



Școala
informală
de IT

Intro in ASP.NET MVC



Curriculum

1. The HTTP Protocol

2. The MVC Pattern

- Model, View, Controller
- The MVC Pattern for Web and Examples

3. ASP.NET MVC

- ASP.NET MVC Advantages

4. Creating ASP.NET MVC Project

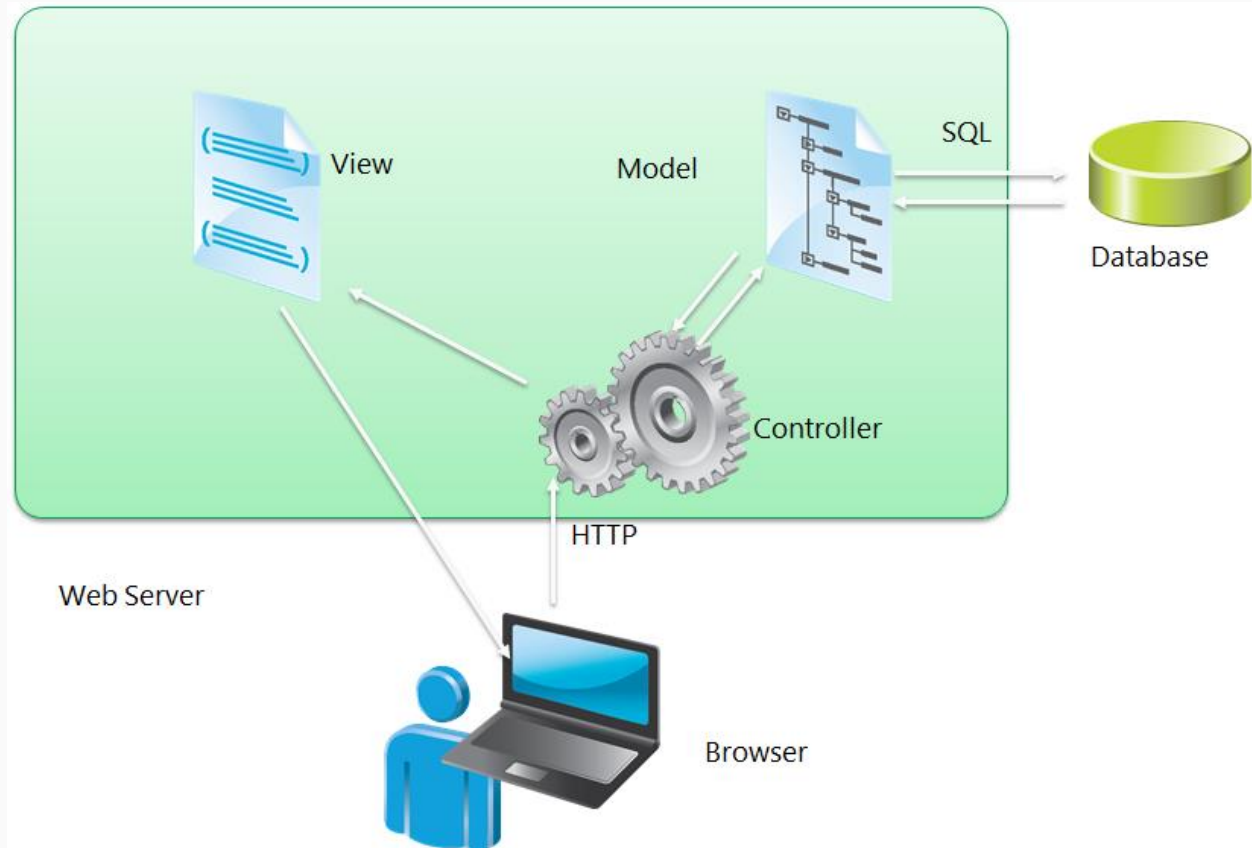
5. NuGet Package Management



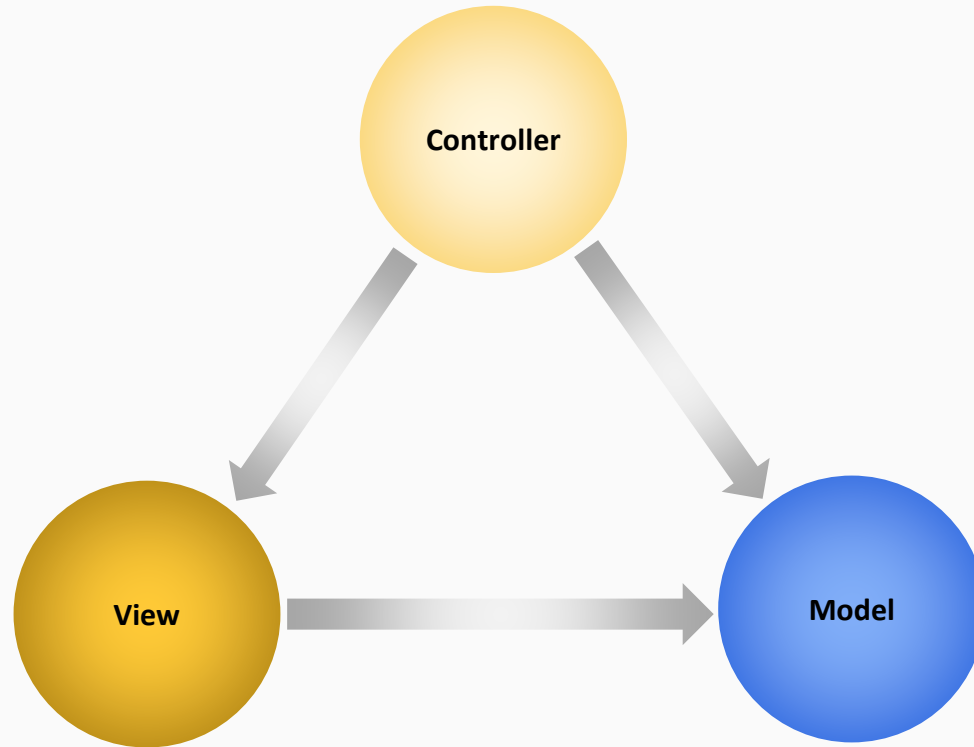
The MVC Pattern



The MVC Pattern



The MVC Pattern



The MVC Pattern

- **Model–view–controller (MVC) is a software architecture pattern**
- **Originally formulated in the late 1970s by Trygve Reenskaug as part of the Smalltalk**
- **Code reusability and separation of concerns**
- **Originally developed for desktop, then adapted for internet applications**

Model

- Set of classes that describes the data we are working with as well as the business
- Rules for how the data can be changed and manipulated
- May contain data validation rules
- Often encapsulate data stored in a database as well as code used to manipulate the data
- Apart from giving the data objects, it doesn't have significance in the framework

