshtml File in the Views/Shared Folder

```
@model Result
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped table-sm">
        <tr><th>Controller:</th><td>@Model.Controller</td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
        }
    </table>
</body>
</html>
```

View-ul conține un tabel care afișează proprietățile din obiectul model într-un tabel care este conceput folosind Bootstrap. Pentru a adăuga Bootstrap la proiect, am folosit șablonul Bower Configuration File item pentru a crea fișierul bower.json și am adăugat pachetul Bootstrap în secțiunea de dependențe, așa cum se arată în Exemplul 7.

Exemplul 7. Adding the Bootstrap Package in the bower.json File in the UrlsAndRoutes Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Pregătirea finală este crearea fișierului **_ViewImports.cshtml** din folderul Views, care configurează elementele tag helpers încorporate pentru a fi utilizate în View-urile Razor și importă spațiul de nume al modelului, așa cum se arată în Exemplul 8.

Exemplul 8. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@using UrlsAndRoutes.Models
```

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

Configurația din clasa **Startup** nu conține instrucțiuni privind modul în care MVC ar trebui să asocieze cererile HTTP către controlleri și acțiuni. Ca urmare când pornim aplicația, orice adresă URL pe care o solicităm va avea ca rezultat un răspuns 404 – Not Found.

Introducerea pattern-urilor URL. Sistemul de rutare folosește un set de rute. Aceste rute conțin împreună schema URL pentru o aplicație, care este setul de adrese URL pe care aplicația noastră le va recunoaște și la care va răspunde.

Nu este necesar să scriem manual toate adresele URL pe care dorim să le accesăm în aplicația noastră. Fiecare rută conține un pattern URL, care este comparat cu adresele URL primite. Dacă o adresă URL de intrare se potrivește cu patternul, atunci sistemul de rutare este utilizat pentru a procesa acea adresă URL. Iată o adresă URL simplă: <http://mysite.com/Admin/Index>.

Adresele URL pot fi defalcate pe segmente. Acestea sunt părțile URL, excluzând numele de host și șirul de interogare, care sunt separate de caracterul /. În URL-ul din exemplu, există două segmente. Primul segment conține cuvântul Admin, iar al doilea segment conține cuvântul Index. Este evident că primul segment se referă la controler și al doilea segment se raportează la acțiune. Trebuie să exprim această relație folosind un pattern URL care poate fi înțeles de sistemul de rutare. Iată un pattern URL care se potrivește cu URL-ul din exemplu:

{controller}/{action}

Atunci când prelucrează o cerere HTTP de intrare, sistemul de rutare trebuie să găsească un pattern URL cu care se potrivește URL-ul și să extragă valorile din URL pentru variabilele de segment definite în pattern. Variabilele segment sunt exprimate folosind acolade {și}. Patternul din exemplu are două variabile de segment controller și action, deci valoarea variabilei controller va fi Admin, iar valoarea variabilei action va fi Index.

O aplicație MVC va avea de obicei mai multe rute, iar sistemul de rutare va compara adresa URL primită cu patternul URL până când va găsi o potrivire. În mod implicit, un pattern se va potrivi cu orice adresă URL care are numărul corect de segmente. De exemplu, patternul {controller} / {action} se va potrivi cu orice adresă URL care are două segmente, așa cum este descris mai jos.

http://mysite.com/Admin/Index	controller = Admin action = Index
http://mysite.com/Admin	No match—too few segments
http://mysite.com/Admin/Index/Soccer	No match—too many segments

Exemplul acesta evidențiază două comportamente cheie implicite ale patternurilor URL.

- Se vor potrivi numai cu adresele URL care au același număr de segmente ca patternul.
- Conținutul segmentelor cu care se potrivesc este arbitrar. Dacă o adresă URL are numărul corect de segmente, modelul va extrage valoarea fiecărui segment pentru o variabilă de segment, oricare ar fi ea.

Acestea sunt comportamentele implicite, care sunt fundamentale pentru înțelegerea mecanismului de potrivire a patternurilor URL. Vom vedea cum poate fi modificat comportamentul implicit al acestor patternuri.

Crearea și înregistrarea unei rute simple. Rutele sunt definite în fișierul **Startup.cs** și sunt transmise ca argumente la metoda **UseMvc** care este utilizată pentru a configura MVC în metoda **Configure**. Exemplul 9 arată o rută de bază care mapează cererile către controlerele din aplicația din exemplu.

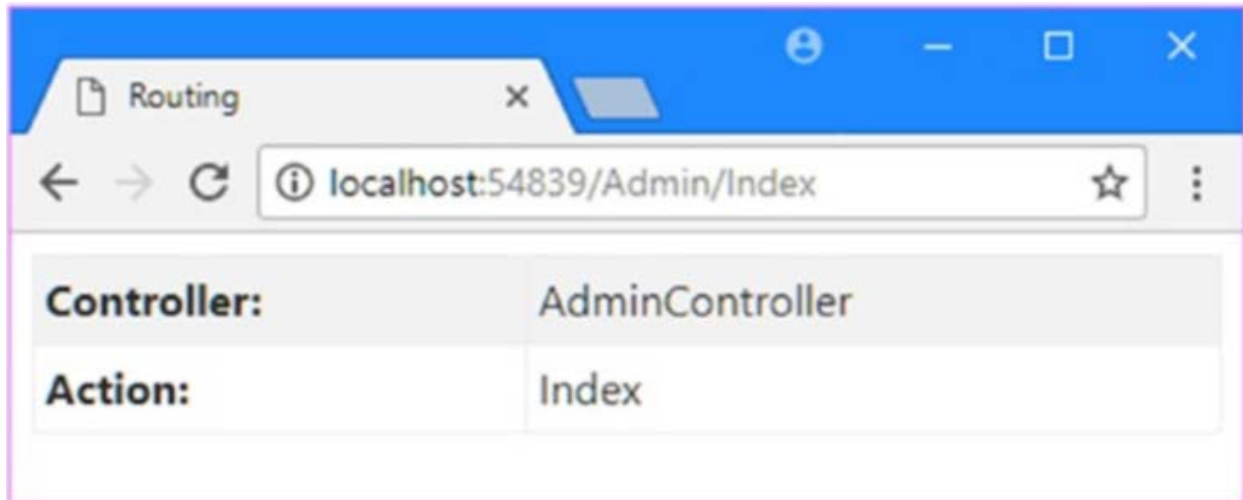
Exemplul 9. Defining a Basic Route in the Startup.cs File in the UrlsAndRoutes Folder

```
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;
namespace UrlsAndRoutes {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "default", template: "{controller}/{action}");
            });
        }
    }
}
```

Rutele sunt create folosind o expresie lambda transmisă ca argument la metoda de configurare **UseMvc**. Expresia primește un obiect care implementează interfața **IRouteBuilder** din spațiul de nume **Microsoft.AspNetCore.Routing**, iar rutele sunt definite folosind metoda de extensie **MapRoute**.

Putem acum vedea efectul modificărilor pe care le-am făcut la rutare dacă lansăm aplicația și navigăm la o adresă URL care se potrivește cu modelul `{controller} / {action}`, vom vedea un rezultat precum cel prezentat mai jos.



Motivul pentru care URL-ul rădăcină al aplicației nu funcționează este faptul că ruta pe care am adăugat-o în fișierul Startup.cs nu spune MVC cum să selecteze o clasă controller și o acțiune atunci când adresa URL solicitată nu are segmente.

Definirea valorilor implicite. Am explicat mai devreme că modelele URL se vor potrivi doar cu adresele URL cu numărul specificat de segmente. O modalitate de a schimba acest comportament este utilizarea valorilor implicite. O valoare implicită este aplicată atunci când adresa URL nu conține un segment care poate fi asociat cu patternul de rutare. Exemplul 10 definește o ruta care utilizează o valoare implicită.

Exemplul 10. Providing a Default Value in the Startup.cs File in the UrlsAndRoutes 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;

namespace UrlsAndRoutes {

public class Startup {

public void ConfigureServices(IServiceCollection services) {
 services.AddMvc();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
 app.UseDeveloperExceptionPage();
 app.UseStatusCodePages();
 app.UseStaticFiles();
 app.UseMvc(routes => {
 routes.MapRoute(

```

        name: "default",
        template: "{controller}/{action}",
        defaults: new { action = "Index" });
    });
}
}
}

```

Valorile implicite sunt furnizate ca proprietăți într-un tip anonim, trecute la metoda MapRoute ca argument implicit. În listă, am furnizat o valoare implicită a Indexului pentru variabila de acțiune. Acest traseu se va potrivi cu toate adresele URL cu două segmente, așa cum a făcut anterior. De exemplu, dacă URL-ul `http://mydomain.com/Home/Index` este solicitat, ruta va extrage Home ca valoare pentru controller și va extrage Index ca valoare pentru action.

Dar acum că există o valoare implicită pentru segmentul de acțiune, ruta se va potrivi și cu URL-urile cu un singur segment. Când prelucrați o adresă URL de un singur segment, sistemul de rutare va extrage valoarea controlorului din URL și va utiliza valoarea implicită pentru variabila de acțiune.

Definirea valorilor implicite în linie. Valorile implicite pot fi, de asemenea, exprimate ca parte a modelului URL, care este un mod mai concis de a exprima rutele, așa cum se arată în Exemplul 11. Sintaxa liniară poate fi utilizată numai pentru a furniza valori implicite pentru variabilele care fac parte din modelul URL, dar, după cum veți afla, este adesea util să puteți furniza valori implicite în afara aceluiași model. Din acest motiv, este util să înțelegeți ambele moduri de exprimare a valorilor prestabilite.

Exemplul 11. Defining Inline Default Values in the Startup.cs File in the UrlsAndRoutes 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;
namespace UrlsAndRoutes {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
        }
    }
}

```



```

app.UseMvc(routes => {
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}");
    });
}
}
}

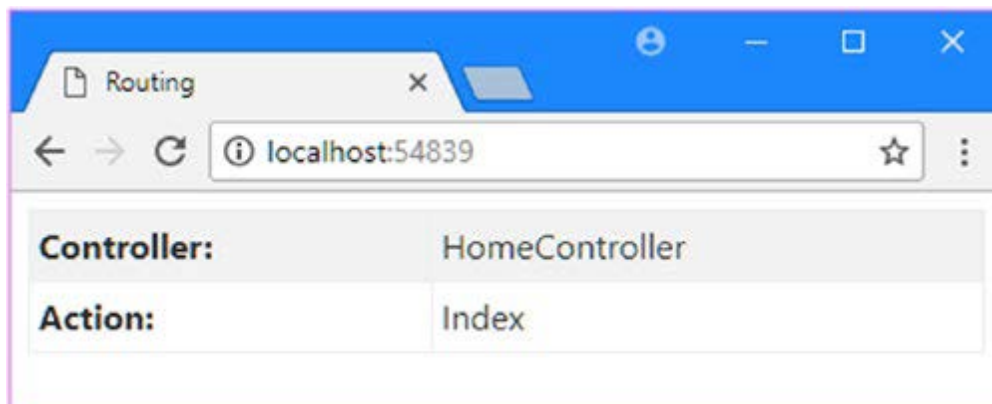
```

În acest exemplu putem potrivi adrese URL care nu conțin deloc variabile de segment, bazându-ne doar pe valorile implicite pentru a identifica acțiunea și controlorul.

