ASP.NET MVC Essentials

Routing, Controllers, Actions, Views, Areas…

Table of Contents

UNIVERSITY of GREENWICH

Alliance with FPT Education

SCAFFOLDING

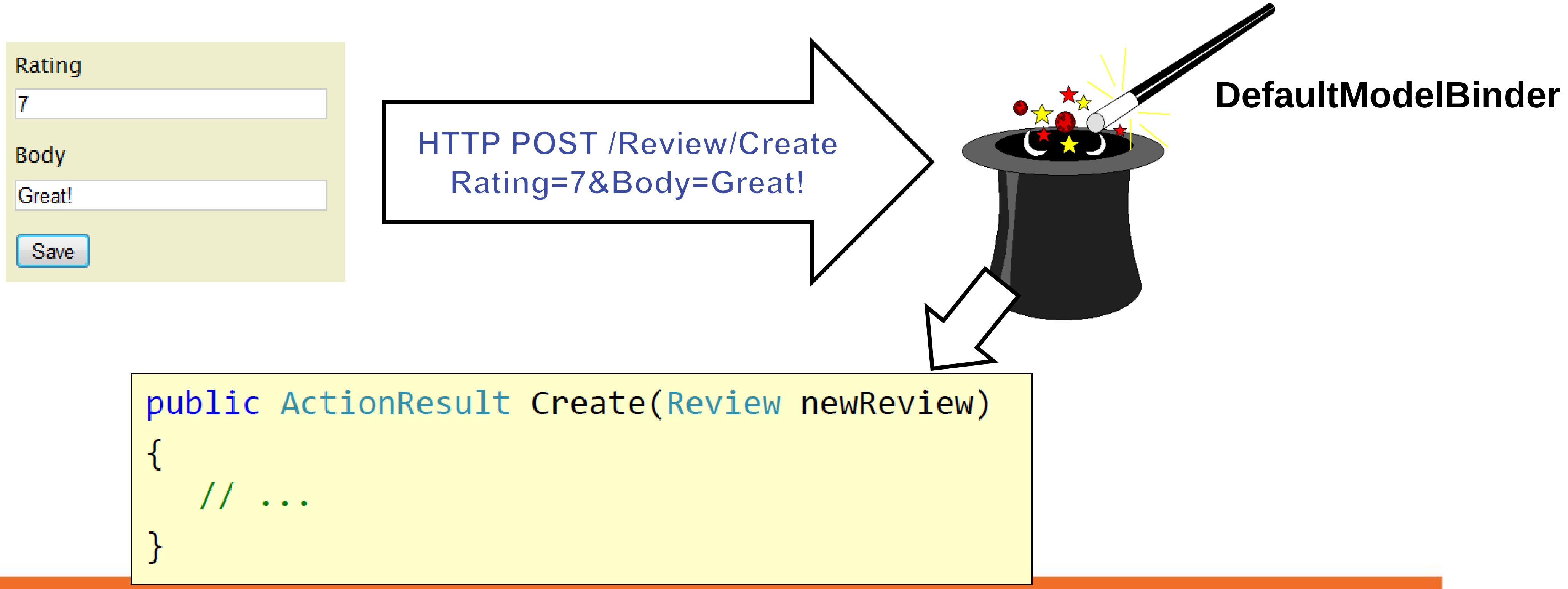
What is ASP.NET Scaffolding?

- Code generation framework for ASP.NET
  - When you want to quickly add boilerplate code that interacts with data models
- Developer productivity enhancer
  - Can reduce the amount of time to develop standard data operations in your project
- Enables customization
  - Provides an extensibility mechanism to customize generated code
- VS 2013 includes pre-installed code generators for MVC, and Web API

MODEL BINDERS

Model Binders

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



Rating

7

Body

Great!

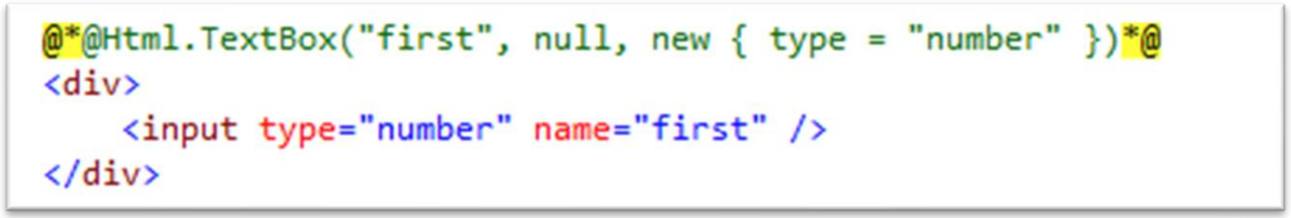
Save

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

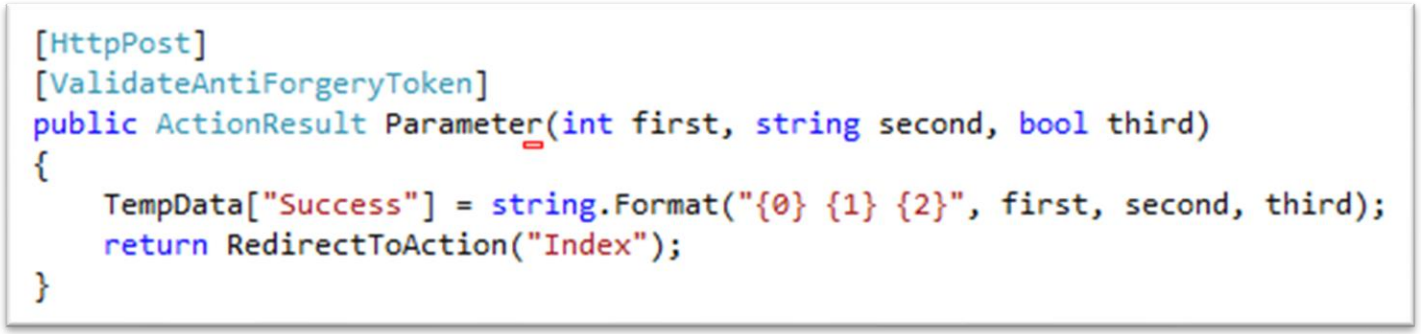
```
public ActionResult Create(Review newReview)
{
    // ...
}
```

Model Binders

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



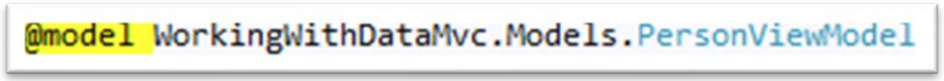
```
@*@Html.TextBox("first", null, new { type = "number" })*@
<div>
    <input type="number" name="first" />
</div>
```