View

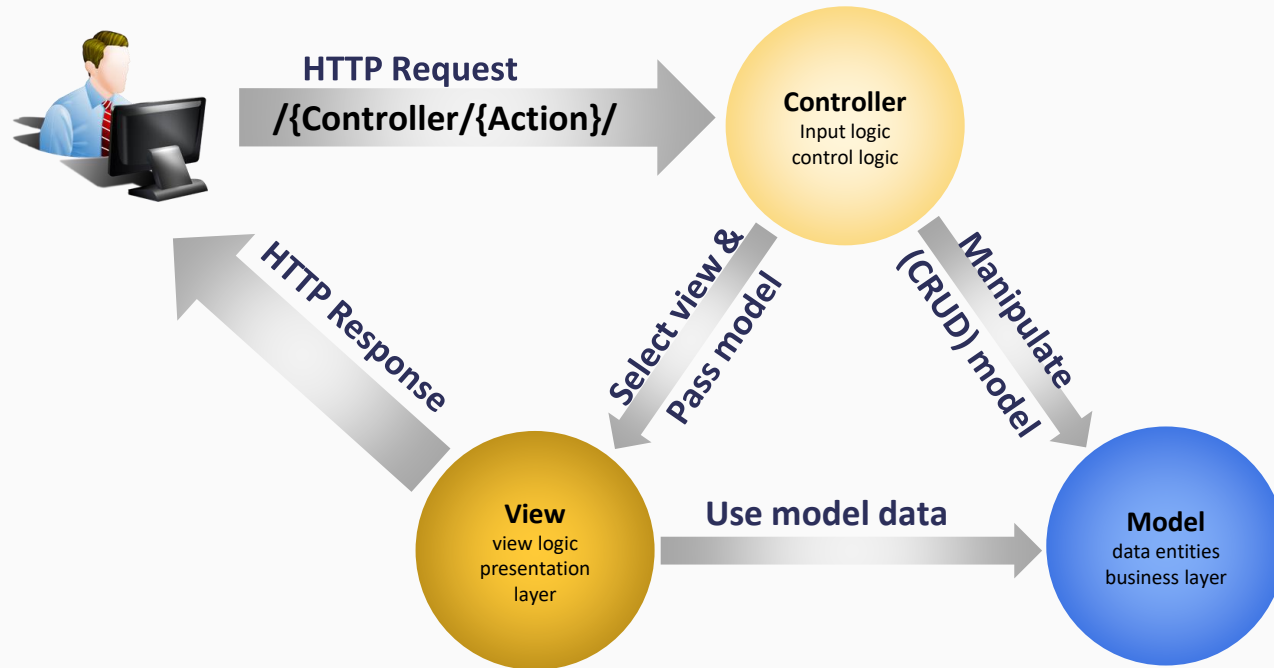
- Defines how the application's user interface (UI) will be displayed
- May support master views (layouts) and sub-views (partial views or controls)
- Web: Template to dynamically generate HTML

Controller

- The core MVC component
- Process the requests with the help of views and models
- A set of classes that handles
 - Communication from the user
 - Overall application flow
 - Application-specific logic
- Every controller has one or more "Actions" mapped to Verbs

The MVC Pattern

- Model–view–controller



MVC Steps

- Incoming HTTP request routed to **Controller**
- Controller processes request and creates presentation **Model**
 - Controller also selects appropriate result (view)
- **Model** is passed to **View**
- **View** transforms **Model** into appropriate output format (HTML)
- Response is rendered (HTTP Response)

ASP.NET MVC General



ASP.NET MVC

- **Runs on top of ASP.NET**
- **Embrace the web**
 - **User/SEO friendly URLs, HTML 5, SPA**
 - **Adopt REST concepts**
- **Uses MVC pattern**
 - **Conventions and Guidance**
 - **Separation of concerns**

ASP.NET MVC Core

- **Tight control over markup**
- **Testable**
- **Loosely coupled and extensible**
- **Convention over configuration**
- **Easy to configure just respecting the conventions**

Extensible

- **Replace any component of the system**
 - Interface-based architecture
- **Almost anything can be replaced or extended**
 - Model binders (request data to CLR objects)
 - Action/result filters (e.g. `OnActionExecuting`)
 - Custom action result types
 - View engine (Razor, WebForms)
 - View helpers (HTML, AJAX, URL, etc.)
 - Custom data providers (ADO.NET), etc.

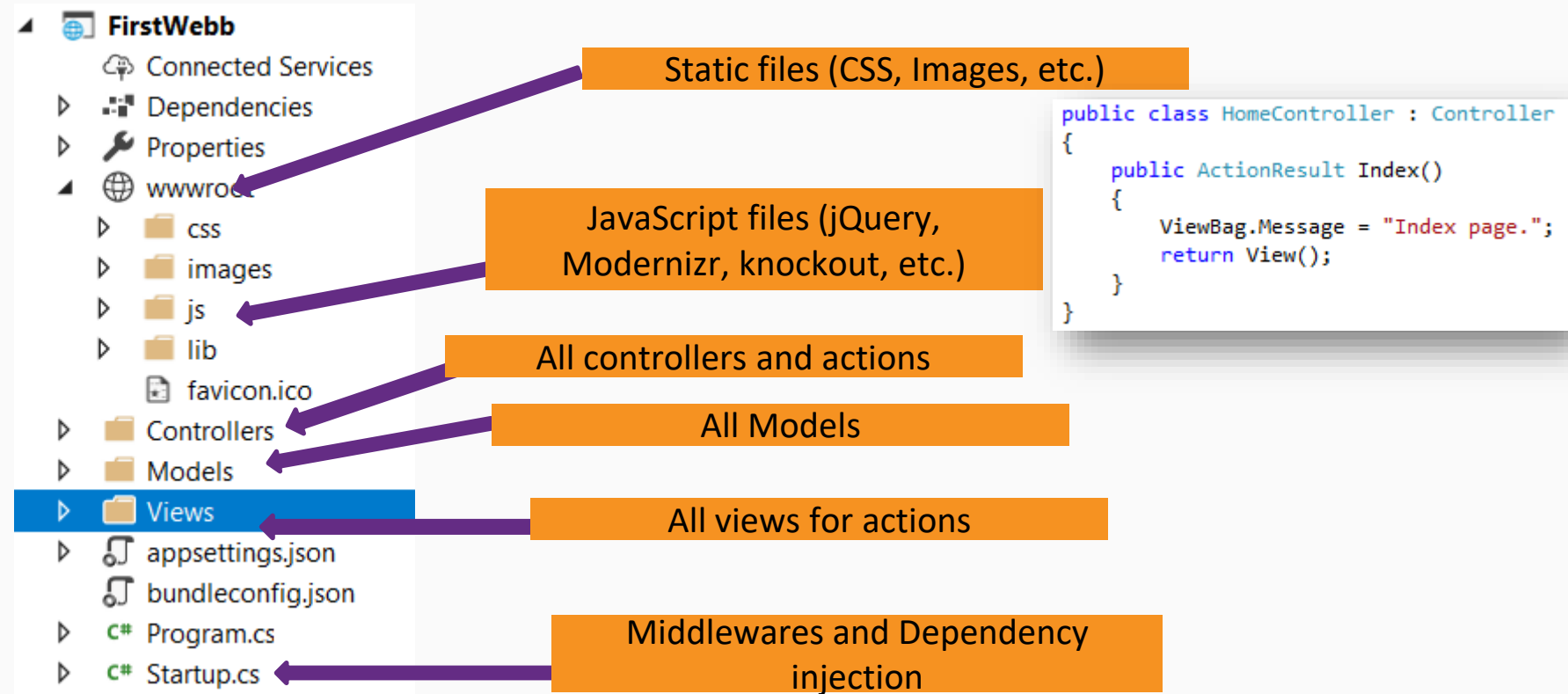
Creating ASP.NET MVC Project



The Technologies

- **Technologies that ASP.NET MVC uses**
 - **C# (OOP, unit testing, async, etc.)**
 - **ASP.NET**
 - **HTML 5 and CSS 3**
 - **JavaScript (jQuery, AngularJS, ReactJS etc.)**
 - **AJAX, SPA (Single-page apps)**
 - **Databases (MS SQL)**
 - **ORM (Entity Framework and LINQ)**
 - **Web and HTTP**