Prin furnizarea de valori implicite atât pentru variabilele de controller, cât și pentru variabilele de acțiune, ruta se va potrivi cu adresele URL care au zero, unul sau două segmente, așa cum se arată mai jos.

0 / controller = Home action = Index
1 /Customer controller = Customer action = Index
2 /Customer/List controller = Customer action = List
3 /Customer/List/All No match—too many segments

Putem vedea efectul valorilor implicite pornind aplicația din exemplu. Atunci când browserul solicită URL-ul rădăcină pentru aplicație, vor fi utilizate valorile implicite pentru variabilele controlerului și ale segmentului de acțiune, ceea ce va determina MVC să invoce metoda de acțiune Index pe controlerul de pornire, așa cum se arată în figura de mai jos.



Utilizarea segmentelor URL statice. Nu toate segmentele dintr-un pattern URL trebuie să fie variabile. Putem crea modele care au segmente statice. Să presupunem că aplicația trebuie să corespundă adreselor URL prefixate cu Public, astfel:

<http://mydomain.com/Public/Home/Index>

Acest lucru se poate face utilizând un pattern URL precum cel prezentat în Exemplul 12.

Exemplul 12. Using Static Segments in the Startup.cs File in the UrlsAndRoutes Folder

```

namespace UrlsAndRoutes {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}");
                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}/{action=Index}");
            });
        }
    }
}

```

Acest nou model se va potrivi doar cu adresele URL care conțin trei segmente, dintre care primul trebuie să fie Public. Celelalte două segmente pot conține orice valoare și vor fi utilizate pentru variabilele controller și action. Dacă ultimele două segmente sunt omise, atunci valorile implicite vor fi utilizate.

Putem crea, de asemenea, modele de adrese URL care au segmente care conțin atât elemente statice, cât și variabile, cum ar fi cel prezentat în Exemplul 13.

Exemplul 13. Mixing Segments in the Startup.cs File in the UrlsAndRoutes Folder

```

namespace UrlsAndRoutes {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute("", "X{controller}/{action}");
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}");
                routes.MapRoute(name: "",

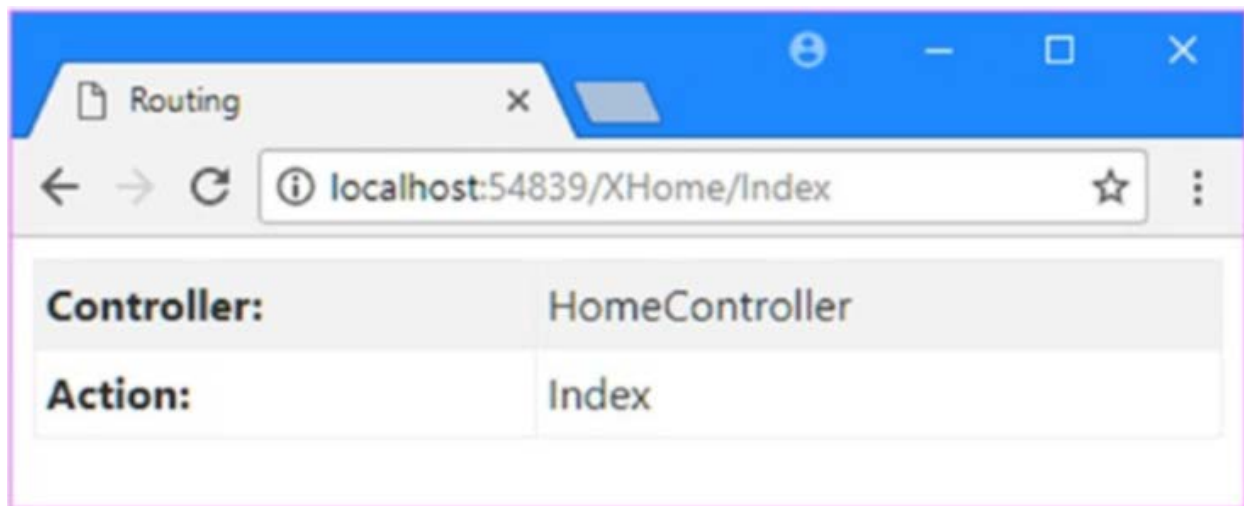
```

```

        template: "Public/{controller=Home}/{action=Index}");
    });
}
}
}

```

Modelul din această rută se potrivește cu orice URL de două segmente în care primul segment începe cu litera X. Valoarea controlerului este preluată din primul segment, excluzând X. Valoarea acțiunii este preluată din al doilea segment. Putem vedea efectul acestei rute dacă pornim aplicația și navigăm la / XHome / Index, al cărui rezultat este ilustrat mai jos.



Ordinea rutelor. În Exemplul 13, am definit o ruta noua și am plasat-o înaintea tuturor celorlalte. Am făcut acest lucru deoarece rutele sunt aplicate în ordinea în care sunt definite. Sistemul de rutare încearcă să potrivească o adresa URL primita cu patternul URL al rutei care a fost definit mai întâi și trece la ruta următoare numai dacă nu există nicio potrivire. Rutele sunt încercate în succesiune până la găsirea unei potriviri sau până la epuizarea setului de rute. În consecință, trebuie definite mai întâi cele mai specifice rute. Să presupunem că am inversat ordinea rutelor, astfel:

```

...
routes.MapRoute("MyRoute", "{controller=Home}/{action=Index}");
routes.MapRoute("", "X{controller}/{action}");
...

```

Prima rută, se potrivește oricărui URL cu zero, unu sau două segmente, va fi întotdeauna cea care este utilizată. Traseul mai specific, care este acum pe locul doi în listă, nu va fi parcurs niciodată.

Definirea variabilelor de segment personalizate. Putem defini si variabile de segment personalizate, așa cum se arată în Exemplul 14.

Exemplul 14. Defining Additional Variables in the Startup.cs File in the UrlsAndRoutes Folder

```
namespace UrlsAndRoutes {  
    public class Startup {  
        public void ConfigureServices(IServiceCollection services) {  
            services.AddMvc();  
        }  
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {  
            app.UseDeveloperExceptionPage();  
            app.UseStatusCodePages();  
  
            app.UseStaticFiles();  
            app.UseMvc(routes => {  
                routes.MapRoute(name: "MyRoute",  
                    template: "{controller=Home}/{action=Index}/{id=DefaultId}");  
            });  
        }  
    }  
}
```

Patternul URL definește variabilele controller și action standard, precum și o variabilă personalizată numită id. Aceasta ruta se va potrivi cu orice adresă URL de la zero la trei segmente. Conținutul celui de-al treilea segment va fi alocat variabilei id, iar dacă nu există un al treilea segment, se va utiliza valoarea implicită.

Clasa Controller, care este baza pentru controlere, definește o proprietate **RouteData** care returnează un obiect **Microsoft.AspNetCore.Routing.RouteData** care oferă detalii despre sistemul de rutare și modul în care cererea curentă a fost dirijată. În cadrul unui controler, pot accesa oricare dintre variabilele de segment dintr-o metodă de acțiune folosind proprietatea **RouteData.Values**, care returnează un dicționar care conține variabilele segmentului. Pentru a demonstra, am adăugat o metodă de acțiune la controlerul de **Home** numită **CustomVariable**, așa cum se arată în Exemplul 15.