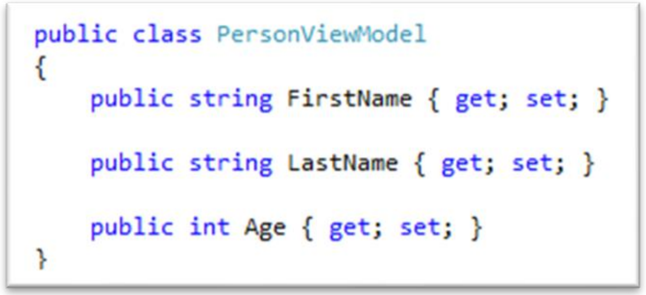
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Parameter(int first, string second, bool third)
{
    TempData["Success"] = string.Format("{0} {1} {2}", first, second, third);
    return RedirectToAction("Index");
}
```

Model Binders

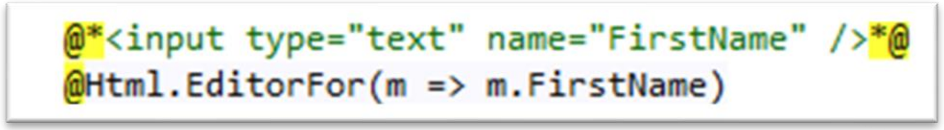
- Object binding
  - Model binder will try to "construct" the object based on the name attributes on the input HTML elements



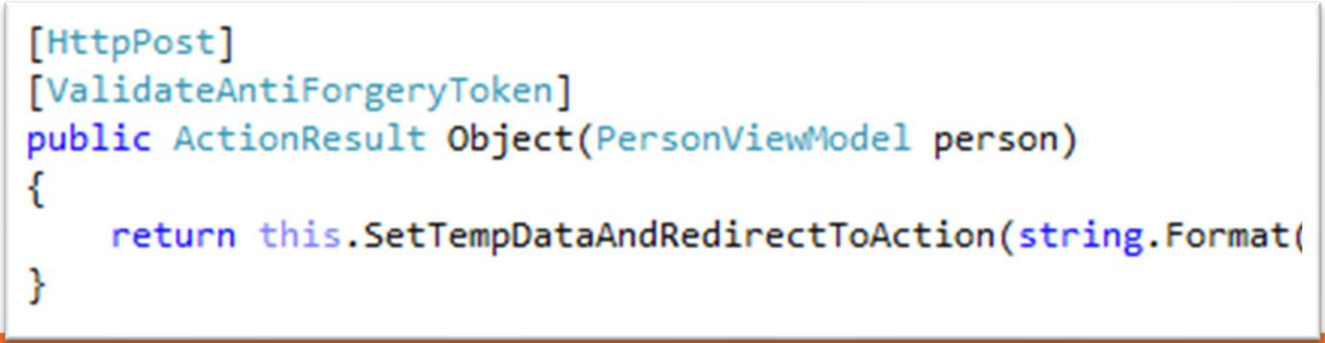
```
@model WorkingWithDataMvc.Models.PersonViewModel
```



```
public class PersonViewModel
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

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



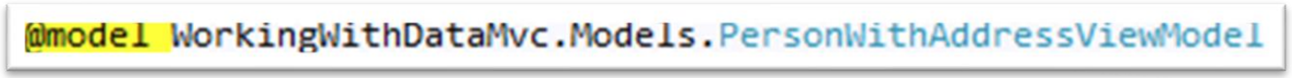
```
@*<input type="text" name="FirstName" />*@
@Html.EditorFor(m => m.FirstName)
```



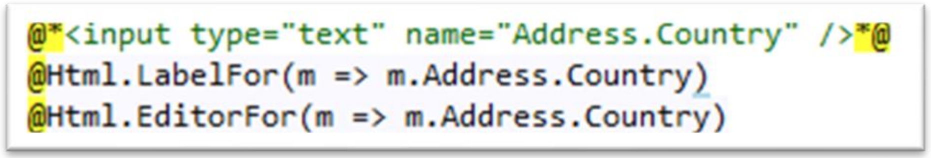
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Object(PersonViewModel person)
{
    return this.SetTempDataAndRedirectToAction(string.Format(
}
```

Model Binders

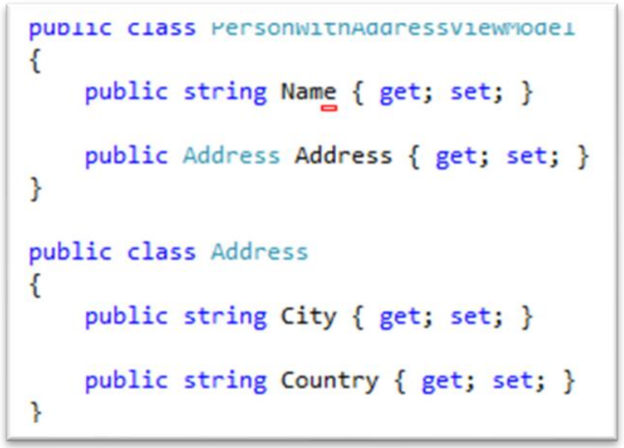
- Nested Objects binding
  - Use name attributes as following "{obj}.{nestedObj}" or use EditorFor



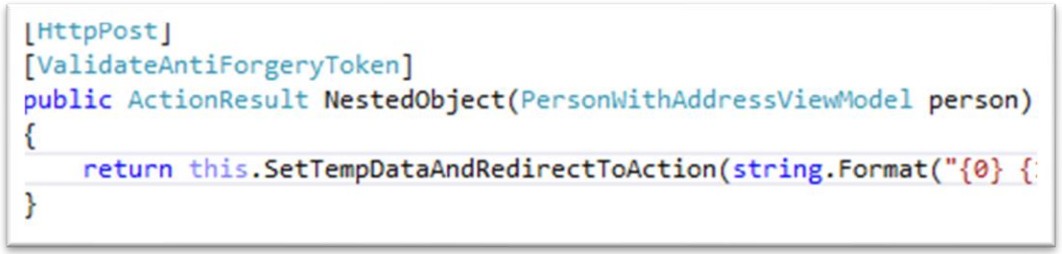
```
@model WorkingWithDataMvc.Models.PersonWithAddressViewModel
```



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



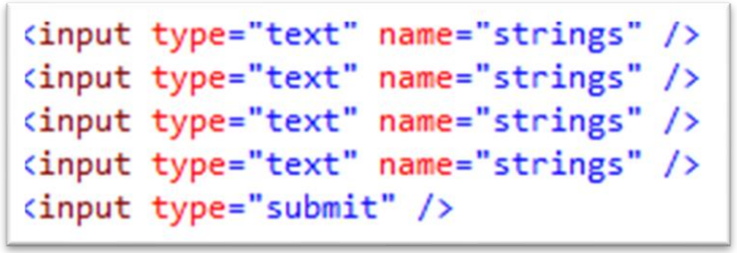
```
public class PersonWithAddressViewModel
{
    public string Name { get; set; }

    public Address Address { get; set; }
}

public class Address
{
    public string City { get; set; }

    public string Country { get; set; }
}
```



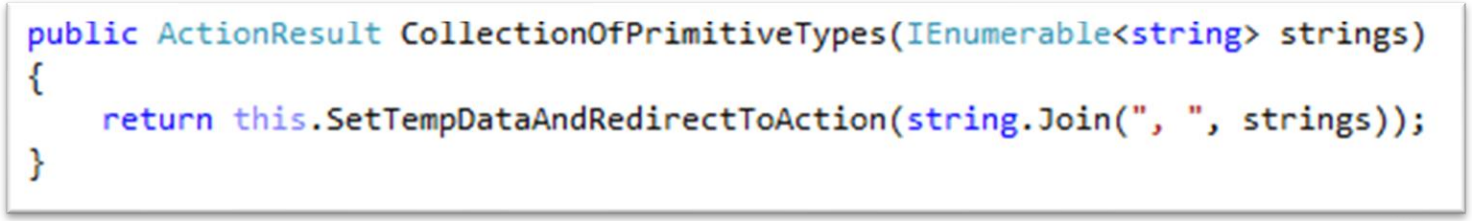
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult NestedObject(PersonWithAddressViewModel person)
{
    return this.SetTempDataAndRedirectToAction(string.Format("{0} {
}
```

