

Table of Contents

Table of Contents	1
ASP.NET MVC – Introducere - (C#)	4
MVC aplicat la framework-ul Web	4
Istoric MVC	5
Exemplificari caracteristici MVC	6
Motor vizualizare Razor	6
Rezolvare dependente	7
Construire componente slab cuplate – Dependency Injection	8
Aplicare Dependency Injection in ASP.NET MVC	10
Structura unei aplicatii ASP.NET MVC	13
ASP.NET MVC si Conventii	14
Procesul de executie al unei aplicatii MVC (C#)	14
Etapale executiei unui proiect MVC Web sunt:	15
Modele, View-uri si Controller-e (C#)	15
Rutarea	15
Controllers	16
Views	17
RenderBody()	20
RenderPage()	20
RenderSection()	20
Creare Rute (C#)	20
ASP.NET MVC Controller (C#)	39
Rezultatul actiunii – mostenite din clasa ActionResult	40
Creare “Action” in cadrul unui Controller (C#)	42
Atributul [NonAction]	43
ASP.NET MVC View (C#)	44
Ce sunt Views?	44
Adaugare continut la View	44
“HTML Helpers” - generare continut View	45
Pasare informatii la View	47
Adaugare continut dinamic la View	49
Utilizare sectiuni	50

Teste pentru sectiuni	53
Redare optionala a sectiunilor.....	53
Utilizare vizualizari partiale – Html.Partial()	55
Utilizare actiuni descendente (Child Actions)	57
Metode Helper	59
Creare metode helper personalizate	60
Proprietati utile din HtmlHelper	63
Gestionarea codficarii stringurilor intr-o metoda helper	64
Custom helper	65
Creare elemente de tip Form - exmplu	67
Specificare ruta folosita de form.....	72
Generare element de intrare dintr-o proprietate a modelului.....	75
Utilizare helper pentru intrare, puternic tipizati.....	76
Creare elemente ce contin liste de valori	76
Metode helper – sabloane	78
Utilizare metode helper.....	79
Metode helper - sabloane	80
Utilizare metadata model	82
Excludere proprietate din scaffolding	83
ASP 5 - MVC 6.....	84
Anatomia aplicatiei	84
Servicii	85
Middleware	85
Servere	86
Webroot.....	86
Configurare	86
Pattern Options.....	87
Controller	91
Creare proiect: nume ASPMVC6Demo.....	91
Metode helper in ASP 5.....	96
Ce sunt Tag Helpers?	96
Ce furnizeaza Tag Helpers?	96
Domeniul de vizibilitate – Tag Helpers	97
Suport IntelliSense pentru Tag Helpers	99

MVC 6 – Creare Tag Helpers personalizati.....	100
---	-----

ASP.NET MVC – Introducere - (C#)

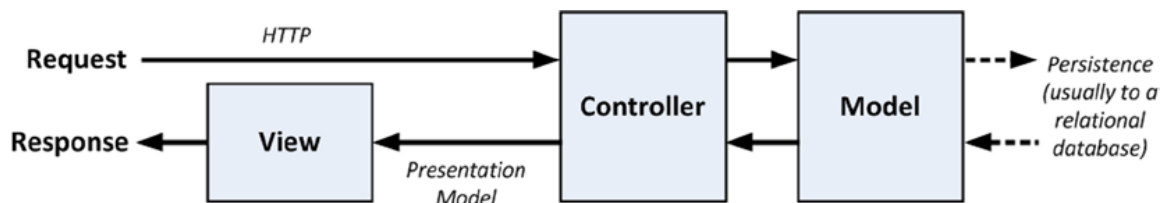
Pattern-ul Model-View-Controller (MVC) separa o aplicatie in trei componente principale:

- ✓ **M: model (Model).**
- ✓ **V: vizualizare (View User Interface).**
- ✓ **C: controler (Controller).**

Observatie

Ceea ce am definit mai sus constituie pattern-ul MVC pentru UI – *User Interface*. Pattern-ul MVC nu spune nimic despre modul cum accesam datele, cum interactioneaza serviciile, etc.

Interactiunea intr-o aplicatie ASP.NET MVC poate fi reprezentata astfel:



MVC aplicat la framework-ul Web

In ASP.NET MVC, MVC este translatat astfel:

- ✓ **Model.** *Model* contine clase ce reprezinta domeniul aplicatiei. Aceste obiecte incapsuleaza adesea date memorate intr-o baza de date precum si cod folosit pentru a procesa datele si a executa actiuni specifice logicii aplicatiei. Cu ASP.NET MVC, acesta este vazut mai ales ca un *Data Acces Layer – DAL* – de un anumit tip, utilizand de exemplu Entity Framework sau NHibernate combinat cu cod specific logicii aplicatiei.
- ✓ **View.** *View* defineste cum va arata interfata aplicatiei. View este un template pentru a genera in mod dinamic HTML.
- ✓ **Controller.** *Controller* este o clasa speciala ce gestioneaza relatiile dintre *View* si *Model*. Controller-ul raspunde la actiunile utilizatorului, comunica cu modelul si decide ce vizualizare va afisa (daca exista una). In ASP.NET MVC, numele acestei clase contine sufixul *Controller*.

Istoric MVC

ASP.NET MVC 1 – 13 Martie 2009

- ✓ Primele concepte de rutare si implementare MVC.
- ✓ Conventii de realizare a aplicatiilor.
- ✓ Structurarea directoarelor, etc.

ASP.NET MVC 2 – Martie 2010

Trasaturi principale incluse:

- ✓ Model de validare bazat pe attribute.
- ✓ Template-uri personalizabile pentru generare UI.
- ✓ Tools-uri noi in VS IDE.
- ✓ Suport pentru partitionarea aplicatiilor mari in “area”.
- ✓ Suport pentru controller-i asincroni.
- ✓ Noi functii ajutatoare, utilitare si API.

ASP.NET MVC 3 – Ianuarie 2011

Trasaturi noi:

- ✓ Motorul de vizualizare Razor.
- ✓ Suport pentru .NET 4 Data Annotations.
- ✓ Imbunatatire model de validare.
- ✓ Control mai mare si flexibilitate pentru rezolvarea dependentelor, filtre globale.
- ✓ Suport mai bun pentru JavaScript, jQuery Validation si legatura cu JSON.
- ✓ Utilizare NuGet pentru a distribui software si gestiona dependentele in toate platformele.

ASP.NET MVC 4

Trasaturi noi

- ✓ ASP.NET Web API.
- ✓ Template-uri pentru proiecte pe mobile.
- ✓ Grupari si minimizari de fisiere.

ASP.NET Web API (referit ca *Web API*) este adaptat pentru a scrie servicii HTTP.

ASP.NET MVC 5

Trasaturi noi

- ✓ Rutare bazata pe attribute.
- ✓ ASP.NET Identity.
- ✓ Generare cod automat din model.
- ✓ Filtre locale, filtre globale.

ASP.NET MVC 6

ASP.NET 5

In summary, with ASP.NET 5 you gain the following foundational improvements:

- New light-weight and modular HTTP request pipeline
- Ability to host on IIS or self-host in your own process
- Built on .NET Core, which supports true side-by-side app versioning
- Ships entirely as NuGet packages
- Integrated support for creating and using NuGet packages
- Single aligned web stack for Web UI and Web APIs
- Cloud-ready environment-based configuration
- Built-in support for dependency injection
- New tooling that simplifies modern web development
- Build and run cross-platform ASP.NET apps on Windows, Mac and Linux
- Open source and community focused

Exemplificari caracteristici MVC

Motor vizualizare Razor

In MVC 1 si 2 motorul de vizualizare se numea *Web Form* pentru ca folosea sintaxa din Web Forms.

Sa urmarim urmatorul exemplu cu vechea sintaxa si apoi in noua sintaxa data de *Razor*.

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseV
iewModel>"
%>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent"
```

```

        runat="server">
            Browse Albums
    </asp:Content>
    <asp:Content ID="Content2" ContentPlaceHolderID="MainContent"
        runat="server">
    <div class="genre">

        <h3><em><%: Model.Genre.Name %></em> Albums</h3>

        <ul id="album-list">
            <% foreach (var album in Model.Albums) { %>
                <li>
                    <a href="<%: Url.Action("Details",
                        new { id = album.AlbumId }) %>">
                        <img alt="<%: album.Title %>"
                        src="<%: album.AlbumArtUrl %>" />
                        <span><%: album.Title %></span>
                    </a>
                </li>
            <% } %>
        </ul>
    </div>
</asp:Content>

```

iar cu Razor

```

@model MvcMusicStore.Models.Genre
@{ViewBag.Title = "Browse Albums";}
<div class="genre">
    <h3><em>@Model.Name</em> Albums</h3>
    <ul id="album-list">
        @foreach (var album in Model.Albums)
        {
            <li>
                <a href="@Url.Action("Details",
                    new { id = album.AlbumId })">
                    
                    <span>@album.Title</span>
                </a>
            </li>
        }
    </ul>
</div>

```

Observatie

Sintaxa <% %/> este inlocuita cu @. Razor vine cu o noua sintaxa. Razor nu e un limbaj de programare. Sintaxa Razor este asistata de Intellisense.

Rezolvare dependente

MVC 3 a introdus un nou concept numit *dependency resolver* ce simplifica utilizarea *dependency injection* in cadrul aplicatiei, fiind mai usor de decuplat componentele aplicatiei, facandu-le configurabile si mai usor de testat.

Suportul a fost adaugat pentru inregistrare si injectare la urmatoarele componente:

- ✓ Controller
- ✓ View
- ✓ Action filter
- ✓ Model binders
- ✓ Model validation providers
- ✓ Model metadata providers
- ✓ Value providers.

O descriere in detaliu despre Dependency Injection gasiti la adresa :

<http://www.asp.net/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-dependency-injection>

Un alt articol despre DI/IoC gasiti la adresa :

<http://martinfowler.com/articles/injection.html>

Rezolvare DI/IoC cu Autofac - free. Vezi <http://docs.autofac.org/en/latest/index.html> si un exemplu complex la adresa <http://www.codeproject.com/Articles/25380/Dependency-Injection-with-Autofac>

Ninject – pentru DI/IoC – free.

Integrare ASP.NET MVC cu Autofac :

<http://docs.autofac.org/en/latest/integration/mvc.html>

Construire componente slab cuplate – Dependency Injection

MVC permite crearea de componente care sa fie cat mai independente posibil si sa aiba cat mai putine interdependente. Situatiia ideala presupune ca fiecare componenta nu stie nimic despre alte componente din aplicatie si interactioneaza cu acestea prin intermediul interfetelor.

Dependency Injection este un pattern ce ajuta o clasa sa separe logica crearii obiectelor dependente. Rezultatul acestei separari este un sistem slab cuplat unde nu exista o dependenta rigida intre doua implementari concrete.

Sa presupunem ca am definit o clasa in C# astfel:

```
public class Customer
{
    private DatabaseHelper helper;
    public Customer()
    {
        helper = new DatabaseHelper();
    }
    ...
}
```


Clasa *Customer* declara o variabila de tip *DatabaseHelper*. Clasa *DatabaseHelper* se presupune ca executa anumite operatii (Select, Update, Insert, Delete) asupra unei baze de date. Constructorul clasei *Customer* creaza o instanta a clasei *DatabaseHelper* pe care o memoreaza in variabila locala *helper*. In acest caz clasa *DatabaseHelper* constituie o dependenta pentru clasa *Customer*, saul altfel spus clasa *Customer* depinde de clasa *DatabaseHelper*.

Proiectarea de mai sus a creat doua clase puternic cuplate. Daca in viitor vom dori sa inlocuim clasa *DatabaseHelper* cu o alta clasa (*XMLHelper* de exemplu), va trebui sa modificam codul din calasa *Customer* pentru ca aceasta instantiaza in mod direct clasa *DatabaseHelper*. Pentru a preveni aceasta problema vom folosi principiul DIP – Dependency Inversion Principle.

DIP precizeaza ca – *Modulele de nivel (high level) inalt nu ar trebui sa depinda de modulele de nivel redus (low level). Ambele ar trebui sa depinda de abstractizari.*

Acest lucru inseamna ca clasa *Customer* nu ar trebui sa depinda de o implementare concreta a clasei *DatabaseHelper* ci de o abstractizare a acesteia. La nivel de cod aceasta inseamna ca clasa *Customer* nu ar trebui sa aiba definita o variabila de tip *DatabaseHelper*, ci o variabila de tip interfata sau clasa abstracta.

```
public class Customer
{
    private IStorageHelper helper;
    public Customer()
    {
        helper = new DatabaseHelper();
    }
    ...
}
```

Acum clasa *Customer* foloseste o variabila de tip *IStorageHelper*. *IStorageHelper* este o interfata implementata de clasa *DatabaseHelper* si alte clase de acest fel. Problema nu este inca rezolvata deoarece in constructorul clasei *Customer* se creaza o instanta a tipului *DatabaseHelper*. Rezolvarea consta in a folosi principiul **IoC – Inversion of Control**.

IoC precizeaza : *Controlul crearii dependentelor ar trebui facut de un sistem extern si nu de clasa in cauza.*

Aceasta inseamna ca in exemplul considerat, clasa *Customer* nu ar trebui sa creeze o instanta a tipului *DatabaseHelper* ci sa primeasca aceasta instanta de la un sistem extern.

Dependency Injection constituie o modalitate de a implementa **IoC** astfel incat dependentele sunt « injectate » in clasa dintr-o sursa externa. Dependentele injectate pot fi primite ca parametru al constructorului clasei sau pot fi atribuite la proprietati din clasa, proprietati proiectate special in acest scop.

Exemplul devine (forma des intalnita in ASP.NET MVC):

```
public class Customer
{
    private IStorageHelper helper;

    public Customer(IStorageHelper helper)
    {
        this.helper = helper;
    }
    ...
}
```

Aplicare Dependency Injection in ASP.NET MVC

Exemplu despre cum este folosit DI in controller-e ASP.NET MVC.

(http://www.codeguru.com/csharp/.net/net_asp/mvc/understanding-dependency-injection.htm)

Consideram urmatoarea clasa controller:

```
public class HomeController : Controller
{
    ICustomerRepository repository = null;

    public HomeController(ICustomerRepository repository)
    {
        this.repository = repository;
    }

    public ActionResult Index()
    {
        List<CustomerViewModel> data = repository.SelectAll();
        return View(data);
    }
}
```

Clasa *HomeController* declara o variabila de tip *ICustomerRepository*. Interfata *ICustomerRepository* este proiectata sa implementeze pattern-ul Repository :

```
public interface ICustomerRepository
{
    List<CustomerViewModel> SelectAll();
}
```

```
CustomerViewModel SelectByID(string id);  
void Insert(CustomerViewModel obj);  
void Update(CustomerViewModel obj);  
void Delete(CustomerViewModel obj);  
}
```

In codul de mai sus implementarea interfeței *ICustomerRepository* este injectata in *HomeController* prin constructorul clasei. Metoda *Index()* apeleaza metoda *SelectAll()* din repository pentru a regasi o lista a tuturor obiectelor *CustomerViewModel*. Clasa *CustomerViewModel* poate avea urmatoarea definitie:

```
public class CustomerViewModel  
{  
    public string CustomerID { get; set; }  
    public string CompanyName { get; set; }  
    public string ContactName { get; set; }  
    public string Country { get; set; }  
}
```

In codul de mai sus apare o problema in crearea instantei clasei *HomeController*. ASP.NET MVC foloseste **constructor fara parametri** cand creaza o instanta a controller-ului. ASP.NET MVC permite de a specifica in mod explicit cum ar trebui instantiate controller-ele. Acest lucru poate fi facut prin crearea unui Controller Factory, adica a unei clase derivate din **DefaultControllerFactory** si care este reponsabila pentru crearea instantelor controller-elor. Dupa ce am creat o asemenea clasa trebuie sa o inregistram cu ASP.NET MVC framework.

```
public class MyControllerFactory:DefaultControllerFactory  
{  
    private Dictionary<string, Func<RequestContext, IController>>  
        controllers;  
  
    public MyControllerFactory(ICustomerRepository repository)  
    {  
        controllers = new Dictionary<string,  
            Func<RequestContext,IController>>();  
        controllers["Home"] = controller =>  
            new HomeController(repository);  
    }  
  
    public override IController CreateController(  
        RequestContext requestContext,  
        string controllerName)  
    {
```

```

        if (controllers.ContainsKey(controllerName))
        {
            return controllers[controllerName](requestContext);
        }
        else
        {
            return null;
        }
    }
}

```

Constructorul clasei *MyControllerFactory* accepta o instanță de tip *ICustomerRepository*. În acest mod *MyControllerFactory* nu depinde de nici o implementare concretă a interfeței *ICustomerRepository*. Clasa suprascrie metoda **CreateController()** din clasa de bază. Un dicționar de obiecte menține o listă a controller-elor din aplicație. Clasa *HomeController* este instantiată în constructorul clasei *MyControllerFactory* și este memorat cu cheia “*Home*”. Metoda **CreateController()** returnează instanța clasei *HomeController* din acest dicționar.

Pentru a înregistra clasa *MyControllerFactory* cu ASP.NET MVC framework construim clasa următoare:

```

public class ControllerFactoryHelper
{
    public static IControllerFactory GetControllerFactory()
    {
        string repositoryTypeName =
            ConfigurationManager.AppSettings["repository"];
        var repositoryType = Type.GetType(repositoryTypeName);
        var repository = Activator.CreateInstance(repositoryType);
        IControllerFactory factory = new MyControllerFactory(
            repository as ICustomerRepository);
        return factory;
    }
}

```

Clasa *ControllerFactoryHelper* are o metodă statică, **GetControllerFactory()**, care are ca sarcină să instanțieze clasa *MyControllerFactory* și să pregătească înregistrarea. În acest loc este nevoie de implementarea concretă a interfeței *ICustomerRepository*. Constructorul clasei *MyControllerFactory* are nevoie de un obiect ce implementează *ICustomerRepository*. Codul de mai sus presupune că aceste detalii sunt memorate în secțiunea **<appSettings>** din fișierul de configurare:

```
<appSettings>
    ...
    <add key="repository"
        value="DIDemo.Repositories.CustomerRepository"/>
</appSettings>
```

Metoda **GetControllerFactory()** citește cheia pentru repository și creează o instanță a tipului *DIDemo.Repositories.CustomerRepository* folosind *reflection*. În continuare se instanciază *MyControllerFactory* prin pasarea acestui obiect. Obiectul *factory* este apoi returnat apelantului.

Ultima etapă constă în a înregistra *factory controller*, ceea ce înseamnă adăugarea următorului cod în *Global.asax*:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    ControllerBuilder.Current.SetControllerFactory(
        ControllerFactoryHelper.GetControllerFactory());
}
```

În codul ce tratează evenimentul **Start** al aplicației, **Application_Start()**, instanța **ControllerBuilder** apelează metoda **SetControllerFactory()**. Obiectul *factory* returnat de metoda **GetControllerFactory()** este pasat ca parametru la metoda **SetControllerFactory()**.