Internet App Project Files



Model



Model

- **Should have validation**
- **Used to transport data to the view**

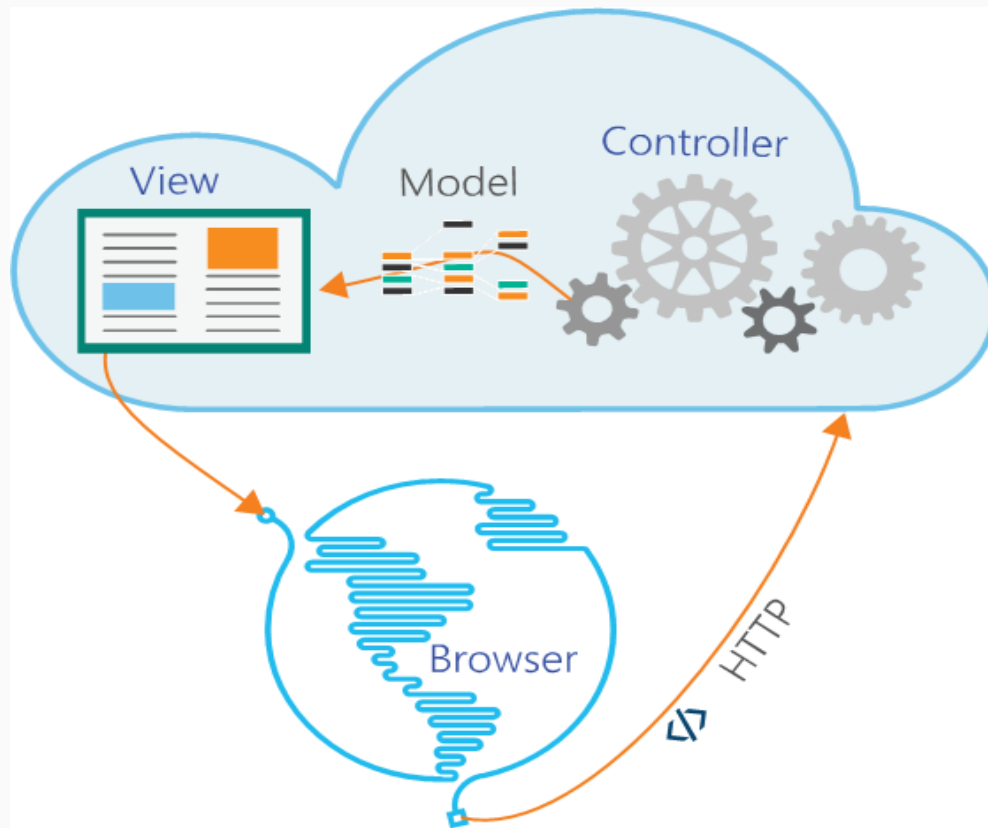
Custom View Models

- You can have nested types in a View

```
namespace ContosoUniversity.ViewModels
{
    public class ExamForStudent
    {
        public int ExamId { get; set; }
        public Course course { get; set; }
        public Student student { get; set; }
    }
}
```

Controller





What is Controller

- It is a class
- Derives from the base `System.Web.Mvc.Controller` class
- Generates the response to the browser request

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        ViewBag.Message = "Welcome to ASP.NET MVC!";

        return View();
    }

    public IActionResult About()
    {
        return View();
    }
}
```


Controller Actions


- Public method of the Controller class
- Cannot be a static method
- Returns a View will search for and About.cshtml in Views/ControllerName folder or under Views/Shared folder

```
public IActionResult About()  
{  
    return View();  
}
```

Controller Actions

- You can pass a model
- Or you can pass a model and a view name.
- The same model type needs to be declared in View using **@model**

```
public IActionResult Create()  
{  
    Dog model=new Dog();  
    return View(model);  
}
```



Will search a view called Create.cshtml
under Views/{ControllerName}
or
under Views/Shared

Action Selectors

- **ActionName(string name)**
- **AcceptVerbs**
 - **HttpPost**
 - **HttpGet**
 - **HttpDelete**
 - **HttpOptions**
 - ...
- **NonAction**
- **RequireHttps**

```
public class UsersController : Controller
{
    [ActionName("UserLogin")]
    [HttpPost]
    [RequireHttps]
    public ActionResult Login(string
pass)
    {
        return Content(pass);
    }
}
```

View



View

- Handle the presentation logic
- The path to the view is inferred from the name of the controller and the name of the controller action.
 - `http://{hostname}/{Controller}/{Action}`
- A view is a CSHTML document that can contain **html and server-side code**
 - `@` - inline expressions
 - `@{...}` - code blocks

Pass Data to a View

- **With ViewData:**

- ViewData["message"] = "Hello World!";
- Strongly typed ViewData:
 - ViewData.Model = model;

- **With ViewBag:**

- ViewBag.Message = "Hello World!";
- ViewBag is dynamic property of BaseController class

- **With a model passed as a parameter from the action**

Action Selectors

- **ActionName(string name)**
- **AcceptVerbs**
 - **HttpPost**
 - **HttpGet**
 - **HttpDelete**
 - **HttpOptions**
 - ...
- **NonAction**
- **RequireHttps**

```
public class UsersController : Controller
{
    [ActionName("UserLogin")]
    [HttpPost]
    [RequireHttps]
    public ActionResult Login(string
pass)
    {
        return Content(pass);
    }
}
```

Razor syntax



Razor Syntax

- if, else, for, foreach, etc. C# statements
 - HTML markup lines can be included at any part
 - @: – For plain text line to be rendered

```
<div class="products-list">
@if (Model.Products.Count() == 0)
{
    <p>Sorry, no products found!</p>
}
else
{
    @:List of the products found:
    foreach(var product in Model.Products)
    {
        <b>@product.Name, </b>
    }
}
</div>
```

Razor Syntax

- Comments

```
@*  
    A Razor Comment  
*@  
  
@{  
    // A C# comment  
  
    /* A Multi  
       line C# comment  
    */  
}
```

- What about "@" and emails?