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)



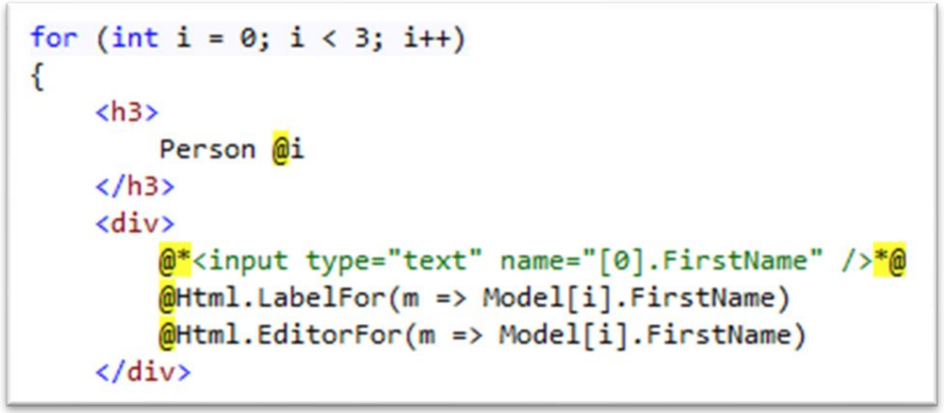
```html
<input type="text" name="strings" />
<input type="text" name="strings" />
<input type="text" name="strings" />
<input type="text" name="strings" />
<input type="submit" />
```



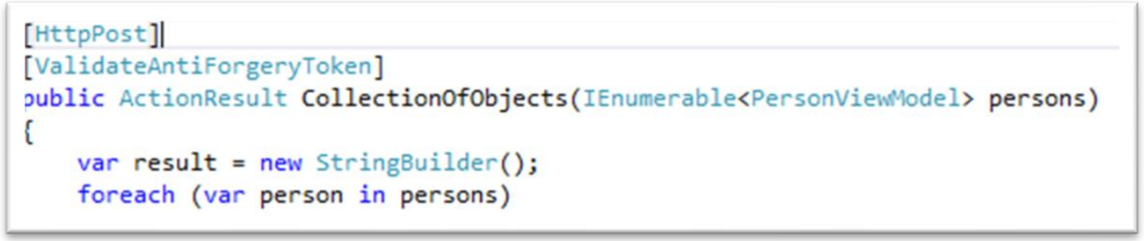
```csharp
public ActionResult CollectionOfPrimitiveTypes(IEnumerable<string> strings)
{
    return this.SetTempDataAndRedirectToAction(string.Join(", ", strings));
}
```

Model Binders

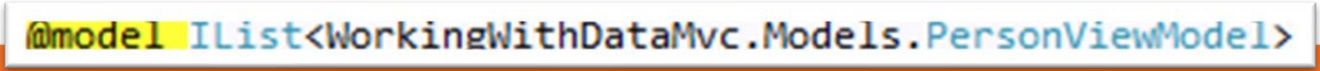
- Collection of objects binding
  - Use name attributes like "[{index}].{property}" or use EditorFor in a



```
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>
```



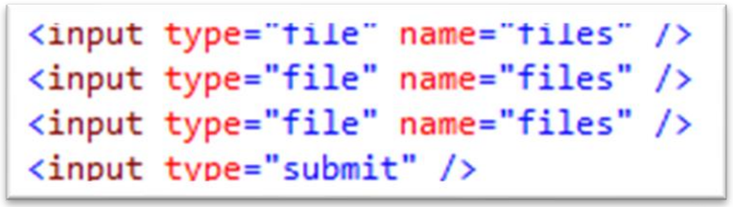
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult CollectionOfObjects(IEnumerable<PersonViewModel> persons)
{
    var result = new StringBuilder();
    foreach (var person in persons)
```



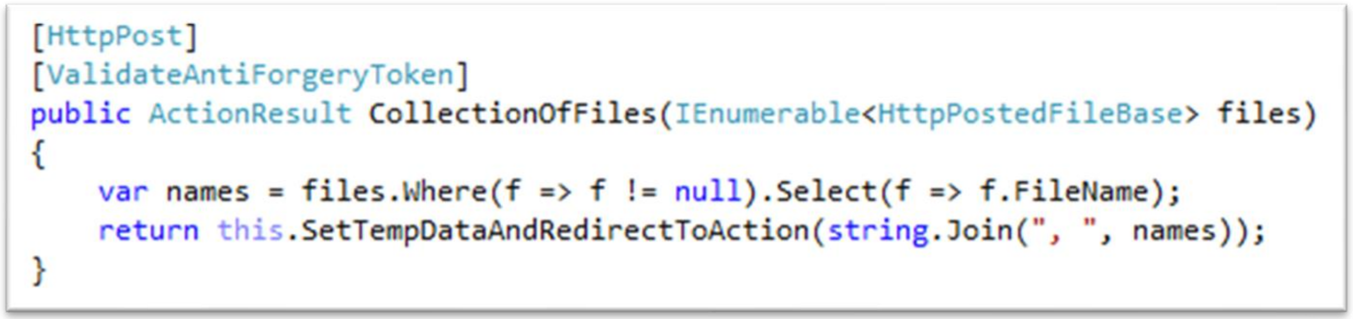
```
@model IList<WorkingWithDataMvc.Models.PersonViewModel>
```

Model Binders

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

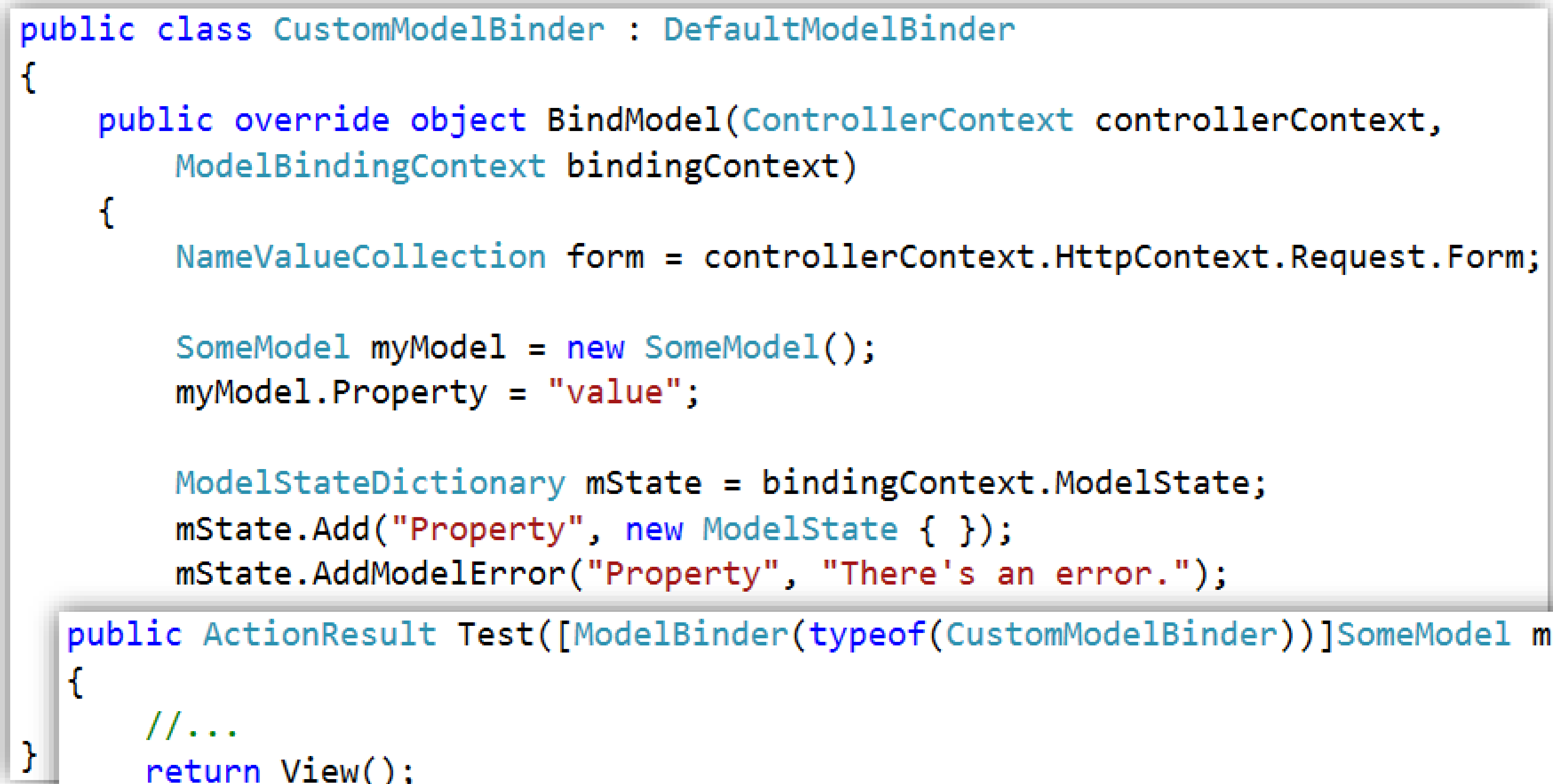