■ End DI

Structura unei aplicații ASP.NET MVC

Director	Scop
/Controllers	Contine clasele pentru Controller ce gestionează cererile URL
/Models	Contine clasele ce reprezintă datele modelului, gestionarea acestor obiecte.
/Views	Contine fișiere template UI responsabile pentru afișarea rezultatului.
/Scripts	Contine biblioteci JavaScript și scripturi (.js).
/Images	Contine imaginile folosite în cadrul aplicației.
/Content	Putem pune CSS sau alt conținut dar nu scripturi și/sau imagini.
/Filters	Contine codul pentru filtre.
/App_Data	Contine fișierele de date Read/Write.
/App_Start	Contine cod de configurare, grupări de fișiere și Web API.

Observatie

Aceasta structura este generata (nu in totalitate) de catre VS. Se poate folosi si o structura personalizata in sensul adaugarii de noi directoare.

ASP.NET MVC si Conventii

MVC foloseste o conventie bazata pe numele structurii de directoare cand rezolva template pentru *View* si acest lucru ne permite sa ometem calea cand referim un anumit View din clasa derivata din **Controller**. Implicit, ASP.NET MVC cauta in locatia `|Views|[ControllerName]|` vizualizarea pentru o anumita actiune.

Acest concept este numit “*conventie peste configurare*”.

- Fiecare clasa controller are un nume ce termina cu *Controller* si clasa se gaseste in directorul */Controllers*.
- Exista un singur director */Views* pentru toate vizualizarile din aplicatie.
- Vizualizarile pe care le folosesc actiunile din controller se gasesc in directorul `|Views|[ControllerName]|`.
- Toate elementele UI reutilizabile sunt in directorul */Shared* din folder-ul *Views*.

Procesul de executie al unei aplicatii MVC (C#)

Cererile catre o aplicatie Web bazata pe ASP.NET MVC trec printr-un obiect **UrlRoutingModule**, obiect ce este un modul HTTP. Modulul parseaza cererea si determina calea de urmat (*route selection*).

Un obiect “*route*” este o instanta a unei clase ce implementeaza **RouteBase**, si in mod obisnuit este o instanta a clasei **Route**. Daca obiectul **UrlRoutingModule** nu poate determina o cale pentru a continua, cererea este returnata catre procesul ASP.NET sau IIS. Din obiectul **Route** selectat, obiectul **UrlRoutingModule** obtine un obiect **IRouteHandler**, obiect ce este asociat cu obiectul **Route**. In mod obisnuit, intr-o aplicatie MVC, acesta va fi o instanta a clasei **MvcRouteHandler**. Instanta **IRoutehandler** creaza un obiect **IHttpHandler** si ii transmite obiectul **IHttpContext**. Implicit, instanta **IHttpHandler** pentru MVC este obiectul **MvcHandler**. Obiectul **MvcHandler** va selecta controller-ul ce va trata cererea.

Modulul si *handler-ul* sunt puncte de intrare in framework ASP.NET MVC. Sunt realizate urmatoarele actiuni:

- Selectare controller.
- Obtinere instanta pentru controller-ul specificat.

- Apel metoda **Execute()** pe acea instanta.

Etapele executiei unui proiect MVC Web sunt:

Etapa	Detalii
Primește prima cerere	În fișierul <code>Global.asax</code> , obiecte Route sunt adăugate la obiectul RouteTable .
Execută rutarea	Modulul UrlRoutingModule folosește primul obiect Route (ce se potrivește) din tabela RouteTable și creează obiectul RouteData folosit pentru a crea obiectul RequestContext (HttpContext) .
Creare handler cerere MVC	Obiectul MvcRouteHandler creează o instanță a clasei MvcHandler .
Creare controller	Obiectul MvcHandler folosește instanța RequestContext pentru a identifica obiectul IControllerFactory (în general o instanță a clasei DefaultControllerFactory) și a crea instanța controller-ului.
Execută controller	MvcHandler apelează din controller metoda Execute .
Invocare acțiune	Se determină ce acțiune trebuie apelată (ce metodă din controller), se transferă parametrii (dacă există) la acea metodă și apoi se apelează acea metodă (acțiune).
Execute result	După executarea cererii se obține rezultatul care poate fi: ViewResult , RedirectToRouteResult , RedirectResult , ContentResult , JsonResult sau EmptyResult .

Modele, View-uri și Controller-e (C#)

Un URL nu este egal cu o pagină.

În ASP.NET Web Forms este o corespondență între URL și pagină (pentru fiecare pagină se apelează fișierul `*.aspx` corespunzător).

În Asp MVC cererile din browser sunt mapate la acțiuni din controller.

Rutarea

Se folosește o tabelă de rutare pentru a trata cererile ce apar. Rutarea din ASP.NET este folosită de ASP.NET MVC.

Metoda folosită pentru a înregistra o *rută* este **RegisterRoutes** în care se adaugă o intrare în **RouteCollection** folosind metoda **MapRoute** sau **MapPageRoute**. Codul se plasează în `Global.asax` în cadrul apelului metodei **Application_Start**.

O rută este compusă din :

- Nume
- URL cu parametri : "{controller}/{action}/{id}"
- Parametri dati sub forma unui obiect. Un exemplu este :
`new { controller = "Home", action = "Index", id = "" }`

Exemplu - Global.asax

```
...
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id = "" } // Parameter defaults
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Ruta implicita din *Global.asax* include valori implicate pentru cei trei parametri:

`"{controller}/{action}/{id}"`.

Din exemplul de mai sus avem:

`controller = "Home", action = "Index", id = ""`

Controllers

Controller-ul este responsabil cu tratarea input-ului de la utilizator (raspunde la interactiunile utilizatorului cu aplicatia). Controllerul contine logica desfasurarii aplicatiei. Un controller este o clasa al carei nume trebuie sa se termine (obligat) cu *Controller* si este derivat din clasa **Controller**.

HomeController.cs


```
...
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Title"] = "Home Page";
            ViewData["Message"] = "Welcome to ASP.NET MVC!";
            return View();
        }

        public ActionResult About()
        {
            ViewData["Title"] = "About Page";
            return View();
        }
    }
}
```

Views

Actiunile expuse, in exemplul de mai sus, de catre controller sunt: *Index()* [exact ca *index.html*] si *About()*.

View este echivalent cu o pagina.

View-urile trebuiesc create in locatia corecta. De exemplu, actiunii **Index** din controller **Home** ii corespunde vizualizarea *Index.cshtml* plasata in folderul **\Views\Home** si are numele **Index.cshtml**.

Regula generala:

Pentru fiecare actiune exista un *view* cu acelasi nume si extensia *.cshtml*, fisier plasat in directorul:

|Views|<nume_controller_fara_sufix_Controller>|

Pentru exemplul de mai sus avem:

\Views\Home\Index.cshtml
\Views\Home\About.cshtml

Sa urmarim un mic exemplu in care se evidentiaza cod in actiuni din controller si modelele folosite in vizualizari.

Optiunea About...

Cod partial

Controller Home

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVC3.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Welcome to ASP.NET MVC!";
            ViewBag.AResult = MVC3.A.Ma();
            ViewBag.TextIndex = "Un text plasat in metoda
                                Index din controller Home";
            return View();
        }

        public ActionResult About()
        {
            // Asa pot aduce inregistrari din baza de date
            // de vazut directiva @model din About.cshtml
            // Vom transmite aceasta colectie catre vizualizare
            // (About.cshtml)
            List<string> str = new List<string>();
            for (int i = 0; i < 10; i++)
            {
                str.Add("Item " + i.ToString());
            }
            ViewBag.TextAbout = "Am transmis la View o
                                colectie List<string>";
            return View(str);
        }
    }
}
```

Vizualizarea pentru actiunea About (se gaseste in View -> Home->About.cshtml)

```
@model List<string>
@{
    ViewBag.Title = "About Us";
}

<h2>@ViewBag.Message</h2>
<p>
    Adaugam continut aici...
    <br />
```

```

    <h2>@ViewBag.TextAbout</h2>
    <ul>
    <li>@ViewBag.TextAbout</li>
        @foreach (string s in Model)
        {
            <li>@s</li>
        }
    </ul>
</p>

```

Pagina de start este in *_ViewStart.cshtml* ce contine urmatoarea declaratie – pagina master folosita :

```

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

iar continutul paginii *_Layout.cshtml* este:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
        type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"
        type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <!-- <header> -->
        <div id="title">
            <h1>My MVC Application</h1>
        </div>
        <div id="logindisplay">
            @Html.Partial("_LogOnPartial")
        </div>
        <nav>
            <ul id="menu">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                <li>@Html.ActionLink("Next", "LogOn", "Account")</li>
                <li>@Html.ActionLink("About", "About", "Home")</li>
                <li>@Html.ActionLink("Default1 controller",
                    "Index", "Default1")</li>
                <li>@Html.Label("expresie", "Text label")</li>
            </ul>
        </nav>
        <!-- </header> -->
        <section id="main">
            @RenderBody()
        </section>
        <footer>

```

```
</footer>
</div>
</body>
</html>
```

RenderBody()

Metoda este referita ca pagina de Layout. Poate exista numai o metoda **RenderBody** per Layout pagina. Este asemanatoare cu controlul **ContentPlaceHolder**. Metoda indica unde va fi plasat template-ul vizualizarii in continutul elementului <body>.

RenderPage()

Paginile layout pot avea continut ce poate fi adus de pe alte pagini. **RenderPage** face exact acest lucru. Metoda are unul sau doi parametri. Primul parametru indica locatia fizica a fisierului, iar al doilea, ce este optional, contine un array de obiecte ce pot fi plasate pe pagina.

Exemplu

```
@RenderPage("~/Views/Shared/_AnotherPage.cshtml")
```

Observatie

ASP.NET nu afiseaza direct pagini ce incep cu _ (underscore) in denumire.

RenderSection()

Metoda are un parametru ce indica numele sectiunii si unul de tip *bool* ce semnifica daca sectiunea este optionala sau nu. Views-urile pot adauga sectiuni folosind urmatorul cod :

```
@section footer
{ <b> Pagina subsol aici </b> }
```

Creare Rute (C#)

O ruta informeaza ASP.NET Framework despre modul cum sa proceseze un URL ce nu corespunde la un fisier .aspx de pe hard disc.

Conventia pentru utilizarea rutarii este de a crea un folder *App_Start* si apoi crearea in acest folder a unei clase numita *RouteConfig* (fisier *RouteConfig.cs*), clasa ce va contine o metoda ce configureaza sistemul de rutare pentru aplicatia pe care o cream. Metoda din aceasta clasa o vom apela din clasa globala a aplicatiei (*Global.asax*). Continutul pentru *App_Start/RouteConfig.cs* este :

```
using System.Web.Routing;
namespace Routing // numele namespace-ului poate fi modificat
{
    public class RouteConfig
```

```

    {
        public static void RegisterRoutes(RouteCollection routes)
        {
        }
    }
}

```

Observatie

Parametrul metodei **RegisterRoutes** este un obiect de tip **RouteCollection**, clasa definita in **System.Web.Routing**.

In aceasta metoda se definesc *cai virtuale* la aplicatia web prin adaugarea de « rute » la colectia **RouteCollection**. Cand se primeste o cerere de catre ASP.NET, se proceseaza rutele de catre un modul ce rescrie caile conform configurarii create. Codul pentru apelul configurarii rutelor este definit in *Global.asax.cs* in cadrul metodei **Application_Start** astfel :

```

using System;
using System.Web.Routing;
namespace Routing
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            // cod lipsa
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}

```

Observatie

Framework-ul apeleaza metoda **Application_Start** pentru evenimentul **Start** al aplicatiei.

Rute fixe – metoda MapPageRoute

Ruta fixa nu inseamna altceva decat crearea unei cai virtuale fixe ce are ca efect afisarea unui formular Web al carui nume il stim in momentul crearii rutei.

Metoda folosita este **MapPageRoute(...)**. Exemplu:

```

using System.Web.Routing;
namespace Routing {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapPageRoute("default", "", "~/Default.aspx");
            routes.MapPageRoute("cart1", "cart", "~/Store/Cart.aspx");
            routes.MapPageRoute("cart2", "apps/shopping/finish",
                                "~/Store/Cart.aspx");
        }
    }
}

```

Descrierea metodei **MapPageRoute** in MSDN este data astfel :

Name	Description
MapPageRoute(String, String, String)	Provides a way to define routes for Web Forms applications.
MapPageRoute(String, String, String, Boolean)	Provides a way to define routes for Web Forms applications.
MapPageRoute(String, String, String, Boolean, RouteValueDictionary)	Provides a way to define routes for Web Forms applications.
MapPageRoute(String, String, String, Boolean, RouteValueDictionary, RouteValueDictionary)	Provides a way to define routes for Web Forms applications.
MapPageRoute(String, String, String, Boolean, RouteValueDictionary, RouteValueDictionary, RouteValueDictionary)	Provides a way to define routes for Web Forms applications.

Observatie

Metoda este echivalenta cu apelul metodei **Add** pentru o colectie si pasand un obiect **Route**, obiect ce este creat de clasa **PageRouteHandler**.

Semnificatia parametrilor din aceasta metoda (ultimul prototip din lista de mai sus).

```
public Route MapPageRoute (
    string routeName,
    string routeUrl,
    string physicalFile,
    bool checkPhysicalUrlAccess,
    RouteValueDictionary defaults,
    RouteValueDictionary constraints,
    RouteValueDictionary dataTokens
)
```

Parameters

routeName

Type: [System.String](#)

The name of the route.

routeUrl

Type: [System.String](#)

The URL pattern for the route.

*physicalFile*Type: [System.String](#)

The physical URL for the route.

*checkPhysicalUrlAccess*Type: [System.Boolean](#)

A value that indicates whether ASP.NET should validate that the user has authority to access the physical URL (the route URL is always checked). This parameter sets the [PageRouteHandler.CheckPhysicalUrlAccess](#) property.

*defaults*Type: [System.Web.Routing.RouteValueDictionary](#)

Default values for the route parameters.

*constraints*Type: [System.Web.Routing.RouteValueDictionary](#)

Constraints that a URL request must meet in order to be processed as this route.

*dataTokens*Type: [System.Web.Routing.RouteValueDictionary](#)

Values that are associated with the route that are not used to determine whether a route matches a URL pattern.

Return Value

Type: [System.Web.Routing.Route](#)

The route that is added to the route collection.

Exemplu (MSDN)

```
routes.MapPageRoute("ExpenseDetailRoute",
    "ExpenseReportDetail/{locale}/{year}/{*queryvalues}",
    "~/expenses.aspx",
    false,
    new RouteValueDictionary
    { { "locale", "US" }, { "year", DateTime.Now.Year.ToString() } },
    new RouteValueDictionary
    { { "locale", "[a-z]{2}" }, { "year", @"\d{4}" } },
    new RouteValueDictionary
    { { "account", "1234" }, { "subaccount", "5678" } }
);
```

Observatie

1. Numele rutei trebuie sa fie *unic* la nivel de aplicatie si nu trebuie sa inceapa cu */*.
2. URL fizic al rutei trebuie sa fie specificat *relativ* la radacina aplicatiei, adica folosirea notatiei *~*.
3. Pattern-ul URL al rutei (al doilea argument) se mai numeste si *cale virtuala*.

Cateva explicatii pentru rutele create cu codul

```
using System.Web.Routing;
namespace Routing {
    public class RouteConfig {
```

```

        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapPageRoute("default", "", "~/Default.aspx");
            routes.MapPageRoute("cart1", "cart", "~/Store/Cart.aspx");
            routes.MapPageRoute("cart2", "apps/shopping/finish",
                                "~/Store/Cart.aspx");
        }
    }
}

```

Rute indicate corect

Cale virtuala	Pagina Web form
/ (the root URL)	/Default.aspx
/cart	/Store/Cart.aspx
/apps/shopping/finish	/Store/Cart.aspx

Rute indicate corect / incorect pentru ruta="cart2"

Cale virtuala	Se potriveste cu pattern-ul indicat?
/apps/shopping/finish	Da. Cele trei segmente sunt identice.
/apps/shopping	Nu. Prea putine segmente.
/apps/shopping/finish/cart	Nu. Prea multe segmente.
/app/shopping/checkout	Nu. Numar segmente corect, dar nu se potriveste cu pattern-ul de la adaugarea rutei.

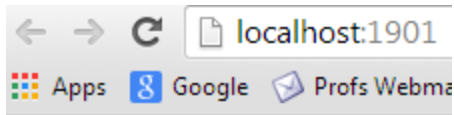
Obtinere informatii despre rute

Se pot obtine informatii despre cum sistemul de rutare a procesat o cerere si a selectat pagina care se potriveste pattern-ului folosind proprietatea **RouteData** din clasa **Page**, proprietate ce returneaza un obiect **System.Web.Routing.RouteData** de unde obtinem informatiile necesare.

Proprietati importante din clasa **RouteData**

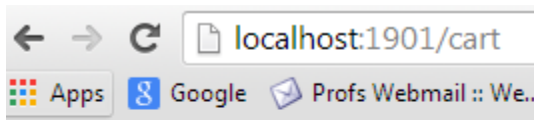
DataTokens	Returneaza o colectie cheie/valoare asociata cu ruta ce a fost aplicata cererii curente.
Route	Returneaza un obiect RouteBase ce furnizeaza informatii despre ruta ce a fost aplicata cererii curente.
RouteHandler	Returneaza IRouteHandler ce a mapat cererea cererea la un IHttpHandler .
Values	Retrunceaza o multime de valori ale parametrilor pentru ruta.

Ruland aplicatia cu diverse URL-uri obtinem urmatoarele:



This is Default.aspx

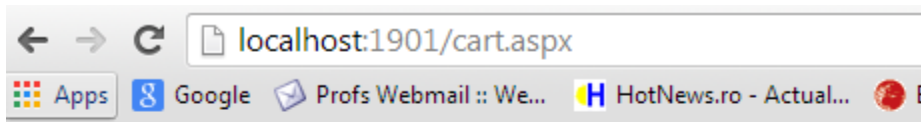
Route Path:



This is /Store/Cart.aspx

Route Path: cart

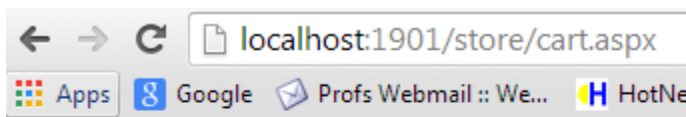
URL: .../cart.aspx



Server Error in '/' Application.

The resource cannot be found.

URL: .../store/cart.aspx



This is /Store/Cart.aspx

Route Path: Unknown RouteBase

Observatie

Proprietatea **Url** returneaza calea folosita pentru a defini ruta, nu URL-ul cererii curente.