```
<p>  
    This is the sign that separates email names from domains: @@<br />  
    And this is how smart Razor is: spam_me@@gmail.com  
</p>
```

Razor Syntax

- **@(...)** – Explicit code expression

```
<p>  
  Current rating(0-10): @Model.Rating / 10.0  
  Current rating(0-1): @(Model.Rating / 10.0)  
  spam_me@@Model.Rating  
  spam_me@(Model.Rating)  
</p>
```

@* 6 / 10.0 *@
@* 0.6 *@
@* spam_me@Model.Rating *@
@* spam_me6 *@

- **@using** – for including namespace into view
- **@model** – for defining the model for the view

```
@using MyFirstMvcApplication.Models;  
@model UserModel  
<p>@Model.Username</p>
```

Helpers



HTML Helpers

- Extension methods which generate html elements
 - Example: `Html.TextBox()`
- It is advisable to use "For" extension methods to use strong-type model
 - Example: `Html.TextBoxFor()`
- Usage is optional
- You can create your own HTML Helpers

HTML Helpers

Html Helper	Strongly Typed Html Helpers	Html Control
<i>Html.ActionLink</i>		Anchor link
<i>Html.TextBox</i>	<i>Html.TextBoxFor</i>	Textbox
<i>Html.TextArea</i>	<i>Html.TextAreaFor</i>	TextArea
<i>Html.CheckBox</i>	<i>Html.CheckBoxFor</i>	Checkbox
<i>Html.RadioButton</i>	<i>Html.RadioButtonFor</i>	Radio button
<i>Html.DropDownList</i>	<i>Html.DropDownListFor</i>	Dropdown, combobox
<i>Html.ListBox</i>	<i>Html.ListBoxFor</i>	multi-select list box
<i>Html.Hidden</i>	<i>Html.HiddenFor</i>	Hidden field
<i>Html.Password</i>	<i>Html.PasswordFor</i>	Password textbox
<i>Html.Display</i>	<i>Html.DisplayFor</i>	Html text
<i>Html.Label</i>	<i>Html.LabelFor</i>	Label
<i>Html.Editor</i>	<i>Html.EditorFor</i>	Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type

Html.TextBox vs Html.TextBoxFor

```
@Html.TextBox("StudentName", "John", new { @class = "form-control" })
```

Renders to

```
<input class="form-control" id="StudentName" name="StudentName" type="text" value="John" />
```

and...

```
@model Student
```

```
@Html.TextBoxFor(m => m.StudentName, new { @class = "form-control" })
```

Renders to

```
<input class="form-control" id="StudentName" name="StudentName" type="text" value="John" />
```

Html.TextBox vs Html.TextBoxFor

- **@Html.TextBox()** is loosely typed method whereas
- **@Html.TextBoxFor()** is a strongly typed (generic) extension method.
- **TextBox** requires property name as string parameter and **TextBoxFor()** requires lambda expression as a parameter.
- **TextBox** doesn't give you compile time error if you have specified wrong property name. It will throw runtime exception.
- **TextBoxFor** is generic method so it will give you compile time error if you have specified wrong property name or property name changes.

Html.EditorFor

Property DataType	Html Element
<i>string</i>	<input type="text" >
<i>int</i>	<input type="number" >
<i>decimal, float</i>	<input type="text" >
<i>boolean</i>	<input type="checkbox" >
<i>Enum</i>	<input type="text" >
<i>DateTime</i>	<input type="datetime" >

Html.Editor, Html.EditorFor

```
StudentId:      @Html.Editor("StudentId")
Student Name:   @Html.Editor("StudentName")
Age:            @Html.Editor("Age")
Password:       @Html.Editor("Password")
IsNewlyEnrolled: @Html.Editor("IsNewlyEnrolled")
Gender:         @Html.Editor("Gender")
DoB:           @Html.Editor("DoB")
```

```
StudentId:      @Html.EditorFor(m => m.StudentId)
Student Name:   @Html.EditorFor(m => m.StudentName)
Age:            @Html.EditorFor(m => m.Age)
Password:       @Html.EditorFor(m => m.Password)
IsNewlyEnrolled: @Html.EditorFor(m => m.IsNewlyEnrolled)
Gender:         @Html.EditorFor(m => m.Gender)
DoB:           @Html.EditorFor(m => m.DoB)
```

Output of Editor / EditorFor helper

StudentId:	<input type="text" value="1"/>
Student Name:	<input type="text" value="Jogn"/>
Age:	<input type="text" value="19"/>
Password:	<input type="text" value="sdf"/>
isNewlyEnrolled:	<input checked="" type="checkbox"/>
Gender:	<input type="text" value="Boy"/>
DoB:	<input type="text" value="02-06-2015 11:39:15"/>

Tag Helpers



Input TagHelper

- **Html Element**

```
<input type="text" id="Age" name="Age" value="@Model.Age" class="form-control" />
```

- **Html Helper**

```
@Html.TextBoxFor(m => m.Age, new { @class = "form-control" })
```

- **TagHelper**

```
<input asp-for="Age" class="form-control" />
```

- **To make available the tag helpers add next lines in _ViewImports.cshtml**

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

TagHelpers

- Anchor tag

```
<a asp-action="ActionName" asp-controller="ControllerName" asp-route="RouteName">...</a>
```

- Form tag

```
<form asp-action="ActionName" asp-controller="ControllerName" method="post"></form>
```

- Input tag

```
<input asp-for="FieldName" class="form-control" />
```

- Validation tag and validation summary

```
<span asp-validation-for="FieldName"></span>
```

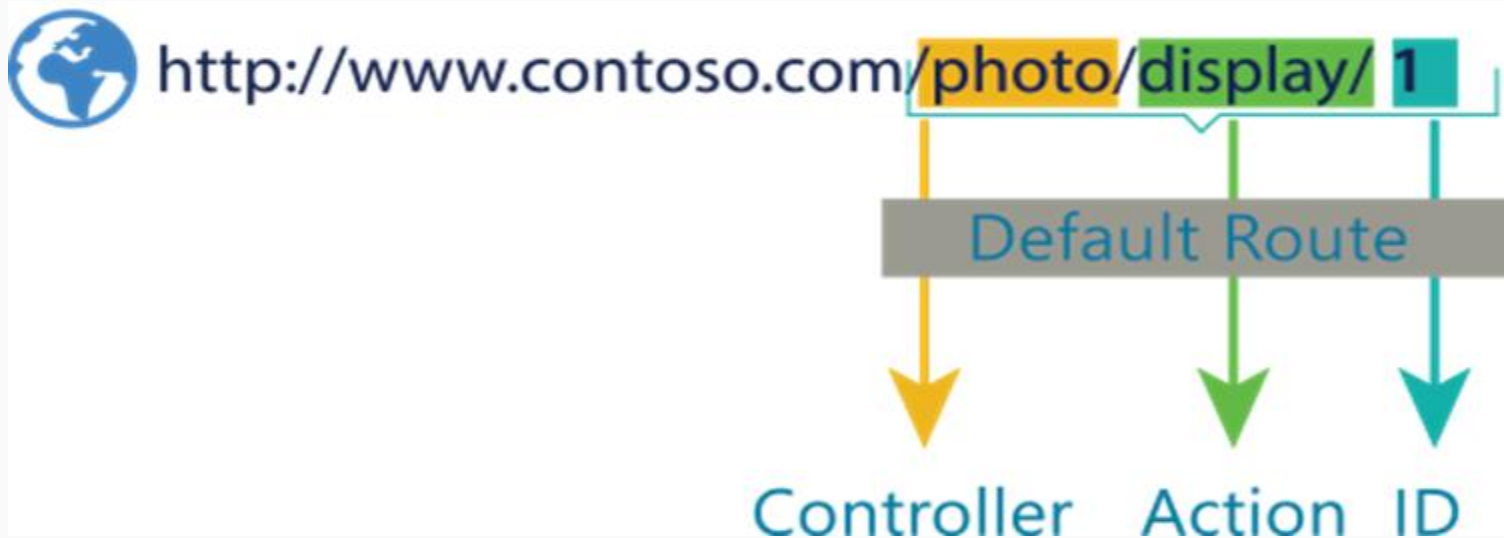
```
<div asp-validation-summary="ModelOnly"></div>
```

Routing



Routing

- The Routing module is responsible for mapping incoming browser requests to particular MVC controller actions.



Convention Based Routing

- The Routing module is responsible for mapping incoming browser requests to particular MVC controller actions.
- Setup places:

- **Startup.cs file**