```
<input type="file" name="files" />
<input type="file" name="files" />
<input type="file" name="files" />
<input type="submit" />
```

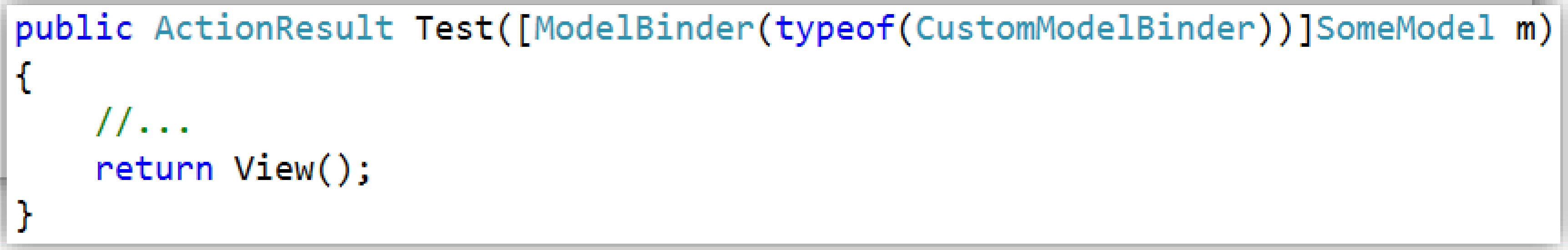


```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult CollectionOfFiles(IEnumerable<HttpPostedFileBase> files)
{
    var names = files.Where(f => f != null).Select(f => f.FileName);
    return this.SetTempDataAndRedirectToAction(string.Join(", ", names));
}
```

Custom Model Binder



```
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.");
}
```

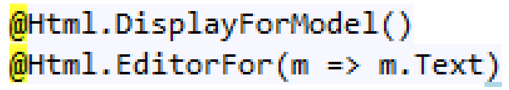


```
public ActionResult Test([ModelBinder(typeof(CustomModelBinder))]SomeModel m)
{
    //...
    return View();
}
```

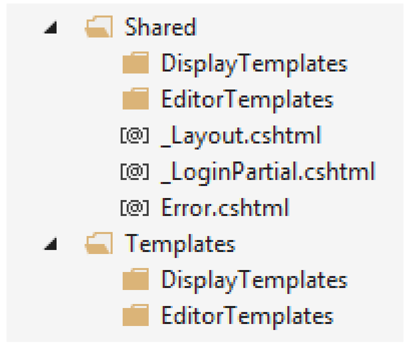
DISPLAY & EDITOR TEMPLATES

Templates

- ASP.NET MVC comes with helpers methoc
  - `DisplayFor()`, `DisplayForModel()`
  - `EditorFor()`, `EditorForModel()`
- There are default implementation
- Easily to be configured
- Create folders "**DisplayTemplates**" and "**EditorTemplates**" in the "**Shared**" folder or in the "**Views/{Controller}**" folder



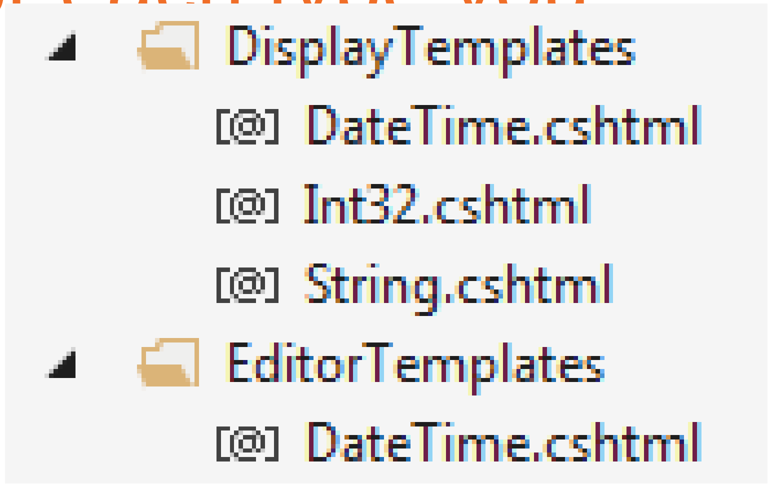
```
@Html.DisplayForModel()
@Html.EditorFor(m => m.Text)
```