Exemplul 15. Accessing a Segment Variable in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;  
using UrlsAndRoutes.Models;  
namespace UrlsAndRoutes.Controllers {  
    public class HomeController : Controller {  
        public IActionResult Index() => View("Result",  
            new Result {  
                Controller = nameof(HomeController),  
                Action = nameof(Index)  
            });  
        public IActionResult CustomVariable() {  
            Result r = new Result {
```

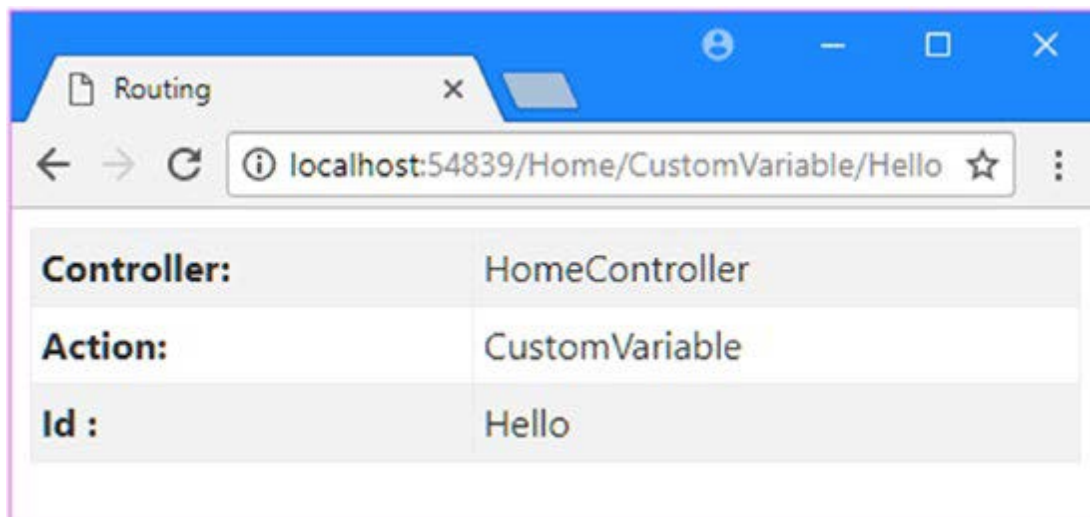
```

        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
    };
    r.Data["Id"] = RouteData.Values["id"];
    return View("Result", r);
}
}
}

```

Această metodă de acțiune obține valoarea variabilei personalizate id în modelul URL-ului din ruta folosind proprietatea `RouteData.Values`, care returnează un dicționar al variabilelor produse de sistemul de rutare. Variabila personalizată este adăugată la obiectul modelului de vizualizare și poate fi văzută rulând aplicația și solicitând următoarea adresă URL:

/Home/CustomVariable/Hello



Utilizarea variabilelor personalizate ca parametri ai metodei de acțiune. Utilizarea colecției `RouteData.Values` este doar o modalitate de a accesa variabilele personalizate ale rutei. Dacă o metodă de acțiune definește parametrii cu nume care se potrivesc cu variabilele din patternul URL, MVC va trece automat valorile obținute de la URL ca argumente la metoda de acțiune.

Variabila personalizată definită în ruta din Exemplul 15 se numește id. Putem modifica metoda de acțiune `CustomVariable` în controlerul `Home`, astfel încât să aibă un parametru cu același nume, așa cum se arată în Exemplul 16.

Exemplul 16. Adding an Action Parameter in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

```

```

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });
        public ActionResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id;
            return View("Result", r);
        }
    }
}

```

Atunci când sistemul de rutare se potrivește cu o adresă URL față de ruta definită în Exemplul 16, valoarea celui de-al treilea segment din URL este atribuită variabilei personalizate id. MVC compară lista variabilelor de segment cu lista parametrilor metodei de acțiune și, dacă numele se potrivesc, trece valorile de la URL la metodă.

Definirea segmentelor URL opționale. Un segment URL opțional este unul pe care utilizatorul nu trebuie să îl specifice și pentru care nu este specificată nicio valoare implicită. Un segment opțional este notat cu un semn de întrebare (caracterul?) După numele segmentului, așa cum se arată în **Exemplul 17**.

Exemplul 17. Specifying an Optional Segment in the Startup.cs File in the UrlsAndRoutes Folder

```

namespace UrlsAndRoutes {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}

```

```
}
```

Aceasta ruta se va potrivi cu adresele URL indiferent dacă segmentul `id` a fost furnizat sau nu.

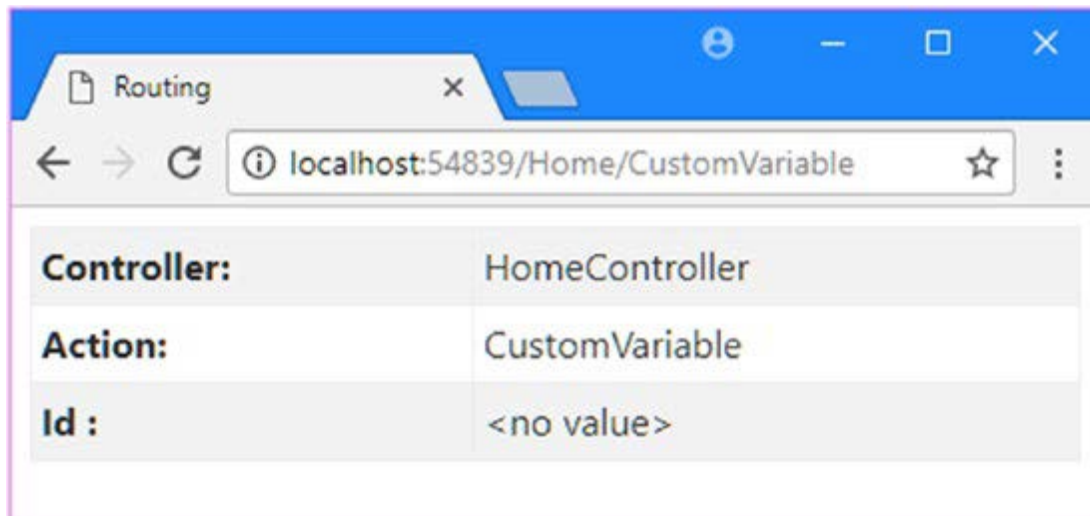
0 /	controller = Home action = Index
1 /Customer	controller = Customer action = Index
2 /Customer/List	controller = Customer action = List
3 /Customer/List/All	controller = Customer action = List id = All
4 /Customer/List/All/Delete	No match—too many segments

După cum puteți vedea din tabel, variabila `id` este adăugată la setul de variabile doar atunci când există un segment corespunzător în adresa URL de intrare. Această caracteristică este utilă dacă trebuie să știți dacă utilizatorul a furnizat o valoare pentru o variabilă de segment. Când nu a fost furnizată nicio valoare pentru o variabilă de segment opțională, valoarea parametrului corespunzător va fi nulă. Am actualizat controlerul `Home` pentru a răspunde atunci când nu este furnizată nicio valoare pentru variabila de segment `ID` în Exemplul 18.

Exemplul 18. Checking for a Segment in the HomeController.cs File in the Controllers Folder
using `Microsoft.AspNetCore.Mvc;`
using `UrlsAndRoutes.Models;`

```
namespace UrlsAndRoutes.Controllers {  
    public class HomeController : Controller {  
        public IActionResult Index() => View("Result",  
            new Result {  
                Controller = nameof(HomeController),  
                Action = nameof(Index)  
            });  
        public IActionResult CustomVariable(string id) {  
            Result r = new Result {  
                Controller = nameof(HomeController),  
                Action = nameof(CustomVariable),  
            };  
            r.Data["Id"] = id ?? "<no value>";  
            return View("Result", r);  
        }  
    }  
}
```

Figura de mai jos arată rezultatul pornirii aplicației și navigării către URL-ul `/ Home / CustomVariable`, care nu include o valoare pentru variabila segmentului `id`.



Definirea rutelor de lungime variabilă. Un alt mod de modificare a conventiei implicite pentru patternurile URL este acceptarea unui număr variabil de segmente URL. Aceasta ne permite să rutăm adresele URL de lungimi arbitrare într-o singură rută. Un segment de lungime variabilă se specifică prin desemnarea uneia dintre variabilele de segment ca un nume prefixat cu un asterisc (caracterul *), așa cum se arată în Exemplul 19.

Exemplul 19. Designating a Catchall Variable in the Startup.cs File in the UrlsAndRoutes Folder using `Microsoft.AspNetCore.Builder`;
using `Microsoft.Extensions.DependencyInjection`;
namespace UrlsAndRoutes {
 public class Startup {

```
        public void ConfigureServices(IServiceCollection services) {  
            services.AddMvc();  
        }  
        public void Configure(IApplicationBuilder app) {  
            app.UseStatusCodePages();  
            app.UseDeveloperExceptionPage();  
            app.UseStaticFiles();  
            app.UseMvc(routes => {  
                routes.MapRoute(name: "MyRoute",  
                                template: "{controller=Home}/{action=Index}/{id?}/{*catchall}");  
            });  
        }  
    }
```

Această rută va corespunde acum oricărei URL, indiferent de numărul de segmente pe care le conține sau de valoarea oricăruia dintre aceste segmente. Primele trei segmente sunt utilizate

pentru a seta valori pentru variabilele de controller, action și, respectiv, id. Dacă adresa URL conține segmente suplimentare, toate sunt alocate variabilei catchall, așa cum se arată mai jos.

0 /	controller = Home action = Index
1 /Customer	controller = Customer action = Index
2 /Customer/List	controller = Customer action = List
3 /Customer/List/All	controller = Customer action = List id = All
4 /Customer/List/All/Delete	controller = Customer action = List id=All catchall=Delete
5 /Customer/List/All/Delete/Perm	controller = Customer action = List id = All catchall = Delete/Perm

În **Exemplul 20**, am actualizat controlerul client, astfel încât acțiunea List să treacă valoarea variabilei catchall la View prin intermediul obiectului model.