RouteData returneaza un obiect **Route**. Clasa **Route** are mai multe proprietati dintre care amintim (MSDN) :

Constraints	Gets or sets the constraints used to limit the range of URLs that the route will match.
DataTokens	Gets or sets the data tokens associated with the route.
Defaults	Gets or sets the default values for variable segments.
RouteExistingFiles	Gets or sets whether the routing system should handle requests that match existing files.
RouteHandler	Gets or sets the IRouteHandler implementation associated with the route.
Url	Gets or sets the path used by the route.

Adaugare de segmente variabile

Segmentul variabil permite ca o singura ruta sa se potriveasca cu URL-uri multiple. Segmentul variabil este notat intre { } si contine numele unei variabile.

Exemplu

```
routes.MapPageRoute("dall", "{app}/default", "~/Default.aspx");  
routes.MapPageRoute("d4", "store/default", "~/Store/Cart.aspx");
```

cand sistemul de rutare intalneste un segment variabil va mapa orice valoare pentru acest segment si ca atare cererile */Iasi/default*, */Copou/default* vor fi toate mapate catre **Default.aspx**.

Observatie

In exemplul de mai sus am creat o ruta care are doua segmente din care ultimul segment este *default*. Din cauza ca sistemul de rutare trateaza prima ruta pe care o intalneste in colectie, va rezulta ca ruta */store/default* va fi tratata de catre *Default.aspx* si nu de */Store/cart.aspx*.

Rezolvarea problemei consta din a plasa rutele fara segmente variabile in fata rutelor cu segmente variabile. In cazul nostru ordinea corecta ar fi:

```
routes.MapPageRoute("d4", "store/default", "~/Store/Cart.aspx");  
routes.MapPageRoute("dall", "{app}/default", "~/Default.aspx");
```

Modelul de asociere la valorile segmentului din ruta

Acest model permite sa integram valorile din segmentul de rutare in cod. Explicam acest lucru pe baza urmatorului exemplu.

In metoda **RegisterRoutes** adaugam urmatorul cod (o noua ruta):

```
routes.MapPageRoute("loop", "{count}", "~/Loop.aspx", false,  
    new RouteValueDictionary { { "count", "3" } },  
    new RouteValueDictionary { { "count", "[0-9]*" } });
```

Segmentul variabil este {*count*}. Dorim ca valorile pe care le primește *count* sa fie afisate in pagina Web numita *Loop.aspx*.

Loop.aspx este definita astfel:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Loop.aspx.cs"
    Inherits="Routing.Loop" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <title></title>
  </head>
  <body>
    <p>This is the Loop.aspx Web Form</p>
    <ul>
      <asp:Repeater ID="Repeater1" ItemType="System.Int32"
        SelectMethod="GetValues" runat="server">
        <ItemTemplate>
          <li><%# Item %></li>
        </ItemTemplate>
      </asp:Repeater>
    </ul>
  </body>
</html>
```

In controlul **asp:Repeater** vom afisa valorile furnizate de metoda *GetValues* indicata de atributul **SelectMethod**.

In fisierul Loop.aspx.cs avem urmatorul cod:

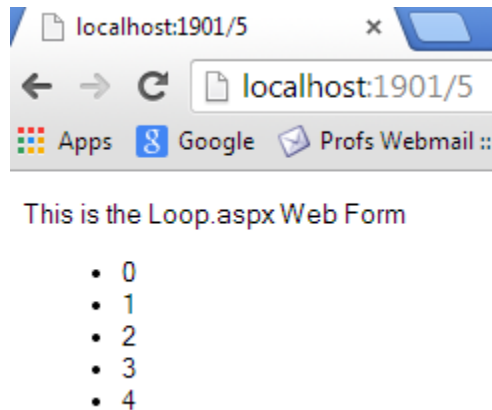
```
using System.Collections.Generic;
using System.Web.ModelBinding; // de adaugat aceasta referinta
namespace Routing
{
    public partial class Loop : System.Web.UI.Page
    {
        public IEnumerable<int> GetValues([RouteData("count")] int? count)
        {
            for (int i = 0; i < (count ?? 3); i++)
            {
                yield return i;
            }
        }
    }
}
```

Parametrul *count* al metodei *GetValues*, definit ca **Nullable**, este adnotat cu atributul **RouteData**. In cadrul acestui atribut specificam numele segmentului pentru care dorim sa-i preluam valoarea, in cazul nostru *count*. Atributul **RouteData** este definit in **System.Web.ModelBinding**, si are rolul de a furniza valoarea argumentului luata din segmentul variabil numit *count*, din ruta.

Observatie

In bucla **for** avem **i < (count ?? 3)**, ceea ce inseamna ca daca *count* nu are valoare, adica **count.HasValue** este *false*, se va utiliza valoarea implicita 3.

Rulam aplicatia si vom avea implicit URL: localhost:*port*. Modificam acest URL in localhost:port/5 si rezultatul va fi:



Exemplu : Ne propunem sa cream o ruta numita *Blog* si care sa trateze cereri de forma /Archive/*entrydate*, unde “entrydate” este de tip Data calendaristica.

Ruta se creaza in fisierul *Global.asax*.

Modificarile in Global.asax sunt urmatoarele:

Global.asax (cu custom route)

```
using System.Web.Mvc;
using System.Web.Routing;
namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute( "Blog", // Route name
                "Archive/{entryDate}", // URL with parameters
                new { controller = "Archive", action = "Entry" }
                // Parameter defaults
            );

            routes.MapRoute( "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id = "" }
                // Parameter defaults
            );
        }
    }
}
```

```
    }

    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
}
```

Ordinea rutelor pe care o adaugam la tabela de rutare este importanta.

Ruta *Blog* creata mai sus va fi folosita de orice cerere ce starteaza cu */Archive/*. In concluzie aceasta se potriveste pentru orice cerere de forma :

/Archive/01-01-2011
/Archive/31-12-2010
/Archive/Iasi

Controller-ul va fi *Archive* si va fi invocata metoda *Entry('01-01-2011')*, *Entry(Iasi)*, etc., metoda al carei prototip este :

```
public string Entry(DateTime entryDate);
```

Observatie:

- In exemplul de mai sus parametrul metodei *Entry* se presupune ca este de tip **DateTime**.
- Conversia parametrului catre tipul **DateTime** este facuta de ASP. NET MVC.

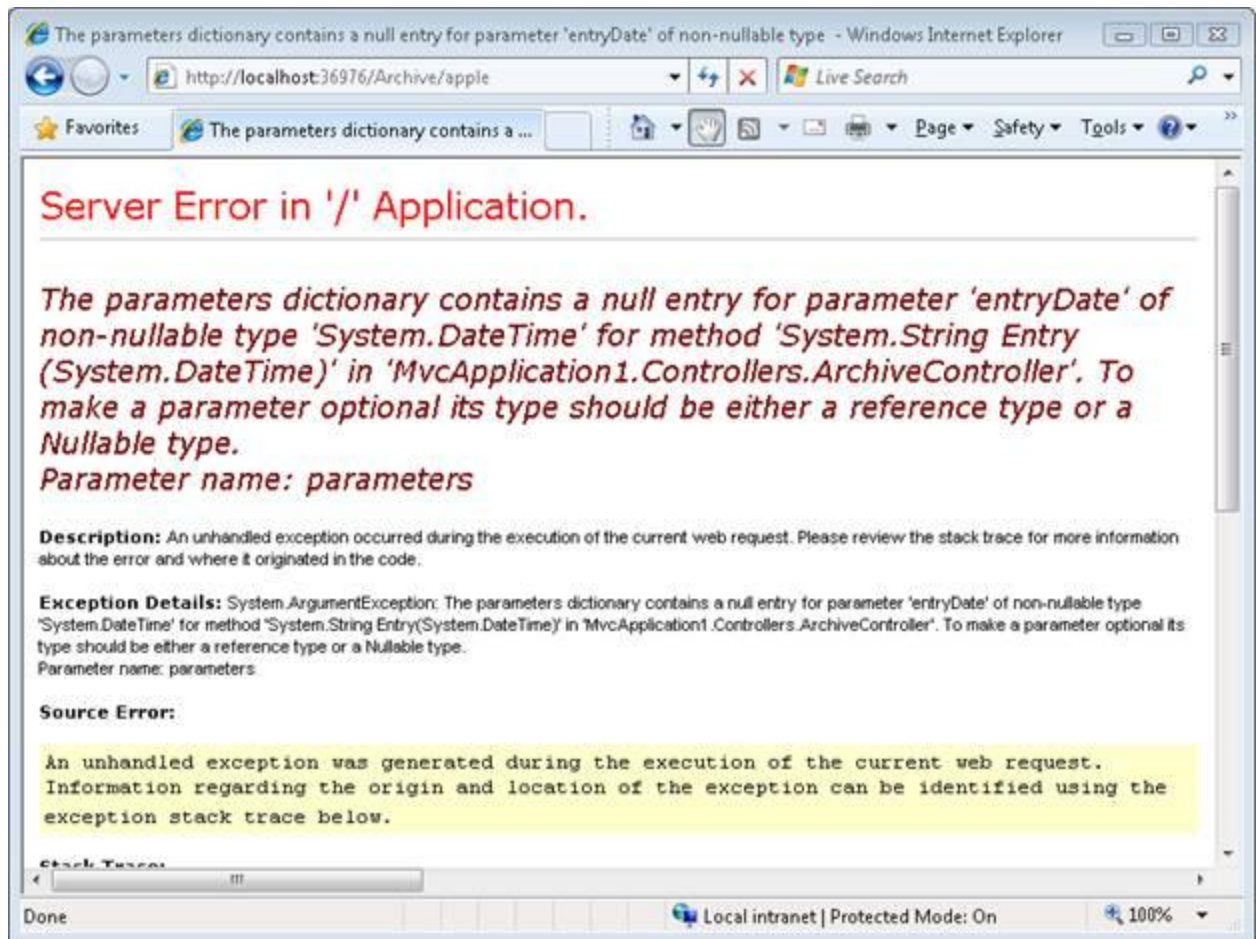
Codul din controller poate fi urmatorul:

ArchiveController.cs

```
using System;
using System.Web.Mvc;
namespace MvcApplication1.Controllers
{
    public class ArchiveController : Controller
    {
        public string Entry(DateTime entryDate)
        {
            return "You requested the entry from " +
                entryDate.ToString();
        }
    }
}
```

Observatie

Daca parametrul din metoda *Entry* nu poate fi convertit la *DateTime*, se genereaza o pagina ce contine o descriere a erorii.



Alte exemple de rute (MSDN) – codurile de mai jos se adauga in RegisterRoutes.

- ❖ Se adauga o ruta fara nume ce are pattern URL care contine valoarea literala « *SalesReportSummary* » si un parametru URL (placeholder) numit *year*. Ruta mapeaza catre un fisier numit *Sales.aspx*.


```
routes.MapPageRoute("",
    "SalesReportSummary/{year}",
    "~/sales.aspx");
```
- ❖ 2. Acest cod adauga o ruta numita *SalesRoute*. Aceasta ruta trebuie sa aiba un nume deoarece are aceeasi parametri ca ruta anterioara si altfel nu se poate face diferenta.


```
routes.MapPageRoute("SalesRoute",
    "SalesReport/{locale}/{year}",
    "~/sales.aspx");
```
- ❖ 3. Se adauga o ruta cu numele *ExpensesRoute* si include un parametru ce poate contine orice valoare, parametru numit *extrainfo*. Pe langa acest parametru mai sunt definiti alti doi parametri, *locale* cu valoarea implicita "US" si *year* cu valoarea anul curent. Pattern-ul pentru acesti doi parametri sunt dati in (expresii regulate) :

```
new RouteValueDictionary {  
    { "locale", "[a-z]{2}" },  
    { "year", @"\d{4}" } }  
};
```

iar ruta se inregistreaza astfel:

```
routes.MapPageRoute("ExpensesRoute",  
    "ExpenseReport/{locale}/{year}/{*extrainfo}",  
    "~/expenses.aspx", true,  
    new RouteValueDictionary {  
        { "locale", "US" },  
        { "year", DateTime.Now.Year.ToString() } },  
    new RouteValueDictionary {  
        { "locale", "[a-z]{2}" },  
        { "year", @"\d{4}" } }  
);
```

Creare hyperlink-uri folosind rute

Modalitati de creare

- 1. *hard code*
- 2. specificare nume parametri din *route* si valorile corespunzatoare, iar ASP.NET va genera URL ce corespunde cu acestia. Se poate specifica si numele rutei pentru a identifica in mod unic o ruta.

Metoda 1. (*hard code*)

```
<asp:HyperLink ID="HyperLink1" runat="server"  
    NavigateUrl="~/salesreportsummary/2010">  
    Sales Report - All locales, 2010  
</asp:HyperLink>  
<br />  
<asp:HyperLink ID="HyperLink2" runat="server"  
    NavigateUrl="~/salesreport/WA/2011">  
    Sales Report - WA, 2011  
</asp:HyperLink>  
<br />  
<asp:HyperLink ID="HyperLink3" runat="server"  
    NavigateUrl="~/expensereport">  
    Expense Report - Default Locale and Year (US, current year)  
</asp:HyperLink>  
<br />
```

Metoda 2

In .aspx

```
<asp:HyperLink ID="HyperLink6" runat="server">
    Expense Report - CA, 2008
</asp:HyperLink>
<br />
```

iar in cod (metoda *Page_Load*) trebuie sa scriem ceva de genul:

```
RouteValueDictionary parameters =
    new RouteValueDictionary
    {
        { "locale", "CA" },
        { "year", "2008" },
        { "category", "recreation" }
    };
```

Acest cod creaza o instanta a clasei *RouteValueDictionary* ce contine trei parametri. Al treilea parametru este *category* si valoarea lui va fi redata ca un parametru “*query string*”.

```
VirtualPathData vpd =
    RouteTable.Routes.GetVirtualPath(null, "ExpensesRoute",
    parameters);

HyperLink6.NavigateUrl = vpd.VirtualPath;
```

Accesare parametri din URL in .aspx

```
<h1>
    Expense Report for
    <asp:Literal ID="Literal1"
        Text="<%=RouteValue.locale%"
        runat="server"></asp:Literal>,
    <asp:Literal ID="Literal2"
        Text="<%=RouteValue.year%"
        runat="server"></asp:Literal>
</h1>
```

Accesare parametri din URL in cod

Valorile pentru Literal vor fi setate in cod.

```
<h1>
    Sales Report for
    <asp:Literal ID="LocaleLiteral" runat="server"></asp:Literal>,
    <asp:Literal ID="YearLiteral" runat="server"></asp:Literal>
</h1>
```

In *Page_Load()*


```
LocaleLiteral.Text = Page.RouteData.Values["locale"] == null ?  
    "All locales" : Page.RouteData.Values["locale"].ToString();
```

```
YearLiteral.Text = Page.RouteData.Values["year"].ToString();
```

Exemple de creare rute

Rute cu valori fixe

```
routes.MapRoute( "MyRoute", "{controller}/{action}");
```

Aceasta ruta are doua segmente: *controller* si *action* plasate intre { si }. “MyRoute” este numele rutei. Rutele care se potrivesc cu aceasta declaratie sunt:

```
http://www.partyplanner.com/party/index  
http://www.partyplanner.com/comment/post  
http://www.mysite.com/home/index
```

Rute care nu se potrivesc cu aceasta declaratie:

```
http://www.partyplanner.com/party  
http://www.partyplanner.com/comment  
http://www.mysite.com/home/  
http://www.mysite.com
```

Rute cu valori implicite

```
routes.MapRoute(  
    "MyRoute",  
    "{controller}/{action}",  
    new { controller = "party", action = "index" }  
);
```

Rute care se potrivesc cu aceasta declaratie

```
http://www.partyplanner.com/party/index  
http://www.partyplanner.com/comment/post  
http://www.mysite.com/home/index  
http://www.mysite.com
```

Potrivre cu toate segmentele – {*param}

```
routes.MapRoute(  
    "CustomRoute",  
    "product/{*param}",  
    new { controller = "Product", action = "Index" }  
);
```

Rute care se potrivesc cu aceasta declaratie

```
http://mysite.com/product/hello  
http://mysite.com/product/hello/a/b/c
```

Potrivire cu toate rutele – {*url}

```
routes.MapRoute(  
    "CatchAllRoute",  
    "{*url}",  
    new { controller = "Home", action = "Index",  
          url = UrlParameter.Optional }  
);
```

Rute care se potrivesc cu aceasta declaratie

```
http://mysite.com  
http://mysite.com/product/hello  
http://mysite.com/home/index/hello/text/1
```

Rute cu separatori diferiti - ~

```
routes.MapRoute(  
    "MyRoute",  
    "{controller}~{action}~{id}"  
);
```

In loc de / separam segmentele cu ~.

```
http://mysite.com/product~list~1
```

Ignorare rute

Se instiinteaza infrastructura de rutare sa nu trateze anumite cereri prin ignorarea rutelor. Implicit infrastructura de rutare nu trateaza cereri pentru resurse statice si acest lucru poate fi controlat de proprietatea **RouteExistingFiles**. Definitia pentru ignorarea rutelor este data astfel:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

Declaratia de mai sus spune ca se vor ignora cererile ce contin extensia ".axd". Un fisier cu extensia .axd este un fisier HTTP Handler. Metoda **IgnoreRoute** este suprascrisa si putem pasa constrangeri in al doilea parametru al metodei.

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}",  
    new { resource = new CustomConstraint() });
```

Proprietatea *RouteExistingFiles* are implicit valoarea false. Se seteaza pe true aceasta proprietate dar dupa metoda IgnoreRoute.

MVC 5 – Definire rute folosind atribute

Acest mecanism introdus in MVC 5 permite de a defini ruta in acelasi loc unde am definit si actiunea. Atributul folosit se numeste **Route** si poate fi atasat direct la metoda (actiune) din controller.

```
[Route("Products/Electronics/{id}")]
public ActionResult GetElectronicItems(string id)
{
    ViewBag.Id = id;
    return View();
}
```

In fisierul *RouteConfig* trebuie sa adaugam urmatorul cod:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapMvcAttributeRoutes();
}
```

Parametrul optional se defineste in pattern-ul URL folosind notatia **?**.

```
[Route("Products/Electronics/{id?}")]
public ActionResult GetElectronicItems(int? id)
{
    ViewBag.Id = id; return View();
}
```

id este un tip **Nullable**, si existenta valorii se testeaza folosind proprietatea **HasValue**.

Constrangeri : se specifica tipul exact al parametrului.

```
[Route("Products/Electronics/{id:int}")]
```

Alte constrangeri folosite in atributul **Route**

Route Constraint	Used To
x:bool	Match a bool parameter
x:maxlength(n)	Match a string parameter with maximum length of n characters
x:minlength(n)	Match a string parameter with minimum length of n characters
x:max	Match an integer parameter with a maximum value of n.
x:min	Match an integer parameter with a minimum value of n.
x:range	Match an integer parameter within a range of values.

x:float	Match floating-point parameter.
x:alpha	Match uppercase or lowercase alphabet characters
x:regex	Match a regular expression.
x:datetime	Match a DateTime parameter.

Atributul **RoutePrefix**

In cazul cand un controller contine mai multe actiuni ce folosesc acelasi prefix, putem atasa atributul **RoutePrefix** la controller si valorile se vor propaga si pentru actiuni.