```
◢ Shared
    DisplayTemplates
    EditorTemplates
    _Layout.cshtml
    _LoginPartial.cshtml
    Error.cshtml
◢ Templates
    DisplayTemplates
    EditorTemplates
```

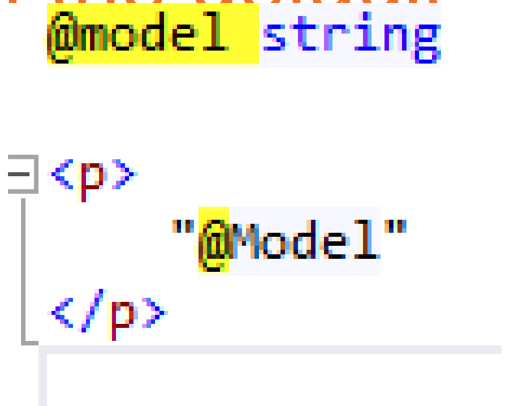
Custom Templates

- In the two new folders create a view for each type you want
  - `string` -> **String.cshtml**
  - `int` -> **Int32.cshtml**
  - `DateTime` -> **DateTime.cshtml**
  - `Student` -> **Student.cshtml**
- The name of the files must reflect the data types and the **@model** in them



Custom Templates

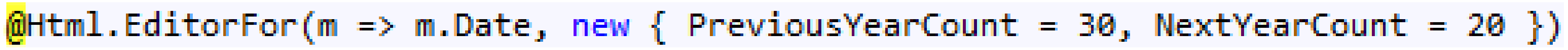
- These view are normal view files
- The framework will start using them instead of the default implementations
- For example in the String.cshtml
- Now all strings will be in paragraph element and will have quotes surrounding them
- DisplayFor, EditorFor -> for properties
- DisplayForModel, EditorForModel -> for model



```
@model string

<p>
    "@Model"
</p>
```

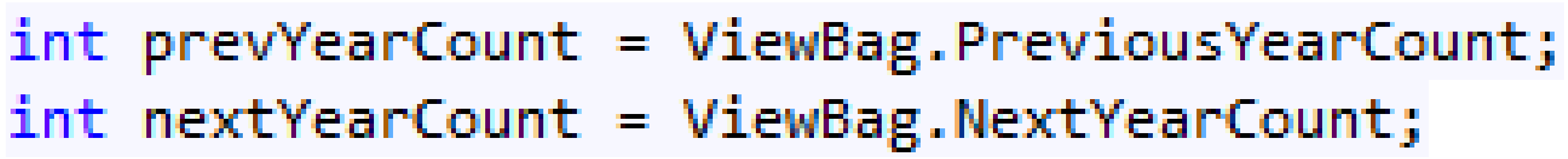
Custom Templates

- Passing additional information to the templates
  - There is an object "additionalViewData" in the helper methods as parameter
  - You can pass anything there as anonymous type



```
@Html.EditorFor(m => m.Date, new { PreviousYearCount = 30, NextYearCount = 20 })
```

  - And get the values from the ViewData/ViewBag



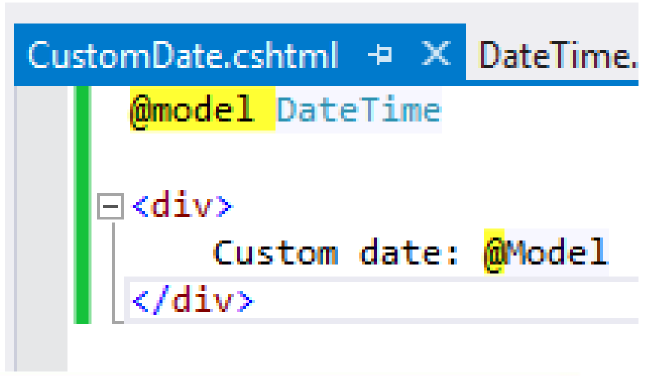
```
int prevYearCount = ViewBag.PreviousYearCount;
int nextYearCount = ViewBag.NextYearCount;
```

Custom Template Name

- Sometimes you need two templates for one data type
  - Create the template with custom name
  - Decorate the property in the model with the UIHint attribute specifying the template name
  - You can set the name in the helpers too

```
[UIHint("CustomDate")]
public DateTime AnotherDate { get; set; }
```

```
@Html.DisplayFor(m => m.AnotherDate, "CustomDate")
```



```
@model DateTime

<div>
    Custom date: @Model
</div>
```

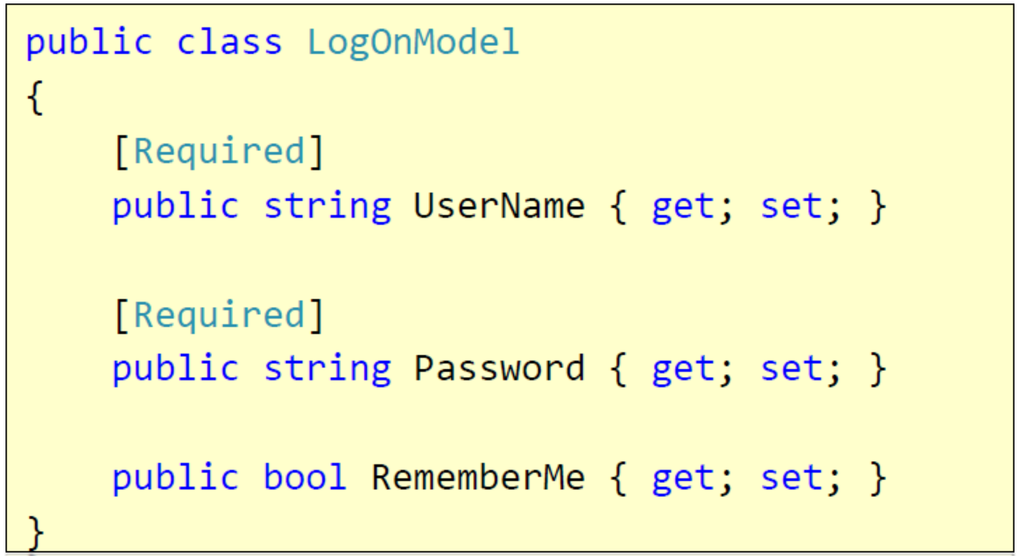
UNIVERSITY of GREENWICH

Alliance with FPT Education

DATA VALIDATION

Validation with Annotations

- Attributes are defined in
  - System.ComponentModel.DataAnnotations
- Covers common validation patterns
  - Required
  - StringLength
  - Regex
  - Range



```
public class LogOnModel
{
    [Required]
    public string UserName { get; set; }