Exemplul 20. Updating an Action in the CustomerController.cs File in the Controllers Folder using **Microsoft.AspNetCore.Mvc;**

using UrlsAndRoutes.Models;

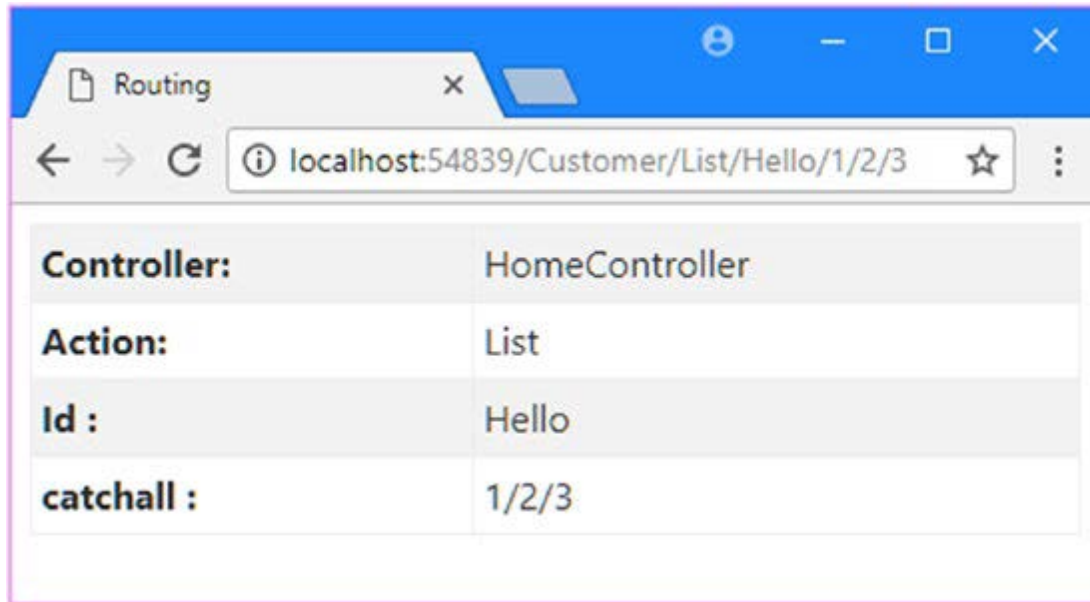
```
namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {
        public IActionResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });
        public IActionResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["Id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

Pentru a testa segmentul catchall, rulați aplicația și solicitați următoarea adresă URL:

/Customer/List/Hello/1/2/3

Nu există o limită superioară la numărul de segmente cu care se va potrivi modelul URL din acest traseu.

Figura de mai jos arată efectul segmentului catchall.



Controller:	HomeController
Action:	List
Id :	Hello
catchall :	1/2/3

Constrângeri de rutare. Acum eln aceasta sectiune vom vedea modul în care, se poate restricționa setul de adrese URL cu care se va potrivi o rută. Exemplul 21 demonstrează utilizarea unei constrângeri simple care limitează adresele URL cu care se va potrivi o rută.

Exemplul 21. Constraining a Route in the Startup.cs File in the UrlsAndRoutes 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;

namespace UrlsAndRoutes {

 public class Startup {

 public void ConfigureServices(IServiceCollection services) {

 services.AddMvc();

 }

 public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

 app.UseDeveloperExceptionPage();

 app.UseStatusCodePages();

 app.UseStaticFiles();

 app.UseMvc(routes => {

 routes.MapRoute(name: "MyRoute",

 template: "{controller=Home}/{action=Index}/{id:int?}");

 });

 }

```
}  
}
```

Constrângerile sunt separate de numele variabilei segment cu două puncte (: caracterul).

Constrângerea din Exemplu este `int` și a fost aplicată pe segmentul `id`. Acesta este un exemplu de constrângere în linie, care este definit ca parte a modelului URL aplicat unui singur segment:

```
...  
template: "{controller}/{action}/{id:int?}",  
...
```

Limitarea `int` permite patternul URL să corespundă segmentelor a căror valoare poate fi analizată la o valoare întreagă. Segmentul ID este opțional, deci ruta se va potrivi cu segmentele care omit segmentul de `id`, dar dacă segmentul este prezent, atunci trebuie să fie o valoare întreagă, așa cum este rezumat mai jos.

<code>/</code>	controller = Home action = Index id = null
<code>/Home/CustomVariable/Hello</code>	No match—id segment cannot be parsed to an int value.
<code>/Home/CustomVariable/1</code>	controller = Home action = CustomVariable id = 1
<code>/Home/CustomVariable/1/2</code>	No match—too many segments

Constrângerile pot fi de asemenea specificate în afara modelului URL, folosind argumentul constrângerilor la metoda `MapRoute` la definirea unei rute.

Exemplul 22. Expressing a Constraint in the Startup.cs File in the `UrlsAndRoutes` 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 Microsoft.AspNetCore.Routing.Constraints;  
namespace UrlsAndRoutes {  
    public class Startup {  
        public void ConfigureServices(IServiceCollection services) {  
            services.AddMvc();  
        }  
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {  
            app.UseDeveloperExceptionPage();  
            app.UseStatusCodePages();  
            app.UseStaticFiles();  
            app.UseMvc(routes => {
```

```

        routes.MapRoute(name: "MyRoute",
            template: "{controller}/{action}/{id?}",
            defaults: new { controller = "Home", action = "Index" },
            constraints: new { id = new IntRouteConstraint() });
    });
}
}
}

```

Argumentul constraints la metoda MapRoute este definit folosind un tip anonim ale cărui nume de proprietate corespund variabilei de segment care este restricționată. Microsoft.AspNetCore.

Routing.Constraints namespace conține un set de clase care pot fi utilizate pentru a defini constrângerile individuale.

Descriem setul complet de clase de constrângere din Microsoft.AspNetCore.Routing.

alpha/ AlphaRouteConstraint() - Caractere din alfabet (A – Z, a – z)

bool/ BoolRouteConstraint - Corespunde unei valori bool

datetime/ DateTimeRouteConstraint() - Se potrivește cu o valoare DateTime

decimal/ DecimalRouteConstraint() - Se potrivește cu o valoare zecimală

double/ DoubleRouteConstraint() - Coordonează o valoare double

float/ FloatRouteConstraint() - Corespunde cu o valoare float

guid/ GuidRouteConstraint() - Asortează o valoare la un identificator unic global

int/ IntRouteConstraint () - Corespunde unei valori int

length(lungime), length(min, max) /LengthRouteConstraint (len), LengthRouteConstraint (min, max) - Corespunde unei valori cu numărul specificat de caractere sau care este între min și max caractere. în lungime (inclusiv)

long/ LongRouteConstraint lung() - Se potrivește cu o valoare long.

maxlength(len)/ MaxLengthRouteConstraint (len) - Corespunde unei șiruri cu cel puțin len caractere

max(val)/ MaxRouteConstraint(val) - Se potrivește cu o valoare int dacă valoarea este mai mică decât val

minlength(len)/MinLengthRouteConstraint(len) - Corespunde unei șiruri cu cel puțin len caractere

min(val)/ MinRouteConstraint(val) - Corespunde cu o valoare int dacă valoarea este mai mare decât val

range(min, max)/RangeRouteConstraint(min, max) - Corespunde unei valori int dacă valoarea este între min și max (inclusiv)

regex(expr)/ RegexRouteConstraint(expr) - Corespunde cu o expresie regulată

Constrângerea unei rute folosind o expresie regulate. Constrângerea care oferă cea mai mare flexibilitate este regex, care se potrivește cu un segment folosind o expresie obișnuită.

În **Exemplul 23**, am restricționat segmentul controlorului pentru a limita gama de adrese URL pe care le va potrivi.

Exemplul 23. Using a Regular Expression in the Startup.cs File in the UrlsAndRoutes Folder

```
namespace UrlsAndRoutes {  
    public class Startup {  
        public void ConfigureServices(IServiceCollection services) {  
            services.AddMvc();  
        }  
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {  
            app.UseDeveloperExceptionPage();  
            app.UseStatusCodePages();  
            app.UseStaticFiles();  
            app.UseMvc(routes => {  
                routes.MapRoute(name: "MyRoute",  
                                template: "{controller:regex(^H.*)=Home}/"  
                                + "{action:regex(^Index$|^About$)=Index}/{id?}");  
            });  
        }  
    }  
}
```

Constrângerea pe care am folosit-o restricționează ruta, astfel încât acesta să corespundă doar adreselor URL unde segmentul controlorului începe cu litera H. Expresiile regulate pot constrânge o ruta, astfel încât doar valori specifice pentru un segment de URL să provoace o potrivire. Aceasta se face folosind caracterul bara (|). Aceasta înseamnă că ruta din Exemplu se va potrivi cu adresele URL doar atunci când variabila controller începe cu litera H și variabila actions este Index sau About.

În continuare vom vedea cum putem utiliza schema de rutare pentru a genera URL-uri de ieșire pe care le putem încorpora în View-uri. Sistemul de rutare oferă suport pentru generarea URL-urilor în View-uri.

Generarea adreselor URL de ieșire în View-uri. În aproape orice aplicație MVC, va trebui să permitem utilizatorului să navigheze dintr-un View în altul, facilitate care, de obicei, se va baza pe includerea unui link în primul View care vizează metoda de acțiune care generează al doilea View.

Putem să utilizăm doar un element static al cărui atribut href adresează metoda de acțiune, astfel:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Presupunând că aplicația folosește configurația de rutare implicită, acest element HTML creează o legătură care va adresa metoda de acțiune CustomVariable din controlerul Home. Adresele URL definite manual ca acestea sunt rapid și simplu de creat dar, sunt greu de modificat atunci când se dorește schimbarea schemei URL pentru o aplicație.

Generarea de legături de ieșire. Cea mai simplă modalitate de a genera o adresă URL de ieșire într-un View este să folosim mecanismul tag helper, care va genera atributul href pentru un HTML, așa cum este ilustrat în Exemplul 24.