```
[RoutePrefix("Products")] // aplicat la controller
```

Ceea ce am declarat in **RoutePrefix** va prefixa rutele definite cu atributul **Route** pentru actiuni.

La actiuni acum putem scrie

```
[Route("Electronics/{id}")] // ruta va deveni  
/Products/Electronics/{id}
```

Creare ruta cu constrangeri (C#)

Sa ne imaginam urmatoarea ruta, definita in Global.asax :

```
routes.MapRoute(  
    "Product",  
    "Product/{productId}",  
    new {controller="Product", action="Details"}  
);
```

In acest caz *productId* trebuie sa fie de tip *int*.

Ideea este ca trebuie sa verificam daca valoarea din cerere este de tipul asteptat (int) iar in caz contrar sa afisam o pagina cu eroare.

Putem verifica parametrii astfel :

- sa folosim constrangeri pentru ruta ;
- sa folosim expresii regulate pentru a specifica o constrangere pentru ruta.

Controller-ul pentru aceasta ruta este :

Controllers\ProductController.cs

```
using System.Web.Mvc;  
namespace MvcApplication1.Controllers  
{
```

```
public class ProductController : Controller
{
    public ActionResult Details(int productId)
    {
        return View();
    }
}
```

Actiunea *Details* are ca parametru un int.

Pentru a verifica ca am pasat un int in cerere, modificarea trebuie sa o facem in crearea rutei, ideea este de a nu permite browser-ului sa selecteze o anumita ruta daca nu toti parametrii sunt de tipul specificat.

In Global.asax ar trebui sa scriem (se folosesc expresii regulate):

Global.asax.cs

```
routes.MapRoute( "Product",
    "Product/{productId}",
    new {controller="Product", action="Details"},
    new {productId = @"\d+" }
);
```

Cererile browser-ului ce nu se potrivesc unei rute stabilite vor avea ca efect returnarea erorii: *The resource could not be found.*

Creare constrangeri pentru o ruta personalizata (C#)

Rol de baza interfata : **IRouteConstraint**

- defineste contractul pe care o clasa trebuie sa-l implementeze pentru a putea verifica daca valoarea unui parametru URL este valida pentru o constrangere.

Namespace: System.Web.Routing

Assembly: System.Web (in System.Web.dll)

Pentru a exemplifica vom crea o ruta ce va avea ca efect tratarea numai a cererilor ce vin de la calculatorul local, celelalte cereri vor fi anulate. Cream ruta numita *Localhost* si constrangeri pentru aceasta.

Interfata **IRouteConstraint** contine o singura metoda (Doc. din MSDN)

```
bool IRouteConstraint.Match(
    HttpContextBase httpContext,
    Route route,
```

```

        string parameterName,
        RouteValueDictionary values,
        RouteDirection routeDirection
    )

```

Parameters

HttpContext

Type: [System.Web.HttpContextBase](#)

An object that encapsulates information about the HTTP request.

route

Type: [System.Web.Routing.Route](#)

The object that is being checked to determine whether it matches the URL.

parameterName

Type: [System.String](#)

The name of the parameter that is being checked.

values

Type: [System.Web.Routing.RouteValueDictionary](#)

An object that contains the parameters for a route.

routeDirection

Type: [System.Web.Routing.RouteDirection](#)

An object that indicates whether the constraint check is being performed when an incoming request is handled or when a URL is generated.

Return Value

Type: [System.Boolean](#)

true if the request was made by using an allowed HTTP verb; otherwise, false. The default is true.

Observatie: Daca metoda returneaza false, atunci ruta asociata cu constrangere nu se potriveste cu cererea din browser. Metoda se foloseste pentru a adauga o validare logica pentru constrangere. Metoda este apelata de framework – nu de programator – :

- cand se proceseaza o cerere;
- cand se construieste un nou URL. Putem determina in ce situatie ne aflam prin examinarea parametrului `routeDirection`.

Cod in LocalhostConstraint.cs

```

using System.Web;
using System.Web.Routing;
namespace MvcApplication1.Constraints
{
    public class LocalhostConstraint : IRouteConstraint
    {
        public bool Match ( HttpContextBase httpContext,
                           Route route,
                           string parameterName,
                           RouteValueDictionary values,
                           RouteDirection routeDirection )
        {
            return httpContext.Request.IsLocal;
        }
    }
}

```

```
    }  
}
```

iar in

Global.asax

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
using System.Web.Routing;  
using MvcApplication1.Constraints;  
namespace MvcApplication1  
{  
    public class MvcApplication : System.Web.HttpApplication  
    {  
        public static void RegisterRoutes(RouteCollection routes)  
        {  
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
            routes.MapRoute( "Admin", "Admin/{action}",  
                new {controller="Admin"},  
                new {isLocal=new LocalhostConstraint()}  
            );  
        }  
  
        protected void Application_Start()  
        {  
            RegisterRoutes(RouteTable.Routes);  
        }  
    }  
}
```

Observatie:

Daca ruta *Default* ramane in Global.asax, atunci ruta *Admin* poate fi accesata de utilizatori externi.

ASP.NET MVC Controller (C#)

Fiecare cerere este mapata la un controller particular. Controller-ul este responsabil pentru generarea raspunsului.

Controllers\ProductController.cs

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
using System.Web.Mvc.Ajax;
```

```
namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Products/

        public ActionResult Index()
        {
            // Add action logic here
            return View();
        }
    }
}
```

Un controller este o clasa derivata din `System.Web.Mvc.Controller`.

Actiuni

Un controller expune actiuni ; actiunea este in fapt o metoda din cadrul clasei ce defineste controller-ul.

- actiune din controller trebuie sa fie metoda *publica* in clasa.
- metoda folosita ca actiune *nu poate fi supraincarcata si nu poate fi statica*.

Rezultatul actiunii – mostenite din clasa `ActionResult`

O actiune returneaza ceea ce se numeste rezultat actiune. Tipurile suportate ca rezultat al unei actiuni sunt :

1. **ViewResult** - Reprezinta HTML si markup.
2. **EmptyResult** – Reprezinta nici un rezultat.
3. **RedirectResult** – Reprezinta o redirectare la un nou URL.
4. **JsonResult** – Reprezinta un rezultat “JavaScript Object Notation” ce poate fi folosit intr-o aplicatie AJAX.
5. **JavaScriptResult** - Reprezinta un script JavaScript.
6. **ContentResult** – Reprezinta un rezultat text.
7. **FileContentResult** – Reprezinta un fisier downladabil cu continut binar.
8. **FilePathResult** - Reprezinta un fisier downladabil (cu cale).
9. **FileStreamResult** - Reprezinta un fisier downladabil (cu stream).

Controllers\BookController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
```



```
{  
    public class BookController : Controller  
    {  
        public ActionResult Index()  
        {  
            // Add action logic here  
            return View();  
        }  
    }  
}
```

Observatie:

Pentru rezultat se apeleaza de obicei o metoda din clasa de baza **Controller**.

1. **View** - Returns a ViewResult action result.
2. **Redirect** - Returns a RedirectResult action result.
3. **RedirectToAction** - Returns a RedirectToRouteResult action result.
4. **RedirectToRoute** - Returns a RedirectToRouteResult action result.
5. **Json** - Returns a JsonResult action result.
6. **JavaScriptResult** - Returns a JavaScriptResult.
7. **Content** - Returns a ContentResult action result.
8. **File** - Returns a FileContentResult, FilePathResult, or FileStreamResult depending on the parameters passed to the method.

Observatie:

Daca dorim sa redirectam utilizatorul de la o actiune dintr-un controller la o actiune din alt controller apelam metoda **RedirectToAction()**.

CustomerController.cs

```
using System.Web.Mvc;  
  
namespace MvcApplication1.Controllers  
{  
    public class CustomerController : Controller  
    {  
        public ActionResult Details(int? id)  
        {  
            if (!id.HasValue)  
                return RedirectToAction("Index");  
  
            return View();  
        }  
  
        public ActionResult Index()  
        {  
            return View();  
        }  
    }  
}
```

Observatie

- Putem folosi **ContentResults** pentru a returna rezultatul unei actiuni sub forma *plain text*.

Controllers\StatusController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class StatusController : Controller
    {
        public ActionResult Index()
        {
            return Content("Hello World!"); // nu returneaza View()
        }
    }
}
```

Observatie

- Daca o actiune din controller returneaza un rezultat ce nu e un “action result”, atunci rezultatul este transformat in **ContentResults** in mod automat.

WorkController.cs

```
using System;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class WorkController : Controller
    {
        public DateTime Index()
        {
            return DateTime.Now; // conversie la ContentResults
        }
    }
}
```

Creare “Action” in cadrul unui Controller (C#)

Se adauga o noua metoda in clasa derivata din **Controller**.

Listing 1 - Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public string SayHello()
        {
            return "Hello!";
        }
    }
}
```

Pentru ca o metoda sa fie *actiune* trebuiesc indeplinite urmatoarele conditii;

- Trebuie sa fie publica.
- Nu poate fi statica.
- Nu poate fi o metoda extinsa.
- Nu poate fi ctor, getter sau setter.
- Nu poate avea tipuri generice.
- Nu este o metoda a clasei de baza.
- Nu poate contine parametrii cu **ref** sau **out**.

Interzicerea unei metode publice de a fi invocata

Atributul [NonAction]

Controllers\WorkController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class WorkController : Controller
    {
        [NonAction]
        public string CompanySecrets()
        {
            return "This information is secret.";
        }
    }
}
```

ASP.NET MVC View (C#)

Ce sunt Views?

In ASP.NET MVC o pagina este un “view”- rezultatul unei actiuni.

HomeController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Details()
        {
            return RedirectToAction("Index");
        }
    }
}
```

Linia de cod de mai sus returneaza un view localizat la **\Views\Home\Index.cshtml**

Calea este obtinuta din numele controller-ului si din numele actiunii.

Observatie: Urmatorul cod returneaza un view cu numele Fred.

View(Fred);

iar calea corecta este:

|Views\Home\Fred.aspx

Adaugare continut la View

View este un document standard (X)HTML ce poate contine scripturi. Scripturile pot adauga continut dinamic la view.

\Views\Home\Index.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
```

```
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

        The current date and time is
        <% Response.Write(DateTime.Now) ;%>
    <br/>
```

sau echivalent

```
<br/>
<%=DateTime.Now %>

    </div>
</body>
</html>
```

“HTML Helpers” - generare continut View

Un « *Html Helper* » este o metoda ce genereaza un string. Putem folosi HTML Helpers pentru a genera elemente standard HTML : textbox, link, liste dropdown, listbox.

In ex. urmator se folosesc **BeginForm()**, **TextBox()** si **Password()**.

\Views\Home\Login.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form</title>
</head>
<body>
    <div>

        <% using (Html.BeginForm())
        { %>

            <label for="UserName">User Name:</label>
            <br />
            <%= Html.TextBox("UserName") %>

            <br /><br />

            <label for="Password">Password:</label>
            <br />
            <%= Html.Password("Password") %>
```

```

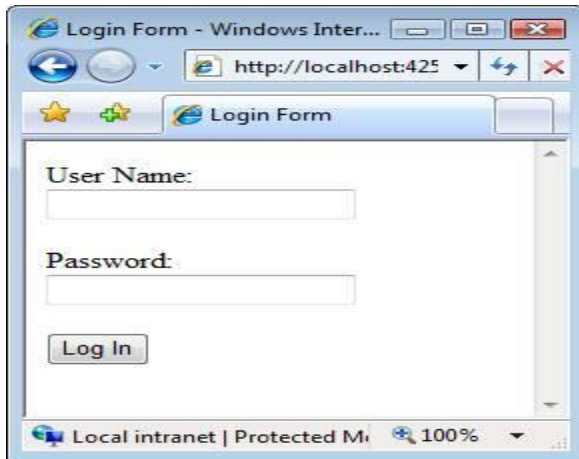
        <br /><br />

        <input type="submit" value="Log in" />

    <% } %>

</div>
</body>
</html>

```



Toate metodele HTML Helpers sunt apelate pe proprietatea `Html` a view-ului (de ex. `Html.TextBox()`, etc.).

Observatie:

Trebuie sa folosim `Response.Writer()` pentru a reda stringul in browser.

Folosirea HTML Helpers este optionala. Exemplul de mai sus fara HTML Helper.

\Views\Home\Login.aspx

```

<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Index4.aspx.cs" Inherits="MvcApp.Views.Home.Index4" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form without Help</title>
</head>
<body>
    <div>
        <form method="post" action="/Home/Login">
            <label for="userName">User Name:</label>
            <br />
            <input name="userName" />
            <br /><br />
            <label for="password">Password:</label>
            <br />
            <input name="password" type="password" />

```

```
                <br /><br />
                <input type="submit" value="Log In" />
            </form>
        </div>
    </body>
</html>
```

ASP.NET MVC include urmatoarele metode HTML Helpers (cele mai folosite):

- `Html.ActionLink()`
- `Html.BeginForm()`
- `Html.CheckBox()`
- `Html.DropDownList()`
- `Html.EndForm()`
- `Html.Hidden()`
- `Html.ListBox()`
- `Html.Password()`
- `Html.RadioButton()`
- `Html.TextArea()`
- `Html.TextBox()`

Pasare informatii la View

Putem folosi **ViewData**, **ViewBag**, **TempData** pentru a pasa Data la View. Toate informatiile (datele) pasate de la controller la View se trimit sub forma unui pachet.

ProductController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        public ActionResult Index()
        {
            ViewData["message"] = "Hello World!";
            return View();
        }
    }
}
```

Proprietatea **ViewData** reprezinta o colectie de perechi (nume, valoare).

Regasirea informatiei se face astfel:

`\Views\Product\Index.aspx`

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Product Index</title>
</head>
<body>
    <div>

        <%= Html.Encode(ViewData["message"]) %>

    </div>
</body>
</html>
```

Observatie

Oricand se reda continut trimis de user catre site web, trebuie folosita metoda `Html.Encode()` pentru a preveni atacuri cu JavaScript.

Adaugare continut dinamic la View

Un View prezinta utilizatorului parti ale modelului. Continutul dinamic este generat la runtime si depinde de cererea facuta.

Modalitati de daugare continut dinamic la View:

- **Cod inline:** bucati mici de cod inclus in View (if, foreach).
- **Metode helper HTML:** genereaza elemente HTML sau o colectie de acestea, in mod obisnuit bazate pe model sau valorile datelor din view.
- Utilizare **sectiuni** unde va fi inserat continut.
- **Vizualizari partiale:** utilizate pentru partajarea subsectiunilor unui view. Vizualizarile partiale pot contine cod inline, metode helper HTML, si referinte la alte vizualizari partiale. Vizualizarile partiale nu invoca o actiune (metoda din controller) si deci nu pot fi utilizate pentru a descrie logica modelului.
- **Actiuni pe elemente descendente:** utilizate pentru a crea controale UI reutilizabile si pot contine elemente de logica aplicatiei. Acestea invoca o actiune din controller, genereaza un view si plaseaza rezultatul in stream-ul de raspuns.

Exemplificam posibilitatile de mai sus pe baza unui exemplu. Construiesc un proiect ASP.NET MVC in care fac urmatoarele modificari:

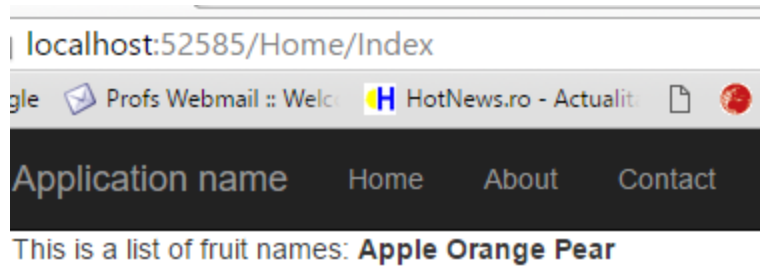
In *HomeController*:

```
public ActionResult Index()
{
    string[] names = { "Apple", "Orange", "Pear" };
    return View(names);
}
```

Index.cshtml - exemplificare *cod inline*

```
@model string[]
@{
    ViewBag.Title = "Index";
}
This is a list of fruit names:
@foreach (string name in Model)
{
    <span><b>@name</b></span>
}
```

Rezultatul este:



Utilizare sectiuni

Razor suporta conceptul de sectiuni, ce permite sa furnizam regiuni de continut in interiorul unui layout. Exemplu de definire sectiuni in view. Continut Index.cshtml din ~/Views/Home/Index:

```
@model string[]
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@section Header
{
    <div class="view">
        @foreach (string str in new [] {"Home", "List", "Edit"})
        {
            @Html.ActionLink(str, str, null, new { style =
                "margin: 5px" })
        }
    </div>
}
<div class="view">
    This is a list of fruit names:
    @foreach (string name in Model)
    {
        <span><b>@name</b></span>
    }
</div>
@section Footer
{
    <div class="view">
        This is the footer
    </div>
}
```

Pentru definire sectiune se foloseste tag-ul **@section** urmat de numele sectiunii.

O sectiune contine markup HTML si tag-uri Razor. Sectiunile sunt definite in view, dar aplicate intr-un layout cu metoda helper **@RenderSection**.Urmatorul exemplu descrie acest layout. Continutul fisierului */Views/Shared/_Layout.cshtml* este:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")
    <div class="layout">
        This is part of the layout
    </div>
    @RenderBody()
    <div class="layout">
        This is part of the layout
    </div>
    @RenderSection("Footer")
    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

Cand Razor parseaza layout-ul, metoda **RenderSection** este inlocuita cu continutul sectiunii din view. Identificarea se face dupa numele sectiunii. Parti ale vizualizarii ce nu sunt continute in sectiuni sunt inserate in vizualizare folosind metoda helper **RenderBody**.

Un view poate defini numai sectiuni ce sunt referite in layout. Se va genera o exceptie in cazul cand definim o sectiune in view si nu exista **RenderSection** in layout pentru acea sectiune.

Observatie: Se recomanda definirea sectiunilor fie la inceputul sau la sfarsitul view-ului pentru a putea vedea mai usor regiunea de continut a view-ului redada de **RenderBody**. O varianta poate fi (Index.cshtml):

```
@model string[]
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

```

}
@section Header
{
    <div class="view">
    @foreach (string str in new [] {"Home", "List", "Edit"})
    {
        @Html.ActionLink(str, str, null, new { style =
        "margin: 5px" })
    }
    </div>
}
@section Body
{
    <div class="view">
        This is a list of fruit names:
    @foreach (string name in Model)
    {
        <span><b>@name</b></span>
    }
    </div>
}
@section Footer
{
    <div class="view">
        This is the footer
    </div>
}

```

In acest caz **RenderBody** este inlocuit cu **RenderSection**.

RenderSection("Body") in fisierul *_Layout.cshtml*

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")
    <div class="layout">

```

```

        This is part of the layout
    </div>
    @RenderSection("Body")
    <div class="layout">
        This is part of the layout
    </div>
    @RenderSection("Footer")
    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>

```

Teste pentru sectiuni

Putem verifica daca o sectiune este definita intr-un view folosind metoda helper **IsSectionDefined**.