```
app.UseMvcWithDefaultRoute();
```

```
http://www.mysite.com/Products/ById/3  
// controller = Products  
// action = ById  
// id = 3
```

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

Route name

Route pattern

Default parameters

Attribute Routing

- Attribute Routing

```
public class ProductController : Controller
{
    [Route("")]
    [Route("Product")]
    [Route("Product/Index")]
    public IActionResult Index()
    {
        return View();
    }
}
```

- Attribute routing with Http[Verb]

```
[HttpGet("/products/{id}")]
public IActionResult GetProducts(int id)
{
    // ...
}
```

- Attribute routing tokens

```
[HttpGet("[controller]/[action]/{id}")]
public IActionResult Edit(int id)
{
    // ...
}
```

Attribute Routing Constraints

alpha - Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z)

bool - Matches a Boolean value.

datetime - Matches a DateTime value.

decimal - Matches a decimal value.

double - Matches a 64-bit floating-point value.

float - Matches a 32-bit floating-point value.

guid - Matches a GUID value.

int - Matches a 32-bit integer value.

length - Matches a string with the specified length or within a specified range of lengths.

long - Matches a 64-bit integer value.

max - Matches an integer with a maximum value.

maxlength - Matches a string with a maximum length.

min - Matches an integer with a minimum value.

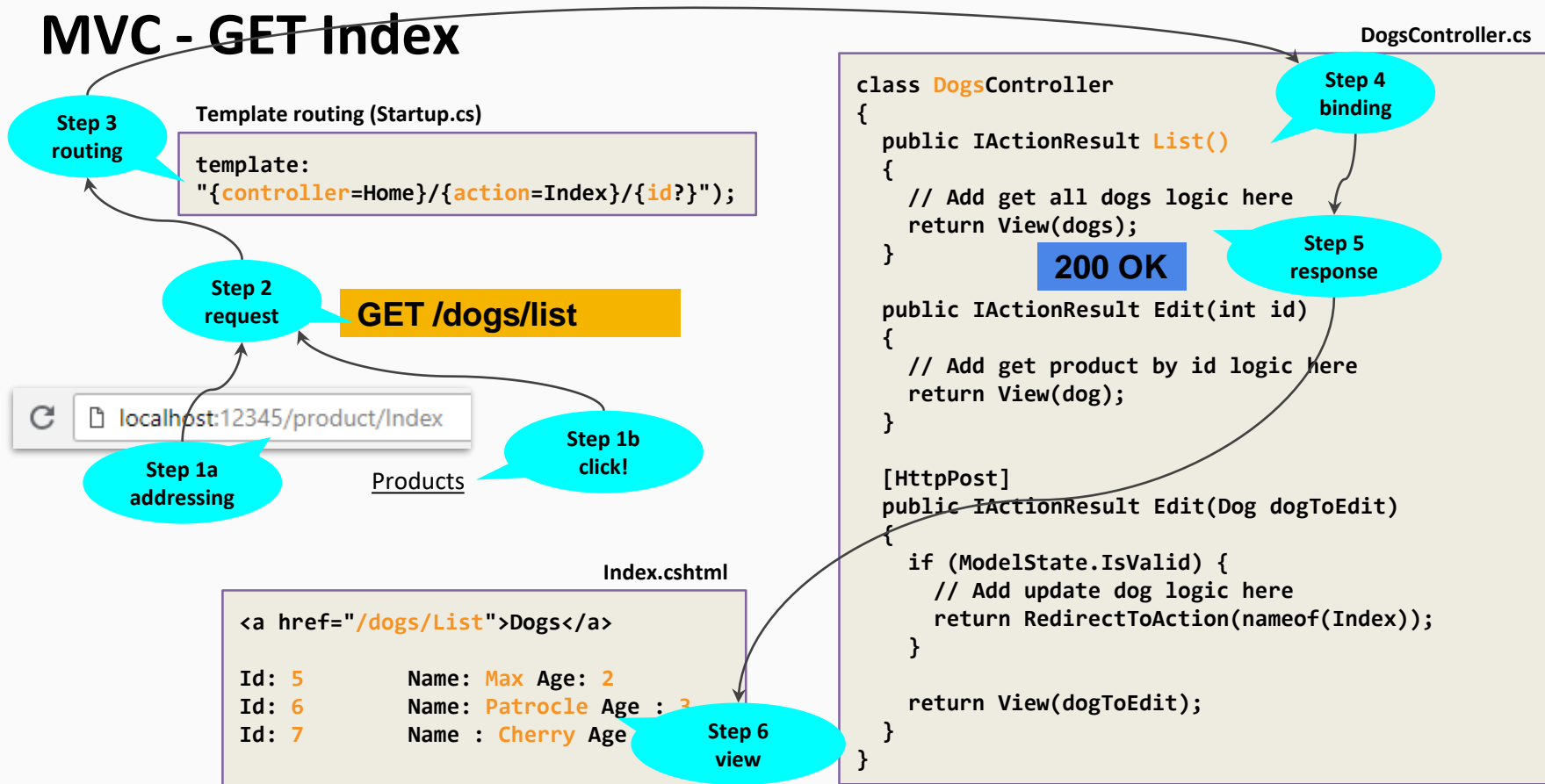
minlength - Matches a string with a minimum length.

range - Matches an integer within a range of values.

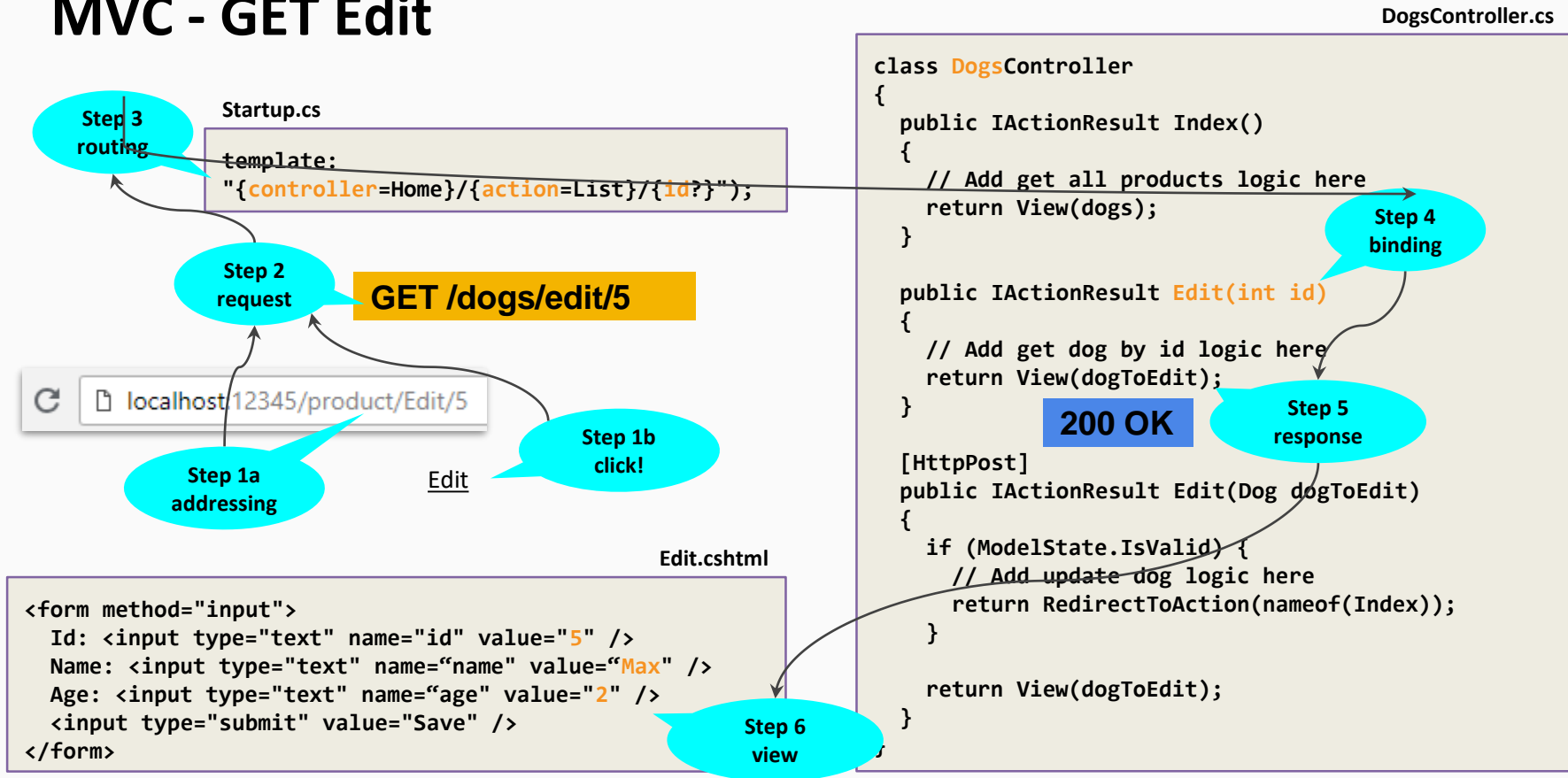
regex - Matches a regular expression.