    [Required]
    public string Password { get; set; }

    public bool RememberMe { get; set; }
}
```

Data Validation Attributes

Attribute	Description
Compare	Checks whether two specified properties in the model have the same value.
CustomValidation	Checks the value against the specified custom function.
EnumDataType	Checks whether the value can be matched to any of the values in the specified enumerated type.
Range	Checks whether the value falls in the specified range. It defaults to numbers, but it can be configured to consider a range of dates, too.
RegularExpressi on	Checks whether the value matches the specified expression.
Remote	Makes an Ajax call to the server, and checks whether the value is acceptable.
Required	Checks whether a non-null value is assigned to the property. It can be configured to fail if an empty string is assigned.
StringLength	Checks whether the string is longer than the specified value

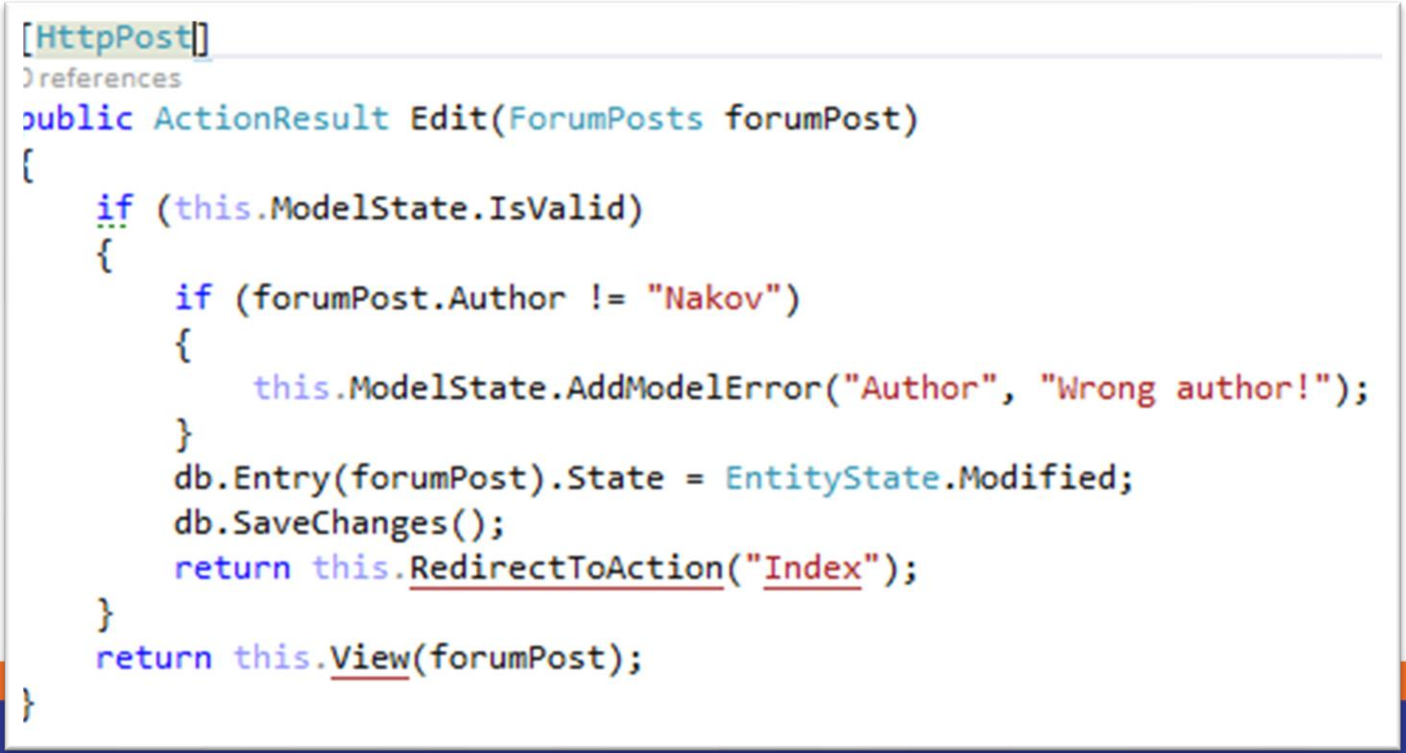
Custom Validation

- Custom attributes
- Inherit **ValidationAttribute**

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class MinLengthAttribute : ValidationAttribute
{

    // …

    public override bool IsValid(object value)
    {
        string valueAsString = value as string;
        return (valueAsString != null &&
                valueAsString.Length >= _minCharacters);
    }
}
```

Validating Model – Controller

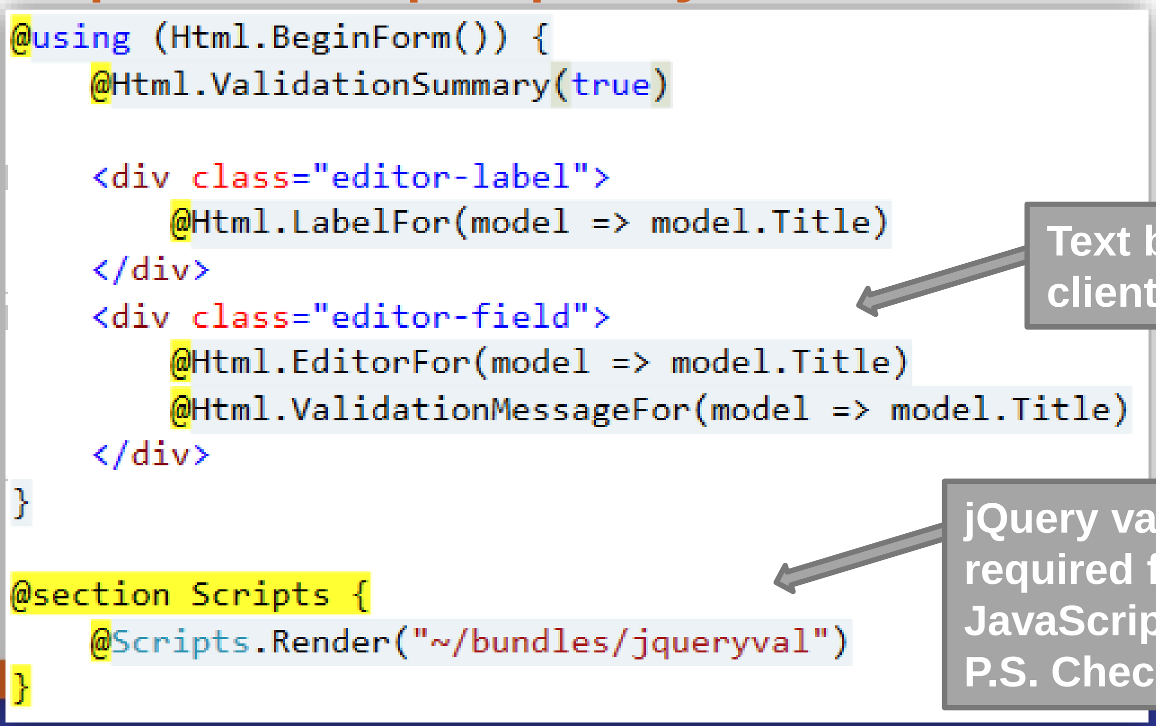
- `ModelState.IsValid` – will give us information about the data validation success
- `ModelState.AddModelError` – custom error



```
[HttpPost]
) references
public ActionResult Edit(ForumPosts forumPost)
{
    if (this.ModelState.IsValid)
    {
        if (forumPost.Author != "Nakov")
        {
            this.ModelState.AddModelError("Author", "Wrong author!");
        }
        db.Entry(forumPost).State = EntityState.Modified;
        db.SaveChanges();
        return this.RedirectToAction("Index");
    }
    return this.View(forumPost);
}
```