Exemplul 24. Using the Anchor Tag Helper in the Result.cshtml File in the Views/Shared Folder

@model Result

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Routing</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
```

```
</head>
```

```
<body class="m-1 p-1">
```

```
    <table class="table table-bordered table-striped table-sm">
```

```
        <tr><th>Controller:</th><td>@Model.Controller</td></tr>
```

```
        <tr><th>Action:</th><td>@Model.Action</td></tr>
```

```
        @foreach (string key in Model.Data.Keys) {
```

```
            <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
```

```
        }
```

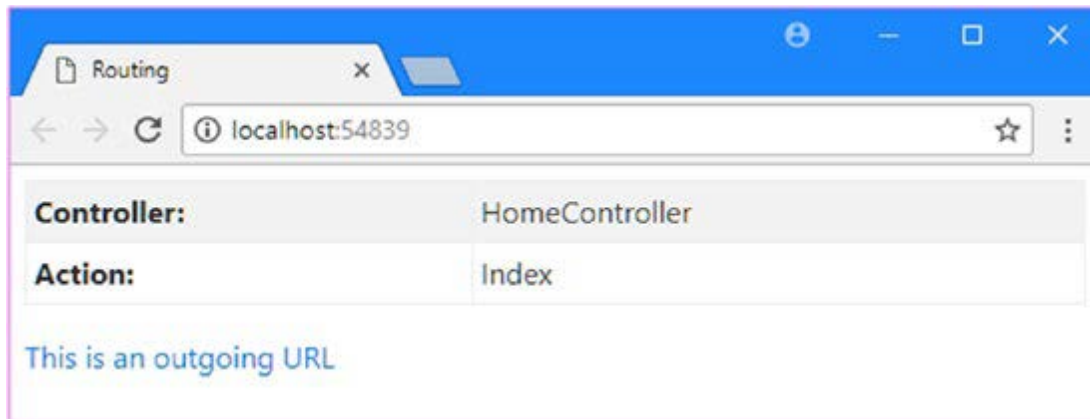
```
    </table>
```

```
    <a asp-action="CustomVariable">This is an outgoing URL</a>
```

```
</body>
```

```
</html>
```

Atributul asp-action este utilizat pentru a specifica numele metodei de acțiune pe care URL-ul din atributul href ar trebui să o țintească. Putem vedea rezultatul pornind aplicația, așa cum se arată mai jos.



Tag helper-ul stabilește atributul href pe un element folosind configurația de rutare curentă. Dacă inspectăm HTML-ul trimis către browser, vom vedea că acesta conține următorul element:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Beneficiul acestei abordări este că răspunde automat la modificările din configurația de rutare.

Dirjecționarea către alt controller. Când specificați atributul asp-action pe un element, asistentul de etichetare presupune că doriți să accesați o acțiune în același controler care a determinat redarea View-ului. Pentru a crea o adresă URL de ieșire care vizează un alt controler, putem utiliza atributul asp-controller, așa cum se arată în **Exemplul 25**.

Exemplul 25. Targeting a Different Controllers in the Result.cshtml File in the Views/Shared Folder

@model Result

@{ Layout = null; }

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Routing</title>

<link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />

</head>

<body class="m-1 p-1">

<table class="table table-bordered table-striped table-sm">

<tr><th>Controller:</th><td>@Model.Controller</td></tr>

<tr><th>Action:</th><td>@Model.Action</td></tr>

@foreach (string key in Model.Data.Keys) {

<tr><th>@key :</th><td>@Model.Data[key]</td></tr>

}

</table>

<a asp-controller="Admin" asp-action="Index">

This targets another controller

</body>

</html>

Când redam View-ul, vom vedea următorul HTML generat:

```
<a href="/Admin">This targets another controller</a>
```

Sistemul de rutare știe că ruta definită în aplicație va utiliza în mod implicit metoda de acțiune Index, permițându-i să omită segmentele implicite.

Generarea de adrese URL complet calificate. Toate legăturile generate până acum conțineau URL-uri relative, dar sistemul de rutare poate genera, de asemenea, adrese URL complet calificate, așa cum se arată în **Exemplul 26**.

Exemplul 26. Generating a Fully Qualified URL in the Result.cshtml File in the Views/Shared Folder

```
@model Result
```

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Routing</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
```

```
</head>
```

```
<body class="m-1 p-1">
```

```
    <table class="table table-bordered table-striped table-sm">
```

```
        <tr><th>Controller:</th><td>@Model.Controller</td></tr>
```

```
        <tr><th>Action:</th><td>@Model.Action</td></tr>
```

```
        @foreach (string key in Model.Data.Keys) {
```

```
            <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
```

```
        }
```

```
    </table>
```

```
    <a asp-controller="Home" asp-action="Index" asp-route-id="Hello"
```

```
        asp-protocol="https" asp-host="myserver.mydomain.com"
```

```
        asp-fragment="myFragment">
```

```
        This is an outgoing URL
```

```
    </a>
```

```
</body>
```

```
</html>
```

Trebuie să avem grijă când utilizăm adrese URL complet calificate, deoarece creează dependențe de infrastructura aplicației și, atunci când infrastructura se schimbă, va trebui să facem modificările corespunzătoare View-urilor MVC.

Generarea de adrese URL (și nu link-uri). Limitarea mecanismului tag helpers constă în faptul că transformă elemente HTML și nu poate fi utilizat cu ușurință dacă trebuie să generăm o adresă URL pentru aplicație fără HTML. MVC oferă o clasă de ajutor care poate fi folosită pentru a crea URL-uri direct, disponibile prin metoda `Url.Action`, așa cum se arată în **Exemplul 27**.

Exemplul 27. Generating a URL in the Result.cshtml File in the Views/Shared Folder

```
@model Result
```

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```



```

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
    <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <p>URL: @Url.Action("CustomVariable", "Home", new { id = 100 })</p>
</body>
</html>

```

Argumentele pentru metoda `Url.Action` specifică metoda de acțiune, controlerul și valorile pentru orice variabile de segment. Rezultatul din acest exemplu generează următoarea ieșire:

```
<p>URL: /Home/CustomVariable/100</p>
```

Generarea URL-urilor în metode de acțiune. Metoda `Url.Action` poate fi folosită și în metodele de acțiune pentru a crea adrese URL în cod C#. În Exemplul 28, am modificat una dintre metodele de acțiune ale controlerului `Home` pentru a genera o adresă URL folosind `Url.Action`.

Exemplul 28. Generating a URL in an Action Method in the `HomeController.cs` File in the `Controllers` Folder

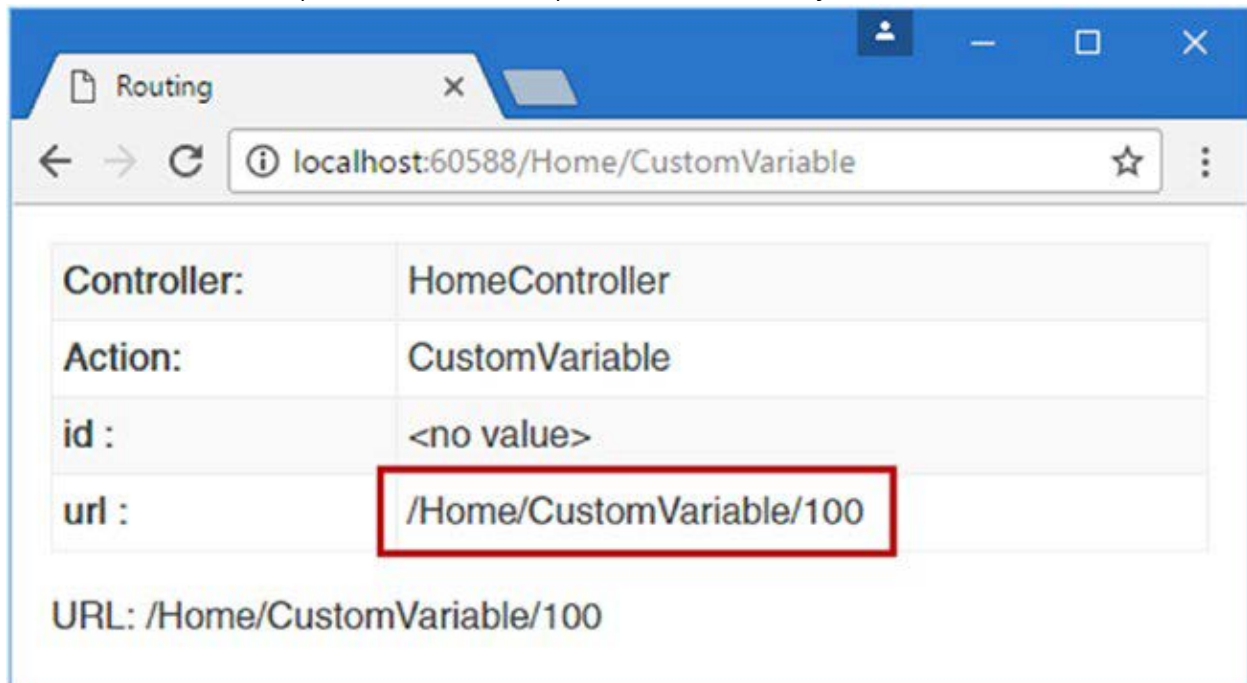
```

using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;
namespace UrlsAndRoutes.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
      });
    public IActionResult CustomVariable(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["Url"] = Url.Action("CustomVariable", "Home", new { id = 100 });
      return View("Result", r);
    }
  }
}

```

}

Dacă executăm exemplul și solicităm URL-ul / Home / CustomVariable, vom vedea că există un rând în tabelul care afișează adresa URL, așa cum se arată mai jos.



Controllers and Actions

Fiecare solicitare este tratată de un controlor. În ASP.NET Core MVC, controlerele sunt clase .NET care conțin logica necesară pentru gestionarea unei solicitări. Rolul controller-ului este să încapsuleze logica aplicației. Aceasta înseamnă că controlerele sunt responsabile de procesarea cererilor primite, de activarea operațiunilor pe modelul de domeniu și de selectarea View-urilor pentru a fi redată utilizatorului. În această secțiune vom vedea cum sunt implementate controlerele și diferitele moduri prin care le putem utiliza pentru a primi și genera ieșiri. Controlerele conțin logica pentru primirea cererilor, actualizarea stării modelului aplicației și selectarea răspunsului care va fi trimis clientului. Controlerele sunt clase C# ale căror metode publice sunt invocate pentru a gestiona o solicitare HTTP.

Crearea proiectului pentru exemplificare. Pentru această secțiune, folosim șablonul ASP.NET Core Web Application (.NET Core) pentru a crea un nou proiect Empty numit ControllersAndActions. În Exemplul 29, am adăugat declarații la clasa Startup pentru a activa cadrul MVC și alte componente de middleware.