```
// combining multiple constraints using colon (:)  
"/{id:alpha:minlength(6)}"
```

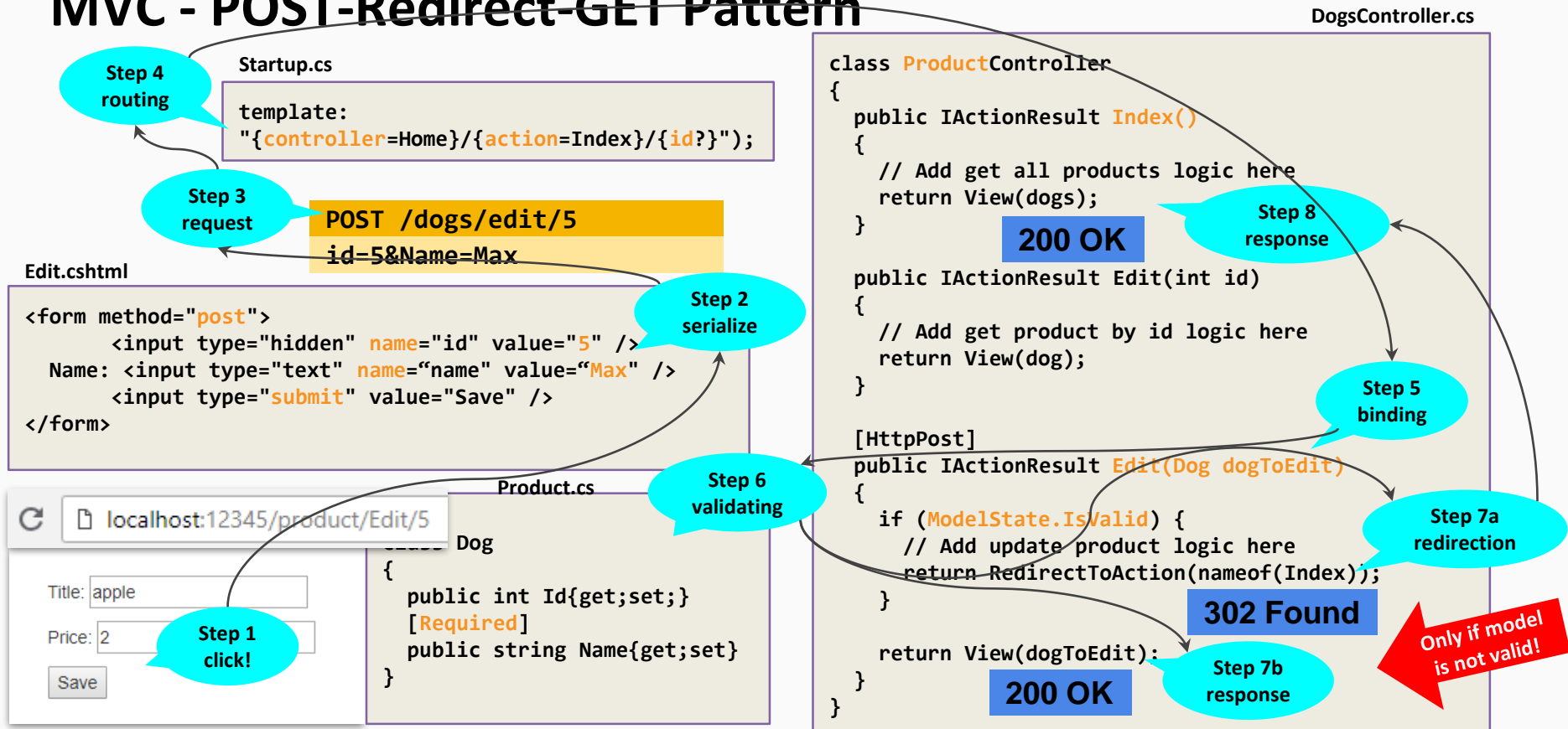
MVC - GET Index



MVC - GET Edit



MVC - POST-Redirect-GET Pattern



Posting data to controllers





A web form on a monitor screen. It has two text input fields. The first is labeled "First Name" and contains the text "James". The second is labeled "Last Name" and contains the text "Smith". Below the input fields is a blue button with the text "Submit my name".



Person
FirstName=James
LastName=Smith



```
[HttpPost]  
public IActionResult GetName(Person person)
```

Post data – View side

- The model specified in the view will reach the controller specified in form element

```
@model Models.Dog
```

```
<h2>Create</h2>
```

```
<h4>Dog</h4>
```

```
<hr />
```

```
<form asp-action="Create" asp-controller="Dogs">
```

```
  <label asp-for="Name" class="control-label"></label>
```

```
  <input asp-for="Name" class="form-control" />
```

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

```
</form>
```

Will send the model to
/Dogs/Create

Post data – Controller side

- The action method in the controller accepts the values posted from the view.
- The view form fields must match the same names in the controller.