Validating Model – View

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



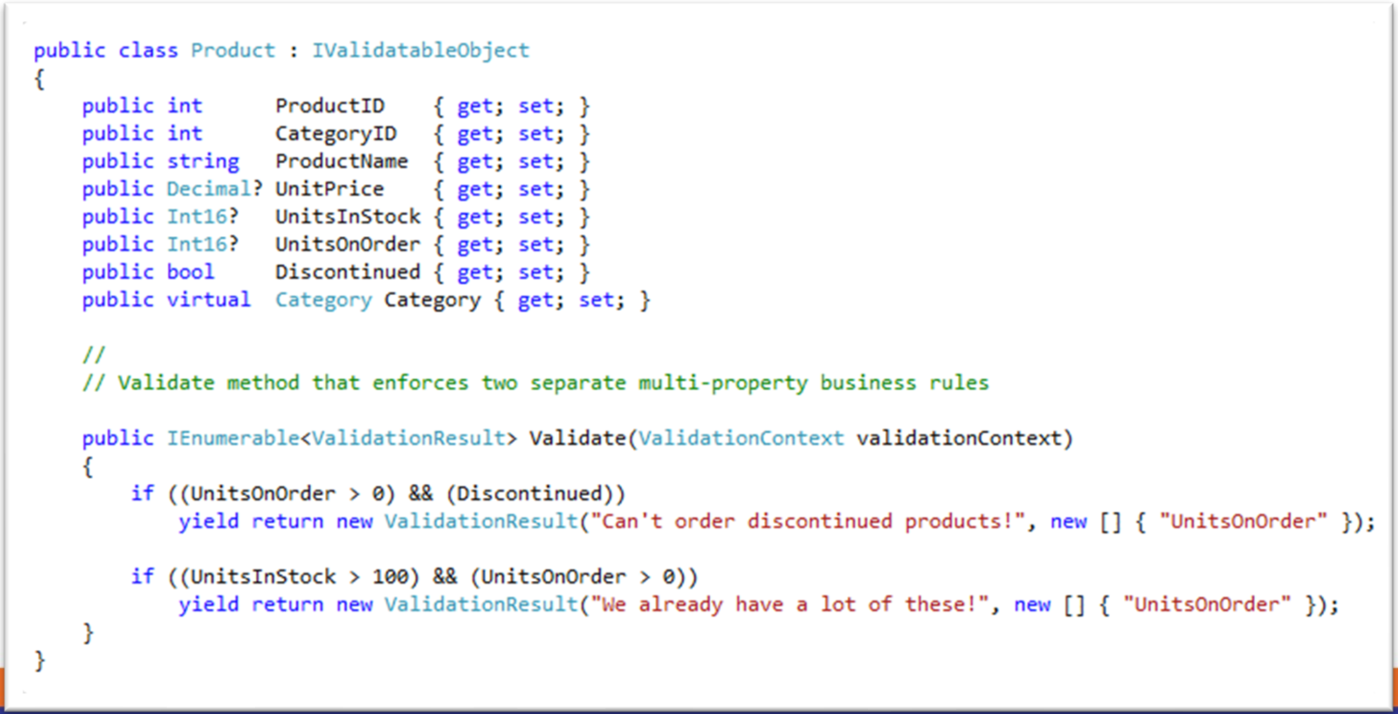
```
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <div class="editor-label">
        @Html.LabelFor(model => model.Title)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title)
    </div>
}

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Text box with integrated client-side validation

jQuery validation library required for unobtrusive JavaScript validation
P.S. Check web.config

Class-Level Model Validation

- Your model should implemented **IValidatableObject**
- From now on, MVC (works with EF too) will validate the object by your custom rules



```
public class Product : IValidatableObject
{
    public int      ProductID    { get; set; }
    public int      CategoryID   { get; set; }
    public string   ProductName  { get; set; }
    public Decimal? UnitPrice    { get; set; }
    public Int16?   UnitsInStock { get; set; }
    public Int16?   UnitsOnOrder { get; set; }
    public bool     Discontinued { get; set; }
    public virtual  Category Category { get; set; }

    //
    // Validate method that enforces two separate multi-property business rules

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if ((UnitsOnOrder > 0) && (Discontinued))
            yield return new ValidationResult("Can't order discontinued products!", new [] { "UnitsOnOrder" });