Exemplul 29. Adding MVC and Other Middleware in the Startup.cs File the ControllersAndActions 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;
namespace ControllersAndActions {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvcWithDefaultRoute();
        }
    }
}

```

Metodele AddMemoryCache și AddSession creează servicii care sunt necesare pentru gestionarea sesiunilor. Metoda UseSession adaugă o componentă middleware pipeline-ul care asociază datele sesiunii cu solicitări și adaugă cookie-uri la răspunsuri pentru a vă asigura că viitoarele solicitări pot fi identificate. Metoda UseSession trebuie apelată înainte de metoda UseMvc, astfel încât componenta de sesiune să poată intercepta cererile înainte de a ajunge la middleware MVC și să poată modifica răspunsurile după ce au fost generate. Celelalte metode stabilesc pachetele standard necesare.

Pregătirea View-urilor. Pentru a pregăti proiectul, vom defini câteva View-uri care ne vor ajuta să demonstrăm modul în care funcționează controlerele. Vom crea aceste View-uri în folderul Views/Shared, astfel încât să le putem utiliza de la oricare dintre controlerele pe care le vom crea.

Adaugăm în folderul Views/Shared, un fișier View Razor numit Result.cshtml și îl modificăm ca în Exemplul 30.

Exemplul 30. The Contents of the Result.cshtml File in the Views/Shared Folder

```

@model string
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    Model Data: @Model

```

```
</body>
```

```
</html>
```

Modelul pentru acest View este un string, care ne va permite să afișăm mesaje simple. În continuare, am creat un View numit DictionaryResult.cshtml, tot în folderul Views/ Shared, și am adăugat conținutul din Exemplul 31. Modelul pentru acest View este un dicționar, care afișează date mai complexe.

Exemplul 31. The Contents of the DictionaryResult.cshtml File in the Views/Shared Folder

```
@model IDictionary<string, string>
```

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Controllers and Actions</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
```

```
</head>
```

```
<body class="m-1 p-1">
```

```
    <table class="table table-bordered table-sm table-striped">
```

```
        <tr><th>Name</th><th>Value</th></tr>
```

```
        @foreach (string key in Model.Keys) {
```

```
            <tr><td>@key</td><td>@Model[key]</td></tr>
```

```
        }
```

```
    </table>
```

```
</body>
```

```
</html>
```

În continuare, am creat un View numit SimpleForm.cshtml, tot în folderul Views/Shared, care arată ca în Exemplul 32. După cum sugerează și numele său, această vizualizare conține un formular HTML simplu care va colecta date de la utilizator.

Exemplul 32. The Contents of the SimpleForm.cshtml File in the Views/Shared Folder

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Controllers and Actions</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
```

```
</head>
```

```
<body class="m-1 p-1">
```

```
    <form method="post" asp-action="ReceiveForm">
```

```
        <div class="form-group">
```

```
            <label for="name">Name:</label>
```

```
            <input class="form-control" name="name" />
```

```
        </div>
```

```
    </div class="form-group">
```

```

        <label for="name">City:</label>
        <input class="form-control" name="city" />
    </div>
    <button class="btn btn-primary center-block" type="submit">Submit</button>
</form>
</body>
</html>

```

Vizualizările folosesc ajutoarele de etichetă încorporate pentru a genera URL-uri din sistemul de rutare. Pentru a activa tag helpers, am creat un fișier de import pentru View-uri numit `_ViewImports.cshtml` în folderul Views și am adăugat expresia declarată din Exemplul 33.

Exemplul 33. The Contents of the `_ViewImports.cshtml` File in the Views Folder

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

Vizualizările pe care le-am creat în folderul Views / Shared depind de pachetul Bootstrap CSS. Pentru a adăuga Bootstrap la proiect, am folosit șablonul Bower Configuration File pentru a crea fișierul `bower.json` și am adăugat pachetul prezentat în **Exemplul 34**.

Exemplul 34. Adding a Package in the `bower.json` File in the ControllersAndActions Folder

```

{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}

```

Controlerele sunt clase C# ale căror metode publice (cunoscute sub denumirea de acțiuni sau metode de acțiune) sunt responsabile pentru gestionarea unei cereri HTTP și pregătirea răspunsului care va fi returnat clientului. MVC folosește sistemul de rutare pentru a identifica clasa de control și metoda de acțiune de care are nevoie pentru a gestiona o solicitare. MVC creează o nouă instanță a clasei de controler, invocă metoda de acțiune și folosește rezultatul metodei pentru a produce răspunsul către client.

MVC oferă metode de acțiune cu date de context. Când MVC invocă o metodă de acțiune, răspunsul metodei descrie răspunsul care trebuie trimis clientului. Cel mai obișnuit tip de răspuns este creat prin redarea unei View Razor, astfel încât metoda de acțiune își folosește răspunsul pentru a spune MVC ce View trebuie să folosească și ce View a modelelor de date ar trebui furnizate. Dar există și alte tipuri de răspunsuri disponibile, iar metodele de acțiune pot face totul, pentru a solicita MVC să trimită o redirecționare HTTP către client până la trimiterea de obiecte de date complexe.

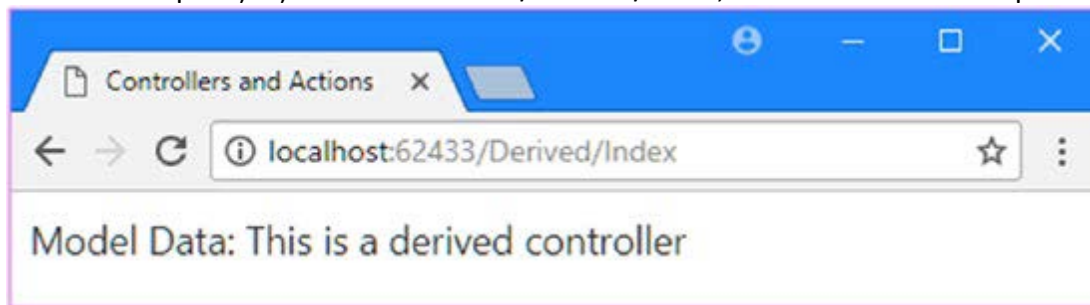
Crearea controlerelor. Vom vedea, în continuare, diferitele moduri prin care pot fi create controlerele.

O modalitate naturală de a crea controlere este de a deriva clase din clasa `Microsoft.AspNetCore.Mvc.Controller`, care definește metodele și proprietățile care oferă acces la caracteristicile MVC într-o manieră concisă și simplă. Pentru a demonstra, vom adăuga un fișier Controller numit `DerivedController.cs` în folderul Controllers și definim controllerul prezentat în Exemplul 34.

Exemplul 34. Deriving from the Controller Class in the DerivedController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
namespace ControllersAndActions.Controllers {
    public class DerivedController : Controller {
        public IActionResult Index() =>
            View("Result", $"This is a derived controller");
    }
}
```

Dacă rulăm aplicația și solicităm URL-ul /Derived/Index, vom vedea rezultatul prezentat mai jos.



Metoda View este moștenită de la clasa de bază a Controller.

Primirea datelor de context. Controllele trebuie să acceseze date din cererea primită, cum ar fi valorile șirului de interogare, valorile formelor și parametrii URL din rute, cunoscute ca date de context. Există trei moduri principale de accesare a datelor de context.

- Extragere date din obiecte de context
- Primirea datelor ca parametru pentru o metodă de acțiune
- Invocarea explicită a facilității de legare a modelului.

Obținerea de date din obiecte de context.

Unul dintre avantajele principale ale utilizării clasei de bază Controllers pentru a crea controlere este accesul convenabil la un set de obiecte de context care descriu cererea curentă, răspunsul care este pregătit și starea aplicației. Proprietățile de context ale controlorului sunt:

Request - Această proprietate returnează un obiect HttpRequest care descrie solicitarea primită de la client.

Response - Această proprietate returnează un obiect HttpResponse care este utilizat pentru a crea răspunsul pentru client.

HttpContext - Această proprietate returnează un obiect HttpContext, care este sursa multor obiecte returnate de alte proprietăți, cum ar fi Request și Response. De asemenea, oferă informații despre caracteristicile HTTP disponibile.

RouteData - Această proprietate returnează obiectul RouteData produs de sistemul de rutare atunci când a corespuns cererii.

ModelState - Această proprietate returnează un obiect ModelStateDictionary, care este utilizat pentru validarea datelor trimise de client.

User - Această proprietate returnează un obiect ClaimsPrincipal care descrie utilizatorul care a făcut solicitarea.

Proprietățile HttpRequest care sunt cele mai utile la scrierea controlerelor sunt:

Path - Această proprietate returnează secțiunea de cale a adresei URL a solicitării.

QueryString - Această proprietate returnează secțiunea șirului de interogare a adresei URL a solicitării.

Headers - Această proprietate returnează un dicționar al antetelor cererii, indexat după nume.

Body - Aceasta proprietate returnează un flux care poate fi utilizat pentru a citi corpul cererii.

Form - Această proprietate returnează un dicționar al datelor formularului din cerere, indexat după nume.

Cookie - Această proprietate returnează un dicționar al cookie-urilor solicitate, indexat după nume.

Valorile datelor din formular sunt accesate prin proprietatea Request.Form a clasei Controller. Pentru a demonstra acest lucru, adăugăm clasa HomeController.cs. prezentată în Exemplul 35.