```
[HttpPost]
public IActionResult Create(Dog model)
{
    if (ModelState.IsValid)
    {
        // save the dog in database
        return RedirectToAction("List");
    }

    return View(model);
}
```

Model passed from the View



Model Binders



Model Binders

- To make easy of handling HTTP post request
- Help the populating the parameters in action methods

Rating

Body

HTTP POST /Review/Create
Rating=7&Body=Great!



DefaultModelBinder

```
public ActionResult Create (Review newReview)
{
    // ...
}
```

Model Binders

- Parameter binding

- The name attribute of the input HTML element should be the same as the name of parameter in the action

```
@*Html.TextBox("first", null, new { type = "number" })*@  
<div>  
    <input type="number" name="first" />  
</div>
```

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public ActionResult Parameter(int first, string second, bool third)  
{  
    TempData["Success"] = string.Format("{0} {1} {2}", first, second,  
third);  
    return RedirectToAction ("Index");  
}
```

Model Binders

- Object binding
 - Model binder will try to "construct" the object based on the name attributes on the input HTML elements

```
@model WorkinWithDataMvc.Models.PersonViewModel
```

```
@*<input type="text" name="FirstName" />*@  
@Html.EditorFor(m => m.FirstName)
```

```
public class PersonViewModel  
{  
    public string FirstName { get;  
    set; }  
    public string LastName { get;  
    set; }  
    public int Age { get; set; }  
}
```

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public ActionResult Object(PersonViewModel person)  
{  
    return this.SetTempDataAndRedirectToAction(string.Format(  
}
```

Model Binders

- Nested Objects binding

- Use name attributes as following "{obj}.{nestedObj}" or use **EditorFor**

```
@model WorkingWithDataMvc.Models.PersonWithAddressViewModel
```

```
@*<input type="text" name="Address.Country" />*@  
@Html.LaberFor(m => m.Address.Country)  
@Html.EditorFor(m => m.Address.Country)
```

```
public class PersonWithAddressViewModel  
{  
    public string Name {get; set; }  
    public Address Address { get; set; }  
}  
public class Address  
{  
    public string City { get; set; }  
    public string Country { get; set; }  
}
```

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public ActionResult NestedObject(PersonWithAddressViewModel person)  
{  
    return this.SetTempDataAndRedirectToAction(string.Format("{0} {1}"))  
}
```

Model Binders

- Collection of objects binding
 - Use name attributes like "**[{index}].{property}**" or use **EditorFor** in a **for** loop

```
for (int i = 0; i < 3; i++)
{
    <h3>
        Person @i
    </h3>
    <div>
        @*<input type="text" name[0].FirstName" />*@
        @Html.LabelFor(m => Model[i].FirstName)
        @Html.EditorFor(m => Model[i].FirstName)
    </div>
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult CollectionOfObjects(IEnumerable<PersonViewModel> persons)
{
    var result = new StringBuilder();
    foreach (var person in persons)
```

```
@model IList<WorkingWithDataMvc.Models.PersonViewModel>
```

Model Binders

- Collection of files binding
 - Use the same name attribute on all input type files as the name of the collection

```
<input type="file" name="files" />
<input type="file" name="files" />
<input type="file" name="files" />
<input type="submit" />
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult CollectionOfFiles(IEnumerable<HttpPostedFileBase> files)
{
    var names = files.Where(f => f != null).Select(f => f.FileName);
    return this.SetTempDataAndRedirectToAction(string.Join(", ",
names));
}
```


Custom Model Binders

```
public class CustomModelBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext)
    {
        NameValueCollection form =
        controllerContext.HttpContext.Request.Form;

        SomeModel myModel = new SomeModel();
        myModel.Property = "value";

        ModelStateDictionary mState = bindingContext.ModelState;
        mState.Add("Property", new ModelState { });
        mState.AddModelError("Property", "There's an error.");

        return myModel; // return your model
    }
}
```

```
public ActionResult Test([ModelBinder(typeof(CustomModelBinder))]SomeModel m)
{
    // ...
    return View();
}
```

Model Validation



Validation

- **Model State**
 - represents the submitted values and errors during a POST
- **Validation Helpers**
 - **Html.ValidationMessageFor()** displays only errors for to the property specified.
 - **Html.ValidationSummary()** reads all errors from the model state and displays them in a bulleted list

Validation Model - Controller

- **ModelState.IsValid** – will give us information about the data validation success
- **ModelState.AddModelError** – doesn't depend on the model

```
[HttpPost]
public ActionResult Edit(Dog dog)
{
    if (ModelState.IsValid)
    {
        if (dog.Age>20)
            ModelState.AddModelError("Age", "Too old dog!");
    }
    ...
    return View();
}
```

Validation Attributes

- **Model validation is done by adding attributes over properties, in model**
- **Belong to System.ComponentModel.DataAnnotations namespace**

```
public class Dog
{
    public int Id { get; set; }

    [Required]
    [MinLength(3)]
    public string Name { get; set; }

    public int Age { get; set; }
}
```

Validation with Annotations

- **[CreditCard]**: Validates the property has a credit card format.
- **[Compare]**: Validates two properties in a model match.
- **[EmailAddress]**: Validates the property has an email format.
- **[Phone]**: Validates the property has a telephone format.
- **[Range]**: Validates the property value falls within the given range.
- **[RegularExpression]**: Validates that the data matches the specified regular expression.
- **[Required]**: Makes a property required.
- **[StringLength]**: Validates that a string property has at most the given maximum length.
- **[Url]**: Validates the property has a URL format.