        if ((UnitsInStock > 100) && (UnitsOnOrder > 0))
            yield return new ValidationResult("We already have a lot of these!", new [] { "UnitsOnOrder" });
    }
}
```

OTHER ANNOTATIONS

Display / Edit Annotations

Attribute	Description
DisplayColumn	Specify the property of a model class for simple text display.
HiddenInput	Render value in a hidden input (when editing).
UIHint	Specify the name of the template to use for rendering.
DataType	Common templates (email, password, URL, currency)
ReadOnly	Specify a read-only property (for model binding).
DisplayFormat	Format strings and null display text
ScaffoldColumn	Turn off display and edit capabilities
DisplayName	Friendly name for labels
Bind	Tells the model binder which properties to include/exclude

UNIVERSITY of GREENWICH

Alliance with FPT Education

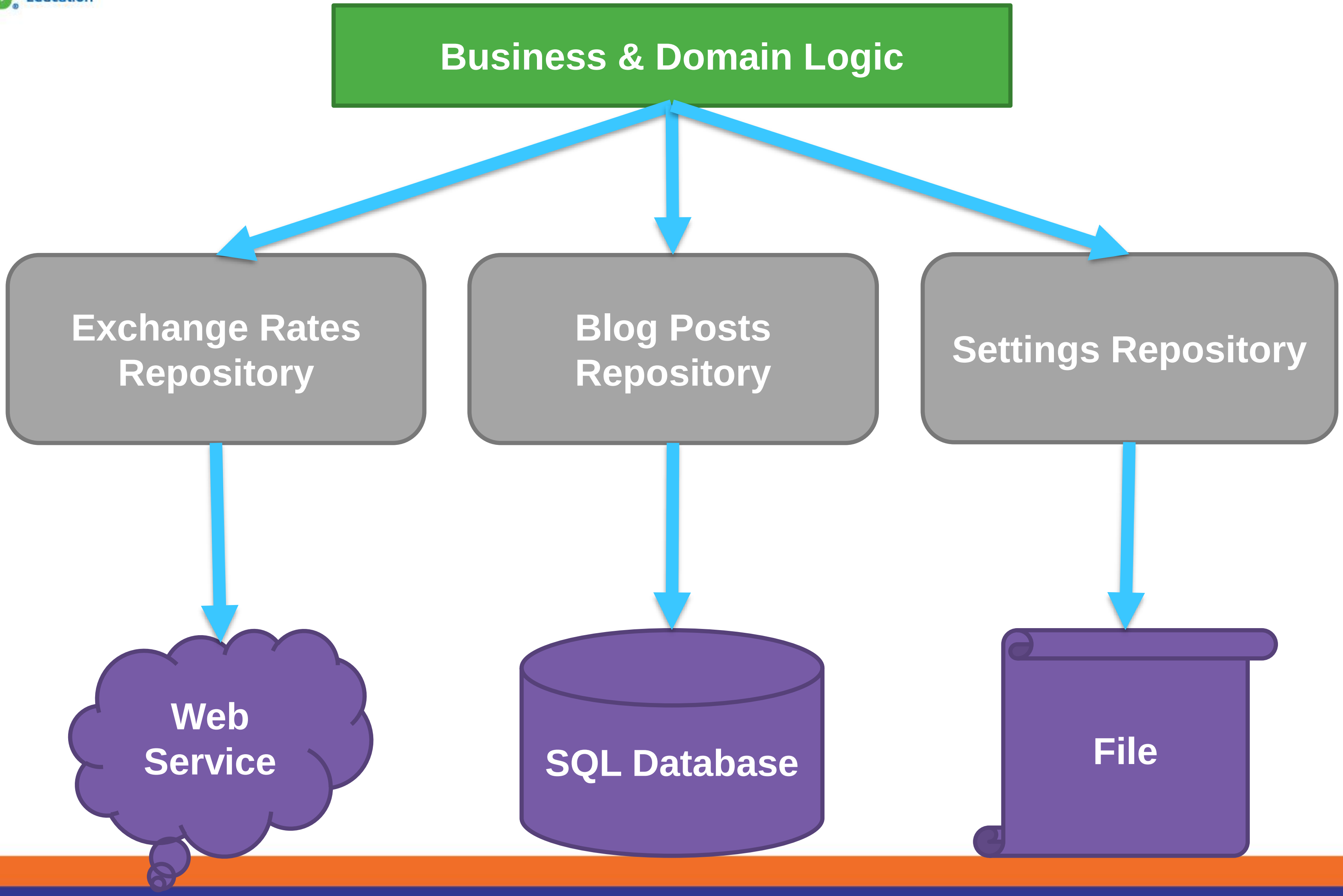
WORKING WITH DATA SOURCE

Repository pattern and Unit of Work pattern

Repository Pattern

- Separate business code from data access
  - Separation of concerns
  - Testability
- Encapsulate data access
- Increased level of abstraction
  - More classes, less duplicated code
  - Maintainability, Flexibility, Testability
- Generic repositories
  - IRepository<T>

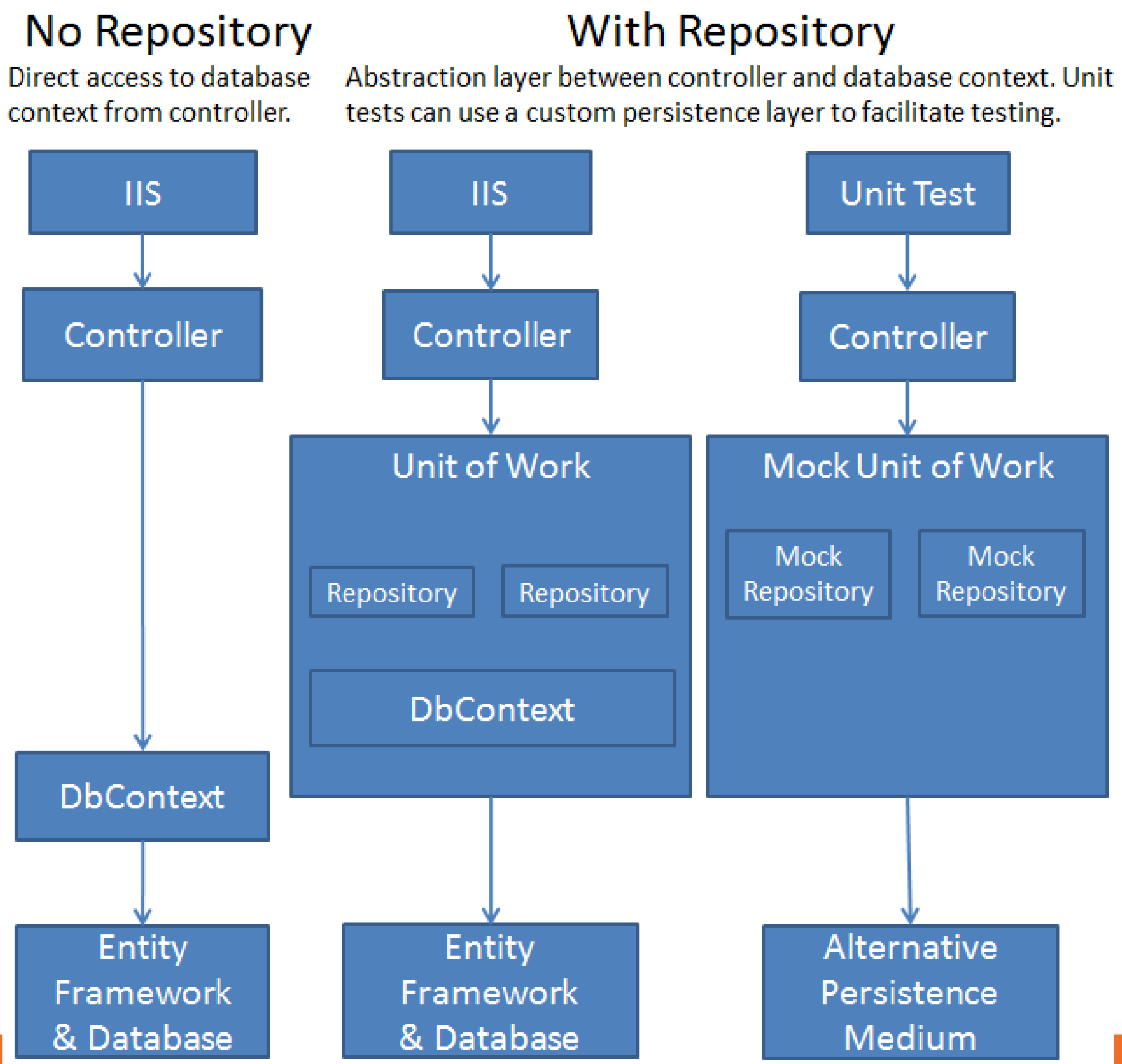
Repository Pattern (2)



Unit of Work

- Track changes in persistent objects
  - Efficient data access
  - Manage concurrency problems
  - Manage transactions
- Keep business logic free of data access code
- Keep business logic free from tracking changes
- Allow business logic to work with logical transactions

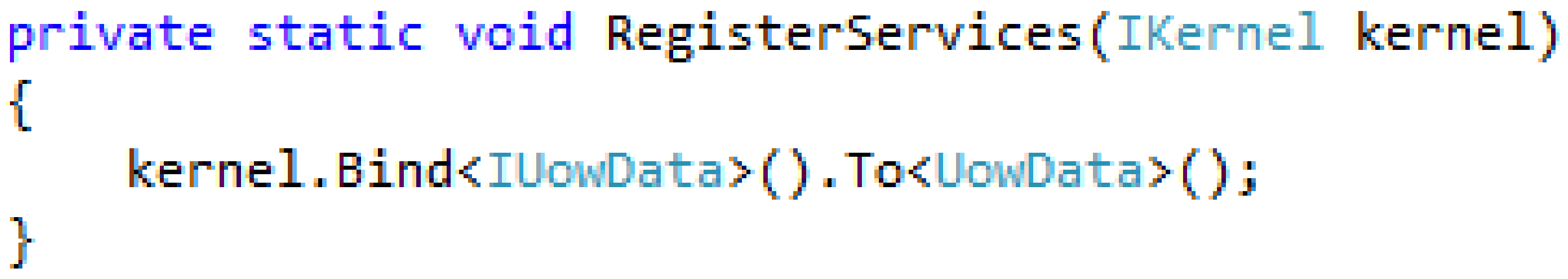
Repository and UoW Patterns in an ASP.NET MVC



No Repository
Direct access to database context from controller.
With Repository
Abstraction layer between controller and database context. Unit tests can use a custom persistence layer to facilitate testing.
IIS
Controller
DbContext
Entity Framework & Database
IIS
Controller
Unit of Work
Repository
Repository
DbContext
Entity Framework & Database
Unit Test
Controller
Mock Unit of Work
Mock Repository
Mock Repository
Alternative Persistence Medium

Ninject IoC

- You may want to use IoC for dependency inversion
- Ninject is quite easy to do
- Install Ninject.MVC5 from NuGet
- In App_Data/NinjectWebCommon add your bindings in RegisterServices method



```
private static void RegisterServices(IKernel kernel)
{
    kernel.Bind<IUowData>().To<UowData>();
}
```