Taken from: *~~Step-by-Step ASP.NET MVC Tutorial for Beginners –~~* ~~from the~~ *~~Programming with Mosh~~* ~~YouTube channel~~

*The Complete ASP.NET MVC 5 Course on Udemy* – from Mosh

# Section 1 – Getting Started

## Architectural pattern

MVC is a development pattern, which has not only been adopted by ASP.NET, but also by Ruby on Rails and Express

* **Model**: application data and behavior, independent of the UI
  + in the course application, this consist of classes like Movie, Customer, Rental, Transaction (POCO's)
  + these classes have properties and methods that represent the application state and rules, not the user interface
* **View**: the HTML markup that we display to users
* **Controller**: a class that handles the HTTP request by way of its methods =>=> which are here called **Actions**.
* for example, when receiving an HTTP request, a controller will be selected to handle it, which will for example get data from the model, put them in a view and return in to the client or browser
* **Router**: this selects the correct Controller that matches an HTTP request (process determined by rules)

## Solution Structure

Directories:

* **App\_Data** : database file goes there
* **App\_Start** : classes that are called when the application is started
  + **RouteConfig.cs** : here the routing rules are configured, like **url pattern,** and sets some default values for handling an incomplete url . When working with the **ID parameter,** you can pass in a specific resource id
* **Content** : here .css files or images are stored, and any other client side assets
* **Controllers**:
  + AccountController.cs : in the sample template, this handles user authentication / profiles
* **fonts**: not important for now
* **Models**: classes live here (only domain classes! Not view models!)
* **Scripts**: this is where the javascript files go
* **Views**: stores the views, named after the controllers in the application + 1 shared (shared view stuff)
  + **IMPORTANT:** ASP will look for a view in the same name as the controller, minus the word ‘Controller’

Miscellaneous files:

* Favicon.ico : icon of the app in the browser
* Global.asax.cs : this class bootstraps the application
* Packages.config: used by Nuget packagemanager
* Startup.cs: will replace Global.asax in the future (.NET Core?)
* Web.config : contains the configuration of the application
  + Most important:
    - ConnectionStrings: database connections
    - AppSettings: app configurations

## Controllers

When creating a new Controller, just right click the Controller file and select Add => new controller. You will then be presented with several templates which are called **scaffolding**. The name you enter **must end with ‘Controller’!** It always inherits from Controller class.

## Views

When creating a view, you can create a **partial view**, **not a complete page**, which creates a kind of widget that can be used on different views. Another option is **use a layout page,** which allows you to select a sort of template, which can for example provide the same “look and feel” on all your pages. This would typically be under \**Views\Shared**

The view is then created as a **.cshtml file** which mixes html and C# (the latter prefixed with an *@* sign. The default part sets two properties: the **ViewBag.Title** (which will be the title of the page in the browser) and the reference to the selected **layout**

When you type *Model*in a C# block on the view, you access its Model property, which allows it to access the Model that has been passed to it in the Controller. However, the type of the Model property is *dynamic*, so you must include a directive a the top of the .cshtml file to specify its type (which contains the fully specified name of the model class

## Useful stuff

Type (snippet) *prop* and hit Tab to **create a property** => enter its type => hit Tab again to name it

Hit *Ctrl Tab* to switch between open tabs

If you press *Ctrl F5* to run debug, it will take you to the view page where you currently are in Visual Studio

In resharper, **rename** something by clicking on the text and press F2

You can **make method parameters nullable** by adding a question mark behind the type when required, like for ints

Another useful code snippet quick-creates an Action in a Controller: *mvcaction4* + Tab

Putting an @ sign in front of a string means *verbatim*, which has the effect that you don’t need to do stuff like escape backslashes. Useful when defining RegEx!

In the NPM Console, *cls* clears out the terminal.

## Changing the CSS template

The standard ASP.NET MVC template app uses Bootstrap as CSS framework. You can find different free templates for Bootstrap here: <https://bootswatch.com/>

From there, select a .css template you like, download the .css file for it and save it in your project to the content folder. Then, add it to the solution.

Then, go to the App\_Start folder and open up the BundleConfig.cs file. This is where **bundles of client side assets** are defined. Multiple assets can be bundled up and thereby reduce the number of HTTP requests required for the application, making the page load faster. These definitions can work with **pattern matching**.

For changing the .css template, look for the bundle that references the \Content\ directory. The \site.css file contains some generic styles for the application. The one we want to change here is bootstrap.css.