In *_Layout.cshtml* putem scrie:

```

...
@if (IsSectionDefined("Footer"))
{
    @RenderSection("Footer")
} else {
    <h4>This is the default footer</h4>
}
...

```

Redare optionala a sectiunilor

Implicit un view contine toate sectiunile pentru care este apelata **RenderSection** in layout, in caz contrar generandu-se o exceptie. In *_Layout.cshtml* vedeti ce e cu rosu. Aceasta sectiune nu exista in view.

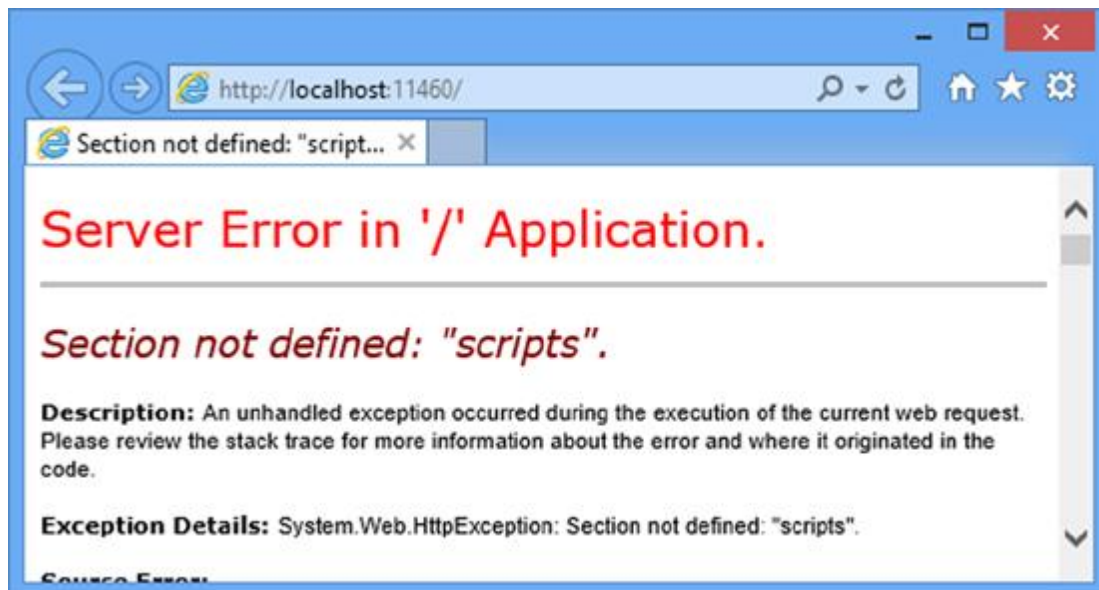
```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>

```

```
<title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")
    <div class="layout">
        This is part of the layout
    </div>
    @RenderSection("Body")
    <div class="layout">
        This is part of the layout
    </div>
    @if (IsSectionDefined("Footer")) {
        @RenderSection("Footer")
    } else {
        <h4>This is the default footer</h4>
    }
    @RenderSection("scripts")
    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

Rezultatul este



Pentru a înălța această excepție putem folosi o altă definiție a metodei `RenderSection` și anume:

...

```
@RenderSection("scripts", false)
```

...

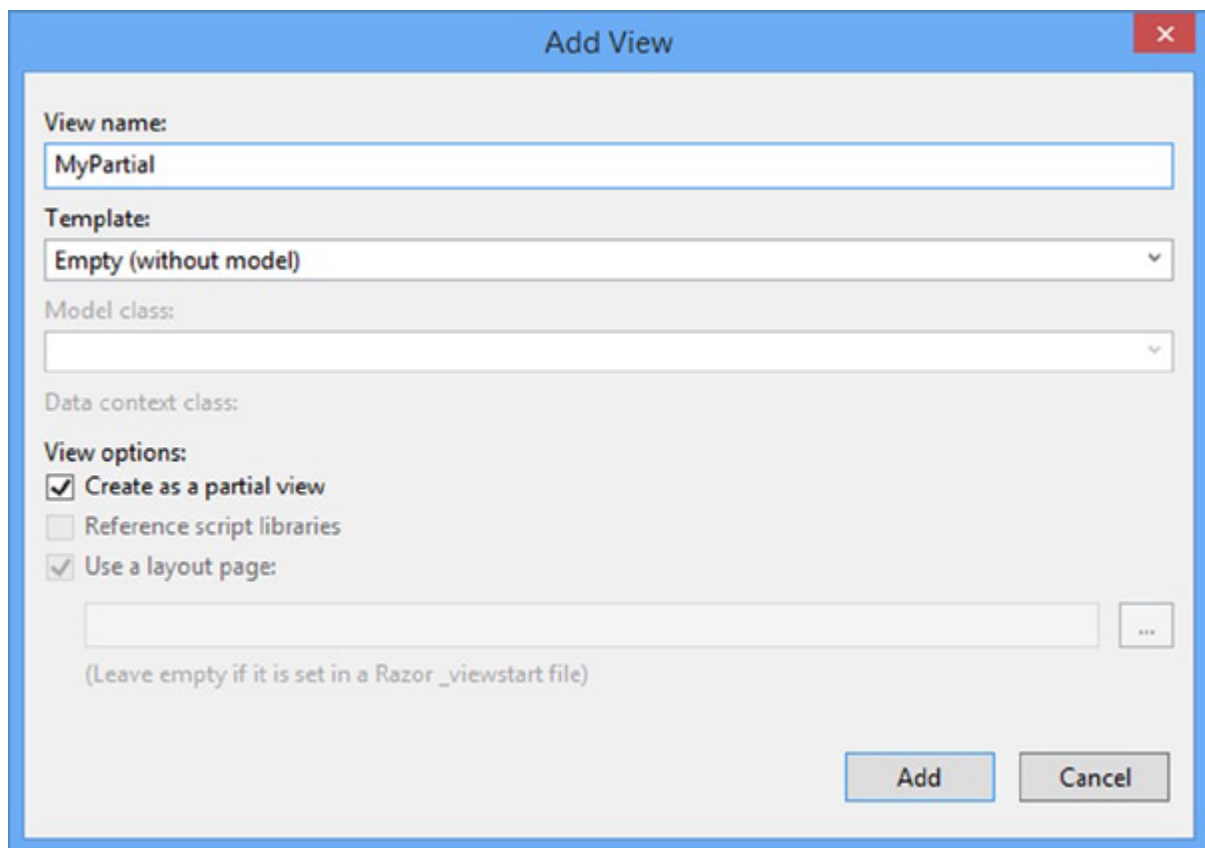
Continutul va fi inserat numai daca aceasta sectiune este definita.

Utilizare vizualizari partiale – Html.Partial()

Partial views sunt fisiere separate (pentru view) ce contin fragmente de tag-uri si markup ce poate fi inclus in alte view-uri. Exemplu arata cum transferam date la o vizualizare partiala.

Creare vizualizare partiala

Cream o vizualizare partiala in folderul Views/Shared, numita *MyPartialView*.



Continutul fisierului *MyPartial.cshtml*

```
<div>  
    This is the message from the partial view.  
    @Html.ActionLink("This is a link to the Index action",  
        "Index")  
</div>
```

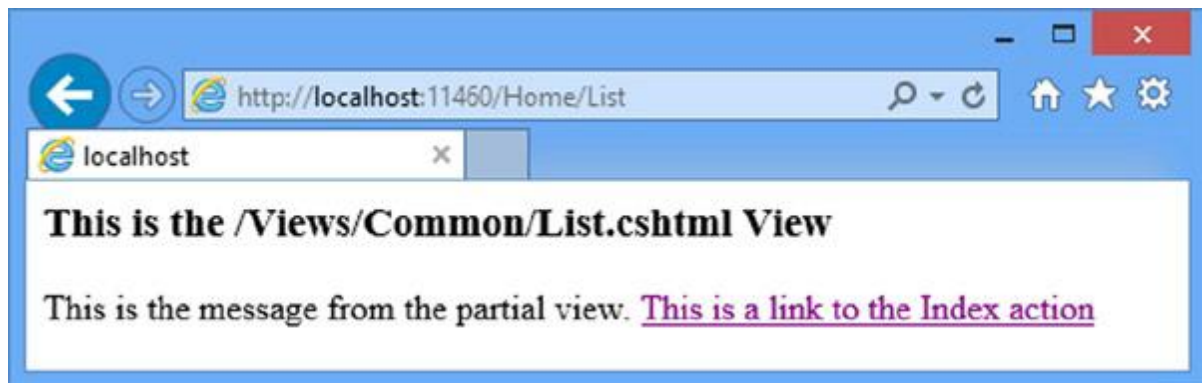
O vizualizare partiala este consumata apeland metoda helper **Html.Partial()** din interiorul altui view.

```
@{
    ViewBag.Title = "List";
    Layout = null;
}
<h3>This is the /Views/Common/List.cshtml View</h3>
@Html.Partial("MyPartial")
```

View este cautat in Views/controller_name/ si apoi in ~/Views/Shared.

Atentie **Layout = null**.

Action Index definta in vizualizarea partiala este din controllerul de unde s-a apelat aceasta vizualizare, in cazul nostru **HomeController**.



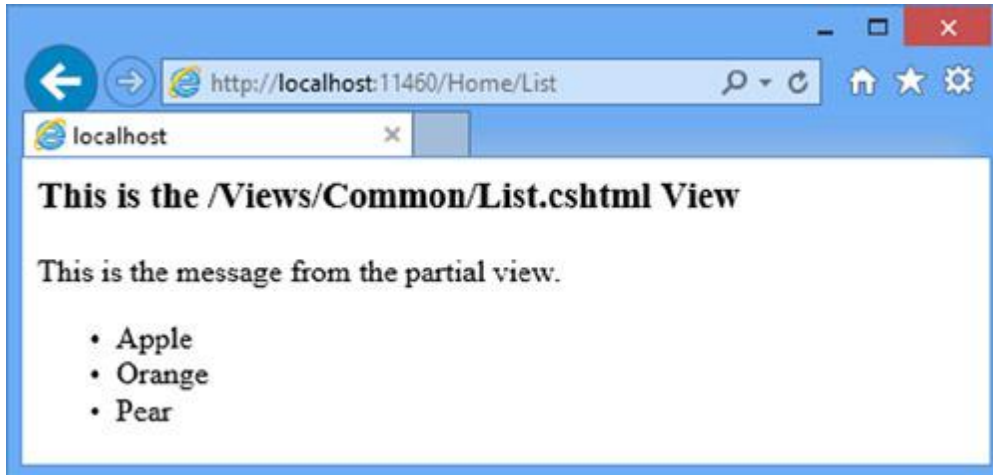
Utilizare vizualizari partiale puternic tipizate

Cream o vizualizare partiala numita *MyStronglyTypedPartial.cshtml*

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

Consumarea acestei vizualizari se face adaugand intr-un view apelul metodei **Html.Partial** ce contine parametri de aceasta data.


```
@{
    ViewBag.Title = "List";
    Layout = null;
}
<h3>This is the /Views/Common/List.cshtml View</h3>
@Html.Partial("MyStronglyTypedPartial", new [] {"Apple",
"Orange", "Pear"})
```



Utilizare actiuni descendente (Child Actions)

Child actions sunt actiuni invocate din cadrul unui view. Putem folosi aceasta tehnica cand dorim sa afisam anumite date pe pagini multiple. Se foloseste atributul **ChildActionOnly**, ceea ce inseamna ca metoda poate apelata numai din interiorul altui view.

Creare Child Action

Adaugam o actiune, *Time*, in HomeController:

```
using System;
using System.Web.Mvc;
namespace WorkingWithRazor.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }
        public ActionResult List()
        {

```

```

        return View();
    }
    [ChildActionOnly]
    public ActionResult Time()
    {
        return PartialView(DateTime.Now);
    }
}

```

Cream Time.cshtml

```

@model DateTime
<p>The time is: @Model.ToShortTimeString()</p>

```

Redare Child Action

Intr-un view apela `@Html.Action("Time")`

```

@{
    ViewBag.Title = "List";
    Layout = null;
}
<h3>This is the /Views/Common/List.cshtml View</h3>
@Html.Partial("MyStronglyTypedPartial", new [] {"Apple",
"Orange", "Pear"})
@Html.Action("Time")

```



Observatie

Pentru a apela metode din alt controller folosim **@Html.Action("Time", "MyController")**

Putem pasa un parametru la metoda:

```
...
@Html.Action("Time", new { time = DateTime.Now })
...
cu conditia evidenta ca actiunea din controller sa contina parametrul formal.
```

```
[ChildActionOnly]
public ActionResult Time(DateTime time)
{
    return PartialView(time);
}
...
```

Metode Helper

Metodele helper au fost construite pentru a emula tag-uri HTML.

Construim urmatorul exemplu

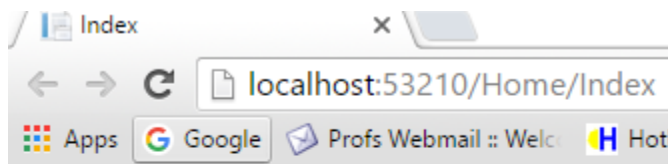
```
using System.Web.Mvc;
namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] { "Apple", "Orange",
            "Pear" };
            ViewBag.Cities = new string[] { "New York",
            "London",
            "Paris" };
            string message = "This is an HTML element:
            <input>";
            return View((object)message);
        }
    }
}
```

//Index.cshtml

```
@model string
@{
    Layout = null;
}
<!DOCTYPE html>
```

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
    Here are the fruits:
    @foreach (string str in (string[])ViewBag.Fruits) {
      <b>@str </b>
    }
  </div>
  <div>
    Here are the cities:
    @foreach (string str in (string[])ViewBag.Cities) {
      <b>@str </b>
    }
  </div>
  <div>
    Here is the message:
    <p>@Model</p>
  </div>
</body>
</html>
```

La executie



Here are the fruits: **Apple Orange Pear**
Here are the cities: **New York London Paris**
Here is the message:

This is an HTML element: **<input>**

Create metode helper personalizate

Create metoda helper inline

In Index.cshtml

```
@model string
@{
    Layout = null;
}
@helper ListArrayItems(string[] items)
{
    foreach(string str in items)
    {
        <b>@str </b>
    }
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits: @ListArrayItems(ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @ListArrayItems(ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

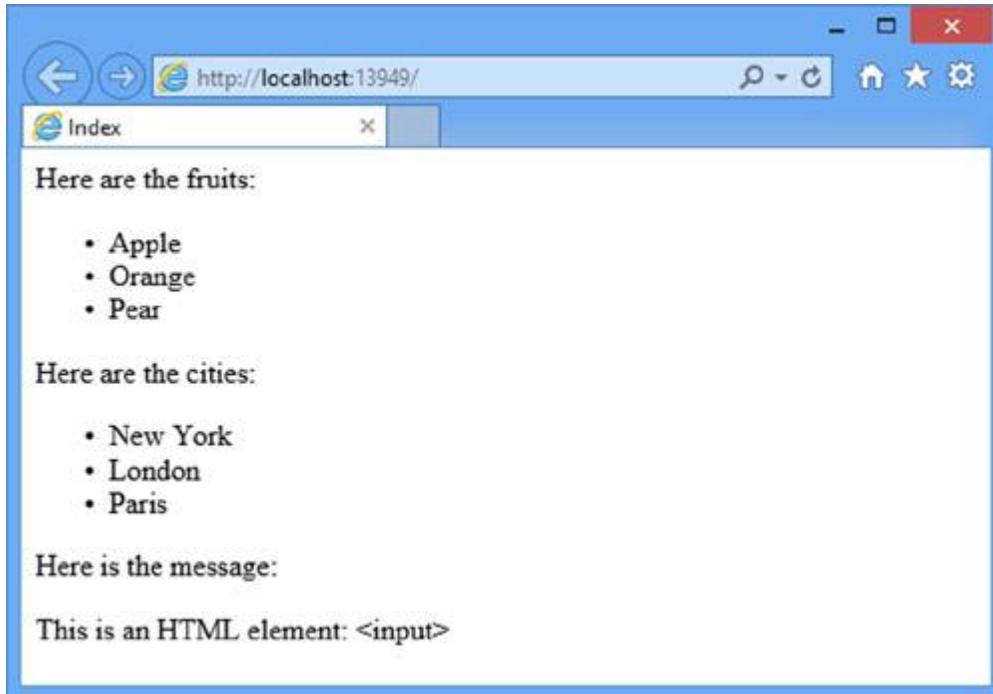
Metoda helper creata se numeste **ListArrayItems** ce are ca parametru un array de tip string.

Modul de afisare al elementelor in view este dat in corpul metodei. Putem modifica metoda astfel:

```
...
@helper ListArrayItems(string[] items)
{
    <ul>
        @foreach(string str in items)
        {
            <li>@str</li>
        }
    </ul>
}
```

```
}  
...
```

si view redat devine:



Creare metoda helper externa

Metodele helper inline sunt folosite numai in view unde au fost definite.

Metodele helper externe trebuie sa defineasca tag-urile folosite din HTML. Vezi codul de mai jos. Este in fapt o metoda extinsa pentru tipul **HtmlHelper**.

```
using System.Web.Mvc;  
namespace HelperMethods.Infrastructure  
{  
    public static class CustomHelpers  
    {  
        public static MvcHtmlString ListArrayItems(this  
            HtmlHelper html, string[] list)  
        {  
            TagBuilder tag = new TagBuilder("ul");  
            foreach(string str in list)  
            {  
                TagBuilder itemTag = new TagBuilder("li");  
                itemTag.SetInnerText(str);  
            }  
            return tag.ToString();  
        }  
    }  
}
```

```

        tag.InnerHtml += itemTag.ToString();
    }
    return new MvcHtmlString(tag.ToString());
}
}
}

```

Proprietati utile din HtmlHelper

Property	Description
RouteCollection	Returns the set of routes defined by the application
ViewBag	Returns the view bag data passed from the action method to the view that has called the helper method
ViewContext	Returns a ViewContext object, which provides access to details of the request and how it has been handled

Clasa **ViewContext** expune proprietati utile:

Property	Description
Controller	Returns the controller processing the current request
HttpContext	Returns the HttpContext object that describes the current request
IsChildAction	Returns true if the view that has called the helper is being rendered by a child action
RouteData	Returns the routing data for the request
View	Returns the instance of the IView implementation that has called the helper method

Putem folosi metode helper preconstruite pentru generarea continutului specific unei cereri.

In exemplul de mai sus am folosit clasa **TagBuilder** din care redam cei mai importanti membri ai clasei.

Member	Description
InnerHtml	A property that lets you set the contents of the element as an HTML string. The value assigned to this property will <i>not be encoded</i> , which means that it can be used to nest HTML elements.
SetInnerText(string)	Sets the text contents of the HTML element. The string parameter <i>is encoded</i> to make it safe to display.
AddCssClass(string)	Adds a CSS class to the HTML element
MergeAttribute(string,string, bool)	Adds an attribute to the HTML element. The first parameter is the name of the attribute, and the second is the value. The bool parameter specifies if an existing attribute of the same name should be replaced.

Rezultatul unei metode HTML helper este un obiect **MvcHtmlString**, al carui continut va fi scris direct in raspunsul catre client.