Validation in controller

- **Model validation is done by adding attributes over properties, in model**
- **Belong to System.ComponentModel.DataAnnotations namespace**

```
[HttpPost]
public IActionResult Create(Dog model)
{
    if (ModelState.IsValid)
    {
        // save the dog in database
        return RedirectToAction("List");
    }

    return View(model);
}
```

Validation - View

- **@Html.ValidationSummary** – output errors
- **@Html.ValidationMessageFor(...)** – outputs validation message for specified property

```
@using (Html.BeginForm())  
{  
    @Html.ValidationSummary(true)  
  
    @Html.LabelFor(m => m.Name)  
    @Html.EditorFor(m => m.Name)  
    @Html.ValidationMessageFor(m => m.Name)  
    @Html.LabelFor(m => m.Price)  
    @Html.EditorFor(m => m.Price)  
    @Html.ValidationMessageFor(m => m.Price)  
}
```

Text box with integrated
client-side validation



Data Binding



Model Binders

- To make easy of handling HTTP request
- Help the populating the parameters in action methods

Rating

Body

**HTTP POST /Review/Create
Rating=7&Body=Great!**

```
public ActionResult Create (Review review)
{
    // ...
}
```

```
class Review
{
    public int Rating;
    public string Body;
}
```

Model Binders

- **Parameter binding**

- The name attribute of the input HTML element should be the same as the name of parameter in the action

```
Html.TextBox("FirstName", "John", new { type = "text" })
```

```
<div>  
    <input type="text" name="FirstName" value="John" />  
</div>
```

**HTTP POST /Binding/UpdateUsingParameter
FirstName=John**

```
[HttpPost]  
public ActionResult UpdateUsingParameter(string firstName)  
{  
    TempData["Success"] = $"{{firstName}}";  
    return RedirectToAction("Index");  
}
```

Model Binders

- Object binding

- Model binder will try to "construct" the object based on the name attributes on the input HTML elements

```
@model WorkinWithDataMvc.Models.PersonViewModel
@Html.EditorFor(m => m.FirstName)
```

```
<input type="text" name="FirstName" value="John" />
```

```
[HttpPost]
public ActionResult UpdateUsingObject(PersonViewModel person)
{
    TempData["Success"] = $"{person.FirstName}";
    return RedirectToAction("Index");
}
```

```
public class PersonViewModel
{
    public string FirstName { get;
set; }
    public string LastName { get;
set; }
    public int Age { get; set; }
}
```

HTTP POST /Binding/UpdateUsingObject
FirstName=John

Model Binders

- Nested Objects binding

- Use name attributes as following "{obj}.{nestedObj}" or use **EditorFor**

```
@model WorkingWithDataMvc.Models.PersonWithAddressViewModel  
  
@Html.LabelFor(m => m.Address.Country)  
@Html.EditorFor(m => m.Address.Country)
```

```
<input type="text" name="Address.Country" />
```

```
[HttpPost]  
public ActionResult UpdateUsingNestedObject(PersonWithAddressViewModel person)  
{  
    TempData["Success"] = $"{{person.Address.Country}}";  
    return RedirectToAction("Index");  
}
```

```
public class PersonWithAddressViewModel  
{  
    public string Name {get; set;}  
    public Address Address { get; set; }  
}  
public class Address  
{  
    public string City { get; set; }  
    public string Country { get; set; }  
}
```

**HTTP POST /Binding/UpdateUsingNestedObject
Address.Country=Romania**

Model Binders

- Collection of primitive types binding

- Use the same name attribute on every input element and the parameter name of the collection in the action (you can use loops)

```
<input type="text" name="firstNames" value="John" />
<input type="text" name="firstNames" value="Jim" />
<input type="text" name="firstNames" value="Steve" />
<input type="text" name="firstNames" value="Scott" />
<input type="submit" />
```

HTTP POST /Binding/CollectionOfPrimitiveTypes

firstNames=John&firstNames=Jim&firstNames=Steve&firstNames=Scott

```
[HttpPost]
public ActionResult CollectionOfPrimitiveTypes(IEnumerable<string> firstNames)
{
    TempData["Success"] = $"{string.Join(',', firstNames)}";
    return this.RedirectToAction("Index");
}
```

Model Binders

- Collection of objects binding

- Use name attributes like "[{index}].{property}" or use **EditorFor** in a **for** loop

```
@for (int i = 0; i < 3; i++)
{
    <h3>Person @i</h3>
    <div>
        <input type="text" name="name[@i].FirstName"
    />
    </div>
}
```

```
@model IList<PersonViewModel>

@for (int i = 0; i < 3; i++)
{
    <h3>Person @i</h3>
    <div>
        @Html.EditorFor(m => m.FirstName)
    </div>
}
```

```
[HttpPost]
public ActionResult CollectionOfObjects(IEnumerable<PersonViewModel> persons)
{
    var result = new StringBuilder();
    foreach (var person in persons)
    {
        result.Append(person.FirstName);
    }
}
```

Model Binders

- Collection of files binding

- Use the same name attribute on all input type files as the name of the collection

```
<input type="file" name="files" />
<input type="file" name="files" />
<input type="file" name="files" />
<input type="submit" />
```

```
[HttpPost]
public ActionResult CollectionOfFiles(IEnumerable<HttpPostedFileBase> files)
{
    var names = files.Where(f => f != null).Select(f => f.FileName);
    TempData["success"] = string.Join(',', names);
    return RedirectToAction("Index");
}
```


Model Binding Rules

- **By default, Web API uses the following rules to bind parameters:**
 - **If the parameter is a "simple" type, Web API tries to get the value from the URI**
 - **Simple types include the .NET primitive types (int, bool, double, and so forth), plus TimeSpan, DateTime, Guid, decimal, and string, plus any type with a type converter that can convert from a string. (More about type converters later.)**
 - **For complex types, Web API tries to read the value from the message body, using a media-type formatter.**

Model Binding Behavior

- **Data binding source order:**
 - **Form values:** Values in the form (via HTTP POST).
 - **Route values:** Values provided by the Routing system. (only for primitive types)
 - **Query string:** Values found in the URL's query string
- **Binding a value that is supposed to be an integer, it now uses the default value of the type (0 for integers) if no value is posted. This saves you from null exceptions as well as the need to assign a default value yourself. See BindRequired.**

Model Binding Behavior

- **[BindRequired]**: This attribute adds a model state error if binding cannot occur.
- **[BindNever]**: Tells the model binder to never bind to this parameter.
- **[ModelBinder]**: Used to override the default model binder, binding source and name.

CLI



Dotnet CLI (Command Line Interface)

- In .NET Core, the entire set of fundamental development tools—those used to build, test, run, and publish applications—is also available as command-line applications.

```
dotnet [host-options] [command] [arguments] [common-options]
```

```
dotnet dev-certs https --trust
```

```
dotnet --info  
dotnet --version  
dotnet --verbose  
  
dotnet <command> --help
```

Dotnet Commands

Command	Description
new	Creates a new .NET Core application starting from one of the available templates. Default templates include console applications as well as ASP.NET MVC applications, test projects, and class libraries. Additional options let you indicate the target language and the name of the project.
restore	Restores all the dependencies of the project. Dependencies are read from the project file and restored as NuGet packages consumed from a configured feed.
build	Builds the project and all its dependencies. Parameters for the compilers (such as whether to build a library or an application) should be specified in the project file.
run	Compiles the source code if required, generates an executable and runs it. It relies on the command build for the first step.
test	Runs unit tests in the project using the configured test runner. Unit tests are class libraries with a dependency on a particular unit test framework and its runner application.
publish	Compiles the application if required, reads the list of dependencies from the project file and then publishes the resulting set of files to an output directory.
pack	Creates a NuGet package out of the project binaries.
migrate	Migrates an old project.json-based project to a msbuild-based project.
clean	Cleans the output folder of the project.