**Afterwards, recompiling is required**, so hit Ctrl-F5.

# Section 2 – ASP.NET MVC Fundamentals

## Action Results

This is what an Action in a Controller returns, its **outputs**. This type **ActionResult** is the base class for all Action Results in ASP.NET MVC.In the example MoviesController.Random action, the View() method is inherited from the **Controller base class.** Theoretically in this case, you could reset the return type of the method to **ViewResult** (which inherits from ActionResult), but depending on internal logic, the Action might return different ActionResults. Therefore, this generic approach can be better, although ViewResult simplifies unit testing.

The subtypes of ActionResult are many: ViewResult, RedirectResult, JsonResult or FileResult are just a few of these. Finally, there is also EmptyResult, which is like the *void* return typein regular C#. For this one though, **there is no helper method in the Controller base class**. Therefore, use **return new EmptyResult();**

You can also do **return RedirectToAction(**“action”,”controller”**);** as an ActionResult. When you wish to pass additional parameters, you can use an **anonymous object**, whose properties will be passed as a query string.

## Action Parameters

These are the **inputs** to Controller Actions. ASP.NET MVC automatically maps request data to Action method parameter values through the process of **Parameter Binding**. This mapping is done on a **name basis**! The request data can be contained in the following sources:

* In the URL */movies/edit/1\**
* In the query string /*movies/edit?id=1*
* In the data posted by a form data *id=1*

In this case, the parameter name will be as specified in the **default route**  **in RouteConfig.cs**, in this case ‘id’. This means that if your Action Parameter has any other name **ASP.NET will not be able to map it**!

## Convention-based (custom) routing

Apart from the default route, you might want to define additional routes, or **custom routes**, to be able to **pass parameters as part of the URL**, instead of having to pass them as a query string.

Adding a new route is done in the RouteConfig.cs class. **The order of the routes listed here is important**! The order goes from **most specific to most generic**. These will be iterated through until one matches the request url.

When you want to **enforce constraints on the url parameters**, you can add these in an anonymous object as **RegEx** in the custom route definition.

**There is a problem with this approach to adding custom routes.** For one, should you rename your action, this is not updated in RouteConfig.cs and your route will bring. Also, if for a large application you add many routes, this file will become a mess.

## Attribute Routing (in custom routing)

The preferred modern way is using **attribute routing.** Enable this by removing your convention-based routing statements in RouteConfig.cs and **adding routes.MapMvcAttributeRoutes();**

Then, back in the Controller, **add the Route decorator to the class** and **declare the custom route in the decorator**, including the RegEx (this is not a verbatim string, so escape necessary characters). Example:

[Route("movies/released/{year:regex(\\d{4})}/{month:regex(\\d{2}),range(1,12)}")]

public ActionResult ByReleaseDate(int year, int month)

{ }

There are multiple other constraints available, like *min, max, int, guid* or *maxlength* (for strings). Google for more options :)

## Passing Data to Views

In the View receiving a model, when working with Lists, it is better to code to an Interface like so:

@model IEnumerable<Vidly.Models.Customer>

Earlier, we passed a movie object to the View from the MovieController.Random Action by passing it as an argument to the View() method in the return statement.

A different way is to use the **ViewData property of the Controller class**, which is of type **ViewDataDictionary**. This can then be accessed on the view (.cshtml). However, because each item in the Dictionary is of type *object*, it requires explicit casting, which adds overhead. Also, this is a fragile method, since the string that accesses the key **must be the same**, even when it changes in for example the Controller. Example:

<h2>@( ((Movie)ViewData["Movie"]).Name)</h2>

Another approach is using the ViewBag “**magic properties**”, whose type is dynamic. But this is not type-safe and naming is still a vulnerability.

Using the method below is safer.

<h2>@Model.Name</h2>

In effect, what you are doing is

var movie = new Movie() { Name = "Shrek" };

var ViewResult = new ViewResult();

ViewResult.ViewData.Model = movie;

return ViewResult;

An important thing to note is that **ViewData is a ViewDataDictionary,** which means that you can use it with **key-value pairs**, or you can use its **Model property as an** **object**, which is the preferred way.

## View Models

When combining data from multiple domain models (i.e. *customer* and *movie*), you need to pass in **multiple models, but ViewData has only 1 Model property.** To solve this issue, you can work with **View Models**, which are **models that are specifically built for a view (containing data AND rules specific to that view**).

Remember: **view models do not go in the Models folder! Create a separate file called ViewModels** under the project! ViewModels are then classes that you add into this folder, with **the convention that ViewModel is it’s suffix.**

In the Controller, you can then pass the ViewModel to the View. **Remember to update the type of the model in the View itself**, in the @model statement at the top of the .cshtml file.

## Razor Syntax

In the View files (.cshtml), you **mix HTML markup with C# code, which you prefix with the @symbol**.

In foreach blocks, for example when iterating through a List coming from the model, you can also mix C# with HTML markup.