```
...  
return new MvcHtmlString(tag.ToString());  
...
```

Aceasta clasa este folosita pentru a genera tag-urile descrise si textul corespunzator.
Folosire metode helper externe create de dezvoltator (custom)

In view unde folosim metoda helper creata trebuie sa indicam si *spatiul de nume* unde a fost creata. Vezi exemplu.

```
@model string  
@using HelperMethods.Infrastructure  
@{  
    Layout = null;  
}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        Here are the fruits:  
        @Html.ListArrayItems((string[])ViewBag.Fruits)  
    </div>  
    <div>  
        Here are the cities:  
        @Html.ListArrayItems((string[])ViewBag.Cities)  
    </div>  
    <div>  
        Here is the message:  
        <p>@Model</p>  
    </div>  
</body>  
</html>
```

Unde folosim metodele helper?

Le folosim pentru a simplifica modalitatea de afisare a continutului. Pentru markup complex se recomanda folosirea vizualizarilor partiale.

Gestionarea codficarii stringurilor intr-o metoda helper

Urmatorul cod in *Index* action (HomeController). De urmarit acest exemplu pentru a vedea ce erori putem face.

```
public ActionResult Index()
{
    ViewBag.Fruits = new string[] { "Apple", "Orange", "Pear" };
    ViewBag.Cities = new string[] { "New York", "London", "Paris" };
    string message = "This is an HTML element: <input>";
    return View((object)message);
}
```

Custom helper

```
using System;
using System.Web.Mvc;
namespace HelperMethods.Infrastructure
{
    public static class CustomHelpers
    {
        public static MvcHtmlString ListArrayItems(this HtmlHelper html,
            string[] list)
        {
            TagBuilder tag = new TagBuilder("ul");
            foreach (string str in list)
            {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }
            return new MvcHtmlString(tag.ToString());
        }
        public static MvcHtmlString DisplayMessage(this HtmlHelper html,
            string msg)
        {
            string result = String.Format("This is the message: <p>{0}</p>",
                msg);
            return new MvcHtmlString(result);
        }
    }
}
```

Observatie

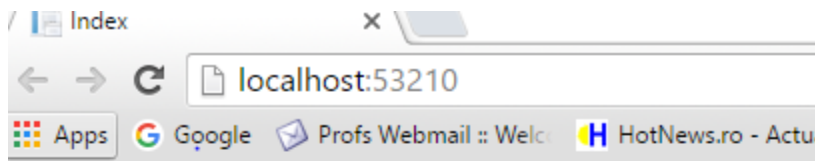
Vedeti modul de formatare al stringului in metoda **DisplayMessage**

Continut Index.cshtml

```
@model string
@using HelperMethods.Infrastructure
@{
    Layout = null;
}
```

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <p>This is the content from the view:</p>
  <div style="border: thin solid black; padding: 10px">
    Here is the message:
    <p>@Model</p>
  </div>
  <p>This is the content from the helper method:</p>
  <div style="border: thin solid black; padding: 10px">
    @Html.DisplayMessage(Model)
  </div>
</body>
</html>
```

si rezultatul afisarii este:



This is the content from the view:

Here is the message:

This is an HTML element: <input>

This is the content from the helper method:

This is the message:

This is an HTML element:

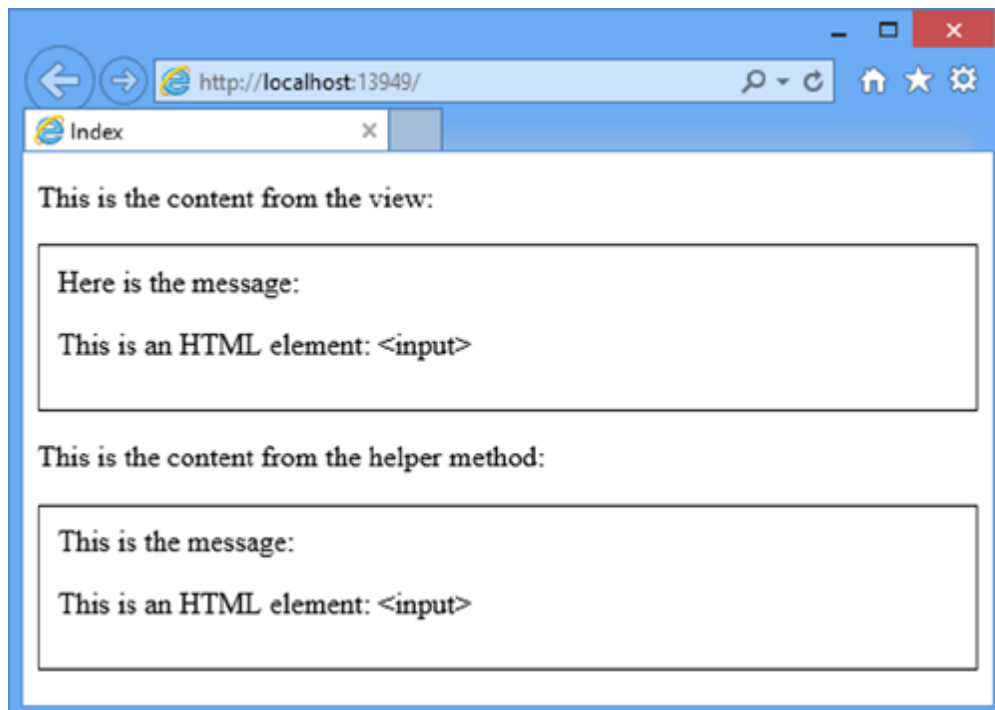
A aparut un element de tip *input*.

Modalitati de rezolvare

1. Modificam tipul returnat de **DisplayMessage**. Va fi string si nu **MvcHtmlString**.
2. Folosirea metodei **Encode**.

Codul mai bun (recomandat) in **DisplayMessage** este:

```
...
public static MvcHtmlString DisplayMessage(this HtmlHelper html,
string msg) {
    string encodedMessage = html.Encode(msg);
    string result = String.Format("This is the message:
    <p>{0}</p>", encodedMessage);
    return new MvcHtmlString(result);
}
...
```



Utilizare metoda helper preconstruita pentru Form

Creare elemente de tip Form - exemplu

Pentru demonstratie consideram urmatorul model:

```
using System;
namespace HelperMethods.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
    }
}

```

Actiunea Index din HomeController

```

using System.Web.Mvc;
using HelperMethods.Models;
namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] { "Apple",
            "Orange", "Pear" };
            ViewBag.Cities = new string[] { "New York",
            "London", "Paris" };
            string message = "This is an HTML element:
            <input>";
            return View((object)message);
        }

        public ActionResult CreatePerson()
        {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person)
        {

```

```

        return View(person);
    }
}

```

Atentia ne este indreptata asupra modului de generare al elementelor in view.

CreatePerson.cshtml

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
</form>

```

Observatie

Fiecare element are atributul **name** definit, atribut folosit de MVC in procesul de binding.

Fisierul *_Layout.cshtml* are continutul

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
    </style>

```

```

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

```

Rezultatul este



Creare elemente form fara a folosi tag **<form />**

In exemplul de mai sus am folosit tag-ul **<form/>**.

Vom folosi in continuare o metoda helper din Html.

Metoda este **Html.BeginForm()** ce prezinta mai multe prototipuri (13 variante).

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@{Html.BeginForm();}
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>

```

```

        </div>
        <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
        </div>
        <input type="submit" value="Submit" />
@{Html.EndForm();}

```

Varianta acceptata este cea data mai jos in care se vede clar continutul unui form, plus ca se apeleaza Dispose() in mod automat.

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginForm())
{
    <div class="dataElem">
<label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}

```

Supraincarcari ale metodei helper BeginForm

Overload	Description
BeginForm()	Creates a form which posts back to the action method it originated from
BeginForm(action, controller)	Creates a form which posts back to the action method and controller, specified as strings
BeginForm(action, controller, method)	As for the previous overload, but allows you to specify the value for the method attribute using a value from the System.Web.Mvc.FormMethod enumeration

BeginForm(action, controller, method, attributes) As for the previous overload, but allows you to specify attributes for the form element an object whose properties are used as the attribute names

BeginForm(action, controller, routeValues, method, attributes) As for the previous overload, but allows you to specify values for the variable route segments in your application routing configuration as an object whose properties correspond to the routing variables

Exemplu

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using (Html.BeginForm("CreatePerson", "Home",
new { id = "MyIdValue" }, FormMethod.Post))
{
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

Tag-ul HTML generat de acest cod este:

```
...
<form action="/Home/CreatePerson/MyIdValue" class="personClass"
data-formType="person" method="post">
...
```

Specificare ruta folosita de form

Daca vrem sa modificam ruta existenta putem folosi metoda helper **BeginRouteForm**.

In RouteConfig.cs exista urmatorul cod:

```
using System;
```



```

using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace HelperMethods
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
            );
            routes.MapRoute(
                name: "FormRoute",
                url: "app/forms/{controller}/{action}"
            );
        }
    }
}

```

Cream alta ruta in CreatePerson.cshtml. Ruta **FormRoute** o utilizam in codul ce urmeaza.

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
}

```

```

    </div>
    <div class="dataElem">
    <label>Last Name</label>
    <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}

```

Rezultatul este:

```

<form action="/app/forms/Home/CreatePerson" class="personClass"
data-formType="person" method="post">
...

```

Observatie

BeginRouteForm prezinta mai multe supraincari (prototipuri).

Utilizare helper pentru intrare (introducere date)

Exemple de metode helper folosite pentru preluare date

Check box **Html.CheckBox**("myCheckbox", false)

Output:

```

<input id="myCheckbox" name="myCheckbox" type="checkbox" value="true" />

```

```

<input name="myCheckbox" type="hidden" value="false" />

```

Hidden field **Html.Hidden**("myHidden", "val")

Output:

```

<input id="myHidden" name="myHidden" type="hidden" value="val" />

```

Radio button **Html.RadioButton**("myRadiobutton", "val", true)

Output:

```

<input checked="checked" id="myRadiobutton" name="myRadiobutton"
type="radio" value="val" />

```

Password **Html.Password**("myPassword", "val")

Output:

```

<input id="myPassword" name="myPassword" type="password" value="val" />

```

Text area **Html.TextArea**("myTextarea", "val", 5, 20, null)

Output:

```

<textarea cols="20" id="myTextarea" name="myTextarea" rows="5">
val</textarea>

```

Text box **Html.TextBox**("myTextbox", "val")

Output:

```

<input id="myTextbox" name="myTextbox" type="text" value="val" />

```

Fiecare din aceste metode este supraincercata.

Exemplu de utilizare

@model HelperMethods.Models.Person

```
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
new { @class = "personClass", data_formType="person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("personId", @Model.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName", @Model.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName", @Model.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```

Generare element de intrare dintr-o proprietate a modelului

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
new { @class = "personClass", data_formType="person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
    </div>
}
```

```

        @Html.TextBox("lastName")
    </div>
    <input type="submit" value="Submit" />
}

```

Argumentul de la **Html.TextBox** este cautat in ViewBag si apoi in @Model.DataValue
Prima valoare gasita este cea folosita.

Utilizare helper pentru intrare, puternic tipizati

Se folosesc expresii lambda. Mai puțin expus la erori de codare. Exemplu

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
new { @class = "personClass", data_formType="person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <input type="submit" value="Submit" />
}

```

Creare elemente ce contin liste de valori

```

Drop-down list Html.DropDownList("myList", new SelectList(new []
{"A", "B"}), "Choose")

```

Output:

```

<select id="myList" name="myList">
    <option value="">Choose</option>
    <option>A</option>

```

```
<option>B</option>
</select>
```

Drop-down list `Html.DropDownListFor(x => x.Gender, new SelectList(new [] { "M", "F" }))`

Output:

```
<select id="Gender" name="Gender">
  <option>M</option>
  <option>F</option>
</select>
```

Multiple-select `Html.ListBox("myList", new MultiSelectList(new [] { "A", "B" }))`

Output:

```
<select id="myList" multiple="multiple" name="myList">
  <option>A</option>
  <option>B</option>
</select>
```

Multiple-select `Html.ListBoxFor(x => x.Vals, new MultiSelectList(new [] { "A", "B" }))`

Output:

```
<select id="Vals" multiple="multiple" name="Vals">
  <option>A</option>
  <option>B</option>
</select>
```

Creare element de tip select pentru proprietatea *Person.Role* in fisierul *CreatePerson.cshtml*

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
new { @class = "personClass", data_formType="person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
}
```

```

<div class="dataElem">
    <label>Role</label>
    @Html.DropDownListFor(m => m.Role,
        new
        SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
</div>
<input type="submit" value="Submit" />
}

```

Metode helper – sabloane

În metodele helper preconstruite și bazate pe model specificăm proprietatea ce dorim să o afișăm și MVC va alege elementul corespunzător.

Exemplu – folosim același model ca mai înainte

În *HomeController* există două metode *CreatePerson* și ambele folosesc *CreatePerson.cshtml*.

@model HelperMethods.Models.Person

@{

```

    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);

```

}

<h2>CreatePerson</h2>

```

@using (Html.BeginRouteForm("FormRoute", new { },
    FormMethod.Post,

```

```

new { @class = "personClass", data_formType = "person" }))
{

```

```

    <div class="dataElem">

```

```

        <label>PersonId</label>

```

```

        @Html.TextBoxFor(m => m.PersonId)

```

```

    </div>

```

```

    <div class="dataElem">

```

```

        <label>First Name</label>

```

```

        @Html.TextBoxFor(m => m.FirstName)

```

```

    </div>

```

```

    <div class="dataElem">

```

```

        <label>Last Name</label>

```

```

        @Html.TextBoxFor(m => m.LastName)

```

```

    </div>

```

```

    <div class="dataElem">

```

```

        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,
            new
            SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}

```

Utilizare metode helper

Ne îndreptăm atenția asupra metodelor **Html.Editor** și **Html.EditorFor**. Metoda **Editor** are ca parametru un string ce reprezintă proprietatea unui obiect din model.

```

@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.Editor("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.Editor("FirstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.EditorFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.EditorFor(m => m.Role)
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
        @Html.EditorFor(m => m.BirthDate)
    </div>
}

```

```

    <input type="submit" value="Submit" />
}

```

The screenshot shows a web browser window with the address bar displaying `http://localhost:13949/Home/CreatePerson`. The page title is "CreatePerson". The form contains the following fields:

- PersonId:
- First Name:
- Last Name:
- Role:
- Birth Date:

A "Submit" button is located at the bottom left of the form.

Metode helper - sabloane

Helper	Example	Description
Display	Html.Display("FirstName")	Renders a read-only view of the specified model property, choosing an HTML element according to the property's type and metadata
DisplayFor	Html.DisplayFor(x => x.FirstName)	Strongly typed version of the previous helper
Editor	Html.Editor("FirstName")	Renders an editor for the specified model property, choosing an HTML element according to the property's type and metadata
EditorFor	Html.EditorFor(x => x.FirstName)	Strongly typed version of the previous helper
Label	Html.Label("FirstName")	Renders an HTML <label> element referring to the specified model property
LabelFor	Html.LabelFor(x => x.FirstName)	Strongly typed version of the previous helper

Utilizare

MVC definește metode helper ce operează pe obiecte în întregime (se are în vedere toate proprietățile obiectelor), proces numit *scaffolding*.

Helper	Example	Description
DisplayForModel	<code>Html.DisplayForModel()</code>	Renders a read-only view of the entire model object
EditorForModel	<code>Html.EditorForModel()</code>	Renders editor elements for the entire model object
LabelForModel	<code>Html.LabelForModel()</code>	Renders an HTML <label> element referring to the entire model object

Exemplu cu LabelForModel si EditorForModel

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson: @Html.LabelForModel()</h2>
@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
new { @class = "personClass", data_formType="person"}))
{
    @Html.EditorForModel()
    <input type="submit" value="Submit" />
}
}
```

Nu intotdeauna acest mod de lucru creaza vizualizari “corecte” in sensul ca tot ce e vizibil nu e mereu de folos. Proprietatea *Role* ar trebui sa fie prezentata ca o selectie si nu ca un element input.

O alta problema consta in aceea ca trebuie sa modificam CSS pentru o vizualizare corecta.