Exemplul 35. The Contents of the HomeController.cs File in the Controllers Folder

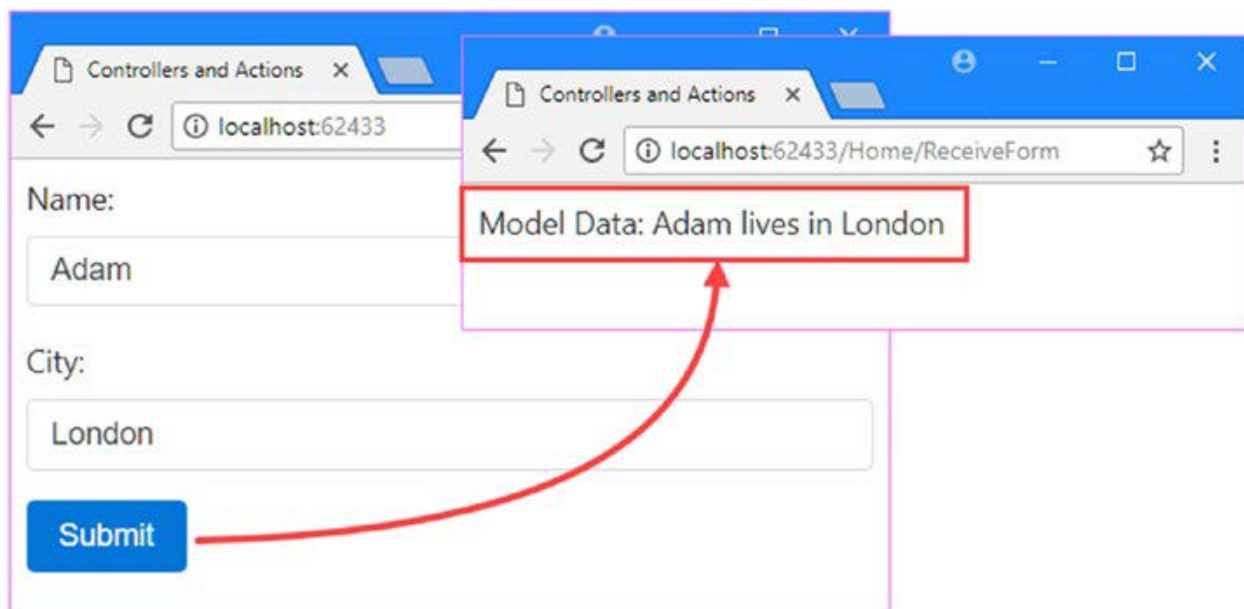
using Microsoft.AspNetCore.Mvc;

```
namespace ControllersAndActions.Controllers {  
    public class HomeController : Controller {  
        public IActionResult Index() => View("SimpleForm");  
        public IActionResult ReceiveForm() {  
            var name = Request.Form["name"];  
            var city = Request.Form["city"];  
            return View("Result", $"{name} lives in {city}");  
        }  
    }  
}
```

Metoda de acțiune Index din acest controller redă View-ul SimpleForm pe care l-am creat în folderul Views/Shared. Este interesantă metoda ReceiveForm, deoarece folosește obiectul de context HttpRequest pentru a obține valori ale formularului din cerere.

După cum este descris în tabelul 17-4, proprietatea Form definită de clasa HttpRequest returnează o colecție care conține valorile datelor de formular, indexate cu numele elementului HTML asociat. Există două elemente de intrare în View-ul SimpleForm (name și city) și le extragem valorile din obiectul contextual și le transmitem View-ului Result ca model.

Dacă rulăm aplicația și solicităm adresa URL / Home, vom vedea un formular. Dacă completăm câmpurile și facem clic pe butonul Submit, browserul va trimite datele formularului ca parte a unei solicitări POST HTTP care va fi gestionată prin metoda ReceiveForm, producând rezultatul prezentat mai jos.



Această abordare prezentată în Exemplul 35 funcționează perfect, dar există o alternativă mai elegantă. Metodele de acțiune pot defini parametrii care sunt folosiți de MVC pentru a transmite datele de context la un controler, inclusiv detalii despre solicitarea HTTP. Aceasta este mai bună decât extragerea directă a obiectelor din context și produce metode de acțiune mai ușor de citit. Pentru a primi datele formularului, declarăm parametrii pentru metoda de acțiune cu nume corespund valorilor datelor din formular, așa cum se arată în Exemplul 36.

Exemplul 36. Receiving Context Data as Parameters in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
namespace ControllersAndActions.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() => View("SimpleForm");
        public IActionResult ReceiveForm(string name, string city)
            => View("Result", $"{name} lives in {city}");
    }
}
```

Metoda de acțiune revizuită produce același rezultat, dar codul este mai ușor de citit și de înțeles.

Producerea unui răspuns. După ce o metodă de acțiune a terminat procesarea unei cereri, trebuie să genereze un răspuns. Există multe caracteristici disponibile pentru generarea de ieșire din metodele de acțiune.

Nivelul cel mai scăzut de a genera răspuns este de a folosi obiectul context `HttpResponse`, care este modul în care ASP.NET Core oferă acces la răspunsul HTTP care va fi trimis clientului.

Proprietățile oferite de clasa `HttpResponse`, care este definită în spațiul de nume `Microsoft.AspNetCore.Http` sunt:

StatusCode - Această proprietate este utilizată pentru a seta codul de stare HTTP pentru răspuns.

ContentType - Această proprietate este utilizată pentru a seta antetul Content-Type al răspunsului.

Headers - Această proprietate returnează un dicționar al antetelor HTTP care vor fi incluse în răspuns.

Cookies - Această proprietate returnează o colecție folosită pentru a adăuga cookie-uri la răspuns.

Body - Această proprietate returnează un obiect System.IO.Stream care este utilizat pentru a scrie datele corpului pentru răspuns.

În lista 17-14, am actualizat controlerul principal, astfel încât acțiunea sa `ReceiveForm` generează un răspuns folosind obiectul `HttpResponse` returnat de proprietatea `Controller.Request`.

În Exemplul 37 controlerul `Home`, prin acțiunea sa `ReceiveForm` generează un răspuns folosind obiectul `HttpResponse` returnat de proprietatea `Controller.Request`.

Exemplul 37. Producing a Response in the `HomeController.cs` File in the `Controllers` Folder

using Microsoft.AspNetCore.Mvc;

using System.Text;

namespace ControllersAndActions.Controllers {

public class HomeController : Controller {

public IActionResult Index() => View("SimpleForm");

public void ReceiveForm(string name, string city) {

Response.StatusCode = 200;

Response.ContentType = "text/html";

byte[] content = Encoding.ASCII

.GetBytes(\$"<html><body>{name} lives in {city}</body>");

Response.Body.WriteAsync(content, 0, content.Length);

}

}

}

using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc {

public interface IActionResult {

Task ExecuteResultAsync(ActionContext context);

}

}

Această interfață poate părea simplă, dar aceasta se datorează faptului că MVC nu dictează ce tipuri de răspuns poate produce un rezultat al acțiunii. Când o metodă de acțiune returnează un rezultat al acțiunii, MVC apelează la metoda `ExecuteResultAsync`, care este responsabilă cu generarea răspunsului în numele metodei de acțiune. Argumentul `ActionContext` oferă date de context pentru generarea răspunsului, inclusiv obiectul `HttpResponse`.

Metoda **`ExecuteResultAsync`** returnează un `Task` asincron;

Acest lucru se potrivește cu clasa `CustomHtmlResult`, care returnează o metodă `Task` pe care o pot folosi ca rezultat al acțiunii. În Exemplul 38, am aplicat clasa **`CustomHtmlResult`** rezultatului acțiunii **`ReceiveForm`**, simplificând metoda de acțiune.

Exemplul 38. Using an Action Result in the HomeController.cs File in the Controllers Folder
using Microsoft.AspNetCore.Mvc;

using System.Text;

using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

public class HomeController : Controller {

public IActionResult Index() => View("SimpleForm");

public IActionResult ReceiveForm(string name, string city)

=> new CustomHtmlResult {

Content = \$"{name} lives in {city}"

};

}

}

Codul care trimite răspunsul este acum definit separat de datele pe care le conține răspunsul, ceea ce simplifică metoda de acțiune și permite producerea aceluiași tip de răspuns în alte metode de acțiune fără a dubla același cod.

Clasa Controller oferă mai multe versiuni diferite ale metodei View, care permit selectarea vizualizării care va fi redată și furnizată cu date de model.

View() - Această versiune creează un obiect ViewResult pentru vizualizarea implicită asociată metodei de acțiune, astfel încât apelarea View() într-o metodă numită MyAction va face o vizualizare numită MyAction.cshtml. Nu sunt utilizate date de model.

View(view) - Această versiune creează un ViewResult care va reda vizualizarea specificată, astfel încât apelarea View ("MyView") va face o vizualizare numită MyView.cshtml. Nu sunt utilizate date de model.

View(model) - Această versiune creează un obiect ViewResult pentru vizualizarea implicită asociată metodei de acțiune și folosește obiectul specificat ca date de model.

View(view, model) - Această versiune creează un obiect ViewResult pentru vizualizarea specificată și folosește obiectul specificat ca date de model.

Când MVC apelează la metoda ExecuteResultAsync a obiectului ViewResult, va începe căutarea View-ului specificat. În mod implicit, MVC va căuta View-ul în următoarele locații:

/Views/<ControllerName>/<ViewName>.cshtml

/Views/Shared/<ViewName>.cshtml

Căutarea începe cu folderul dedicat controlerului curent, astfel încât folderul pentru clasa HomeController este Views/Home.

Dacă numele View-ului nu este specificat în obiectul ViewResult, va fi utilizată valoarea variabilei de acțiune din datele de rutare. Dacă controlorul face parte dintr-o zonă (Areas), atunci locațiile de căutare sunt:

/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml

/Areas/<AreaName>/Views/Shared/<ViewName>.cshtml

/Views/Shared/<ViewName>.cshtml

Imediat ce localizează o potrivire, utilizează acel View pentru a reda rezultatul metodei de acțiune.

Dacă dorim să redăm un View specific, putem face acest lucru oferind o cale explicită ca în exemplul de mai jos.

```

using Microsoft.AspNetCore.Mvc;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() {
            return View("/Views/Admin/Index");
        }
    }
}

```

Când specificăm un View în acest fel, calea trebuie să înceapă cu / sau ~/ și include extensia de nume de fișier (care se presupune a fi .cshtml dacă nu este specificată).

Transferul datelor de la acțiune la View. MVC oferă diferite modalități pentru transferul datelor de la o metodă de acțiune la un View. Descriem în continuare câteva :

Utilizarea unui obiect model. Putem trimite un obiect model, trecându-l ca parametru la metoda View, care are ca efect setarea proprietății ViewData.Model a obiectului IActionResult care este creat. Pentru a avea această facilitare trebuie să activăm în clasa Startup următoarele metode:

Metoda **AddMemoryCache** configurează stocarea de date din memorie;

Metoda **AddSession** înregistrează serviciile utilizate pentru a accesa datele sesiunii;

Metoda **UseSession** permite sistemului de sesiune să asocieze automat cererile cu sesiunile atunci când acestea ajung de la client.