Example:

@if (Model.Customers.Count == 0)

{

<text>No one has rented this movie before.</text>

}

else

{

<ul>

@foreach (var customer in Model.Customers)

{

<li>@customer.Name</li>

}

</ul>

}

Within a @{ } block you can write any C# code you want, for example declare a variable that you use later on. This can for example be used to give elements an attribute. If the variable that you use for this evaluates to a *null* value, the attribute won’t be rendered into the element.

Comments go into statements like this @\* \*@

## Partial Views

**A partial view is like a small view that you can reuse on different views.** However, it can also be used, no for markup reuse, but for **breaking down large views into smaller, more manageable views.**

\_Layout.cshtml in \Views\Shared\ is the **bootstrap template** that defines the **overall** **look-and-feel** of the website.

**Within the Body element, there is a call to @RenderBody().** This is where all the other Views are rendered.

By convention, **all PartialViews are prefixed with an underscore sign.** Place them in the \Views\Shared\ folder and be sure to **check the box *Create as partial view***, so that it will not have a layout.

Afterwards, in the View where you actually want to use it, call it like so:

@Html.Partial("\_NavBar")

Important note: **the Partial method has an optional Model parameter**. **If this is not specified, the same Model that is passed to the calling View will also be passed to the partial view**. You can override this by passing in a specific view.

# Section 3 – Working with Data

## Entity Framework

It is a **tool to access a database**. Officially, it is an **object/relation mapper (ORM)**, which sits between your application objects and the relational data. In that sense, it is an abstraction layer over the relational data.

The gateway to the database is the class **DbContext**. This class has **1 or more DbSets,** which are tables in the database.

The DbSets are **queried with LINQ,** which EF translates into SQL queries. It then queries the database, reads the results, **maps them to objects**, and then adds them to DbSet.

**CRUD** operations are performed on the DbSet, which will keep track of changes, and then **persists them to the database** when instructed to do so.

## Code-first vs. Database-first

Database-first

Traditionally, people designed the database first and then have EF generate the domain classes.

Code-first

In this case, the domain classes are made first, after which EF creates the database tables. This approach is faster, because code is easily made. It is also easier to build an integration test database and keeps track of versioning of the database.

## Code-first migrations

**Any time you modify your domain model** (the collection of your application classes) by adding or modifying a class, **you create a migration** and then write it to the database.

Start by **enabling migrations** through the Nuget PM Console and type *enable-migrations*.

This will create a **new folder** under the project called **Migrations**. To create your first migration, in the NPM Console, type *add-migration [some name].* An example is *add-migration InitialModel.* This name should reflect the kind of changes you have made.

If you look at the migration that has been created, you will notice that any of the custom classes are not in there. This is because they have **not been included in the DbContext**.Once it has been **added as a DbSet property**, the DbContext will become aware of it.

If you want to **overwrite an existing migration**, postfix the *-force* switch.

After generating the migration class, to **persist the changes**, in the NPM Console, run *update-database.* When this successfully runs, if you select *Show All Files* in the Solution Explorer, you will see an .mdf file in the App\_Data folder. When you double click it, you can see the database with the tables. It also has a default table called **\_MigrationHistory which you should not touch and that tracks the executed migrations.**

Also, when you make changes, **don’t put them all in one big migration**. Instead, run it on the database to simplify fixes when something goes wrong. A bit like *git commit*.

## Changing the model

When creating a new entity that will be persisted to EF, **make sure to include a key**. By convention, this property must be named **id or [*yourtypename*]id** (to ensure **proper mapping to the table’s Primary Key**).

When you assign one type to the property type of another type, this is what’s called a **navigation property.** This works in effect as a FOREIGNKEY-PRIMARYKEY relationship, binding entities together. When you **don’t need the entire related object, but just its key**, add [yourtypename]Id behind the full reference. ASP.NET will recognize this convention. Example:

public class Customer

{

public int Id { get; set; }

public string Name { get; set; }

public bool IsSubscribedToNewsLetter { get; set; }

public MembershipType MembershipType { get; set; }

public byte MembershipTypeId { get; set; }

}

## Seeding the database

It is important when creating **reference data** that this is **never done on the database** in a code-first project! All changes must be done through migrations!

To add values into the table, first create an empty migration (no model changes) with an appropriate name. Example:

*add-migration PopulateMembershipTypes*

Open up the migration and, in the Up() method, you can call the Sql() method and pass in a SQL string:

*INSERT INTO Table (Column1, Column2) VALUES (Value1, Value2)*

Afterwards, just run ­*update-database*.

## Overriding EF Conventions

One of the conventions in EF is that string attributes will also be an NVARCHAR(MAX) NULLABLE**.** This might not be what you want.