The screenshot shows a web browser window with the address bar displaying 'http://localhost:13949/app/forms/Home/CreatePerson'. The page title is 'CreatePerson'. The form content is as follows:

CreatePerson:

PersonId
0

FirstName
[text input]

LastName
[text input]

BirthDate
01/01/0001 00:00:00

IsApproved
☐

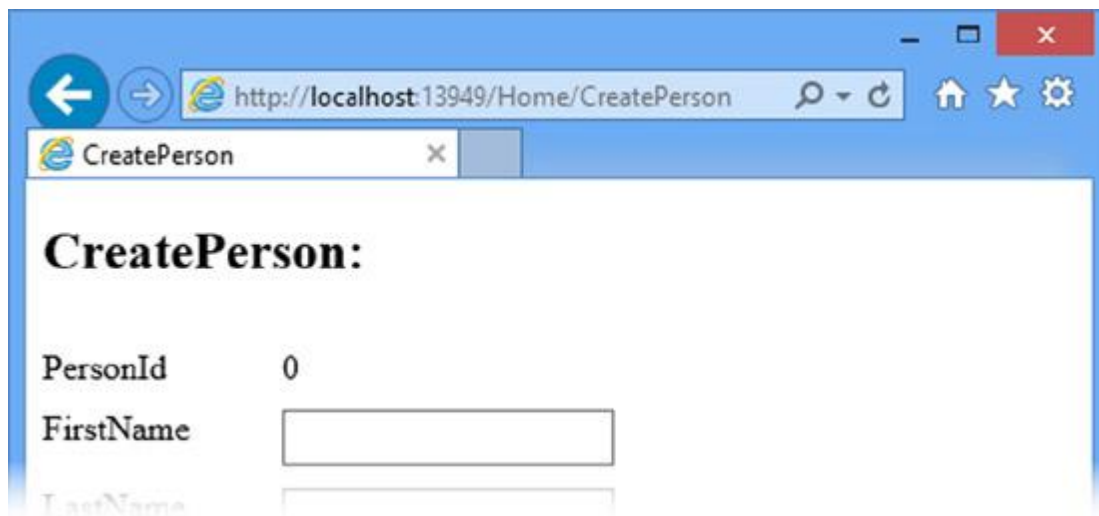
Role
Admin

Submit

Utilizare metadata model

Se folosesc attribute. **HiddenAttribute**.

```
using System;
using System.Web.Mvc;
namespace HelperMethods.Models {
public class Person
{
    [HiddenInput]
    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; }
}
// ...other types omitted for brevity...
}
```

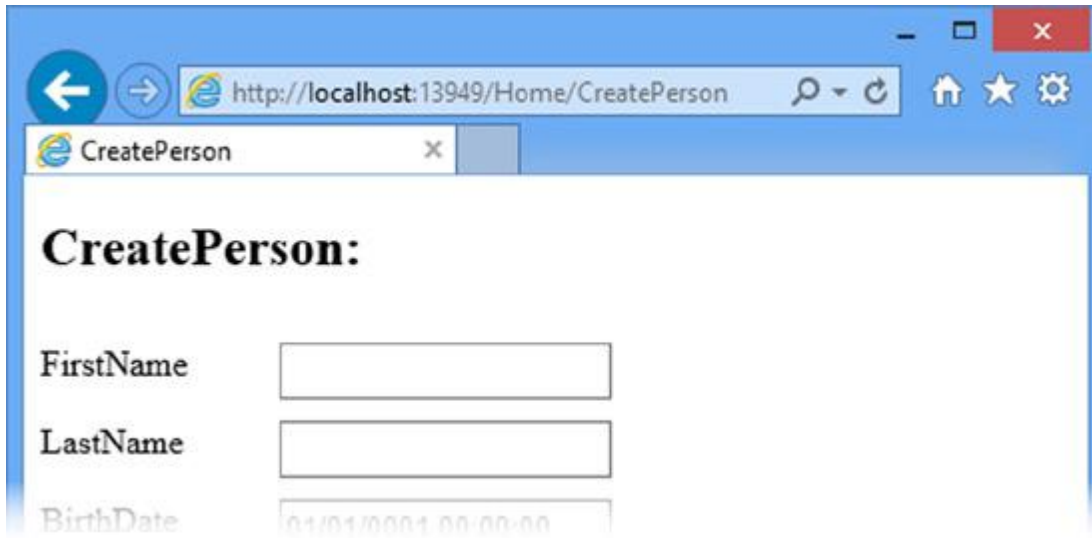


Creare proprietatea read only

Daca dorim ca PersonId sa nu fie afisat atunci ar trebuie sa scriem:

```
[HiddenInput(DisplayValue=false)]
public int PersonId { get; set; }
```

si vom obtine



The screenshot shows a web browser window with the address bar displaying 'http://localhost:13949/Home/CreatePerson'. The browser has a single tab titled 'CreatePerson'. The page content features a heading 'CreatePerson:' followed by three input fields. The first field is labeled 'FirstName', the second 'LastName', and the third 'BirthDate'. The 'BirthDate' field contains the text '01/01/0001 00:00:00'.

Excludere proprietate din scaffolding

Atributul folosit este **ScaffoldColumn(false)** aplicat la model.

```
...  
[ScaffoldColumn(false)]  
public int PersonId { get; set; }  
...
```

ASP 5 - MVC 6

Anatomia aplicatiei

Aplicatiile ASP.NET 5 sunt construite si ruleaza noul *.NET Execution Environment (DNX)*.

Fiecare proiect ASP.NET 5 este un proiect DNX.

ASP.NET 5 se integreaza cu DNX prin intermediul pachetului *ASP.NET Application Hosting*.

DNX = *.Net Execution Environment*. Mai multe detalii despre DNX si nu numai la adresa:

<https://docs.asp.net/en/latest/dnx/overview.html>

The new ASP.NET 5.0 or the .NET Full CLR enables you to utilize all the .NET components that are available and supports backward compatibility. The ASP.NET Core 5.0 or .NET Core CLR is a refactored version of .NET. It was redesigned to be modular which allows developers to plug components which are only required for your project and it is a cloud optimized runtime.

What is the .NET Execution Environment?

The .NET Execution Environment (DNX) is a software development kit (SDK) and runtime environment that has everything you need to build and run .NET applications for Windows, Mac and Linux. It provides a host process, CLR hosting logic and managed entry point discovery. DNX was built for running cross-platform ASP.NET Web applications, but it can run other types of .NET applications, too, such as cross-platform console apps.

Aplicatiile ASP.NET 5 sunt definite folosind clasa publica *Startup*:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }
    public void Configure(IApplicationBuilder app)
    {
    }
    public static void Main(string[] args) =>
        WebApplication.Run<Startup>(args);
}
```

```
}
```

Metoda *ConfigureServices* definește serviciile folosite de aplicație iar metoda *Configure* este folosită pentru a defini mijlocul de comunicare al cererilor.

Clasa *Startup* (totdeauna publică) furnizează punctul de intrare (entry point) pentru o aplicație și este necesară pentru toate aplicațiile. ASP.NET caută în assembly primar pentru o clasă cu acest nume. Putem specifica un assembly diferit unde este definită această clasă folosind cheia de configurare *Hosting:Application*. Vezi secțiunea [Configurare](#).

Servicii

Un serviciu este o componentă care este destinată a fi consumată în aplicație. Serviciile sunt disponibile prin DI (*Dependency Injection*). ASP.NET 5 include un container IoC pre construit ce suportă injectia prin constructor dar care poate fi înlocuit de un alt container IoC, conform preferințelor dezvoltatorului.

Serviciile în ASP.NET 5 vin în următoarele forme: *singleton*, *scoped* (vizibile într-un anumit domeniu), *transitorii* și de *instanță*.

Serviciile *transitorii* sunt create de fiecare dată când sunt cerute dintr-un container.

Serviciile *scoped* sunt create numai dacă nu există în domeniul de vizibilitate curent. Pentru aplicații Web, un container “scope” este creat pentru fiecare cerere, deci putem gândi aceste servicii ca fiind per cerere.

Serviciile *singleton* sunt create o singură dată pentru aplicație.

Pentru serviciile de *instanță* trebuie să cream noi instanțe tipului ce implementează serviciul.

Middleware

Middleware este o parte de cod ce poate procesa o cerere.

ASP.NET 5 vine cu *middleware* pre construit:

- Lucrul cu fișiere statice
- Rutarea
- Diagnostice
- Autentificare

În ASP.NET 5 cream cererea folosind *Middleware*. *Middleware* execută logica asincron pe un **HttpContext** și opțional poate invoca următorul *middleware* în secvență sau să

termine cererea in mod direct. In general folosim *middleware* prin invocarea unei metode extinse pe **IApplicationBuilder** in metoda *Configure*.

Metodele folosite pentru a adauga middleware sunt: Use, Map si Run. Metoda Run nu da controlul mai departe in lantul de tratare al cererii.

Servere

Modelul *ASP.NET Application Hosting* nu asculta in mod direct pentru cereri, dar in schimb se bazeaza pe un server HTTP si include suport pentru rulare pe IIS sau *self-hosting* intr-un proces propriu. In Windows putem gazdui aplicatia in afara IIS folosind serverul *WebListener* ce este bazat pe *HTTP.sys*. Putem de asemenea gazdui aplicatia intr-un mediu non-Windows folosind serverul web **Kestrel**.

Webroot

Radacina Web a aplicatiei este radacina locatiei in proiectul nostru, din care cererile HTTP sunt gestionate. Radacina Web a unei aplicatii ASP.NET 5 este configurata folosind proprietatea "webroot" din fisierul *project.json*.

Configurare

ASP.NET 5 foloseste un model de configurare nou ce nu mai e bazat pe *System.Configuration* sau *web.config* ci pe o multime ordonata de furnizori de configurare. Furnizorii de configurare preconstruiti suporta o varietate de formate de fisiere (XML, JSON, INI) si de asemenea variabile de mediu pentru a permite configurare bazata pe acestea.

O **configurare** este o simpla lista ierarhica de perechi *nume-valoare* in care nodurile sunt separate prin :

Pentru a lucra cu setari intr-o aplicatie ASP.NET, se recomanda instantierea clasei *Configuration* in clasa *Startup* si apoi folosirea patternului **Options** pentru a accesa setari individuale. Clasa *Configuration* o putem privi ca o colectie de Providers, ce furnizeaza abilitatea de a citi si scrie perechi nume/valoare. Trebuie sa configuram cel putin un provider pentru ca *Configuration* sa lucreze corect. Vezi si exemplul urmator.

```
// assumes using Microsoft.Framework.ConfigurationModel is specified
var builder = new ConfigurationBuilder();
builder.Add(new MemoryConfigurationProvider());
var config = builder.Build();
config.Set("somekey", "somevalue");

// do some other work

string setting2 = config["somekey"]; // also returns "somevalue"
```

In exemplul de mai jos se adauga furnizori, pentru configurare, de tip fisiere si variabile de mediu.

```
var builder = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json")
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

if (env.IsDevelopment())
{
    // For more details on using the user secret store see
    // http://go.microsoft.com/fwlink/?LinkID=532709
    builder.AddUserSecrets();
}
builder.AddEnvironmentVariables();
Configuration = builder.Build();
```

ASP.NET 5 este proiectat sa se integreze cu o varietate de framework-uri pe partea de client, cum ar fi: [AngularJS](#), [KnockoutJS](#) si [Bootstrap](#). Pentru detalii a se consulta *Client-Side Development*.

Pattern Options

Folosind *Options* putem converti orice clasa (sau POCO – Plain Old CLR Object) intr-o clasa pentru setari. Se recomanda sa respectam principiile *Interface Segregation Principle* (ISP) (clasele depind numai de setarile de configurare pe care le folosesc) si *Separation of Concerns* (setarile pentru parti disparate ale aplicatiei sunt gestionate separat).

Exemplu

```
public class MyOptions
{
    public string Option1 { get; set; }
    public int Option2 { get; set; }
}
```

Optiunile pot fi injectate in aplicatie folosind serviciul *IOptions<TOptions>*. De exemplu, urmatorul controller foloseste *IOptions<MyOptions>* pentru a accesa setarile necesare pentru a reda vizualizarea *Index*.

```
public class HomeController : Controller
{
```

```
public HomeController(IOptions<MyOptions> optionsAccessor)
{
    Options = optionsAccessor.Value;
}

MyOptions Options { get; }

// GET: /<controller>/
public IActionResult Index()
{
    return View(Options);
}
}
```

Pentru a seta *IOptions<TOptions>* trebuie sa adaugam acest serviciu, deci este necesar un apel de forma

```
services.AddOptions();
```

in cadrul metodei *ConfigureServices* definita in clasa *Startup*.

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options with DI
    services.AddOptions();

    // Configure MyOptions using config
    services.Configure<MyOptions>(Configuration);

    // Configure MyOptions using code
    services.Configure<MyOptions>(myOptions =>
    {
        myOptions.Option1 = "value1_from_action";
    });

    // Add framework services.
    services.AddMvc();
}
```

Cand asociem optiuni pentru a configura fiecare proprietate din *MyOptions* (de exemplu), tipul optiune (*MyOptions*) este asociat la o cheie de configurare de forma:

property:subproperty:...

De exemplu, proprietatea *MyOptions.Option1* este asociata la cheia *Option1* care este citita din proprietatea *option1* in *appsettings.json*. Cheile de configurare sunt case insensitive.

Fiecare apel la *Configure<TOption>* adauga un serviciu *IConfiguration<TOption>* in containerul de servicii ce este folosit de serviciul *IOptions<TOption>* pentru a furniza optiunile configurate. Daca dorim sa configuram optiunile altfel (citire din baza de date, etc.) se poate folosi metoda extinsa *ConfigureOptions<TOption>* pentru a specifica un serviciu custom *IConfigurations<TOption>* in mod direct.

Pot exista mai multe servicii *IConfigureOptions<TOption>* pentru acelasi tip de optiune si acestea sunt aplicate in ordinea in care au fost scrise. In exemplul de mai sus *Option1* si *Option2* sunt specificate in *appsettings.json*, dar valoarea lui *Option1* este suprascrisa de configurarea data de delegate:

```
services.Configure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "value1_from_action";
});
```

Exemplu:Setari folosind Entity Framework

Problema

Dorim sa memoram setari ppentru aplicatie intr-o baza de date si sa accesam baza de date folosind Entity Framework. Consideram ca avem o singura tabela ce contine doua coloane key/value. Definim entitatea:

```
public class ConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```

Definim contextul (clasa derivata din *DbContext*):

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) :
        base(options)
    {
    }
    public DbSet<ConfigurationValue> Values { get; set; }
}
```

Cream un furnizor pentru configurare, un tip derivat din *ConfigurationProvider*. Datele de configurare sunt incarcate prin suprascrierea metodei *Load()* ce va citi datele din baza de date. In codul ce urmeaza se face si o initializare a bazei de date in cazul cand aceasta nu a fost creata si populata cu date (se foloseste Code First).

```
public class EntityFrameworkConfigurationProvider : ConfigurationProvider
{
    public EntityFrameworkConfigurationProvider(
```

```

        Action<DbContextOptionsBuilder> optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<ConfigurationContext>();
        OptionsAction(builder);

        using (var dbContext = new ConfigurationContext(builder.Options))
        {
            dbContext.Database.EnsureCreated();
            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private IDictionary<string, string> CreateAndSaveDefaultValues(
        ConfigurationContext dbContext)
    {
        var configValues = new Dictionary<string, string>
        {
            { "key1", "value_from_ef_1" },
            { "key2", "value_from_ef_2" }
        };
        dbContext.Values.AddRange(configValues
            .Select(kvp => new ConfigurationValue()
            {
                Id = kvp.Key,
                Value = kvp.Value
            })
            .ToArray());
        dbContext.SaveChanges();
        return configValues;
    }
}

```

Mai trebuie sa adaugam si o metoda extinsa *AddEntityFramework* pentru a adauga furnizorul de configurare.

```

public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEntityFramework(
        this IConfigurationBuilder builder,
        Action<DbContextOptionsBuilder> setup)
    {
        return builder.Add(new EntityFrameworkConfigurationProvider(setup));
    }
}

```

Controller

În versiunile anterioare ale lui ASP.NET MVC, controller-ele MVC sunt diferite de controller-ele Web API. Un controller MVC folosește clasa de bază

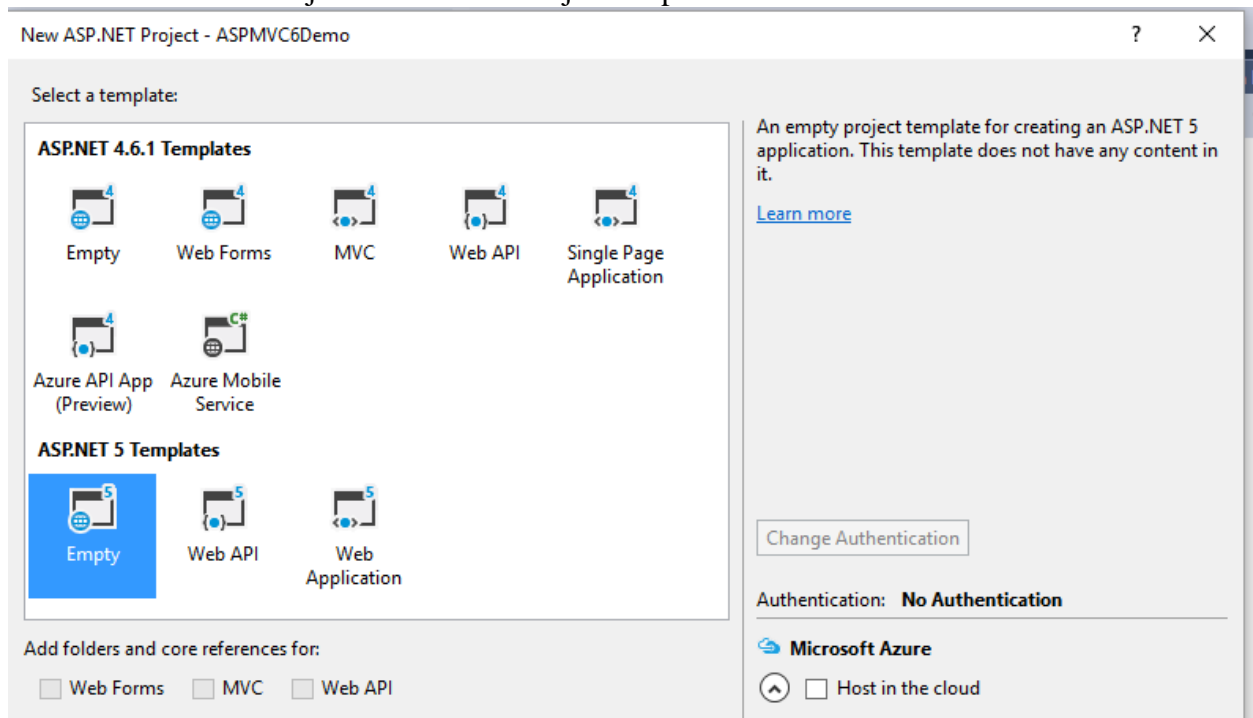
System.Web.Mvc.Controller și un controller Web API folosește clasa de bază **System.Web.Http.ApiController**.

În MVC 6, există o singură clasă de bază **Controller** atât pentru MVC cât și pentru Web API și aceasta este **Microsoft.AspNetCore.Mvc.Controller**.

S-a unificat și partea cu metodele helper HTML, privitoare la MVC și Web Pages.

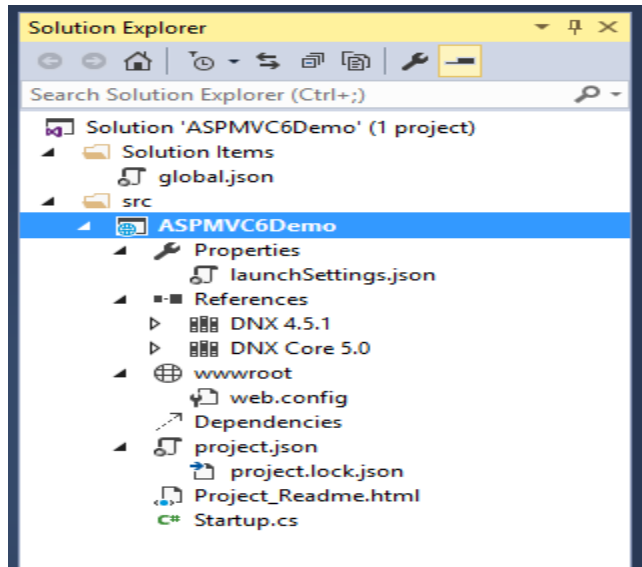
Creare proiect: nume ASPMVC6Demo

Meniu File->New->Project->ASP.NET Project și apoi



și în continuare clic pe OK

În Solution Explorer vom vedea ce s-a generat.