```

public class Startup {
    public void ConfigureServices(IServiceCollection services) {
        services.AddMvc();
        services.AddMemoryCache();
        services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        app.UseStatusCodePages();
        app.UseDeveloperExceptionPage();
        app.UseStaticFiles();
        app.UseSession();
        app.UseMvcWithDefaultRoute();
    }
}

```

Exemplul 39 arată o nouă clasă ExampleController pe care am adăugat-o în folderul Controllers și care trece un obiect model de la acțiune la View prin metoda View.

Exemplul 39. The Contents of the ExampleController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() => View(DateTime.Now);
    }
}

```

```

    }
}

```

Pentru a accesa obiectul din interiorul View-ului, folosim cuvântul cheie Razor Model. Am creat folderul View/ Example și am adăugat un View numit Index.cshtml, care este prezentat în Exemplul 40.

Exemplul 40. The Contents of the Index.cshtml File in the Views/Example Folder

```

@model DateTime
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Controllers and Actions</title>
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    Model: @Model.DayOfWeek
</body>
</html>

```

Am specificat tipul de model de vizualizare folosind cuvântul cheie model Razor. Observați că folosim o minusculă m atunci când specific tipul modelului și o majusculă M la citirea valorii.

Transferul datelor cu View Bag. Această caracteristică ne permite să definim proprietățile unui obiect dinamic într-un controller și să le accesăm într-un View.

Obiectul dinamic este accesat prin proprietatea ViewBag furnizată de clasa Controller, așa cum este demonstrat în Exemplul 41.

Exemplul 41. Using the View Bag Feature in the ExampleController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public IActionResult Result() => View((object)"Hello World");
    }
}

```

Am definit proprietățile ViewBag numite Message și Date, alocând valori acestora. Înainte de acest moment, nu existau aceste proprietăți și nu am făcut nicio pregătire pentru crearea lor. Pentru a citi datele în View avem acces aceleași proprietăți pe care le-am setat în metoda de acțiune, așa cum se arată în Exemplul 42.

Exemplul 42. Reading Data from the ViewBag in the Index.cshtml File in the Views/Example Folder

```
@model DateTime
```

```
@{ Layout = null; }
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Controllers and Actions</title>
```

```
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
```

```
</head>
```

```
<body class="m-1 p-1">
```

```
    <p>The day is: @ViewBag.Date.DayOfWeek</p>
```

```
    <p>The message is: @ViewBag.Message</p>
```

```
</body>
```

```
</html>
```

ViewBag are un avantaj față de utilizarea unui obiect model prin faptul că permite trimiterea mai multor obiecte în View. Atenție, Visual Studio nu poate face validări pentru obiecte dinamice, inclusiv ViewBag, iar erorile nu vor fi descoperite decât la redarea View-ului.

Performing Redirections

Efectuarea redirectarilor. Un rezultat obișnuit dintr-o acțiune este redirecționarea clientului către altă adresă URL. De cele mai multe ori, această adresă URL este o altă acțiune care generează rezultatele pe care dorim să le vadă utilizatorul.

Mai multe rezultate ale acțiunii pot fi utilizate pentru a efectua o redirecționare și anume:

RedirectResult produs de **Redirect**, **RedirectPermanent**. Acest rezultat al acțiunii trimite un răspuns care redirecționează clientul către o nouă adresă URL;

LocalRedirectResult produs de **LocalRedirect**, **LocalRedirectPermanent**. Acest rezultat al acțiunii redirecționează clientul către o adresă URL locală;

RedirectToActionResult produs de **RedirectToAction**, **RedirectToActionResultPermanent**. Acest rezultat al acțiunii redirecționează clientul către o acțiune specifică și un controller;

RedirectToRouteResult produs de **RedirectToRoute**, **RedirectToRoutePermanent**. Acest rezultat al acțiunii redirecționează clientul către o adresă URL generată dintr-o anumită rută.

Diferența dintre **Redirect** și **RedirectPermanent** constă în codul de stare HTTP. Când efectuăm o redirecționare, trimitem unul dintre cele două coduri HTTP către browser.

Cod HTTP 302, care este o redirecționare temporară. Acesta este cel mai frecvent tip de redirecționare.

Cod HTTP 301, care indică o redirecționare permanentă. Acest lucru ar trebui utilizat cu precauție, deoarece recomandă destinatarului codului HTTP să nu solicite niciodată URL-ul original și să utilizeze noua adresă URL inclusă alături de codul de redirecționare.

Redirecționarea la un URL. Redirecționare de bază a unui browser este cea prin metoda **Redirect** oferită de clasa Controller, care returnează o instanță a clasei **RedirectResult**, așa cum se arată în Exemplul 42.

Exemplul 42. Redirecting to a Literal URL in the ExampleController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public IActionResult Result() => View((object)"Hello World");
        public IActionResult Redirect() => Redirect("/Example/Index");
    }
}
```

URL-ul de redirectare este exprimat ca un argument string la metoda Redirect, care produce o redirectionare temporară. Puteți efectua o redirectionare permanentă folosind metoda RedirectPermanent, așa cum se arată în Exemplul 43.

Exemplul 43. Permanently Redirecting in the ExampleController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public IActionResult Result() => View((object)"Hello World");
        public IActionResult Redirect() => RedirectPermanent("/Example/Index");
    }
}
```

Redirectarea utilizand sistemul de rutare. Putem utiliza sistemul de rutare pentru a genera URL-uri valide cu metoda RedirectToRoute, care creează o instanță a clasei RedirectToRouteResult așa cum se poate vedea în Exemplul 44.

Exemplul 44. Redirecting to a Routing URL in the ExampleController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public IActionResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
```

```

        return View();
    }
    public ViewResult Result() => View((object)"Hello World");
    public RedirectToRouteResult Redirect() =>
        RedirectToRoute(new { controller = "Example",
                               action = "Index",
                               ID = "MyID" });
    }
}

```

Metoda RedirectToRoute emite o redirectare temporară. Utilizăm metoda RedirectToRoutePermanent pentru redirectionări permanente. Ambele metode transmit rezultatul sistemului de rutare pentru a genera o adresă URL.

Redirecționarea către o metodă de acțiune. Putem redirecționa controlul către o metodă de acțiune mai elegant folosind metoda RedirectToAction (pentru redirecții temporare) sau metoda RedirectToActionPermanent (pentru redirectionări permanente).

Acestea sunt doar ambalaje pentru metoda RedirectToRoute, care ne permit să specificăm valori pentru metoda de acțiune și controler, fără a fi necesar să cream un tip anonim, așa cum se arată în Exemplul 45.

Exemplul 45. Using the RedirectToAction Method in the ExampleController.cs File in the Controllers

Folder

```

using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
    public class ExampleController : Controller {
        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public RedirectToActionResult Redirect() => RedirectToAction(nameof(Index));
    }
}

```

Dacă specificăm doar o metodă de acțiune, atunci se presupune că ne referim la o metodă de acțiune din controlerul curent. Dacă dorim să redirecționăm către un alt controler, trebuie să furnizăm și numele controlorului ca parametru, astfel:

```

...
public RedirectToActionResult Redirect()
=> RedirectToAction(nameof(HomeController), nameof(HomeController.Index));
...

```

Există și alte versiuni supraîncărcate pe care le putem utiliza pentru a oferi valori suplimentare pentru generarea URL-ului.

Transfer date prin Temp Data. O redirecție determină browserul să trimită o solicitare HTTP complet nouă, ceea ce înseamnă că nu există acces la datele formularului din solicitarea inițială.

ASP.NET oferă o serie de moduri diferite de a stoca starea de sesiune, inclusiv stocarea stării în memorie. Aceasta are avantajul simplității, dar înseamnă că datele sesiunii se pierd atunci când aplicația este oprită sau repornită.

Putem utiliza facilitatea de date tempore pentru a păstra datele de la o solicitare la alta, disponibilă printr-o proprietate a clasei Controller numită TempData așa cum se poate vedea în Exemplul 46.

Exemplul 46. Using Temp Data in the HomeController.cs File in the Controllers Folder

using Microsoft.AspNetCore.Mvc;

using System.Text;

using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

public class HomeController : Controller {

public IActionResult Index() {

return View("SimpleForm");

}

[HttpPost]

public IActionResult ReceiveForm(string name, string city) {

TempData["name"] = name;

TempData["city"] = city;

return RedirectToAction(nameof(Data));

}

public IActionResult Data() {

string name = TempData["name"] as string;

string city = TempData["city"] as string;

return View("Result", \$"{name} lives in {city}");

}

}

}

Acțiunea ReceiveForm utilizează proprietatea TempData, (dicționar), pentru a stoca numele orașul înainte de a redirecționa clientul către acțiunea Data. Metoda Data folosește aceeași proprietate TempData pentru a prelua valorile datelor și le folosește pentru a crea datele model care vor fi afișate de vizualizare.

Returnarea diferitelor tipuri de conținut. HTML nu este singurul tip de răspuns pe care îl pot genera metodele de acțiune. În afara de HTML acțiunile mai pot genera ca răspuns diferite tipuri de date:

JsonResult – Generat de metoda Json

Acest rezultat al acțiunii serializează un obiect în JSON și îl returnează clientului.

ContentResult – Generat de metoda Content

Acest rezultat al acțiunii trimite un răspuns al cărui corp conține un obiect specificat.

ObjectResult - nu este disponibilă o metoda

Acest rezultat al acțiunii va folosi negocierea conținutului pentru a trimite un obiect clientului.

OkObjectResult – Generat de metoda Ok

Acest rezultat al acțiunii va folosi negocierea conținutului pentru a trimite un obiect clientului cu un cod de stare HTTP 200 dacă negocierea conținutului are succes.

NotFoundObjectResult – Generat de metoda NotFound

Acest rezultat al acțiunii va utiliza negocierea conținutului pentru a trimite un obiect clientului cu un cod de stare HTTP 404 dacă negocierea conținutului are succes.

Exercitii.

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