Descriere

- **folder src** – contine toate proiectele ce contin codul sursa pentru aplicatie.
- **global.json** – aici se pun setari la nivel de solutie si permite referinte la alte proiecte.
- **wwwroot** – este un folder in care vor fi plasate toate fisierele statice (HTML, CSS, Image si JavaScript). Aceste fisiere vor ajunge la client.
- **project.json** – contine setarile proiectului.
- **startup.cs** – contine codul pentru lansare si configurare aplicatie.

La *References* s-au creat doua foldere DNX 4.5.1 si DNX Core 5.0 care la randul lor contin o structura arborescenta de foldere si in final assemblies. Acestea sunt referinte la ASP.NET 5.0 si ASP.NET 5.0 Core.

Observam ca nu s-a generat structura de directoare pe care o stim de la MVC. Vom crea aceste directoare: Controllers, Models si Views.

project.json

Acest fisier contine printre altele toate dependentele pe care le cere aplicatia. Intellisense este disponibil in cadrul acestui fisier. Pachetele pe care le adaugam (ca nume) in acest fisier vor fi aduse automat din NuGet. Daca eliminam un pachet din acest fisier, NuGet va elimina toate referintele la acesta din tot proiectul.

Pentru a adauga MVC la proiect va trebui sa modificam acest fisier prin adaugarea "Microsoft.AspNet.Mvc" sub sectiunea **dependencies**.

Initial continutul fisierului project.json este:

```
{  
  "version": "1.0.0-*",  
  "compilationOptions": {
```

```

    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
  "publishExclude": [
    "**.user",
    "**.vspscc"
  ]
}

```

Dupa adaugare dependenta:

```

"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
  "Microsoft.AspNet.Mvc": "6.0.0-rc1-final"
},

```

In acest moment este disponibila versiunea 6.0.0-rc1-final.

Configurare pipeline aplicatie

Pentru acest lucru trebuie sa adaugam cod in fisierul *Startup.cs*, in metoda *Configure*. Codul adaugat specifica ruta de folosit. Dupa modificare metoda *Configure* are urmatorul cod:

```

// This method gets called by the runtime. Use this method to
// configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();

    app.UseMvc(m => m.MapRoute(
        name: "Default",
        template: "{controller}/{action}",
        defaults: new { controller = "Home", action = "Index" }));

    app.Run(async (context) =>

```

```

    {
        await context.Response.WriteAsync("Hello World!");
    });
}

```

iar in servicii trebuie sa adaugam:

```

public void ConfigureServices(IServiceCollection services)
{
    #region
    //
    // Summary:
    //     Adds MVC services to the specified
    //     Microsoft.Extensions.DependencyInjection.IServiceCollection.
    //
    // Parameters:
    //     services:
    //     The Microsoft.Extensions.DependencyInjection.IServiceCollection
    //     to add services to.
    //
    // Returns:
    //     A reference to this instance after the operation
    //     has completed.
    #endregion
    services.AddMvc();
}

```

Adaugare Models

In folderul Models adaug urmatoarele clase:

```

using System;
using System.Linq;
namespace ASPMVC6Demo.Models
{
    public class Post
    {
        public Post()
        {
        }
        public int ID { get; set; }
        public string Description { get; set; }
        public string Type { get; set; }
    }

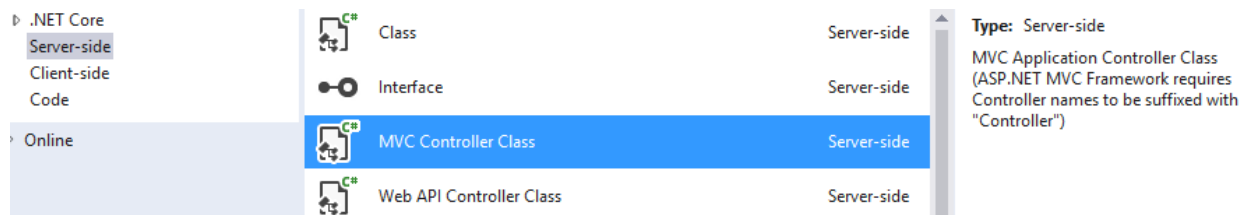
    public class PostRepository
    {
        private readonly List<Post> data = new List<Post>()
        {
            new Post {ID = 1, Description = "Entity Framework next vesrion",
                Type = "new"},
            new Post {ID = 2, Description = "WCF next update", Type = "new" },
            new Post {ID = 3, Description = "ADO.NET ", Type = "old" }
        };
    }
}

```

```
public List<Post> Data { get { return data; } }  
public List<Post> GetPostByType(string type)  
{  
    return Data.Where(x => x.Type == type);  
}  
}
```

Adaugarea unui Controller

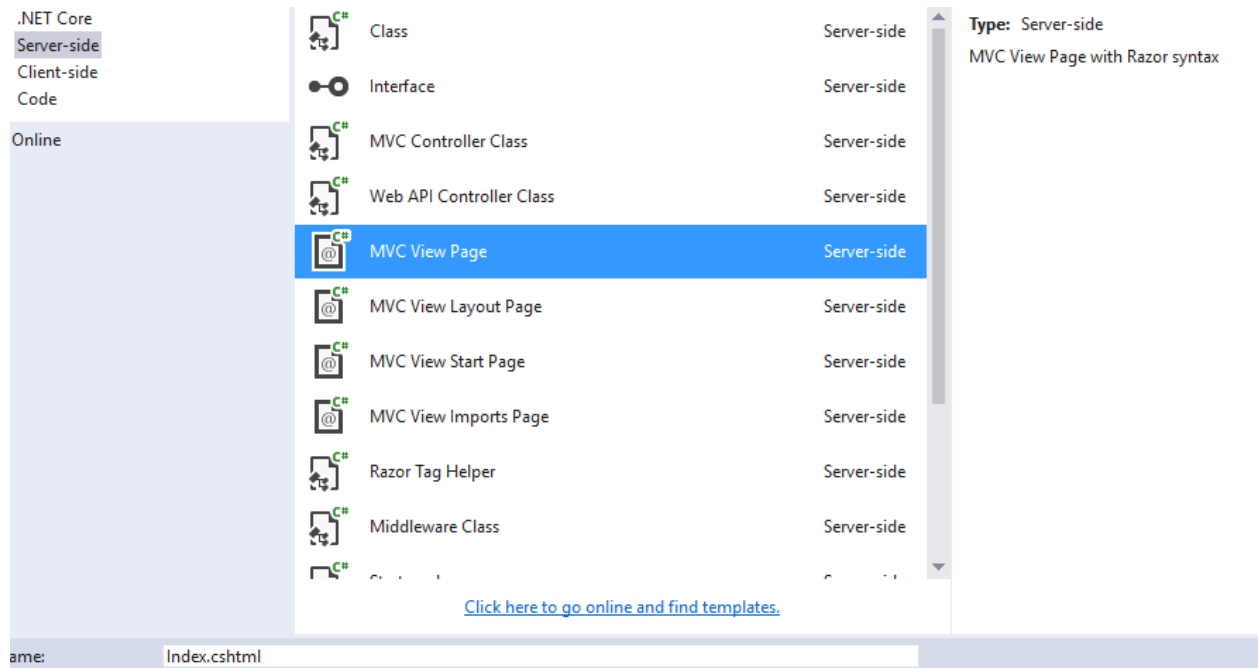
Adaugarea unui controller se face la fel ca in MVC 5. Din meniul contextual pe folder *Controllers* selectam *Add > New Item*, iar in dialogul ce apare selectam MVC Controller Class:



Adaugarea unei vizualizari

Trebuie sa ne asiguram mai intai ca am creat folderul cu numele controller-ului in folderul Views.

In meniul contextual pe nume folder din Views (de exemplu Home) selectam *Add > New Item > MVC View Page*:



Metode helper in ASP 5

Ce sunt Tag Helpers?

Tag Helpers permite codului de pe partea de server de a participa în crearea și redarea elementelor HTML în fișiere Razor. Există Tag Helpers preconstruiți pentru activitățile comune, cum ar fi: creare de formulare, link-uri, label-uri, input-uri, etc. Tag Helpers identifică elementele HTML după numele elementului, numele atributului sau tag-ul părinte. De exemplu **LabelTagHelper** generează elementul HTML **<label>**.

Ce furnizează Tag Helpers?

Tag Helpers arată ca standardul HTML. Proiectanții din front-end ce folosesc HTML/CSS/JavaScript pot edita Razor fără a învăța sintaxa C# Razor.

IntelliSense este disponibil pentru crearea marcarilor HTML și Razor.

Majoritatea Tag Helpers preconstruiți sunt pentru elementele HTML existente și furnizează atribute pe partea de server pentru elementele respective. De exemplu, elementul **<input>** conține atributul **asp-for**, atribut ce extrage numele proprietății specifice din model în elementul HTML redat. Următorul marcaj Razor:

```
<label asp-for="Email"></label>
```

generează:


```
<label for="Email">Email</label>
```

Atributul **asp-for** este disponibil prin proprietatea **For** in **LabelTagHelper**.

Domeniul de vizibilitate – Tag Helpers

Domeniul de vizibilitate este controlat de directivele **@addTagHelper**, **@removeTagHelper** si caracterul “!”.

```
@addTagHelper
```

Face disponibil Tag Helpers

Fisierul *Views/_ViewImports.cshtml* contine

```
@using Nume_Aplicatie
```

```
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
```

Acest lucru semnifica faptul ca pentru aceasta aplicatie toate vizualizarile au acces la tag helpers.

Toate vizualizarile mostenesc in mod implicit *_ViewImports.cshtml*. “*” semnifica faptul ca toate Tag Helpers din assembly **Microsoft.AspNet.Mvc.TagHelpers** sunt disponibile oricarei vizualizari din directorul *Views* sau subdirectoare ale acestuia.

Pentru a expune toate tag helpers in cadrul unui proiect, proiect ce creaza un assembly numit, de exemplu, *MyAppTagHelpers* continutul fisierului *_ViewImports.cshtml* trebuie sa fie:

```
@using MyAppTagHelpers
```

```
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
```

```
@addTagHelper "*", MyAppTagHelper"
```

In loc de *, primul parametru al directivei `addTagHelper`, putem scrie numele calificat complet al tag helper-ului, de exemplu:

```
@addTagHelper "MyAppTagHelper.TagHelper.EmailTagHelper, MyAppTagHelper"
```

Primul parametru specifica numele pentru Tag Helper, iar al doilea parametru specifica numele assembly unde se afla definit tag Helper. Puetem folosi directiva

@addTagHelper in fisiere ce contin vizualizari daca optam ca Tag Helper sa fie disponibil numai pentru acestea.

@removeTagHelper

Aceasta directiva elimina Tag Helpers. Sintaxa este aceeaasi ca la **@addTagHelper**. Daca **@removeTagHelper** este specificat intr-o vizualizare, atunci se elimina Tag Helper specificat din acea vizualizare. Daca **@removeTagHelper** este specificat in **_ViewImports.cshtml** din folderul Views, atunci se elimina Tag Helpers specificati pentru toate vizualizarile din Views.

Utilizare *_ViewImports.cshtml*

Putem adauga un fisier **_ViewImports.cshtml** la orice folder din Views, si motorul de vizualizare va adauga directivele din acest fisier la cele continute in **Views/_ViewImports.cshtml**. Adaugarea este aditiva. Un fisier vid, **_ViewImports.cshtml** plasat intr-un director din Views nu va produce modificari asupra utilizarii Tag Helpers.

Optional – elemente individuale

Putem dezactiva Tag Helper la nivel de element folosind sintaxa “!”. De exemplu, nu dorim ca tag helper Email sa fie folosit in ****, si atunci vom scrie:

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

Trebuie sa aplicam “!” atat pe tag-ul de deschidere cat si pe cel de inchidere.

Utilizare explicita Tag Helper - @tagHelperPrefix

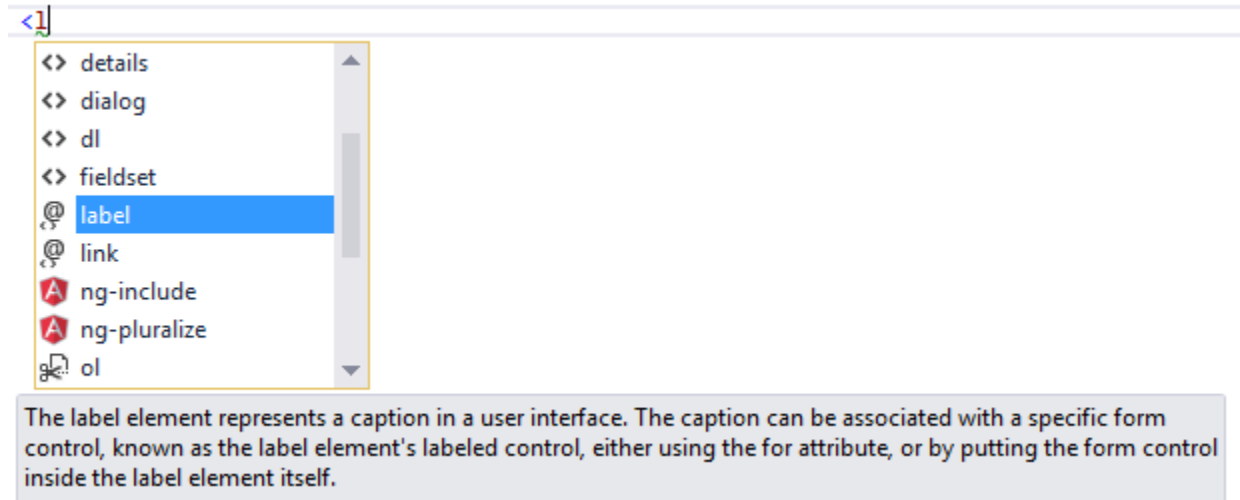
Directiva **@tagHelperPrefix** permite sa specificam un prefix al tag-ului (string) pentru a activa suportul Tag Helper si a permite utilizarea explicita a Tag Helper. Elementele HTML vor fi precedate de acest prefix urmat de : si apoi numele elementului.

```
@tagHelperPrefix "th:"
<div class="form-group">
  <th:label asp-for="Password" class="col-md-2 control-label"></th:label>
  <div class="col-md-10">
    <input asp-for="Password" class="form-control" />
    <th:span asp-validation-for="Password" class="text-danger"></th:span>
  </div>
</div>
```

Suport IntelliSense pentru Tag Helpers

In `project.json` ar trebui sa avem o referinta la `"Microsoft.AspNet.Tooling.Razor"`, in caz contrar adaugam referinta.

Cand vom scrie un element HTML in View, de exemplu `<1` (dorim sa scriem `<label>`), IntelliSense va afisa:

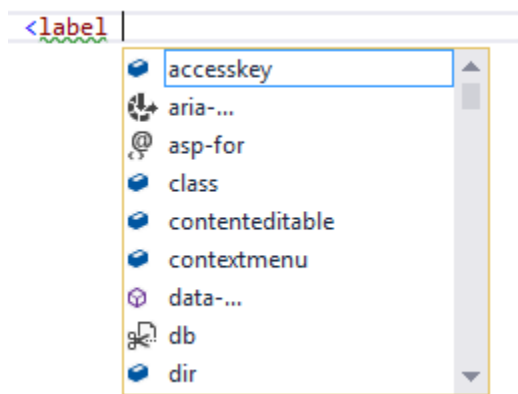


Observam drept icoane (simbol @ si `<>` sub el).



Elementele ce prezinta asemenea icoane au preconstruite tag helpers. Elementele pure HTML prezinta drept icoana `<>`.

Dupa ce selectam un element, IntelliSense va afisa attributele disponibile:



MVC 6 – Creare Tag Helpers personalizati

Creăm un tag helper pentru transmitere email dintr-o pagina.

Dorim ca în pagina să avem un marcaj de forma:

```
<address>
    <strong>Info 3:</strong><email mail-to="info3"></email><br />
    <strong>Administrator:</strong><email mail-to="admin"></email>
</address>
```

iar în pagina generată să rezulte:

[Info 3:info3@info.uaic.ro](mailto:info3@info.uaic.ro)
[Administrator:admin@info.uaic.ro](mailto:admin@info.uaic.ro)

Observăm aici că *info.uaic.ro* nu apare în marcaj.

Pentru aceasta vom crea o clasă derivată din TagHelper (am creat un folder Helpers) ce va avea următoarea definiție:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "info.uaic.ro";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag
        var address = MailTo + "@" + EmailDomain;
        output.Attributes["href"] = "mailto:" + address;
        output.Content.SetContent(address);
    }

    // Varianta asincrona
    /*
    public override async Task ProcessAsync(TagHelperContext context,
        TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag
        var content = await context.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes["href"] = "mailto:" + target;
        output.Content.SetContent(target);
    }
    */
}
```

```
}
```

Daca ne uitam in *View page source* din browser vom vedea marcajul generat ca fiind:

```
<address>
  <strong>Info 3:</strong><a href="mailto:info3@info.uaic.ro">info3@info.uaic.ro</a><br />
  <strong>Administrator:</strong><a href="mailto:admin@info.uaic.ro">admin@info.uaic.ro</a>
</address>
```

3. Cream un tag helper ce va inlocui peste tot atributul bold (daca exista) cu

```
<strong>
```

Clasa va fi:

```
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

iar marcajul in view

```
<p bold>Un text cu bold transformat in strong</p>
```

si rezultatul in view page source:

```
<p><strong>Un text cu bold transformat in strong</strong></p>
```

Exemplu

In view vrem sa avem marcajul:

```
<h3> web site info </h3>
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1790,
    Approved = true,
    TagsToShow = 131 }" />
```

iar pagina generata sa arate astfel:

```
<h3> web site
info </h3>
```

```
<section><ul><li><strong>Version:</strong> 1.3</li>
<li><strong>Copyright Year:</strong> 1790</li>
<li><strong>Approved:</strong> True</li>
<li><strong>Number of tags to show:</strong>
131</li></ul></section>
```

Deci pentru tag **website-information** sa se genereze marcajul de mai sus.

Rezolvare

Cream clasa WebsiteContext in folder Models:

```
public class WebsiteContext
{
    public Version Version { get; set; }
    public int CopyrightYear { get; set; }
    public bool Approved { get; set; }
    public int TagsToShow { get; set; }
}
```

Cream tag helper WebsiteInformation astfel:

```
[HtmlTargetElement("Website-Information")]
public class WebsiteInformationTagHelper : TagHelper
{
    public WebsiteContext Info { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.TagName = "section";
        output.Content.SetHtmlContent(
            $"<ul><li><strong>Version:</strong> {Info.Version}</li>
            <li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
            <li><strong>Approved:</strong> {Info.Approved}</li>
            <li><strong>Number of tags to show:</strong>
            {Info.TagsToShow}</li></ul>");
        output.TagMode = TagMode.StartTagAndEndTag;
    }
}
```

Rezultatul este:

web site info

- **Version:** 1.3
- **Copyright Year:** 1790
- **Approved:** True
- **Number of tags to show:** 131